Wonderware® I/O Server Toolkit

User's Guide

Revision L
October 2001

Wonderware Corporation

I/O Server Toolkit

100 Technology Drive
Irvine, CA 92618
U.S.A.
(949) 727-3200
http://www.wonderware.com

Trademarks

Contents

Documentation Conventions

The following conventions are used throughout this manual to define syntax:

Convention	Description
Bold Text	Denotes a Wonderware I/O Server Toolkit function name, for example, **Wizard_New**
Bold & Underlined	Denotes a Wonderware I/O Server Toolkit function name that must be written and included in the I/O server to manage logical devices, for example, **ProtAllocateLogicalDevice**
Italic text	Denotes a parameter value, for example, *wCommand*
CAPITALS	Indicates return type (or most return types) also filenames and paths
`Courier 9 Shaded Box`	`Code Examples and Syntax spacing samples`

Terms Used in this Document

Term	Definition
Developer	Proficient Windows C Programmer; person developing the code
Windows NT™	32-bit platform

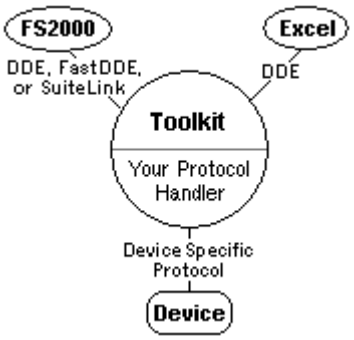
Limitation Summary

As the owner of the **I/O Server Toolkit** you can:

- Develop multiple servers
- Receive four hours of telephone support within one year from date of purchase
- Purchase additional support beyond four hours
- The **I/O Server Toolkit** license does not allow:
    1. License transfer without the express written consent of Wonderware
    2. Release/resale of **I/O Server Toolkit** source code or libraries
    3. Support for other than the registered user

CHAPTER 1

Introduction to the I/O Server Toolkit



The **I/O Server Toolkit** provides a high level application program interface (API) that does not require detailed handling or understanding of low level details of the client/server communication protocol (DDE or SuiteLink). The Toolkit has been optimized for performance in real-time data acquisition applications.

The **I/O Server Toolkit** is based on a library that includes high level functions that take care of the more difficult complications associated with the development of a well-behaved, high performance **I/O Server** (herein referred to as the server).

The **I/O Server Toolkit** was originally developed for internal use in developing other Wonderware products. It has been used to develop dozens of Wonderware I/O Servers with a worldwide installed base of thousands of sites. These servers are generally in 24-hour use in demanding factory automation and process control applications.

The Toolkit minimizes the learning curve associated with Dynamic Data Exchange (DDE) and SuiteLink implementation by utilizing the man-years of development and testing that have gone into Wonderware **InTouch**™ and I/O Server products. The Toolkit provides library functions that support multiple clients, thousands of data items, DDE and SuiteLink protocol error detection, recovery and support for several commonly used but unpublished high performance private data formats, such as Wonderware **InTouch** FastDDE and Microsoft Excel table format.

Contents

What's New?

With FactorySuite 2000, several important new features have been added to the Wonderware I/O Server Toolkit:

- **SMP (symmetrical multiprocessing machine support)**
  The I/O Server will run on a SMP (symmetrical multiprocessing). With the newly added API from the library (WWSetAffinityToFirstCPU), the I/O Server can be locked to the first CPU on the SMP machine.

Installation

Refer to the accompanying instruction pamphlet for installation procedures.

Note We strongly recommend before you begin installation procedures that your take the time to familiarize yourself with Chapter 2, "Getting Started with the I/O Server Toolkit." This chapter covers in detail how the Toolkit environment will be set up on your system.

Communication Protocols

Dynamic Data Exchange (DDE) is a communication protocol developed by Microsoft to allow applications in the Windows environment to send/receive data and instructions to/from each other. It implements a client-server relationship between two concurrently running applications. The server application provides the data and accepts requests from any other application interested in its data. Requesting applications are called clients. Some applications such as InTouch and Microsoft Excel can simultaneously be both a client and a server.

FastDDE provides a means of packing many proprietary Wonderware DDE messages into a single Microsoft DDE message. This packing improves efficiency and performance by reducing the total number of DDE transactions required between a client and a server. Although Wonderware's FastDDE has extended the usefulness of DDE for our industry, this extension is being pushed to its performance constraints in distributed environments.

NetDDE™ extends the standard Windows DDE functionality to include communication over local area networks and through serial ports. Network extensions are available to allow DDE links between applications running on different computers connected via networks or modems. For example, NetDDE supports DDE between applications running on IBM® compatible computers connected via LAN or modem and DDE-aware applications running on non-PC based platforms under operating environments such as VMS™ and UNIX®.

SuiteLink uses a TCP/IP based protocol and is designed specifically to meet industrial needs such as data integrity, high-throughput, and easier diagnostics. This protocol standard is only supported on Microsoft WindowsNT 4.0 and Windows 2000 or higher.

SuiteLink is not a replacement for DDE, FastDDE, or NetDDE. The protocol used between a client and a server depends on your network connections and configurations. SuiteLink was designed to be the industrial data network distribution standard and provides the following features:

Value Time Quality (VTQ) places a time stamp and quality indicator on all data values delivered to VTQ-aware clients.

Extensive diagnostics of the data throughput, server loading, computer resource consumption, and network transport are made accessible through the Microsoft Windows NT and Windows 2000 operating system Performance Monitor. This feature is critical for the scheme and maintenance of distributed industrial networks.

Consistent high data volumes can be maintained between applications regardless if the applications are on a single node or distributed over a large node count.

The network transport protocol is TCP/IP using Microsoft's standard WinSock interface.

What is DDE?

Dynamic Data Exchange (DDE) is a method of communication that allows concurrently running programs to exchange data with each other. It implements a client-server relationship between the applications. A server application accepts requests from any client application interested in the data. Clients can both read and write data maintained by the server.

DDE is often used to gather and distribute "live" data such as production measurements from a factory floor, scientific instrument readings or stock price quotations. Clients can use DDE for one-time data transfers or for ongoing exchanges in which updates will be sent as soon as new information is available. DDE's data writing mechanism can be used to issue data. For example, in a factory automation system, DDE can allow clients applications to control temperature set points in ovens.

A DDE interface has become a standard feature of Windows applications that can benefit from data links to other applications. Examples of DDE compliant applications include Microsoft Excel, Lotus 123 for Windows and Wonderware **InTouch**.

Network extensions are available to allow DDE links between applications running on different computers connected via networks or modems. For example, Wonderware **NetDDE**™ supports DDE between applications running on IBM PCs connected via LAN or modem as well as DDE capable applications running on non-PC based platforms such as VAX or UNIX minicomputers.

The Windows environment has a message based architecture as its foundation. Messages are used for passing keyboard and mouse movement information to Windows application programs. Each message needs only two parameters for passing data. DDE is based on the same message transport mechanism. As a result, these message parameters must refer indirectly to other pieces of data (objects) if more than a few words of information are passed between applications. These secondary objects are located in globally shared memory.

DDE Protocol

The DDE protocol specification is the precise interface that applications must implement to support DDE. The specification includes standardized formats for messages to be interchanged between DDE compliant applications. It is possible to ignore all or part of the DDE protocol if you are writing a set of applications that will communicate in a closed environment. However, in order to reliably communicate with other standard, off-the-shelf applications it is necessary to implement an interface that supports the DDE protocol documented by Microsoft.

The DDE protocol is nominally documented in the Microsoft Windows Software Development Kit (SDK). This documentation however is deceptively simple. It is particularly light in coverage related to performance optimization and error recovery. Sources such as back issues of the *Microsoft Systems Journal* and the *Microsoft Support Knowledge Database* CD ROM may provide the documentation required in order to implement a "well-behaved" DDE application.

"Ill-behaved" DDE applications can hang or crash the Windows environment. Early editions of certain very popular Windows applications suffered anomalies in their DDE implementation that are recognized and tolerated by "well-behaved" DDE applications.

To implement a reliable, high performance DDE protocol there are a multitude of potential error conditions that must be properly addressed. Still further, complications arise when optimal performance is desired and when partially compliant third party DDE applications must be tolerated.

Dynamic Data Exchange Management Library

Based on the recognition of DDE compliance problems associated with certain applications released by major software publishers, Microsoft released the Dynamic Data Exchange Management Library (DDEML) with the Microsoft Windows 3.1 SDK.

The DDEML is a dynamic link library (DLL) that applications running with the Microsoft Windows operating system can use to share data. The DDEML provides an application programming interface (API) that simplifies the task of adding DDE capability to a Windows application. The API hides interaction with Windows messages and provides new mechanisms for managing global shared memory objects.

The 3.1 SDK Programmer's Reference Manual states that "DDEML ensures compatibility among DDE applications by forcing them to implement the DDE protocol in a consistent manner." Concurrent with the release of DDEML, documentation of the DDE protocol was dropped from the SDK Programmer's Reference Manual.

In keeping with the Windows 3.1 philosophy, the principal purpose for promoting use of DDEML, as opposed to a direct implementation of the DDE protocol, would seem to be that it more strongly enforces parameter checking on system calls made by applications.

DDEML makes it more difficult to implement an ill-behaved DDE application — it does not make it easier to implement a well-behaved application. For example: calling GlobalAddAtom() (a function commonly used in a direct implementation of the DDE protocol) takes only one parameter; DdeCreateStringHandle(), its replacement under the DDEML, requires three parameters. The revised interface appears to be engineered to allow Windows to perform stronger type checking.

The implementation of a well-behaved, high performance DDE application using the DDEML will still require reference to the DDE protocol specification. The DDEML still uses the DDE protocol "under the covers." A thorough understanding of the underlying protocol is necessary to implement reliable error handling and recovery and to engineer an efficient internal program structure.

DDEML does not simplify the problem of multiple clients using potentially different data formats. For example, Excel uses Excel Table Format, and **InTouch** uses **InTouch** proprietary format. The Toolkit allows you to keep data in its natural form: discrete, integer, real or string. DDEML does not implicitly support these.

What is SuiteLink?

SuiteLink is a proprietary communication protocol created by Wonderware that can be used as an addition to DDE or as an alternative to DDE. Where DDE provides communication between Windows programs via global memory buffers and Windows messages, SuiteLink uses TCP/IP sessions and Windows Sockets. As with Wonderware's FastDDE format, SuiteLink supports high-performance data transfer between applications by sending multiple commands and data items in each block of information that gets sent.

Since it uses TCP/IP, SuiteLink can handle data transfer within a single computer node or between nodes across a network.

So far as an I/O Server is concerned, the type of communication channel between client and server is transparent. That is, *the server-specific code does not know – and does not need to know – whether a client is connected via DDE or via SuiteLink.*

SuiteLink Protocol

The SuiteLink protocol was created by Wonderware, specifically to support data acquisition for such programs as I/O Servers. While the details of the implementation are proprietary, an overview of the protocol is provided in the chapter titled "SuiteLink."

Server Application Requirements

The **I/O Server Development Toolkit** is used to either write an I/O Server application or add server capability to an existing Windows application. Here are a few requirements:

1. The application must run in a time slice manner. Periodically, the server will be called to allow it to run its protocol and provide fresh data to the Toolkit database. The time slice is configurable by the server, but it still must give up control to allow other Windows applications to run on the same PC.
2. The application program must be written in Microsoft C or C++, to interface with the Toolkit program library.
3. The Toolkit database will handle the following data types:

Discrete	(0 or 1)
Integers	(signed 32-bit integer)
Reals	(single precision floating point, IEEE 32-bit)
Strings	(series of 8-bit characters terminated with a null)

Within the above constraints, a large variety of applications can easily be adapted to transfer data as a I/O Server. Here are a few examples:

Gathering and sending data to a device that can be connected to the serial communication port of the PC.

Interface to a plug in board that has a memory-mapped interface.

A stand-alone application that can accept data and/or supply data. For example, a specialized calculation program that takes input data values and does involved computations or time-based integrations.

Toolkit Content Overview

The Toolkit allows the software developer to concentrate on the problem at hand, transferring data to and from an external communication device or special data generation application. It also allows the application program to deal with data in its native form: discrete, integer, real or string types. Source code (in the C language) for two example servers is included.

The examples include source code for the maintenance of extensible, heap allocated data structures for tracking data items that are on advise or "hot linked." The implementation of these data structures alone could take hundreds of hours.

The examples can also be expected to speed the learning process for programmers who lack prior Windows programming experience.

The **I/O Server Development Toolkit** consists of four essential parts:

1. This document describes the Toolkit's library routines and gives helpful hints for generating a I/O Server application from scratch or adapting an existing one.
2. An object-code library that supports the I/O Server functions and memory management. These are described later in this document.
3. Two server code examples are included to demonstrate the Toolkit. They provide a starting point for developers as well as showing useful data structures and routines for most device drivers.
4. Technical support for the I/O Server Toolkit provided by Wonderware's technical support staff.

Requirements for Developing on Windows 98, Windows NT and Windows 2000

The FactorySuite 2000 I/O Server Toolkit supports development of I/O Servers only on platforms that support Win32 – Windows NT, Windows 2000 or Windows 98 Second Edition. It does not support operation using the Win32S subset that is available for Windows 3.1.

Also, it should be noted that Windows NT /2000 and Windows 98 have different implementations of Win32. Consequently, there may be some functional differences when the same program is run on these operating systems. In particular, SuiteLink is functional only for an I/O Server running on Windows NT and Windows 2000.

The following requirements are necessary to successfully develop a FactorySuite 2000 Windows I/O Server application:

1. IBM PC or compatible that is set up to run Microsoft Windows 98 Second Edition, Windows NT 4.0, Windows 2000 or later.
2. Microsoft Visual C/C++ Compiler, version 6.0 sp3 or later.
   A mastery of the C language is essential for developing a Windows server application that uses the Toolkit.
   While a familiarity with Microsoft Windows software development is not required to use this Toolkit, it is highly recommended.

The following are recommended "extras" to your development system. Their relatively low cost is greatly offset by the time you'll save in debugging alone.

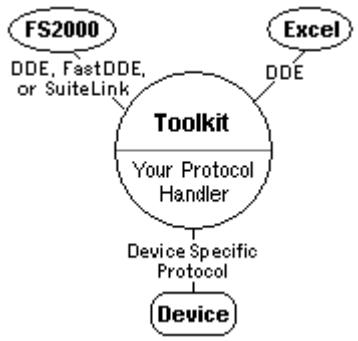
Microsoft MASM 5.0 or later. MASM is the Microsoft assembler necessary for modifying assembly language that Wonderware has provided, as well as for developing assembly language based routines.

Microsoft Word, an excellent documentation tool, can be used to generate the documentation files required for the "Help" compiler. It can be referenced in the subtitled section called "Adding Help to the I/O Server" in the I/O Server Toolkit Functions chapter.

The book, Programming Windows by Charles Petzold. (Microsoft Press, ISBN 1-55615-264-7)

CHAPTER 2

Getting Started with the I/O Server Toolkit



The Toolkit installation procedures are described in the installation pamphlet included with the **I/O Server Toolkit**. This chapter describes the basics for getting started quickly.

Contents

Installation Process

The instruction pamphlet describes the required procedure to install the Toolkit software. It is important to know that if an earlier version of the Toolkit is installed on your computer and the new version is installed the main directory should be renamed.

If You Have a Prior Installation of the I/O Server Toolkit

Rename the old directory to another name using Windows Explorer. This will prevent obsolete files from being accidentally referenced in your new installation.

General Instructions

Click Start/Run. Enter the following in the text field of the Run dialog box:

{x:\path}\setup

where:

{x:\path} is the drive:\path

This is where the **I/O Server Toolkit** distribution media is located.

InstallShield will direct the installation process and the installation process will create the following main directories:

drive:\path\ioserver
drive:\path\ioservertoolkit
drive:\path\uninst

where:

{drive:\path} is user defined

Example:

C:\Ww\Ioserver
C:\Ww\Ioservertoolkit
C:\Ww\Uninst

File Description

The following sections summarize some of the key files installed on your hard disk.

Include Files

\Ww\Ioservertoolkit\Inc\Tkitstrt.rci
This file is a resource include file containing the startup dialog WWStartup for the Toolkit. The project resource file (.RC) must include this file.

\Ww\Ioservertoolkit\Inc*.h
There are various include files contained in the includes directory that must be included in the source code to define function prototypes and structures for the Toolkit. See the sample servers for examples.

Utility Files

\Program Files\FactorySuite\Common\wwclient.exe
The WWClient utility replaces the DDEAPP utility that was originally developed by Microsoft. WWClient has been specifically written by Wonderware to exercise the basic DDE and SuiteLink client functions to test the server. It is a 32-bit application, which can run on Windows NT/2000 or Windows 98 Second Edition. For details, refer to the "Debugging and Testing" chapter.

\Program Files\FactorySuite\Common\TESTPROT.exe
The TestProt server has been specifically written by Wonderware to exercise the basic DDE and SuiteLink server functions. It is a 32-bit application which can run on Windows NT/2000 or Windows 98 Second Edition, and can be used to verify that WWClient can actually access an I/O Server on your system configuration. For details, refer to the “Debugging and Testing” chapter.

\Program Files\FactorySuite\Common\wwlogvwr.exe
The Wonderware logger, wwlogvwr.exe, will display and save to disk debugging messages from the server and the Toolkit. A version has been supplied that is native to each supported platform.

\Ww\Ioservertoolkit\Lib\W32\I386\Toolkit7.lib
The I/O Server Toolkit library.

Note: Recent versions (Build 060 or later) of the Toolkit library contain debug information resulting in a larger file size. To build a debug version of a server, use the added debug information. Building a release version of a server will remove the debug information.

\Ww\Ioservertoolkit\Inc\Protlib.str
This file contains the string resources used by **Toolkit7.lib**. These strings should only be modified for language conversion. Do not delete or change the order of any strings within the Toolkit. The project resource file (.RC) must include this file.

Ww\Ioserver\Udsample
This subdirectory contains source files for the board and serial sample servers.

Sample Servers

The toolkit provides a sample server, UDSAMPLE. The sample server is created using common source files and specific files for either the board or serial version.

\Ww\Ioserver\Udsample\Common
This main sample directory contains the common source files from which a complete server can be developed, once the specific files for a serial or board server are copied from one of the two corresponding subdirectories.

\Ww\Ioserver\Udsample\Udboard
This directory contains three files UDSAMPLE.C, UDSAMPLE.H, and UDSAMPLE.ICO which can be copied to the main sample directory to build a board server, which demonstrates the typical use of a memory mapped interface device in Windows.

\Ww\Ioserver\Udsample\Udserial
This directory contains three files UDSAMPLE.C, UDSAMPLE.H, and UDSAMPLE.ICO which can be copied to the main sample directory to build a serial server, which demonstrates the typical use of a serial communications (COM port) interface device in Windows.

Help Files

Two Help files have been provided with the **I/O Server Toolkit**.

\Ww\Ioservertoolkit\IOSrv_Toolkit.hlp
This is the Help file containing the API information for the Toolkit. This file will be a useful resource when developing a new server. Execute this Help file from Explorer. Creating a Help file icon in Program Manager may be helpful.

\Ww\Ioserver\Udsample\Udsample.hlp
This Help file is a template for the UDSAMPLE sample server. Copy it to the directory containing the UDSAMPLE.EXE file to run the sample.

Online Book

Also provided is an online book for the **I/O Server Toolkit**.

\WW\Ioservertoolkit\IOSrv_Toolkit.pdf
The document in this directory is in an Abode Acrobat .PDF file format.

Compiling a Server

The sample server includes project definition files for Visual C/C++ for Windows NT/2000 or Windows 98 Second Edition. environments. Refer to these examples to determine the proper options for compiling. The server sample is ready for compiling and linking. The Microsoft Visual C/C++ environment use for doing the builds. For specific compiling instructions using Microsoft Visual C/C++, Version 6.0 sp3 or later, refer to the instructions in the chapter on the I/O Server Code Examples.

The following (brief) compiler switches are recommended:

Windows 32: /GX /YX /MD /W4/ "WIN32"

Warning The definition of the preprocessor symbol "WIN32" is necessary. If you do not define this symbol , any conditional code based on "#ifdef WIN32" will not be compiled correctly.

The include files for the **I/O Server Toolkit** are in the \WW\IOSERVERTOOLKIT\INC subdirectory. Make sure that your compiler is able to find these includes by properly configuring your INCLUDE environment variable or dialog settings in Microsoft Visual C/C++.

The Microsoft compilers provide some necessary include files in the \SYS sub-directory. Make sure that your compiler is able to find these includes. For example, \MSVC\INCLUDE\SYS or \INCLUDE\SYS needs to be defined in your INCLUDE environment variable or settings.

Linking a Server

You will need to link your server against the proper TOOLKIT7.LIB static library. The FactorySuite 2000 Toolkit is located in the directory.

\WW\IOSERVERTOOLKIT\LIB\W32\I386\TOOLKITS7.LIB
Windows NT/2000 or Windows 98Intel I/O Server Toolkit Object Library

If doing command line builds, make sure your LIB environment variable references the proper directory above for your environment. If using Visual C/C++, reference the proper library in your project definition.

Running a Server

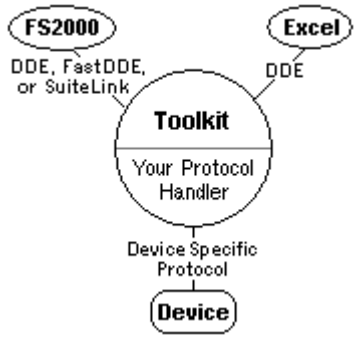
If your **I/O Server** uses the Wonderware Common Dialog DLL, you must make sure that the appropriate DLL can be found by the operating system. This can be achieved by placing the DLL in a directory that is referenced by your PATH environment variable or by placing the DLL in the directory with your server executable.

The DLLs provided with the toolkit are:

COMMONUI.DLL
DDECLIKT.DLL
WWCOMMON.DLL
WWDLG32A.DLL
WWDEBUG.DLL
WWCLINTF.DLL
WWPERF.DLL
WWPERFM.DLL
HOOKAGNT.DLL
HOOKNDDE.DLL
SLS_PERF.DLL
SUITELINK_PERF.DLL
WWSLSFIX.DLL
WWSL.DLL

Note: Supply the proper selection of these DLLs to your target system or customer along with the server executable. These DLLs are part of Factory Suite 2000 Common Components. To install the FactorySuite 2000 Common Component files:
Run SETUP.EXE in the \FS2kCOMM\IOServer\Common\Win32\ sub-directory on the installation CD.

CHAPTER 3



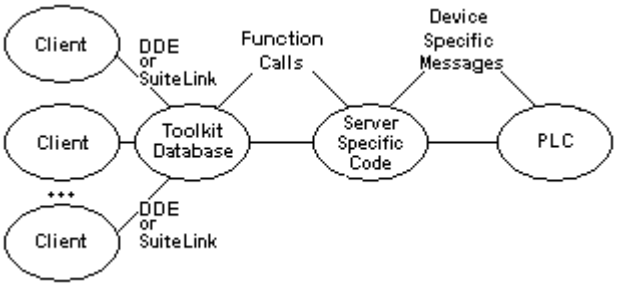
Overview of an I/O Server

DDE (Dynamic Data Exchange) is a protocol defined by Microsoft. SuiteLink is a proprietary protocol defined by Wonderware. Both these protocols allow independently developed Windows application programs to exchange data and commands.

Contents

Data Flow

The flow of data through an I/O Server can be diagrammed as follows:



All connections are bi-directional, i.e. data can go in either direction.

The **Toolkit** maintains a **database** of all the **topics** and **points** that are active in the I/O Server, and keeps track of the current data for each point within a topic. It also keeps track of **connections** to multiple **clients**, and handles fan-in and fan-out via the Toolkit database. The **server-specific code** (i.e. the code you write) does not need to know how many clients are connected, or whether they are communicating by DDE or SuiteLink. Instead, the server code can focus on communicating to the **PLC** by whatever device-specific protocol has been established by the manufacturer, and handling updates to and from the Toolkit database as if it were the only client.

Value/Time/Quality

The current information stored in the Toolkit database for each data point consists of three things:

- **Value**: The *value* of a data point represents the contents of some item within the PLC – a memory cell, a register, a bit flag, a string, etc.

  Examples: 5, 3.27, 1, "This is a string"

- **Time**: The *date/time stamp* (also sometimes called the *time mark*) for a data point represents the time at which the information about that data point was last updated.

  Examples: 03/27/1997 10:23:58.010

- **Quality**: The *quality* for a data point represents the validity or trustworthiness of the value for that point. If there are no problems, quality for the point is set to *good*. If there are errors or the data is out of range, the quality is set to *bad*. Specific quality flag settings are used to indicate particular problems with the data.

  Examples: Good, Clamped High, Communications Failed

DDE and SuiteLink Conversations

Two Windows applications wishing to exchange data must establish a *conversation*. This is accomplished by the *client* program opening a connection to the *server* application.

A client opens a connection by specifying two things: the *application name* of the server executable, and the *topic name* of interest. The application name depends upon the server program. For example, if you are using the Modicon MODBUS I/O Server, the application name would be "MODBUS". If using Microsoft Excel, it would be "Excel". For the Wonderware **InTouch** runtime program, **WindowViewer**, it would be "View".

Topic names are application-dependent. For Excel, the topic is a spreadsheet name, e.g., TOTAL.XLS, etc. For an I/O Server, the topics are names defined when the server is configured, with each name representing a *logical device* to which the server is connected, e.g., PLCFast, PLCSlow, etc. The topic in **WindowViewer** is always the word "TAGNAME" when accessing data elements in the **InTouch** runtime database. (The data is exchanged via item or point names.)

Depending on the configuration and the communication mechanism, a client may need to specify additional information to open a conversation:

- For a DDE conversation within a single computer (i.e. a single node configuration), the application name and topic name are sufficient.
- For a DDE conversation between two computers over a network (i.e. a multiple node configuration), the client must also specify the *node name* for the computer on which the server is running. For example, if the server is running on a remote computer named "FSTest", the node name "FSTest" must be specified. Also, you must have NetDDE running on both computers.
- For a SuiteLink conversation, the client must also specify the *node name* for the computer on which the server is running – on a remote computer (multiple node configuration). Also, you must have the SuiteLink service running (for the multiple node configuration, it must be running on both computers).

Note **FastDDE** is a proprietary protocol defined by Wonderware that uses DDE to transfer information between the client and server. This protocol uses DDE for the transport mechanism, but speeds up the transfer of data by using larger blocks of information in each transfer operation. Each block contains many operations for reading and writing individual values, for acknowledging commands, and for flagging the success or failure of those commands. **SuiteLink** also uses a block-oriented protocol, but uses TCP/IP and Windows Sockets as the transport mechanism.

Logical Devices and Points

Accessing information within an I/O Server requires an understanding of the **InTouch** concept of *logical devices* (also called *topics*) and *points*. The data that will be interfaced with the I/O Server must be treated as one or more logical devices connected to the server. *Points* within the logical devices can be read and written.

Generally, an I/O Server is connected to one or more I/O channels (i.e. serial ports, adapter cards, network connections, etc.), and each I/O channel is connected to one or more PLC devices. A *logical device* refers to a particular PLC – or even to a portion of a PLC – that is connected to the I/O Server. A user who is configuring an I/O Server will usually take into account such considerations as the following:

- Which I/O channel do I use to talk to the PLC? (e.g. COM1 or COM2, etc.)
- How do I address a particular PLC?
  In a networked or multi-drop system, it may be necessary to provide an address or recognition code that identifies a particular device.
- What variation of the protocol do I use to access a particular PLC?
  A single I/O Server may be connected to several different models from the same manufacturer, each with its own version of the manufacturer's protocol. If the server supports multiple protocols, it will be necessary to select which one to use.
- How often do I access the logical device?

  Some devices may be limited in how often they can be accessed (e.g. remote devices connected by a radio modem that requires warm-up and cool-down).

  The user may also wish to define several logical devices for a single PLC in order to poll groups of points at different intervals – e.g. read some points once per second and other points once per minute.

For example, let's consider the Modicon MODBUS I/O Server. A *logical device* for MODBUS would define the communications port and slave ID, while *points* within a logical device would be either coils or registers. The *points* will become data values that are transferred via DDE or SuiteLink to other applications. These may be Boolean (0 or 1), integer (signed 32-bit number), real (single precision floating point, 32-bit), or character strings (series of 8-bit characters terminated with a null, 0).

Logical Devices/Topics

For an I/O Server, there is only one application name, while the topic names have a one-to-one correspondence with logical devices. The notation convention for representing an application and topic is:

Application|Topic

Examples:

Excel|[Book1.XLS]Sheet1
View|Tagname
Modbus|ReactorPLCFast

A logical device (topic) becomes active whenever at least one conversation has been established between the server's logical device and the outside world's applications (*clients*). The Toolkit library routines call the server code when a client has initiated a conversation to a topic for the first time.

When the last conversation to a topic has terminated, the logical device will be deactivated. This allows the server code to do start up and shut down of a device or communication port, as required. The Toolkit simplifies DDE and SuiteLink conversation protocol by hiding conversations to multiple clients from the server. The server will only see a simple logical device activate or deactivate sequence no matter how many clients are requesting data.

Items/Points

Within a DDE or SuiteLink conversation (application|topic) the individual pieces of data that are passed between applications are known as *items* or *points*. For example, an *item* in a conversation with Excel would be the identification of the cell in a spreadsheet that contains the data value, e.g., R1C1 (row 1 column 1). For a conversation with **WindowViewer**, an item would be a tagname defined in the database, e.g., ReactorLvl. For a conversation with a Modicon MODBUS I/O Server, an item would be a register or coil number, e.g., 40001. Therefore, when developing the I/O Server, be sure to select the point naming convention that makes sense for your application. The DDE address convention for representing an application, topic and item is:

Application|Topic!Item

Examples:

Excel|[Book1.XLS]Sheet1!R1C1
View|Tagname!ReactorLvl
Modbus|ReactorPLCFast!40001

Points will be set active or inactive depending on usage by clients. Within the I/O Server task, a point is considered active if any DDE or SuiteLink conversations are referencing the item associated with the data point. If only an Excel spreadsheet is referencing an item in the server, that point is considered active. If the spreadsheet is closed, that point would become inactive. The same principle applies to **WindowViewer**.

Assume a point is not trended or alarmed, the point becomes active in the server when the point is displayed on the screen. When the point is no longer displayed on the screen, the point is inactive (assuming that this was the only conversation accessing the point).

The Toolkit library routines will call the server code whenever the active status of a point changes. This allows the server to adjust its polling (data gathering) sequence as required.

Advises, Requests and Pokes

A client can get continuous updates for a point by doing an **Advise** operation. If a point is not already "on advise," the Toolkit calls the functions **ProtCreatePoint**( ) and **ProtActivatePoint**( ) to activate the point. These functions, which must be implemented by the server programmer, perform whatever operations are necessary to establish periodic polling of the PLC to obtain the current value for the data point. When new data is obtained, the server-specific code must interpret the message from the PLC, extract the new value for the point, and pass the new point information to the Toolkit via a call to **ProtNewVTQFromDevice**( ). The Toolkit will then pass the data on to the client(s). Note that such updates to the client(s) are done by exception – that is, new information for a point is sent to the client(s) only if it is different from the previous information. (See the section below regarding the Toolkit database.)

A client can also do a **Request** operation to get the current value of a point, even if the client does not have that point "on advise." This invokes a one-time data transfer of the point information, instead of providing a continuous update. Note, however, that the information transferred comes from the Toolkit's database. A Request does not generate on-demand polling of the PLC. If the Toolkit's database does not have a current value for the point, it will wait until normal polling provides a value – up to the time limit specified by the parameter **ValidDataTimeout**. If the point is not currently being polled, the Toolkit will put it on "temporary advise" for this client to initiate polling of the PLC for a value.

A client can change the value of a point by doing a **Poke** operation, which is handled by a DDE Poke or SuiteLink Write. The Toolkit passes the new value onto the server-specific code via the function **ProtNewValueForDevice**( ). This function, which must be implemented by the server programmer, takes the new value and performs whatever operations are necessary to send that value to the PLC. When a client pokes a new value, that value is "reflected" to the other clients, i.e. all other clients with that point "on advise" or "on request" are notified of the new value directly from the Toolkit – i.e. before it is even sent out to the PLC. Poke reflection is built into how the Toolkit handles DDE Pokes; however, it is optional for SuiteLink Writes. See the chapter on SuiteLink and SuiteLink Debug Flags for more information.

Toolkit Database

The I/O Server Toolkit maintains a database of the logical devices (topics) and points that are being accessed in an I/O Server. It also keeps track of client connections to each Topic!Point on a per-client basis. This means that the Toolkit knows the current information for each point and it knows which clients have been provided with that information.

Note Multiple DDE and/or SuiteLink conversations accessing the same points (topic!items) can be established to the I/O Server at the same time. For example, **WindowViewer** and an Excel spreadsheet and a dozen other applications may be referencing the same logical device and item/point at the same time. The Toolkit library software will automatically handle these situations for the server. The server code need only be concerned with whether or not the point is active.

When a point is active, the server code should be supplying fresh data from the PLC to the Toolkit. The Toolkit will store the value/time/quality information in its database and send updates to all concerned clients. The server code only needs to gather the active data and pass it on. The Toolkit will handle the rest.

Note, however, that new updates for points "on advise" are passed on to the clients by exception, i.e. only when the point ***value*** or ***quality*** changes. No update is passed on if only the ***time*** changes. This reporting mechanism was chosen to reduce bandwidth in the DDE and SuiteLink channels for points that are polled frequently but do not change much. The exception mechanism has been in place for all previous versions of the Wonderware I/O Server Toolkit.

The Toolkit makes calls to **ProtAllocateLogicalDevice**( ) only when the first client connects to that logical device. If any clients remain connected to the logical device, the Toolkit keeps the logical device open. Only when the last client disconnects from a particular logical device does the Toolkit call **ProtFreeLogicalDevice**( ).

The same is true for the point functions. The Toolkit calls **ProtCreatePoint**( ) only when the first client connects to a point. And only when the last client disconnects from a point it calls **ProtDeletePoint**( ). The Toolkit will call **ProtActivatePoint**( ) and **ProtDeactivatePoint**( ) multiple times as clients (like an InTouch app with switching windows) activate and deactivate items. Some deactivation calls will be 'folded' due to efficient logic built into the Toolkit.

Note Recommended IOServer Implementation:

Only react to the first activation of an item. Ignore all subsequent activation for the item.

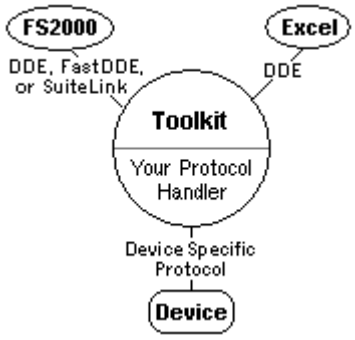
When the server recieves a deactivation call, only react to the first deactivation for the item. Since the item is now deactivated subsequent deactivation calls can be ignored.

The Toolkit database uses the case-insensitive string comparison function **stricmp**( ) to determine whether the name of a logical device or point is the same as one it already has in its database. When implementing a server, take special care to ensure that if the Toolkit treats two names as the same or different, your server-specific code treats them the same way. For example, if the point names "V10R" and "V10." both refer to a real value at PLC address V10, keep in mind that the Toolkit will treat these as two different points, even if they are functionally the same. Your implementation of functions such as **LogicalAddrCmp**( ) [logical address compare for points] should take this into account.

The Toolkit database is organized as a **hierarchy** of points within a logical device. If a point is referenced on two different topics, that means it has two different entries in the database – one for each logical device. This is important, because if the topics are being polled at different rates, the information for the two entries may differ.

For example, suppose an integer point is incrementing once per second. If one topic is polled at 500-millisecond intervals and the other is polled at 5-second intervals, their values may not match except at the 5-second marks. A client doing a Request at 2.5 second intervals would see the difference. Of course, this depends upon how the server-specific code is implemented – it would be possible to update both points at the 500-millisecond interval. However, if a user has set up two different scan rates for the same point, it might be for a specific reason. When developing a server, take care not to put so much "finesse" into the code that it yields results that are counter-intuitive.

CHAPTER 4



Designing an I/O Server

Designing an I/O Server with the **I/O Server Toolkit** can logically be divided into three areas.

For a "working" model of an I/O Server, see the sample server that is supplied on the Toolkit CD. From the sample code, you can construct either a board server or a serial server.

Contents

Configuring an I/O Server

An I/O Server can only respond to requests for data from a DDE or SuiteLink *client*. Those requests for data are handled by the Server Toolkit portion of the I/O Server. The *topic* (or logical device) specifies where the data is located. However, the server needs more information than just the logical device name to supply the data. The server may need to know the device identification, device model, or network configuration. Additionally, other parameters may be needed in order to acquire the data, such as the communication port being used, the baud rate, parity, etc. If the I/O Server is getting data from a board device, it may need to know the memory address of the board. When configuring the I/O Server, specify how frequently the device is polled for data and how long to wait for a timeout to occur.

The configuration is performed by the user through the use of menu items in the server's program window. Typically, there are menu items for communications parameters or board definition, topic parameters, and timing parameters for the server. The entire definition of a logical device (*topic*) may include many hardware and timing parameters. For example, the Modicon MODBUS I/O Server Topic Definition includes the following information:

COM Port	(COM1, COM2, etc.)
Slave ID	(Modicon slave number)
Poll Frequency	(Frequency each point is polled)
Coil Read Size	(Number of coils read in each message)
Register Read Size	(Number of registers read in each message)

We recommend that you provide the user with the ability to name and fully define each topic supported by the server. For example, with the Modicon MODBUS I/O Server, the user could, arbitrarily, create a file such as the following:

COM1	9600,8,1,n
PLCFast	COM1,1,1000,512,64
PLCSlow	COM1,1,5000,512,64

The server could read this file at start up, allowing the user to use meaningful names to access each logical device. This configuration file could just as easily be created by the server using disk writes and selection boxes to interface to the user. The method used is entirely up to the user. There is no requirement for the server to support multiple topics (logical devices) or a configuration file. If the server is being developed to interface to a simple device containing relatively few points, only one topic needs to be defined.

The two sample programs, UDSERIAL and UDBOARD are good examples of how to implement a configure menu option.

In order to interface to the Toolkit, the topic naming and item naming conventions must be defined. The *topic name* is generally a text string that contains a meaningful name. It is highly recommended that the item names match the convention used in the logical device and provide enough information to determine the data type, address and possibly precision of the point. The points that are to be supported should be decided during the design stage.

Executing the Protocol

The protocol is executed when the I/O Server receives a timer event from the Toolkit. The exact data items that are needed have been set up by previous calls from the Toolkit. As the designer of the server, you must design the item names such that each is a unique address within the device from which you are requesting data.

Each topic may have a set of active item names. This set of items results in a message being created and sent to the device. This message is sent and data received when the server gets a timer event from the Toolkit. Since communications protocols are as varied as the devices themselves, there is not "one way" to accomplish this. The UDSERIAL example is a skeleton for a serial communication driver. The UDBOARD example is a skeleton for a board based I/O Server.

Communication with the Device

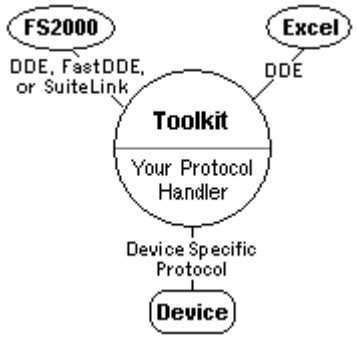
The Toolkit supplies two samples of I/O Servers, based on two different models of communications:

1. The SERIAL sample is based on a device that is accessed through the communications port of the PC.
2. The BOARD sample is based on a device that is accessed through a special board in the PC (Windows Only). A device driver would also be necessary for Windows NT and Windows 2000.

These samples are actually template programs for working I/O Servers. They are the starting point for all Wonderware servers and have most of the required infrastructure built in to develop a working server. The portions of code that must be written by the developer are noted by comments within the source.

After determining which type of server to be developed, refer to that specific server in the sample servers section. Most developers choose to modify the few necessary routines in the sample programs rather than starting from scratch.

CHAPTER 5



SuiteLink

This section describes how the I/O Server Toolkit uses the Wonderware SuiteLink communication protocol to transfer data between the I/O Server and clients.

Contents

SuiteLink Overview

With the advent of FactorySuite 2000, the Wonderware I/O Server Toolkit now provides SuiteLink, a proprietary communication protocol that can be used as in addition to DDE or as an alternative to DDE. Generally speaking, the SuiteLink protocol works much like the Wonderware FastDDE protocol:

- The sender allocates a block of memory and fills it with a sequence of items detailing events and data. Events/commands include the following:
  - "Registering" a point (assigning a handle or ID number)
  - "Advising" a point for continuous update
  - "Requesting" the current value of a point
  - Writing a value to a point
  - Providing new polled data for a point
  - "Unadvising" a point
  - "Unregistering" a point
  - Acknowledging a command, indicating success or failure
- The SuiteLink transport moves the contents of this block from the sender to the receiver.
- The receiver then interprets the block of events, performs the indicated operations, and creates a response block of ACKs, Nacks, data, etc.
- The SuiteLink transport moves the contents of the response block to the original sender.

The main difference is that SuiteLink uses TCP/IP sessions as a transport instead of Dynamic Data Exchange (DDE). Where an I/O Server is concerned, the type of communication channel between client and server is transparent. That is, *the server-specific code does not know – and does not need to know to know – whether a client is connected via DDE or via SuiteLink.*

The following are some of the features SuiteLink offers over DDE:

- Performance optimizations for networked installations with large fan-in and fan-out network node count requirements.
- Reduced interaction between connections when an outage or delay occurs on one of the connections.
- Better diagnostic and monitoring capability.
- Transport of value, time, quality (VTQ) information.
- Use of standard TCP/IP session based transport at fixed port address (5413) to enable deployment over WAN intranet, RAS dial-up, or Internet if desired.

The following factors may make the continued deployment of DDE/NetDDE preferable in some installations:

- When client and server are on the same PC, DDE may offer reduced data reporting latency under conditions of low data item counts combined with high (> once per second) data change rates.
- When client and server are on different PCs, but on the same subnet, and NetBUI is configured as the NetDDE transport, NetDDE may offer reduced data reporting latency under conditions of low data item counts combined with high (> once per second) data change rates.

A server built with the I/O Server Toolkit will automatically support *both* SuiteLink and DDE/NetDDE connections. Both varieties of connections can be used simultaneously.

Components (Files) Associated with SuiteLink

A server supporting the SuiteLink protocol relies upon the following components:

- WWSL.DLL, the SuiteLink API library
- WWSLS.EXE, the underlying SuiteLink transport library
- WWPERF.DLL and WWPERFM.DLL, extensions to the Windows NT/ 2000 Performance Monitor that to allow it to track SuiteLink operation. (in addition, the performance monitor will automatically create WWSL_PERF.DLL and WWSLS_PERF.DLL DLLs the first time that a SuiteLink enabled server is run.)
- SLSSVC.EXE, the SuiteLink service

In addition, the WinSock TCP/IP DLL that ships with Windows NT/ 2000 is required.

The SuiteLink NT service (SLSSVC.EXE) must be installed and started on the server PC to allow SuiteLink client applications to connect to the server, even if the client and server are on the same PC.

Starting Up a Server

When the I/O Server starts, the Toolkit attempts to load the SuiteLink API library (WWSL.DLL on Windows NT / 2000), unless SuiteLink is disabled via a Registry entry, see the "SuiteLink Debug Flags" section below. If the DLL can be successfully loaded, the Toolkit attempts to initialize the SuiteLink API. If both these steps are successful, the server will be ready to communicate with clients via SuiteLink. Otherwise, the Toolkit sends messages to **Wonderware Logger** indicating that the DLL could not be loaded, or that it could not be initialized.

Even if SuiteLink cannot be started, an I/O Server can still run with communication to clients only by way of DDE. If you wish to prevent a server from running if SuiteLink is unavailable, this can be set up with a Registry entry. See the section "SuiteLink Debug Flags" below.

Automatic Throttling of the Data Rate

SuiteLink is capable of transporting data at high throughput rates – rates high enough that some programs (particularly those that perform a lot of graphics) may not be able to keep up. For SuiteLink, the Toolkit automatically adjusts the data rate by tracking the size and number of communication blocks are waiting to be accepted by the receiver. If the amount of waiting data exceeds an upper threshold (the "high water mark"), data transport to the client is "held off," and no more polling updates are sent until the waiting data falls below a lower threshold (the "low water mark").

This works a little differently from the DDE protocol – where new data is not sent until the previous data has been acknowledged – but it has the same overall effect of keeping the throughput at a level that the client can accommodate. DDE incurs round trip packet delays even under condition where throttling is not required – SuiteLink does not.

Since each client application may have a different capacity at which it can digest data updates, the adjustment of the data throughput is done on a client-by-client basis. One client may throttle back its data rate without reducing the update rate provided to other clients.

It should be noted that while the Toolkit is waiting for a client to reach its "low water mark," the polling process can still continue (and should still continue) making updates to the Toolkit database. While communication to a client is being "held off," if a change occurs to a point, the Toolkit sets an internal flag for that point/client connection indicating that an update needs to be sent to the client. When the "low water mark" is reached, the most current value in the database will be sent to the client, for each point that has a change flagged. Thus, if the point changes several times while the client is being "held off," only the most recent value will be sent. This matches the functionality of "data folding" in DDE.

SuiteLink Debug Flags

SuiteLink has several Registry flags that can be set for operational or debug purposes. Each such flag applies only to the specific I/O Server for which it is entered, under the Registry key

HKEY_LOCAL_MACHINE\SOFTWARE\Wonderware\<server_name>

The list of flags is as follows:

EnableSuiteLink (default = 0x1)

A value of 0 disables SuiteLink for the server.

RunWithoutSuiteLinkDLL (default = 0x1)

A value of 0 prevents the server from running if the SuiteLink DLL cannot be loaded.

SuiteLinkReflectPokes (default = 0x1)

A value of 0 prevents the server from "reflecting" pokes received via SuiteLink to other clients. See the section "Preventing a Server from Reflecting SuiteLink Pokes" later in this chapter. [Pokes received from clients via DDE will still be reflected, however.]

SuiteLinkFlushReadOnDemand (default = 0x1)

Normally, a buffer full of read data is "flushed" to the transport process when it is full or when the SuiteLink protocol demands an immediate flush. A value of 0 causes the flush of a partial buffer to occur only on a Protocol Timer Tick event.

SuiteLinkFlushWriteOnDemand (default = 0x1)

Normally, a buffer full of write responses is "flushed" to the transport process when it is full or when the SuiteLink protocol demands an immediate flush. A value of 0 causes the flush of a partial buffer to occur only on a Protocol Timer Tick event.

DebugSuiteLinkEvents (default = 0x0)

A value of 1 causes the Toolkit to issue messages to **Wonderware Logger** for tracing the processing of SuiteLink events.

DebugSuiteLinkCalls (default = 0x0)

A value of 1 causes the Toolkit to issue messages to **Wonderware Logger** for tracing SuiteLink API calls.

More details on some of these flags are provided on the following pages.

Deactivating SuiteLink for a Particular Server

SuiteLink can be deactivated for a particular I/O Server via an entry in the Registry. For example, to deactivate SuiteLink for the server TESTPROT, use the Registry Editor to access or create the following Registry key:

HKEY_LOCAL_MACHINE
SOFTWARE
Wonderware
TESTPROT_IOServer

and for the key TESTPROT_IOServer, edit or create the value
EnableSuiteLink: REG_DWORD: 0

A value of 0 disables SuiteLink, while a value of 1 enables SuiteLink. If SuiteLink is disabled, the server will not try to load the SuiteLink DLL.

Preventing a Server from Running if SuiteLink is Unavailable

Normally, an I/O Server will still be able to run if SuiteLink cannot be started up. In that circumstance, it will only be able to communicate with clients via DDE (both regular DDE and FastDDE). However, in some environments you may wish to prevent the server from starting if the SuiteLink DLL cannot be loaded. This can be done via an entry in the Registry.

For example, to prevent the server TESTPROT from running without WWSL.DLL, use the Registry Editor to access or create the following Registry key:

HKEY_LOCAL_MACHINE
SOFTWARE
Wonderware
TESTPROT_IOServer

and for the key TESTPROT_IOServer, edit or create the value
RunWithoutSuiteLinkDLL: REG_DWORD: 0

A value of 0 requires the SuiteLink DLL to be available, while a value of 1 enables the server to run without SuiteLink. If the SuiteLink DLL cannot be loaded and RunWithoutSuiteLinkDLL is disabled, the Toolkit will display a Message Box indicating the problem and will not start the server.

Preventing a Server from Reflecting SuiteLink Pokes

Normally, when a client "pokes" data for a point to an I/O Server (via a DDE POKE or a SuiteLink Write), the Toolkit updates its database, sends the new value out to all clients connected to that point (via a REQUEST or an ADVISE), and then sends the value to the server-specific code via ProtNewDataForDevice( ) for transfer to the PLC. This process of sending the new data to all connected clients is called "reflecting" the poked data, and is an inherent operation of how the Toolkit handles the DDE protocol. However, with SuiteLink, you can suppress the "reflecting" of poked data for a particular server, via an entry in the Registry.

For example, to prevent the server TESTPROT from reflecting SuiteLink pokes, use the Registry Editor to access or create the following Registry key:

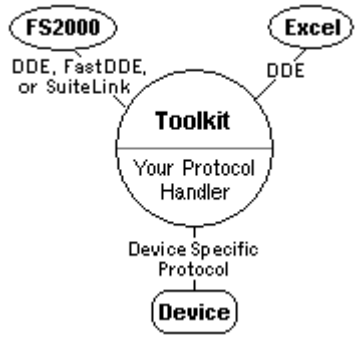
HKEY_LOCAL_MACHINE
    SOFTWARE
        Wonderware
            TESTPROT_IOServer

and for the key TESTPROT_IOServer, edit or create the value
    SuiteLinkReflectPokes: REG_DWORD: 0

A value of 0 prevents the Toolkit from reflecting pokes received from clients via SuiteLink. Even clients that are connected via DDE will not receive updates, if the point is poked via SuiteLink. (However, pokes received via DDE will still be reflected, to all clients whether they are connected with DDE or with SuiteLink.) This means that clients will only see the new data when it is obtained from the PLC via the normal polling process. This helps confirm that the data has actually been written to the PLC – but only if the point is being polled in the first place.

CHAPTER 6

Time Marks



This section describes the handling of time marks used by the Wonderware I/O Server Toolkit.

Contents

Reading Time Marks

The I/O Server Toolkit keeps value, time, and quality for each point in the Toolkit's internal database, organized by the Topic!Point hierarchy. Whenever a data point is updated (whether for value, quality, or both), the **time** of the update is stored in the Toolkit database on a topic-by-topic, point-by-point basis.

To support the transfer of Value/Time/Quality (VTQ) information, the Toolkit API provides several function calls, in particular the following:

DbNewVTQFromDevice( ) updates value, time, and quality
DbNewTQFromDevice( ) updates time and quality only (value unchanged)

In addition, there are calls in which the time mark is not passed explicitly, but a current default time mark is used:

DbNewValueFromDevice( ) updates value (quality assumed good)
DbNewQFromDevice( ) updates quality only (value unchanged)

Most PLCs do not have a date/time clock that is accessible to the I/O Server. For those which do, you have the option of using that date/time clock as your time mark. In this case, you should convert the time from the PLC's format to a FILETIME value (see below). Otherwise, you should use a Toolkit API call to read the computer's time.

Reading the time is accomplished by a call to DbGetGMTasFiletime( ), which internally makes calls to the Win32 API routines GetSystemTime( ) and SystemTimeToFileTime( ). The FILETIME structure is a 64-bit value that encodes the number of 100 nanosecond intervals since January 1, 1601. (Which means the time mark is good for about 30,000 years -- no Year 2000 problem, here!) However, these are rather expensive calls to make, as regards performance, so getting the time mark should be done only as much as is actually needed.

The Toolkit provides a default time mark only if one is needed. The way it does this is as follows. Just before it calls the server-specific function ProtTimerEvent( ), the Toolkit clears an internal flag, indicating that no default time mark is available. Then later, if some server-specific code calls a function [such as DbNewValueFromDevice( ) or DbNewQFromDevice( )] that needs a default time mark, the flag is checked and, if it is FALSE, the current time is read and saved as the default time mark for use on later calls during the same timer event, and the flag is set TRUE. This way, if a single loop makes 1000 calls to DbNewValueFromDevice( ), we don't wind up making 1000 calls to get the default current time (which won't differ by much, under such circumstances, but the calls will cause a performance hit).

If you are explicitly obtaining the time via a call to DbGetGMTasFiletime( ), you should take into account what these calls will do to server performance. If you're updating 10,000 points per second, their time marks aren't going to be very different from one another; so you probably don't need 10,000 calls to DbGetGMTasFiletime( ). You should limit calls to this function in a way that maximizes performance while maintaining a reasonable update rate for the time mark. Reasonable places to get the time mark include the following:

- Once per entry to ProtTimerEvent( ), and then only if you need it
- Once per response message (which would normally encode data for many points)

Of course, depending on the device being served and the application to which it is put, it may make sense to get time marks more or less often than once per timer event or once per message. This is largely a judgment call, balancing performance against the requirements of the server, or of the application.

The following excerpt from the routine UdprotExtractDbItem( ) in UDPROTCL.C of the sample server code illustrates one way of getting the time mark only as often as needed:

```
    /* scan through range of symbols covered by this message */
    done = FALSE;
    while ((!done) && (lpSymEnt != (SYMPTR) NULL)) {
        /* check whether symbol is active
           and polled by this message */
        if (compValue.SymHandle != 0) {
            /* match found,
               get parameters from symbol table entry */
            myAddr   = lpSymEnt->msAddr1;
            count    = lpSymEnt->msCount;
            numBytes = (WORD) lpSymEnt->msNumBytes;
```

```
            /* check whether already have date/time stamp */
            if (!bHaveDateTimeStamp) {
                /* set up date/time stamp */
#ifdef WIN32
                DbGetGMTasFiletime( &ptTime );
#else
                ptTime.dwLowDateTime = 0;
                ptTime.dwHighDateTime = 0;
#endif
                /* indicate date/time stamp ready */
                bHaveDateTimeStamp = TRUE;
            }
```

Understanding Time Marks

It is important to understand what time marks mean, i.e. what time they represent and when the information is transferred from server to client or vice versa.

When a client sends data to an I/O Server (via a DDE Poke or a SuiteLink Write), the Toolkit reads the current time mark and stores it with the new value in the Toolkit database. (Note: the quality received from the client is assumed to be good.) The new value, time, and quality are then "reflected" to all other clients that are connected to that point (via an ADVISE or a REQUEST for data). The Toolkit then calls the function ProtNewValueForDevice( ) and passes the value to the server-specific code for transfer to the PLC.

When the I/O Server obtains a new value from the PLC, it passes the value, time, and quality to the Toolkit via a call to a function such as DbNewVTQFromDevice( ). The Toolkit stores the new value, time, and quality. It then compares the new **value** to the old **value** that was in the database and the new **quality** to the old **quality** that was in the database. If the values and/or qualities are different, the new VTQ is reported to all SuiteLink clients. If the values are different, the new value is reported to all DDE clients (i.e. DDE clients don't care about quality). No comparison is made regarding the **time** marks. However, since the new time mark does get stored in the Toolkit database, any client connected to that point (via an ADVISE or a REQUEST) will get the most recent time mark with the VTQ if it is sent.

In short, a given client will get updated only when the value or quality changes (i.e. **reporting by exception**). Thus, if you have a point in the PLC that is unchanging, but is being polled repeatedly, a client with that point on ADVISE will probably not have the most current time stamp – i.e. it will have the time stamp corresponding to the last update the client received. However, the time stamp in the Toolkit's database does get updated every time the server passes a new value or quality to the Toolkit.

By way of illustration, suppose you have a PLC that is counting an integer, 1..2..3..4.., with the count increasing once per second. And suppose you set the server so that it polls the PLC at a sample rate of once per 400 milliseconds. Then the polled readings will be

Time (msec)	Value
0	1
400	1
800	1
1200	2
1600	2
2000	3
2400	3
2800	3
3200	4
3600	4

A client with this point on ADVISE will see the updates when the value actually changes

0	1
1200	2
2000	3
3200	4

But if a client were to REQUEST the data at, say, 1700 msec into the process, he would get

1600 2

representing the timestamp of the most recent polled reading.

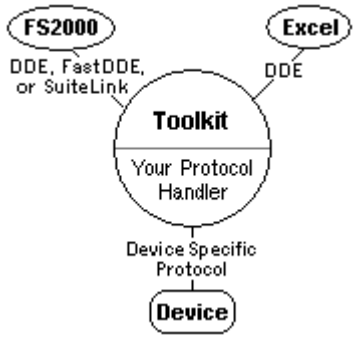
This last example brings up another important issue: how the Toolkit fulfills a data REQUEST. This is actually a multi-stage process that may or may not generate a new polling message or time stamp for the point:

1. When a client issues a REQUEST for a data point, the Toolkit looks to see whether the point is already being polled (i.e. because it is on ADVISE for another client).
2. If the point is being polled, the Toolkit checks whether it already has data for that point in its database.
    a. If so, the Toolkit sends the value, time, and quality from the database.
    b. If not, the Toolkit waits for normal polling to update the point, and then sends the value, time, and quality to the client.
3. If the point is not being polled, the Toolkit calls ProtActivatePoint( ) to set up polling for the point, waits for the point to be updated, and then sends the value, time, and quality to the client. It then calls ProtDeactivatePoint( ) to stop polling.

Note that steps 2b and 3 amount to putting the point on "temporary advise" for the client. The Toolkit waits up to the time limit indicated by the WIN.INI setting ValidDataTimeout. If no data has arrived by then, the Toolkit sends a NAK, indicating that the REQUEST could not be fulfilled. But it should be noted that if the Toolkit database already contains a value for the indicated point, the reported time stamp will represent the last time the point was updated, not the time the client issued the REQUEST.

Finally, it is also important to understand that the Toolkit database is organized as a Topic!Point hierarchy – that is, points within a topic. This means that if a point is being accessed via two different topics, there will be a separate entry for each point. That means two time marks! It is possible for a server to cross-reference points on different topics, and make sure that if a point is updated on one topic it is updated on all topics. However, most servers do not do this – and it may not even be desirable. Even when multiple topics actually refer to points in the same memory space of the same PLC, the different topics may have been created expressly to provide different "scan" rates for particular points.

CHAPTER 7



Data Quality Flags

This section describes the data quality flags used by the Wonderware I/O Server Toolkit.

Contents

Quality Flags

Wonderware I/O Servers can report six (6) mutually exclusive states of quality of data being sent back to their clients. They are as follows:

1. Good
2. Clamped High
3. Clamped Low
4. Cannot Convert
5. Cannot Access Point
6. Communications Failed

The conditions under which each of these quality states will be reported are as follows:

1. Good

a. The Communications link has been verified.
b. The PLC understood our Poll request and returned a valid response packet.
c. If a write occurred, there were no errors during the write process.
d. There were no conversion problems with the data contained in the response packet.

EXAMPLE: The value 0x0000A is returned due to a poll of a register containing 10 (decimal).

2. Clamped High

a. The Communications link has been verified.
b. The PLC understood our Poll request and returned a valid response packet.
c. The register was read or written without error.
d. It was necessary to clamp its intended value to a limit because the value was larger than the maximum allowed.
e. In the case of a string, it is truncated.

EXAMPLE: A floating-point value is clamped to FLT_MAX.

3. Clamped Low

a. The Communications link has been verified.
b. The PLC understood our Poll request and returned a valid response packet.
c. The register was read or written without error.
d. It was necessary to clamp its intended value to a limit because the value was smaller than the minimum allowed.

EXAMPLE: A floating-point value is clamped to FLT_MIN.

4. Cannot Convert

a. The Communications link has been verified.
b. The PLC understood our Poll request and returned a valid response packet.
c. The data from the PLC could not be converted into the desired format.
d. The server may return a constant (e.g. zero) in place of the data, or return quality information alone.
e. The data is not usable.
f. It is not known whether the value is too large or too small.
g. The data returned from the PLC is of the incorrect data type.
h. A Floating Point number is returned, but is not a value (i.e. Not A Number).

EXAMPLE: The value of 0x000A is returned from a BCD register in a PLC.

5. Cannot Access Point

a. The Communications link has been verified.
b. The PLC understood our Poll request and returned a valid response packet.
c. The PLC reported that it could not access the requested point.
d. Possibilities for lack of accessibility include, but are not limited to:
    1. Item does not exist in PLC memory.
    2. Item is not currently available (locked in some way due to resource contention).
    3. Item is not of the correct format / data type.
    4. A write attempt was made, but item is read-only.

        a. In most cases, a group of items will be affected when one item is invalid. This is due to the block-polling scheme used by the servers. For example, if one item in a block of 10 is invalid, then entire block is marked invalid by the PLC. The server will report invalid quality for all items in the block.
        b. The data is unusable.

EXAMPLE: Attempting to read R40001; but R40001 is not defined in the PLC's memory map.

6. Communications Failed

a. Data communications are down.
b. The Topic is in slow poll mode (or equivalent).
c. There have been no link validating messages.
d. Lack of resources in the server, e.g. a TSR (or driver) cannot allocate memory.
e. Lack of resources in the communications link.
f. The communications link is off-line.
g. All communications channels are in use.
h. The network is unable to route the message to the PLC.

EXAMPLE: Attempting to read data from a PLC which has been powered off.

Quality Flag Settings

The data quality settings and their relation to the quality flags used by OLE for Process Control (OPC) are defined in the file WWSQUAL.H, as follows:

```
// Wonderware Server Data Quality Flags
//
//                        Hex          OPC              OPC
//                       Value     Quality Bits     Quality Bytes
//                                                  Quality
//                                 MSByte   LSByte   |   SubStatus
//                                XXXXXXXX QQSSSSLL  |   |   Limits
#define WW_SQ_GOOD      0x00C0  00000000 11000000  Q=3 S=0 L=0
#define WW_SQ_CLAMPHI   0x0056  00000000 01010110  Q=1 S=5 L=2
#define WW_SQ_CLAMPLO   0x0055  00000000 01010101  Q=1 S=5 L=1
#define WW_SQ_NOCONVERT 0x0040  00000000 01000000  Q=1 S=0 L=0
#define WW_SQ_NOACCESS  0x0004  00000000 00000100  Q=0 S=1 L=0
#define WW_SQ_NOCOMM    0x0018  00000000 00011000  Q=0 S=6 L=0
```

It should be noted that when a client sends updated information to a server (via a DDE POKE or a SuiteLink Write), the data quality is assumed WW_SQ_GOOD, and this is the quality setting that is stored in the Toolkit database. The other flag settings, for "bad" quality, are set by the server-specific code and passed to the Toolkit via the API calls. These quality settings are then sent on to the client(s) as part of the VTQ data.

Updating Quality Flags

There are typically four places where you should update the quality flags for a point:

1. In UdprotPrepareWriteMsg( ) if there is any problem with poked data. When a client "pokes" a value to the server, ProtNewValueForDevice( ) passes the value to the server-specific code. This, in turn, calls a routine such as UdprotPrepareWriteMsg( ), which creates the byte sequence that will be sent to the device. If a string is too long, or a value is out of range, the server-specific code should force the value to a legal setting, set a quality of WW_SQ_CLAMPLO or WW_SQ_CLAMPHI, and report the information back to the Toolkit with DbNewVQFromDevice( ). Also, if the point is inaccessible or read-only, it may be appropriate to set the quality to WW_SQ_NOACCESS and call DbNewVQFromDevice( ).

2. In UdprotExtractDbItem( ) when the response from a device is interpreted as point data and reported to the Toolkit via DbNewVTQFromDevice( ). When a response carries valid data for the point, the quality should normally be set to WW_SQ_GOOD. However, if a string is too long, or a value is out of range, or a data conversion yields an invalid result, the quality should be set to WW_SQ_CLAMPLO, WW_SQ_CLAMPHI, or WW_SQ_NOCONVERT as is appropriate. Also, some PLCs return status information along with the data. This status information should be mapped to the corresponding WW_SQ_XX quality setting.

3. In ProcessValidResponse( ) and other locations where an error response is received, a bad quality should be reported for each point on a message that has a response pending. If a polling (read) message elicits an error response, a quality of WW_SQ_NOACCESS should be reported for each point that is being actively polled by that message. The following excerpts from UDPROTCL.C of the sample server illustrate this:

```
/*************************************************************/
/** Process validated response back from the device **/

void
WINAPI
ProcessValidResponse(LPPORT   lpPort)
{
    LPUDMSG         lpMsg;
    LPSTAT          lpTopic;
    BYTE FAR       *rsp;

    /* get pointer to current message, if any */
    lpMsg = lpPort->mbCurMsg;
    if (lpMsg == (LPUDMSG) NULL) {
        /* no current message, just return */
        return;
    }
```

```
    /* check whether message has changed */
    if (lpMsg->mmChanged) {
        /* If was changed after it was written
           -- ignore the response */
        return;
    }

    /* get pointer to corresponding station */
    lpTopic = lpMsg->mmTopic;
    if (lpTopic == (LPSTAT) NULL) {
        /* unable to access station structure, just return */
        return;
    }

    /* reset station retry limits */
    lpTopic->statRetries = TOPIC_CONSEC_FAILURE_LIMIT;
    lpTopic->statPortRetries = TOPIC_NORMAL_RETRIES;

    /*******************************************************\
      How messages are handled may depend on the protocol.
      The following example handles reads and writes
      and also processes error responses from the device.
    \*******************************************************/

    /* check type of message received */
    rsp = (BYTE FAR *) lpPort->mbRspBuffer;
    switch (*rsp) {
        case 0x00:    /* error message */
            /* indicate error */
            UdprotSetMsgQuality (lpTopic, lpMsg, WW_SQ_NOACCESS);
            if (ShowingErrors) {
                debug ("Error message received.");
                showReceivedData (lpPort);
            }
            break;
        default:
            /* check type of message */
            if (lpMsg->mmRead) {
                /* was a "read"
                   -- extract data from response and report it */
                UdprotExtractReadData(lpPort, lpTopic, lpMsg);
            }
            break;
    } /* switch */

    /* check type of message */
    if (!lpMsg->mmRead) {
        /* was a "write" -- delete the write message */
        UdprotDeleteCurWriteMsg(lpPort);
    }
    /* check station status,
       ensure we're out of slow poll mode */
    UdprotSetTopicStatus (lpPort, lpTopic, FALSE);
} /* ProcessValidResponse */
```

```
/*********************************************************/
/* set indicated quality for all symbols on a read message */

static
void
WINAPI
UdprotSetMsgQuality (LPSTAT    lpTopic,
                     LPUDMSG   lpMsg,
                     PTQUALITY ptQuality )
{
    LPEXTARRAY      lpSymbol_table;
    SYMPTR          lpSymEnt;
    unsigned long   firstSymIdx, lastSymIdx;
    BOOL            done;
    CHAINSCANNER    symbol_scanner;
    ACTIVE_CHECK    compValue;

    /* check whether pointers are valid */
    if ((lpMsg == (LPUDMSG) NULL) || (lpTopic == (LPSTAT) NULL))
        /* no message or no station, just return */
        return;

    /* only set quality if this is a READ message
       with active points */
    if ((!lpMsg->mmRead) || (lpMsg->mmActiveCt == 0))
        return;

    /* get pointer to symbol table */
    lpSymbol_table = &lpTopic->statSymTab;
    if (lpSymbol_table == (LPEXTARRAY) NULL) {
        /* cannot access symbol table structure, just return */
        return;
    }

    /* set up comparison value */
    compValue.lpMsg       = lpMsg; /* pointer to this message */
    compValue.finalHandle = lpMsg->mmLastSym;
                                   /* last symbol in message */
    compValue.SymHandle   = 0;     /* no symbol found yet */

    /* get handles for first and last symbols in this message */
    firstSymIdx = lpMsg->mmFirstSym;
    lastSymIdx  = lpMsg->mmLastSym;
    /* get pointer to first symbol entry
       referenced by the message */
    lpSymEnt = (SYMPTR) NULL;
    if (firstSymIdx >= SYM_OFFSET) {
        /* valid symbol handle, get pointer to symbol entry */
        lpSymEnt = (SYMPTR) GetExtArrayMemberPtr (lpSymbol_table,
                                      firstSymIdx - SYM_OFFSET);
```

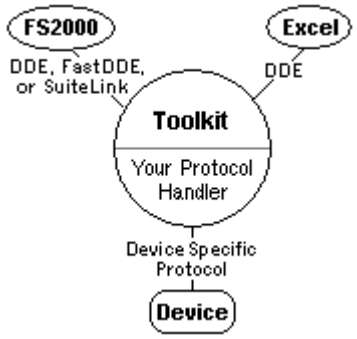
```
  /* get first active symbol referenced by this message */
        /* Note:  if found, compValue.SymHandle
                  will be non-zero */
        if (lpSymEnt != (SYMPTR) NULL)
            lpSymEnt = (SYMPTR) FindItemStartingAt (
                                (LPCHAINLINK) lpSymEnt,
                                &lpTopic->statSymUsed,
                                SCAN_FROM_HEAD,
                                IsActiveOnMessage,
                                &compValue,
                                &symbol_scanner);
    }

    /* scan through range of symbols covered by this message */
    done = FALSE;
    while ((!done) && (lpSymEnt != (SYMPTR) NULL)) {
        /* check whether symbol is active
           and polled by this message */
        if (compValue.SymHandle != 0) {
            /* match found,
               report new quality on indicated point */
            DbNewQFromDevice(lpMsg->mmIdLogDev,
                             lpSymEnt->msDbHnd, ptQuality);
        }
        /* check whether to continue scanning */
        if (lpSymEnt->msIndex + SYM_OFFSET == lastSymIdx) {
            /* last symbol handle for message, exit */
            done = TRUE;
        } else {
            /* get pointer to next active symbol on message,
               if any */
            lpSymEnt = (SYMPTR) FindNextItem (&symbol_scanner);
        }
    }
} /* UdprotSetMsgQuality */
```

4. In UdprotSetTopicStatus( ) when the communication with the PLC fails, bad quality of WW_SQ_NOCOMM should be flagged on all points on all messages for that topic. This would be at the point where the server goes into slow poll mode, attempting to re-establish communications with the PLC. The following excerpts from UDPROTCL.C of the sample server illustrate this:

```
/*******************************************************/
/** set STATUS item in station structure
    return TRUE if status is changed **/

static
BOOL
WINAPI
UdprotSetTopicStatus (LPPORT lpPort, LPSTAT lpTopic, unsigned
bFailed)
{
    BOOL changed;

    /* initialize return value */
    changed = FALSE;

    /* no change if pointer is NULL */
    if (lpTopic != (LPSTAT) NULL) {
        /* check current station status */
        if (lpTopic->statFailed != bFailed) {
            /* different from old status, update it */
            changed = TRUE;
            lpTopic->statFailed = bFailed;
            /* check whether station is active */
            if (lpTopic->statStatusActive) {
                /* active, indicate that station STATUS is due */
                lpTopic->statStatusDue = TRUE;
            }
            /* check whether new status is OK or failed */
            if (bFailed) {
                /* have logger indicate entering
                   slow poll mode */
                debug ("Entering slow poll mode"
                       " on topic \"%Fs\" on port %Fs.",
                       lpTopic->statTopicName,
                       lpPort->mbPortName);
                /* report new STATUS value if necessary */
                UdprotCheckAndUpdateStatus (lpTopic);
                /* set "no communication" quality flags
                   for all points polled on topic */
                UdprotSetTopicQuality (lpTopic, WW_SQ_NOCOMM);
            } else {
                /* have logger indicate leaving slow poll mode */
                debug ("Leaving slow poll mode"
                       " on topic \"%Fs\" on port %Fs.",
                       lpTopic->statTopicName,
                       lpPort->mbPortName);
            }
        }

        /* check whether station is failed */
        if (bFailed) {
            /* set up slow poll mode */
            UdprotSetSlowPollMode (lpPort, lpTopic);
        }
    }
    /* indicate whether station status changed */
    return (changed);
} /* UdprotSetTopicStatus */

/*******************************************************/
/* set indicated quality for all items on a topic */

static
void
```

```
WINAPI
UdprotSetTopicQuality( LPSTAT      lpTopic,
                       PTQUALITY   ptQuality )
{
    LPUDMSG         lpMsg;
    CHAINSCANNER    message_scanner;

    /* check whether pointer is valid */
    if (lpTopic == (LPSTAT) NULL)
        /* no station, just return */
        return;

    /* set indicated quality
       on every read message for this station */
    lpMsg = (LPUDMSG) FindFirstItem (&lpTopic->statReadMsgList,
                                     SCAN_FROM_HEAD,
                                     NULL, NULL, &message_scanner);
    while (lpMsg != (LPUDMSG) NULL) {
        /* set indicated quality on message */
        UdprotSetMsgQuality (lpTopic, lpMsg, ptQuality);
        /* advance to next message on this station */
        lpMsg = (LPUDMSG) FindNextItem (&message_scanner);
    }
} /* UdprotSetTopicQuality */
```

CHAPTER 8



Statistics Functions

This section describes the Statistics API functions provided by the Wonderware I/O Server Toolkit.

Contents

Overview

With the advent of FactorySuite 2000, the Wonderware I/O Server Toolkit provides extensions to the Toolkit API which allow I/O Servers to easily provide statistical measurements including counters and rates to I/O Clients. This API allows the server developer to register a statistic with a particular item name with the Toolkit. The Toolkit manages point validation and sends values to any interested clients. For statistics which indicate counters, the server protocol code simply manipulates the counter's value with the supplied function calls. For statistics which indicate rates, the server-specific code modifies the input counter and allows the Toolkit to calculate the rate of change. In addition to this capability for registering server-specific statistics, the Toolkit automatically provides its own internal statistics related to I/O Server activity and function call activity for performance and diagnostic purposes.

Statistics from a Client Perspective

Since the primary purpose of statistics is to provide users a convenient and easy way to gather performance and diagnostic information about their I/O Servers, it is appropriate to first discuss statistics from a user or client program perspective.

There are two basic types of statistics available to an interested I/O Client: **counters** and **rates**.

Statistical Counters

A statistical counter is used to represent an accumulated quantity (i.e. "a count") or state information. Most often, a counter will be used to indicate the number of operations or events that have occurred. For example, a counter would be useful for representing the number of transactions or the number of errors that have happened since the server was started. Alternatively, a counter can be used to represent a discrete state or a text string. So, for example, a counter could be used to indicate a 1/0 (good/bad) condition, a string listing all topics currently available on the server, or a string containing the description of the last error that occurred (e.g. "The connection has timed out").

The statistical API in the Toolkit makes it very easy to register and accumulate counters and have them automatically broadcast to any interested clients. A statistical counter can be any one of the four Toolkit datatypes: PTT_INTEGER, PTT_REAL, PTT_STRING, or PTT_DISCRETE. Functions for setting the value, adding/subtracting, or incrementing/decrementing counters are provided.

Statistical Rates

A statistical rate is used to represent the current rate at which a particular operation or event is occurring. For example, a rate would be useful for representing the number of read operations per second on a particular communications port.

The statistical API in the Toolkit makes it very easy to register and calculate rates and have them automatically broadcast to any interested clients. The server code registers the rate item with the Toolkit, assigning it a name, topic, update interval, and time units. Then, the server code simply needs to update the counter which feeds into the rate calculation when appropriate. The Toolkit takes care of calculating the rate of change of the counter in the specified time units and updates clients that are interested in the rate item. The counter value which is the input value to the rate calculation can optionally be associated with a previously registered statistical counter. This allows a server to provide both an accumulator and a rate of change for a particular measurement to clients without having to update two different counters.

Statistics Categories

Statistics should be segregated into different categories or groups depending on the type of object they are measuring. A group of statistics for a communications port, for example, should be separated from statistics related to DDE or SuiteLink traffic. To facilitate this categorization, several new standard logical devices, or I/O Server topics, have been defined as part of the FS2000 Toolkit. The purpose of each of these standard topics is to hold a logically related group of statistical information. These standard topics will be created by the Toolkit at server startup and deleted at server shutdown. They require no user configuration. Statistical items can be assigned to these new standard topics or to any user-defined topic in an I/O Server. This assignment of a statistic to a particular device or topic is the responsibility of the server developer and is controlled at statistic registration time via the API.

The new system standard topics are summarized in the following table.

Topic Name	Description	Symbol
SYSTEM	I/O Server System Topic	STDDEV_SYSTEM
$SERVER	contains general server wide statistics	STDDEV_SERVER
$PORT	contains statistics related to communication ports/boards	STDDEV_PORT
$DEVICE	contains statistics related to physical devices such as PLCs.	STDDEV_DEVICE
$DDE	contains statistics related to DDE and SuiteLink communications.	STDDEV_DDE
$TKITIF	contains statistics related to Toolkit API function calls.	STDDEV_TKITIF

In general, an I/O Server you create will only register statistics for the “$SERVER”, “$PORT”, and “$DEVICE” standard topics and for the user-defined topics known to the server. The generation of statistics for the “SYSTEM”, “$DDE”, and “$TKITIF” should be left to the I/O Server Toolkit.

Reset of Counters

I/O Clients have the ability to reset entire groups of statistical counter items. The statistical counters can be reset on a per-topic or per-server basis, depending on the item name and topic used in the DDE Poke or SuiteLink Write operation. The reset operation is controlled exclusively by the Toolkit and does not require any programming by the server developer or any special notification. Only counters which are integer or floating point accumulators are affected by a reset.

Topic	Item Name	Action when poked
SYSTEM	ResetAllStats	Resets all counter statistics on all topics in the I/O Server.
*	ResetStats	Resets all counter statistics on the indicated topic.

Manipulation of Counter Update Interval

The interval at which a counter is updated to any interested clients is set by the topic's counter interval. This interval defaults to 10 seconds but can be manipulated in several ways. First, the default value for all topics can be changed by adding an entry under the server section in the WIN.INI file in the Windows directory. The entry name is **StatCountersInterval**, and the units are in milliseconds.

The counter interval can also be controlled on an individual topic basis in two ways. First, the programmer can control it with the **StatSetCountersInterval()** function call. This function can be called anytime after a topic is created. Second, an I/O Client can control the counter update interval on a topic by manipulating the "CounterInterval" item name for the topic with a DDE Poke or SuiteLink Write. This mechanism requires no programming by the server developer.

Manipulation of Rate Intervals

The interval at which a particular statistical rate item is calculated is set to an initial default value by the server developer at the time the statistic is registered via the API. A general guideline is that the statistical rate item should be calculated at an interval greater than or equal to the topic update interval. It is important to remember that a server which has many rate statistics being calculated frequently may have some performance impact on the rest of the system. The server developer may want to provide some way to disable rate calculations if performance may be a concern. This might be accomplished by a configuration dialog which sets a flag in the server configuration file, the WIN.INI file, or the system Registry.

It may be desirable at times to allow a user, or I/O Client, to manipulate the interval at which a particular rate operation is performed. This capability is provided automatically by the I/O Server Toolkit. Once a rate item has been registered with the Toolkit, the rate interval for that item can be changed at any time by an I/O Client. The item name to be used for reading and writing the rate interval will always be the name of the rate item itself followed by the following string: "$INTERVAL". So, for example, a rate item of "SENDSRATE" could have its calculation interval manipulated by a DDE Poke or SuiteLink Write operation to the item "SENDSRATE$INTERVAL". This I/O Server Item is always an integer with millisecond units. Again, this dynamic rate interval manipulation is controlled exclusively by the Toolkit and does not require any programming by the server developer.

Statistics Names

The names assigned to statistical items may be controlled either by the I/O Server Toolkit or by the server developer. Certain pre-defined statistics, such as those related to DDE, SuiteLink, and Toolkit interface calls, will always exist in every server since the Toolkit defines and maintains them. (See the Toolkit Standard Statistics section below.) The naming of protocol-specific statistics will be the responsiblity of the server developer.

Statistical items are read-only, with the exception of special items for reset of counters and manipulation of counter and rate update intervals.

Toolkit Standard Statistics

The following table defines the standard statistical items which are provided automatically by the Toolkit:

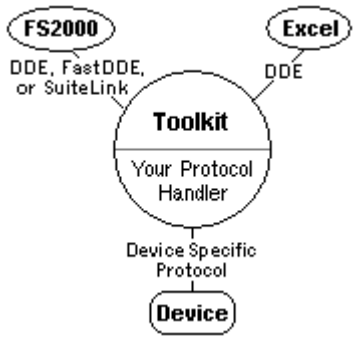
Topic	Item	Description
SYSTEM	CounterInterval	Interval in milliseconds at which counters on topic will update (integer).
SYSTEM	Formats	Tab separated list of DDE Formats supported by server (string).
SYSTEM	LastResetTime	Last time statistics in topic were reset (string).
SYSTEM	TopicItemList	Tab separated list of all I/O items available on topic (string).
SYSTEM	SysItems	Tab separated list of all I/O items available on system topic (string).
SYSTEM	Topics	Tab separated list of all I/O Topics, or logical devices, available in the server (string).
SYSTEM	ResetAll	Resets all counters on all topics.
SYSTEM	ResetStats	Resets all counters on topic (discrete).
SYSTEM	WatchDog	Watchdog timer which increments every interval indicated by CounterInterval (integer).
$SERVER	CounterInterval	Interval in milliseconds at which counters on topic will update (integer).
$SERVER	LastResetTime	Last time statistics in topic were reset (string).
$SERVER	ResetStats	Used to reset all counters on topic (discrete).
$SERVER	ServerBusyRate	Indicates fraction of CPU time during which server is busy (real).
$SERVER	ServerStartTime	Time server was started (string).
$SERVER	TopicItemList	Tab separated list of all I/O items available on topic (string).
$SERVER	WatchDog	Watchdog timer which increments every interval indicated by CounterInterval (integer).

Topic	Item	Description
$TKITIF	ActivatePoint	
$TKITIF	ActivatePointFailures	
$TKITIF	AllocateLogicalDevice	
$TKITIF	AllocateLogicalDeviceFailures	
$TKITIF	CounterInterval	
$TKITIF	CreatePoint	
$TKITIF	CreatePointFailures	
$TKITIF	DeactivatePoint	
$TKITIF	DeactivatePointFailures	
$TKITIF	DeallocateLogicalDevice	
$TKITIF	DeallocateLogicalDeviceFailures	
$TKITIF	DeletePoint	
$TKITIF	DeletePointFailures	
$TKITIF	LastResetTime	
$TKITIF	NewValueForDevice	
$TKITIF	NewValueForDeviceFailures	
$TKITIF	NewValueForDeviceRate	
$TKITIF	NewValueForDeviceRate$Interval	
$TKITIF	NewValueFromDevice	
$TKITIF	NewValueFromDeviceFailures	
$TKITIF	NewValueFromDeviceRate	
$TKITIF	NewValueFromDeviceRate$Interval	
$TKITIF	ResetStats	
$TKITIF	TopicItemList	
$TKITIF	WatchDog	
$DDE	CounterInterval	
$DDE	DDEAcksReceived	
$DDE	DDEAcksSent	
$DDE	DDEAdviseFailures	
$DDE	DDEAdvises	
$DDE	DDEAdvises	
$DDE	DDEBusyReceived	
$DDE	DDEDataBytesReceived	
$DDE	DDEDataBytesReceivedRate	
$DDE	DDEDataBytesReceivedRate$Interval	
$DDE	DDEDataBytesSent	
$DDE	DDEDataBytesSentRate	
$DDE	DDEDataBytesSentRate$Interval	
$DDE	DDEExecuteFailures	
$DDE	DDEExecutes	
$DDE	DDEInitiateFailures	

Topic	Item	Description
$DDE	DDEInitiates	
$DDE	DDENaksReceived	
$DDE	DDENaksSent	
$DDE	DDEPokeFailures	
$DDE	DDEPokes	
$DDE	DDEPokesRate	
$DDE	DDEPokesRate$Interval	
$DDE	DDERequestFailures	
$DDE	DDERequests	
$DDE	DDETerminateFailures	
$DDE	DDETerminates	
$DDE	DDEUnadviseFailures	
$DDE	DDEUnadvises	
$DDE	LastResetTime	
$DDE	ResetStats	
$DDE	TopicItemList	
$DDE	WatchDog	
*	CounterInterval	
*	LastResetTime	
*	ResetStats	
*	TopicItemList	Tab-separated list of statistical items.
*	WatchDog	

* Indicates all other topics.

CHAPTER 9



I/O Server Toolkit Function Summary

This chapter summarizes **Server Toolkit** Application Programming Interface (API) functions. The following briefly describes the categories of the **I/O Server Toolkit** functions and their primary purpose.

The discussions and illustrations on the following pages describe how the Toolkit handles client/server activities, such as initiating connections, advising points, poking values, etc. While the discussion is based on DDE, please note that the corresponding activities also take place when SuiteLink is used.

Note Wonderware I/O Server Toolkit functions that are underlined and bold, e.g., **ProtInit**(), must be developed. Wonderware I/O Server Toolkit functions that are provided by the Toolkit will be bolded but not underlined, e.g., **DbNewValueFromDevice**(). For more information, see the "API Function Reference" chapter for complete function descriptions.

Contents

Protocol Initialization & Setup Functions

This section describes the functions (indicated by the underlined names) that must be written and included in the I/O Server 's initialization. These functions are called by the Toolkit.

Function	Description
ProtClose()	Called immediately prior to the server closing. Performs any necessary clean-up such as freeing memory, closing communications ports, etc.
ProtDefWindowProc()	Allows the server application window to be customized.
ProtGetDriverName()	Provides the Toolkit with the name of the I/O Server.
ProtGetValidDataTimeout()	Returns the length of time the Toolkit is to wait for data before issuing a timeout to the DDE client requesting information.
ProtTimerEvent()	Called at the frequency specified in the call to **SysTimerSetupProtTimer**(). Allows the server to execute and perform tasks necessary to obtain the requested data.
ProtInit()	Allows the server to perform required initialization such as: • Obtaining defaults from WIN.INI (if applicable) • Reading the configuration file (if any) • Calls **SysTimerSetupProtTimer**() • Calls **SysTimerSetupRequestTimer**()

Logical Device Management Functions

The following functions are used by the Toolkit to allocate and deallocate logical devices.

Function	Description
ProtAllocateLogicalDevice()	Called when a client initiates a DDE conversation to the server. Returns a handle that the Toolkit will use in all future calls to routines dealing with that logical device *hLogDev*. The handle is an unsigned long which is meaningful to the server. It is used in all subsequent calls to identify the logical device instead of using the name. This way, the name (which is a string) need only be decoded once, when **ProtAllocateLogicalDevice**() is called. Must validate the topic name (i.e., logical device name). Must save the *idLogDev* for use in other Toolkit routines. **Note** If NULL is returned, the Initiate will not be *Acked*, i.e., the conversation will not be established.
ProtExecute()	Called when the client sends an Execute DDE message to a conversation on the server. This function should execute the string supplied by the client and return success or failure based on the command supplied.
ProtFreeLogicalDevice()	Called when the last client has sent a terminate to the server for this topic. All memory associated with this logical device must be freed.

Examples of Logical Device Management

The following examples summarize what happens regarding logical device management on a typical conversation initiation and termination.

What is called on Initiate

```
      DDE User         Toolkit       The Server
---- WM_DDE_INITIATE --->|
                         |--- ProtAllocateLogicalDevice ()--->
                         |    [if first use of device]
                         |<-- return handle to logical device ---
                         |
<--- WM_DDE_ACK ---------|    [if return is non-NULL or device
                         |     already exists]
```

What is called on a Terminate

```
      DDE User         Toolkit       The Server
---- WM_DDE_TERMINATE --->|
                          |--- ProtFreeLogicalDevice ()
                          |---> [if last use of device]
                          |<-----------------------------
<--- WM_DDE_TERMINATE ----|
```

Point/Item Management Functions

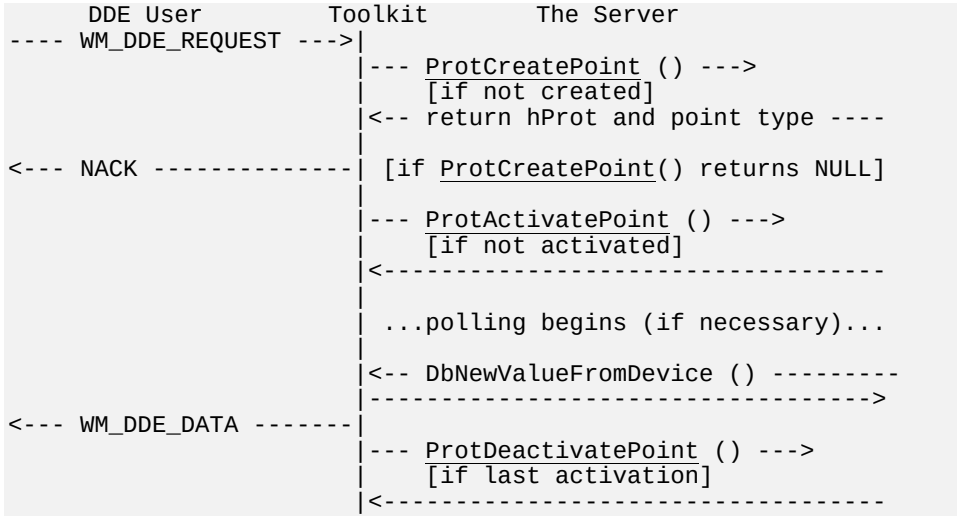
Point management includes defining points (setting up data structures), starting and stopping polling, sending and receiving data values and shutting down. Each item will have two handles (unique identifiers). One is given to the server by the Toolkit *hDb,* and used when the server calls the Toolkit concerning an item. The other is provided by the server *hProt* and given to the Toolkit. It will be used when the Toolkit calls the server concerning a point. When a valid point is being created, you must save the *hDb* database handle parameter. It will be the handle (unique identifier) used by the Toolkit database to find this item. The return *hProt* for a valid item is a non-zero handle that is unique to this point in this topic. The choice of these *hProt* handles is completely up to you and should make your later point processing more simple, e.g., use the index or memory handle pointing to an element in a chain of created points. Regarding point management, the following set of functions must be supplied to the Toolkit.

Function	Description
ProtActivatePoint()	Called when the Toolkit wants the server to start reporting changing point values by calling **DbNewValueFromDevice**(). At this point the server should start "polling" the point from the appropriate logical device. Calls to send data to the Toolkit via **DbNewValueFromDevice**() should only be done between the **ProtActivatePoint**() and the **ProtDeactivatePoint**() calls.
ProtCreatePoint()	Validates the point name and returns the type of point the name represents (discrete, integer, etc.) Must remember the *hDb* passed in since this is required for all calls to the Toolkit database **DbNewValueFromDevice**(). Must return *hProt* which represents a handle that is important to you. All subsequent calls that the Toolkit makes to you regarding this point will pass *hLogDev*, returned from **ProtAllocateLogicalDevice**() and *hProt*, returned from **ProtCreatePoint**(). This function should return NULL if the point is invalid or not accessible for this device. If this routine returns NULL, the DDE message that caused the call to **ProtCreatePoint**() will return a *NAACO*. (Do **not** call **DbNewValueFromDevice**() until the **ProtActivatePoint**() has been called.)

Function	Description
ProtDeactivatePoint()	Called when the Toolkit wants to stop reporting point value changes. At this point, the server should stop polling for this point.
ProtNewValueForDevice()	Called when the Toolkit wants to write a new value to the device. The *ptValue* parameter is a union that is used to pass point values to and from the Toolkit and supports a discrete, integer, real or string.
ProtDeletePoint()	Called when the Toolkit determines that no more clients are interested in this point. The server should delete or release any memory structures associated with the point. If a write is still outstanding, the server should still perform the write.

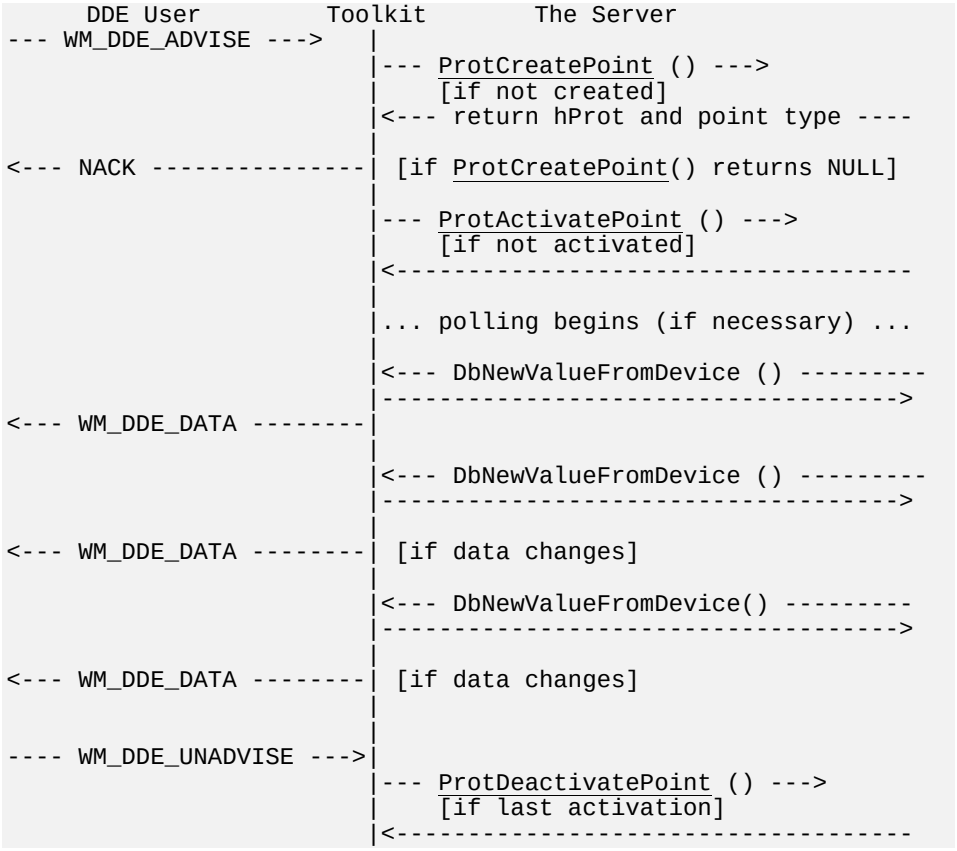
What is called on a Request

A *Request* is a one-time data transfer.



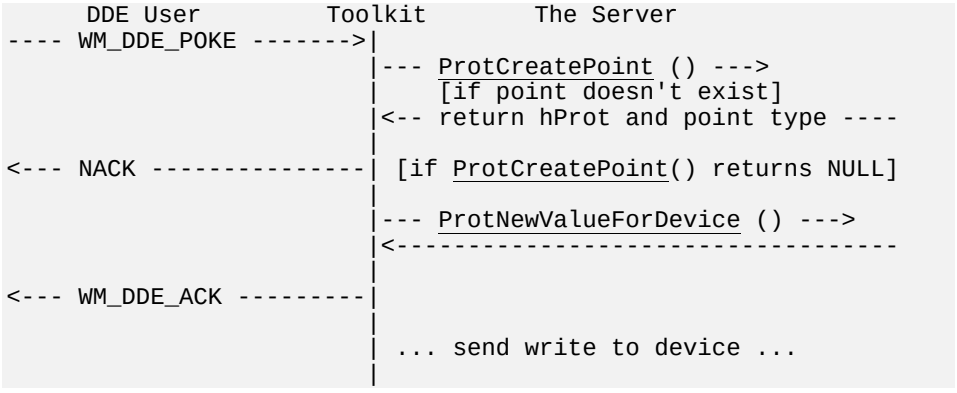
What is called on an Advise/Unadvise

An *Advise* asks for a notification of changes. An *Unadvise* asks for a discontinuation of the notifications.



What is called on a Poke

A *Poke* is a one-time write of a point value.



A *ptValue* is a union that is used to pass point values to and from the Toolkit. The *ptValue* supports discrete, integer, real and string. Strings are handled in a special way - via the functions that are provided for you in TOOLKIT7.LIB. Refer to the API Function Reference chapter for more details on the above routines.

Toolkit Database Interface for Protocol Functions

This section describes the functions that are provided in TOOLKIT7.LIB to allow the server to update the Toolkit database.

Function	Description
DbNewVTQFromDevice()	Is called every time a point value is retrieved from the device. The Toolkit will determine whether or not the value has changed since the last time it was read. This is critical for compatibility with later versions of the Toolkit.
DbNewTQFromDevice()	Is called when communication problems with the PLC occur. Errors or a lost connection means every point on the effected message or topic should be marked as having bad quality.
DbSetHProt()	Allows the server to change the *hProt* value for a point (item identifies from your code to the Toolkit database).

Timer Functions

This section describes the timer functions that must be called to set up the Toolkit timers at rates appropriate for the protocol being used.

Function	Description
SysTimerSetupProtTimer()	Sets up a timer that goes off every *dwMsec* milliseconds and calls **ProtTimerEvent**().
	This timer should be set to a value that is reasonable for the protocol data supply rate. Intervals that are arbitrarily too short will result in needless system overhead.
SysTimerSetupRequestTimer()	Sets up a timer that goes off every *dwMsec* milliseconds and checks for valid data timeout errors within the Toolkit database.
	This timer should be set to a value that is reasonable for the protocol data timeout. The interval should be a reasonable factor of the ValidDataTimeout parameter returned in the **ProtGetValidDataTimeout**() calls. Intervals that are arbitrarily too short will result in needless system overhead.

String PTVALUE Manipulation Functions

The PTVALUE union is used to pass point values between the various functions described in other sections of this document (for example, **ProtNewValueForDevice**()).

The following describes the functions that **must** be used to manipulate the *hString* field of this structure.

Function	Description
StrValSetNString()	Initializes a *ptValue* string and does not require a NULL-terminated string.
StrValSetString()	Initializes a *ptValue* string.
StrValStringFree()	Frees the associated string memory.
StrValStringLock()	Locks a string in memory to allow access to it.
StrValStringUnlock()	Unlocks the memory associated with a string.

Example Code #1 - Sending Data to the Toolkit Database

```
/* Send a string item to the database. Replace our local
    string first. */
ptValue.hString = NULL;
ptValue = StrValSetString(ptValue,lpszValNew);
DbNewValueFromDevice( idDev, hDbItem, ptValue );
    /* The Toolkit will free the ptValue memory for me.
        This ptValue is now untouchable, forget it. */
```

Example Code #2 - Accepting Data from the Toolkit for the Device

```
ProtNewValueForDevice( hLogDev, hProt, ptValue )
    ...
    switch (info[i].ptype)
    case PTT_STRING:
        lpszValNew = StrValStringLock( ptValue );
    ...
        /* use string */
    ...
        StrValStringUnlock( ptValue );
        break;
    ...
```

Caveats with StrVal Strings

Caveat #1 - Modifying PTVALUE without StrValSetString()

Caution must be taken when using strings in ptValue.

The following code is a BAD example:

```
BOOL
FAR PASCAL
ChangeValue( ptValue, lpszNewVal )
PTVALUE     ptValue;
LPSTR       lpszNewVal;
{
    LPSTR   lpszVal;
    BOOL    rtn;
    rtn = FALSE;
    if( ptValue.hString != NULL )  {
        lpszVal = StrValStringLock( ptValue );
        if( lpszVal )  {
       /* UNSAFE - the memory allocated for
        ptValue may not be enough for lpszNewVal! */
        lstrcpy( lpszVal, lpszNewVal );
        StrValStringUnlock( ptValue );
        rtn = TRUE;
        }
    }
    return( rtn );
}
```

The moral of the above example is to always use **StrValSetString**() to change the value of a string. This will correctly size the memory needed to store the string (by freeing the memory associated with the old string and allocating new memory for the new string).

Caveat #2 - Modifying PTVALUEs from Database

When the database calls **ProtNewValueForDevice**(), consider the *ptValue* to be read-only. The *hString* in the *ptValue* passed to **ProtNewValueForDevice**() is stored in the Toolkit database. If you cause this memory to be freed, the database may malfunction. Thus, only call **StrValStringLock**() and **StrValStringUnlock**() on the *ptValue* received from **ProtNewValueForDevice**()! **Never** call **StrValSetString**() or **StrValStringFree**() with this *ptValue*!

An example of **correct** usage of a string in **ProtNewValueForDevice**() is as follows:

```
BOOL
FAR PASCAL
ProtNewValueForDevice( hLogDev, hProt, ptValue )
HLOGDEV    hLogDev;
HPROT      hProt;
PTVALUE    ptValue;
{
char    buffer[200];
BOOL    rtn;
    rtn = FALSE;
    /* This device handles only string point values. */
    if( hLogDev = 1  &&  hProt = 1  &&
        ptValue.hString != NULL )  {
        /* Lock the read-only input string. */
        lpszVal = StrValStringLock( ptValue );
        if( lpszVal )  {
            lstrcpy( (LPSTR)buffer, lpszVal );
            StrValStringUnlock( ptValue );
            /* Now we can process the input string in the local
                buffer*/
            rtn = TRUE;
        }
    }
    return( rtn );
}
```

Memory Management Functions

Wonderware has developed a heap manager that is similar to GlobalAlloc(), GlobalLock(), GlobalUnlock(), GlobalFree(), and GlobalReAlloc(). This heap manager allows the developer to allocate many different sized memory blocks while minimizing the number of actual global memory handles (selectors) consumed. The corresponding calls are **wwHeap_Init**(), **wwHeap_AllocPtr**(), **wwHeap_ReAllocPtr**(), **wwHeap_FreePtr**() and **wwHeap_Release**(). All of these functions are described in detail in the "API Function Reference" chapter.

For compatibility reasons, Wonderware has extended the heap manager to run on Windows NT/ 2000. A new common API has been developed which supports heap management on both Windows 98 and Windows NT / 2000. This new heap API will work on either platform. If you have an existing I/O Server developed using the old heap API functions, you will need to convert these function calls to the new API prior to building your server. Refer to the "Getting Started with the I/O Server Toolkit" chapter for details.

Function	Description
wwHeap_AllocPtr()	Allocate the specified amount of memory using the heap specified by *hHeap*.
wwHeap_FreePtr()	Free the allocated memory specified by *lpPtr*.
wwHeap_Init()	Create and initialize a heap.
wwHeap_ReAllocPtr()	Re-allocate the specified amount of memory used in the heap specified by *lpPtr*.
wwHeap_Release()	Free a heap created with **wwHeap_Init**().

Memory Access Permission Functions - Windows Only

For specialized applications that require access to fixed real mode addresses in memory, there are two routines to support this addressing in Windows protected-mode: **RequestPermission**() and **RelinquishPermission**().

Function	Description
RelinquishPermission()	Will release the selector that was allocated to this memory address range, identified by *hPermission*.
RequestPermission()	Access memory at a fixed Real Mode location.

Common Dialog Functions

Version 5.0 of the I/O Server Toolkit provides a new Dynamic Link Library (DLL) called WWCOMDLG.DLL. This DLL provides a set of functions that are available to the I/O Server developer for the purpose of providing a standard look and feel for dialogs, message boxes, and graphical controls that are common to all servers. All serial servers, for example, need to provide some way of configuring communications ports. The Wonderware Common Dialog DLL provides the **WWConfigureComPort**() function for this purpose.

Although not required, the Wonderware Common Dialog functions are recommended for all I/O Servers. They provide a standard graphical user interface for server configuration and message boxes. The UDBOARD and UDSERIAL sample servers included with the Toolkit have been upgraded to include usage of the Wonderware Common Dialog functions.

Function	Description
WWCenterDialog()	Places the specified dialog at the center of the screen.
WWConfigureComPort()	Displays and manages the communications port settings configuration dialog. It should only be used by serial servers. This dialog allows configuration of communications parameters for one or more serial communications ports. These settings are written to the **WW_CP_DLG_LABELS** structure for use by the server.
WWConfigureServer()	Displays and manages the server parameter configuration dialog. This dialog allows configuration of several parameters related to overall operation of the I/O Server. These settings are written out as profile information to the WIN.INI file by this dialog function and only take effect upon restarting of the server.
WWConfirm()	Displays and manages the confirmation dialog which indicates the directory or file where server settings are to be saved. It should only be called when the configuration file does not currently exist.
WWDisplayAboutBox()	Displays and manages the dialog displaying copyright and version information. It is generally intended for use by Wonderware servers since it displays the Wonderware copyright information. However, it does provide facilities for easily displaying version and date information and is available for use by other servers.
WWDisplayConfigNotAllow()	Displays a message box indicating that configuration of the server is not allowed while the server is in use.
WWDisplayErrorCreating()	Displays a message box indicating that an error was encountered while creating the specified file.
WWDisplayErrorReading()	Displays a message box indicating that an error was encountered while reading the specified file.
WWDisplayErrorWriting()	Displays a message box indicating that an error was encountered while writing the specified file.

Function	Description
WWDisplayKeyNotEnab()	Displays a message box indicating that the installed security key does not enable operation of this I/O Server.
WWDisplayKeyNotInst()	Displays a message box indicating that the required security key is not installed on the system.
WWDisplayOutofMemory()	Displays a message box indicating that an error was encountered while allocating memory for the specified object.
WWFormCpModeString()	Creates a null-terminated string containing device control information.
WWGetDialogHandle()	Returns a window handle to the top-most dialog in the current application.
WWInitComPortComboBox()	Creates a communications port selection box for display on a dialog. Most commonly, it will be used in a topic configuration dialog for selection of the communications port for serial communications.
WWSelect()	Displays a dialog containing a list box which will contain a list of strings specified by the server. The user will be provided options for adding, modifying, or deleting entries from this list. This function is most commonly used to display a list of topics or boards for configuration.
WWTranslateCDlgToWinBaud()	Translates the WWCOMDLG constant for baud rate to the Windows equivalent.
WWTranslateCDlgToWinData()	Translates the WWCOMDLG constant for number of data bits to the Windows equivalent.
WWTranslateCDlgToWinParity()	Translates the WWCOMDLG constant for parity to the Windows equivalent.
WWTranslateCDlgToWinStop()	Translates the WWCOMDLG constant for number of stop bits to the Windows equivalent.
WWTranslateWinBaudToCDlg()	Translates the Windows constant for baud rate to the WWCOMDLG equivalent.
WWTranslateWinDataToCDlg()	Translates the Windows constant for number of data bits (7 or 8) to the WWCOMDLG equivalent.

Function	Description
WWTranslateWinParityToCDlg()	Translates the Windows constant for parity to the WWCOMDLG equivalent.
WWTranslateWinStopToCDlg()	Translates the Windows constant for number of stop bits to the WWCOMDLG equivalent.
WWVerifyComDlgRev()	Verifies that the version of WWCOMDLG.DLL installed on the system is at least as new as the specified version. It also returns the major and minor version numbers of the installed WWCOMDLG.DLL to the server. This function is intended for compatibility checking.

Selection Boxes - Optional

The following list of functions provide selection box capability, as used in **InTouch**. In general, these functions are no longer necessary or recommended since the **WWSelect**() function provides similar functionality.

Function	Description
SelBoxAddEntry()	Add a string entry to a selection box set of choices to be displayed when the **SelBoxUserSelect**() call is done.
SelBoxSetupStart()	Does the basic setup for a selection box.
SelBoxUserSelect()	Actually displays the selection box and processes all the buttons.
SelBoxUserSelection()	Call this routine after **SelBoxUserSelect**() if you wish to get a list of what the user selected.
SelListFree()	Frees the memory associated with the selection list.
SelListGetSelection()	Gets the indicator value for a selection at the specified number.
SelListNumSelections()	Returns how many entries are in the specified *hSelList*.

Miscellaneous Functions

This section describes the miscellaneous functions that can be used.

Function	Description
AdjustWindowSizeFromWinIni()	Adjusts the size of the server window to the last saved size.
CheckConfigFileCmdLine()	Checks the command line for optional configuration file path. By default, most of the I/O Servers save their configuration file path in the Windows WIN.INI file. If the user defines the file path on the command line (default), or an alternate file is used, the server, with proper coding, can read the configuration file other than the one specified in the WIN.INI file.
debug()	Sends a debug message to WWLOGGER.EXE which can log it to disk, monochrome adapter, AUX port, etc. The **#include "debug.h"** is a required include file.
GetAppName()	Returns the application name for the server. This application name is stored in the Toolkit and initially set by a call to **ProtGetDriverName**() during server startup.
GetString()	Retrieves a string from the resource file.
UdInit()	Is intended for use by a Windows application that already exists and needs to be extended to include the DDE capability provided by the Toolkit. It is used to initialize the I/O Server Toolkit and should only be used by a Windows application which supplies its own WinMain() function. **UdInit**() should be called early in the activation of such applications. Most I/O Servers do not have to call this function since they allow the Toolkit to supply the WinMain() function. The parameter list is identical to the parameter list for the Windows WinMain() function.
UdReadAnyMore()	Reads a *bAnyMore* flag from the configuration file. This flag is used within the configuration file to indicate whether more records of a certain type exist. Such a flag is usually necessary when the number of records is unknown.
UdReadVersion()	Reads the version number from the server configuration file. It also verifies that the magic number stored in the file matches the specified magic number.

Function	Description
UdTerminate()	Is intended for use by Windows applications that already exist and need to be extended to include the DDE capability provided by the toolkit. It is used to close the I/O Server Toolkit, but it should only be used by a Windows application which supplies its own WinMain() function and calls the **UdInit**() function to initialize the toolkit. **UdTerminate**() should be called at application shutdown time. Most I/O Servers do not have to call this function since they allow the toolkit to supply the WinMain() function.
UdWriteAnyMore()	Writes a *bAnyMore* flag to the configuration file. This flag is used within the configuration file to indicate whether more records of a certain type exist.
UdWriteVersion()	Writes the version number, magic number, date, and time to the server configuration file.
WriteWindowSizeToWinIni()	Saves the size of the server window to the last adjusted size.

Windows NT Porting Functions

Version 5.0 of the I/O Server Toolkit includes an Windows NT version that provides support for the Windows NT/2000 operating system. Since there are a large number of I/O Servers which currently only operate on the Windows operating system, a significant amount of porting work will be required to provide native Windows NT support for these servers. To minimize this porting effort and, at the same time, to simplify software maintenance, Wonderware has developed a set of porting functions and macros that can be utilized for Windows NT. These tools will significantly reduce the amount of time required to port a Windows server to the Windows NT environment.

Although not required, the Wonderware NT porting functions and macros are recommended for all I/O Servers. They provide a common interface that allow the same function call to be used on both Windows and Windows NT. The UDBOARD and UDSERIAL sample servers included with the Toolkit have been upgraded to include usage of Wonderware NT porting functions and macros.

There are three specific areas for Windows NT porting, they are:

- Windows Function Emulators
- Windows and Windows NT Compatibility Functions
- Macros for Portability

Windows Function Emulators

The following set of functions provide Windows NT emulation of Windows functions. They are only available in the Windows NT I/O Server Toolkit since the Windows API already incorporates them.

Function	Description
CloseComm()	Closes a serial communications port.
FlushComm()	Flushes all characters from the transmission or receiving queue of the specified communications device.
GetCommError()	Retrieves the most recent error value and current status for the specified device. It also clears the error.
GetCommEventMask()	Retrieves and then clears the event word for the specified communications device.
GetTextExtent()	Provides emulation of the Windows **GetTextExtent**() function for the Windows NT platform.
OpenComm()	Opens a serial communications port for communications.
ReadComm()	Reads up to a specified number of bytes from the given communications device.
SetCommEventMask()	Does nothing on Windows NT. It is provided for common code convenience only.
WriteComm()	Writes the specified bytes to the specified communications device.

Windows/Windows NT Compatibility Functions

These functions are available in both the Windows and Windows NT I/O Server Toolkit and provide a compatible function to allow common code on both platforms.

Function	Description
EnableCommNotification()	Simulates the Windows **EnableCommNotification**() function for Windows versions older than Windows 3.1 and for Windows NT. It performs no operation on these platforms and is provided for common code convenience only. On Windows versions prior to 3.0, it enables or disables WM_COMMNOTIFY message posting to the given window.
NTSrvr_BuildCommDCB()	Translates a device-definition string into appropriate serial device control block codes. The return value from this function is compatible with the Windows version of the **BuildCommDCB**() function.
NTSrvr_GetCommState()	Retrieves the device control block for the specified device. The return value from this function is compatible with the Windows version of the **GetCommState**() function.
NTSrvr_SetCommState()	Sets a communications device to the state specified by a device control block. The return value from this function is compatible with the Windows version of the **BuildCommDCB**() function.
NTSrvr_SetDCB_Dtr()	Modifies the DTR (data-terminal-ready) flow-control setting in the device control block.
NTSrvr_SetDCB_Rts()	Modifies the RTS (request-to-send) flow-control setting in the device control block.
PfnSendEmSelectAll()	Selects all the text in the identified edit control. It will also scroll the caret into view if the bScrollCaret flag is TRUE.
PfnSendEmSelectRange()	Selects a range of text in the identified edit control. It will also scroll the caret into view if the bScrollCaret flag is TRUE.

Macros for Portability

The following macros have been provided in the NTCONV.H header file to assist in developing common code servers for Windows and Windows NT. These macros should be self-explanatory by looking at their definitions .

Windows NT-only MACROS

```
#define huge
#define LocalInit(a, b, c)      (1)
#define LockData(a)

#define lstrchr                 strchr
#define lstrncpy                strncpy
#define lstrcpyn                strncpy

#define MoveTo(A,B,C)           MoveToEx(A, B, C, NULL)
```

Windows and Windows NT MACROS

```
#ifdef WIN32
#define FARWNDPROC              WNDPROC
#else
#define FARWNDPROC              FARPROC
#endif

#ifdef WIN32
#define SM_MINUS_ONE            (HWND) 0xFFFFFFFF
#else
#define SM_MINUS_ONE            0xFFFF
#endif

#ifdef WIN32

#define OPENRead(A)             _open(A, _O_RDONLY | _O_BINARY)
#define OPENCreate(A)           _open(A, _O_BINARY | _O_RDWR |
_O_TRUNC | _O_CREAT,
     _S_IREAD | _S_IWRITE);
#define LWRITE                  _write
#define LREAD                   _read
#define LSEEK                   _lseek
#define LCLOSE                  _close
#else
#define OPENRead(A)             open(A, O_RDONLY | O_BINARY)
#define OPENCreate(A)           open(A, O_RDWR | O_BINARY |
O_TRUNC | O_CREAT, S_IREAD |
     S_IWRITE);
#define LWRITE                  _lwrite
#define LREAD                   _lread
#define LSEEK                   _llseek
#define LCLOSE                  close
#endif
```

Windows-only MACROS

```
#define InSendMessage()         (FALSE)
#define FreeDDElParam(a, b)
#define PackDDElParam(a, b, c)      (MAKELONG((b), (c)))
#define UnpackDDElParam(a, b, c, d) ((*(c) = LOWORD(b)) | (*(d) =
HIWORD(b)) | 1)

#if (WINVER < 0x030a)
typedef unsigned int    UINT ;
typedef UINT            WPARAM ;
typedef long            LPARAM ;
#endif
```

Additional Information

Required External Data

The following data items are required by the I/O Server Toolkit:

```
extern  HWND        hWndParent
extern  HANDLE      hInst
```

These data variables must be declared in the server for the Toolkit to function properly.

About the .DEF File

A .DEF file for the main executable is not needed for Win32 servers.

About the .RC File

The .RC (resource) file must contain the following items for the Toolkit to work properly:

- Stringtable with PROTLIB.STR
- **#include** "**tkitstrt.rci**"

Note If the **#include** "**tkitstrt.rci**" is not in the resource file, the I/O Server will not start.

Adding Help to the I/O Server

Windows will use the program WINHELP.EXE to display the on-line help *.hlp files provided with any application. The TOOLKIT7.LIB provides a standard mechanism for providing Help for the I/O Servers. We recommend that you use Microsoft Word to generate the basic documentation file that will be used by the help compiler for Windows (provided with the SDK) to generate the server .HLP file.

Note Refer to the Microsoft Professional Tools User's Guides for detailed information.

The serial and board sample server programs both show the basic use of a Help menu item. If you start with an existing server and need to add the Help capability, follow these general steps:

1. Add the following include statement to the .RC file and .C file that does initialization and message processing.

   #include "srvrhelp.h"

2. Add the following help menu items to the .RC file:

```
POPUP  "&Help"
     BEGIN
     MENUITEM  "&Contents",    MENU_HELP_INDEX
     MENUITEM  "&How to Use Help",   MENU_HELP_ON_HELP
     MENUITEM  SEPARATOR
     MENUITEM  "&About",       MENU_HELP_ABOUT
     END
```

3. In one of the .C files add the following variable declaration statement, which will be picked up by the Toolkit functions:

   BOOL bDoHelp = TRUE;

4. Add "About" message processing for the WM_COMMAND / MENU_HELP_ABOUT. Refer to the "Serial" sample server UDMAIN.C to see sample About processing.

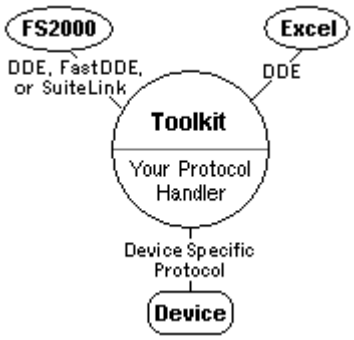
5. N ow you must generate the server .HLP file, where server is the same name used for the I/O Server's .EXE file and application name. For example, UDSAMPLE.HLP for the server UDSAMPLE.EXE.

6. Be sure to copy the .HLP file to the installation floppy and/or into a directory in the PATH.

Note There are several third-party help development software packages on the market that can be used to simplify the creation of the help file once the documentation file has been created. For example, "RoboHelp" by Blue Sky Software in La Jolla, California.

CHAPTER 10

API Function References



This chapter is a complete reference section for the **I/O Server Toolkit** Application Programming Interface (API) functions. They are presented in alphabetic order. The purpose, syntax, parameters and possible return values for all functions are included. Functions that appear both **bolded** and **underlined** must be written and included in the I/O Server. These functions are called by the Toolkit.

AdjustWindowSizeFromWinIni

VOID WINAPI

AdjustWindowSizeFromWinIni (HWND *hWnd*)

This function adjusts the size and position of the server window to the last saved size.

Parameter	Description
hWnd	Window to size.

CheckConfigFileCmdLine

VOID WINAPI

CheckConfigFileCmdLine(LPSTR *lpszCmdLine*,
LPSTR *lpszCfgPath*,
int *nMaxString*)

This function will check the command line for a string prefaced by "/D:". Be sure to add CMDLNPTH.H to the list of included files and add this reference:

extern char szCmdLine[ ];

Parameter	Description
lpszCmdLine	Far string pointer to the *szCmdLine*.
lpszCfgPath	Far string pointer to the string containing the final configuration path that may be used for reading the file.
nMaxString	Maximum length of the *szCfgPath*.
Return Value	None.
Comments	The syntax supporting this command line file path definition is: SERVER EXENAME /D:{FILEPATH} For example: MODBUS /D:C:\MBTEST

Note Before **ProtInit**() is called, the Toolkit initializes the string variable, szCmdLine, to contain the command line that invoked the driver. The variable szCmdLine should be the first argument to **CheckConfigFileCmdLine**().

Example

```
char szCfgPath[ PATH_STRING_SIZE ];
VOID FAR PASCAL GetConfigFilePath( VOID ){
    char  driverName[ 20 ];
    extern  char  szCmdLine[];
    ProtGetDriverName( driverName, sizeof(driverName) );
    /* Get the ConfigurationFilePath from the WIN.INI file
       or set it to default to the current directory.   */
    GetProfileString( driverName, "ConfigurationFile", "",
        szCfgPath, PATH_STRING_SIZE );

    /*  If the command line specified a config file path,
        let it override the path stored in WIN.INI.    */
    CheckConfigFileCmdLine( szCmdLine, szCfgPath,
        PATH_STRING_SIZE);
    if( strlen( szCfgPath ) == 0 ) {
        getcwd( szCfgPath, PATH_STRING_SIZE );
        if (Verbose ){
            debug( "Config path is CWD %s", szCfgPath );
        }
    }
    if ( szCfgPath[strlen(szCfgPath)-1] != '\\' ) {
        strcat( szCfgPath, "\\" );
    }
}
```

CloseComm

int

CloseComm(int *idComDev*)

This function closes a serial communications port.

Parameter	Description
idComDev	The *id* of the device to close. The **OpenComm**() function returns this value.
Return Value	The return value is zero if the function is successful. Otherwise, it is less than zero.
Comments	Windows NT/2000 only. Emulates Windows function.

DbDevGetName

void WINAPI

DbDevGetName (IDLDEV *idLogDev*,
LPSTR *lpszTopicName*)

Get name corresponding to indicated logical device (i.e. topic).

Parameter	Description
idLogDev	Topic (logical device) identifier that was supplied by the Toolkit as a parameter in the **ProtAllocateLogicalDevice**() call.
lpszTopicName	Far pointer to the string buffer where the name will be returned.

DbGetGMTasFiletime

BOOL WINAPI

DbGetGMTasFiletime(FILETIME **pFileTime*)

Get Greenwich Mean Time as a Win32 FILETIME structure.

FILETIME is a 64-bit value defined in Win32 as the number of 100 nanosecond intervals since January 1, 1601. It is organized as two DWORDS, dwLowDateTime and dwHighDateTime.

Parameter	Description
pFileTime	Far pointer to a FILETIME structure in which to store the date/time stamp.
Return Value	TRUE if the date/time stamp was obtained successfully.

DbGetName

void WINAPI

DbGetName(IDLDEV *idLogDev*,
HDB *hDb*,
LPSTR *lpszName*)

Get name of database item for indicated logical device.

Note: This is the same as the following function:

VOID WINAPI UDDbGetName (IDLDEV idLogDev, HDB hDb, LPSTR lpszName)

Parameter	Description
idLogDev	Topic (logical device) identifier that was supplied by the Toolkit as a parameter in the **ProtAllocateLogicalDevice**() call.
hDb	Handle from the Toolkit that is unique to this item and was supplied as a parameter in the **ProtCreatePoint**() call.
lpszName	Far pointer to the string buffer where the name will be returned.

DbGetPointType

PTTYPE WINAPI

DbGetPointType (IDLDEV *idLogDev*,
HDB *hDb*)

Get point type of indicated point from database.

Returns 0 (PTT_UNKNOWN) if database pointer invalid.

Parameter	Description
idLogDev	Topic (logical device) identifier that was supplied by the Toolkit as a parameter in the **ProtAllocateLogicalDevice**() call.
hDb	Handle from the Toolkit that is unique to this item and was supplied as a parameter in the **ProtCreatePoint**() call.
Return Value	The type index should be one of the following, according to the data type for the point: PTT_DISCRETE, PTT_INTEGER, PTT_REAL, PTT_STRING. If the database pointer *hDb* is invalid, a value of 0 (PTT_UNKNOWN) is returned.

DbGetPtQuality

PTQUALITY WINAPI

DbGetPtQuality (IDLDEV *idLogDev*,
HDB *hDb*)

Get quality flags for indicated point.

Parameter	Description
idLogDev	Topic (logical device) identifier that was supplied by the Toolkit as a parameter in the **ProtAllocateLogicalDevice**() call.
hDb	Handle from the Toolkit that is unique to this item and was supplied as a parameter in the **ProtCreatePoint**() call.
Return Value	The quality flags indicate whether the Toolkit database entry for the point contains good data or some problem exists. See the chapter on Quality Flags for information on the specific flag settings.

DbGetPtTime

PTTIME WINAPI

DbGetPtTime(IDLDEV *idLogDev*,
HDB *hDb*)

Get date/time stamp for indicated point

Parameter	Description
idLogDev	Topic (logical device) identifier that was supplied by the Toolkit as a parameter in the **ProtAllocateLogicalDevice**() call.
hDb	Handle from the Toolkit that is unique to this item and was supplied as a parameter in the **ProtCreatePoint**() call.
Return Value	The date/time stamp is stored as a Win32 FILETIME structure, which is a 64-bit value indicating the number of 100 nanosecond intervals elapsed since January 1, 1601. The date/time stamp indicates when the point information was last updated in the Toolkit database.

DbGetValueForComm

PTVALUE WINAPI

DbGetValueForComm(IDLDEV *idLogDev*,
HDB *hDb*)

Retrieve value for indicated point/logical device from the database.

Generally, this is used internally by the Toolkit to get the data value that is to be sent to a particular client.

Parameter	Description
idLogDev	Topic (logical device) identifier that was supplied by the Toolkit as a parameter in the **ProtAllocateLogicalDevice**() call.
hDb	Handle from the Toolkit that is unique to this item and was supplied as a parameter in the **ProtCreatePoint**() call.
Return Value	A union of type PTVALUE, the value corresponding to the indicated data point. Note that for points of type PTT_STRING, the value contains a pointer to a memory buffer where the string is actually stored.

DbNewQForAllPoints

BOOL WINAPI

DbNewQForAllPoints(IDLDEV *idLogDev*,
PTQUALITY *ptQuality*)

This function is generally used when a problem occurs communicating to the PLC. Then the quality flags and time stamps for all affected points should be set to indicate the problem, without changing the value of the points. This function sets the quality on all points in the Topic. The Toolkit supplies a default date/time stamp. See the chapter on Time Marks for more information on the default time.

Note: You may instead prefer to scan through the list(s) of points yourself and call **DbNewTQFromDevice**() for specific points, depending on the specific needs of your server and whether different quality settings are needed for different points.

Parameter	Description
idLogDev	Topic (logical device) identifier that was supplied by the Toolkit as a parameter in the **ProtAllocateLogicalDevice**() call.
ptQuality	The quality flags, indicating whether the point was accessed properly and whether the data had any problems.
Return Value	TRUE means the parameters were acceptable and the time and quality have been moved into the database.
Comments	If the quality of a point changes in the Toolkit database, the value, time, and quality will be passed on to any clients who have *advise*d the item. (DDE clients are updated only when the value changes.) When a client requests an item, the current database VTQ will be sent (once the item's value has been set).

DbNewQFromDevice

BOOL WINAPI

DbNewQFromDevice(IDLDEV *idLogDev*,
HDB *hDb*,
PTQUALITY *ptQuality*)

This function is generally used when a problem occurs communicating to the PLC. Then the quality flags and time stamps for all affected points should be set to indicate the problem, without changing the value of the points.

Functionally, this is similar to **DbNewTQFromDevice**(), except that the Toolkit supplies a default date/time stamp. See the chapter on Time Marks for more information on the default time.

Parameter	Description
idLogDev	Topic (logical device) identifier that was supplied by the Toolkit as a parameter in the **ProtAllocateLogicalDevice**() call.
hDb	Handle from the Toolkit that is unique to this item and was supplied as a parameter in the **ProtCreatePoint**() call.
ptQuality	The quality flags, indicating whether the point was accessed properly and whether the data had any problems.
Return Value	TRUE means the parameters were acceptable and the time and quality have been moved into the database.
Comments	If the quality of the point changes in the Toolkit database, the value, time, and quality will be passed on to any clients who have *advise*d the item. (DDE clients are updated only when the value changes.) When a client requests an item, the current database VTQ will be sent (once the item's value has been set).

DbNewTopicList

BOOL WINAPI

DbNewTopicList(LPSTR lpszTopicList)

Sets the value for the statistic "Topics" item on the SYSTEM topic. The "Topics" item is a tab-separated list of topics available in the I/O Server. The server will normally call this function once at startup, and subsequently only when the configured list of topics available in the server changes.

Parameter	Description
lpszTopicList	Points to a null-terminated character string containing a tab-separated list of topics available in the server's protocol engine. If *lpszTopicList* is a NULL pointer or points to an empty string, the list of protocol-specific topics is considered to be empty.
Return Value	TRUE if the topic list has been set successfully.
Comments	This function call is optional. Note that at a minimum, the "Topics" item will always contain the standard I/O Server developer topics: SYSTEM, $DDE, $PORT, $DEVICE, $SERVER, and $TKITIF. The Toolkit appends this list of topics to the server-specified list of topics.

DbNewTQFromDevice

BOOL WINAPI

DbNewTQFromDevice(IDLDEV *idLogDev*,
HDB *hDb*,
LPPTTIME *lpPtTime*,
PTQUALITY *ptQuality*)

This function is generally used when a problem occurs communicating to the PLC. Then the quality flags and time stamps for all affected points should be set to indicate the problem, without changing the value of the points.

Parameter	Description
idLogDev	Topic (logical device) identifier that was supplied by the Toolkit as a parameter in the **ProtAllocateLogicalDevice**() call.
hDb	Handle from the Toolkit that is unique to this item and was supplied as a parameter in the **ProtCreatePoint**() call.
lpPtTime	Far pointer to a FILETIME structure that indicates when the point information was updated. This can be obtained via a call to **DbGetGMTasFiletime**().
ptQuality	The quality flags, indicating whether the point was accessed properly and whether the data had any problems.
Return Value	TRUE means the parameters were acceptable and the time and quality have been moved into the database.
Comments	If the quality of the point changes in the Toolkit database, the value, time, and quality will be passed on to any clients who have *advise*d the item. (DDE clients are updated only when the value changes.) When a client requests an item, the current database VTQ will be sent (once the item's value has been set).

DbNewValueFromDevice

BOOL WINAPI

DbNewValueFromDevice(IDLDEV *idLogDev*,
HDB *hDb*,
PTVALUE *value*)

Note: With the advent of FactorySuite 2000, this function is outmoded. However, it remains available for backwards compatibility. Internally, the Toolkit provides a default date/time stamp and a default quality of *good*, then calls **DbNewVTQFromDevice**(). See the chapters on Time Marks and Quality Flags for more information on the default time and quality settings.

When you receive a new point value from the device, you can tell the Toolkit's database through a call to **DbNewValueFromDevice**().

Parameter	Description
idLogDev	Topic (logical device) identifier that was supplied by the Toolkit as a parameter in the **ProtAllocateLogicalDevice**() call.
hDb	Handle from the Toolkit that is unique to this item and was supplied as a parameter in the **ProtCreatePoint**() call.
value	A union of type PTVALUE. The user must know the type of data associated with this point (set by the server in **lpPtType* during the **ProtCreatePoint**() call). Based on the point type, use the appropriate field in this structure (it contains fields for discrete, integer, and real, as well as a handle to memory containing a string).
Return Value	TRUE means the parameters were acceptable and the value has been moved into the database.
Comments	This function is supplied for backward compatibility with previous versions of the Toolkit. You should call **DbNewVTQFromDevice**() instead.See the comments on **DbNewVTQFromDevice**() for further information about how the Toolkit updates clients that have points on *advise* or that make *requests*.

DbNewVQFromDevice

BOOL WINAPI

DbNewVQFromDevice(IDLDEV *idLogDev*,
HDB *hDb*,
PTVALUE *ptValue*,
PTQUALITY *ptQuality*)

Functionally, this is similar to **DbNewVTQFromDevice**(), except that the Toolkit supplies a default date/time stamp. See the chapter on Time Marks for more information on the default time.

When you receive a new point value from the device, you can tell the Toolkit's database through a call to **DbNewVQFromDevice**().

Parameter	Description
idLogDev	Topic (logical device) identifier that was supplied by the Toolkit as a parameter in the **ProtAllocateLogicalDevice**() call.
hDb	Handle from the Toolkit that is unique to this item and was supplied as a parameter in the **ProtCreatePoint**() call.
ptValue	A union of type PTVALUE. The user must know the type of data associated with this point (set by the server in **lpPtType* during the **ProtCreatePoint**() call). Based on the point type, use the appropriate field in this structure (it contains fields for discrete, integer, and real, as well as a handle to memory containing a string).
ptQuality	The quality flags, indicating whether the point was accessed properly and whether the data had any problems.
Return Value	TRUE means the parameters were acceptable and the value, time, and quality have been moved into the database.
Comments	To ensure compatibility with future Toolkit releases, this function must be called for every poll. The Toolkit keeps track in its database of the value of the data, the timestamp, and the quality, and identifies when any of these changes. If the value or quality changes, the value, time, and quality will be passed on to any clients who have *advised* the item. (DDE clients are updated only when the value changes.) When a client *requests* an item, the current database VTQ will be sent (once the item's value has been set by a call to this function).

DbNewVTQFromDevice

DbNewVTQFromDevice(IDLDEV *idLogDev*,
HDB *hDb*,
PTVALUE *value*,
LPPTTIME *lpPtTime*,
PTQUALITY *ptQuality*)

When you receive a new value for a point, tell the Toolkit's database through **DbNewVTQFromDevice**(). You should provide a date/time stamp and appropriate quality flags according to when the update occurred and how valid the data is.

Parameter	Description
idLogDev	Topic (logical device) identifier that was supplied by the Toolkit as a parameter in the **ProtAllocateLogicalDevice**() call.
hDb	Handle from the Toolkit that is unique to this item and was supplied as a parameter in the **ProtCreatePoint**() call.
value	A union of type PTVALUE. The user must know the type of data associated with this point (set by the server in **lpPtType* during the **ProtCreatePoint**() call). Based on the point type, use the appropriate field in this structure (it contains fields for discrete, integer, and real, as well as a handle to memory containing a string).
lpPtTime	Far pointer to a FILETIME structure that indicates when the point information was updated.
ptQuality	The quality flags, indicating whether the point was accessed properly and whether the data had any problems.
Return Value	TRUE means the parameters were acceptable and the value, time, and quality have been moved into the database.
Comments	To ensure compatibility with future Toolkit releases, this function must be called for every poll. The Toolkit keeps track in its database of the value of the data, the timestamp, and the quality, and identifies when any of these changes. If the value or quality changes, the value, time, and quality will be passed on to any clients who have *advised* the item. (DDE clients are updated only when the value changes.) When a client *requests* an item, the current database VTQ will be sent (once the item's value has been set by a call to this function).

DbRegisterDemandScan

void WINAPI

DbRegisterDemandScan(DemandScanFncCallback *lpfn*)

Register a callback function for demand scan control.

The server-specific code calls this routine during initialization to provide the address of a function that is implemented in the server-specific code, which will be called if a client writes to the special statistical flag **DemandScan** to force an immediate scan of all points on the corresponding topic. The default value for the function pointer is NULL, meaning that no support is provided for the demand scan functionality.

The parameter type is defined as follows:

typedef BOOL (CALLBACK *DemandScanFncCallback) (HLOGDEV);

Parameter	Description
lpfn	Pointer to a function with a prototype of form BOOL CALLBACK DemandScan(HLOGDEV); which is implemented in the server-specific code.
Return Value	None.
Comments	When a client writes (pokes) a value of 1 to the special statistical flag **DemandScan** on a topic, the Toolkit checks whether a *demand scan callback function* has been registered.If not, no action is performed, and the Toolkit sends a NACK to the client indicating that demand scan was not set up. If a *demand scan callback function* has been registered, theToolkit calls the indicated function, passing the server-specific handle *hLogDev* for the topic.It is the responsibility of the programmer to implement the *demand scan callback function* so that it forces an immediate scan of all points on the indicated topic – even if those points are already scheduled to be polled on a periodic basis. The function should return TRUE if the setup for demand scan is successful (in which the Toolkit will send an ACK to the client), FALSE otherwise (in which case the Toolkit will send a NACK). If a client pokes a value other than 1, the Toolkit will perform no action and will return an ACK to the client.

DbRegisterScanState

void WINAPI

DbRegisterScanState(OnOffScanFncCallback *lpfn*)

Register a callback function for on/off scan control.

The server-specific code calls this routine during initialization to provide the address of a function that is implemented in the server-specific code, which will be called if a client writes to the special statistical flag **OnScan** to put a topic on-scan or take it off-scan. The default value for the function pointer is NULL, meaning that no support is provided for the on/off scan functionality.

The parameter type is defined as follows:

typedef BOOL (CALLBACK *OnOffScanFncCallback) (HLOGDEV, INTG);

Parameter	Description
lpfn	Pointer to a function with a prototype of form BOOL CALLBACK OnOffScan(HLOGDEV, INTG); which is implemented in the server-specific code.
Return Value	None.
Comments	When a client writes (pokes) to the special statistical flag **OnScan** on a topic, the Toolkit checks whether an *on/off scan callback function* has been registered. If not, no action is performed, and the Toolkit sends a NACK to the client indicating that on/off scan was not set up. If an *on/off scan callback function* has been registered, the Toolkit calls the indicated function, passing the server-specific handle *hLogDev* for the topic and a mode selection *nMode*. It is the responsibility of the programmer to implement the *on/off scan callback function* so that puts the topic on-scan or off-scan according to the value of *nMode*. Generally, if *nMode* is non-zero, the topic should be on-scan, i.e. all active points on the topic are being polled. If *nMode* is zero, the topic should be off-scan, i.e. none of the points on the topic should be polled. The function should return TRUE if the setup for on/off scan is successful (in which the Toolkit will send an ACK to the client), FALSE otherwise (in which case the Toolkit will send a NACK).

DbSetHProt

BOOL WINAPI

DbSetHProt(IDLDEV *idLogDev*,
HDB *hDb*,
HPROT *hProtOld*,
HPROT *hProtNew*)

This function allows the server to change the *hProt* value for a point (item identifier from your code to the Toolkit database). Generally, this is most useful if you are changing how your server is keeping track of the memory structures for one or more points (e.g. reducing the size of a symbol table or re-ordering its members).

Parameter	Description
idLogDev	Topic (logical device) identifier that was supplied as a parameter in the **ProtAllocateLogicalDevice**() call.
hDb	Handle from the Toolkit that is unique to this item and was supplied as a parameter in the **ProtCreatePoint** () call.
hProtOld	Handle identifying the point which was the return from the **ProtCreatePoint**() or the previous **DbSetHProt**() call.
hProtNew	The new handle (a non-zero number chosen by you and unique to this item) that identifies this point/item in any future **ProtNewValueForDevice**() , **ProtActivatePoint**(), and **ProtDeactivatePoint**() calls.
Return Value	TRUE means the *hProt* value for this item has been changed.
Comments	The *hProt* value is used during calls to the server to identify the point quickly (e.g., **ProtNewValueForDevice**() or **ProtActivatePoint**()). This may be useful when managing the data structures used to keep track of points in the server (e.g., *hProtNew* can be the index entries into a new symbol table).

DbSetPointType

BOOL WINAPI

DbSetPointType(IDLDEV *idLogDev*,
HDB *hDb*,
PTTYPE *ptType*)

Deferred setting of point type for indicated point on indicated logical device.

Returns TRUE if successful.

Parameter	Description
idLogDev	Topic (logical device) identifier that was supplied by the Toolkit as a parameter in the **ProtAllocateLogicalDevice**() call.
hDb	Handle from the Toolkit that is unique to this item and was supplied as a parameter in the **ProtCreatePoint**() call.
ptType	One of the standard data types: PTT_DISCRETE, PTT_INTEGER, PTT_REAL, PTT_STRING.
Return Value	TRUE if the data type for the point was successfully updated.

DbValueWriteConfirm

BOOL WINAPI

DbValueWriteConfirm(IDLDEV *idLogDev*,
HDB *hDb*,
BOOL *bSuccess*,
BYTE *bReason*)

When a client writes a value to the device (via a DDE POKE or a SuiteLink Write), the value is passed to the server-specific code via **ProtNewValueForDevice**(). The server-specific code then performs whatever actions are necessary to write that data to the PLC. However, depending upon the communication protocol, this update may be immediate, or there may be a considerable delay before the new data actually reaches the PLC. (For some devices, e.g. those involving radio modems, this delay may be as long as several minutes!) Sometimes, the only way to verify that the data actually arrived is to subsequently poll the PLC for the point – or to poll other points that are affected by a write-only point. More commonly, the operation used to write the value causes the PLC to respond with some indicator of whether the new data was accepted or rejected; and if rejected, the response usually includes a *reason code* indicating why it was rejected. When such an accept/reject response is received, you can use **DbValueWriteConfirm**() to tell the Toolkit whether the point was written successfully.

Note: This API call is at present non-functional. It is reserved for future expansion.

Parameter	Description
idLogDev	Topic (logical device) identifier that was supplied as a parameter in the **ProtAllocateLogicalDevice**() call.
hDb	Handle from the Toolkit that is unique to this item and was supplied as a parameter in the **ProtCreatePoint**() call.
bSuccess	TRUE if the point was successfully written, FALSE otherwise.
bReason	The reason code, in case the write failed. This can be a generic code, or can be specific to the PLC protocol.
Return Value	TRUE means the notification to the Toolkit was successful, i.e. that the parameters were valid.
Comments	This API call is presently non-functional. It is reserved for future expansion.

debug

VOID far cdecl

debug(LPSTR *lpszFmt*,
... *args*)

This function sends a debug message to WWLOGGER.EXE which can log it to disk, monochrome adapter, AUX port, etc.

Parameter	Description
lpszFmt	Format text string, see printf in C-Library.
args	Any set of arguments applicable to printf.
Return Value	None.
Comments	The interface in the code is very straightforward. Simply include *debug.h* and use **debug**() with parameters identical to the C-Library **printf**(). Notice the WWLOGGER.EXE options; the debug messages to be saved on disk, displayed in the Wonderware Logger window, and/or sent to the AUX port. Since it is recommended that the Wonderware Logger always be running, you should not leave extraneous debug messages in the server. You should also make sure that frequently occurring debug messages can be turned off via a switch in the WIN.INI file.

Warning Note The **ProtGetDriverName**() function may **not** contain any calls to **debug**() because **debug**() calls **ProtGetDriverName**() and an infinite loop will result.

The **debug**() function is often useful to have some debugging messages while developing the server. In the sample application, an interface to the Wonderware Logger WWLOGGER.EXE is provided. This interface allows you to optionally log each debug message to disk and also send each message to either:

1. A window on the screen.
2. The AUX port (COM1 by default, or the monochrome display if OX.SYS is installed).
3. A printer connected directly to the PC.

Wonderware Logger can be used during development to help debug the server. Hardware or software problems should be logged for operator examination. Be advised that extraneous logger messages frustrate users and should be eliminated when the product is finished. It is useful to enable debug messages via a switch in the server section of the WIN.INI in order to support customers at a later time.

EnableCommNotification

BOOL WINAPI

EnableCommNotification(int *idComDev*,
HWND *hwnd*,
int *cbWriteNotify*,
int *cbOutQueue*)

This function simulates the Windows EnableCommNotification() function for Windows versions older than Windows 3.1 and for Windows NT/2000. It performs no operation on these platforms and is provided for common code convenience only. On Windows versions prior to 3.0, it enables or disables WM_COMMNOTIFY message posting to the given window.

Parameter	Description
idComDev	The *id* of the communications device. The **OpenComm**() function returns this value.
hwnd	Identifies the window whose WM_COMMNOTIFY message posting will be enabled or disabled.
cbWriteNotify	Indicates the number of bytes the COM driver must write to the application's input queue before sending a notification message.
cbOutQueue	Indicates the minimum number of bytes in the output queue. When the number of bytes falls below this number, the COM driver sends that application a notification message.
Return Value	The return value is non-zero if the function is successful. Otherwise, it is zero.
Comments	None.

FlushComm

int WINAPI

FlushComm(int *idComDev*,
int *fnQueue*)

This function flushes all characters from the transmission or receiving queue of the specified communications device.

Parameter	Description
idComDev	The *id* of the communications device to be flushed. The **OpenComm**() function returns this value.
fnQueue	Specifies the queue to be flushed. If this parameter is zero, the transmission queue is flushed. If the parameter is 1, the receiving queue is flushed.
Return Value	The return value is zero if the function is successful. Any other value indicates an error.
Comments	When flushing the transmission queue, beware that this function will clear all characters in it. Thus, this function can conceivably terminate a previous write operation before all submitted characters have been sent from the UART. This may lead to unpredictable behaviors in your server since the receiving device will not receive the entire message intended. Windows NT /2000 only. Emulates Windows function.

GetAppName

PSTR WINAPI

GetAppName(VOID)

This function returns the application name for the server. This application name is stored in the Toolkit and initially set by a call to **ProtGetDriverName**() during server startup.

Parameter	Description
Return Value	Pointer to a character string containing the application name for the server.
Comments	None.

GetCommError

int WINAPI

GetCommError(int *idComDev*,
COMSTAT * *lpStat*)

This function retrieves the most recent error value and current status for the specified device. It also clears the error.

Parameter	Description
idComDev	The *id* of the communications device to be examined. The **OpenComm**() function returns this value.
lpStat	Points to the COMSTAT structure that is to receive the device status. If this parameter is NULL, this function returns only the error values. See the appropriate Microsoft documentation for a description of this structure.
Return Value	The return value is a 32-bit value containing a mask indicating the type of error. See discussion of the ClearCommError() function in the appropriate Microsoft Windows NT/2000 documentation for more information.
Comments	This function clears the error condition. Windows NT/2000only. Emulates Windows function.

GetCommEventMask

UINT WINAPI

GetCommEventMask(int *idComDev*,
int *fnEvtClear*)

This function retrieves and then clears the event word for the specified communications device.

Parameter	Description
idComDev	The *id* of the communications device to be examined. The **OpenComm**() function returns this value.
fnEvtClear	This parameter is ignored on Windows NT/2000.
Return Value	The return value specifies the current 32-bit event mask value for the specified communications device if the function is successful. Each bit in the event mask specifies whether a given event has occurred; a bit is set (to 1) if the event has occurred.
Comments	Windows NT/2000 only. Emulates Windows function.

GetIOServerLicense

BOOL WINAPI

GetIOServerLicense(void FAR **lpKeyInfo*,
BOOL *bExitOnFail)*

This function should be called in **ProtInit**() to check whether the user has a license that will allow the server to run. If a license is unavailable, this function presents a MessageBox that gives the user the options of trying again, running the server in ***timed demo*** mode, or exiting the program. If the function returns FALSE, the server should return FALSE from **ProtInit**(), indicating the server could not run.

Note If you are developing products for Wonderware, this function will be of interest. Note that a special version of the Toolkit is necessary to access the Wonderware License Manager. Contact Wonderware for more information.

Parameter	Description
lpKeyInfo	A pointer to a buffer into which feature enablement flags will be placed. If this parameter is NULL, no specific feature flags will be returned.
bExitOnFail	If TRUE, the function will exit the program if it is unable to obtain a license. If FALSE, it will return, and the return value will indicate whether the server should run.
Return Value	TRUE if the server has permission to run. This includes the case where no license is available, but user has selected running in the ***timed demo*** mode. If FALSE, The server should return from **ProtInit**() with a return value of FALSE, indicating the server cannot run.
Comments	For the ***timed demo*** mode, the Toolkit keeps track of how long it has been since the server started up. When the time limit is reached, the Toolkit automatically issues a message to the **Wonderware Logger** and shuts down the server.

The following code would be placed in **ProtInit**(). If a valid license could not be obtained, and the timed DEMO mode was not selected, the server should exit.

Example

```
/* check whether this product has a license to run */
if (!GetIOServerLicense (NULL, FALSE))
    return FALSE;}
```

GetServerNameExtension

void WINAPI

GetServerNameExtension(void)

Set-up name extension for use in storing and retrieving Registry settings

Parameter	Description
Return Value	None.
Comments	A call to **GetServerNameExtension**() should appear in **ProtGetDriverName**(). This ensures that the function is called early in the program, at the same time as the Toolkit first gets the server's "short" name, so that the correct full name is generated for the I/O Server.

Example

```
BOOL WINAPI ProtGetDriverName(LPSTR lpszName, int nMaxLength)
{
    /**  WARNING: No calls to debug()...
    debug() calls ProtGetDriverName(),
    therefore ProtGetDriverName() cannot call debug(). **/
    lstrcpy(lpszName, GetString(STRUSER + 76) /* "UDSAMPLE" */ );
#ifdef WIN32
    GetServerNameExtension();
#endif
    return (TRUE);
} /* ProtGetDriverName */
```

GetString

PSTR WINAPI

GetString(int *nString*)

This function will return a string from the resource file.

Parameter	Description
nString	The offset into the STRINGTABLE.
Return Value	Pointer to the string.
Comments	This function is used to reduce the amount of memory that the server consumes and to allow the creation of a foreign language version of the server. The resource file can be edited later to change text or languages, this capability is also know as localization. **GetString**() accomplishes this by using Windows' string resources to move string constants from the resource file to the data segment. A sample follows: MessageBox(GetFocus(), GetString(STRUSER+1) /* "Hello There" */, GetString(STRUSER+2) /* "Application" */, MB_OK); STRUSER is defined in udgetstr.h and all strings that you use can go in udprot.str. The following is an example of the string file that would correspond to the above: STRUSER+1, "Hello There" STRUSER+2, "Application" **GetString**() loads the string named as its argument in a temporary buffer and returns a pointer to the temporary buffer. This is much easier than the standard method used in Windows applications, which is to load the string into a buffer and then use it. Each string that is not put in the resource file consumes memory byte-for-byte in the application's data segment (DS). By using resource strings, code size can be slightly increased. Code segments can be swapped - data segments cannot! **GetString**() also uses 20 string buffers circularly, so only the most recent 20 strings are valid at any instant.

Note *hInst* must be initialized before making the first call to **GetString**(). Each string in the STRINGTABLE is limited to 200 bytes, including the NULL.

GetTextExtent

DWORD

GetTextExtent(HDC *hdc*,
LPCSTR *lpString*,
int *cbString*)

This function provides emulation of the Windows **GetTextExtent**() function for the Windows NT/2000 platform.

Parameter	Description
hdc	Identifies the device context.
lpString	Points to a string of text. The string does not need to be zero-terminated.
cbString	Specifies number of characters in the string.
Return Value	The low-order word of the return value contains the string width, in logical units, if the function is successful; the high-order word contains the string height.
Comments	Windows NT/2000 only. Emulates Windows function.

NTSrvr_BuildCommDCB

int WINAPI

NTSrvr_BuildCommDCB(LPSTR *lpszDef*,
DCB FAR **lpdcb*)

This function translates a device-definition string into appropriate serial device control block codes.

The return value from this function is compatible with the Windows version of the BuildCommDCB() function.

Parameter	Description
lpszDef	Points to a null-terminated string that specifies device-control information. The string must have the same form as the parameters used in the MS-DOS or Windows NT mode command.
lpdcb	Points to a DCB structure that will receive the translated string. The structure defines the control settings for the serial-communications device.
Return Value	The return value is zero if the function is successful. Otherwise, it is -1.
Comments	This function only fills the *lpdcb* buffer. To apply the settings to a port, the server should use the **NTSrvr_SetCommState**() function.

NTSrvr_GetCommState

int WINAPI

NTSrvr_GetCommState(int *idComDev*,
DCB FAR **lpdcb*)

This function retrieves the device control block for the specified device. The return value from this function is compatible with the Windows version of the GetCommState() function.

Parameter	Description
idComDev	The *id* of the communications device to be examined. The **OpenComm**() function returns this value.
lpdcb	Points to a DCB structure that is to receive the current device control block. The DCB structure defines the control settings for the device.
Return Value	The return value is zero if the function is successful. Otherwise, it is less than zero.
Comments	None.

NTSrvr_SetCommState

int WINAPI

NTSrvr_SetCommState(int *idComDev*,
DCB FAR **lpdcb*)

This function sets a communications device to the state specified by a device control block. The return value from this function is compatible with the Windows version of the BuildCommDCB() function.

Parameter	Description
idComDev	The *id* of the communications device to be set. The **OpenComm**() function returns this value.
lpdcb	Points to a DCB structure that contains the desired communications settings for the device.
Return Value	The return value is zero if the function is successful. Otherwise, it is less than zero.
Comments	This function reinitializes all hardware and controls as defined by the DCB structure. It does not empty transmission or receiving queues.

NTSrvr_SetDCB_Dtr

int WINAPI

NTSrvr_SetDCB_Dtr(DCB FAR *_lpdcb_,
int _iMask_)

This function modifies the DTR (data-terminal-ready) flow-control setting in the device control block.

Parameter	Description
lpdcb	Points to a DCB structure which is to be modified using the specified iMask setting.
iMask	Specifies the DTR flow control setting. Must be one of the following: NTSrvr_DTR_DISABLE, NTSrvr_DTR_ENABLE, NTSrvr_DTR_HANDSHAKE.
Return Value	The return value is zero if the function is successful. Otherwise, it is less than zero.
Comments	NTSrvr_DTR_DISABLE disables the DTR line when the device is opened and leaves it disabled. NTSrvr_DTR_ENABLE enables the DTR line when the device is opened and leaves it enabled. NTSrvr_DTR_HANDSHAKE enables DTR handshaking. Include header file: NTSRVR.H for bit mask definitions.

NTSrvr_SetDCB_Rts

int WINAPI

NTSrvr_SetDCB_Rts(DCB FAR **lpdcb*,
int *iMask*)

This function modifies the RTS (request-to-send) flow-control setting in the device control block.

Parameter	Description
lpdcb	Points to a DCB structure which is to be modified using the specified iMask setting.
iMask	Specifies the RTS flow control setting. Must be one of the following: NTSrvr_RTS_DISABLE, NTSrvr_RTS_ENABLE, NTSrvr_RTS_HANDSHAKE.
Return Value	The return value is zero if the function is successful. Otherwise, it is less than zero.
Comments	NTSrvr_RTS_DISABLE disables the RTS line when the device is opened and leaves it disabled. NTSrvr_RTS_ENABLE enables the RTS line when the device is opened and leaves it enabled. NTSrvr_RTS_HANDSHAKE enables RTS handshaking. Include header file: NTSRVR.H for bit mask definitions.

OpenComm

int WINAPI

OpenComm(LPCSTR *lpszDevControl*,
UINT *cbInQueue*,
UINT *cbOutQueue*)

This function opens a serial communications port for communications.

Parameter	Description
lpszDevControl	Points to a null-terminated string that specifies the device name in the form COMn, where n is the device number.
cbInQueue	Specifies the size, in bytes, of the receiving queue.
cbOutQueue	Specifies the size, in bytes, of the transmission queue.
Return Value	The return value identifies the open device if the function is successful. Otherwise, it is less than zero.
Comments	The return value is actually a file handle when positive. Windows NT/2000 only. Emulates Windows function.

PfnSendEmSelectAll

void WINAPI

PfnSendEmSelectAll(HWND *hDlg*,
int *idControl*,
BOOL *bScrollCaret*)

This function selects all the text in the identified edit control. It will also scroll the caret into view if the *bScrollCaret* flag is TRUE.

Parameter	**Description**
hDlg	Handle of dialog box window.
idControl	Identifier of control to be selected.
bScrollCaret	Scroll caret flag. TRUE indicates scroll caret into view.
Return Value	None.
Comments	This function calls SendMessage() internally.

PfnSendEmSelectRange

void WINAPI

PfnSendEmSelectRange(HWND *hDlg*,
int *idControl*,
int *nStart*,
int *nEnd*,
BOOL *bScrollCaret*)

This function selects a range of text in the identified edit control. It will also scroll the caret into view if the *bScrollCaret* flag is TRUE.

Parameter	Description
hDlg	Handle of dialog box window.
idControl	Identifier of control to be selected.
nStart	Starting text position.
nEnd	Ending text position.
bScrollCaret	Scroll caret flag. TRUE indicates scroll caret into view.
Return Value	None.
Comments	This function calls SendMessage() internally.

ProtActivatePoint

BOOL WINAPI

ProtActivatePoint(HLOGDEV *hLogDev*,
HPROT *hProt*)

This function activates the specified point for reading (i.e., start supplying the Toolkit database with fresh data for this point). The *hProt* is the handle you returned during the **ProtCreatePoint**() call. As soon as you have valid data from the device, call **DbNewVTQFromDevice**().

Parameter	Description
hLogDev	Handle identifying the logical device which was returned from the **ProtAllocateLogicalDevice**() call.
hProt	Handle identifying the point which was the return from the **ProtCreatePoint**() call.
Return Value	TRUE means a valid point has been activated.
Comments	A point will be activated when a client has requested data or has asked to be advised of data changes. The server is then responsible for providing fresh data for this point to the Toolkit.

ProtAllocateLogicalDevice

HLOGDEV WINAPI

ProtAllocateLogicalDevice(LPSTR *lpszTopicName*,
IDLDEV *idLogDev*)

This function is called when a DDE conversation has been initiated to the topic name (i.e., logical device name) *lpszTopicName*. You must first validate the topic name. If the name is valid, return a non-NULL value. If the name is invalid, return NULL.

Parameter	Description
lpszTopicName	Far pointer to string that came from the client initiating a new topic conversation.
idLogDev	This is the Toolkit's handle to this logical device. This parameter must be saved for subsequent calls to **DbNewVTQFromDevice**() to identify the source device (topic).
Return Value	Handle (a non-zero number chosen which is unique to this topic) that identifies this logical device (topic) in future calls to **ProtCreatePoint**(), etc. NULL means no device was allocated and the topic conversation is rejected (i.e., the conversation is not established).
Comments	Usually a topic name is associated with a single I/O device. Be sure to save *idLogDev* (the Toolkit's handle for this topic) for subsequent calls to **DbNewVTQFromDevice**(). Return NULL only if the topic name is illegal or you are out of memory - **not** if communication with the logical device fails. After one DDE client establishes a conversation to the server with a specific topic name, subsequent conversations to the same topic will NOT cause a call to **ProtAllocateLogicalDevice**(), i.e., this function will only be called when the first conversation to a topic is established.

Note The name is a character string that may contain spaces and have mixed case (e.g., topic-Name). It is suggested that the lstrcmpi() function be used to compare the full string and ignore the case. Do not call Windows MessageBox() during this function.

ProtClose

VOID WINAPI

ProtClose(void)

This function is called when the server is shut down.

Parameter	Description
Return Value	None.
Comments	At this point, it is intended for the server to free all logical devices, free any allocated memory, close serial ports, etc. **ProtClose** () is only called immediately prior to the server closing.

ProtCreatePoint

HPROT WINAPI

ProtCreatePoint(HLOGDEV *hLogDev*,
HDB *hDb*,
LPSTR *lpszName*,
LPPTTYP *lpPtType*)

This function is called when a conversation references an item (named **lpszName*). You must validate the name and determine point type (discrete, integer, real or string). Return NULL if the point name is illegal or out of memory, etc. Otherwise, store the point type in **lpPtType*. Remember the *hDb* for this point, and return a handle (a non-zero number chosen by you which is unique to this item) that subsequently will be used to identify the point.

Parameter	Description
hLogDev	Handle identifying the logical device (generated by the server code and returned from the call to **ProtAllocateLogicalDevice**()). This ties the point/item to a particular logical device (topic).
hDb	Handle from the Toolkit which is unique to this item and must be used in future calls to **DbNewVTQFromDevice**().
lpszName	Far pointer to a string which contains the item name as it came from the client.
lpPtType	Return the point type by storing one of the following in **lpPtType*:

Value	Meaning
PTT_DISCRETE	(0 or 1)
PTT_INTEGER	(Signed 32-bit integer)
PTT_REAL	(IEEE 32-bit floating point, single precision)
PTT_STRING	(Text string terminated by a NULL character)

Return Value Handle (a non-zero number chosen by you and unique to this item) that identifies this point/item in any future **ProtNewValueForDevice**() , **ProtActivatePoint**() and **ProtDeactivatePoint**() calls.

Comments The name is a character string that may contain spaces and have mixed cases (e.g., "This Is An Item Name"). The lstrcmpi() function should be used to compare the full string and ignore the case. The HPROT value returned can be any non-zero number. Therefore, choose one that is useful to your protocol (e.g., an index into a symbol table). Based upon the item name, the data type must be set in **lpPtType*. Save the *hDb* for this item. It must be used later when calling **DbNewVTQFromDevice**().

ProtDeactivatePoint

BOOL WINAPI

ProtDeactivatePoint(HLOGDEV *hLogDev*,
HPROT *hProt*)

This function deactivates (i.e., stops supplying data for) the specified point. No DDE conversations are interested in the value of this point. Do **not** delete the symbol for the point though - **ProtDeletePoint**() will be called if the symbol should actually be deleted from your internal symbol tables.

Parameter	Description
hLogDev	Handle identifying the logical device which was the return value from the **ProtAllocateLogicalDevice**() call.
hProt	Handle identifying the point which was the return from the **ProtCreatePoint**() call.
Return Value	TRUE means the point is deactivated but still valid.
	FALSE means error, invalid logical unit, symbol table, point, etc. Return value is not used at this time, but should be observed for future compatibility.
Comments	When this function is called, stop polling the device for this point and stop sending data for this point to the Toolkit database. The point still has valid identifiers do not delete it completely. This function should undo anything that was done as a result of a previous **ProtActivatePoint**() call. For example, it may need to delete a poll message that was created in **ProtActivatePoint**().
	If a point is only *poke*d (a one-time write to the point), the point will be created with **ProtCreatePoint**(), changed with **ProtNewValueForDevice**(). If a point is *advise*d (a request for notification whenever the point changes), the point will be created with **ProtCreatePoint**(), activated with **ProtActivatePoint**() (in response to the *advise*) and changed with **ProtNewValueForDevice**() (in response to *poke*s). Eventually the point will be deactivated with **ProtDeactivatePoint**() (in response to an *unadvise*). **ProtActivatePoint**() and **ProtDeactivatePoint**() can be called many times between the **ProtCreatePoint**() and **ProtDeletePoint**() calls.

ProtDefWindowProc

LRESULT CALLBACK

ProtDefWindowProc(HWND *hWnd*,
VINT *message*,
WPARAM *wParam*,
LPARAM *lParam*)

This function is called by the Toolkit to handle window messages directed to this application.

Parameter	Description
Return Value	If no processing was done, use the following code return (*hWnd*, *message*, *wParam*, *lParam*);. If processing was done, the return is message-specific. Consult the Windows SDK manuals.
Comments	This function is only of interest if you wish to do some specialized Windows programming. If not, leave the code in **ProtDefWindowProc** () the same as delivered in the sample application. Otherwise, the I/O Server will not work.

Note When doing Windows programming, any message can be processed in **ProtDefWindowProc**(). If adding your own user interface for the server, menu items can be added to ProtMenu in the .RC file and WM_COMMAND messages can be intercepted in **ProtDefWindowProc**(), etc. Examples of this are shown in the sample servers.

ProtDeletePoint

BOOL WINAPI

ProtDeletePoint(HLOGDEV *hLogDev*,
HPROT *hProt*)

This function will delete the specified point. No DDE conversations are interested in this point's value. However, if a write is outstanding - still perform the write.

Parameter	Description
hLogDev	Handle identifying the logical device which was the return from the **ProtAllocateLogicalDevice**() call.
hProt	Handle identifying the point which was the return from the **ProtCreatePoint**() call.
Return Value	TRUE means point has been deleted.
	FALSE means error, invalid logical unit, symbol table, point, etc. Return value is not used at this time, but should be observed for future compatibility.
Comments	After this call, the *hProt* value will no longer be used (it can then be reused by the server if so desired).

ProtExecute

BOOL WINAPI

ProtExecute(HLOGDEV *hLogDev*,
LPSTR *lpszName*)

This function is called when the client sends an Execute DDE message to a conversation on this server. The purpose is to communicate control commands to the I/O Server. This function is optional since the Toolkit supplies a default **ProtExecute**()

Parameter	Description
hLogDev	Handle which identifies this logical device (topic) as returned by **ProtAllocateLogicalDevice**() call.
lpszName	Far pointer to the command string.
Return Value	TRUE means the command was executed. FALSE means the server had a problem with the command.
Comments	This function should execute the string supplied by the client and return success or failure based on the command supplied.

Note Please use the execute format defined in the Microsoft Windows SDK documentation.

Example

```
[opcodestring] {[opcodestring]} ...

where opcodestring is

        opcode { ( parameter {, parameter } ... ) }

For example:

    [connect][download(query1,results.txt)][disconnect]
```

ProtFreeLogicalDevice

BOOL WINAPI

ProtFreeLogicalDevice(HLOGDEV *hLogDev*)

This function is called when there are no remaining DDE conversations on this logical device identified by *hLogDev*. Delete all information associated with it.

Parameter	Description
hLogDev	Handle which identified this logical device (topic) as returned by the **ProtAllocateLogicalDevice**() call earlier.
Return Value	TRUE means valid *hLogDev* device was freed.
Comments	Each individual point may not be deactivated or deleted when you receive this message. Be prepared to delete the logical device and ALL associated information (e.g., point data structures, messages, symbol table, etc.). This function will **only** be called when the last conversation to a topic is terminated.

ProtGetDriverName

BOOL WINAPI

ProtGetDriverName(LPSTR *lpszName*,
int *nMaxLength*)

This function will supply the driver name to the Toolkit to be used during the initiation of DDE conversations.

Parameter	Description
lpszName	Far pointer to the server name stored as a null terminated character string no longer than *nMaxLength*.
nMaxLength	Maximum size for the server name string including the NULL terminator character.
Return Value	TRUE means the name stored.
Comments	Place the name of the server in *lpszName*. *nMaxLength* will be 9 (8 chars & NULL), since the server name must be convertible to a .EXE filename. Unless you have some reason not to, this name should be the same as the .EXE name.

Note The **ProtGetDriverName**() function **cannot** contain any calls to **debug**() because **debug**() calls **ProtGetDriverName**() and an infinite loop will result.

ProtGetValidDataTimeout

DWORD WINAPI

ProtGetValidDataTimeout(void)

This function supplies the timeout value (in milliseconds) for the Toolkit when it is waiting for first time data supplied by the protocol.

Parameter	Description
Return Value	Return (in milliseconds) how long the Toolkit should wait for the protocol to supply data for the first time.
Comments	When a WM_DDE_REQUEST from a client comes in for a point, and the server has not received any values for that point; the Toolkit will wait some amount of time for the server to set a new value via **DbNewVTQFromDevice**(). If it takes longer than that amount of time to get a value from the device, the Toolkit will *Nack* the item WM_DDE_REQUEST from the client.

Note NetDDE Considerations The value returned for the valid data timeout must be 1 msec if the server is supplying data through **NetDDE**. This is a special case where the Toolkit *Nacks* requests if there isn't valid data immediately available. Generally, this timeout value is supplied in the configuration process of the server. The user must be informed to set this for **NetDDE**.

ProtInit

BOOL WINAPI

ProtInit(void)

This function is called to perform any necessary initialization. It is the first function called within the server.

Parameter	Description
Return Value	TRUE means that the server is initialized and ready to start DDE conversations (i.e., all basic requirements for this server have been met). FALSE means the server cannot continue and will cause it to exit.
Comments	This function can perform any overall initialization needed by the server, read configuration files, obtain information from the WIN.INI file (through calls to GetProfileInt() and GetProfileString()), etc. Specifically, this function should call **SysTimerSetupProtTimer**(), **SysTimerSetupRequestTimer**(), and **AdjustWindowSizeFromWinIni**().

ProtNewValueForDevice

BOOL WINAPI

ProtNewValueForDevice(HLOGDEV *hLogDev*,
HPROT *hProt*,
PTVALUE *value*)

ProtNewValueForDevice() will be called whenever a new value for the point is received from DDE.

Parameter	Description
hLogDev	Handle identifying the logical device which was the return from the **ProtAllocateLogicalDevice**() call.
hProt	Handle identifying the point which was the return from the **ProtCreatePoint**() call.
value	The user must know the type of data associated with this point (set by the server in **lpPtType* during the **ProtCreatePoint**() call). Based on the point type, use the appropriate field in this structure (it contains fields for discrete, integer, and real, as well as a handle to memory containing a string).
Return Value	TRUE means the point is still valid and it is allowed to send data to this point. FALSE indicates the point cannot be written.
Comments	The server should respond as soon as possible - i.e., schedule a message to write the new value to the device and return immediately (if applicable).

ProtTimerEvent

VOID WINAPI

ProtTimerEvent(DWORD *dwTime*)

This function will be called at the interval set by the last call to **SysTimerSetupProtTimer**().

Parameter	Description
dwTime	Time in milliseconds that have passed since the last call to this function.
Return Value	None.
Comments	Through this function's periodic execution, you can drive the device protocol and supply data to the Toolkit database. This is accomplished by making a call to **DbNewVTQFromDevice**() with each of the data items available in this time cycle. The *dwTime* is provided to the protocol in case it is needed for timing related activities.

Note Due to the nature of Windows and loading of the PC with other applications, this function may not be called on at a precisely regular interval.

ReadComm

int WINAPI

ReadComm(int *idComDev*,
void **lpvBuf*,
int *cbRead*)

This function reads up to a specified number of bytes from the given communications device.

Parameter	Description
idComDev	The *id* of the communications device to be read. The **OpenComm**() function returns this value.
lpvBuf	Points to the buffer for the read bytes.
cbRead	Specifies the maximum number of bytes to be read.
Return Value	The return value is the number of bytes read if successful. Otherwise, it is less than zero and its absolute value is the number of bytes read.
Comments	Windows only. Emulates Windows function. When an error occurs, the cause of the error can be determined by using the **GetCommError**() function to retrieve the error value and status. Since errors can occur when no bytes are present, if the return value is zero the **GetCommError**() function should be called to ensure that no error occurred. The return value is less than the number of bytes specified by *cbRead* if the number of bytes in the receiving queue is less than *cbRead*.

RelinquishPermission - Windows Only

VOID WINAPI

RelinquishPermission(HANDLE *hPermission*)

RelinquishPermission() will release the selector that was allocated to this memory address range, identified by *hPermission*.

Parameter	**Description**
hPermission	The permission handle that was set when permission is granted by **RequestPermission**().
Return Value	None.
Comments	None.

RequestPermission - Windows Only

BOOL WINAPI

RequestPermission(WORD *wSegment*,
WORD *wOffset*,
DWORD *dwLength*,
LPSTR FAR **lplpProt*,
LPHANDLE *lphPermission*)

RequestPermission() is used to access memory at a fixed real mode location.

Parameter	**Description**
wSegment	Real-mode address segment.
wOffset	Real-mode address offset.
dwLength	Length of the area for permission.
lplpProt	Far pointer to the protected-mode far pointer. The Protected-mode pointer is stored after the permission process is completed. It may then be used to access the special fixed memory block.
lphPermission	Far pointer to a permission handle that is set when permission is granted. This handle must be saved and used when calling **RelinquishPermission**().
Return Value	TRUE means permission was granted. FALSE means there was a problem and no permission granted.
Comments	Many plug-in I/O boards use memory mapped I/O to transfer data and commands which will then be relayed to an external device (e.g., the MODBUS Plus or AB 1784-KT boards). In order to access non-standard memory from Windows while in protected-mode, permission must be granted and a far pointer set up. The permission handle must be saved and used later to call **RelinquishPermission**(). A FALSE return indicates an error condition in the request. After permission is granted, that portion of memory may be freely accessed using the *lpProt* pointer. When the protocol is shut down, all requested blocks of memory must be relinquished by the user.

Example

```
LPSTR     lpProt;
HANDLE    hPermission;
if( RequestPermission( 0x0000, 0x40, (DWORD)0x0008,
          &lpProt, &hPermission) ) {
    /* lpProt can be used to access 0:40 through 0:47. */
    RelinquishPermission( hPermission );
}
```

SelBoxAddEntry

BOOL WINAPI

SelBoxAddEntry(LPSTR *string*,
LONG *value*,
WORD *wFlags*)

This function adds a string entry to a selection box set of choices to be displayed when the **SelBoxUserSelect**() call is done.

Parameter	Description
string	Far pointer to the string to be added to the selection box entries.
value	If this entry is picked by the user, this value will be returned as an indicator.
wFlags	Control flags allowed: SBENTRY_DISABLED SBENTRY_SELECTED
Return Value	FALSE means there was an out of memory error adding the entry.
Comments	None.

SelBoxSetupStart

VOID WINAPI

SelBoxSetupStart(HWND *hWnd*,
PSTR *title*,
PSTR *noEntryMsg*,
int *numCols*,
LONG *lFlags*)

SelBoxSetupStart() does the basic setup for a selection box.

Parameter	Description
hWnd	In most cases use the extern *hWndParent*.
title	Pointer to the text to be displayed in the caption.
noEntryMsg	Pointer to the text to be displayed in the window when there are no entries.
numCols	Number of columns to be used. **0** will allow the Toolkit to pick the optimum based on the text lengths.
lFlags	Control flags can be one of the following: SBSTYLE_RADIO_BUTTONS (default is check-box style) SBSTYLE_RETURN_ON_SELECTION (default is wait for OK) SBSTYLE_SORT_ENTRIES (default is put in order of **SelBoxAddEntry**() calls) SBSTYLE_ENTRIES_PRESORTED (put in order but knows that they are sorted)
Return Value	None.
Comments	None.

SelBoxUserSelect

LONG WINAPI

SelBoxUserSelect(HANDLE *hInst*,
LONG *lButtons*,
int *vPosition*,
int *nFixed*)

This function actually displays the selection box and processes all the buttons.

Parameter	Description
hInst	Always use the extern *hInst* for this application.
lButtons	Specifies which of the following buttons should be enabled (i.e., visible): SB_BUTTON_NEW SB_BUTTON_MODIFY SB_BUTTON_DELETE SB_BUTTON_CANCEL SB_BUTTON_OK
vPosition	Sets the vertical screen position of the box.
nFixed	Set to **0** to allow free-format lengths. By setting *nFixed* to non-zero, the sizes for the entries are fixed at the length specified by *nFixed*.
Return Value	One of the buttons was picked by the user: SB_BUTTON_OK SB_BUTTON_CANCEL SB_BUTTON_NEW SB_BUTTON_MODIFY SB_BUTTON_DELETE
Comments	When this function returns, the selection box is no longer on the screen. The user selections can now be processed based on which button was selected, see the **SelBoxUserSelection**() function below.

SelBoxUserSelection

HANDLE WINAPI

SelBoxUserSelection(void)

This function is called after **SelBoxUserSelect**() if you wish to get a list of what the user selected.

Parameter	Description
Return Value	The return is the handle of a selection list *hSelList* that represents what the user selected during the last call to **SelBoxUserSelect**().
Comments	**SelBoxUserSelection**() is called after **SelBoxUserSelect**() to see what the user selected. Use the following functions to access this list: **SelListNumSelections**(), **SelListGetSelection**(), and **SelListFree**(). (Refer to the sample server code for examples of the selection box usage).

SelListFree

VOID WINAPI

SelListFree(HANDLE *hSelList*)

This function will free the memory associated with the selection list.

Parameter	Description
hSelList	The selection list handle obtained by calling **SelBoxUserSelection**().
Return Value	None.
Comments	None.

SelListGetSelection

LONG WINAPI

SelListGetSelection(HANDLE *hSelList*,
int *nSelection*)

This function gets the indicator value for a selection at the specified number.

Parameter	Description
hSelList	The selection list handle obtained by calling **SelBoxUserSelection**().
nSelection	The selection to be examined, the entries start at entry **0** and increase to **SelListNumSelections**(*hSelList*) **-1**.
Return Value	The value set for this entry as an indicator by the **SelBoxAddEntry**() call.
Comments	None.

SelListNumSelections

int WINAPI

SelListNumSelections(HANDLE *hSelList*)

This function returns how many entries are in the specified *hSelList*.

Parameter	Description
hSelList	The selection list handle obtained by calling **SelBoxUserSelection**().
Return Value	Count of selected entries during the **SelBoxUserSelect**() call.
Comments	None.

SetCommEventMask

UINT FAR *WINAPI

SetCommEventMask(int *nCid*,
UINT *fnEvt*)

The **SetCommEventMask**() function does no operation on Windows NT/2000. It is provided for common code convenience only.

Parameter	**Description**
nCid	This parameter is ignored on Windows NT/2000.
fnEvt	This parameter is ignored on Windows NT/2000.
Return Value	NULL pointer.
Comments	Windows NT/2000 only but provides no operation. Emulates Windows function.

SetSplashScreenParams

void WINAPI

SetSplashScreenParams(BOOL *bSuppressSplashScreen*,
int *nSplashSelect*,
UINT *iProductID*,
LPSTR *lpszPrivateString*)

A call to this function should be placed in **ProtInit**() to enable or disable the splash screen at start-up, and determine how it will be displayed.

Parameter	Description
bSuppressSplashScreen	If TRUE, suppresses the splash screen entirely. This may be appropriate if your program displays a splash screen itself elsewhere. If FALSE, the selected splash screen will be displayed for a few seconds and will then be erased.
nSplashSelect	If zero, displays the original Wonderware splash screen using the dialog resource WWStartup. This is provided for backward compatibility. If non-zero, displays the Common User Interface splash screen.
iProductID	Identifies the type of product. For a Win32 server, this value should be COMMON_IOSERVER32ID. For the Common UI splash screen, this is used to select the bitmap file that will be displayed in the splash screen. For the WWStartup splash screen, it has no function.
lpszPrivateString	Pointer to a string that can be displayed in the Common UI splash screen as additional information about the program. For the WWStartup splash screen, it has no function.
Return Value	None.
Comments	The default is for the Toolkit to display the Common User Interface splash screen at start-up. Since the server-specific code in **ProtInit**() is called before the Toolkit displays the splash screen, this is the programmer's opportunity to control or alter this behavior. If you are using the CommonUI splash screen, you might prefer instead to call the function **WWAnnounceStartup**(), which displays the CommonUI splash screen and also issues a start-up message that appears in the **Wonderware Logger**.

StatAddValue

void WINAPI

StatAddValue(HSTAT *hStat*,
PTVALUE *value*)

Add indicated value to indicated statistics item; for strings and discretes, no change.

Parameter	Description
hStat	Handle from the Toolkit that identifies a particular statistics item, assigned by a call to either **StatRegisterCounter**() or **StatRegisterRate**().
value	A union of type PTVALUE. The user must know the type of data associated with this statistical point. Based on the point type, use the appropriate field in this structure (it contains fields for discrete, integer, and real, as well as a handle to memory containing a string).
Return Value	None.
Comments	This routine assumes that the statistics item and the value in *value* are of the same data type – i.e. both are integers or both are reals. No conversion is made. Mixing data types will yield unpredictable results. This function should not be called if the statistic has been unregistered.

StatDecrementValue

void WINAPI

StatDecrementValue(HSTAT *hStat*)

Decrement value for indicated statistics item by 1; for strings and discretes, no change.

Parameter	Description
hStat	Handle from the Toolkit that identifies a particular statistics item, assigned by a call to either **StatRegisterCounter**() or **StatRegisterRate**().
Return Value	None.
Comments	This routine checks the type of data encoded in the statistics item. If it is a numerical data type, the value is decremented by 1 (by 1.0 in the case of reals). This function should not be called if the statistic has been unregistered.

StatGetValue

PTVALUE WINAPI

StatGetValue(HSTAT *hStat*)

Retrieve the current value for indicated statistics item.

Parameter	Description
hStat	Handle from the Toolkit that identifies a particular statistics item, assigned by a call to either **StatRegisterCounter**() or **StatRegisterRate**().
Return Value	A union of type PTVALUE, the value corresponding to the indicated statistics point. Note that for points of type PTT_STRING, the value contains a pointer to a memory buffer where the string is actually stored.
Comments	For a rate statistic, the value returned is always the current calculated rate value, and will always be accessed from ptValue.real. For a counter statistic, the union member containing the value is determined by the ptType passed to **StatRegisterCounter**(). If the statistics handle *hStat* is invalid, a value of PTVALUE.intg==0 is returned.

StatIncrementValue

void WINAPI

StatIncrementValue(HSTAT *hStat*)

Increment value for indicated statistics item by 1; for strings and discretes, no change.

Parameter	Description
hStat	Handle from the Toolkit that identifies a particular statistics item, assigned by a call to either **StatRegisterCounter**() or **StatRegisterRate**().
Return Value	None.
Comments	This routine checks the type of data encoded in the statistics item. If it is a numerical data type, the value is incremented by 1 (by 1.0 in the case of reals). This function should not be called if the statistic has been unregistered.

StatRegisterCounter

HSTAT WINAPI

StatRegisterCounter(IDLDEV *idLogDev*,
IDSTDDEV *idStdDev*,
LPSTR *lpszName*,
PTTYPE *ptType*)

Register a statistical counter with the indicated name and point type. The statistic can be associated either with a logical device (topic) defined in the protocol or with a standard statistics device (standard topic) – such as $PORT, $DEVICE, or $SERVER. Once the item is registered, the Toolkit will automatically handle validating the point name when a client attempts to access it, and passing updated values to interested clients. The server-specific code is responsible for updating the value indicated by HSTAT via calls to **StatIncrementValue**(), **StatAddValue**(), etc.

Parameter	Description
idLogDev	Identifier of the logical device (topic) to which this statistic is to be assigned. This is the handle supplied by the Toolkit as a parameter in the **ProtAllocateLogicalDevice**() call. If *idLogDev* is zero, the statistic will be assigned to one of the standard devices (topics) maintained internally by the Toolkit, identified by the second parameter *idStdDev*.
idStdDev	Identifier of the standard device (topic) to which this statistic is to be assigned. This parameter is used only if *idLogDev* is zero, and must be one of the following: STDDEV_PORT, STDDEV_DEVICE, or STDDEV_SERVER.
lpszName	Far pointer to a string which contains the name of the statistics item.
ptType	Identifies the point type for the item. Typically this is PTT_INTEGER, but can be any of the other point types also: PTT_DISCRETE, PTT_REAL, PTT_STRING.
Return Value	Handle of statistical item, a non-zero number assigned by the Toolkit and unique to this item that identifies this statistical point/item in any future calls to the statistical functions. Returns NULL handle if unsuccessful. If the indicated statistic is already registered, the handle of that statistic will be returned.
Comments	The statistic will automatically be initialized with a value of zero (or for strings, to a NULL pointer, indicating "no string assigned"). Also, the statistic will be reset to zero if an I/O client pokes any value to "ResetAll" on the SYSTEM topic or to "ResetStats" on the topic to which this counter belongs. If the statistic becomes invalid, you should call the function **StatUnregisterCounter**() to unregister the statistic point.

StatRegisterRate

HSTAT WINAPI

StatRegisterRate(IDLDEV *idLogDev*,
IDSTDDEV *idStdDev*,
LPSTR *lpszName*,
DWORD *interval*,
BYTE *units*,
PTTYPE *ctrType*,
HSTAT *hStatCtr*)

Register a statistical rate with the indicated name. The statistic can be associated either with a logical device (topic) defined in the protocol or with a standard statistics device (standard topic) – such as $PORT, $DEVICE, or $SERVER. Once the item is registered, the Toolkit will automatically handle validating the point name when a client attempts to access it, and passing updated values to interested clients.

The input value to the rate calculation is a counter value, which can either be another statistical item counter or a counter value maintained internally within the rate item itself. The output of the rate calculation will always be of point type PTT_REAL.

Parameter	Description
idLogDev	Identifier of the logical device (topic) to which this statistic is to be assigned. This is the handle supplied by the Toolkit as a parameter in the **ProtAllocateLogicalDevice**() call. If *idLogDev* is zero, the statistic will be assigned to one of the standard devices (topics) maintained internally by the Toolkit, identified by the second parameter *idStdDev*.
idStdDev	Identifier of the standard device (topic) to which this statistic is to be assigned. This parameter is used only if *idLogDev* is zero, and must be one of the following: STDDEV_PORT, STDDEV_DEVICE, STDDEV_SERVER.
lpszName	Far pointer to a string which contains the name of the statistics item.
interval	Unsigned 32-bit integer indicating the interval between rate calculations, in milliseconds – i.e. how often the rate value will be updated. Clients can modify this value by accessing the point with the name specified by *lpszName* concatenated with "$INTERVAL".
units	Identify the units for the divisor in the rate calculation:

Value	Meaning
STATS_TIME_MSEC	milliseconds
STATS_TIME_SEC	seconds
STATS_TIME_MIN	minutes
STATS_TIME_HR	hours
STATS_TIME_DAY	days

Example: If the rate indicates changes per second, this should be STATS_TIME_SEC.

ctrType	Identify the point type for the input counter item for the rate calculation. Must be PTT_INTEGER or PTT_REAL. Note that this parameter is ignored if the parameter *hStatCtr* is non-zero.
hStatCtr	Handle of the associated statistics **counter** item for this rate calculation. If *hStatCtr* is zero, this indicates that no input counter is used and the counter will instead be maintained internally within the rate item itself. In this case, the internal counter must be manipulated via the functions which manipulate statistics values [**StatAddValue**(), **StatIncrementValue**(), etc]. If *hStatCtr* is non-zero, then the rate calculation will be performed using the counter value maintained separately by the handle returned from **StatRegisterCounter**().
Return Value	Handle of statistical rate item, a non-zero number assigned by the Toolkit and unique to this item that identifies this statistical point/item in any future calls to the statistical functions. Returns NULL handle if unsuccessful. If the indicated statistic is already registered, the handle of that statistic will be returned.
Comments	Providing both an interval and units allows complete flexibility in the rate calculation frequency and units. For example, you can get a per-hour rate (units = TIME_HR) calculated once per minute (interval = 60000). This function call will automatically create two I/O items – one for the rate itself and one for the rate item interval, which can be accessed by clients via the rate item name concatenated with the string "$INTERVAL" (e.g. HOURLY_RATE$INTERVAL). If the statistic becomes invalid, you should call the function **StatUnregisterRate**() to unregister the statistic point.

StatSetCountersInterval

BOOL WINAPI

StatSetCountersInterval(IDLDEV *idLogDev*,
IDSTDDEV *idStdDev*,
DWORD *interval*)

Set a new update (i.e. reporting) interval for the counters on the indicated logical device (topic) or standard statistics device (standard topic – such as $PORT, $DEVICE, or $SERVER). This defines the time interval at which all registered statistical counters on the device will be updated to any interested clients. If this function is not called, the value specified by the WIN.INI setting "StatCountersInterval" (with a default interval of 10000 msec) is used. Note that the counter interval for a given logical device can also be manipulated by a client via the "CounterInterval" item name on the topic.

Parameter	Description
idLogDev	Identifier of the logical device (topic) to which this statistic is to be assigned. This is the handle supplied by the Toolkit as a parameter in the **ProtAllocateLogicalDevice**() call. If *idLogDev* is zero, the statistic will be assigned to one of the standard devices (topics) maintained internally by the Toolkit, identified by the second parameter *idStdDev*.
idStdDev	Identifier of the standard device (topic) to which this statistic is to be assigned. This parameter is used only if *idLogDev* is zero, and must be one of the following: STDDEV_PORT, STDDEV_DEVICE, STDDEV_SERVER.
interval	Unsigned 32-bit integer indicating the interval between updates, in milliseconds.
Return Value	TRUE if successful.
Comments	This function only affects statistical counters registered with **StatRegisterCounter**(). This setting is not used to control the interval at which the actual counter value changes in the statistical sub-system – it is used only to control the rate at which updates to external clients are performed.

StatSetRateInterval

BOOL WINAPI

StatSetRateInterval(HSTAT *hStat*,
DWORD *interval*)

Set new update interval for indicated rate item.

Parameter	Description
hStat	Handle from the Toolkit that identifies a particular statistics item, assigned by a call to **StatRegisterRate**().
interval	Unsigned 32-bit integer indicating the interval between rate calculations, in milliseconds.
Return Value	TRUE if successful.
Comment	This is equivalent to accessing the rate interval created when **StatRegisterRate**() was called. Note that clients can access this interval via the rate item name concatenated with the string "$INTERVAL" (e.g. HOURLY_RATE$INTERVAL).

StatSetValue

void WINAPI

StatSetValue(HSTAT *hStat*,
PTVALUE *ptValue*)

Set new value for indicated statistics item; this is used for counters and to calculate rates.

Parameter	Description
hStat	Handle from the Toolkit that identifies a particular statistics item, assigned by a call to either **StatRegisterCounter**() or **StatRegisterRate**().
ptValue	A union of type PTVALUE. The user must know the type of data associated with this statistical point. Based on the point type, use the appropriate field in this structure (it contains fields for discrete, integer, and real, as well as a handle to memory containing a string).
Return Value	None.
Comment	This function should not be called if the statistic has been unregistered.

StatSubtractValue

void WINAPI

StatSubtractValue(HSTAT *hStat*,
PTVALUE *ptValue*)

Subtract indicated value from indicated statistics item; for strings and discretes, no change.

Parameter	Description
hStat	Handle from the Toolkit that identifies a particular statistics item, assigned by a call to either **StatRegisterCounter**() or **StatRegisterRate**().
ptValue	A union of type PTVALUE. The user must know the type of data associated with this statistical point. Based on the point type, use the appropriate field in this structure (it contains fields for discrete, integer, and real, as well as a handle to memory containing a string).
Return Value	None.
Comments	This routine assumes that the statistics item and the value in *value* are of the same data type – i.e. both are integers or both are reals. No conversion is made. Mixing data types will yield unpredictable results. This function should not be called if the statistic has been unregistered.

StatUnregisterCounter

BOOL WINAPI

StatUnregisterCounter(IDLDEV *idLogDev*,
IDSTDDEV *idStdDev*,

HSTAT *hStat*)

Unregister the indicated counter on the indicated logical device. If the counter handle *hStat* is NULL, unregister ALL counter statistics on this topic. The server-specific code should call this function if the statistic counter is no longer valid. After unregistration, this item will no longer be automatically validated or updated to any interested clients. However, any clients which currently have the point on advise will retain the last updated item for the value.

Parameter	Description
idLogDev	Identifier of the logical device (topic) to which this statistic is to be assigned. This is the handle supplied by the Toolkit as a parameter in the **ProtAllocateLogicalDevice**() call. If *idLogDev* is zero, the statistic will be assigned to one of the standard devices (topics) maintained internally by the Toolkit, identified by the second parameter *idStdDev*.
idStdDev	Identifier of the standard device (topic) to which this statistic is to be assigned. This parameter is used only if *idLogDev* is zero, and must be one of the following: STDDEV_PORT, STDDEV_DEVICE, STDDEV_SERVER.
hStat	Handle from the Toolkit that identifies a particular statistics counter item, assigned by a call to **StatRegisterCounter**(). If NULL, unregister all counter statistics in this topic.
Return Value	TRUE if successful. FALSE indicates that either *idLogDev*, *idStdDev*, or *hStat* was unknown to the Toolkit.
Comment	This function should be called when the statistic counter becomes invalid. However, it need not be called as part of a **ProtFreeLogicalDevice**() function call for the indicated *idLogDev*. In this case, the Toolkit calls this function implicitly.

StatUnregisterRate

BOOL WINAPI

StatUnregisterRate(IDLDEV *idLogDev*,
IDSTDDEV *idStdDev*,
HSTAT *hStat*)

Unregister the indicated rate on the indicated logical device. If the rate handle *hStat* is NULL, unregister ALL rate statistics on this topic. The server-specific code should call this function if the statistic rate is no longer valid. After unregistration, this item will no longer be automatically validated or updated to any interested clients. However, any clients which currently have the point on advise will retain the last updated item for the value.

Parameter	Description
idLogDev	Identifier of the logical device (topic) to which this statistic is to be assigned. This is the handle supplied by the Toolkit as a parameter in the **ProtAllocateLogicalDevice**() call. If *idLogDev* is zero, the statistic will be assigned to one of the standard devices (topics) maintained internally by the Toolkit, identified by the second parameter *idStdDev*.
idStdDev	Identifier of the standard device (topic) to which this statistic is to be assigned. This parameter is used only if *idLogDev* is zero, and must be one of the following: STDDEV_PORT, STDDEV_DEVICE, STDDEV_SERVER.
hStat	Handle from the Toolkit that identifies a particular statistics rate item, assigned by a call to **StatRegisterRate**(). If NULL, unregister all rate statistics for this topic.
Return Value	TRUE if successful. FALSE indicates that either *idLogDev*, *idStdDev*, or *hStat* was unknown to the Toolkit.
Comment	This function should be called when the statistic rate becomes invalid. However, it need not be called as part of a **ProtFreeLogicalDevice**() function call for the indicated *idLogDev*. In this case, the Toolkit calls this function implicitly.

StatZeroValue

void WINAPI

StatZeroValue(HSTAT *hStat*)

Set zero value for indicated statistics item; for strings, clear string buffer pointer.

Parameter	Description
hStat	Handle from the Toolkit that identifies a particular statistics item, assigned by a call to either **StatRegisterCounter**() or **StatRegisterRate**().
Return Value	None.
Comments	This routine checks the data type of the indicated statistics point. Numerical and Boolean items are set to zero. For strings, the buffer pointer is cleared, indicating no string is assigned. This function should not be called if the statistic has been unregistered.

StrValSetNString

PTVALUE WINAPI

StrValSetNString(PTVALUE *ptValueOld*,
LPSTR *lpszValue*,
int *nMax*)

StrValSetNString() is used to initialize or change the value of a *ptValue* string while limiting the length of the input.

Parameter	**Description**
ptValueOld	If this is the initial call, set *hString* to NULL before this call. If you are changing an existing string, this is the *ptValue* used previously that may be removed or overwritten.
lpszValue	Far pointer to the new string (null terminated or to a maximum of *nMax*) which will be moved into the system memory for use by the Toolkit database.
nMax	The maximum number of characters read from the *lpszValue* string.
Return Value	New *ptValue* to be saved.
Comments	This function is functionally identical to **StrValSetString**() except that a limit can be put on the string size. This function will move characters until a NULL is reached or the maximum limit (then a NULL is stored).

Example

```
/* only sets the value to "This " */
ptValue = StrValSetNString( ptValue, "This is a new value", 5 );
```

StrValSetString

PTVALUE WINAPI

StrValSetString(PTVALUE *ptValueOld*,
LPSTR *lpszValue*)

StrValSetString() is used to initialize or change the value of a *ptValue* string.

Parameter	Description
ptValueOld	If this is the initial call, set *hString* to NULL before this call. If you are changing an existing string, this is the *ptValue* used previously that may be removed or overwritten.
lpszValue	Far pointer to the new string (null terminated) which will be moved into the system memory for use by the Toolkit database.
Return Value	New *ptValue* to be saved.
Comments	Be aware that calls to **StrValSetString**() (as well as **StrValSetNString**()) cause heap memory to be allocated. Only set the *hString* to NULL prior to the first call, or memory will be lost. When you are done with the *ptValue*, call **StrValStringFree**().

Example

```
/* We have never used the ptValue before so null it. */
ptValue.hString = NULL;
/* Now store strings ready to be sent to the Toolkit Database */
ptValue = StrValSetString( ptValue, "This is the new value" );
ptValue = StrValSetString( ptValue, "This is a newer value &
    size" );
```

Note Version 5.0 and later of the **I/O Server Toolkit** allocate string memory from the heap rather than using system global memory resources.

StrValStringFree

PTVALUE WINAPI

StrValStringFree(PTVALUE *ptValue*)

StrValStringFree() will free the memory used for a *ptValue* string.

Parameter	Description
ptValue	This is the *ptValue* supplied by the **StrValSetString**() call.
Return Value	This is the new *ptValue* to be saved for future string function calls.
Comments	Free strings only when the server has been using them internally. If the *ptValue* came from or is going to the Toolkit database, do not free it. To save or manipulate the string, copy the string to an internal string element.

StrValStringLock

LPSTR WINAPI

StrValStringLock(PTVALUE *ptValue*)

This function will lock a string in memory and return a far pointer to the beginning of the string memory.

Parameter	Description
ptValue	This is the *ptValue* supplied by the **StrValSetString**() call.
Return Value	Far pointer to the string (null terminated) which is pointed to by the *ptValue* memory handle. A NULL return means the string memory could not be locked.
Comments	Be sure to unlock any locked memory at the earliest possible opportunity. When memory is locked, it cannot be moved by Windows memory management. The less locked memory, the better.

Note Be sure to execute the same number of locks and unlocks on any piece of memory. If, for any reason, the memory is locked more than once, it must be unlocked more than once.

Example (function using *lock/unlock*)

```
BOOL
FAR PASCAL
CompareString( ptValue )
PTVALUE ptValue;  /* String from database or device */
{
LPSTR    lpszVal;
BOOL     rtn;
    rtn = FALSE;
    if( ptValue.hString != NULL )  {
        lpszVal = StrValStringLock( ptValue );
        if( lpszVal )  {
        if( lstrcmpi( lpszVal, "Value" ) == 0 )  {
            rtn = TRUE;}
        StrValStringUnlock( ptValue );
        }
    }
    return( rtn );
}
```

StrValStringUnlock

VOID WINAPI

StrValStringUnlock(PTVALUE *ptValue*)

This function will *unlock* a string in memory previously locked by **StrValStringLock**().

Parameter	Description
ptValue	This is the *ptValue* supplied by the **StrValSetString**() call.
Return Value	None.
Comments	**StrValStringLock**() and **StrValStringUnlock**() are used to *lock/unlock* the value of a *ptValue* string. Refer to the **StrValStringLock**() code example.

Note Be sure to execute the same number of locks and unlocks on any piece of memory. If, for any reason, the memory is locked more than once, it must be unlocked more than once.

SysTimerSetupProtTimer

BOOL WINAPI

SysTimerSetupProtTimer(DWORD *dwMsec*)

This function sets up a timer that goes off every *dwMsec* milliseconds and calls **ProtTimerEvent** ().

Parameter	Description
dwMsec	Interval timer range from 1 to 32767 milliseconds.
Return Value	TRUE means that the interval requested was acceptable. FALSE means that it was out of range and should be changed.
Comments	This timer should be set to a value that is reasonable for the protocol data supply rate. Intervals that are arbitrarily too short will result in needless system overhead.

SysTimerSetupRequestTimer

BOOL WINAPI

SysTimerSetupRequestTimer(DWORD *dwMsec*)

This function sets up a timer that goes off every *dwMsec* milliseconds and checks for valid data timeout errors within the Toolkit database.

Parameter	Description
dwMsec	Interval timer range from 1 to 32000 milliseconds.
Return Value	TRUE means that the interval requested was acceptable. FALSE means that it was out of range and should be changed.

Note This timer should be set to a value that is reasonable for the protocol data timeout. The interval should be a reasonable factor of the *ValidDataTimeout* parameter return in the **ProtGetValidDataTimeout**() calls. Intervals that are arbitrarily too short will result in needless system overhead.

UdAddFileTimeOffset

void WINAPI

UdAddFileTimeOffset(FILETIME **ft1*,
long *lDelta*,
FILETIME **ft2*)

Add indicated time difference (in 100 nsec / 8192) to source time mark.

Return result in destination time mark.

Note: FILETIME is a 64-bit value defined in Win32 as the number of 100 nanosecond intervals since January 1, 1601. It is organized as two DWORDS, dwLowDateTime and dwHighDateTime.

Parameter	Description
ft1	Pointer to a FILETIME structure for a date/time stamp which serves as the source time mark, i.e. the date/time to which the indicated interval will be added.
lDelta	Interval to be added to source date/time, in units of 100 nanoseconds / 8192, i.e. 1.220703125 milliseconds.
ft2	Pointer to a FILETIME structure for a date/time stamp which serves as the destination time mark, i.e. the resulting date/time stamp after the interval *lDelta* has been added to the source date/time stamp.
Return Value	None.
Comments	The use of the 100 nsec / 8192 is provided for convenience and rapid calculations, as it is close to a 1 msec interval, but requires only a simple shift operation to calculate, instead of division or multiplication of a 64-bit number. You may prefer to use the function **UdAddTimeMsec**() instead

UdAddTimeMSec

void WINAPI

UdAddTimeMSec(FILETIME **ft1*,
long *lDelta*,
FILETIME **ft2*)

Add indicated time difference (in msec) to source time mark.

Return result in destination time mark.

Note: FILETIME is a 64-bit value defined in Win32 as the number of 100 nanosecond intervals since January 1, 1601. It is organized as two DWORDS, dwLowDateTime and dwHighDateTime.

Parameter	Description
ft1	Pointer to a FILETIME structure for a date/time stamp which serves as the source time mark, i.e. the date/time to which the indicated interval will be added.
lDelta	Interval to be added to source date/time, in units of 1 millisecond.
ft2	Pointer to a FILETIME structure for a date/time stamp which serves as the destination time mark, i.e. the resulting date/time stamp after the interval *lDelta* has been added to the source date/time stamp.
Return Value	None.
Comments	The use of 1 millisecond intervals is convenient, but conversions to and from units of 100 nanoseconds require division or multiplication of 64-bit numbers. If large numbers of high-speed calculations are required, you may prefer to use calls to the function **UdAddFileTimeOffset**() instead.

UDDbGetName

VOID WINAPI

UDDbGetName(IDLDEV *idLogDev*,
HDB *hDb*,
LPSTR *lpszName*)

Get name of database item for indicated logical device.

Note: this is the same as the following function:

VOID WINAPI DbGetName (IDLDEV idLogDev, HDB hDb, LPSTR lpszName);

Parameter	Description
idLogDev	Topic (logical device) identifier that was supplied by the Toolkit as a parameter in the **ProtAllocateLogicalDevice**() call.
hDb	Handle from the Toolkit that is unique to this item and was supplied as a parameter in the **ProtCreatePoint**() call.
lpszName	Far pointer to the string buffer where the name will be returned.

UdDeltaFileTime

long WINAPI

UdDeltaFileTime(FILETIME **ft1*,
FILETIME **ft2*)

Calculate signed difference between 2 FILETIMEs in 100 nsec / 8192

delta = (ft1 - ft2) / 8192

Note: FILETIME is a 64-bit value defined in Win32 as the number of 100 nanosecond intervals since January 1, 1601. It is organized as two DWORDS, dwLowDateTime and dwHighDateTime.

Parameter	Description
ft1	Pointer to a FILETIME structure for a date/time stamp which serves as the ending time mark, i.e. the date/time stamp at which some event or interval finished.
ft2	Pointer to a FILETIME structure for a date/time stamp which serves as the starting time mark, i.e. the date/time stamp at which some event or interval started.
Return Value	Interval between the two date/time stamps, in units of 100 nanoseconds / 8192, i.e. 1.220703125 milliseconds. Value is returned as a signed LONG integer.
Comments	The use of the 100 nsec / 8192 is provided for convenience and rapid calculations, as it is close to a 1 msec interval, but requires only a simple shift operation to calculate, instead of division or multiplication of a 64-bit number. You may prefer to use the function **UdDeltaTimeMsec**() instead

UdDeltaTimeMSec

long WINAPI

UdDeltaTimeMSec(FILETIME **ft1*,
FILETIME **ft2*)

Calculate signed difference between 2 FILETIMEs in msec

(delta = (ft1 - ft2)/10000)

Return difference as a signed LONG integer.

Note: FILETIME is a 64-bit value defined in Win32 as the number of 100 nanosecond intervals since January 1, 1601. It is organized as two DWORDS, dwLowDateTime and dwHighDateTime.

Parameter	Description
ft1	Pointer to a FILETIME structure for a date/time stamp which serves as the ending time mark, i.e. the date/time stamp at which some event or interval finished.
ft2	Pointer to a FILETIME structure for a date/time stamp which serves as the starting time mark, i.e. the date/time stamp at which some event or interval started.
Return Value	Interval between the two date/time stamps, in units of 1 millisecond. Value is returned as a signed LONG integer.
Comments	The use of 1 millisecond intervals is convenient, but conversions to and from units of 100 nanoseconds require division or multiplication of 64-bit numbers. If large numbers of high-speed calculations are required, you may prefer to use calls to the function **UdDeltaFileTime**() instead.

UdInit

BOOL WINAPI

UdInit(HANDLE *hInstance*,
HANDLE *hPrevInstance*,
LPSTR *lpszCmdLine*,
int *nCmdShow*)

This function is intended for use by a Windows application that already exists and needs to be extended to include the DDE capability provided by the Toolkit. It is used to initialize the I/O Server Toolkit and should only be used by a Windows application which supplies its own WinMain() function. **UdInit**() should be called early in the activation of such applications. Most I/O Servers do not have to call this function since they allow the Toolkit to supply the WinMain() function. The parameter list is identical to the parameter list for the Windows WinMain() function.

Parameter	Description
hInstance	Handle of the current instance of the application.
hPrevInstance	Handle of the previous instance of the application.
lpszCmdLine	Pointer to a string containing the command line for the application.
nCmdShow	Specifies how the window is to be shown.
Return Value	TRUE indicates the Toolkit initialized successfully.
	FALSE indicates an initialization failure and that the application should terminate.
Comments	This function should only be called by an application which must supply its own WinMain() function rather than using the one supplied in the Toolkit.

UdReadAnyMore

BOOL WINAPI

UdReadAnyMore(HFILE *hCfgFile*,
short FAR **pbAnyMore*)

This function reads a *bAnyMore* flag from the configuration file. This flag is used within the configuration file to indicate whether more records of a certain type exist. Such a flag is usually necessary when the number of records is unknown.

Parameter	Description
hCfgFile	File handle of the configuration file.
pbAnyMore	Pointer to the boolean value to be set with the value read out of the configuration file.
Return Value	TRUE if the read operation succeeded. FALSE otherwise.
Comments	None.

UdReadVersion

BOOL WINAPI

UdReadVersion(long *lMagic*,
long FAR **lVersion*,
HFILE *hCfgFile*)

This function reads the version number from the server configuration file. It also verifies that the magic number stored in the file matches the specified magic number.

Parameter	**Description**
lMagic	Magic number to be checked against the magic number in the configuration file.
lVersion	Points to a longword variable to receive the configuration file's version number.
hCfgFile	File handle of the configuration file.
Return Value	TRUE if the magic number matches. FALSE otherwise.
Comments	Configuration file must be previously opened.

UdTerminate

void WINAPI

UdTerminate(HINSTANCE *hInstance*)

This function is intended for use by Windows applications that already exist and need to be extended to include the DDE capability provided by the Toolkit. It is used to close the I/O Server Toolkit, but it should only be used by a Windows application which supplies its own WinMain() function and calls the **UdInit**() function to initialize the Toolkit. **UdTerminate**() should be called at application shutdown time. Most I/O Servers do not have to call this function since they allow the Toolkit to supply the WinMain() function.

Parameter	Description
hInstance	Handle of the current instance of the application.
Return Value	None.
Comments	This function should only be called by an application which must supply its own WinMain() function rather than using the one supplied in the Toolkit.

UdWriteAnyMore

BOOL WINAPI

UdWriteAnyMore(HFILE *hCfgFile*,
short *bAnyMore*)

This function writes a *bAnyMore* flag to the configuration file. This flag is used within the configuration file to indicate whether more records of a certain type exist.

Parameter	Description
hCfgFile	File handle of the configuration file.
bAnyMore	Boolean value to be written to the configuration file.
Return Value	TRUE if the write operation succeeded. FALSE otherwise.
Comments	Configure file must be previously opened.

UdWriteVersion

BOOL WINAPI

UdWriteVersion(long *lMagic*,
long *lVersion*,
HFILE *hCfgFile*,
char FAR **szDate*,
char FAR **szTime*)

This function writes the version number, magic number, date, and time to the server configuration file.

Parameter	Description
lMagic	Magic number to be written to the configuration file.
lVersion	Version number to be written to the configuration file.
hCfgFile	File handle of the configuration file.
szDate	Points to a date string to be written.
szTime	Points to a time string to be written.
Return Value	TRUE if the write operation succeeds. FALSE otherwise.
Comments	Configuration file must be previously opened.

WriteComm

int WINAPI

WriteComm(int *idComDev*,
void **lpvBuf*,
int *cbWrite*)

This function writes the specified bytes to the specified communications device.

Parameter	Description
idComDev	The *id* of the communications device to be written to. The **OpenComm**() function returns this value.
lpvBuf	Points to the buffer that contains the bytes to be written.
cbWrite	Specifies the number of bytes to be written.
Return Value	The return value specifies the number of bytes written, if successful. The return value is less than zero if an error occurs, making the absolute value of the return value the number of bytes written.
Comments	When an error occurs, the cause of the error can be determined by using the **GetCommError**() function to retrieve the error value and status.Windows NT/2000 only. Emulates Windows function.

WriteWindowSizeToWinIni

VOID WINAPI

WriteWindowSizeToWinIni(HWND *hWnd*)

This function saves the size of the server window to the last adjusted size.

Parameter	Description
hWnd	Handle of window whose size is to be saved.
Return Value	None.
Comments	None.

WWAnnounceStartup

void WINAPI

WWAnnounceStartup(UINT *iProductID*,
LPSTR *lpszSplashString*

Set up to display Common User Interface splash screen and log start-up message, using indicated product ID and private strings.

Parameter	Description
iProductID	Identifies the type of product. For a Win32 server, this value should be COMMON_IOSERVER32ID. For the Common UI splash screen, this is used to select the bitmap file that will be displayed in the splash screen.
lpszSplashString	Pointer to a string that can be displayed in the splash screen as additional information about the program.
Return Value	None.
Comments	The default is for the Toolkit to display the Common User Interface splash screen at start-up. Since the server-specific code in **ProtInit**() is called before the Toolkit displays the splash screen, this is the programmer's opportunity to control or alter this behavior. If do not wish to use the CommonUI splash screen, you should call **SetSplashScreenParams**() to select the WWStartup dialog resource or to suppress the splash screen altogether. In this case, you should also call **debug**() to display your own startup message in the **Wonderware Logger**.

Note If you are developing products for Wonderware, this function and its corresponding definitions will be of interest..

The following code would appear in ProtInit().

Example

```
/* set up string with server name and version number */
strcpy (szServerIDstring,
        GetString (STRUSER+142));   /* "Sample I/O Server" */
L = strlen (szServerIDstring);
sprintf (&szServerIDstring[L],
         GetString (STRUSER+145),   /* "- Version %s" */
          SERVER_VERSION);
/* set up to display common start-up message and splash screen */
WWAnnounceStartup (COMMON_IOSERVER32ID,
                   (LPSTR) szServerIDstring);
```

WWCenterDialog

VOID WINAPI

WWCenterDialog(HWND *hDlg*)

This function places the specified dialog at the center of the screen.

Parameter	Description
hDlg	Window handle for the dialog to be centered.
Return Value	None.
Comments	None.

WWConfigureComPort

BOOL WINAPI

WWConfigureComPort(LPWW_CP_DLG_LABELS *lpDlgLabels*)

This function displays and manages the communications port settings configuration dialog. It should only be used by serial servers. This dialog allows configuration of communications parameters for one or more serial communications ports. These settings are written to the **WW_CP_DLG_LABELS** structure for use by the server.

Parameter	Description
lpDlgLabels	Points to a **WW_CP_DLG_LABELS** structure that contains initial and final dialog values and dialog control information.
Return Value	The return value is TRUE if the dialog box display is successful. Otherwise, it is FALSE.
Comments	The **WW_CP_DLG_LABELS** structure must be properly initialized prior to calling this function. For a complete description of how to implement, refer to "I/O Server Toolkit Data Structures" section.
Structure	typedef struct tagWW_CP_DLG_LABELS {

```
HWND hwndOwner;
char far *szDriverName;
LPWW_CP_PARAMS lpDefaultCpParams;
LPWW_CP_PARAMS lpCpParams;
int iNumPorts;
BOOL bAllowBaud110;
BOOL bAllowBaud300;
BOOL bAllowBaud600;
BOOL bAllowBaud1200;
BOOL bAllowBaud2400;
BOOL bAllowBaud4800;
BOOL bAllowBaud9600;
BOOL bAllowBaud14400;
BOOL bAllowBaud19200;
BOOL bAllowBaud38400;
BOOL bAllowBaud56000; /* not used */
BOOL bAllowBaud57600; /* not used */
BOOL bAllowBaud115200; /* not used */
BOOL bAllowBaud128000; /* not used */
BOOL bAllowDatabits7;
BOOL bAllowDatabits8;
BOOL bAllowStopbits1;
BOOL bAllowStopbits2;
BOOL bAllowParityEven;
BOOL bAllowParityOdd;
BOOL bAllowParityNone;
BOOL bAllowParityMark;
BOOL bAllowParitySpace;
char far *szCustom1BoxLabel;
char far *szCustom1Radio1Label;
char far *szCustom1Radio2Label;
struct tagWW_CP_DLG_LABELS FAR
        *lpRadio2DlgLabels;
char far *szCustom2BoxLabel;
char far *szCustom2Radio1Label;
```

```
    char far *szCustom2Radio2Label;
    char far *szCustom3BoxLabel;
    char far *szCustom3Radio1Label;
    char far *szCustom3Radio2Label;
    char far *szCheck1Label;
    char far *szCheck2Label;
    char far *szCustomEditLabel;
    UINT uCustomEditBase;
    UINT uCustomEditLowLimit;
    UINT uCustomEditHighLimit;
    FARPROC lpfnConfigureSave;
    int iInternalUseOnly;
    char reserved[16];

} WW_CP_DLG_LABELS, FAR
*LPWW_CP_DLG_LABELS;
```

A variable with the type of **WW_CP_PARAMS** structure is a member of **WW_CP_DLG_LABELS** and can be written and read directly from the server configuration file. It is properly aligned for all platforms and has the following form:

```
typedef struct tagWW_CP_PARAMS {

    unsigned long uBaud;
    unsigned long uDataBits;
    unsigned long uStopBits;
    unsigned long Parity;
    unsigned long uReplyTimeout;
    unsigned long uCustomEdit;
    short bCustom1Radio;
    short bCustom2Radio;
    short bCustom3Radio;
    short bCheck1;
    short bCheck2;
    char reserved[30];   /* pad to 64 bytes */

} WW_CP_PARAMS, FAR *LPWW_CP_PARAMS;
```

For a full description of this structure, refer to the "I/O Server Toolkit Data Structures" chapter.

WWConfigureServer

BOOL WINAPI

WWConfigureServer(LPWW_SERV_PARAMS *lpServerParams*)

This function displays and manages the server parameter configuration dialog. This dialog allows configuration of several parameters related to overall operation of the I/O Server. These settings are written out as profile information to the WIN.INI file by this dialog function and only take effect upon restarting of the server.

Parameter	Description
lpServerParams	Points to a **WW_SERV_PARAMS** structure that contains initial and final dialog values and dialog control information.
Return Value	The return value is TRUE if the dialog box display is successful. Otherwise, it is FALSE.
Comments	The **WW_SERV_PARAMS** structure must be properly initialized prior to calling this function. For a complete description of how to implement, refer to "I/O Server Toolkit Data Structures" for details.
Structure	typedef struct tagWW_SERV_PARAMS { HWND *hwndOwner*; char far **szCfgPath*; int *iSizeOfszCfgPath*; char far **szDriverName*; BOOL *bIndefWriteRetrySupported*; BOOL *bIndefWriteRetry*; BYTE *bPreventChanges*; BYTE *bNotService*; BYTE *bCFGfileUnused;* BYTE *nNTServiceSetting*; char far **szCaption*; char *reserved[8]*; } WW_SERV_PARAMS, FAR *LPWW_SERV_PARAMS;

The value returned in *nNTServiceSetting* is an OR of the following bits:

```
#define WW_NTSERVICE_IS_SERVICE   (0x01)  /* set to run as NT service */
#define WW_NTSERVICE_CHANGED      (0x02)  /* setting was changed */
#define WW_NTSERVICE_ERROR        (0x04)  /* unable to establish new
                                             service settings */
```

WWConfirm

BOOL WINAPI

WWConfirm(LPWW_CONFIRM *lpConfirm*)

This function displays and manages the confirmation dialog which indicates the directory or file where server settings are to be saved. It should only be called when the configuration file does not currently exist.

Parameter	Description
lpConfirm	Points to a **WW_CONFIRM** structure that contains initial and final dialog values and dialog control information.
Return Value	The return value is TRUE if the dialog box display is successful. Otherwise, it is FALSE.
Comments	The **WW_CONFIRM** structure must be properly initialized prior to calling this function. For a complete description of this structure refer to the "I/O Server Toolkit Data Structures" chapter.
Structure	typedef struct tagWW_CONFIRM { HWND *hwndOwner*; * Identifies Window that owns the dialog box */ char far **szCfgPath*; /* points to a buffer that holds pathname */ int *iSizeOfszCfgPath*; char far **szDriverName*; char *reserved[16]*; } WW_CONFIRM, FAR *LPWW_CONFIRM;

WWDisplayAboutBox

BOOL WINAPI

WWDisplayAboutBox(LPWW_AB_INFO *lpAbout*)

This function displays and manages the dialog displaying copyright and version information. It is generally intended for use by Wonderware servers since it displays the Wonderware copyright information. However, it does provide facilities for easily displaying version and date information and is available for use by other servers.

Parameter	Description
lpAbout	Points to a **WW_AB_INFO** structure that contains dialog control information.
Return Value	The return value is TRUE if the dialog box display is successful. Otherwise, it is FALSE.
Comments	The **WW_AB_INFO** structure must be properly initialized prior to calling this function. For a complete description on this structure, refer to the I/O Server Toolkit Data Structures chapter.
Structure	typedef struct tagWW_AB_INFO { HWND *hwndOwner*; char far **szDriverName*; char far **szId*; char far **szVersion*; char far **szCopyright*; HICON *hIcon*; char far *szComment; char *reserved[12]*; } WW_AB_INFO, FAR *LPWW_AB_INFO;

WWDisplayAboutBoxEx

BOOL WINAPI

WWDisplayAboutBoxEx(LPWW_AB_INFO *lpAbout*,

UINT *iProductID*,

LPSTR *szPrivateStr*)

Display Common User Interface about box, using indicated product ID and private string, or Wonderware Common Dialog about box, as is appropriate.

Note If you are developing products for Wonderware, this function will be of interest. This function serves primarily to provide source code compatibility between FS2000 and pre-FS2000 servers.

Parameter	Description
lpAbout	Points to a **WW_AB_INFO** structure that contains dialog control information.
iProductID	Identifies the type of product. For a Win32 server, this value should be COMMON_IOSERVER32ID. For the Common UI splash screen, this is used to select the bitmap file that will be displayed in the splash screen.
szPrivateStr	Pointer to a string that can be displayed in the About Box as additional information about the program.
Return Value	The return value is TRUE if the dialog box display is successful. Otherwise, it is FALSE.
Comments	The **WW_AB_INFO** structure must be properly initialized prior to calling this function. For a complete description on this structure, refer to the I/O Server Toolkit Data Structures chapter.
Structure	typedef struct tagWW_AB_INFO { HWND *hwndOwner*; char far **szDriverName*; char far **szId*; char far **szVersion*; char far **szCopyright*; HICON *hIcon*; char far *szComment; char *reserved[12]*; } WW_AB_INFO, FAR *LPWW_AB_INFO;

WWDisplayConfigNotAllow

VOID WINAPI

WWDisplayConfigNotAllow(LPSTR *szAppName*)

This function displays a message box indicating that configuration of the server is not allowed while the server is in use.

Parameter	Description
szAppName	Pointer to a character string containing the server's application name.
Return Value	None.
Comments	None.

WWDisplayErrorCreating

VOID WINAPI

WWDisplayErrorCreating(LPSTR *szAppName*,

LPSTR *szFileName*)

This function displays a message box indicating that an error was encountered while creating the specified file.

Parameter	Description
szAppName	Pointer to a character string containing the server's application name.
szFileName	Pointer to a character string containing the file name for which the create operation failed.
Return Value	None.
Comments	None.

WWDisplayErrorReading

VOID WINAPI

WWDisplayErrorReading(LPSTR *szAppName*,
LPSTR *szFileName*)

This function displays a message box indicating that an error was encountered while reading the specified file.

Parameter	Description
szAppName	Pointer to a character string containing the server's application name.
szFileName	Pointer to a character string containing the file name for which the read operation failed.
Return Value	None.
Comments	None.

WWDisplayErrorWriting

VOID WINAPI

WWDisplayErrorWriting(LPSTR *szAppName*,
LPSTR *szFileName*)

This function displays a message box indicating that an error was encountered while writing the specified file.

Parameter	Description
szAppName	Pointer to a character string containing the server's application name.
szFileName	Pointer to a character string containing the file name for which the write operation failed.
Return Value	None.
Comments	None.

WWDisplayKeyNotEnab

int WINAPI

WWDisplayKeyNotEnab(LPSTR *szAppName*)

This function displays a message box indicating that the installed security key does not enable operation of this I/O Server.

Parameter	Description
szAppName	Pointer to a character string containing the server's application name.
Return Value	Returns the MessageBox() return code.
Comments	This function is not used by servers which do not utilize a hardware security key.

WWDisplayKeyNotInst

int WINAPI

WWDisplayKeyNotInst(LPSTR *szAppName*)

This function displays a message box indicating that the required security key is not installed on the system.

Parameter	Description
szAppName	Pointer to a character string containing the server's application name.
Return Value	Returns the MessageBox() return code.
Comments	This function is not used by servers which do not utilize a hardware security key.

WWDisplayOutofMemory

VOID WINAPI

WWDisplayOutofMemory(LPSTR *szAppName*,
LPSTR *szObjectName*)

This function displays a message box indicating that an error was encountered while allocating memory for the specified object.

Parameter	Description
szAppName	Pointer to a character string containing the server's application name.
szObjectName	Pointer to a character string containing the description of the object for which the memory allocating failed.
Return Value	None.
Comments	None.

WWFormCpModeString

VOID WINAPI

WWFormCpModeString(LPWW_CP_PARAMS *lpComPortParams*,
int *index*,
char FAR **szMode*)

This function creates a null-terminated string containing device control information.

Parameter	Description
lpComPortParams	Points to a **WW_CP_PARAMS** structure defining communications port parameters. This structure is read directly from the configuration file and returned by the **WWConfigureComPort()** function.
index	Communications port index (e.g. 1 for COM1).
**szMode*	Points to the string to receive the resulting device control information string.
Return Value	None.
Comments	This string will have the same format as the MS-DOS mode command.

WWGetDialogHandle

HWND WINAPI

WWGetDialogHandle(void)

This function returns a window handle to the top-most dialog in the current application.

Parameter	Description
Return Value	Window handle of the top-most dialog in the current application.
Comments	None.

WWGetDriverNameExtension

BOOL WINAPI
WWGetDriverNameExtension(LPSTR *lpszNameExt*,
int *nLen*)

Get name extension for use in storing and retrieving Registry settings.

This function is implemented in the Wonderware Common Dialog DLL and is called by **GetServerNameExtension**(). Locating the source of the extension string in a single place helps ensure consistency between the server-specific code and the Common Dialogs. The string is "_IOServer" and is appended to the "short" server name to produce the full name of the server, which is used for accessing the Registry and the Service Control Manager.

Parameter	Description
lpszNameExt	Points to a string buffer to which the extension string can be copied.
nLen	Length of the buffer at *lpszNameExt*.
Return Value	True if successful.
Comments	The string returned is "_IOServer" and is obtained from the Wonderware Common Dialogs to ensure that it is consistent.

WWGetExeFilePath

char *

WWGetExeFilePath(char **szCfgPathStr*,
int *maxlen*)

Get path to directory where executable is located.

Replacement for _getcwd (temp_szCfgPath, PATH_STRING_SIZE);

Use this function to get the path to the server EXE file, instead of using getcwd() or _getcwd(). The reason for this is that on a Windows NT/2000 platform, the CWD (current working directory) may in fact be the directory of another program that is starting up your server. [You can even encounter this problem when developing with Microsoft Visual C++, if your project directory and your executable directory are different.] **WWGetExeFilePath**() will get the correct path to the server executable, regardless of how the server is invoked.

Parameter	Description
szCfgPathStr	Points to a string buffer into which the path string can be stored.
maxlen	Length of the buffer at *szCfgPathStr*.
Return Value	Pointer to the path string. Returns NULL if unsuccessful.
Comments	The maximum length of a path string is defined in Microsoft Visual C++ by the constant _MAX_PATH.

WWGetOsPlatform

DWORD

WWGetOsPlatform(void);

Get platform, operating system info, summarize in global variable dwWWOsPlatform.

If dwWWOsPlatform is zero, perform operating system calls to determine the platform; otherwise, just return the present value.

This routine is called by the I/O Server Toolkit early in its start-up sequence. You can get the value from the global variable

```
extern DWORD dwWWOsPlatform;
```

or you can force a re-read of the operating system by forcing the value to zero and calling the function:

```
dwWWOsPlatform = 0;
WWGetOsPlatform( );
```

The platform information is returned as a combination of bits, defined as follows:

```
#define WW_OSPLATFORM_W16         (0x00000001)   /* Win 16              */

#define WW_OSPLATFORM_W32         (0x00000002)   /* Win 32              */
#define WW_OSPLATFORM_WIN31       (0x00000100)   /* Windows 3.1x        */
#define WW_OSPLATFORM_WIN95       (0x00000200)   /* Windows 95          */
#define WW_OSPLATFORM_NT          (0x00000400)   /* Windows NT          */
#define WW_OSPLATFORM_WOW         (0x00010000)   /* WOW (Win16 on NT)   */
#define WW_OSPLATFORM_INTEL       (0x01000000)   /* Intel processor     */
#define WW_OSPLATFORM_ALPHA       (0x02000000)   /* DEC Alpha processor */
#define WW_OSPLATFORM_MIPS        (0x04000000)   /* MIPS processor      */
#define WW_OSPLATFORM_PPC         (0x08000000)   /* Power PC processor  */
#define WW_OSPLATFORM_UNKNOWN     (0x80000000)   /* unable to read info */
```

Example

```
/* determine platform */
WWGetOsPlatform();
if ((dwWWOsPlatform & (WW_OSPLATFORM_W32 | WW_OSPLATFORM_NT)) ==
    (WW_OSPLATFORM_W32 | WW_OSPLATFORM_NT))
{
    /* Win32 on Windows NT */
    debug ("Running on Windows NT");
}
else
{
    /* not Windows NT -- assume running on Windows 95 */
    debug ("Running on Windows 95 or earlier");
}
```

wwHeap_AllocPtr

LPVOID WINAPI

wwHeap_AllocPtr(HHEAP *hHeap*,
WORD *wGmemFlags*,
DWORD *dwSize*)

wwHeap_AllocPtr() is used to allocate the specified amount of memory using the heap specified by *hHeap*.

Parameter	Description
hHeap	The heap handle supplied by **wwHeap_Init**().
wGmemFlags	Control flags for the allocated memory: GMEM_ZEROINIT GMEM_MOVEABLE
dwSize	Number of bytes needed in this piece of memory.
Return Value	A far pointer to the memory allocated. A NULL is returned if the memory could not be allocated.
Comments	When Windows memory is low or large blocks cannot be allocated, a return of NULL may result. The application must handle this situation gracefully. For example, displaying a Message Box warning the operator about low memory; then rejecting the current operation.

Note Message Boxes indicating low memory must be SYSTEM modal or they won't appear in low memory conditions. See **WWDisplayOutofMemory**() function.

wwHeap_FreePtr

VOID WINAPI

wwHeap_FreePtr(HHEAP *hHeap*,
LPVOID *lpPtr*)

wwHeap_FreePtr() is used to free the allocated memory specified by *lpPtr*.

Parameter	Description
hHeap	The heap handle supplied by **wwHeap_Init**().
lpPtr	Long pointer to the allocated memory.
Return Value	None.
Comments	None.

wwHeap_Init

HHEAP WINAPI

wwHeap_Init(void)

wwHeap_Init() is used to create and initialize a heap.

Parameter	Description
Return Value	The handle for this heap. NULL means there was an error and **no** memory heap was allocated.
Comments	This call must be done prior to any **wwHeap_AllocPtr**(), **wwHeap_FreePtr**(), and **wwHeap_ReAllocPtr**().

wwHeap_ReAllocPtr

LPVOID WINAPI

wwHeap_ReAllocPtr(HHEAP *hHeap*,
LPVOID *lpPtr*,
WORD *wGmemFlags*,
DWORD *dwSize*)

wwHeap_ReAllocPtr() is used to re-allocate the specified amount of memory used in the heap specified by *lpPtr*.

Parameter	Description
hHeap	The heap handle supplied by **wwHeap_Init**().
lpPtr	Long pointer to the allocated memory.
wGmemFlags	Control flags for the allocated memory:
	GMEM_ZEROINIT
	GMEM_MOVEABLE
dwSize	Number of bytes needed in this piece of memory.
Return Value	A long pointer to the memory allocated. A NULL is returned if the memory could not be allocated.
Comments	When Windows memory is low or large blocks cannot be allocated, a return of NULL may result. The application must handle this situation gracefully. For example, displaying a message box to warn the operator about low memory; then rejecting the current operation.

Note Message boxes indicating low memory must be SYSTEM modal or they won't appear in low memory conditions. See **WWDisplayOutofMemory**() function.

wwHeap_Release

BOOL WINAPI

wwHeap_Release(HHEAP *hHeap*)

This function is used to release a heap which was created with **wwHeap_Init**().

Parameter	Description
hHeap	The heap handle supplied by **wwHeap_Init**().
Return Value	TRUE indicates success. FALSE indicates failure.
Comments	None.

WWInitComPortComboBox

VOID WINAPI

WWInitComPortComboBox(HWND *hDlg*,
int *iNumPorts*,
int *idControl*)

This function creates a communications port selection box for display on a dialog. Most commonly, it will be used in a topic configuration dialog for selection of the communications port for serial communications.

Parameter	Description
hDlg	Window handle of the dialog to contain the combo box.
iNumPorts	Number of communications ports to be listed.
idControl	Control identifier of the selection box.
Return Value	None.
Comments	None.

WWReadAnyMore

Note: the following functions have been retired, and no longer work:

WWReadVersion(), **WWWriteVersion**(),
WWReadAnyMore(), **WWWriteAnyMore**()

Either replace them with their corresponding UdXXX functions

UdReadVersion(), **UdWriteVersion**(),
UdReadAnyMore(), **UdWriteAnyMore**()

or create user-defined functions to accomplish the same result.

WWReadVersion

Note: the following functions have been retired, and no longer work:

WWReadVersion(), **WWWriteVersion**(),
WWReadAnyMore(), **WWWriteAnyMore**()

Either replace them with their corresponding UdXXX functions

UdReadVersion(), **UdWriteVersion**(),
UdReadAnyMore(), **UdWriteAnyMore**()

or create user-defined functions to accomplish the same result.

WWSelect

BOOL WINAPI

WWSelect(LPWW_SELECT *lpSelectParams*)

This function displays a dialog containing a list box which will contain a list of strings specified by the server. The user will be provided options for adding, modifying, or deleting entries from this list. This function is most commonly used to display a list of topics or boards for configuration.

Parameter	Description
lpSelectParams	Points to a **WW_SELECT** structure that contains information necessary for displaying the selection list.
Return Value	The return value is TRUE if the dialog box display is successful. Otherwise, it is FALSE.
Comments	The **WW_SELECT** structure contains several pointers to callback functions which must be supplied by the server developer. For a complete description of how to implement, refer to "I/O Server Toolkit Data Structure" chapter.
Structure	typedef struct tagWW_SELECT { HWND *hwndOwner*; char far **szTitle*; char far **szGroupBoxLabel*; GETLISTHEADPROC *lpfnGetListHead*; GETNEXTNODEPROC *lpfnGetNextNode*; GETNODENAMEPROC *lpfnGetNodeName*; ADDNODEPROC *lpfnAddNode*; CONFIGNODEPROC *lpfnConfigNode*; DELETENODEPROC *lpfnDeleteNode*; BOOL *bAddDeleteModifyEnabled*; unsigned char *bDoNotConfirmDeletes*; char *reserved[15]*; } WW_SELECT, FAR *LPWW_SELECT;

WWSetAffinityToFirstCPU

void WINAPI

WWSetAffinityToFirstCPU(void)

This function locks the I/O Server execution to the first CPU on a SMP (symmetrical multiprocessor) machine only.

Parameter	Description
Return Value	None.
Comments	None.

WWTranslateCDlgToWinBaud

UINT WINAPI

WWTranslateCDlgToWinBaud(UINT *uCDlgBaud*)

This function translates the WWCOMDLG constant for baud rate to the Windows equivalent.

Parameter	Description
uCDlgBaud	Specifies WWCOMDLG constant representing the baud rate for the characters sent and received.
Return Value	The return value is the Windows constant corresponding to the baud rate specified.
Comments	None.

WWTranslateCDlgToWinData

UINT WINAPI

WWTranslateCDlgToWinData(UINT *uCDlgData*)

This function translates the WWCOMDLG constant for number of data bits to the Windows equivalent.

Parameter	Description
uCDlgData	WWCOMDLG constant representing number of bits in the characters sent and received.
Return Value	The return value is the Windows constant corresponding to the number of data bits specified.
Comments	None.

WWTranslateCDlgToWinParity

UINT WINAPI

WWTranslateCDlgToWinParity(UINT *uCDlgParity*)

This function translates the WWCOMDLG constant for parity to the equivalent Windows.

Parameter	Description
uCDlgParity	Specifies WWCOMDLG constant representing the parity for the characters sent and received.
Return Value	The return value is the Windows constant corresponding to the parity specified.
Comments	None.

WWTranslateCDlgToWinStop

UINT WINAPI

WWTranslateCDlgToWinStop(UINT *uCDlgStop*)

This function translates the WWCOMDLG constant for number of stop bits to the Windows equivalent.

Parameter	Description
uCDlgStop	WWCOMDLG constant representing number of stop bits.
Return Value	The return value is the Windows constant corresponding to the number of stop bits specified.
Comments	None.

WWTranslateWinBaudToCDlg

UINT WINAPI

WWTranslateWinBaudToCDlg(UINT *uBaud*)

This function translates the Windows constant for baud rate to the WWCOMDLG equivalent.

Parameter	Description
uBaud	Specifies baud rate for the characters sent and received. Must be one of CS_BAUD_110, CS_BAUD_300, CS_BAUD_600, CS_BAUD_1200, CS_BAUD_2400, CS_BAUD_4800, CS_BAUD_9600, CS_BAUD_14400, CS_BAUD_19200, CS_BAUD_38400.
Return Value	The return value is the WWCOMDLG constant corresponding to the baud rate specified.
Comments	None.

WWTranslateWinDataToCDlg

UINT WINAPI

WWTranslateWinDataToCDlg(UINT *uWinData*)

This function translates the Windows constant for number of data bits (7 or 8) to the WWCOMDLG equivalent.

Parameter	Description
uWinData	Specifies number of bits in the characters sent and received. Can be 7 or 8.
Return Value	The return value is the WWCOMDLG constant corresponding to the number of data bits specified.
Comments	None.

WWTranslateWinParityToCDlg

UINT WINAPI

WWTranslateWinParityToCDlg(UINT *uParity*)

This function translates the Windows constant for parity to the WWCOMDLG equivalent.

Parameter	Description
uParity	Specifies parity for the characters sent and received. Must be one of CS_PARITY_EVEN, CS_PARITY_ODD, CS_PARITY_NONE, CS_PARITY_MARK, CS_PARITY_SPACE.
Return Value	The return value is the WWCOMDLG constant corresponding to the parity specified.
Comments	None.

WWTranslateWinStopToCDlg

UINT WINAPI

WWTranslateWinStopToCDlg(UINT *uStopBits*)

This function translates the Windows constant for number of stop bits to the WWCOMDLG equivalent.

Parameter	Description
uStopBits	Specifies number of stop bits in the characters sent and received. Must be ONESTOPBIT or TWOSTOPBITS.
Return Value	The return value is the WWCOMDLG constant corresponding to the number of stop bits specified.
Comments	None.

WWVerifyComDlgRev

BOOL WINAPI

WWVerifyComDlgRev(LPSTR *szAppName*,
int *iRequiredRev*,
int FAR **piMajorRev*,
int FAR **piMinorRev*)

This function verifies that the version of WWCOMDLG.DLL installed on the system is at least as new as the specified version. It also returns the major and minor version numbers of the installed WWCOMDLG.DLL to the server. This function is intended for compatibility checking.

Parameter	Description
szAppName	Pointer to a character string containing the server's application name.
iRequiredRev	Minimum required major version number for WWCOMDLG.DLL.
piMajorRev	Pointer to an integer to receive the major version number of WWCOMDLG.DLL.
piMinorRev	Pointer to an integer to receive the minor version number of WWCOMDLG.DLL.
Return Value	TRUE if the installed WWCOMDLG.DLL is compatible with the server. FALSE otherwise.
Comments	The server typically will call this function during initialization in **ProtInit**() to verify that the installed WWCOMDLG.DLL is compatible with the server. This function will display a message box if the DLL is incompatible. It is the server's responsibility to exit if the return value is FALSE.

WWWriteAnyMore

Note: the following functions have been retired, and no longer work:

WWReadVersion(), **WWWriteVersion**(),
WWReadAnyMore(), **WWWriteAnyMore**()

Either replace them with their corresponding UdXXX functions

UdReadVersion(), **UdWriteVersion**(),
UdReadAnyMore(), **UdWriteAnyMore**()

or create user-defined functions to accomplish the same result.

WWWriteVersion

Note: the following functions have been retired, and no longer work:

WWReadVersion(), **WWWriteVersion**(),
WWReadAnyMore(), **WWWriteAnyMore**()

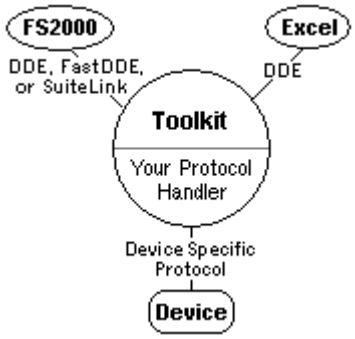
Either replace them with their corresponding UdXXX functions

UdReadVersion(), **UdWriteVersion**(),
UdReadAnyMore(), **UdWriteAnyMore**()

or create user-defined functions to accomplish the same result.

CHAPTER 11

The Chain Manager



This section describes the specifications and usage of the Chain Manager library, a software tool for handling linked lists of any type of data – including mixed types.

Contents

Background

The Chain Manager Library contains a set of software tools for handling linked lists and extensible arrays. In many ways, it is like the container class library of C++ or Java. However, it has been implemented in C and can be used with ordinary C or with C++ and does not require templates. Among the key capabilities provided by the Chain Manager are the following:

- Implementation of doubly-linked lists, with routines for adding items at the head of the list, the tail of the list, or in the middle with or without sorting. No C++ templates are needed, and lists may contain mixed data types.
- Pointers from one item to the next may be either C-type far pointers or unsigned long offsets from a base pointer.
- FindFirst ( ) and FindNext ( ) routines that accept pointers to user routines. These permit easy searches of a list for items satisfying any criteria that the user wishes to implement.
- DeleteList ( ) and DeleteFromList ( ) routines that accept pointers to user routines. These permit easy implementation of clean-up processes (much like destruct methods in C++).
- Implementation of extensible arrays. Allocation and extension routines accept pointers to user routines that perform the actual memory management, permitting allocation on the stack, heap, or anywhere else the user requires.

In many parts of an I/O Server, there are collections of structures that can be implemented as linked lists. Among the most common are the following:

- Lists of Boards, COM Ports, or other configured I/O channels
- Lists of Topics
- Lists of Messages
- Symbol Tables

The process of writing servers can be greatly simplified if one can offload the management of linked lists (which are here called CHAINs) to a set of standard, pre-tested routines.

Typically, there are two basic kinds of data that might be linked into chains:

- Data whose location in memory remains fixed
  Examples: Topic configurations, Board or COM Port definitions, pending messages
- Data whose location might change
  Example: Symbol Tables

The example I/O Server UDSAMPLE includes sample code which allocates a symbol table as an array and links the elements together in two separate lists (with forward and backward pointers): symbol entries **in use** and **unused** symbol entries. The symbol table is extended by reallocating the array to a larger size. This means that the actual location of the array elements in RAM may be different after the extension of the array. The Chain Manager APIs take care of that by giving you a choice as to whether the CHAIN uses pointers or offsets from a base pointer. The distinction is made by leaving a base pointer NULL or setting it to the current base location. All the API calls for handling the CHAIN are identical in both cases. The nice thing about this is that it enables one to manage symbol tables VERY simply using the same set of linked list routines for inserting, finding, unchaining, deleting, etc.

The Chain Manager also includes a set of APIs for handling extensible arrays. Essentially, all you have to do is declare an extensible array structure, and provide the routines for allocating, reallocating, and freeing the memory associated with the array. This allows the programmer to specify whether to allocate from the stack, the heap, etc.

Chain Data Structures

Generally, a linked list can handle structures that have a CHAINLINK as the first part of their structure:

```
typedef struct tagCHAINLINK FAR *LPCHAINLINK;    /* pointer to chainlink */

typedef union tagCHAINLINKPTR       /* chainlink member, pointer or offset */
      {LPCHAINLINK ptr;
       unsigned long offs;
      } CHAINLINKPTR;

typedef struct tagCHAINLINK         /* chainlink structure */
      {CHAINLINKPTR next_item;
        CHAINLINKPTR prev_item;
      } CHAINLINK;
```

Then the general item can have any size or structure the programmer requires, so long as the first structure element is a CHAINLINK:

```
typedef struct tagITEM
      {chain_link
        CHAINLINK;
       ....
      } ITEM;
```

Note that a list may contain items of different types and sizes. In such a mixed list, it is up to the programmer to determine how to identify the type and/or size of any given item. Items may be created on the stack, the heap, etc.

A list of items should be identified by a CHAIN manager structure

```
typedef struct tagCHAIN             /* chain manager structure */
      {CHAINLINKPTR first_item;
       CHAINLINKPTR last_item;
       unsigned long item_count;
       char FAR *base;
      } CHAIN;

typedef CHAIN FAR *LPCHAIN;         /* pointer to chain */
```

If the element ***base*** is NULL, the chain uses pointers for addressing. That is, the forward and backward pointers in a CHAINLINK are pointers, and so are the pointers to first_item and last_item. If the element ***base*** is set to a non-NULL location, the chain uses offsets for addressing.

Setting Up a Chain and Linking Items

```
/*********************************************************************/
/** Initialize a chain
    Clear all pointers, counts to create an empty chain
    Returns TRUE if successful **/
BOOL InitializeChain (LPCHAIN chain);

/*********************************************************************/
/** Set the base pointer for a based chain
    Returns TRUE if successful **/
BOOL SetChainBase (LPCHAIN chain, VOID FAR *new_base);

/*********************************************************************/
/** Insert an item at the head of a chain
    Returns TRUE if successful **/
BOOL InsertItemAtHead (LPCHAIN chain, LPCHAINLINK new_item);

/*********************************************************************/
/** Append an item at the tail of a chain
    Returns TRUE if successful **/
BOOL AppendItemAtTail (LPCHAIN chain, LPCHAINLINK new_item);

/*********************************************************************/
/** Insert an item before a specific point in a chain
    Returns TRUE if successful **/
BOOL InsertItemBefore (LPCHAIN chain,
                       LPCHAINLINK new_item, LPCHAINLINK item);

/*********************************************************************/
/** Insert an item after a specific point in a chain
    Returns TRUE if successful **/
BOOL InsertItemAfter (LPCHAIN chain,
                       LPCHAINLINK new_item, LPCHAINLINK item);

/*********************************************************************/
/** Insert an item in the middle of a chain
    using a comparison routine to keep the chain sorted
    Returns TRUE if successful **/
BOOL InsertItemInMiddle (LPCHAIN chain, LPCHAINLINK new_item,
                       LPCOMPARISONROUTINE compare_routine, BOOL fwd);
```

Note The definition of LPCOMPARISONROUTINE is provided below, in the section on User-Supplied Functions.

Searching For Items in a Chain

Generally, the programmer will search through a CHAIN in order to find a particular item or an item with certain desired properties, do something with that item, and then possibly search for and process other such items in the chain. To facilitate this process, the Chain Manager provides several APIs for setting up a CHAINSCANNER which handles the search. In a sense, this is much like setting up an iterator in C++ or Java; but in practice it is more like using the C function FindFirstFile( ) with a FIND_DATA structure, and continuing with the function FindNextFile( ).

The CHAINSCANNER structure is defined as follows:

```
typedef struct tagCHAINSCANNER      /* chain scanner structure */
    {LPCHAIN chain;                 /* chain being searched */
     CHAINLINKPTR item;             /* last item found, if any */
     BOOL forward;                  /* TRUE if scanning forward */
     void FAR *found_routine;       /* returns TRUE if item found */
     void FAR *comparison_value;    /* structure to use for evaluating criteria */
    } CHAINSCANNER;

typedef CHAINSCANNER FAR *LPCHAINSCANNER;
                                    /* pointer to chain scanner */
```

Note The definition of the ***found_routine*** is described below, in the section on User-Supplied Functions.

```
/*********************************************************************/
/** Check whether item is in indicated chain
    Returns TRUE if item is in chain **/
BOOL IsInChain (LPCHAIN chain, LPCHAINLINK item);

/*********************************************************************/
/** Find first item in chain that satisfies criteria of found routine
    If found_routine is NULL, this routine uses AlwaysFound as a default.
    Search can proceed from head of chain (forward) or from tail (backward)
    Returns pointer if successful **/
LPCHAINLINK FindFirstItem (LPCHAIN chain, BOOL forward,
                           LPFOUNDROUTINE found_routine,
                           void FAR *comparisonValue,
                           LPCHAINSCANNER scanner);

/*********************************************************************/
/** Find first item in chain that satisfies criteria of found routine, beginning with
    indicated item. This allows a search to begin in the middle of a chain, as needed,
    starting with a particular item.
    If found_routine is NULL, this routine uses AlwaysFound as a default.
    Search can proceed from head of chain (forward) or from tail (backward)
    If start_item is NULL, starts with the head or tail of the chain.
    Returns pointer if successful **/
LPCHAINLINK FindItemStartingAt (LPCHAINLINK start_item,
                                LPCHAIN chain, BOOL forward,
                                LPFOUNDROUTINE found_routine,
                                void FAR *comparisonValue,
                                LPCHAINSCANNER scanner);
```

```
/*********************************************************************/
/** Find first item in chain that satisfies criteria of found routine, beginning with item
    that follows indicated item.  This allows a search to begin in the middle of a chain,
    as needed, starting just after a particular item.
    If found_routine is NULL, this routine uses AlwaysFound as a default.
    Search can proceed from head of chain (forward) or from tail (backward)
    If start_item is NULL, starts with the head or tail of the chain.
    Returns pointer if successful **/
LPCHAINLINK FindItemFollowing (LPCHAINLINK start_item,
                               LPCHAIN chain, BOOL forward,
                               LPFOUNDROUTINE found_routine,
                               void FAR *comparisonValue,
                               LPCHAINSCANNER scanner);

/*********************************************************************/
/** Find next item in chain that satisfies criteria
    Returns pointer if successful **/
LPCHAINLINK FindNextItem (LPCHAINSCANNER scanner);

/*********************************************************************/
/** Get next item that scanner would examine.
    Note that this does not check whether the item satisfies the criteria.
    Also, it does not advance the scan pointer. **/
LPCHAINLINK GetScannerNextItem (LPCHAINSCANNER scanner);
```

Removing Items From a Chain

```
/*************************************************************************/
/** Remove item from chain
    Returns pointer if successful **/
BOOL UnchainItem (LPCHAIN chain, LPCHAINLINK item);

/*************************************************************************/
/** Perform indicated delete operation on item and remove it from chain
    If delete operation is unsuccessful, item is NOT removed from chain
    Returns TRUE if successful **/
BOOL DeleteItem (LPCHAIN chain,
                 LPCHAINLINK item, DELETEROUTINE delete_routine);

/*************************************************************************/
/** Perform indicated delete operation on all items in chain and remove them from
    the chain
    If delete operation is unsuccessful on any item, operation stops and the pointer
    to the item is returned.
    If all items are deleted successfully, NULL is returned **/
LPCHAINLINK DeleteChain (LPCHAIN chain, DELETEROUTINE delete_routine);
```

Note The definition of the *delete_routine* is described below, in the section on User-Supplied Functions.

User-Supplied Chain Item Functions

For the search, comparison, and delete operations, the user must provide a routine to perform the appropriate action. The following definitions are used:

```
typedef BOOL FAR FOUNDROUTINE (LPCHAINLINK item,
                               void FAR *comparisonValue);
        /** result = TRUE if item satisfies criterion **/

typedef FOUNDROUTINE FAR *LPFOUNDROUTINE;
```

The pointers to ***item*** and to ***comparisonValue*** are of types LPCHAINLINK and void FAR *, respectively, so that they can match any structures the programmer defines. Inside the function, the programmer is responsible for casting ***item*** to a pointer to the appropriate structure type and for defining the structure of the ***comparisonValue*** – which can be anything from a simple basic type to something more complicated. The programmer is also responsible for defining what portions of the structures need to be compared and how to determine whether a match has been found.

```
typedef int FAR COMPARISONROUTINE (LPCHAINLINK item,
                                   LPCHAINLINK new_item);
        /** result = -1 if item < new_item
                   =  0 if item = new_item
                   = +1 if item > new_item **/

typedef COMPARISONROUTINE FAR *LPCOMPARISONROUTINE;
```

The pointers to ***item*** and ***new_item*** are of type LPCHAINLINK so they can match any structures the programmer defines. As with the FOUNDROUTINE, the programmer is responsible for casting ***item*** and ***new_item*** to the appropriate type(s) and for defining how the comparison will be performed.

```
typedef BOOL FAR DELETEROUTINE (LPCHAINLINK item);
        /** result = TRUE if routine successful **/

typedef DELETEROUTINE FAR *LPDELETEROUTINE;
```

The pointer to ***item*** is of type LPCHAINLINK so that it can match any structure the programmer defines. Inside the comparison function, the programmer is responsible for casting item to the appropriate structure type, for performing the clean-up, and for determining whether the clean-up was successful.

Also provided are two defaults, in case the user just wants to pass NULL for the pointer to the found routine or to the delete routine:

```
/** default found routine -- always returns TRUE **/
BOOL FAR AlwaysFound (LPCHAINLINK item, void FAR *comparisonValue);

/** default delete routine -- does nothing, always returns TRUE **/
BOOL FAR AlwaysDeleted (LPCHAINLINK item);
```

Extensible Array Data Structures

An extensible array structure identifies where the start of the array is located, how it is to be allocated, and how it is to be extended.

```
typedef struct tagEXTARRAY
      {void FAR *first_member;          /* location of first member */
       unsigned long member_count;      /* number of members in array */
       unsigned long member_size;       /* number of bytes per member */
       unsigned long init_count;        /* number of members to allocate first time */
       unsigned long extension_count;   /* number of members to add when
                                           extending */
       void FAR *alloc_routine;         /* pointer to allocation routine */
       void FAR *extend_routine;        /* pointer to extension routine */
       void FAR *delete_routine;        /* pointer to delete routine */
      } EXTARRAY;
typedef EXTARRAY FAR *LPEXTARRAY;

typedef LPVOID FAR ALLOCARRAYROUTINE (LPEXTARRAY array);
typedef ALLOCARRAYROUTINE FAR *LPALLOCARRAYROUTINE;

typedef LPVOID FAR EXTENDARRAYROUTINE (LPEXTARRAY array);
typedef EXTENDARRAYROUTINE FAR *LPEXTENDARRAYROUTINE;

typedef BOOL FAR DELETEARRAYROUTINE (LPEXTARRAY array);
            /** result = TRUE if routine successful **/
typedef DELETEARRAYROUTINE FAR *LPDELETEARRAYROUTINE;
```

Allocating, Extending, and Deleting an Extensible Array

```
/*********************************************************************/
/** Initialize an extensible array:
    Set pointers to functions to allocate, extend, and delete array,
    Clear all pointers, counts to create an empty array
    Returns TRUE if successful **/
BOOL InitializeExtArray (LPEXTARRAY array,
                unsigned long initCount,
                unsigned long memberSize,
                unsigned long extCount,
                LPALLOCARRAYROUTINE allocRoutine,
                LPEXTENDARRAYROUTINE extendRoutine,
                LPDELETEARRAYROUTINE deleteRoutine);

/*********************************************************************/
/** Allocate an extensible array:
    Set all pointers, counts, and attempt to allocate memory
    Returns pointer to first array member if successful **/
LPVOID AllocExtArray (LPEXTARRAY array);

/*********************************************************************/
/** Extend an extensible array
    Update all pointers, counts, and attempt to allocate memory
    Returns TRUE if successful **/
LPVOID ExtendExtArray (LPEXTARRAY array);
```

Note: If ExtendExtArray ( ) is called before AllocExtArray ( ) has been called, the software performs AllocExtArray ( ) instead, ensuring that the array is allocated, if possible.

```
/*********************************************************************/
/** Delete an extensible array
    Returns TRUE if successful **/
BOOL DeleteExtArray (LPEXTARRAY array);

/*********************************************************************/
/** Get pointer to indicated member of extensible array
    Returns pointer if valid, NULL if invalid or out of range **/
void FAR *GetExtArrayMemberPtr (LPEXTARRAY array, unsigned long index);
```

Note: The programmer is responsible for supplying routines to allocate, extend, and delete the array. Examples are provided below for doing this on the heap.

Examples of Usage

Memory Management for Extensible Arrays

```
/*********************************************************/
/** Allocate an extensible array on the heap,
    Returns pointer to first member if successful **/

LPHVOID FAR AllocateHeapArray (LPEXTARRAY array)
{
    unsigned long newSize;
    LPHVOID firstPtr;

    /* initialize return value */
    firstPtr = NULL;

    if (array != NULL)
    {
        /* attempt to allocate initial array */
        if ((array->member_size > 0) &&
            (array->init_count > 0))
        {
            newSize = array->init_count * array->member_size;
            firstPtr = wwHeap_AllocPtr (hHeap,
                                GMEM_MOVEABLE | GMEM_ZEROINIT,
                                newSize);
        }
    }

#ifdef TRACE_HEAP_ARRAY
    if (firstPtr)
        debug ("AllocateHeapArray successful" endln);
    else
        debug ("AllocateHeapArray failed" endln);
#endif
    /* return pointer, if successful */
    return (firstPtr);
} /* AllocateHeapArray */

/*********************************************************/
/** Extend an extensible array on the heap,
    Returns new pointer to first member if successful **/

LPHVOID FAR ExtendHeapArray (LPEXTARRAY array)
{
    unsigned long newCount;
    unsigned long newSize;
    LPHVOID newFirst;

    /* initialize return value */
    newFirst = NULL;

    if (array != NULL)
    {
        /* if array is extensible, attempt to reallocate */
        if ((array->first_member != NULL) &&
            (array->member_size > 0) &&
            (array->extension_count > 0))
```

```
{
            newCount = array->member_count +
                       array->extension_count;
            newSize = newCount * array->member_size;
            newFirst = wwHeap_ReAllocPtr (hHeap,
                            array->first_member,
                            GMEM_MOVEABLE | GMEM_ZEROINIT,
                            newSize);
        }
    }
    /* return pointer, if successful */
    return (newFirst);
} /* ExtendHeapArray */

/*********************************************************/
/** Delete an extensible array on the heap,
    Returns TRUE if successful **/

BOOL FAR DeleteHeapArray (LPEXTARRAY array)
{
    BOOL status;

    /* initialize return value */
    status = FALSE;

    if (array != NULL)
    {
        if (array->first_member != NULL)
        {
            /* free the memory used for the array */
            wwHeap_FreePtr (hHeap, array->first_member);
            status = TRUE;
        }
    }
    /* indicate success or failure */
    return (status);
} /* DeleteHeapArray */
```

Handling Linked Lists

The examples below use a generic item of the following form:

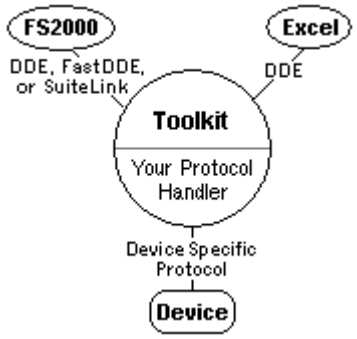
```
typedef struct tagITEM
    {CHAINLINK chain_link;
     int    address;
     char  name[20+1];
     int    contents;
     int    flags;
    } ITEM;
```

```
/*********************************************************/
/** Allocate a new ITEM with indicated properties,
    Return pointer to item if successful, NULL otherwise **/
ITEM FAR *AllocItem (int new_addr, char *new_name,
                     int new_contents, int new_flags)
{
    ITEM FAR *lpItem;

    /* attempt to allocate new item on heap */
    lpItem = wwHeap_AllocPtr (hHeap,
                          GMEM_MOVEABLE | GMEM_ZEROINIT,
                          sizeof(ITEM));
    if (lpItem)
    {
        /* item successfully created, initialize it */
        lpItem->address = new_addr;
        lstrcpy (lpItem->name, new_name);
        lpItem->contents = new_contents;
        lpItem->flags = new_flags;
    }
    /* return pointer to item or NULL */
    return (lpItem);
} /* AllocItem */

/*********************************************************/
/** Destroy indicated ITEM,
    Return TRUE if successful **/
BOOL FAR DeleteItem (LPCHAINLINK lpItem)
{
    /* free item from heap */
    wwHeap_FreePtr (hHeap, lpItem);
    /* indicate successful */
    return (TRUE);
} /* DeleteItem */
```

```
/*************************************************************/
/** Check whether item has indicated name,
    Return TRUE if match found **/
BOOL FAR StringFound (LPCHAINLINK lpChain_link,
                      void FAR *lpComparisonValue)
{
    BOOL status;
    char FAR *lpSt;
    ITEM FAR *lpItem;

    /* initialize return status */
    status = FALSE;
    /* cast parameters as appropriate structures */
    lpItem = (ITEM FAR *) lpChain_link;
    lpSt = (char FAR *) lpComparisonValue;
    /* check whether match was found, return indicator */
    if (lstrcmpi (lpItem->name, lpSt) >= 0)
        status = TRUE;
    /* indicate whether match was found */
    return (status);
} /* StringFound */

/*************************************************************/
/** Compare indicated item1 to item2,
    Return –1 if <, 0 if =, +1 if > */ if item1 < item2 **/
int FAR StringCompare (LPCHAINLINK lpChain_link1,
                       LPCHAINLINK lpChain_link2)
{
    int retval;
    ITEM FAR *lpItem1, FAR *lpItem2;

    /* cast parameters as appropriate structures */
    lpItem1 = (ITEM FAR *) lpChain_link1;
    lpItem2 = (ITEM FAR *) lpChain_link2;
    /* compare items, set return value */
    retval = lstrcmpi (lpItem1->name, lpItem2->name);
    if (retval > 0)
        retval = 1;
    if (retval < 0)
        retval = -1;
    return (retval);
} /* StringCompare */
```

```
/* Examples of chain operations */

CHAIN people;
ITEM FAR *lpItem;
CHAINSCANNER item_scanner;
char search_name[21];

/* initialize the list of people */
InitializeChain (&people);

/* allocate several entries and add to list;
   note that the list ends up sorted,
   but we're not explictly sorting, here */
lpItem = AllocItem (3579, "Fred", 7, 0xF3);
AppendItemAtTail (&people, lpItem);
lpItem = AllocItem (2468, "Jane", 2, 0x0B);
AppendItemAtTail (&people, lpItem);
lpItem = AllocItem (2473, "Jane", 4, 0x37);
AppendItemAtTail (&people, lpItem);
lpItem = AllocItem (1234, "Joe",  5, 0x00);
AppendItemAtTail (&people, lpItem);

/* find all items with indicated name */
strcpy (search_name, "Jane");
lpItem = FindFirstItem (&people, SCAN_FROM_HEAD,
                        StringFound, (LPSTR) search_name,
                        &item_scanner);
while (lpItem) {
    /* display item and address */
    debug ("Item found with name %Fs has address %d",
           (LPSTR) lpItem->name, (int) lpItem->address);
    /* find next item, if any */
    lpItem = FindNextItem (&item_scanner);
}

/* create new item and insert in sorted list */
lpItem = AllocItem (9742, "George", 3, 0x77);
InsertItemInMiddle (&people, lpItem,
                    StringCompare, SCAN_FROM_HEAD);

/* delete certain items from list, one by one */
lpItem = FindFirstItem (&people, SCAN_FROM_HEAD,
                        NULL, NULL, &item_scanner);
while (lpItem) {
    if (lpItem->flags >= 0x50) {
        UnchainItem (&people, (LPCHAINLINK) lpItem);
        wwHeap_FreePtr (hHeap, lpItem);
    }
    /* find next item, if any */
    lpItem = FindNextItem (&item_scanner);
}

/* delete all items in list */
lpItem = DeleteChain (&people, DeleteItem);
```

CHAPTER 12



I/O Server Toolkit Data Structures

The **I/O Server Toolkit** contains several structures associated with the API functions. These structures are defined in alphabetic order in this chapter.

Contents

Data Structure Definitions

PTVALUE

#include "protypes.h"

typedef union {

DISC *disc*;
INTG *intg*;
REAL *real*;
WHMEM *hString*;

} PTVALUE;

The PTVALUE union contains a value to be passed between the I/O Server Toolkit and the server. The element of the union which contains the value is determined by the point's type.

Element	Description
disc	Byte containing value for a discrete point.
intg	Long-word containing value for an integer point.
real	Float containing value for a real point.
hString	Memory handle for string value.
Comments	Do not manipulate the *hString* value directly. Use the string manipulation functions **StrValSetString**(), **StrValSetNString**(), **StrValStringLock**(), **StrValStringUnlock**(), and **StrValStringFree**().

WW_AB_INFO

```
#include "wwcomdlg.h"

typedef struct tagWW_AB_INFO {
                    HWND hwndOwner;
                    char far *szDriverName;
                    char far *szId;
                    char far *szVersion;
                    char far *szCopyright;
                    HICON hIcon;
                    char *szComment;
                    char reserved[12];
} WW_AB_INFO, FAR *LPWW_AB_INFO;
```

The WW_AB_INFO structure is used to configure the Wonderware About box dialog invoked by a call to **WWDisplayAboutBox**() or **WWDisplayAboutBoxEx**().

Element	Description
hwndOwner	Handle of window which owns the dialog box.
szDriverName	Pointer to a string containing the server name.
szId	Pointer to a string describing the purpose of the server. For example, "GE Fanuc Series 90 Protocol".
szVersion	Pointer to a string containing the version number for the server. For example, "5.1".
szCopyright	Pointer to a string containing the year for the copyright information. For example, "1995".
hIcon	Handle of server icon to be displayed within the dialog box.
szComment	Pointer to a string containing a comment to be displayed in the dialog box.
reserved[12]	Reserved buffer space. Currently unused, but should be initialized to 0 by caller.
Comments	None.

WW_CONFIRM

#include "wwcomdlg.h"

typedef struct tagWW_CONFIRM {
HWND *hwndOwner*;
char far **szCfgPath*;
int *iSizeOfszCfgPath*;
char far **szDriverName*;
char *reserved[16];*

} WW_CONFIRM, FAR *LPWW_CONFIRM;

The WW_CONFIRM structure is used to configure the Save Configuration dialog invoked by a call to **WWConfirm**(). This dialog is used to confirm the location for saving server configuration information.

Element	**Description**
hwndOwner	Handle of window which owns the dialog box.
szCfgPath	Points to a string which contains the path where the configuration file is to be saved. Changes typed by the user will be placed in this buffer when **WWConfirm**() returns.
iSizeofszCfgPath	Maximum allowed length for configuration file path name.
szDriverName	Pointer to string containing server name.
reserved[16]	Reserved buffer space. Currently unused, but should be initialized to 0 by caller.
Comments	None.

WW_CP_DLG_LABELS

```
#include "wwcomdlg.h"

typedef struct tagWW_CP_DLG_LABELS {
                    HWND hwndOwner;
                    char far *szDriverName;
                    LPWW_CP_PARAMS lpDefaultCpParams;
                    WW_CP_PARAMS lpCpParams;
                    int iNumPorts;
                    BOOL bAllowBaud110;
                    BOOL bAllowBaud300;
                    BOOL bAllowBaud600;
                    BOOL bAllowBaud1200;
                    BOOL bAllowBaud2400;
                    BOOL bAllowBaud4800;
                    BOOL bAllowBaud9600;
                    BOOL bAllowBaud14400;
                    BOOL bAllowBaud19200;
                    BOOL bAllowBaud38400;
                    BOOL bAllowBaud56000;     /* not used at this time */
                    BOOL bAllowBaud57600;     /* not used at this time */
                    BOOL bAllowBaud115200;    /* not used at this time */
                    BOOL bAllowBaud128000;    /* not used at this time */
                    BOOL bAllowDatabits7;
                    BOOL bAllowDatabits8;
                    BOOL bAllowStopbits1;
                    BOOL bAllowStopbits2;
                    BOOL bAllowParityEven;
                    BOOL bAllowParityOdd;
                    BOOL bAllowParityNone;
                    BOOL bAllowParityMark;
                    BOOL bAllowParitySpace;
                    char far *szCustom1BoxLabel;
                    char far *szCustom1Radio1Label;
                    char far *szCustom1Radio2Label;
                    struct tagWW_CP_DLG_LABELS FAR *lpRadio2DlgLabels;
                    char far *szCustom2BoxLabel;
                    char far *szCustom2Radio1Label;
                    char far *szCustom2Radio2Label;
                    char far *szCustom3BoxLabel;
                    char far *szCustom3Radio1Label;
                    char far *szCustom3Radio2Label;
                    char far *szCheck1Label;
                    char far *szCheck2Label;
                    char far *szCustomEditLabel;
                    UINT uCustomEditBase;
                    UINT uCustomEditLowLimit;
                    UINT uCustomEditHighLimit;
                    FARPROC lpfnConfigureSave;
                    int iInternalUseOnly;
                    char reserved[16];
} WW_CP_DLG_LABELS, FAR *LPWW_CP_DLG_LABELS;
```

The WW_CP_DLG_LABELS structure is used with the **WWConfigureComPort**() function to control the serial port configuration dialog. This structure contains numerous boolean flags used to enable or disable various radio buttons or entire group boxes. The convention for group boxes and labeled controls is that if the label field is NULL, the group box and its contents are hidden from view.

Element	Description
hwndOwner	Handle of window which owns the dialog box.
szDriverName	Pointer to string containing server name.
lpDefaultCpParams	Pointer to a **WW_CP_PARAMS** structure which contains the default port parameter settings. These settings will be applied to the current selection when the "Defaults" button is selected. This structure should be initialized by the caller.
lpCpParams	Pointer to array of **WW_CP_PARAMS** structures. This array contains the initial and final settings for all communications ports. There is one entry for each communications port in the system.
iNumPorts	Integer value indicating total number of entries in the *lpCpParams* array.
bAllowBaud110	Boolean value indicating whether the radio button for 110 baud should be enabled. A value of TRUE will enable the button.
bAllowBaud300	Boolean value indicating whether the radio button for 300 baud should be enabled. A value of TRUE will enable the button.
bAllowBaud600	Boolean value indicating whether the radio button for 600 baud should be enabled. A value of TRUE will enable the button.
bAllowBaud1200	Boolean value indicating whether the radio button for 1200 baud should be enabled. A value of TRUE will enable the button.
bAllowBaud2400	Boolean value indicating whether the radio button for 2400 baud should be enabled. A value of TRUE will enable the button.
bAllowBaud4800	Boolean value indicating whether the radio button for 4800 baud should be enabled. A value of TRUE will enable the button.
bAllowBaud9600	Boolean value indicating whether the radio button for 9600 baud should be enabled. A value of TRUE will enable the button.
bAllowBaud14400	Boolean value indicating whether the radio button for 14400 baud should be enabled. A value of TRUE will enable the button.

Element	Description
bAllowBaud19200	Boolean value indicating whether the radio button for 19200 baud should be enabled. A value of TRUE will enable the button.
bAllowBaud38400	Boolean value indicating whether the radio button for 38400 baud should be enabled. A value of TRUE will enable the button.
bAllowBaud56000	Not supported at this time.
bAllowBaud57600	Not supported at this time.
bAllowBaud115200	Not supported at this time.
bAllowBaud128000	Not supported at this time.
bAllowDatabits7	Boolean value indicating whether the radio button for 7 data bits should be enabled. A value of TRUE will enable the button.
bAllowDatabits8	Boolean value indicating whether the radio button for 8 data bits should be enabled. A value of TRUE will enable the button.
bAllowStopbits1	Boolean value indicating whether the radio button for 1 stop bit should be enabled. A value of TRUE will enable the button.
bAllowStopbits2	Boolean value indicating whether the radio button for 2 stop bits should be enabled. A value of TRUE will enable the button.
bAllowParityEven	Boolean value indicating whether the radio button for even parity should be enabled. A value of TRUE will enable the button.
bAllowParityOdd	Boolean value indicating whether the radio button for odd parity should be enabled. A value of TRUE will enable the button.
bAllowParityNone	Boolean value indicating whether the radio button for no parity should be enabled. A value of TRUE will enable the button.
bAllowParityMark	Boolean value indicating whether the radio button for mark parity should be enabled. A value of TRUE will enable the button.
bAllowParitySpace	Boolean value indicating whether the radio button for space parity should be enabled. A value of TRUE will enable the button.
szCustom1BoxLabel	Pointer to character string for the first custom group box label. If NULL, no first custom group box will be displayed.
szCustom1Radio1Label	Pointer to character string for first radio button in first custom group box. Only used if *szCustom1BoxLabel* is not NULL.

Element	Description
szCustom1Radio2Label	Pointer to character string for second radio button in first custom group box. Only used if *szCustom1BoxLabel* is not NULL.
lpRadio2DlgLabels	Currently unused.
szCustom2BoxLabel	Pointer to character string for the second custom group box label. If NULL, no second custom group box will be displayed.
szCustom2Radio1Label	Pointer to character string for first radio button in second custom group box. Only used if *szCustom2BoxLabel* is not NULL.
szCustom2Radio2Label	Pointer to character string for second radio button in second custom group box. Only used if *szCustom2BoxLabel* is not NULL.
szCustom3BoxLabel	Pointer to character string for the third custom group box label. If NULL, no third custom group box will be displayed.
szCustom3Radio1Label	Pointer to character string for first radio button in third custom group box. Only used if *szCustom3BoxLabel* is not NULL.
szCustom3Radio2Label	Pointer to character string for second radio button in third custom group box. Only used if *szCustom3BoxLabel* is not NULL.
szCheck1Label	Pointer to character string for the first custom check box label. If NULL, no first custom check box will be displayed.
szCheck2Label	Pointer to character string for the second custom check box label. If NULL, no second custom check box will be displayed.
szCustomEditLabel	Pointer to character string containing the label for the custom edit control. If NULL, custom edit control will be displayed.
uCustomEditBase	Indicates radix of the number system for custom edit control. For example, 10 for decimal, 16 for hex, etc.
uCustomEditLowLimit	Indicates lowest allowed number to be entered for custom edit control.
uCustomEditHighLimit	Indicates highest allowed number to be entered for custom edit control.
lpfnConfigureSave	Points to a function which will be called when the port settings are to be saved. This function will be called when the "Save" button is selected or when the port settings have been modified and the user selects a new port. No arguments are passed to this function. This function is expected to write all configuration information contained in the *lpCpParams* array.

Element	Description
iInternalUseOnly	Reserved for internal use. Do not use.
reserved[16]	Reserved buffer space. Currently unused, but should be initialized to 0 by caller.
Comments	None.

WW_CP_PARAMS

```
#include "wwcomdlg.h"

typedef struct tagWW_CP_PARAMS {
                    unsigned long uBaud;
                    unsigned long uDataBits;
                    unsigned long uStopBits;
                    unsigned long uParity;
                    unsigned long uReplyTimeout;
                    unsigned long uCustomEdit;
                    short bCustom1Radio;
                    short bCustom2Radio;
                    short bCustom3Radio;
                    short bCheck1;
                    short bCheck2;
                    char reserved[30];   /* pad to 64 bytes */
} WW_CP_PARAMS, FAR *LPWW_CP_PARAMS;
```

The WW_CP_PARAMS structure contains the configuration settings for a serial communications port.

Element	Description
uBaud	Contains the WWCOMDLG constant indicating the baud rate setting.
uDataBits	Contains the WWCOMDLG constant indicating the number of data bits setting.
uStopBits	Contains the WWCOMDLG constant indicating the number of stop bits setting.
uParity	Contains the WWCOMDLG constant indicating the parity setting.
uReplyTimeout	Contains the reply timeout in seconds.
uCustomEdit	Contains the setting for the custom edit control.
bCustom1Radio	Boolean flag indicating which radio button in the first custom group box is selected. A value of TRUE indicates that the first radio button is selected. A value of FALSE indicates that the second radio button is selected.
bCustom2Radio	Boolean flag indicating which radio button in the second custom group box is selected. A value of TRUE indicates that the first radio button is selected. A value of FALSE indicates that the second radio button is selected.

Element	Description
bCustom3Radio	Boolean flag indicating which radio button in the third custom group box is selected. A value of TRUE indicates that the first radio button is selected. A value of FALSE indicates that the second radio button is selected.
bCheck1	Boolean flag indicating whether the first custom checkbox is selected. A value of TRUE indicates that it is selected.
bCheck2	Boolean flag indicating whether the second custom checkbox is selected. A value of TRUE indicates that it is selected.
reserved[30]	Reserved buffer space. Currently unused, but should be initialized to 0 by caller.
Comments	None.

WW_SELECT

```
#include "wwcomdlg.h"

typedef struct tagWW_SELECT {
                        HWND hwndOwner;
                        char far *szTitle;
                        char far *szGroupBoxLabel;
                        GETLISTHEADPROC lpfnGetListHead;
                        GETNEXTNODEPROC lpfnGetNextNode;
                        GETNODENAMEPROC lpfnGetNodeName;
                        ADDNODEPROC lpfnAddNode;
                        CONFIGNODEPROC  lpfnConfigNode;
                        DELETENODEPROC lpfnDeleteNode;
                        BOOL bAddDeleteModifyEnabled;
                        unsigned char bDoNotConfirmDeletes;
                        char reserved[15];
} WW_SELECT, FAR *LPWW_SELECT;
```

The WW_SELECT structure is used with the **WWSelect**() function to populate and manipulate a list box of items. This list box is typically used for presenting a list of I/O topics. Some servers also use this listbox for presenting a list of interface boards for perusal. The dialog box associated with the list contains buttons for adding, modifying, and deleting entries, as well as the standard listbox traversal operations (arrow keys, etc.)

Element	Description
hwndOwner	Handle of window which owns the dialog box.
szTitle	Pointer to a string to be placed in the title bar of the dialog.
szGroupBoxLabel	Pointer to a string to be used to label the list box.
lpfnGetListHead	Pointer to a function which will be called to obtain a pointer to the head entry of the list.
lpfnGetNextNode	Pointer to a function which will be called to obtain the successor to a given list entry.
lpfnGetNodeName	Pointer to a function which will be called to obtain the name of a given list entry.
lpfnAddNode	Pointer to a function which will be called to add a new entry to the list. This function is called in response to the "New..." button and will most likely put up an application dialog box for collecting information regarding the new item.

Element	Description
lpfnConfigNode	Pointer to a function which will be called to modify an existing entry in the list. This function is called in response to the "Modify..." button and will most likely put up an application dialog box for collecting information regarding the item.
lpfnDeleteNode	Pointer to a function which will be called to delete an existing entry in the list. This function is called in response to the "Delete" button. It is up to the application to determine if the delete operation is allowed. For example, you would not want to delete an adapter card which is in use by any topic.
bAddDeleteModifyEnabled	Boolean value indicating whether the Add and Delete dialog buttons will be enabled. If TRUE, these buttons will be enabled. This parameter will be passed as a parameter to the *lpfnConfigNode* callback function.
bDoNotConfirmDeletes	Boolean value indicating whether a delete confirmation dialog should be automatically displayed. If this value is FALSE, the delete confirmation dialog will be displayed.
reserved[15]	Reserved buffer space. Currently unused, but should be initialized to 0 by caller.
Comments	None.

WW_SERV_PARAMS

```
#include "wwcomdlg.h"

typedef struct tagWW_SERV_PARAMS {
                        HWND hwndOwner;
                        char far *szCfgPath;
                        int iSizeOfszCfgPath;
                        char far *szDriverName;
                        BOOL bIndefWriteRetrySupported;
                        BOOL bIndefWriteRetry;
                        BYTE bPreventChanges;
                        BYTE bNotService;
                        BYTE bCFGfileUnused;
                        BYTE nNTServiceSetting;
                        char far *szCaption;
                        char reserved[8];
} WW_SERV_PARAMS, FAR *LPWW_SERV_PARAMS;
```

The WW_SERV_PARAMS structure is used with the **WWConfigureServer**() function to initialize and manipulate the "Server Settings" dialog. This dialog is used to view and set server settings such as configuration file path and protocol timer tick.

Element	Description
hwndOwner	Handle of window which owns the dialog box.
szCfgPath	Pointer to string containing the configuration file path name.
iSizeOfszCfgPath	Maximum allowed length for configuration file path name.
szDriverName	Pointer to string containing server name.
bIndefWriteRetrySupported	Boolean value indicating whether the server supports indefinite retries of failed writes. If TRUE, this option will be provided to the user.
bIndefWriteRetry	Boolean value indicating whether the server should retry failed writes indefinitely. Only applies if the *bIndefWriteRetrySupported* flag is TRUE.
bPreventChanges	Boolean value indicating whether modifications should be disallowed. If TRUE, the OK button will be disabled, making the dialog read-only.
bNotService	Boolean value indicating whether server can be configured as an NT service. If TRUE, checking the “Run automatically as NT service” checkbox will generate an error message.
bCFGfileUnused	Boolean value indicating whether a configuration file is used. If TRUE, the “Configuration File Directory” edit box will be disabled.
nNTServiceSetting	Integer value, combining several flags. This value is returned as a result from **WWConfigureServer**() to indicate whether changes have been made for running the server as an NT service and with what success.
szCaption	Pointer to a string that is used as the title caption for the dialog. If NULL, the default caption is used.

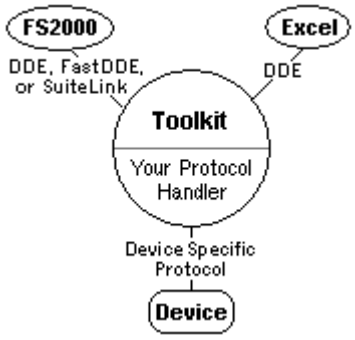
reserved[8] Reserved buffer space. Currently unused, but should be initialized to 0 by caller.

Comments None.

The value returned in *nNTServiceSetting* is an OR of the following bits:

```
#define WW_NTSERVICE_IS_SERVICE   (0x01)  /* set to run as NT service */
#define WW_NTSERVICE_CHANGED      (0x02)  /* setting was changed */
#define WW_NTSERVICE_ERROR        (0x04)  /* unable to establish new
                                             service settings */
```

CHAPTER 13



Common Dialogs

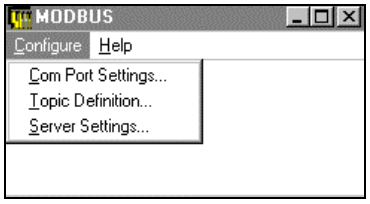
This chapter describes usage of WWDLG32A, a Windows 98/NT/2000 Dynamic Link Library (DLL), which provides a common look and feel throughout the I/O Server family.

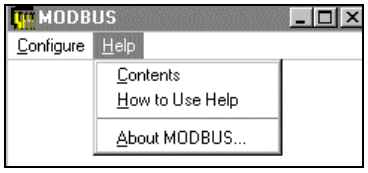
Contents

Main Menu

Serial Servers

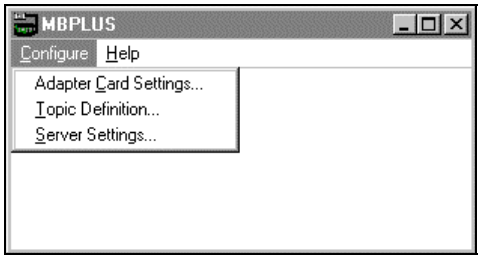
Serial servers should use the following standardized main menus.



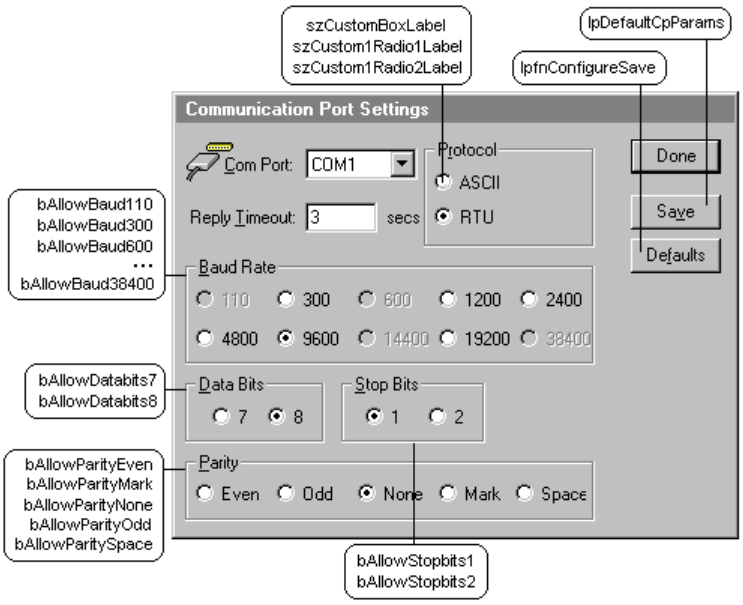


Board-based Servers

Board-based servers are nearly identical, with the exception of the wording on the interface settings menu item:



Com Port Settings



Serial port configuration is stored in a **WW_CP_PARAMS** structure. The Wonderware servers read and write this structure directly to a configuration file. In addition, the configuration dialog used by **WWConfigureComPort**() uses this structure to communicate settings with the caller.

```
typedef struct tagWW_CP_PARAMS {
    unsigned long   uBaud;
    unsigned long   uDataBits;
    unsigned long   uStopBits;
    unsigned long   uParity;
    unsigned long   uReplyTimeout;
    unsigned long   uCustomEdit;
    short           bCustom1Radio;
    short           bCustom2Radio;
    short           bCustom3Radio;
    short           bCheck1;
    short           bCheck2;
    char            reserved[30];   /* pad to 64 bytes */
} WW_CP_PARAMS, FAR *LPWW_CP_PARAMS;
```

The values stored in the fields are generally dialog control IDs. Translation functions are provided to map Windows constants like CBR_9600, etc., to the proper values. The following translators are for mapping the Windows constants for word size, stop bits, parity, and baud rate to the **WW_CP_PARAMS** equivalents:

```
UINT WINAPI WWTranslateWinDataToCDlg(UINT);

UINT WINAPI WWTranslateWinStopToCDlg(UINT);

UINT WINAPI WWTranslateWinParityToCDlg(UINT);
UINT WINAPI WWTranslateWinBaudToCDlg(UINT);
```

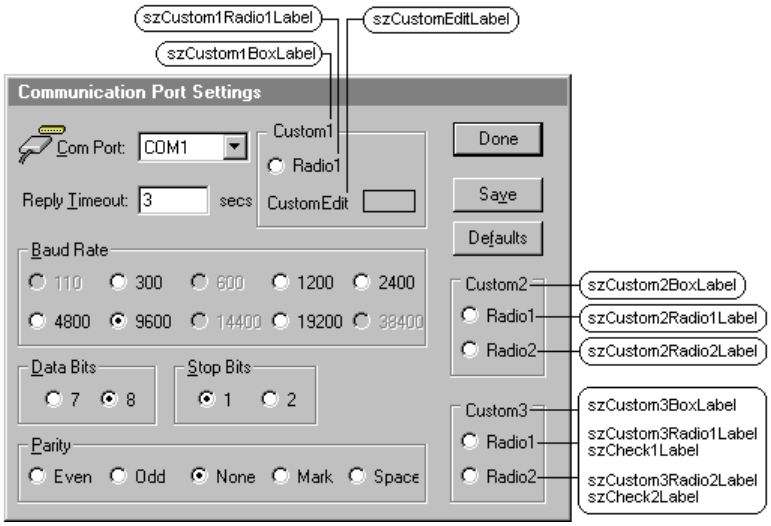
The following translators are for mapping the **WW_CP_PARAMS** constants for word size, stop bits, parity, and baud rate:

```
UINT WINAPI WWTranslateCDlgToWinData(UINT);
UINT WINAPI WWTranslateCDlgToWinStop(UINT);
UINT WINAPI WWTranslateCDlgToWinParity(UINT);
UINT WINAPI WWTranslateCDlgToWinBaud(UINT);
```

The custom fields are for parameters which are peculiar to the particular type of serial server. MODBUS, for instance, uses these fields to indicate whether the communication port is connected to an ASCII- or an RTU- speaking host.

The appearance of the serial port configuration dialog is controlled by the **WW_CP_DLG_LABELS** structure. This structure contains numerous boolean flags used to enable or disable certain radio buttons, e.g. *bAllowParityMark*, or entire group boxes, e.g. *szCustom1BoxLabel*. The convention for group boxes and labeled controls is that if the label field is NULL, the group box and its contents are hidden from view.

The following illustration shows the locations of the custom group boxes. Some fields within the group boxes are mutually exclusive, and are therefore obscured in the picture.



The initial settings for all ports are passed in the *lpCpParams*[] array in the **WW_CP_DLG_LABELS** structure. This array of **WW_CP_PARMS** structures has one entry for each communication port in the system. There should be one entry in the *lpCpParams* array for each entry in the "Com Port" list box. The size of the *lpCpParams* array is indicated in the *iNumPorts* field. Changes to port configuration are placed in the appropriate *lpCpParams* entry for the port selected by the list box.

The "Defaults" button copies the settings from the default port parameters structure *lpDefaultCpParams*. This structure should be initialized appropriately.

Pressing the "Save" button invokes the *lpfnConfigureSave* callback function. Likewise, if the port settings have been modified and the user selects a new port in the "Com Port" list box, the *lpfnConfigureSave* callback is invoked after confirmation from the user. There are no arguments to *lpfnConfigureSave*. The save function is expected to write all configuration information in the *lpCpParams* array.

The CustomEdit control is intended for entering a numeric value. The *uCustomEditBase* field determines the radix for the value (e.g. uCustomEditBase of 16 will allow hex numbers to be entered). The *uCustomEditLowLimit* and *uCustomEditHighLimit* are used by **WWConfigureComPort** to validate the entry. Values below the LowLimit or above the HighLimit will be rejected (with a message box.)

The **WW_CP_DLG_LABELS** structure is shown below in its entirety.

```
typedef struct tagWW_CP_DLG_LABELS {
    char far       *szC
    HWND            hwndOwner;
    char far       *szDriverName;
    LPWW_CP_PARAMS  lpDefaultCpParams;
    LPWW_CP_PARAMS  lpCpParams;
    int             iNumPorts;
    BOOL            bAllowBaud110;
    BOOL            bAllowBaud300;
    BOOL            bAllowBaud600;
    BOOL            bAllowBaud1200;
    BOOL            bAllowBaud2400;
    BOOL            bAllowBaud4800;
    BOOL            bAllowBaud9600;
    BOOL            bAllowBaud14400;
    BOOL            bAllowBaud19200;
    BOOL            bAllowBaud38400;
    BOOL            bAllowBaud56000;  /* not used at this time */
    BOOL            bAllowBaud57600;  /* not used at this time */
    BOOL            bAllowBaud115200; /* not used at this time */
    BOOL            bAllowBaud128000; /* not used at this time */
    BOOL            bAllowDatabits7;
    BOOL            bAllowDatabits8;
    BOOL            bAllowStopbits1;
    BOOL            bAllowStopbits2;
    BOOL            bAllowParityEven;
    BOOL            bAllowParityOdd;
    BOOL            bAllowParityNone;
    BOOL            bAllowParityMark;
    BOOL            bAllowParitySpace;
    char far           szCustom1BoxLabel;
    char far       *szCustom1Radio1Label;
    char far       *szCustom1Radio2Label;
    struct tagWW_CP_DLG_LABELS FAR
                   *lpRadio2DlgLabels;/* Not used at this time */
    char far       *szCustom2BoxLabel;
    char far       *szCustom2Radio1Label;
    char far       *szCustom2Radio2Label;
    char far       *szCustom3BoxLabel;
    char far       *szCustom3Radio1Label;
    char far       *szCustom3Radio2Label;
    char far       *szCheck1Label;
    char far       *szCheck2Label;
    char far       *szCustomEditLabel;
    UINT            uCustomEditBase;
    UINT            uCustomEditLowLimit;
    UINT            uCustomEditHighLimit;
    FARPROC         lpfnConfigureSave;
    int             iInternalUseOnly;
    char            reserved[16];
} WW_CP_DLG_LABELS, FAR *LPWW_CP_DLG_LABELS;
```

The "*iInternalUseOnly*" and "*reserved*" fields are for Wonderware use only. Before using, always initialize these fields to zero (a memset(...,0,sizeof **WW_CP_DLG_LABELS**) works nicely) to ensure compatibility with future versions of the WWDLG32A.

```
void WINAPI WWFormCpModeString(LPWW_CP_PARAMS, int, char FAR *);
```

Given a port parameter array, and a port number (1-based, i.e., 1 is COM1), return a string suitable for use in a Windows **OpenCom** call (e.g. "COM1:9600,8,N,1").

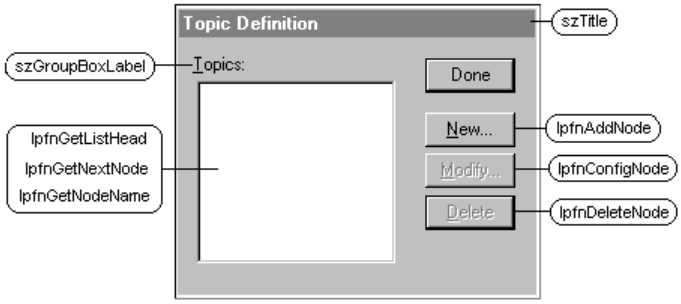
```
void WINAPI WWInitComPortComboBox(HWND, int iNumPorts, int
idControl);
```

Given the control ID of a combo box (typically **CI_TEXT_COMM_PORT** for the standard Communication Port Settings dialog) and the number of ports addressable for the platform (typically 9 for Windows 3.1 and 32 for Windows NT), load the combo box with the names of the ports (e.g. COM1, COM2...COM32).

```
BOOL WINAPI WWConfigureComPort(LPWW_CP_DLG_LABELS);
```

Given the dialog labels structure, display and run the "Communication Port Settings" dialog.

Topic Definition



The **WW_SELECT** data structure and **WWSelect** are used for populating and manipulating a listbox of items. This listbox is typically used for presenting a list of configured topics. Some servers also use this listbox for presenting a list of interface boards for perusal. The dialog box associated with the list contains buttons for adding, modifying, and deleting entries, as well as the standard listbox traversal operations (arrow keys, etc.)

The structure **WW_SELECT** contains the parameters for the **WWSelect** listbox. The fields of **WW_SELECT** are defined as follows:

```
   HWND hwndOwner;
```

This is the window handle of the caller.

```
   char far *szTitle;
```

This is the title string to place in the title bar of the selection dialog ("Topic Definition" in the above dialog.)

```
   char far *szGroupBoxLabel;
```

This is the label to place above the list box ("Topics" in the above dialog.)

```
typedef void far *(CALLBACK   *GETLISTHEADPROC)(void);
   GETLISTHEADPROC lpfnGetListHead;
```

This entry point is called to obtain a pointer to the list head. **WWSelect** does not interpret the pointers representing listbox entries, other than passing them back to the manipulator functions represented here.

```
typedef void far *(CALLBACK   *GETNEXTNODEPROC)(void far *);
   GETNEXTNODEPROC lpfnGetNextNode;
```

Get the successor to a given list node.

```
typedef char far *(CALLBACK *GETNODENAMEPROC)(void far *);
    GETNODENAMEPROC lpfnGetNodeName;
```

Get the name of a given node, (i.e. the Key field) for use in the listbox.

```
typedef void (CALLBACK *ADDNODEPROC)(HWND);
    ADDNODEPROC lpfnAddNode;
```

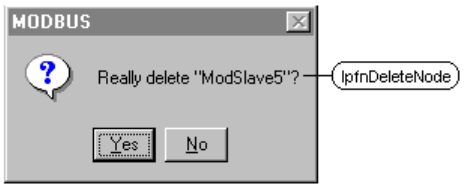
Add a new node to the list. This routine is called in response to the "New..." button, and will most likely need to put up an application specific dialog box for collecting information regarding the new item. Note that there are no list element parameters to the **AddNode** function -- it is up to the application to find the proper position in the list for the new element.

```
   BOOL                    bAddDeleteModifyEnabled;
```

If this value is TRUE, then the Delete and Modify dialog buttons will be enabled. Otherwise, the Delete and Modify dialog buttons will be disabled and the *lpfnConfigNode* and *lpfnDeleteNode* callbacks will never be invoked. Note that the Delete and Modify buttons are also disabled when the listbox is empty.

```
   unsigned char           bDoNotConfirmDeletes;
```

If this value is FALSE, then prior to calling the *lpfnDeleteNode*, the following confirmation dialog will be posted when the Delete button is pressed:



If the value is TRUE, the confirmation dialog will not appear. If the confirmation dialog is enabled, and the user selects the "No" button, the lpfnDeleteNode function will not be called. It is sometimes desirable to perform specialized feasibility checking before presenting the confirmation dialog. In this case, enable the *bDoNotConfirmDeletes* flag, then perform any feasibility checking in the *lpfnDeleteNode* function before presenting your own confirmation dialog. (See the *lpfnDeleteNode* example, below.)

```
typedef void (CALLBACK *CONFIGNODEPROC)(HWND, void far *, BOOL);
   CONFIGNODEPROC lpfnConfigNode;
```

Configure the given list entry. In general, this will put up the same dialog as for the "New..." button, and will initialize the dialog with the values from the given entry. It is reasonable for the user to be allowed to change the name of the entry, thereby creating a new entry. This is equivalent to a "New..." operation.

```
typedef void (CALLBACK *DELETENODEPROC)(HWND, void far *);
          DELETENODEPROC lpfnDeleteNode;
```

Delete the given entry. Called in response to the "Delete" button. It is up to the application to determine if the operation is possible or desirable. For example, the MBPLUS server uses the **WWSelect** listbox for listing SA85 adapter cards. The *lpfnDeleteNode* callback in this case checks to see if the given adapter card is in use by any topics. If so, the user is notified and the card is not deleted.

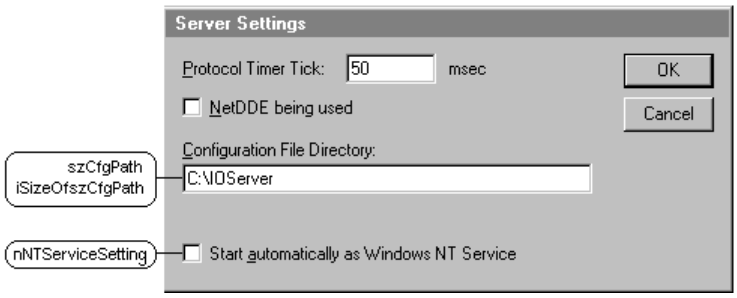
```
   char reserved[15];
```

Expansion space is reserved in the **WW_SELECT** structure. Always initialize this reserved space to 0 to ensure compatibility with future versions of WWDLG32A.

```
BOOL WINAPI WWSelect(LPWW_SELECT);
```

Invoke the **WWSelect**() function with the completed **WW_SELECT** structure to display and process the selection listbox.

Server Settings



The "I/O Server Settings" dialog is one of the simplest dialogs in the WWDLG32A. In response to the "Server Settings..." menu item, the server should initialize a **WW_SERV_PARAMS** structure with the HWND of the main window, a buffer containing the initial configuration path name (a backslash terminated directory name), the size of the buffer, the name of the driver, and a flag indicating whether the server can support the "Server as a service" capability.

The server keeps the configuration path name as a Windows "ProfileString" entry. In the absence of a profile string (e.g. the first time the server is run) the current working directory is provided as the initial value for this string. The values for "Protocol Timer Tick" and "NetDDE being used" and "Start automatically…" are automatically maintained as the "ProfileInt" entries "ProtocolTimer", "ValidDataTimeout". The "ProtocolTimer" value is suitable for use in the call to the Toolkit routine **SysTimerSetupProtTimer**. The "NetDDE being used" flag sets the "ValidDataTimeout" profile entry (an integral value) to a value which is suitable for use in the **ProtGetValidDataTimeout** Toolkit callback. The setting for whether the server is an NT service is stored in the registry in HKEY_LOCAL_MACHINE\SOFTWARE\WONDERWARE\<server registry name>\NTSERVICE and in HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\<server registry name>.

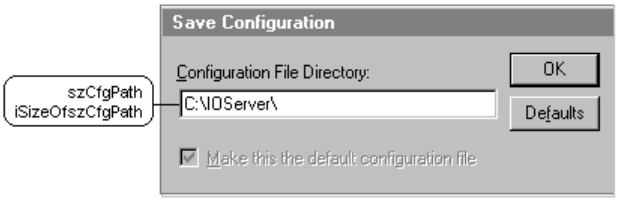
To ensure compatibility with future versions of WWDLG32A, always initialize the reserved area to zero.

```
typedef struct tagWW_SERV_PARAMS {
    HWND        hwndOwner;  /* Identifies Window that owns the
dialog box */
    char far   *szCfgPath;  /* points to a buffer that holds
pathname */
    int         iSizeOfszCfgPath;
    char far   *szDriverName;
    BOOL        bIndefWriteRetrySupported;
    BOOL        bIndefWriteRetry;
    BYTE        bPreventChanges;
    BYTE        bNotService;
    BYTE        bCFGfileUnused;
    BYTE        nNTServiceSetting;
    char far   *szCaption;
    char        reserved[8];
} WW_SERV_PARAMS, FAR *LPWW_SERV_PARAMS;
BOOL WINAPI WWConfigureServer(LPWW_SERV_PARAMS);
```

WWConfigureServer puts up the "Server Settings" dialog box, initialized with the values from the given **LPWW_SERV_PARAMS** structure and the above mentioned WIN.INI profile settings. This function returns TRUE if the OK button was pressed, otherwise it returns FALSE.

The returned value in *nNTServiceSetting* can be examined to determine whether the server has been configured as a Windows NT service, whether that configuration has changed, and whether there was any problem updating the system.

Configuration Files



The "Save Configuration" dialog is a convenience dialog. The server should put up this dialog when a "Save" operation is underway, and the configuration file does not exist in the named directory. This is the last chance for the user to avoid putting a configuration file in the wrong directory.

The **WW_CONFIRM** structure is used to configure the Save Configuration dialog. The *szCfgPath* field should be initialized with the path where the configuration file is to be saved. Changes typed by the user will be placed in this buffer when **WWConfirm** returns. If the "Make this the default configuration file" box is checked, the path in the edit box is written to the "ConfigurationFile" WIN.INI profile string. The "Make this the default configuration file" box will be disabled if the contents of the edit field matches the current default configuration file directory.

The path name is validated before being written to the configuration file. Only valid, existing directory names are accepted.

```
typedef struct tagWW_CONFIRM {
    HWND        hwndOwner;  /* Identifies Window that owns the
dialog box */
    char far   *szCfgPath;  /* points to a buffer that holds
pathname */
    int         iSizeOfszCfgPath;
    char far   *szDriverName;
    char        reserved[16];
} WW_CONFIRM, FAR *LPWW_CONFIRM;

HWND WINAPI WWGetDialogHandle(void);
```

Since **WWConfirm** is generally called from within a dialog procedure, it is not appropriate for the HWND in **WW_CONFIRM** to be the main window. Instead, use *WWGetDialogHandle* to return the window handle for the currently active dialog for use in the *hwndOwner* field.

```
BOOL WINAPI WWConfirm(LPWW_CONFIRM);
```

WWConfirm puts up the "Save Configuration" dialog box.

Convenience Functions

Many Wonderware servers use configuration files with sentinel codes embedded which indicate if more configuration information follows. These functions encapsulate this logic. **UdWriteAnyMore** writes the sentinel indicating if more information will be written to the file. **UdReadAnyMore** reads this sentinel value. Both functions return TRUE unless there was an I/O error, in which case they return FALSE.

```
void WINAPI WWCenterDialog(HWND hDlg);
```

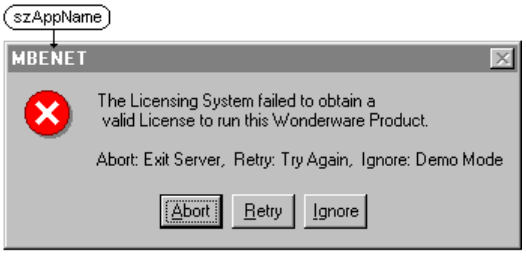
Centers the given dialog within the screen dimensions.

```
void WINAPI WWDisplayErrorReading(LPSTR szAppName, LPSTR
szFileName);
void WINAPI WWDisplayErrorWriting(LPSTR szAppName, LPSTR
szFileName);
void WINAPI WWDisplayErrorCreating(LPSTR szAppName, LPSTR
szFileName);
```

Display a message box containing a message indicating one of three types of file I/O error.

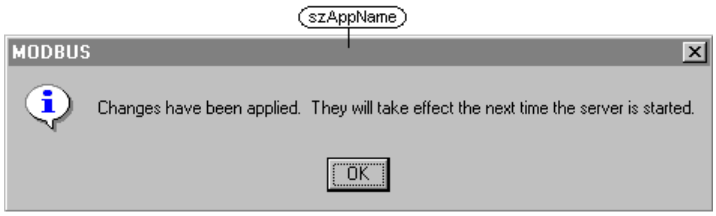
```
void WINAPI WWDisplayOutofMemory(LPSTR szAppName, LPSTR
szObjectName);
```

Display a "Hard System Modal Message Box" indicating an inability to allocate memory for the given object.




```
int WINAPI WWDisplayKeyNotEnab(LPSTR szAppName);
int WINAPI WWDisplayKeyNotInst(LPSTR szAppName);
```

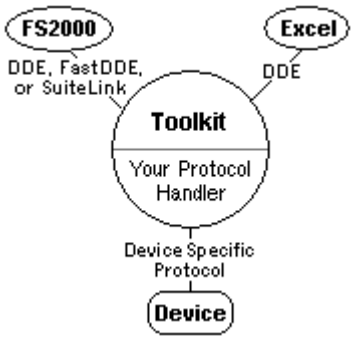
Display a message box indicating problems validating security information with the copy protection device. WWDisplayKeyNotEnab complains that the key does not contain access bits for the current server. **WWDisplayKeyNotInst** complains that the key is not responding.




```
void WINAPI WWDisplayConfigNotAllow(LPSTR szAppName);
```

Display a message indicating that online configuration is not allowed. Servers display this message box if the "Com Port Settings..." or "Topic Definition..." menu items are selected when any topics are open.

CHAPTER 14



Adding the Toolkit to an Existing Windows Application

The Toolkit can be used to add DDE (Dynamic Data Exchange) or SuiteLink capability to an existing Windows application. In order to accomplish this, there are several things that need to be done:

- Must link with the Toolkit library, it is named TOOLKIT7.LIB.
- Must call **UdInit**() early in your application to initialize the Toolkit.
- Must call **UdTerminate**() at application shutdown time to close the Toolkit.
- Application must not register classes containing the application name returned by **ProtGetDriverName**() or the string "WddeWnd".
- Application must define external data *hWndParen*t and *hInst.* These variables are initialized by **UdInit().**

  extern HWND *hWndParent*
  extern HANDLE *hInst*

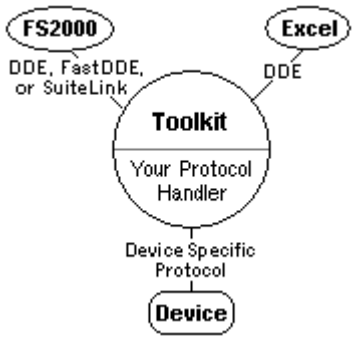
- Application must not use global variable *hWndParent.*
- File UDMAIN.C or file containing equivalent functionality must be added to the application and linked in.
- Unknown messages must be passed to **ProtDefWindowProc**(), not **DefWindowProc** ().
- User must define the boolean variable *bLinkedToExistingEXE* and initialize it to TRUE before calling **UdInit**().

  extern BOOL bLinkedToExistingEXE = TRUE;

- Application's resource (.RC) file must contain STRINGTABLE for protlib.str.
- Application's resource (.RC) file must include tkitstrt.rci.

To add HELP to your application you must do your own calls to WINHELP. The Toolkit cannot do this for you.

CHAPTER 15



Running as an NT Service

This section describes how to set up an I/O Server to run as a Windows NT service.

Contents

Overview of Services

On Windows 98, an I/O Server runs as an application. Typically, this is also how a server runs on a Windows NT/2000 platform. However, it is also possible to configure a server to run on Windows NT/2000 as a service.

From a functional standpoint, the main difference between a service and an application is this: on Windows NT/2000, someone has to be logged on to the computer for an application to run. That is, the computer boots up, Windows is started up, and any applications can then be started. Applications can even be started automatically, e.g. by placing icons for them in the STARTUP folder. This can be set up to happen completely automatically from a cold start on Windows 98. However, on Windows NT/ 2000, the programs in the STARTUP folder are not activated until a user logs on. This can be a problem if someone is trying to set up a completely automatic start-up, as might be desirable for handling recovery from a power-loss.

Windows NT/ 2000 services run under a "system account" that starts up during the boot-up process for Windows NT/ 2000. Many functions are provided by programs running as services "in the background," such as TCP/IP. Also, since some services are dependent upon having other services up and running before they can start, Windows NT/2000 provides several ways of controlling the order in which services are started up.

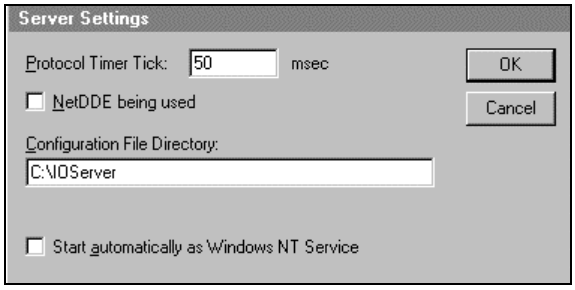
Once services are in place and running, a computer running Windows NT/2000 can continue running these services, even if no user ever logs onto the machine. Also, if someone does log on, performs some operations, and logs off, programs running as services continue to function after the user logs off from the system.

Before a service can be started, it must be ***installed*** via the Service Control Manager. Once it is installed, the start-up for the service can be configured in one of three ways: ***automatic***, ***manual***, or ***disabled***. An I/O Server should be configured to start automatically, since the intention of making an I/O Server run as a service is to have it start as part of the boot-up process for Windows NT/ 2000.

More details about Windows NT/2000 and services can be found in Microsoft's manuals and help files.

Configuration Dialog

The simplest way to set up an I/O Server as a Windows NT service is to use the Wonderware Common Dialog for Server Configuration. This dialog contains a checkbox that enables or disables running the server as an NT service:



When the user clicks this checkbox and then clicks OK, the Common Dialog DLL does two things:

1. It attempts to set a flag in the Registry indicating to the server that it should run as an NT service. This flag is in the key HKEY_LOCAL_MACHINE\SOFTWARE\Wonderware\<server extended name> and is named NTService: REG_DWORD. For example, for the server TESTPROT, the entry would be

   HKEY_LOCAL_MACHINE
   SOFTWARE
   Wonderware
   TESTPROT_IOServer

   With value NTService: REG_DWORD: 1.

2. It attempts to get the Service Control Manager (SCM) to install the server as a service, mark it for automatic start-up, and to set up a standard set of dependencies on other services. These settings can be found in HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\<server extended name>. For example, for the server TESTPROT, the entry would be

   HKEY_LOCAL_MACHINE
   SYSTEM
   CurrentControlSet
   Services
   TESTPROT_IOServer

   Various Registry values get set, including ImagePath (the path to the executable), Start (set to 0x2, i.e. automatic), and DependOnService (set to a list of services upon which the server depends – see below).

If the attempt to establish these settings is successful, the server is ready to run as a service. To ensure proper operation, you should shut the server down and reboot the computer.

Typically, a server calls the Server Configuration dialog in a very generic way:

```
/*********************************************************/
/* set general I/O server settings */

VOID
WINAPI
ServerSettings (HWND hWnd)
{
    WW_SERV_PARAMS  ServParams;
    char driverName [20];
    char temp_szCfgPath [PATH_STRING_SIZE];

    /* get name of driver application */
    ProtGetDriverName (driverName, sizeof (driverName));

    /* try reading the configuration path from WIN.INI */
    temp_szCfgPath [0] = 0;
    GetProfileString(driverName,
                     NAME_PATH,
                     DEFAULT_PATH,
                     temp_szCfgPath,
                     PATH_STRING_SIZE);
    if (strlen (temp_szCfgPath) == 0) {
        /* no config path defined, use default path to EXE */
        WWGetExeFilePath(temp_szCfgPath, PATH_STRING_SIZE);
    }

    /* ensure pathname ends with a backslash */
    if (temp_szCfgPath[strlen (temp_szCfgPath) - 1] != '\\') {
        strcat (temp_szCfgPath, "\\");
    }

    /* set up I/O server settings via common dialog */
    memset (&ServParams, 0, sizeof ServParams);
    ServParams.hwndOwner = hWnd;
    ServParams.szCfgPath = (LPSTR) temp_szCfgPath;
    ServParams.iSizeOfszCfgPath = PATH_STRING_SIZE;
    ServParams.szDriverName = GetAppName();
    ServParams.bIndefWriteRetrySupported = FALSE;
    if (iAllocatedDevices) {
        /* protocol running, don't reconfigure now */
        WWDisplayConfigNotAllow (GetAppName ());
        ServParams.bPreventChanges = 1;
    } else {
        /* nothing allocated, go ahead and configure */
        ServParams.bPreventChanges = 0;
    }
    /* configure, or allow user to look at configuration */
    WWConfigureServer((LPWW_SERV_PARAMS) &ServParams);
} /* ServerSettings */
```

To disallow the user from checking the “run as a service” checkbox, before you call WWConfigureServer(), insert the following statements:

```
/* disallow setting as an NTservice */
ServParams.bNotService = TRUE;
```

This does not disable the checkbox; but the server will display an error message if the user attempts to check it and clicks OK.

After WWConfigureServer() returns, you can check whether the server was configured as an NTservice, whether that setting was changed, and whether there was any problem updating the Registry:

```
if ((ServParams.nNTServiceSetting & WW_NTSERVICE_IS_SERVICE)
    != 0) {
    /* server is configured as an NT service */
}
if ((ServParams.nNTServiceSetting & WW_NTSERVICE_CHANGED) != 0) {
    /* setting was changed by the user */
}
if ((ServParams.nNTServiceSetting & WW_NTSERVICE_ERROR) != 0) {
    /* something went wrong updating the Registry */
}
```

You can use these in various combinations to determine whether to change Registry entries for additional drivers, etc. according to whether the server is being set up as a service or as an application. See the section below on Dependencies.

If the user un-checks the checkbox, the Common Dialog DLL does not uninstall the server. Instead, it tells the Service Control Manager to disable start-up of the service. This is necessary because the I/O Server may be running as a service when the Server Configuration Dialog is accessed, and additional activities may be necessary before shutting down the server. To ensure proper operation, after reconfiguration you should shut down the system and reboot the computer.

Driver Name

It is important to note that the Wonderware Common Dialogs use the extended server name when accessing the Registry and the Service Control Manager. That is, the extension "_IOServer" is appended to the "short" server name to produce the extended server name. For example, TESTPROT becomes TESTPROT_IOServer.

The function GetServerNameExtension( ) sets up the extended server name so the server-specific code can access it. To ensure that the name of your server is handled consistently, you should be sure to insert a call to this function inside the routine ProtGetDriverName( ), which is the routine used by the I/O Server Toolkit to obtain the "short" name of your server:

```
/*********************************************************/
/** Copy name of driver to string at indicated location **/

BOOL
WINAPI
ProtGetDriverName(LPSTR lpszName,
                  int   nMaxLength)
{
    /** WARNING: No calls to debug()...
        debug() calls ProtGetDriverName(),
        therefore ProtGetDriverName() cannot call debug(). **/
    lstrcpy(lpszName, GetString(STRUSER + 76) /* "UDSAMPLE" */ );
#ifdef WIN32
    GetServerNameExtension();
#endif
    return (TRUE);
} /* ProtGetDriverName */
```

This extension was found to be necessary where some I/O Servers had the same name as third-party drivers (such as MODBUS.exe vs. MODBUS.sys), and attempting to configure both as NT services caused name clashes in the Registry and Service Control Manager.

Actually, GetServerNameExtension( ) is a function in the Toolkit that obtains the extension string from WWDLG32A.DLL (the Wonderware Common Dialog DLL). Not all products that use the I/O Server Toolkit need an extended name, which is why this call is performed explicitly. If GetServerNameExtension( ) is never called, the application name and Registry key would merely be the short name (e.g. "TESTPROT"). This is fine for most applications; but it is important to remember that the "Run as NT Service" checkbox in WWDLG32A.DLL assumes the name with the extension.If you don't call the function GetServerNameExtension( ), you must then provide your own set-up dialog and routines for making the application an NTservice.

Service Dependencies

Some programs depend upon the presence of other programs to run properly. For example, if you are using a word processor and want to print a document, you must have a printer driver loaded. It may or may not be possible to start the driver after the program is already running, depending on the nature of the dependency.

If an I/O Server is to run as a Windows NTservice and it depends upon other services to run, it may be necessary to start the other services first before the server will function properly. When the "Run as a service" checkbox is checked in the Server Settings dialog, WWDLG32A.DLL sets up the following default set of dependencies, in this order:

DependOnService: REG_MULTI_SZ: WWLOGSVC WWNETDDE SLSSVC

These services are as follows:

WWLOGSVC
: The Wonderware Logger Service. This passes messages from debug( ) calls to the **Wonderware Logger**.

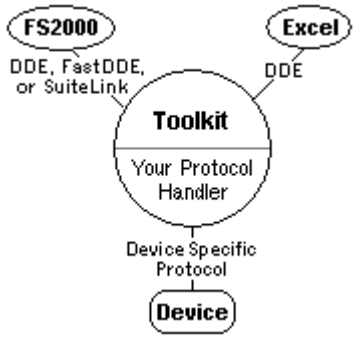
WWNETDDE
: The Wonderware NetDDE Helper Service. This service is necessary for DDE conversations to continue even with no one logged on a the computer.

SLSSVC
: The Wonderware SuiteLink Service. This service passes messages between applications using the SuiteLink protocol.

When the computer is booted up, Windows NT/2000 inspects the Registry for information regarding the order in which various drivers and services are to be started up. Services and drivers can actually be put into specific groups to be loaded in specific orders. Services with no specific group are loaded last. Within a group, if a service depends upon other services, NT/2000 checks first to see whether other services are running and, if not, attempts to start them. Any problems are identified by messages in the Windows NT/2000 Event Log, which can be accessed via the Windows NT/2000 EventViewer.

Your server may have additional services upon which it depends, particularly if it requires a device-specific driver or service to run properly. In this case, you should make your own calls to access the Registry or the Service Control Manager to establish these additional dependencies. If you use the Wonderware Common Server Configuration Dialog, you can check the returned information in **ServParams.nNTServiceSetting** to check whether operation as a service has successfully been changed, enabled, or disabled and determine whether to make your additional Registry changes accordingly.

CHAPTER 16



Porting to Windows NT

This chapter will be useful for two different audiences. The first and primary audience is the developer who wants to port an existing 16-bit Windows server to the Windows NT environment. If you fall in this category, you will find that this chapter provides a useful recipe for accomplishing this port. The goal is to make this port as straightforward as possible by giving you a concise set of step-by-step instructions.

The secondary audience for this chapter is the developer who wants to develop a new server that will operate in both 16-bit Windows and 32-bit Windows. If you fall in this category, the step-by-step instructions for porting an existing server to Windows NT will provide you with some useful ideas for accomplishing this task.

Contents

Primary Goals

Keep the following goals in mind when porting a server from 16-bit Windows to the 32-bit Windows NT environment:

1. The source code for the server should maintain as much common code as possible between the two operating systems. In cases where common code is not possible, condition compiles using "#ifdef WIN32" macros will be used.
2. If there is a server configuration file which is used to store topic, device and communications configurations, it should have the same file structure on both platforms. This allows the same configuration file to be used seamlessly regardless of platform.
3. The server should maintain a common graphical user interface between the two platforms. A user should not have to learn how to use your server again upon moving to a new operating system. A side benefit of this goal is that any user-level documentation you provide for your server will not have to be customized for Windows NT.

Server Porting Instructions

The server porting process is sequential and categorized into several phases.Generally, the earlier phases are more mechanical and simple in nature.As you progress through the phases, you will find they require more time and thought.If you have any questions about the meaning of any of these phases, please refer to the UDSERIAL or UDBOARD sample servers included with this Toolkit. These servers provide useful illustrations of the instructions described below.

Warning Backup your source code to another location prior to beginning this work. Keep in mind the following abbreviations: Win16 refers to 16-bit Windows; Win32 refers to 32-bit Windows.

Phase I - General, Quick Edits of "C", "H" Files

Goal: Do simple, mass edits typical of all servers. Use the sample server as a reference guide.

1. For each C file, move "#include <windows.h>" to the top of the list of includes. Insure that the windows.h is enclosed by the brackets and not double quotes.
2. Replace each "FAR PASCAL" in function definitions and prototypes with the portable "WINAPI".
3. For each C file, delete the large collection of "#define NOxxxxx" macros at or near the top if they exist. They are no longer needed for versions of the Windows SDK greater than 3.0.
4. For each C file, add "#include "ntconv.h"" to the list of includes. This header file contains porting macros which simplify the porting process. Refer to the Windows NT Porting Functions and Macros for a list of the porting macros.
5. Replace the obsolete "assert.h" include with "wwassert.h".
6. Delete the obsolete "tools.h" include, which is no longer needed.
7. Check if you need "lmem.h" and "listrcmp.h"; They are OK but very old. listrcmp is simply lstrcmpi.
8. Replace the obsolete "lmemclr.h" include with "lmem.h" if you are using the "lmemclear()" function call.

Phase II - More Edits

Goal: Do edits typical of all servers which require more time and thought. Use the sample server as a reference guide. Refer to the "I/O Server Toolkit Functions" section for details on Windows NT Porting Functions and Macros.

1. Change all definitions and prototypes for Windows procedures, dialog procedures, and callbacks to use the new parameter declarations.

   From: HWND, unsigned, WORD, LONG
   To: HWND, UINT, WPARAM, LPARAM

2. Search for all occurrences of WM_COMMAND. For each occurrence, make sure that "LOWORD(wParam)" is used to get the control identifier rather than just "wParam".

3. Search for all occurrences of **EM_SETSEL**. The message packing for this function varies between Windows and Windows NT. Use the new portable functions listed below instead of sending an **EM_SETSEL** message directly.

```
#include "portfncs.h"
PfnSendEmSelectAll( HWND hDlg, int idControl, BOOL bScrollCaret
);
PfnSendEmSelectRange( HWND hDlg, int idControl, int nStart, int
nEnd, BOOL bScrollCaret );
```

For more information on these functions, see the "API Function Reference" chapter.

4. If you use the Windows **ChangeMenu**() function, with the 4th parameter set to "-1", change it to "0xffff". If you have time, consider converting to the newer menu functions **AppendMenu**(), **DeleteMenu**(), **InsertMenu**(), **ModifyMenu**(), and **RemoveMenu**().

5. Search for all **SendMessage** function calls. If the first parameter is 0xffff, this is not portable, since it would need to be 0xffffffff on Windows NT. Replace the 0xffff with **SM_MINUS_ONE** (macro in NTCONV.H).

6. Search for references to **ConfigurePort**(). This function name conflicts with a native Windows NT function. Change the function name to **ConfigPort**() or something similar.

7. In all ".C", ".RC", and ".H" files, search for "BAUD_" references and change to "WW_BAUD_" references. This will prevent conflicts with native Windows NT definitions in WINBASE.H.

8. In all ".C", ".RC", and ".H" files, search for "PARITY_" references and change to "WW_PARITY_" references. This will prevent conflicts with native Windows NT definitions in WINBASE.H.

9. If you have not already, convert to the new version of the Wonderware Heap API as described in the "Getting Started with the I/O Server Toolkit" section. Briefly, you will need to change all Wonderware **HeapXxXxxx** style function calls to **wwHeap_XxXxxx**. Add the HHEAP (heap handle) as the new first parameter to the **wwHeap_ReAlloc** and **wwHeap_FreePtr** function calls.

10. Change each "Assert(FALSE);" to "ASSERT_ERROR;".

11. If the Windows function **GetTextExtent**() is used, change it to the portable **GetTextExtentPoint**() function.

12. If the Windows function MoveTo() is used, change it to the portable MoveToEx().

Phase III - Porting Tool (optional step)

Goal: Locate any pending porting problems at this time.

At this time, you are well on the way to taking care of most of the basic porting issues associated with moving a server to Windows NT. Some more difficult areas related to file I/O and communications remain.

The Microsoft Win32 SDK includes a utility called "PortTool" which you can use to locate any potential porting problems which still remain. This is a good time to use this tool on each of your ".C" files. For the most part, it will point out potential difficulties which you have already solved. However, it also may catch some items which would otherwise slip through the cracks and cause more mysterious problems later.

Phase IV - Compiling

Goal: Compile each file in the server on Win32 with no warnings/errors.

You are not yet finished making modifications, but this is a good time to compile your source code and let the compiler catch any obvious problems which exist. Any problems related to communications functions and file I/O can be ignored at this time.

1. Compile under Windows NT. You should expect some warnings and errors at this time.

Some tips on resolving compiler problems:

NULLs need to be cast to the appropriate type.

There may be errors with communications items such as fRtsDisable, fRtsflow, fDtrDisable, fDtrflow. For now, simply comment out the code. It will be fixed in the next phase.

There may be errors with communications items such as **CE_CTSTO**, **CE_DSRTO**, **CE_RLSDTO**. For now, simply put the code under a "#ifndef WIN32" conditional.

Numerical constants such as -1,-2, -3, etc. when used in logical expressions can have problems. Make sure you cast them appropriately.

Phase V - Communications

(Windows COMM Driver Serial Servers Only)

Goal: Add common source communications function calls. Use the sample server source for UDSERIAL as a reference guide. Refer to the "I/O Server Toolkit Functions" section for details on Windows NT Porting Functions and Macros.

Note If you are developing a board-based server on Windows NT, a kernel device driver must also be developed or purchased which interfaces directly with the board of interest. Your server then must be modified to communicate with the device driver in order to send and receive messages. It can no longer access the board directly as was possible under Windows. Documentation of device drivers and device driver interfacing is beyond the scope of these instructions. Please refer to the Microsoft Device Drivers Kit (DDK) for instructions and information.

1. To all ".C" files which call communications functions, add include of "NTSRVR.H". This file provides prototypes for functions which emulate the communications functions found in Windows. Typical files which contain communications functions will be xxLDCFG.C, xxFREE.C, and xxPROTCL.C.
2. Change **BuildCommDCB**() to the compatible function **NTSrvr_BuildCommDCB**().
3. Change **SetCommState**() to **NTSrvr_SetCommState**() and add the new first **parameter.**
4. Change **GetCommState**() to **NTSrvr_GetCommState**(). The **NTSrvr_GetCommState**() call should be located after the **OpenComm**() and before the **NTSrvr_buildCommDCB**(). The DCB needs to be initialized, and the Windows NT version does not initialize all members of the DCB structure. This is the easiest way to accomplish this initialization and it works on both platforms.
5. Group the setting of the DCB's *fRtsflow* and *fRtsDisable* members together and replace with a call to function **NTSrvr_SetDCB_Rts** with the parameter of:

```
NTSrvr_RTS_DISABLE          if the "fRtsDisable" was set.
NTSrvr_RTS_ENABLE           if the "fRtsDisable" was NOT set
                            and "fRtsflow was NOT set.
NTSrvr_RTS_HANDSHAKE        if the "fRtsflow" was set.
```

6. Group the setting of the DCB's *fDtrflow* and *fDtrDisable* together and replace with a call to function **NTSrvr_SetDCB_Dtr**() with the parameter of:

```
NTSrvr_DTR_DISABLE          if the "fDtrDisable" was set.
NTSrvr_DTR_ENABLE           if the "fDtrDisable" was NOT set
                            and "fDtrflow" was NOT set.
NTSrvr_DTR_HANDSHAKE        if the "fDtrflow" was set.
```

Phase VI - Compile Again

Goal: Compile each file in the server on Win32.

At this time, all your communications functions should be in place and ready for compilation. Compile again on both platforms and resolve any warnings or errors.

Phase VII - File I/O

Goal: Replace Win16 file I/O with common source file I/O macros. Use the sample server as a reference guide. Refer to the "I/O Server Toolkit Functions" section for details on Windows NT Porting Functions and Macros.

Your server probably has some file I/O calls such as open(), close(), _lread(), _lwrite(), etc. These function calls should be modified to use the portable macros provided in NTCONV.H and listed in the "I/O Server Toolkit Functions" section. Follow these instructions:

1. Change each read-only open() to "OPENRead(filename)".
2. Change each file-create open() to "OPENCreate(filename)".
3. Change each "read()", "_read()", or "_lread()" to simply "LREAD()".
4. Change the variations of "write" to "LWRITE".
5. Change the variations of "lseek" to "LSEEK".
6. Change the variations of "close" to "LCLOSE".

Phase VIII - Compile Again (see above)

Goal: Compile on both platforms with no warnings or errors.

At this time, all your file I/O functions should be in place and ready for compilation. Compile again on both platforms and resolve any warnings or errors.

Phase IX - Symbol Table Entry Size

In your xxMAIN.C file, there is a check of the symbol table entry size. This check is no longer required under Windows NT, so conditionalize this check of "sizeof(SYMENT)" under a "#ifndef WIN32". See the UDMAIN.C in the UDSERIAL sample server for an example.

Phase X - Compile Again (see above)

Phase XI - Config File Compatibility

Goal: Server configuration file (xxxxxx.CFG) is identical on both platforms. This allows users to move it between platforms with no compatibility issues. Use the sample server as a reference guide.

A major goal is that the configuration (.CFG) file created and used by the server can be moved between the Windows NT and Win16 platforms seamlessly. This requires that extra work be done in the server to retain this compatibility. This is because the structures (specifically the "topic" structure variant) that are used to directly read and write the configuration file may be packed differently between Win16 and Windows NT. This is because Windows NT forces structure members to be aligned on their natural boundaries (e.g. WORD on word boundary, DWORD on double word boundary, etc.). The approach that should be used is to define an extra intermediate (or scratch) structure for each (port and topic) which is used exclusively for the read/write operations. All members of these structures are arrays of type "char" to force byte alignment as on Win16. For example, in the UDSERIAL sample server's UDCONFIG.C you will find the following structure definition:

```
/*
* This structure is the one used to read in the config file into
* a structure that has members whose alignment/packing will be
* identical to the file's.  (This is an NT issue). This means
* words are on even boundaries, and longwords are on longword
* (multiple of 4) boundaries.  To make this easiest and most
* straightforward, each structure member is of type char, and
* will be an array if necessary to match the actual member size
* in number of bytes.  Then, memcpy's will be done out of this
* structure and into the real one.
*/
```

typedef struct tagTOPIC_file {
    char *tp_next[4]*;
    char *tp_comPort[2]*;
    char *tp_name[33]*;
    char *tp_topicAddress*;
    char *tp_coilReadSize[2]*;
    char *tp_regReadSize[2]*;
    char *tp_updateInterval[4]*;
} TOPIC_CFG_FILE;

Then, immediately after the read or immediately before the write, the data is moved from or to this temporary structure into the actual usable structure already in place in the code.

Note You'll need to do this in xxCONFIG.C and xxCONVERT.C (if any). The xxCONVERT.C will only exist if the configuration file format has changed during the release lifetime of the server.

Note The maximum number of serial ports allowed on Windows 16 was previously 4. You have the option of increasing this number to 9 on Windows and to 32 on Windows NT. This will require a change to the configuration file also. See the sample servers for an example of how to add the additional communications ports to the configuration file.

Note The WW_CP_PARAMS communications port structure can be read and written directly to the configuration file. It is properly aligned for all platforms.

Phase XII - Compile Again (see above)

Phase XIII - Common Dialogs (optional)

Goal: Give server standard look and feel. Use sample server for examples.

You may choose to use the new Common Dialog API. This will require substantial changes. Refer to the "Common Dialogs" section. You may want to postpone this phase until you are done testing below.

Phase XIV - Final Compile and Link

Goal: Compile and Link server on Windows NT / 2000 or Windows 98 Second Edition.

At this time, all source code modifications which are necessary to build and execute your server on Windows NT / 2000 should be complete. You should now compile and link your server and prepare for testing and debugging.

Phase XV - Test the Server

Goal: No bugs.

Test on Intel processor first on Windows NT if possible, then Windows 98 Second Edition.

Note that on Windows 98 Second Edition , the SuiteLink protocol may not be available and the server cannot run as a Windows NT service.

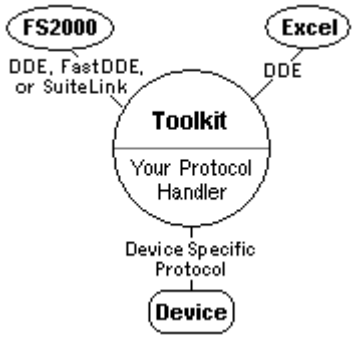
Miscellaneous Debugging Hints

Check "union" type structures for any "int" types. Make sure that you did not want a "short". "short" and "int" are the same size (2 bytes) on Windows 16, but not Windows NT. If a WORD and an "int" are overlaid, you may need to initialize the "int"; you also may need to sign extend, etc. If you have an array overlaid with another array, be careful.

When building messages or extracting data from messages, be careful with data pointers. If you use a BYTE pointer to process the data, all is OK! But if you use WORD, DWORD, or structure pointers to process the data, it MUST be aligned. To get it to be so, you'll have to use memcpys to workaround the problems. Test thoroughly! (i.e. all message types).

When converting data, casting a WORD to an "int" will work on Win16; but on Win32 you need to be careful. If you have 2 bytes of data from the device and the data has been saved in a WORD variable, casting it to an "int" will NOT sign extend (first cast as a "short"). Test thoroughly!

CHAPTER 17



Porting an Existing Server to FS2000

This section describes the software changes necessary to upgrade an existing I/O Server, developed with a previous Wonderware Toolkit, to FS2000.

Contents

Overview

If you have an existing I/O Server written with a previous version of the Wonderware DDE Server Toolkit, porting your code to Factory Suite 2000 should be fairly easy – particularly if your server has been written as a Win32 program for Windows NT. Most of the changes in the Toolkit involve enhancements that are transparent to the programmer. However, you will want to make changes to four basic areas of your code:

- handling the driver name
- handling the start-up Splash Screen and Start-up Message
- handling the About Box
- handling updates to Value/Time/Quality (VTQ)

Driver Name

The routine ProtGetDriverName( ) is the routine used by the I/O Server Toolkit to obtain the “short” name of your program. You should insert a call to the function GetServerNameExtension( ) inside this function:

```
/*********************************************************/
/** Copy name of driver to string at indicated location **/

BOOL
WINAPI
ProtGetDriverName
(LPSTR lpszName,
                  int   nMaxLength)
{
    /** WARNING: No calls to debug()...
        debug() calls ProtGetDriverName(),
        therefore ProtGetDriverName() cannot call debug(). **/
    lstrcpy(lpszName, GetString(STRUSER + 76) /* "UDSAMPLE" */ );
#ifdef WIN32
    GetServerNameExtension();
#endif
    return (TRUE);
} /* ProtGetDriverName */
```

For I/O Servers, the extension is “_IOServer” and this string gets internally stored in the Toolkit. The extension gets appended to the basic name of the server to produce the full name used in Registry keys, the NT service name (if the server is configured as a service), etc. For example, the TESTPROT server uses the full name “TESTPROT_IOServer”. This extension was found to be necessary where some I/O Servers had the same name as third-party drivers (such as Modbus.exe vs. Modbus.sys), and attempting to configure both as NT services caused name clashes in the Registry and Service Control Manager.

Actually, GetServerNameExtension( ) is a function in the Toolkit that obtains a string from WWDLG32A.DLL (the Wonderware Common Dialogs).Not all products that use the I/O Server Toolkit need an extended name, which is why this call is performed explicitly. If GetServerNameExtension( ) is never called, the application name and Registry key would merely be the short name (e.g. “TESTPROT”). This is fine for most applications; but it is important to remember that the “Run as NT Service” checkbox in WWDLG32A.DLL assumes the name WITH the extension. If you don't call the function GetServerNameExtension( ), you must then provide your own set-up dialog and routines for making the application an NT service.

CommonUI Splash Screen and Start-up Message

When an I/O Server starts up, it displays a splash screen – a small window that identifies the program that is starting up. Mostly, the purpose of the splash screen for any program is to give the user something to look at while the program starts up – which might take several seconds.

Previous versions of the Wonderware I/O Server Toolkit displayed a dialog with the resource name WWStartup, which could be generic or which could be tailored by the programmer as needed.This option is still available, but with the advent of FactorySuite 2000, the default behavior is to display the Common User Interface splash screen. The CommonUI routines use one of a set of Wonderware bitmaps for the corresponding product type, examine a particular set of resources for the product name and version number, and interrogate the Wonderware License Manager for license features and expiration dates. The information so obtained is then displayed in the splash screen and in a start-up message. The problem is, this is very Wonderware-centric. If you're developing a Wonderware product, this start-up display has been made quite easy, using a call to WWAnnounceStartup( ). Otherwise, you might want to display your own splash screen, or no splash screen at all.

To tailor the handling of the start-up splash screen, in ProtInit( ) call

```
void WINAPI SetSplashScreenParams (BOOL bSuppressSplashScreen,
    int nSplashSelect, UINT iProductID,
        LPSTR lpszPrivateString);
```

with the appropriate parameters:

bSuppressSplashScreen

If TRUE, suppresses the Splash Screen entirely. This may be appropriate if your program displays a splash screen itself elsewhere.

If FALSE, the selected splash screen will be displayed for a few seconds and will then be erased.

nSplashSelect

If 0, displays the original Wonderware splash screen using the dialog resource WWStartup. This is provided for backward compatibility.

If non-zero, displays the Common User Interface splash screen.

iProductID and lpszPrivateString

These are used as a product selection and special string for the Common User Interface splash screen.

The routine SetSplashScreenParams( ) can be used to select between the CommonUI splash screen, the “old-style” splash screen, or suppressing the splash screen. The following code, taken from UDMAIN.C of the sample server, should appear near the start of the routine ProtInit( ):

[Note: szServerIDstring is defined as a global variable as follows:

```
#ifdef WIN32
 char szServerIDstring[100];
#endif
```

]

```
#ifdef WIN32
    /* set up string with server name and version number */
    strcpy (szServerIDstring,
            GetString(STRUSER+142));  /* "Sample I/O Server" */
    L = strlen (szServerIDstring);
    sprintf (&szServerIDstring[L],
             GetString(STRUSER+145),  /* "- Version %s" */
             SERVER_VERSION);
#endif

#if (defined(USING_WW_COMMONUI) && defined(WIN32))
    /* set up to display common start-up message
       and splash screen */
    WWAnnounceStartup(COMMON_IOSERVER32ID,
                      (LPSTR) szServerIDstring);
#else
    /* Log our own startup message and version number */
    strcpy (tmpBuf, GetString(STRUSER+144)); /* "Startup" */
    L = strlen (tmpBuf);
    sprintf (&tmpBuf[L],
             GetString(STRUSER+145), /* "- Version %s" */
             SERVER_VERSION);
    debug (tmpBuf);
    /* set up to display start-up splash screen
       with resource name WWStartup */
 #ifdef WIN32
    /* select WWStartup splash screen
       instead of Common UI splash screen */
    SetSplashScreenParams (FALSE, 0, COMMON_IOSERVER32ID,
                           (LPSTR) szServerIDstring);
    /* ensure common user interface
       has instance handle for application */
    Common_SetAppInstance (hInst);
 #endif
#endif
```

CommonUI About Box

As with the splash screen, the CommonUI routines support the display of a standard Wonderware FS2000 About Box, which includes information from a standard set of resources and from the Wonderware License Manager. Again, this is very Wonderware-centric. You have a choice of several ways you can display an About Box. All three of these are illustrated by code in the sample server UDSAMPLE.

Display the CommonUI About Box
You will have to provide resource version information and a resource icon named ABOUTICON. Normally, this would be the same as the main icon for your I/O Server.

```
Common_SetAppInstance(hInst);
Common_ShowAboutBox (COMMON_IOSERVER32ID, szServerIDstring);
```

Display the Wonderware Common Dialog About Box
This can be somewhat tailored by putting information into the comment string.

```
WW_AB_INFO    AboutBoxInfo;
HICON         hIcon;

memset (&AboutBoxInfo, 0, sizeof(AboutBoxInfo));
hIcon = LoadIcon(hInst, "Udprot");

AboutBoxInfo.hwndOwner    = hWnd;
AboutBoxInfo.szDriverName = GetAppName();
AboutBoxInfo.szId         = GetString(STRUSER + 142);
                            /* "Sample I/O Server" */
AboutBoxInfo.szVersion    = GetString(STRUSER + 0);
AboutBoxInfo.szCopyright  = GetString(STRUSER + 143);
                            /* "1997" */
AboutBoxInfo.hIcon        = hIcon;
AboutBoxInfo.szComment    = GetString(STRUSER + 149);
                            /* "This is a sample server" */

WWDisplayAboutBox((LPWW_AB_INFO) &AboutBoxInfo);
DestroyIcon (hIcon);
```

Display an About Box of your own design.
You will have to provide a routine UdPrivateAbout( ) to handle the dialog.

```
FARPROC       lpprocAbout;

lpprocAbout = MakeProcInstance((FARPROC) UdPrivateAbout, hInst);
DialogBox(hInst, GetString(STRUSER + 75) /* "ABOUT_BOX" */ ,
          hWnd, lpprocAbout);
#ifndef WIN32
    FreeProcInstance(lpprocAbout);
#endif
```

Value, Time, Quality

One of the main functional differences between the FactorySuite 2000 I/O Server Toolkit and previous versions of the Toolkit is the addition of Time and Quality stamps to the data. The I/O Server can now pass to the client information about when a point was last updated and whether the data is completely trustworthy or has some problems associated with it.

To support the transfer of Value/Time/Quality (VTQ) information, the call to DbNewValueFromDevice( ) has been replaced with several other calls:

DbNewVTQFromDevice( ) updates value, time, and quality
DbNewTQFromDevice( ) updates time and quality only (value unchanged)
DbNewQFromDevice( ) updates quality only (a current default time mark is used)

The function DbNewValueFromDevice( ) is still available, but what happens internally is that it makes a call to DbNewVTQFromDevice( ) using default values for time and quality. Quality is assumed good (WWQ_GOOD). The Toolkit provides a default time mark only if one is needed. You can either choose to leave intact all your calls to DbNewValueFromDevice( ), or you can add code to explicitly get a time mark and pass it to the Toolkit via one of the other functions.

Most PLCs do not have a date/time clock that is accessible to the I/O Server. For those which do, you have the option of using that date/time clock as your time mark. In this case, you should convert the time from the PLC's format to a FILETIME value (see the chapter on Time Marks). Otherwise, you should use the Toolkit API function DbGetGMTasFiletime( ) to read the computer's time.

If you are explicitly obtaining the time via a call to DbGetGMTasFiletime( ), you should take into account what these calls will do to server performance. If you're updating 10,000 points per second, their time marks aren't going to be very different from one another; so you probably don't need 10,000 calls to DbGetGMTasFiletime( ). You should limit calls to this function in a way that maximizes performance while maintaining a reasonable update rate for the time mark. Reasonable places to get the time mark include the following:

- Once per entry to ProtTimerEvent( ), and then only if you need it
- Once per response message (which would normally encode data for many points)

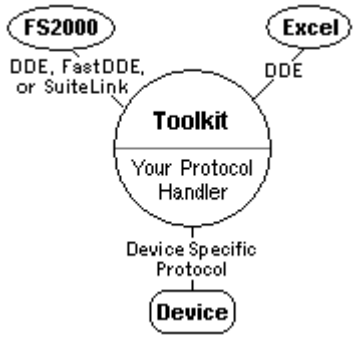
See the chapter on Time Marks for more information on optimizing how often you call DbGetGMTasFiletime( ).

There are four places where you should update the quality flags for a point:

1. In UdprotPrepareWriteMsg( ) if there is any problem with poked data. When a client "pokes" a value to the server, ProtNewValueForDevice( ) passes the value to the server-specific code. This, in turn, calls a routine such as UdprotPrepareWriteMsg( ), which creates the byte sequence that will be sent to the device. If a string is too long, or a value is out of range, the server-specific code should force the value to a legal setting, set a quality of WW_SQ_CLAMPLO or WW_SQ_CLAMPHI, and report the information back to the Toolkit with DbNewVQFromDevice( ). Also, if the point is inaccessible or read-only, it may be appropriate to set the quality to WW_SQ_NOACCESS and call DbNewVQFromDevice( ).
2. In UdprotExtractDbItem( ) when the response from a device is interpreted as point data and reported to the Toolkit via DbNewVTQFromDevice( ). When a response carries valid data for the point, the quality should normally be set to WW_SQ_GOOD. However, if a string is too long, or a value is out of range, or a data conversion yields an invalid result, the quality should be set to WW_SQ_CLAMPLO, WW_SQ_CLAMPHI, or WW_SQ_NOCONVERT as is appropriate. Also, some PLCs return status information along with the data. This status information should be mapped to the corresponding WW_SQ_XX quality setting.
3. In ProcessValidResponse( ) and other locations where an error response is received, a bad quality should be reported for each point on a message that has a response pending. If a polling (read) message elicits an error response, a quality of WW_SQ_NOACCESS should be reported for each point that is being actively polled by that message.
4. In UdprotSetTopicStatus( ) when the communication with the PLC fails, bad quality of WW_SQ_NOCOMM should be flagged on all points on all messages for that topic. This would be at the point where the server goes into slow poll mode, attempting to re-establish communications with the PLC.

See the chapter on Quality Flags for more information and specific examples of setting the quality flags.

CHAPTER 18



I/O Server Code Samples

Example source code for an I/O Server is included on the installation media. It will automatically be copied to your hard disk when you install the Toolkit. The sample code illustrates the concepts of the Toolkit, demonstrates basic problems that must be addressed for all I/O Servers (e.g., dialog boxes, file I/O for configuration), and gives you a starting point for developing code from scratch. The code includes a simple simulated PLC, so you can run the executable and see these concepts in action. Also, the sample server can be built in two ways – as a serial server and as a board server – to illustrate the similarities and differences between these two types of interfaces.

Contents

Overview

Wonderware Corporation has developed all of its I/O Servers using this Toolkit. The sample code included in the Toolkit is the basis for many of the production I/O Servers. We highly recommend experimenting with the sample server to learn more about the Toolkit's library functions. These sample programs will be in the following sub-directories:

\WW\IOSERVER\UDSAMPLE The sample code contained in this directory is a skeleton for a complete I/O Server. To build it as a functional EXE, you must copy three files from one of the subdirectories: either UDSERIAL or UDBOARD.

\WW\ IOSERVER\UDSAMPLE\UDBOARD This subdirectory contains three files: UDSAMPLE..C, UDSAMPLE.H, and UDSAMPLE.ICO. To build the sample server as a board server, type the following:

```
CD \WW\IOSERVER\UDSAMPLE\UDBOARD
COPY UDSAMPLE.* ..\COMMON
```

to copy the three files to the UDSAMPLE\COMMON directory with the appropriate names. This example illustrates communicating to a PLC via a memory-mapped interface device plug-in board (adapter card). The server program can directly access areas of memory that are mapped to a device I/O board's specific protocol.

\WW\ IOSERVER\UDSAMPLE\UDSERIAL This subdirectory contains three files: UDSAMPLE..C, UDSAMPLE.H, and UDSAMPLE.ICO. To build the sample server as a serial server, type the following:

```
CD \WW\IOSERVER\UDSAMPLE\UDSERIAL
COPY UDSAMPLE.* ..\COMMON
```

to copy the three files to the UDSAMPLE \COMMON directory with the appropriate names. The example illustrates communicating to a PLC via a serial interface (COM port).

The UDSAMPLE example server has symbol processing functions to handle many thousands of active data points. It shows you how to set up lists of polling messages, how to optimize polling messages, and much more. This example code is the basis for many of the Wonderware I/O Servers. UDSAMPLE contains everything that you need to write an actual I/O Server. It includes the user interface code, dialog boxes and the code necessary to interact with them, configuration file management, and the necessary data structures. It can be compiled for Win32 programming models. It contains functions to assist in debugging. It also contains device and protocol-specific code for a simple, simulated PLC, to illustrate how to construct polling messages and how to interpret the reply messages. This code and the comments should provide an excellent starting point for the development of a server for your specific needs.

UDSAMPLE Architectural Overview

There are essentially three parts of UDSAMPLE: one part establishes the configuration of the I/O Server, a second part collects and validates information requests, and the third part actually does the communication.

UDSAMPLE establishes its configuration settings via dialogs with the user. This is used to set up the I/O channels (COM ports or boards) and the available topics. The configuration settings are stored in a file which gets loaded from the disk when the server is started up.

The I/O Server receives its information via DDE or SuiteLink requests. The Toolkit calls functions in UDSAMPLE to set up and validate the topics and item names requested. Once the topic and item names have been validated, messages are created to read information from the PLC. UDSAMPLE is based on a master/slave communication model, where the server sends a message to the device and receives a response back.

The I/O Server builds two kinds of messages: *Read Messages* for all the active points and *Write Messages* for writing data back to the PLC.

Read Messages are built from the information in the symbol table and stay around. One read message may read data for several points. Read messages repeatedly get sent to the PLC for as long as the points are active. If points go inactive, the corresponding read messages may be modified or deleted, depending on what points remain active.

Write Messages are built on demand (when a DDE POKE or SuiteLink WRITE is received) and placed on the top of the message queue so they are sent on the next execution of the protocol. Generally one write message is generated for every point that is written, although it may be possible to optimize writes and send values for several points in a single message. Write messages are deleted after they have been successfully received and processed by the PLC.

The heart of the driver is in the **UdprotDoProtocol( )** function. This function executes the protocol. It examines the state of the server and executes the appropriate action. (For example, if the I/O Server is in the *idle* state, it sends the next message to the device. If it is in the *awaiting response* state, it looks for information from the device and processes that information.) The server contains necessary logic to handle error conditions.

Adapting the UDSAMPLE Server

There are several things that should be determined before you start coding a server:

I/O Channel: Does the server communicate over a serial port, a plug-in board (adapter card), an Ethernet connection? What configuration information is necessary to identify which particular I/O channel is being used? What settings need to be specified, e.g. transmission rates, parity, data encoding, etc.?

Protocol Structure: If the protocol supports multiple message types, decide which messages you will need to run the protocol. This should determine if there is any additional information required at configuration time.

Topic Description: The topic configuration generally contains all of the information needed to address a particular device. It contains a topic name, PLC address, update interval, and any other information generally required to build and send a message (except for the particular point to address).

Item Naming Convention: Item names should match the convention used in the PLC and provide you with enough information to determine the data type, address and possibly precision of the point. You should decide which points you are going to support. The sample contains one built-in item name: STATUS. This item name is a discrete variable that is TRUE when communication to the device is active and FALSE when communication is inactive.

Customizing the Start-Up Splash Screen and About Box

When an I/O Server starts up, it normally displays a *Splash Screen* for a few seconds. This is a window that identifies the I/O Server and the Toolkit, and any other information that seems appropriate. When the I/O Server is running and its window is visible, the user can select Help|About to display the *About Box*, an informational window that identifies the product, version number, company, etc.

UDSAMPLE has two macros that function as compile-time flags that can be used to select what kind of Splash Screen and About Box to use:

`#define USING_WW_COMMONUI`

This selects the Wonderware FactorySuite 2000 Splash Screen and About Box, which are normally displayed by a Wonderware FS2000 production I/O Server. Information for the program ID and version are provided in the *version marker code* section of the *.RC file. If this macro is not defined, UDSAMPLE will display a Splash Screen defined by the dialog resource named WWStartup. You can then use the dialog defined in TKITSTRT.RCI or define your own Splash Screen. You can also suppress the display of the splash screen entirely [see the section on the function **SetSplashScreenParams**( )].

`#define USING_WW_ABOUT`

This selects the Wonderware Common Dialog About Box (but only if the macro USING_WW_COMMONUI is not defined). See the section on the Wonderware Common Dialogs for more information on customizing the information displayed. If this macro is not defined, the About Box defined by the dialog resource ABOUT_BOX is displayed. You can customize this resource as needed.

UDSAMPLE also has a macro named USING_WW_LICENSE. If this macro is defined, the server calls the function **GetIOServerLicense**( ) to check whether the server has a license to run. This call requires a special version of the Wonderware Toolkit that provides access to the Wonderware License Manager. Contact Wonderware for further information.

Modifying the User Interface – UDSAMPLE.RC and UDCONFIG.C

The dialog boxes for generic Topic and Board configurations are contained in UDSAMPLE.RC. The corresponding dialog handlers are functions contained in UDCONFIG.C. These can be modified as needed for your protocol. (For example, you may need to differentiate PLC devices.) It is also a good idea to change the name UDSAMPLE to reflect the name of your PLC.

For configuring the serial ports (COM ports), the sample server uses the Wonderware Common Dialogs. See the chapter on the common dialogs for more information on using the optional edit fields, check boxes, and radio buttons.

Storing and Retrieving Configuration Files

Once configuration information has been entered via the user interface, it needs to be stored for retrieval next time the server is started up. The sample server handles this by storing the I/O channel configurations and Topic configurations in a disk file named UDSAMPLE.CFG. Two basic techniques are illustrated:

Configuration using binary structures – UDCFGBIN.C:

The data structures for COM ports, boards, and topics are simply written to a disk file as sequences of bytes, and are read back the same way. While this provides the simplest technique for configuration storage and retrieval, there are some shortcomings:

- If the structures are changed in any way, the format of the data file also changes. For backward compatibility, it is necessary to provide extra code to read old formats and convert to the new data structures.
- On different platforms, alignment issues may actually insert extra bytes between structure members. Care must be taken to ensure compatibility when moving between Win16 and Win32 systems.
- There is usually no verification that the configuration is valid, although verification can be added.
- Configuration must be done using the user interface with a running I/O Server. The configuration file can only be written or read by the I/O Server, and does not easily lend itself to independent verification or to being printed out.

Configuration using ASCII text – UDCFGTXT.C:

The data structures for COM ports, boards, and topics are written to a disk file as configuration “commands” with named parameters. When the file is read in by the I/O Server, the server interprets the commands and parameters. Commands can be in any order (e.g. COM ports then topics, topics then COM ports, or mixed). Within a command, parameters can be in any order. Any missing parameters are given default values; any obsolete parameters are merely ignored. A command can span several lines of text, and ends with a semicolon (‘;’). This technique requires a little more code, but confers several important advantages:

- If the structures are changed, no version tracking is necessary, and backward compatibility is automatic. (This can be even better than C++ serialization!)
- The configuration file is platform-independent. There are no alignment issues.
- As each command and each parameter is processed, the validity of the configuration can be verified.
- Configuration can be done without a server running, and on a completely separate machine. Configuration files can be verified and printed.

UDSAMPLE can read its own configuration files in either binary or ASCII text. By default, it stores configuration files in binary. To store configurations as text, in the [UDSAMPLE] section of WIN.INI, simply include the following line:

```
WriteConfigInASCII=1
```

This setting is read in when the server starts up, in the routine ProtInit( ).

Setting Up Data Points – UDCONFIG.C

First, determine the item/point naming convention. The item/point name should contain enough information for the user to know its data type and address. For example, the item name V23 indicates that this is an integer register in V-memory at address 23.

Function ValidatePoint( )

The function ValidatePoint( ) needs to be written to parse and validate the item names for your specific device. It is passed a text string of the item name, and the function must decode this into information that is eventually placed in the symbol table. This information will be used later to build the messages. The amount of information that needs to be stored into the symbol table depends on what information you need to build the message and extract the data back out. This structure may be modified to meet your needs. UDSAMPLE uses a temporary data structure LPPPS to store information about the item until it has been fully validated. The symbol table entry is built following a return from ValidatePoint( ).

Example of ValidatePoint: Valid Item Names are Vxxx

```
/*
    ValidatePoint() checks to see that a point name conforms to
        the convention:
    Vxxx
    The main purpose of the function is to fill in the LPPPS
        structure.
*/
BOOL
WINAPI
ValidatePoint( LPSTR lpszPointName,
               LPPPS ppp)
/*  use the LPPPS structure to temporarily store the decoded
    information.  UdprotAddSymbol will stuff this information
    into the symbol table entry when we are done.  */
{
    PSTR      pszReference;
    WORD      Reference       = 0;
    PTTYPE    ExplicitType    = NULL;
    BOOL      PointOK         = TRUE;
    char      tc;
    BOOL      input;
    BOOL      Regs;
    if (bVerbose){
        debug( "ValidatePoint: %Fs" ,
            lpszPointName );
    }
    ppp->ppsType      = 0;
    ppp->ppsNumReg    = 0; /* element size */
    ppp->ppsSeg       = 0; /* seg is a field added to the symbol
                              table */
                         /* to identify the segment */
   ppp->ppsAlias     = 0;
```

```
          lstrcpy( tmpBuf, lpszPointName );
    strupr( tmpBuf );
    pszReference = tmpBuf;
    switch( *pszReference ) {
        case 'V':
             ppp->ppsType  = PTT_INTEGER;
            /* check register specification */
             pszReference++;
             if( !NumericsOnly( pszReference ) ) {
                 PointOK = FALSE;
                 break;
             }
             Reference    = atoi( pszReference );
         /* range check the register address if appropriate */
             if( !ValidRegisterAddress( (DWORD)Reference, Regs )){
                 PointOK = FALSE;
                 break;
             }
             ppp->ppsAlias    = Reference - 1;
             /* zero based addressing */
             ppp->ppsSeg     = REGISTER;
             break;
         default:
             PointOK = FALSE;
             break;
         }
    if (bVerbose) {
         if( PointOK ) {
             debug( "Valid Point: %Fs" , lpszPointName );
             debug( "    ppsAlias=%05u    ppsSeg=%u" ,
                 ppp->ppsAlias, ppp->ppsSeg );
             debug( "    ppsType=%04u", ppp->ppsType );
             debug( "    ppsNumReg=%d  ", ppp->ppsNumReg);
         } else {
             debug("Item rejected: %Fs" , lpszPointName );
         }
    }
    return PointOK;
} /* ValidatePoint */

static
BOOL
WINAPI
NumericsOnly( PSTR pstr ) {
    if( !*pstr ) return FALSE;
    while( *pstr ) {
         if( !isdigit( *pstr++ ) ) {
             return FALSE;
         }
    }
    return TRUE;
} /* NumericsOnly */
```

Note The above code does not match the code on the installation media, however, this code is more useful for this example.

Executing the Configuration – UDLDCFG.C

This section describes the functions that build the message queues. They are found in UDLDCFG.C.

LogicalAddrCmp( )

The function LogicalAddrCmp( ) is the function used to sort symbol table entries. Fields from the symbol table entry, such as data type and address, are compared in this function to determine the sort order of the symbol table. It may be necessary to add additional fields to the comparison criteria to ensure that symbols are sorted properly. For example, the server might support bits in registers and the bit identifier would need to be added to the sort criteria. If the symbols are not in the correct order, the data will not be extracted properly.

Building Messages

The message data structure contains information about which symbols are referenced in the message and the actual message itself. UdprotAddPoll( ) contains the logic to build the read message, and UdprotPrepareWriteMsg( ) builds the write message. The actual message that is sent to the device is built by either BldRead( ) or BldWrite( ).

UdprotAddPoll( )

UdprotAddPoll( ) builds a read message to be placed in the message queue. Existing read messages are examined to see if the point can be read from an existing message. If it can fit into an existing message, the information in the LPMSG structure is updated. Otherwise, a new message is built and added to the message queue. BldRead( ) builds the actual sequence of bytes that is sent to the PLC.

UdprotPrepareWriteMsg( )

This function builds the write message. Existing write messages are examined to see if the point can be combined into a message that is already waiting to be sent. If it can fit into an existing message, the information in the LPMSG structure is updated. Otherwise, a new message is built and added to the message queue. BldWrite( ) builds the actual sequence of bytes that is sent to the PLC.

Building Messages – UDBLDMSG.C

BldRead( ), BldWrite( ), BldCks( ): These functions need to be supplied as they are totally device-dependent. Write the code to build the correct message according to your protocol. This message is typically a character buffer that is passed to the I/O driver via the function WritePortMsg( ). The message is part of the UDMSG data structure. Usually, a serial server uses **WriteComm( )** to pass the buffer to the Windows COM driver; while a board server uses a function tailored to the particular board interface.

Executing the Protocol – UDPROTCL.C

This section describes the functions that actually do the communication. The functions are found in UDPROTCL.C.

UdprotDoProtocol( )

This is the function called for every **ProtTimerEvent**. It contains the case statement that determines what happens at a given state. In UDSAMPLE, it is assumed that there are a limited number of states, namely IDLE, WAITRESP, and error conditions. If your device has more states than these, add these states and whatever action needs to be taken.

UdprotGetResponse( )

This function may need to be modified to determine if you have enough of the message to continue processing.

ProcessValidResponse( )

This function validates the response back from the device. (The response is in the lpPort data structure, field *mbRspBuffer*.) It may validate the checksum, look for a particular field in the message, or do anything required to ensure that the entire response has been received correctly.

UdprotExtractReadData( ), UdprotExtractDbItem( )

These functions extract information from the response based on information in the message structure and the symbol table. Modify these to suit your protocol.

UdprotHandleRspError( )

This function handles problems with the communication protocol: incomplete responses, error responses, timeouts, etc.

Data Structures

There are several built-in data structures that the server needs to build messages, send them, and extract the responses. These data structures may be modified to suit your needs.

PORT Data Structure

This data structure contains the information necessary to control the port in processing the protocol. It contains a handle to a linked list of nodes, as well as handles to the currently active node and message. It also contains the *mbRspBuffer* field which is the data coming back from the PLC.

UDMSG Data Structure (Message)

This is the message data structure. It contains the actual message to be sent to the device, a pointer to the topic, and indices into the symbol table for the first symbol contained in the message and the last symbol contained in the message. This information allows the information to be plucked out of the response.

STAT Data Structure (i.e. Station, Node, or Topic)

This structure contains all of the topic information. It has a handle to the associated symbol table and a list of messages sent to a particular node.

SYMENT Data Structure (Symbol Table)

This data structure contains a one-to-one correspondence with the items that are being requested (i.e. there should be one symbol table entry for every item requested). The symbol table should be sorted according to data type and address (at the very least). If the item is a bit address, the bit offset is often stored and sorted as well.

Note For Win16, the symbol table is an array of memory type HUGE, and consequently needs to be allocated with data structure sizes that are a power of 2. In Win32, HUGE does not exist, so the symbol table is a normal allocation.

Compiling the Sample Code

Note The I/O Server Toolkit for FactorySuite 2000 has been built with Microsoft Visual C++ version 6.0. Consequently, you must use version 6.0 sp3 or later to create Win32 I/O Servers using this Toolkit. After following the instructions below, you will be able to build, execute, and debug the sample server from the IDE.

Setting up a project for the sample code with Microsoft Visual C++ 6.0:

Create the new project:
- Click on File|New and select the "Projects" tab.
- Select Win32 Application
- For the project name, enter UDSAMPLE
- For the location, enter the path to the files, e.g., C:\WW\IOSERVER\COMMON\UDSAMPLE
- Click on Create New Workspace
- Click on OK

Copy the specific files for the desired server type to the UDSAMPLE directory

- For a board server, copy the files UDSAMPLE.C, UDSAMPLE.H and UDSAMPLE.ICO from the UDBOARD directory to the C:\WW\IOSERVER\UDSAMPLE\COMMON directory.
- For a serial server, copy the files UDSAMPLE.C, UDSAMPLE.H and UDSAMPLE.ICO from the UDSERIAL directory to the C:\WW\IOSERVER\UDSAMPLE\COMMON directory.

Set up the files for the project
- Click on Project|Add-to-Project|Files
- In the dialog, select the directory where the files are located
- Hold down the control key while clicking to select multiple files
- Add all the *.C files and the *.RC file
- Click on OK
- Click on Project|Add-to-Project|Files again
- In the dialog, select the directory where TOOLKIT7.LIB is located
- Click on TOOLKIT7.LIB
- Click on OK

Set up the appropriate compiler and linker options
- Click on Project|Settings
- Select the "C/C++" tab
- For "Category," select "Code Generation"
- For "Use run-time library," select "Multithreaded DLL"
- Select the "Link" tab
- For "Category," select "Input"
- Under "Ignore Libraries," enter LIBC
- Click on OK

Note The MSVC library OLEAUT32.LIB must also be linked to your project, either explicitly or as one of the "Object/Library modules" on the "Link" tab.

Set up the appropriate directories
- Click on Tools|Options
- Select the "Directories" tab
- For "Show directories for" select "Include files"
- Add the path to the Toolkit include files, e.g., C:\WW\IOSERVERTOOLKIT\INC. Move this path to the top of the list of directories using "Move Item Up."
- Also add the path to C;\Program Files\DevStudio\VC\include\sys. Click on OK

Debug and Support Functions

If you include **DebugMenu=1** in the [UDSAMPLE] section of the Windows WIN.INI file there will be several menu items that appear in the server system menu.

- *Dump* prints the current port, topic, message, and symbol table structures to the **Wonderware Logger**.
- *ShowSend* prints the messages sent to the PLC to the **Wonderware Logger**.
- *ShowReceive* prints the messages received from the PLC to the **Wonderware Logger**.
- *ShowErrors* prints enhanced error message information in the **Wonderware Logger**.
- *Verbose* prints program execution trace information in the **Wonderware Logger**.
- *ShowEvents* is primarily for tracing COM port activity for a serial server when an error occurs. Events such as transmit buffer empty are printed in the **Wonderware Logger**.

Two other menu items are specific to the simulated PLC in UDSAMPLE:

- *Simulator Read Paused* stops the simulated PLC from responding to read messages. This can be used to explore what happens when communication with the PLC fails.
- *Simulator Write Paused* stops the simulated PLC from responding to write messages. This can be used to explore what happens when the server has a chance to accumulate several write operations before the PLC is ready to process them.

If you are working with the Integrated Development Environment for a compiler, such as Microsoft Visual C++ 6.0 sp3 or later, you can use breakpoints, watch windows, and other tools. Note that if you do use a breakpoint to interrupt the normal operation of the program, remember that some activities – such as PLC "keep alive" handshaking – may time out.

For debugging, feel free to make liberal use of **debug**( ) statements sent to the **Wonderware Logger**. Hardware error and serious communications problems should always be sent to the **Wonderware Logger** for operator use. **Debug**( ) statements to the **Wonderware Logger** should be under menu control as Wonderware recommends that **Wonderware Logger** be running at all times.

For more detailed information on debugging refer to the "Debugging and Testing" chapter.

Simulated PLC

The sample I/O Server UDSAMPLE contains a simulator for a simple PLC. This simulator is not intended to reflect the operation of any actual PLC; but it does provide a way to illustrate the complete operation of the sample server. Points can be advised, requested, and poked just as if the server were connected to a real device. The simulator can be paused to explore what happens when communication gets lost, and resumed to explore how the server recovers when communication is restored.

The serial server UDSERIAL and board server UDBOARD talk to the same simulated PLC. The only difference is the communication protocol. The serial server uses hex ASCII strings to send and receive messages, e.g.

:81000301002B4F;

with the start of the message marked with a colon (':') and the end marked with a semicolon (';'). By contrast, the board server uses straight binary to send and receive messages, e.g.

81 00 03 01 00 2B 4F

Otherwise, the points available and the protocols are the same.

For the serial server, one simulated PLC is implemented for each of the COM ports 1..4. For the board server, one simulated PLC is implemented for each of the board segments 0xD000, 0xD400, 0xD800, 0xDC00.

The simulation has two PLC memory segments:

NAME	ADDRESS RANGE	DESCRIPTION	FUNCTION
V	1..512	variable memory	holds value steady
C	1..512	counter memory	increments each time read

Addressing wraps around, so V814 = V((814-1) mod 512)+1 = V302

C-memory counters are always accessed as 16-bit integers.

However, points in V-memory can be accessed in additional ways:

V<number>	16-bit integer	Example: V3
V<number>B	16-bit BCD integer	Example: V3B
V<number>S	16-bit signed integer	Example: V3S
V<number>D	32-bit DWORD	Example: V3D (V3 and V4)
V<number>R	32-bit float (real number)	Example: V3R (V3 and V4)
V<number>:<number>	Bit of word (read only)	Example: V3:12 (V3 bit 12)
V<number>-<number>	String, blank-terminated	Example: V1-5
V<number>-<number>B	String, blank-terminated	Example: V1-5B
V<number>-<number>C	String, C-style	Example: V1-5C
V<number>-<number>P	String, Pascal-style (length byte first)	Example: V1-5P

The PLC simulator in UDSAMPLE provides a way to illustrate various aspects of I/O Server design with actual, running code. Topics can be configured, modified, and deleted. I/O channels (COM ports or boards) can be configured and modified. Messages for reading and writing PLC point values are created, "sent," "received," and interpreted just as if there were a real device at the end of some kind of communication cable. In some cases, those messages are "optimized" to improve performance.

The use of the simulated PLC can be enabled or disabled at compile time, according to whether the macro

```
#define SIMULATING
```

is defined or not in the module UDSAMPLE.C.
This approach makes it possible to include a complete or partial simulator in the source code for a server and mask it out when the server is built for use with real I/O. As a failsafe, the first time the simulator is accessed, it puts the message

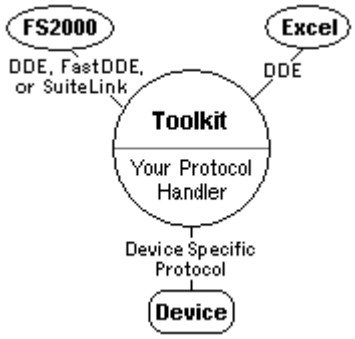
Simulated I/O – for test purposes only!

into the **Wonderware Logger**. This helps ensure that, if you forget to mask out the simulator, the log indicates that simulated I/O is being used.

While it may not be practical to include a complete or even partial simulator for an actual PLC in your server, this technique can often be a significant aid in debugging – especially if an actual PLC is unavailable for certain tests. See the chapter on debugging for additional ideas on testing and debugging your server.

CHAPTER 19

Debugging and Testing



This section presents a basic development and testing methodology that will assist you in completing the I/O Server quickly and efficiently.

Contents

Basic Programming Rules for Windows and Windows NT

The Toolkit uses C-language include files that use the WINAPI parameter passing option. We recommend following Microsoft guidelines for writing compatible code. The samples included have been written to compile Win32 models.

General Debugging Topics

Debug Messages

Liberal use of **debug**( ) message calls can greatly enhance your debugging arsenal. These message calls are most useful at decision points to record data and flow through the logic. The UDSAMPLE code described in the previous section illustrates how **debug**( ) calls can be used to describe the device status returns. Until you become familiar with the use of the Toolkit, we recommend that you put a **debug**( ) call at the entry and exit of every new routine.The following are a few suggestions for **debug**( ) usage:

1. Remember, always use far pointers to strings and be careful to keep the byte size of the parameters equal to the format statement elements.

Example:

```
debug( "UdprotFreeMsg( %04X )", (unsigned) hmsg );
debug( "UdprotFreeSymEnt( %Fp, %Fp, %d )", lpnode, lpSymTab,
        (int) indx );
debug( "UdprotUnchainSymEnt( %Fp, %Fp, %ld )", lpnode, lpSymTab,
        (long) lnum );
```

2. The **debug**( ) calls can take a lot of machine time to execute, especially if they are being logged to disk (check the display options in **Wonderware Logger**). Allow for this at first by selecting long timeouts, slow polling rates, and short messages.

When the server is completed, it is handy to leave a few protocol **debug**( ) calls in the code. By using either a menu selection, a WIN.INI variable, or a Registry setting, you can enable or disable these remaining **debug**( ) calls by putting the appropriate (if) statements around them. This finer level of information gathering is useful when an installation is first started up. When there is enough detail in the debug messages, they will assist in isolating hardware problems, configuration errors, user problems, and the lingering "one last bug" in the software.

DDE Message Traffic Monitoring Using WIN.INI

The Toolkit will handle the entire DDE protocol for the server, but in rare cases you may need to examine the DDE message traffic in detail. The Toolkit can be switched into debug mode by using WIN.INI variables. The Toolkit routines will access the WIN.INI file for a server by the name selected for the application name in the **ProtGetDriverName**( ) function.The following two WIN.INI variables can be set to 1 to force the Toolkit to send DDE debugging messages to the **Wonderware Logger**:

[SERVERNAME]
DebugDDEMessages=1
PreventBlockedDDE=1

Below is a sample from the **Wonderware Logger** with the DebugDDEMessages debug variable set. A simple sequence of INITIATE, REQUEST, POKE, ADVISE, UNADVISE, then TERMINATE was done from DDEAPP.EXE:

```
90/07/24 21:48:08.301/UDSIMPLE/rcvd WM_DDE_REQUEST   4848 24E8 0001 C37D [T1]
90/07/24 21:48:09.851/UDSIMPLE/sent WM_DDE_DATA      24E8 4848 154E C37D [T1]
90/07/24 21:48:21.099/UDSIMPLE/rcvd WM_DDE_POKE      4848 24E8 154E C37D [T1]
90/07/24 21:48:21.264/UDSIMPLE/sent WM_DDE_ACK       24E8 4848 8000 C37D [T1]
90/07/24 21:48:27.646/UDSIMPLE/rcvd WM_DDE_ADVISE    4848 24E8 15FE C37D [T1]
90/07/24 21:48:27.756/UDSIMPLE/sent WM_DDE_ACK       24E8 4848 8000 C37D [T1]
90/07/24 21:48:27.921/UDSIMPLE/sent WM_DDE_DATA      24E8 4848 1026 C37D [T1]
90/07/24 21:48:31.821/UDSIMPLE/sent WM_DDE_DATA      24E8 4848 15FE C37D [T1]
90/07/24 21:48:36.874/UDSIMPLE/sent WM_DDE_DATA      24E8 4848 1636 C37D [T1]
90/07/24 21:48:38.466/UDSIMPLE/rcvd WM_DDE_UNADVISE  4848 24E8 0000 C37D [T1]
90/07/24 21:48:38.631/UDSIMPLE/sent WM_DDE_ACK       24E8 4848 8000 C37D [T1]
90/07/24 21:48:43.289/UDSIMPLE/sent WM_DDE_TERMINATE 24E8 4848 0000 0000 []
```

When using **WindowViewer** as the client to debug the DDE traffic, set the flag PreventBlockedDDE to **1** to keep the server in the standard (slower) DDE mode. **WindowViewer** will normally use its own extra message blocking protocol to speed up the DDE transfers of large numbers of data points greatly.

Note: Remember to either set these both back to **0** or delete them from the WIN.INI file when the server application is complete. With either one of these variables set, performance will be severely degraded.

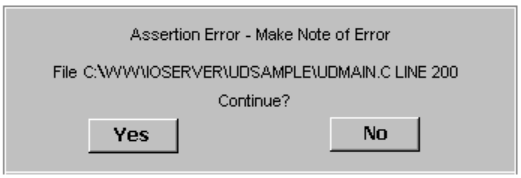
DDE Message Traffic Monitoring Using DDESpy

A Microsoft utility called DDESpy displays all the DDE messages on your computer and tags each message with a window handle that will distinguish the messages from multiple clients. An example of the level of detail that's displayed with each message is the DDE data message, where the output includes the source window handle, the data, the data format and the item name.

You can get DDESpy from the Professional version of Microsoft's Visual C++ or from CompuServe (enter GO WINSDK, file is DDESPY.ZIP). If you run DDESpy and get a GPF or the output suddenly stops updating, you may not have the latest version of DDESpy. Make sure you get the most recent version of DDESpy from CompuServe.

Assertion Errors

Under certain illogical or fatal conditions, an **Assertion Error** message box will be displayed showing a program line number, as shown in the example below:



An application can create these errors by calling ASSERT(boolean expression). If the boolean expression evaluates to FALSE, the message box will come up. You can use these calls at points in your code that should never happen or to indicate an illogical situation.

Note: Use ASSERT_ERROR rather than using ASSERT (FALSE).

When an assertion error comes from the Toolkit library function, write down the conditions that lead up to the error and the information in the message box. You can respond **Yes** to the message box and, depending on the severity of the error condition, the application may continue with no ill effects, get another assertion error, abort the application and produce a DrWatson listing, or hang the system. Therefore, continue at your own risk. If you respond **No** to the message box, the Toolkit will produce a DrWatson listing which can be valuable in tracking the problem. Refer to the *Microsoft Professional Tools User's Guides* for detailed information.

Another useful technique in tracking assertion errors that may come from the Toolkit is to set a breakpoint at AssertLog. Then re-create the error and look at the CALL trace to determine which routine in the server was called. Sometimes there are errors in the calling routines parameters that are invalid and these errors do **not** get trapped as invalid until they are several layers deep within the Toolkit.

Caution Note Assertion errors should be used sparingly since they will disable the operation of your server. They should not be used to indicate communication errors, nuisance errors, etc. Use the **debug**( ) function call to log these types of errors. As a general rule, if the server can recover gracefully from an error, do not use an assert call.

Prior to contacting Technical Support, try to recreate the error and make note of the file specification listed in the assertion message box. In many cases, the assert information will be copied to the **Wonderware Logger**, resulting in an entry similar to the one below:

```
90/10/22 16:13:25.542/UDSAMPLE/Assertion Error: "File
C:\WW\IOSERVER\UDSAMPLE\udmain.c Line 620"
```

Windows and Windows NT Debugging Tools

Microsoft Visual C/C++ Debugger

Microsoft Visual C++ provides an integrated debugger to debug programs. If using Visual C++ as the development environment, choose this full-featured debugger as the debugging tool. It is strongly recommended that when you install Microsoft Visual C++, you also perform the optional step of installing the Win32 symbol table. This will facilitate tracing your program even when it makes calls to the operating system.

The Toolkit Library is now supplied with debugging information (symbols, programmers database, and browsing). This requires Microsoft Visual C/C++ Version 6.0 sp3 or later.

Microsoft WINDBG Debugger

The Microsoft Windows NT SDK includes a powerful symbolic debugger called WINDBG. If using the Windows NT SDK to develop a server, this debugger is an essential tool for diagnosing and solving problems with code.

NuMega Bounds Checker

This tool is often of great help in tracking down memory leaks, bad memory pointers, and out-of-range indexes. It does require a special effort to build your program for this debugging tool, but it does provide a way of trapping mismatched memory allocations/deallocations, resources accessed but not released, and invalid indexes.

NuMega Soft Ice

This tool can be useful if you find yourself debugging a server that uses drivers, such as VxD programs for Windows 98 or kernel device drivers (KDDs) for Windows NT.

Rational/Pure/Atria Quantify Performance Monitor

This tool is of most value when your server is already running, but you want to look for ways to improve performance. It does require a special effort to build your program for this evaluation tool, but it does provide a way to examine areas where program efficiency could be improved.

Testing

WWClient Usage

The next most useful tool for debugging the new server code is the Wonderware program WWClient that is included on the Wonderware Toolkit installation disks. This program allows you to manually exercise DDE and SuiteLink functions in order to test the server. (The program is discussed in the chapter entitled "I/O Server Code Samples".) This program will act as a client application. To use it, run C:\Program Files\FactorySuite\COMMON\wwclient and select the various menu options. The operations we will discuss are:

Connect
Register
Advise
Unadvise
Request
Poke
Unregister
Disconnect

By using WWClient you can simulate all of the typical client DDE and SuiteLink actions. Follow the sequence listed below to observe the basic operations of the new server:

1. Select *Connect* to bring up the Connect dialog. Enter the name of the node to which you want to connect, the application name of the server, and a valid topic name for the server. Then select whether the type of connection you want: DDE or IOT for SuiteLink. [Note: If WWClient and your server are on the same machine, the node name can be left blank for a DDE connection, but not for a SuiteLink connection.] Finally, click on the "Connect" button. This will result in a call to **ProtAllocateLogicalDevice**( ) and you can watch the server do its protocol setup. If the connection is successful, a status line will appear in the WWClient main window. An important test of the server logic is to use an invalid topic name. Also, you can open up connections to several topics and several different servers from this dialog. Click on the "Done" button when you are finished.

2. Select *Item* to bring up the Item dialog. Select which connection you want to use, then enter a valid item name. Then click on the "Register" button. This will not result in any calls to the server itself. However, an entry for each point registered will appear under the corresponding topic in the WWClient main window. You can also register a range of points in a single command. For example, typing "V1..5" and clicking on "Register" will register points V1, V2, V3, V4, and V5. When you register points under different connections, the points will be listed in the WWClient main window under their corresponding connections.

3. With a connection selected and a valid item name or range in the "Item" edit box, click the "Advise" button. This will result in two calls to the server: **ProtCreatePoint**( ) and **ProtActivatePoint**( ). At this point, the server should begin gathering data and return it by calling the Toolkit **DbNewVTQFromDevice**( ) routine. The new data will be displayed in the WWClient, and any of your debugging messages should be examined. Another important test is to use an invalid item name.

4. With a connection selected and a valid item name or range in the "Item" edit box, click the "Unadvise" button. Now **ProtDeactivatePoint**( ) will be called. This should have exercised all of the basic data gathering paths.

5. With a connection selected and a valid item name or range in the "Item" edit box, click the "Request" button. If the point is not already on advise, you will see calls to **ProtCreatePoint**( ) and **ProtActivatePoint**( ), followed by updates to the Toolkit via the **DbNewVTQFromDevice**( ), and a call to **ProtDeactivatePoint**( ). You will also see current value for the point appears in the WWClient main window. To examine what happens with a point that is already on advise, you can use two instances of WWClient, one of which has the point on advise, and use the other instance to request the point data. The instance of WWClient that makes the request will show the current data value at the time of the request, but no updates to that value will occur – even though updates are taking place in the other instance of WWClient. See note below regarding multiple clients.

6. With a connection selected and a valid item name or range in the "Item" edit box, select the type of data you want to send (Integer, Real, Discrete, or String) and enter a new value in the "Value" edit box. Then click on the "Poke" button. Now you will see a call to **ProtNewValueForDevice**( ) with the new data to be written by the server to the device.

7. With a connection selected and a valid item name or range in the "Item" edit box, click on the "Unregister" button. The listing for the point will be removed from the main WWClient window. If the point is on advise, you will also see a call to **ProtDeactivatePoint**( ).

8. Close the Item dialog box by clicking the "Done" button. Select *Disconnect* to close the connection. If there is only one connection active, it will be closed. Otherwise, the Disconnect dialog box will be displayed. You can close a particular connection or all connections. Closing a connection will invoke a call to **ProtFreeLogicalDevice**( ) and effectively shut down that topic in the server. See note below regarding multiple clients.

If you have multiple clients connected to the same topic, the call to **ProtAllocateLogicalDevice**( ) only gets issued when the first client connects to the topic. Likewise, **ProtFreeLogicalDevice**( ) gets called only when the last client disconnects from the topic. Similarly, **ProtCreatePoint**( ) and **ProtActivatePoint**( ) get called only when the first client connects to the point, and **ProtDeactivatePoint**( ) gets called only when the last client disconnects from the point.

After the basic single topic, single item paths through the server have been tested. Now you can move on to a more complicated test. Keep in mind that nearly all of the server functions can be exercised using this simple tool. Plan your testing around the boundary conditions of your protocol. The following are a few other basic test suggestions:

1. If you have implemented the STATUS item for the server's topics, use WWClient to test its operation while causing protocol error conditions. Most of the protocol error handling can be tested during this phase.
2. You can use multiple copies of WWClient to exercise overlapping operations. For example, advise several points to check the protocol for optimizing its polling, and do pokes to others points.
3. Clean up after topic termination (call **ProtFreeLogicalDevice**( )) can be tested. Have WWClient *Advise* several points and then do a *Disconnect* (without doing an *Unadvise* first).
4. Clean up after shut down (calls to **ProtClose**( )) can be tested by closing the server while several points are *Advise*d.

Scripts for WWClient

WWClient can be controlled from a script, so that complex or repetitive tests can be recorded and replayed again later.

```
;Script test.scr
;Perform repeated connect, advise, and disconnect

Loop:-1
Connect_iot:\\FStest05\udsample|topic1
Register:v1..10
Advise:v1..10
Pause:1000
Disconnect:\\FStest05\udsample|topic1
Pause:1000
LoopEnd:
```

Microsoft Excel Usage

Any cell in Excel can be set to input DDE data from the server. This can be useful for testing and obviously for the completed application usage. To cause Excel to read DDE data, select a cell and then enter the full address formula:

= Application|Topic!Item

For example:

=UDSAMPLE|Trans1!T1
=View|Tagname!ReactorLvl
=MODBUS|ReactorPLCFast!'40001'

If the item name contains special characters, Excel may try to evaluate the expression. It is useful to put apostrophe marks (single quotes) around the item name.

In addition, Excel can be forced to do *Poke*s. The following two example pokes can be used:

1. Using DDEView.

   DDEView can be used to poke a single cell to an item or columns of cells to multiple items. DDEView is shipped with Wonderware's NetDDE for Windows or is available in the utilities section of Wonderware's Comprehensive Support CD. Documentation on the installation and usage of DDEView is available in Appendix C of the NetDDE for Windows User's Guide that comes with Wonderware's NetDDE for Windows and the DDEView on-line help.

2. Using Excel 5.0 VBA macros.

   An example of how to poke a single cell to an item using an Excel 5.0 VBA style macro:

```
Sub GetUDSERIAL( )
    Dim rangeToPoke
    Dim channelNumber

    channelNumber = Application.DDEInitiate("UDSERIAL", "topic")
    Set rangeToPoke = Worksheets("Sheet1").Cells(1, 1)
    Application.DDEPoke channelNumber, "R1", rangeToPoke
    Application.DDETerminate channelNumber
End Sub
```

Index

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W