Wonderware® FactorySuite® InTouch® Extensibility Toolkit

User's Guide

Revision B

January, 2000

Wonderware Corporation

All rights reserved. No part of this documentation shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the Wonderware Corporation. No copyright or patent liability is assumed with respect to the use of the information contained herein. Although every precaution has been taken in the preparation of this documentation, the publisher and author assume no responsibility for errors or omissions. Neither is any liability assumed for damages resulting from the use of the information contained herein.

The information in this documentation is subject to change without notice and does not represent a commitment on the part of Wonderware Corporation. The software described in this documentation is furnished under a license or nondisclosure agreement. This software may be used or copied only in accordance with the terms of these agreements.

ã **2000 Wonderware Corporation. All Rights Reserved.**

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

Trademarks

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. Wonderware Corporation cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

Wonderware, InTouch and FactorySuite Web Server are registered trademarks of Wonderware Corporation.

FactorySuite, Wonderware FactorySuite, WindowMaker, WindowViewer, SQL Access Manager, Recipe Manager, SPCPro, DBDump, DBLoad, HDMerge, HistData, Wonderware Logger, Alarm Logger, InControl, InTrack, InBatch, IndustrialSQL, FactoryOffice, FactoryFocus, License Viewer, Scout, SuiteLink and NetDDE are trademarks of Wonderware Corporation.

Contents

CHAPTER 1

Introduction to the InTouch Extensibility Toolkit

The Wonderware® InTouch® Extensibility Toolkit is designed to allow a proficient Windows C/C++ programmer to expand, customize, and add new functionality and capabilities to the InTouch development or run-time environments. By offering powerful development tools, and combining them in an open InTouch environment, the Toolkit allows the creation of more powerful, user specific applications than otherwise possible.

Contents

About the InTouch Extensibility Toolkit

By using the InTouch Extensibility Toolkit, a developer can create Wizards, build custom script functions, and access the InTouch database from other programs through standard APIs or OCXs. Outside programs can be quickly linked in, and custom DLLs can be simply created and utilized.

The following briefly describes each major section contained in the InTouch Extensibility Toolkit:

- **Wizard Software Development Toolkit**

  The Wizard Toolkit provides the developer with the ability to create libraries of wizards that will integrate into the InTouch environment and provide InTouch application developers with a higher level of productivity. Wizards are created to extend the InTouch functionality for a set of specific applications, or for general InTouch development.

- **InTouch Script Functions Software Development Toolkit**

  Script functions can be developed with the Script Functions Toolkit for use within InTouch. These functions are able to utilize a rich set of conditional statements, functions, and data operators available in the C and C++ programming languages. The completed function(s) are integrated with the development environment (WindowMaker™) so that they appear in the script function selection dialog box. Scripts can be initiated by data change, pre-defined conditions and/or operator action. Scripts may also run in the background of the application or based on a window being active.

- **IDEA Software Development Toolkit**

  The IDEA (InTouch Database External Access) Toolkit provides developers with a means of directly accessing data in the InTouch tagname database. IDEA supports the following:

  - Developers who wish to produce separate Windows programs that access and/or change InTouch data
  - Developers of InTouch script functions, who wish to read/write InTouch data from a script function that resides in a DLL
  - Programs written in C/C++ and Microsoft Visual Basic™

- **ITEdit OLE Control**

  ITEdit.OCX is a 32-bit control that can be used in Visual Basic or any other OLE container that provides support for OLE controls. In Visual Basic, InTouch data can be read and written using ITEdit. Only tagnames and access modes need to be defined in Visual Basic. ITEdit provides an event mechanism that responds to tagname value changes in InTouch and monitors whether or not InTouch is running. Also, there is no limitation on the number of applications can access InTouch's database concurrently. ITEdit functionality is provided via the newly created class CIdea, which is a C++ wrapper around the functionality provided within PTACC DLL. It provides access to these features without having to learn the intricacies of PTACC.

- **Tag Access ActiveX Objects**

  Tag Access ActiveX objects provide developers using Microsoft Visual Basic's rapid application development (RAD) tools with the ability to quickly deploy applications that link to the InTouch runtime database. Tag Access is a set of components for anyone planning to integrate InTouch with Visual Basic, plus a set of utility applications and components. Because they utilize standard ActiveX technologies, these tools are also useful in any of the Microsoft Office™ applications as they can be used from within Visual Basic for Applications (VBA) to expose an InTouch tag database object model. These tools can be used to develop extensions to InTouch in a variety of ways:

  - Develop stand-alone applications that integrate with InTouch, such as custom data loggers, setpoint downloading, statistical/advanced numerical analysis, custom InTrack™ clients, and so on.
  - Create ActiveX servers that can be called from within the InTouch scripting environment, allowing Visual Basic to be used for the application scripting.
  - Embedded into other ActiveX controls, enabling them to be used in the creation of custom ActiveX control objects such as special types of animations, charts, or user interface objects that can be used in InTouch or Visual Basic and are bound to data in the InTouch tagname dictionary.

Installing the InTouch Extensibility Toolkit

The InTouch Extensibility Toolkit software package is distributed on a compact disc which runs on the Microsoft Windows 98 and Windows NT (4.0 or later) operating systems. The installation program creates directories as needed, copies files from the CD to your hard drive, and creates the InTouch icons on the Windows **Start** menu.

Hardware/Software Requirements

The following hardware and software is required to use any of the components contained in this toolkit:

- IBM-compatible PC capable of running the Windows 98 or Windows NT 4.0 operating system
- Memory recommendation of at least 16 MB, preferably 32 MB on Windows 98 and upwards of 64 MB to 128 MB if running on Windows NT
- InTouch 7.1 (or later) for Windows 98, or InTouch 7.1 (or later) for Windows NT 4.0 with Service Pack 5
- Windows 98 or Windows NT Software Development Kit (SDK), if not using Microsoft Visual C++
- C or C++ Development Environment capable of creating Windows Dynamic Link Libraries (DLLs). For example, Microsoft Visual C++ Version 6.0 with Service Pack 3

Note In order to completely support wizard development, we recommend the use of Microsoft Visual C++ Version 6.0 with Service Pack 3.

Windows 98 and Windows NT Compatibility

The InTouch Extensibility Toolkit supports both the Windows 98 and Windows NT environments. It is possible to develop Wizards, Script functions, and IDEA support that is compatible with Windows 98 and Windows NT. The InTouch Extensibility Toolkit includes samples and development files that make it possible to develop code that is common to both Windows platforms.

The toolkit user should follow Microsoft guidelines for developing Windows 98 and Windows NT compatible source code.

InTouch Extensibility Toolkit Version 7.1 (or later) only supports development of 32-bit executables (DLL or EXE) on Windows 98 and Windows NT.

Developer Requirements

This manual is written for experienced Windows C/C++ programmers. It assumes that you possess basic Windows application development knowledge and are familiar with the InTouch software. The basic skills and any specific requirements for a user of this toolkit are briefly described as follows:

- **C or C++ Programming**

  The InTouch Extensibility Toolkit currently supports the C and C++ programming languages. The functions specifically provided in the Wizard Toolkit API allow you to develop wizards that have the same "look and feel" as all other InTouch objects. In order to support a common interface for all wizards, standard tools such as color, font and tagname selection dialog boxes can be accessed from the wizard dialog boxes by simply calling Wizard Toolkit API functions.

- **Basics of Windows Application Development**

  The developer must possess basic knowledge of Windows application development such as, how to develop a dialog box. Sample wizards are provided in the Wizard Toolkit that can be used as templates when you start your wizard development. You also need to know the fundamentals of creating a Windows DLL, since wizards are added to the InTouch environment through the DLL mechanism.

- **Constructing Cells in InTouch for Prototyping**

  A Wizard developer needs to become familiar with prototyping wizards in InTouch.

  For more information on prototyping wizards, see Chapter 2, "Getting Started with the Wizard Toolkit."

  All of the objects (database tagnames, drawing objects and animation links) created or manipulated with the Wizard Toolkit are objects available directly in WindowMaker. The following briefly describes prototyping:

When the Wizard Toolkit is installed, the **Generate Wizard** command is automatically added to the WindowMaker **Special** menu. (We highly recommend that you use this command to automatically generate the code for your wizards.) By using this command, you can prototype the wizards you create using InTouch.

To prototype a wizard, you simply create the desired object in WindowMaker (using the standard tools), associate the object to a database tagname, assign the desired animation links, and so on, then make the object into a cell. Once the object has been made into a cell, select the cell and then select the **Generate Wizard** command. Selecting this command will create a file named WIZARD.C in your InTouch application directory.

The WIZARD.C file contains the Windows code required to re-create that cell. This code must be compiled and linked with the other required routines to form a DLL. Windows standard DLLs provide the mechanism to add wizards to the InTouch environment.

Note We recommend the latest edition of the book *Programming Windows* by Charles Petzold for those unfamiliar with Windows programming.

Documentation Conventions

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

Convention	Description
Bold Text	Denotes a function name, for example, **Wizard_New**
Italic text	Denotes a parameter value, for example, *wCommand*
CAPITALS	Indicates return type (or most return types) also filenames and paths.
`Courier 9`	`Code Examples and Syntax spacing samples.`

Terms Used in this Document

Term	Definition
Wizard Developer	Proficient Windows C Programmer; person developing the code for the wizard.
InTouch application developer	Person using the wizards after they have been developed and installed in InTouch WindowMaker.
User	Same as InTouch application developer.

CHAPTER 2

Getting Started with the Wizard Toolkit

The Wizard Toolkit contains tools and information necessary to develop a Wizard of any kind. The included sample source code provides a good starting point for a Wizard developer. The Wizard Toolkit also provides the Wizard developer with utilities to create individual Wizards that can be packaged for distribution with the InTouch family of products.

Wizards are implemented by using a set of Application Programming Interfaces (API) provided by the Wizard Toolkit. The Wizard Toolkit API contains numerous functions for manipulating and creating InTouch objects and database entries including, tagnames, Access Names and Alarm Groups.

This chapter is a tutorial for new Wizard developers. Its objective is to allow you, the Wizard developer, to quickly familiarize yourself with the basics of Wizard development. This tutorial will take you through the development process required to create your first Wizard, including all of its required components. You will learn how to put multiple Wizards in a "library," to make your Wizards configurable and how to change the properties of your Wizards, such as blink speed, location links, size and color links, and so on.

Contents

What is a Wizard?

Before Wizards were introduced, the InTouch application developer created objects on the screen by using the primitive WindowMaker tools. The developer then double-clicked on the object to associate animation links, such as a Value Output, a Color Fill, or a Blink Link to the object. When desired, logic scripts were also attached to the object for more in-depth control. The application developer might then group several objects together into a cell to create a "boilerplate" that could be used again and again. This process worked well, but it had its limitations. Wizards can now erase those limitations and automate the entire process for the InTouch application developer!

Wizards, in their most basic element, could be referred to as "smart objects" that make InTouch application development easier and more efficient. When using a Wizard, you are activating a set of pre-programmed actions that can create or manipulate standard InTouch objects (symbols and cells), primitives (lines, rectangles, buttons, and so on.) and database entries (tagnames, Access Names). All the InTouch application developer needs to do is select and configure the Wizard -- InTouch will draw it, animate it and define it if need be!

To use a Wizard, the InTouch application developer simply selects it from the toolbar, *places* it on the screen, and *configures* it by double-clicking on it and entering values into a standard Windows dialog box. For example, if the Wizard were a slider, its configuration dialog box would include items such as the tagname, the min. and max. range labels, the slide movement, fill color, and so on. Once this information is entered in the dialog box and you click **OK**, the Wizard is redrawn (automatically) and the slider is now ready to be used in WindowViewer™. It's that simple! The InTouch application developer is not responsible for drawing the individual components that make up the slider. The developer is not responsible for typing in the value ranges on the object. The developer is not responsible for animating the object(s). It's all done automatically!

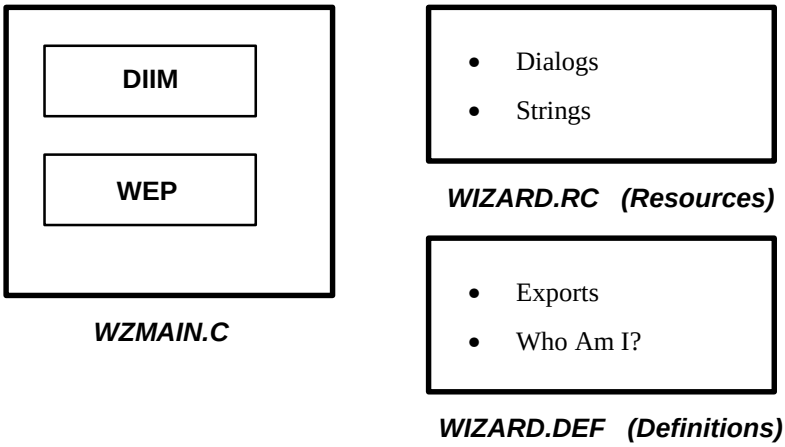
It is the Wizard developer who determines what actually happens when an InTouch application developer uses a Wizard (and makes its all seem automatic). The Wizard developer creates a DLL file that contains the functions and procedures that draw the lines, circles, and text necessary to create the object. WindowMaker calls this DLL when the Wizard is selected and again when it's configured. The Wizard DLL controls all the drawing and all the editing. The Wizard Toolkit provides over 60 API functions that may be utilized by the Wizard developer to do this.

For more information on the API functions, see Chapter 6, "Wizard API Reference."

The Components of a Wizard DLL

Wizards can vary in complexity and functionality. The simplest Wizard to create is a graphical object that is scaleable. A complex Wizard can involve sizable text, scripts, tagname creation, and contain properties that actually affect the extent of the Wizard itself. All Wizards contain a minimum set of components, and therefore all Wizard DLLs must contain a base set of functions. These base functions include both those components necessary to make the program a Windows DLL and those necessary to make it a Wizard DLL. The following briefly describes the Windows components.

There are usually three files that are compiled together to make up a 32-bit Windows DLL, as shown in the following illustration:



The three files include:

WZMAIN.C	Provides the DllMain (Library's main entry point) function.
WIZARD.RC	Provides resources (Dialogs, Bitmaps, and Strings).
WIZARD.DEF	Defines exports (HEAPSIZE, LIBRARY, and so on).

These files provide a basic Windows DLL. Since this is a DLL, inside the WZMAIN.C file, we must supply a DLL entry function (DLLMain for 32-bit Windows).

The following DLLMain should be used in Windows NT (32-bit) to perform the appropriate initialization for use with WindowMaker.

```
int
WINAPI
DllMain( HANDLE hInstance, DWORD ul_reason_being_called,
LPVOID lpReserved )
{
    switch( ul_reason_being_called ) {
    case DLL_PROCESS_ATTACH:
        // Initialize needed globals
        hDrawInst = hInstance;
        hDrawWnd = FindWindow( "Wmak Class", NULL );
        break;
    case DLL_THREAD_ATTACH:
        break;
    case DLL_PROCESS_DETACH:
        break;
    case DLL_THREAD_DETACH:
        break;
    default:
        break;
    }

    return TRUE;
}
```

Next, we need to include the functions that make this a wizard DLL. To understand these functions, we need to discuss what is going on when wizards are used by the InTouch application developer while drawing in WindowMaker.

Wizard Basics

To WindowMaker, a Wizard is like any other native object (rectangle, ellipse, line, and so on) in that the application developer will want to place it on a window, size it, animate it, and click on Undo to correct a mistake made with it. The difference between a native object and a Wizard is that instead of WindowMaker actually doing these things, WindowMaker will call the Wizard DLL to do them. To support these operations, the Wizard developer must provide specific functions within the Wizard DLL.

In general, WindowMaker will expect the following services (Modes) from the Wizard DLL:

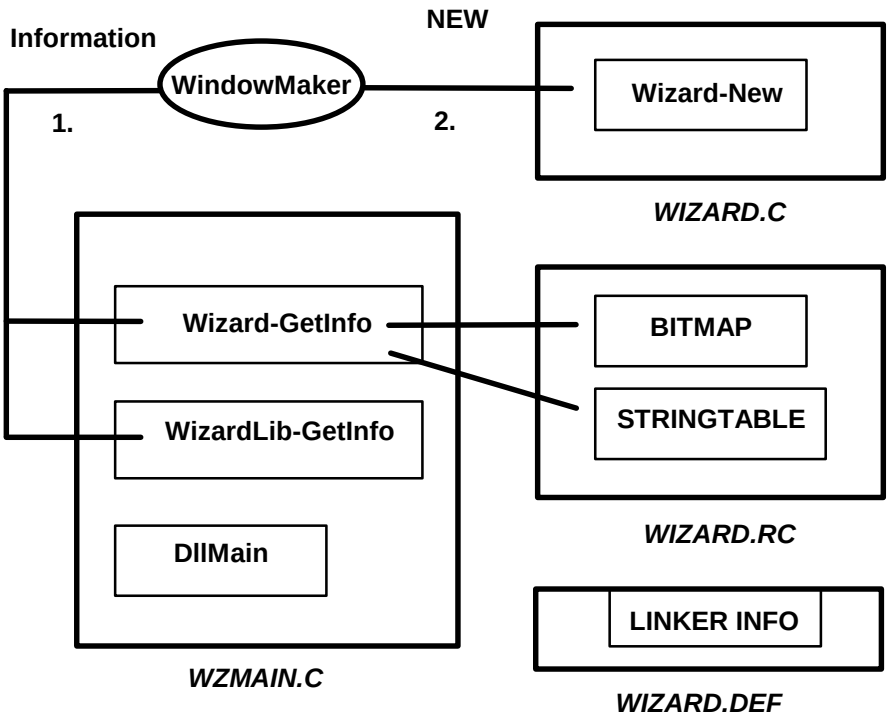
Mode	Application Developer Action
NEW	Placed a new Wizard on the window.
SIZE	Dragged the handles on the Wizard to resize it, either bigger or smaller.
EDIT	Double-clicked on the Wizard to make changes in its appearance or the way it was animated.
RESTORE	Selected the **Undo** command on the **Edit** menu to reverse the last action performed on the Wizard.

In addition, WindowMaker will also expect the Wizard DLL to provide descriptive information about the Wizard (bitmap pictures, text descriptions, version numbers, and so on). WindowMaker then uses this information in its Wizard Selection Dialogs and Wonderware Logger™ entries. The Wizard DLL accommodates information requests by providing functions that support certain information commands. Examples of these are listed in the following table:

Note This is not a complete list of the information commands that the Wizard will need to support. The list is included here only as an example. The exact details are covered later in the manual.

Information Command	Description
WIZ_DESCRIPTION	Long Description and Short Comment that are used in the Wizard Selection Dialog and the Toolbox.
WIZ_BITMAP	Large (64x64) bitmap that is used in the Wizard Selection Dialog.
WIZ_TBOXBITMAP	Small (16x16) bitmaps to be used when creating a button for the Wizard in the toolbar.
WIZ_GROUPNAME	Name of the group in the Wizard Selection Dialog where the Wizard will reside.
WIZ_FLAGS	An indicator of what type of Wizard this is.

To support the Modes and Information Commands, a typical Wizard DLL is structured as follows:



1. When WindowMaker needs to retrieve information about the Wizard (bitmaps, descriptions, help files, and so on.) it will call one of two specific functions in the Wizard DLL; **Wizard_GetInfo** or **WizardLib_GetInfo**. WindowMaker then passes the particular Information Command it is looking for to the Wizard. (These Information Commands include those in the list previously discussed. For example, WIZ_DESCRIPTION, WIZ_BITMAP.) The Wizard developer needs to provide the **Wizard_GetInfo** and **WizardLib_GetInfo** functions, and place these in the WZMAIN.C file.

   Wizard_GetInfo and **WizardLib_GetInfo** will then load bitmaps or strings from the Wizard DLL's resources (specified in the WIZARD.RC file).

2. When the InTouch application developer selects a particular Wizard from the toolbar and places it in a window, WindowMaker considers this Wizard to be in the NEW mode. To handle the processing of a NEW mode Wizard, WindowMaker calls the **Wizard_New** function in the Wizard DLL. **Wizard_New** is then responsible for "drawing" and "placing" the Wizard in the window. **Wizard_New** may include the Wizard API routines that actually "draw" the Wizard or it may dispatch another routine to do it.

When the InTouch application developer resizes a Wizard, WindowMaker considers this Wizard to be in the SIZE mode. To handle the processing of a SIZE mode Wizard, WindowMaker will again call the **Wizard_New** function. **Wizard_New** in this case is responsible for redrawing the Wizard at the new size or dispatching a routine to size it. The Wizard developer is responsible for providing the **Wizard_New** function (and the dispatching routines). This function will normally be placed in a file called WIZARD.C. The dispatch routines (if they exist) would normally reside in separate source files.

For more information on dispatching, see the "Creating Libraries with Multiple Wizards" later in this chapter.

Note The Wizard developer does not have to write all the **Wizard_New** drawing code from scratch. Once the InTouch Extensibility Toolkit has been installed on the developer's PC, a new menu command, **Generate Wizard**, will automatically be added to the WindowMaker **Special** menu. By using this command, the Wizard developer can select a cell in WindowMaker then invoke the command to automatically generate the code. WindowMaker will create the WIZARD.C file and put a **Wizard_New** function inside.

We strongly recommend before you begin developing Wizards, that you take the time to familiarize yourself with Wizard naming conventions. It is not mandatory that you follow these guidelines, but putting them into practice will make it easier to maintain and properly organize your Wizard libraries.

For information on Wizard naming conventions, see Chapter 5, "Style Guide for Wizard Library Development."

Simple Wizard .DEF File Example

Now that we have diagrammed what a simple Wizard DLL looks like, we can create our WIZARD.DEF definition file. The definition file contains information about the library, and, most importantly, the names of any Windows routines that must be exported. (Any routine that is called from outside the DLL must be in the export section.) In the case of a Wizard DLL, the functions that must be exported are those that are called from WindowMaker, like **Wizard_New**, **Wizard_GetInfo**, and **WizardLib_GetInfo**.

```
LIBRARY wzsample
DESCRIPTION 'WIZARD Toolkit Sample DLL'
''
EXPORTS
    Wizard_New
    Wizard_GetInfo
    WizardLib_GetInfo
```

Building a Simple Wizard

In this section, we are going to develop a simple Wizard and see how all of the pieces fit together. The simplest Wizard to create is a scaleable object with no text. Let's start prototyping our Wizard in InTouch by creating an object.

1. Start WindowMaker and create a new window by selecting the **New Window** command on the **File** menu.
2. Draw a rounded-rectangle using the standard WindowMaker tool.
3. Double-click on the object to access the animation links selection dialog box. Click on the **Blink** button in the **Miscellaneous** section to assign a blink link to the object. The **Object Blinking -> Discrete Value** dialog box will appear.
4. Enter a Discrete tagname in the **Expression - Blink When** field and then click **OK**.
5. Transfer to WindowViewer and change the value of Discrete to make sure the object is functioning as we would expect it to. Then switch back to WindowMaker.
6. Select the object again and then, on the **Arrange** menu, select **Make Cell** to turn the object into a cell:
7. With the cell still selected, on the **Special** menu, select **Generate Wizard** to automatically generate the code for the Wizard.

The generated code is now contained in the WIZARD.C file. The WIZARD.C file, along with the other files provided in the Wizard Toolkit, will be the basis for our simple Wizard.

WIZARD.C File

When you select the **Generate Wizard** command on the **Special** menu, the C code for the **Wizard_New** function is automatically written to the WIZARD.C file in your InTouch application directory. The WIZARD.C file contains only the code necessary to re-create that cell. This code must be compiled and linked with the other required functions to make a Wizard DLL.

Note The other routines that are required to link with WIZARD.C are contained in the WZMAIN.C, WIZARD.RC, and WIZARD.DEF. These files are diagrammed in the previous section, "The Components of a Wizard DLL."

For more information on these routines, see the next section, "Integrating a Wizard into WindowMaker."

Our example **Wizard_New** routine first calls **WWKit_Init** to initialize the Wizard library. Then the code sets the pens and brushes that will be used in drawing the object, sets any animation links that were assigned to the InTouch cell, and draws the object. All of these actions are performed by the Wizard API. (The Windows GDI routines are not used.)

```
#include <windows.h>
#include "wizapi.h"
#include "wizstub.h"

int FAR PASCAL Wizard_New(
    HCHUNK  hChunk,
    int     index,
    int     left,
    int     top,
    int     right,
    int     bottom,
    LPSTR   dllName,
    WHMEM   whData,
    int     mode,
    RECT    prevRect,
    WHMEM   FAR *pwhWizard)
{
    WHMEM   whHLObj;
    WHMEM   whObj;
    RECT    oldRect;
    RECT    newRect;

    WWKit_Init();
    SetRect( &oldRect, 419, 19, 581, 91);

    if( left == right && top == bottom ) {
        right = left + oldRect.right - oldRect.left;
        bottom = top + oldRect.bottom - oldRect.top;
    }
    SetRect( &newRect, left, top, right, bottom);
    whHLObj = WizardObj_New( hChunk, (WHMEM) 0, left, top,
        right, bottom, dllName, index, whData );
    wizPen.lopnStyle     = 0;
    wizPen.lopnWidth.x   = 1;
    wizPen.lopnWidth.y   = 1;
```

Continued

```
    wizPen.lopnColor     = RGB( 0x00, 0x00, 0x00 );
    WWKit_SetPen( &wizPen );
    wizBrush.lbStyle = 0;
    wizBrush.lbColor = RGB( 0xff, 0xff, 0xff );
    wizBrush.lbHatch = 4;
    WWKit_SetBrush( &wizBrush );

    SetRect( &tmpRect1, 420, 20, 580, 90);
    Rect_Scale( &tmpRect1, &oldRect, &newRect, 0 );

    whObj = RRectangleObj_New( hChunk, whHLObj,
        tmpRect1.left, tmpRect1.top, tmpRect1.right,
        tmpRect1.bottom, 20, 20 );

    lstrcpy( parseBuf, "Discrete" );

    BlinkLnk_New( hChunk, whObj, parseBuf, FALSE,
        RGB( 0x00, 0x00, 0x00 ), RGB( 0x00, 0x80, 0x40 ),
        RGB( 0x00, 0x00, 0x00 ), BLINK_MEDIUM );

    *pwhWizard = whHLObj;
    return 0;
}
```

For more information on the functions used in the code sample, see the "Wizard Toolkit Functions" section of Chapter 3.

Globals

Declarations for the following globals are provided in a special header file called WIZSTUB.H. They are used by the generated routines and provide useful structures and variables when building non-generated Wizards. The initialization for these global variables is often done in WZMAIN.C or in a extra add-on source file named WZSTUB.C. For our simple Wizard, the globals would be as follows:

```
#include <windows.h>
#include "wizbase.h"

// Globals for wizard dialogs and access to resources

    HANDLE    hDrawInst   = (HANDLE)NULL;
    HWND      hDrawWnd    = (HWND)NULL;

// General purpose globals

    WHMEM     whNull      = (WHMEM) 0;
// Globals for object creation/manipulation

    HANDLE      hBitmap;
    char        tmpBuf[132];
    POINT       tmpPt1;
    POINT       tmpPt2;
    POINT       tmpPts[32];
    RECT        tmpRect1;
    RECT        tmpRect2;

    LOGFONT     wizFont     = { 10, 0, 0, 0, FW_NORMAL,0, 0,
0,
        ANSI_CHARSET, OUT_STROKE_PRECIS, CLIP_STROKE_PRECIS,
        DRAFT_QUALITY, DEFAULT_PITCH|FF_SWISS, "Arial" };

    LOGBRUSH    wizBrush    = { BS_SOLID, RGB(0xff,
0xff,0xff),
        HS_CROSS };

    LOGPEN      wizPen      = { PS_SOLID, { 1, 1 }, 0L };

    LOGBRUSH    wizTextBrush= { BS_SOLID, RGB( 0xff, 0xff,
        0xff ), HS_CROSS };

    LOGPEN      wizTextPen  = { PS_SOLID, { 1, 1 }, 0L };

// Globals for link creation/manipulation

    int     errorCol;
    char    parseBuf[ STMT_STRLEN ];
    EXPR    stmtDown;
    EXPR    stmtUp;
    EXPR    stmtWhile;
    LONG    tmpColors[5];
    REAL    tmpValues[4];
```

Integrating a Wizard into WindowMaker

Once we have created our simple Wizard and generated the WIZARD.C file, we are ready to integrate it into a full Wizard DLL. Before we can do this we need to supply:

- A Bitmap for the Wizard Selection dialog box
- Bitmaps for the WindowMaker toolbar
- A group description
- Information regarding sizing of the Wizard
- Information regarding what environment our Wizard operates in

WindowMaker will get this information by calling the DLL's **Wizard_GetInfo** function several times. (If there is nothing to return, **Wizard_GetInfo** can return zero and the defaults will be assumed.)

The function prototype for **Wizard_GetInfo** is the following:

```
BOOL WINAPI Wizard_GetInfo(
    int      index,
    WORD     wCommand,
    DWORD    dwData,
    LPVOID   lpInfo)
```

For more information on this function, see the "User Supplied Wizard Functions" section of Chapter 4.

The following briefly describes each parameter:

Parameter	Description
index	Refers to the index number of a particular Wizard in a library. For our simple Wizard this will be 1.
wCommand	Specifies which property WindowMaker is requesting information about. This is the Information Command described in "The Components of a Wizard DLL" section.
dwData	This DWORD may contain additional data that is required for the information request.
lpInfo	Pointer to the returned data. **Wizard_GetInfo** provides the information back to WindowMaker by attaching this pointer to it.

The *wCommand* parameter specifies which property is being requested. Through this one function, WindowMaker will request bitmap handles, description strings, environment information, sizing information and so on.

The values requested in *wCommand* and the required actions are:

Command	Required Action
WIZ_DESCRIPTION	Returns either the Long Description or the Short Comment description string, based on the value of the *dwData* parameter which was passed in.
WIZ_BITMAP	Returns the main Wizard Selection Dialog bitmap.
WIZ_TBOXBITMAP	Returns either the Toolbox Button Down or the Toolbox Button Up bitmap, based on the value of the *dwData* parameter which was passed in.
WIZ_GROUPNAME	Returns the string name of the Wizard Selection Dialog group where this Wizards will appear.
WIZ_FLAGS	Returns a flag indicating what type of Wizards this is.
WIZ_SIZEMODE	Returns a flag indicating what sizing restrictions there are (if any) on this Wizard.

For more information on these commands, see the "Functions Required to Integrate Wizards into InTouch" section of Chapter 4.

To supply bitmaps for the Wizard, there are three bitmaps needed:

- 64x64 bitmap for the Wizard Selection Dialog
- 16x16 bitmap for the WindowMaker toolbar when the button is up
- 16x16 bitmap for WindowMaker toolbar when the button is pressed

Note The pair of bitmaps used in the WindowMaker toolbar must be identical, except the fill for the button up should be "buttonface" gray and the fill for the button down (pressed) should be white. (WindowMaker will automatically create the 3-D border.) The bitmap you provide should also **not** be shifted over and down. WindowMaker will handle the "pressed down" effect automatically when the button is depressed.

Wizard_GetInfo Example

For our simple Wizard, the **Wizard_GetInfo** function would be:

```
/*
**    Wizard_GetInfo:
**
**        Handles calls to get information about a Wizard.
**
**    Inputs:
**        index       - used only if we are
**                      implementing a wizard library in which
**                      case this is our internal wizard id.
**        wCommand    - WORD specifying information command
**        dwData      - DWORD passing data to needed to get
**                        info
**
**    Outputs:
**        lpInfo      - points to the returned information
**                      buffer
**    Returns:
**        TRUE if the command is supported by the DLL.
**        FALSE also indicates to do default handling.
*/
BOOL
WINAPI
Wizard_GetInfo(
    int       index,
    WORD      wCommand,
    DWORD     dwData,
    LPVOID    lpInfo )
{
    char      name[128];
    BOOL      bResult = TRUE;

    switch( wCommand ) {
    case WIZ_DESCRIPTION:
        if( (BOOL)dwData ) {
            LoadString( hDrawInst, index * 2 - 1, lpInfo,
                MAX_STRING_LEN );

        } else {
            LoadString( hDrawInst, index * 2, lpInfo,
                MAX_STRING_LEN );
        }

        break;
    case WIZ_BITMAP:
        wsprintf( name, "RRECT%.2dMBMP", index );
        *(HBITMAP FAR*)lpInfo = LoadBitmap( hDrawInst,
            (LPSTR)name );
        break;
    case WIZ_TBOXBITMAP:
        if( ((LP_WIZTBOXBITMAPINFO)dwData)->bPushed ) {
            wsprintf( name, "RRECT%.2dPBMP", index );
        } else {
            wsprintf( name, "RRECT%.2dTBMP", index );
        }
        *(HBITMAP FAR*)lpInfo = LoadBitmap( hDrawInst,
            (LPSTR)name );
        break;
    case WIZ_GROUPNAME:
        lstrcpy( lpInfo, "Rounded Rectangles" );
        break;
```

continued

```
    case WIZ_HELPINFO:
        bResult = FALSE;
        break;
    case WIZ_FLAGS:
        *(LPDWORD)lpInfo = 0;   // No special flags
        break;
    case WIZ_SIZEMODE:
        *(LPDWORD)lpInfo = WIZSIZE_FULL;
        break;
    default:
        bResult = FALSE;
        break;
    }
    return( bResult );
}
```

WizardLib_GetInfo Example

Since Wizards reside in libraries, there is some additional information that needs to be supplied. WindowMaker will retrieve this information by calling the **WizardLib_GetInfo** function. **WizardLib_GetInfo** is similar to **Wizard_GetInfo**, in that it is passed commands for which specific information needs to be returned. These items relate to the Wizard Library in general.

The function prototype is as follows:

```
BOOL
WINAPI
WizardLib_GetInfo(
    WORD      wCommand,
    DWORD     dwData,
    LPVOID    lpInfo)
```

The following briefly describes each parameter:

Parameter	Description
wCommand	Specifies which property it is requesting information about. This is an Information Command described in "The Components of a Wizard DLL" section.
dwData	This DWORD may contain additional data that is required for the information request.
lpInfo	Pointer to the returned data. **WizardLib_GetInfo** provides the information back to WindowMaker by attaching this pointer to it.

The values requested in *wCommand* and the required actions are:

Command	Required Action
WIZ_COMPANYNAME	Returns company description string.
WIZ_LIBNAME	Returns library descriptive name.
WIZ_NEXTWIZID	Returns next Wizard ID. WindowMaker will make multiple calls for this command, building for itself a list of the Wizard IDs contained in particular Wizard DLL
WIZ_VERSIONNUM	Returns library version number.
WIZ_VERSIONSTR	Returns library version string.

For our simple Wizard, the **WizardLib_GetInfo** would be:

```
/*
**    WizardLib_GetInfo:
**
**        Handles calls to get information about this
**        wizard library.
**    Inputs:
**        wCommand    - WORD specifying information
**                      command
**        dwData      - DWORD passing data to needed
**                      to get info
**    Outputs:
**        lpInfo      - points to the returned
**                      information buffer
**    Returns:
**        TRUE if the command is supported by the DLL.
**        FALSE also indicates to do default handling.
*/
BOOL
WINAPI
WizardLib_GetInfo(
    WORD      wCommand,
    DWORD     dwData,
    LPVOID    lpInfo )
{
    BOOL      bResult = TRUE;
switch( wCommand ) {
    case WIZ_COMPANYNAME:
        lstrcpy( lpInfo, "Wonderware Corporation" );
        break;
    case WIZ_VERSIONNUM:
        *(LPDWORD)lpInfo = 0;
        break;
    case WIZ_VERSIONSTR:
        lstrcpy( lpInfo, "Version 5.0 " );
        break;
    case WIZ_LIBNAME:
        lstrcpy( lpInfo, "Wonderware sample rounded rectangle
            Wizards" );
        break;
    case WIZ_NEXTWIZID: // how many wizards in library
        if( dwData == 0 ) {
            *(LPDWORD)lpInfo = 1;    // start of wizards
        }
        else if( dwData >= 2 ) {
            *(LPDWORD)lpInfo = 0;    // indicates last wizard
        }
        else {
            *(LPDWORD)lpInfo = dwData + 1;
        }
        break;
    default:
        bResult = FALSE;
        break;
    }
    return( bResult );
}
```

Simple Wizard .RC File Example

Now that we've provided a mechanism for WindowMaker to load bitmaps, we must provide a link to the actual BMP files. It is the resource file (.RC file) that contains references to the bitmaps that we'll use. It also contains strings and dialog boxes that we'll use later. For the simple Wizard, the resource file simply contains references to our bitmaps. For example:

```
#include "windows.h"

RRECT01TBMP     BITMAP  MOVEABLE PURE   "RRECT01T.BMP"
RRECT01PBMP     BITMAP  MOVEABLE PURE   "RRECT01P.BMP"
RRECT01MBMP     BITMAP  DISCARDABLE     "RRECT01M.BMP"
```

Building the Wizard DLL

We are now ready to build the Wizard DLL that WindowMaker calls whenever the Wizard is used. To build the Wizard DLL, start with the sample project makefile that is supplied with the Wizard Toolkit. (This makefile was created using Visual C++ and is designed for building DLLs.):

- Select the **Edit** command on the **Project** menu to edit the project file to include the source files that we have created while building our sample Wizard
- Regenerate dependencies (happens automatically)
- Select the **Build** command or the **Rebuild** command on the **Project** menu to compile and link the new DLL

Installing the Wizard in WindowMaker

Once the Wizard DLL is built, you can install the Wizard in **WindowMaker** by performing the following steps:

1. Copy the Wizard DLL into your InTouch directory.
2. Change the filename extension from .DLL to .WZU. (The .WZU extension indicates to WindowMaker that this is an Uninstalled Wizard.)
3. On the WindowMaker **Special** menu, select **Configure** and then select **Wizard/ActiveX Installation**. The **Wizard Installation** dialog box will appear.
4. Click **Install** to install the Wizard DLL in WindowMaker. The Wizard is now available from the Wizard Selection Tool  in the WindowMaker toolbar.

Note WindowMaker will automatically change the .WZU extension to .DLL. If you copy the Wizard DLL file directly into your InTouch application directory with the extension .DLL, **it will not work**! It must be installed using the method described here in order to work properly! If you change the name, order, or number of functions/wizards in the library, then you must use the **Wizard Installation** dialog box to remove the old wizard, and then add the new one. Otherwise, a GPF will occur.

Wizard Libraries

Now that we have developed our first Wizard and used it in WindowMaker, the basics of Wizard Development is now hopefully a lot clearer: Wizard developers create Wizard DLLs; Wizard DLLs provide functions that WindowMaker can call; WindowMaker calls these functions when InTouch developers want to use the Wizard.

Having only one Wizard in our DLL is pretty limiting. That would mean that for every new Wizard that we want to develop, it would be necessary to write all the code that we wrote in the previous section over and over again, **Wizard_New**, **Wizard_GetInfo**, and so on. But that is not necessary. Wizard DLLs can contain multiple Wizards which can share these functions. Our Wizard DLL is then referred to as a Wizard Library, and Wizards are treated as individual entries in these libraries. In this next section, we'll examine the issues surrounding Wizard Libraries.

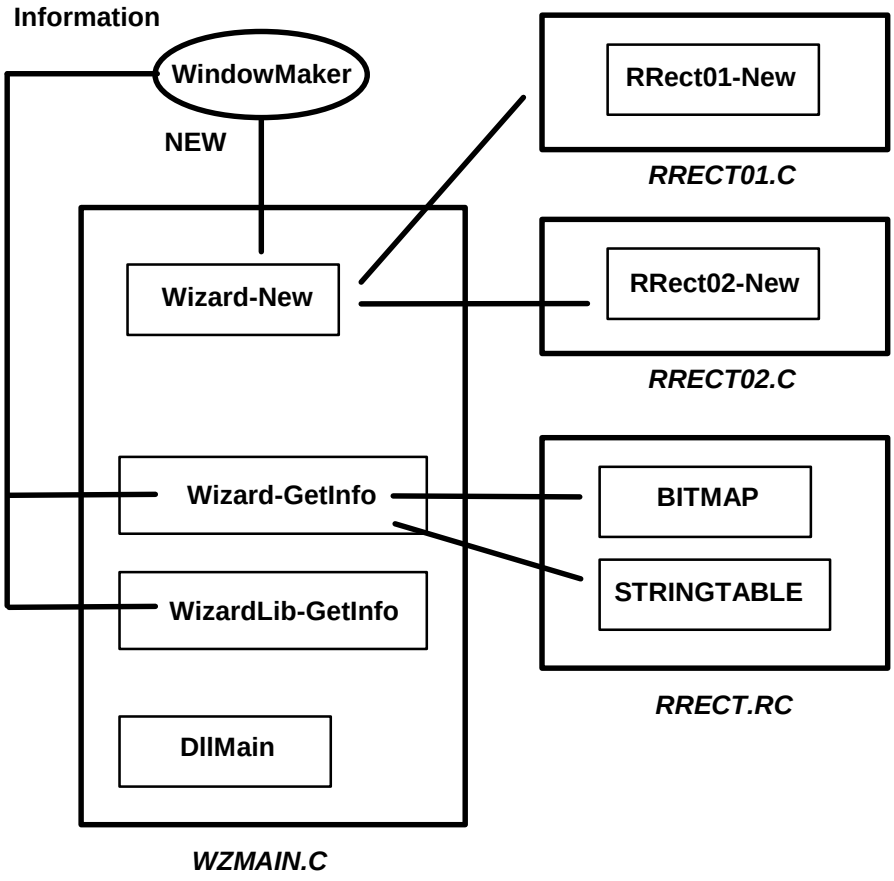
Note The examples from now on, including those in the next section involving building configurable wizards are based upon Wizard Libraries. Almost all Wizards are contained in libraries, and so it is very important to understand the concepts introduced in this section

---

Normally, a Wizard DLL will contain multiple wizards that are related in some way. Building a Wizard Library like this involves a little planning, particularly in the area of naming conventions, which is discussed here as well. Building a Wizard Library correctly from the start will make it very easy to expand later.

Creating Libraries with Multiple Wizards

To support multiple Wizards, a typical Wizard DLL is structured as follows:



When an InTouch application developer selects a Wizard from the toolbar and places it in a window, WindowMaker considers this Wizard to be in the NEW mode. To handle the processing of a NEW mode Wizard, WindowMaker calls **Wizard_New** in the Wizard DLL. This is the same action that occurs with a Wizard DLL that contains only one Wizard.

When the DLL contains multiple Wizards, **Wizard_New** will dispatch other routines, **RRect01New** or **RRect02New** (depending upon the particular Wizard), to handle the actual drawing and resizing. The *index* parameter passed to **Wizard_New** specifies which Wizard in the library is about to be created, and which dispatch function is to be called. Upon completion, these dispatched functions pass control back to **Wizard_New** which gives control back to WindowMaker.

The body of the code (that used to be in **Wizard_New**) is now **RRect01New**. **RRect01New** is called from **Wizard_New** when a Wizard with index number one is about to be created. For example:

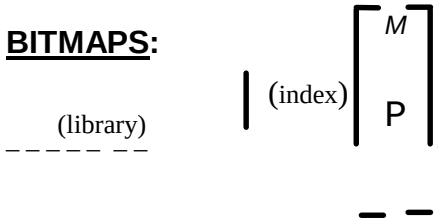
```
int FAR PASCAL Wizard_New(
    HCHUNK    hChunk,
    int       index,
    int       left,
    int       top,
    int       right,
    int       bottom,
    LPSTR     dllName,
    WHMEM     whData,
    int       mode,
    RECT      prevRect,
    WHMEM     FAR *pwhWizard)
{
    int       error;
    error = 0;
    switch( index ) {

    case 1:
        error = RRect01New( hChunk, index, left, top, right,
            bottom, dllName, whData,  mode, prevRect,
            pwhWizard );
        break;

    case 2:
        error = RRect02New( hChunk, index, left, top, right,
            bottom, dllName, whData,  mode, prevRect,
            pwhWizard );
        break;

    default:
        *pwhWizard = whNull;
        error = 1;
        break;
    }
    return( error );
}
```

The remaining changes to a Wizard DLL (to accommodate multiple Wizards) are in the **Wizard_GetInfo** and **WizardLib_GetInfo** functions. The templates that we used in the earlier examples for these routines are already set up to accommodate multiple Wizards in a library. If we follow a few simple naming conventions, these functions become generic and can easily handle Wizard libraries with any number of Wizards.

Naming Conventions

The basis of all the names in your Wizard Library should be a simple mnemonic that categorizes the type of Wizards in the library. For example, METER. The library name is followed by a unique two-digit number to identify the individual Wizard in the library. For example, name all bitmap files using the following convention:

BITMAPS:



METER01M.BMP
METER01T.BMP
METER01P.BMP

The "M" extension is for the (M)ain Wizard Selection dialog box bitmap, the "T" extension is for the (T)oolbar Button Down bitmap, and the "P" extension is for the Toolbar Button (P)ushed-In bitmap.

Name each bitmap resource (in the project's RC file) the same as the filename, without the period (.). For example, the bitmap resources for the bitmaps in the example would be listed as follows:

For example:

```
METER01MBMP      BITMAP        MOVEABLE PURE
    "METER01M.BMP"
METER01TBMP      BITMAP        MOVEABLE PURE
    "METER01T.BMP"
METER01PBMP      BITMAP        MOVEABLE PURE
    "METER01P.BMP"
```

We recommend that all Wizard descriptions reside in the STRINGTABLE for the Wizard library. The string table identifier for each Wizard is generated using the following formula:

Long Description `(index) * 2 - 1`

Short Description `(index) * 2`

Using this formula, a description string table can be constructed for a library of three meter Wizards:

```
STRINGTABLE
BEGIN
1,    "Meter 1 Long Description"
2,    "Meter 1 Short Description"
3,    "Meter 2 Long Description"
4,    "Meter 2 Short Description"
5,    "Meter 3 Long Description"
6,    "Meter 3 Short Description"
END
```

Building a Configurable Wizard

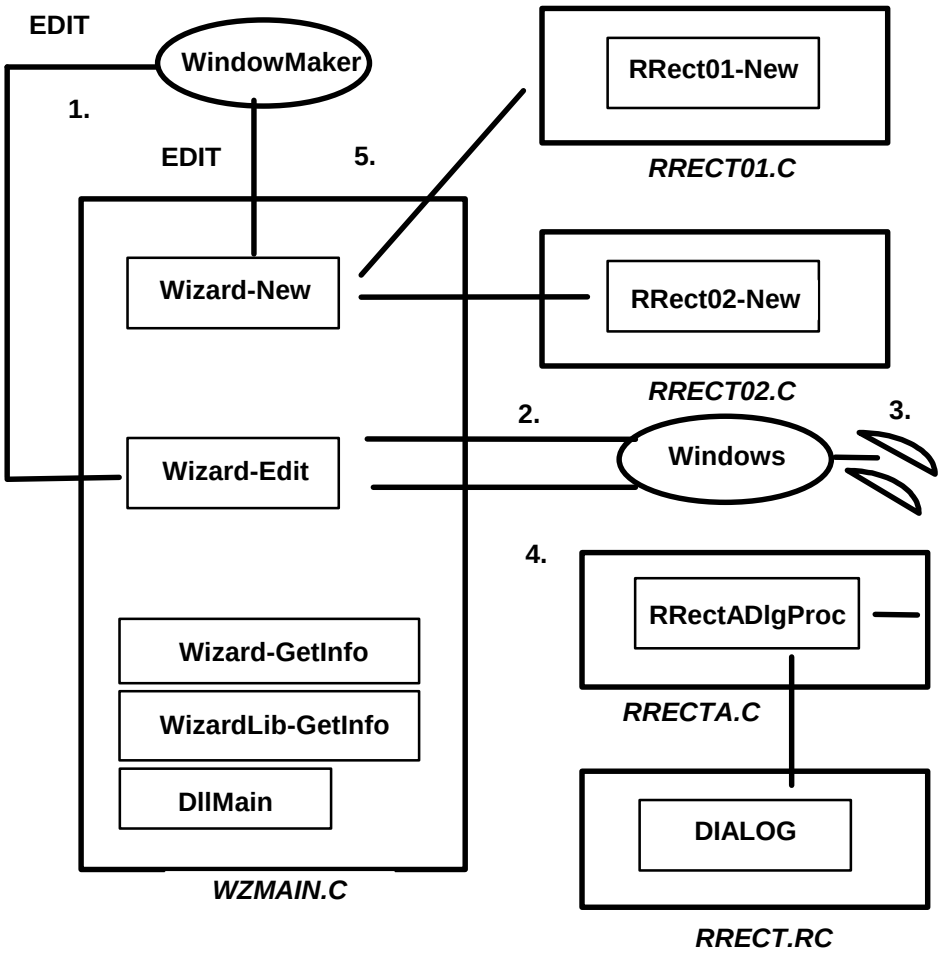
The real power of Wizards lies in the ability of the InTouch application developer to change the characteristics of the Wizard to suit the particular application being built. In our previous examples, we created a Wizard that was sizable with predetermined animation links. In most cases, a Wizard will have certain characteristics (or properties), such as fonts, location links, size links, color links, and so on, that the application developer can choose when using the Wizard. The Wizard developer provides this configurability when writing the Wizard.

There is no pre-determined set of properties that the Wizard developer must use. A Meter Wizard may allow the application developer to configure the number of tick marks, the range, and the label color. A Command push button Wizard may allow the application developer to configure the text on the button and the script associated with the button. All of the animation links that are available in the WindowMaker links dialog box can be exposed as configurable properties to the Wizard user. It is up to the Wizard developer to determine the properties of the InTouch object that should be configured. If desired, certain properties can even be selected from the WindowMaker toolbar while drawing the object.

When an InTouch application developer double-clicks on a Wizard for configuration, the Wizard DLL that originally drew the Wizard will display a dialog box. This dialog box allows the InTouch application developer to configure the Wizard. The Wizard DLL then takes the application developer's dialog box entries and stores them with the Wizard as "Wizard Properties."

A property is defined and set by the Wizard developer via the WizProp_Set* functions. For example, if we allow the InTouch application developer to configure the expression associated with the blink link for the simple Wizard we previously created, we would need to define a property, give it a name, and store the application developer's choice as the value of this property. All of this is called by WindowMaker and happens in a special routine called **Wizard_Edit**. Every time the application developer double-clicks on the Wizard, WindowMaker will call **Wizard_Edit**.

To support the configuration, a typical Wizard DLL is structured as follows:



1. When WindowMaker gets a signal from an application developer to configure a Wizard (double-clicks on the Wizard), WindowMaker will call **Wizard_Edit**. The Wizard developer needs to provide the **Wizard_Edit** and place it in the WZMAIN.C or DLLMAIN.C file.

2. **Wizard_Edit** then calls the **DialogBox()** function to create a dialog box and call a dialog procedure, **RRectADlgProc**. This will effectively "hand over" control to Windows, which will call **RRectADlgProc**.

3. The dialog box and the dialog procedure, **RRectADlgProc** will allow the application developer to modify and save the properties of the Wizard. Windows passes messages to the **RRectADlgProc** as the application developer interacts with the dialog box.

4. Once the application developer clicks **OK** in the dialog box, **RRectADlgProc** will save the properties and return control to Windows. Windows then returns to **Wizard_Edit**. **Wizard_Edit** then returns control to WindowMaker. The Wizard developer needs to provide the toolbar and **RRectADlgProc**. The toolbar resource is placed in the resource (RC) file. **RRectADlgProc** is normally placed in a file by itself, in this case RRECTA.C

5. After the EDIT mode call to **Wizard_Edit** is completed, WindowMaker will automatically make an EDIT mode call to **Wizard_New** so that the DLL can take advantage of the modified properties to redraw the Wizard. In this case, **Wizard_New** will dispatch to either **RRect01New** or **RRect02New** to retrieve the new properties (saved by **RRectADlgProc**) and redraw.

Wizard_Edit Example

The Wizard_Edit routine is similar to the **Wizard_New** routine in that it uses the *index* parameter to decide which Wizard to edit. **Wizard_Edit** creates a dialog box and calls a dialog procedure. The dialog procedure allows the application developer to modify and save the properties of the Wizard. The following **Wizard_Edit** routine will create a dialog box from the resource "RRECTADLG" and call a dialog procedure, **RRectADlgProc**. Notice that the dialog procedure and dialog box do not have to be associated with only one Wizard. It's possible that several Wizards in the DLL will be similar enough that their properties can be configured with same dialog box. For example:

```
/*
**    Wizard_Edit:
**
**        Brings up dialog box to edit the Wizard's
**        configuration.
**        Modify/edit Wizard properties as appropriate and
**              save
**        if user selects OK.
**        Abandon changes if user selects Cancel.
**
**    Process:
**        1.  Initialize globals needed to remember Wizard
**              being edited.
**        2.  Bring up the dialog.
**
**    Inputs:
**        whObj     - wizard's object handle
**        hChunk    - memory allocation handle
**
**    Outputs:
**        None
**
**    Returns:
**        error code or 0 if no error.
*/
```

continued

```
int

WINAPI
Wizard_Edit (
    int       index,
    WHMEM     whObj,
    HCHUNK    hChunk)
{
    editObj   = whObj;
    editChunk = hChunk;
    editIndex = index;
    switch (index) {

    case 1:
    case 2:
        DialogBox (hDrawInst, "RRECTADLG", hDrawWnd,
            RRectADlgProc );
        break;

    default:
        break;
    }

    return FALSE;
}
```

Property Names

In the next example, we have decided to allow the application developer to configure the blink expression associated with the rounded rectangle object. However, before we can do that, we have to discuss one aspect of adding configurability to Wizards that has not been mentioned so far. That is property *names*. When giving a Wizard a characteristic or property that the application developer can configure, the Wizard developer also must determine a name for this property. This property name is a text string that is used to associate a given property with the Wizard. This property *na*me will then be used as an *ID* or *label* for retrieving and storing the *value* of the property. A table containing each property name and its current value will also be stored with the Wizard.

The name of our property will be "BlinkExpression." It will be specified in our code by the constant PROP_RRECTA_BLINKEXPR, which is #define'd in one of our header files.

Dialog Proc Example

In our example the property will be an InTouch expression, therefore we will use the **WizProp_SetExpr** and **WizProp_GetExpr** functions to update and read the value of our property. Making the property an expression (versus a String) allows us to use the **Substitute Tags** command on the WindowMaker **Special** menu on our Wizard. The dialog procedure is as follows:

```
/*
**    RRectADlgProc:
**        Handles our wizard's dialog.
**    Inputs:
**        Std Window's dialog procedure inputs.
**    Outputs:
**        None
**    Returns:
**        Std Window's dialog procedure return.
*/
// Globals for editing purposes used by our dialog.

extern WHMEM    editObj;
extern HCHUNK   editChunk;
extern int      editIndex;

BOOL
WINAPI
RRectADlgProc (
    HWND      hDlg,
    int       message,
    WPARAM    wParam,
    LPARAM    lParam)
{
    RECT      rect;
    char      text[NL_EXPRESSION];

    switch (message) {
    case WM_INITDIALOG:
        /* Center the dialog */
        GetWindowRect (hDlg, &rect);
        MoveWindow (hDlg,
            (GetSystemMetrics (SM_CXSCREEN) -
            (rect.right - rect.left)) / 2,
            (GetSystemMetrics (SM_CYSCREEN) -
            (rect.bottom - rect.top)) / 2,
            rect.right - rect.left,
            rect.bottom - rect.top, FALSE);
        /* Limit the number of characters in the blink
         * expression control
         */
        SendMessage (GetDlgItem (hDlg, ID_RRECTA_BLINKEXPR),
            EM_LIMITTEXT, NL_EXPRESSION - 1, 0);
        /* Get properties and initialize dialog fields  */
```

continued

```
            WizProp_GetExpr (editChunk, editObj,
                PROP_RRECTA_BLINKEXPR, NL_EXPRESSION, text,
                defBlinkExprName);
            SetDlgItemText (hDlg, ID_RRECT_BLINKEXPR, text);

            WWDlg_RegisterTagNameCtrl( hDlg,
              ID_RRECTA_BLINKEXPR );
            SendMessage( GetDlgItem( hDlg, ID_RRECTA_BLINKEXPR ),
                EM_SETSEL, 0, MAKELONG(0,32767) );
            SetFocus( GetDlgItem( hDlg, ID_RRECTA_BLINKEXPR ) );

            return FALSE;

        case WM_COMMAND:
            switch (LOWORD (wParam)) {
            case IDCANCEL:
                WWDlg_UnregisterTagNameCtrl( hDlg,
                    ID_RRECTA_BLINKEXPR );
                EndDialog (hDlg, 0);
                return (TRUE);

            case IDOK:
                if( WWDlg_CheckExprCtrl( hDlg,
                   ID_RRECTA_BLINKEXPR,
                    TYPE_DISCRETE ) ) {
                    return TRUE;

                } else {
                    WWDlg_UnregisterTagNameCtrl( hDlg,
                        ID_RRECTA_BLINKEXPR );
                    WWDlg_UnregisterColorCtrl( hDlg,
                        ID_RRECTA_ONCOLOR, &dwData );
                    WizProp_SetDWord( editChunk, editObj,
                        PROP_RRECTA_ONCOLOR, dwData );
                    dwData = DEF_OFF_COLOR;
                    GetDlgItemText( hDlg, ID_RRECTA_BLINKEXPR,
                      text, NL_TAGNAME );
                    text[NL_TAGNAME-1] = '\0';
                    WizProp_SetString( editChunk, editObj,
                       PROP_RRECTA_BLINKEXPR, text );
                }

                EndDialog( hDlg, 0 );
                return( TRUE );

            default:
                return FALSE;
            }
            return TRUE;

        default:
            return FALSE;
        }
        return (TRUE);
    }
```

In the WM_INITDIALOG message processing, the function **WizProp_GetExpr** retrieves either the current value of the property named PROP_RRECTA_BLINKEXPR or a default value *defBlinkExprName*, if the property does not currently exist (has not been set yet). In this example, the property name PROP_RRECTA_BLINKEXPR and default value *defBlinkExprName* have been defined as follows:

```
#define PROP_RRECTA_BLINKEXPR  rrectBlinkExprStr
LPSTR rrectBlinkExprStr = "BlinkExpression";

LPSTR defBlinkExprName  = "?d:Discrete";
```

When **OK** is clicked, the contents of the edit control for the blink expression is read and stored in the Wizard as the property named PROP_RRECTA_BLINKEXPR, via the **WizProp_SetExpr** function. Once the **WizProp_SetExpr** function has been executed, our Wizard will have a property of name PROP_RRECTA_BLINKEXPR and a value as specified by the user.

Now that we have built a **Wizard_Edit** routine that calls a dialog procedure and the dialog procedure sets the property, where do we go from here? After **Wizard_Edit** returns, **Wizard_New** will be called by WindowMaker to create an object with the new value of any properties modified by **Wizard_Edit**.

Wizard_New calls **RRect01New** to draw the object and configure any links. The **RRect01New** routine needs to set up the blink link from the property (as opposed to the predefined code that was generated from the **Generate Wizard** command). In the original routine, the parameter for the blink expression on the **BlinkLnk_New** function was hard coded to be the expression, Discrete:

```
lstrcpy( parseBuf, "Discrete" );
```

We will make this configurable, by adding the function **WizProp_GetExpr** before the **WizardObj_New** function call to get the current value of the property named PROP_RRECTA_BLINKEXPR. The string returned by that call, is returned into the buffer *blinkExpr*:

```
WizProp_GetExpr ( hChunk, whData, PROP_RRECTA_BLINKEXPR,
        NL_TAGNAME, blinkExpr, defBlinkExprName );
```

That buffer is used as the parameter to **BlinkLnk_New**, which sets the expression to use for the blink link:

```
wsprintf( parseBuf, "%s", (LPSTR)blinkExpr );
    BlinkLnk_New( hChunk, whObj, parseBuf, FALSE,
        RGB( 0x00, 0x00, 0x00 ), RGB( 0x00, 0x80, 0x40 ),
        RGB( 0x00, 0x00, 0x00 ), BLINK_MEDIUM );
```

The first time **Wizard_New** is called, there is no value for the property named PROP_RRECTA_BLINKEXPR therefore, the default value (specified by *defBlinkExprName*) is used (?d:Discrete). Once the InTouch application developer double-clicks the Wizard and specifies a "real" blink expression, the property value is set and **Wizard_Edit** will cause **Wizard_New** to be called again. This time, the **WizProp_GetExpr** will return the value specified by the InTouch application developer.

The modified **RRect01New** function would become:

```
int
WINAPI
RRect01New (
    HCHUNK    hChunk,
    int       index,
    int       left,
    int       top,
    int       right,
    int       bottom,
    LPSTR     dllName,
    WHMEM     whData,
    int       mode,
    RECT      prevRect,
    WHMEM     FAR * pwhWizard)
{
    WHMEM     whHLObj;
    WHMEM     whObj;
    RECT      oldRect;
    RECT      newRect;
    char      blinkExpr[NL_EXPRESSION];

SetRect (&oldRect, 419, 19, 581, 91);
    if (left == right && top == bottom) {
        right = left + oldRect.right - oldRect.left;
        bottom = top + oldRect.bottom - oldRect.top;
    }

SetRect (&newRect, left, top, right, bottom);

    /* Initialize any properties  */
    WizProp_GetExpr (hChunk, whData,
        PROP_RRECTA_BLINKEXPR, NL_EXPRESSION, blinkExpr,
        defBlinkExprName);

    whHLObj = WizardObj_New (hChunk, (WHMEM) 0,
        left, top, right, bottom, dllName, index, whData);

    wizPen.lopnStyle   = 0;
    wizPen.lopnWidth.x = 1;
    wizPen.lopnWidth.y = 1;
    wizPen.lopnColor   = RGB (0x00, 0x00, 0x00);
    WWKit_SetPen (&wizPen);

    wizBrush.lbStyle   = 0;
    wizBrush.lbColor   = RGB (0xff, 0xff, 0xff);
    wizBrush.lbHatch   = 4;
    WWKit_SetBrush (&wizBrush);

    SetRect (&tmpRect1, 420, 20, 580, 90);
    Rect_Scale (&tmpRect1, &oldRect, &newRect, 0);
    whObj = RRectangleObj_New (hChunk, whHLObj,
        tmpRect1.left, tmpRect1.top,
        tmpRect1.right, tmpRect1.bottom, 20, 20);
```

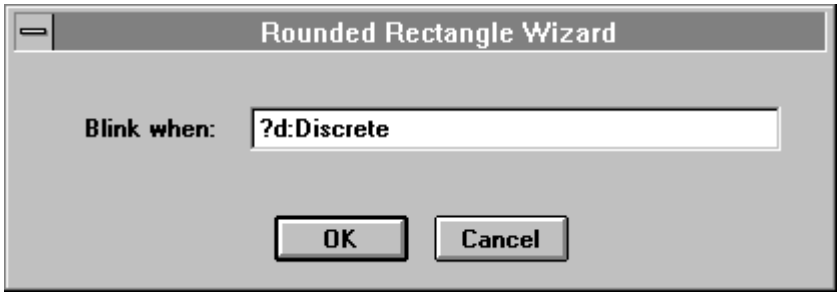
continued

```
    wsprintf (parseBuf, "%s", (LPSTR) blinkExpr);

    BlinkLnk_New (hChunk, whObj, parseBuf, FALSE,
        RGB (0x00, 0x00, 0x00), RGB ( 0x00, 0x80, 0x40),
        RGB (0x00, 0x00, 0x00), BLINK_MEDIUM);

    *pwhWizard = whHLObj;
    return 0;
}
```

Before we are finished, one more change needs to get made. The .DEF file needs to be changed to add the exports for **Wizard_Edit** and **RRectADlgProc**. If these exports are not added, those routines will not be visible outside the DLL and problems will occur when WindowMaker wants to call your Wizard DLL:

```
LIBRARY wzsample
DESCRIPTION 'WIZARD Toolkit Sample DLL'

''EXPORTS
    Wizard_New
    Wizard_GetInfo
    WizardLib_GetInfo
    Wizard_Edit
    RRectADlgProc
```

Now that we have completed these steps, when the InTouch application developer pastes our simple Wizard into a window and then, double-clicks it, the following dialog box will appear:



The InTouch application developer can now enter a valid expression that will be evaluated to control the blink link.

Special Wizard Dialog Controls

The Wizard API provides several functions that expose some of the dialog controls in WindowMaker. For example, if the Wizard developer wanted to have a Wizard property that allows the InTouch application developer to configure the color of the blink link, he could make the InTouch application developer type in the RGB color in the edit control (not very friendly) or use the **WWDlg_RegisterColorCtl** and **WWDlg_UnregisterColorCtl** functions. These functions register a dialog item to use the standard InTouch color choice mechanism.

Let's assume that we wanted to allow the InTouch application developer to define the color that will be used when our simple Wizard is blinking "on." To accomplish this, we will have to first modify the dialog box to allow for the color change item to be specified and then modify the dialog procedure. We will also have to modify the **RRect01New** function to use another property named PROP_RRECTA_ONCOLOR. First, the Wizard configuration dialog box should be modified to allow for the color control. This must be a list box control. For example:

```
RRECTADLG DIALOG 18, 18, 232, 88
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION
CAPTION "Rounded Rectangle Wizard"
FONT 8, "MS Sans Serif"
BEGIN
    LTEXT "Blink when:", -1, 19, 19, 42, 11, WS_CHILD |
        WS_VISIBLE | WS_GROUP
    CONTROL "", ID_RRECT_BLINKEXPR, "EDIT", ES_LEFT |
        ES_AUTOHSCROLL | WS_CHILD | WS_VISIBLE | WS_BORDER |
        WS_TABSTOP, 68, 17, 150, 12
    LTEXT "Fill Color On:", -1, 19, 48, 47, 10, WS_CHILD |
        WS_VISIBLE | WS_GROUP
    CONTROL "", ID_RRECTA_ONCOLOR, "listbox", LBS_NOTIFY |
        WS_BORDER | WS_CHILD | LBS_NOINTEGRALHEIGHT, 70, 48,
          20, 10
    DEFPUSHBUTTON "OK", IDOK, 74, 70, 38, 14, WS_CHILD |
        WS_VISIBLE | WS_TABSTOP
    PUSHBUTTON "Cancel", IDCANCEL, 120, 70, 38, 14, WS_CHILD |
        WS_VISIBLE | WS_TABSTOP
END
```

The dialog procedure is also modified to use the color control. The **WWDlg_RegisterColorCtrl** function is called during WM_INITDIALOG processing. The **WWDlg_RegisterColorCtrl** uses a list box control. When the application developer clicks on the list box the standard InTouch color selection palette appears. Calling **WWDlg_UnregisterColorCtrl** returns the selected color value. The **RRectADlgProc** would be as follows:

```
BOOL
WINAPI
RRectADlgProc (
    HWND      hDlg,
    int       message,
    WPARAM    wParam,
    LPARAM    lParam)
{
    RECT      rect;
    DWORD     dwData;
    char      text[NL_EXPRESSION];

    switch (message) {
    case WM_INITDIALOG:
        /* Center the dialog */
        GetWindowRect (hDlg, &rect);
        MoveWindow (hDlg,
            (GetSystemMetrics (SM_CXSCREEN) -
            (rect.right - rect.left)) / 2,
            (GetSystemMetrics (SM_CYSCREEN) -
            (rect.bottom - rect.top)) / 2,
            rect.right - rect.left,
            rect.bottom - rect.top, FALSE);
        /* Limit the number of characters in the blink
         * expression control
         */
        SendMessage (GetDlgItem (hDlg, ID_RRECT_BLINKEXPR),
            EM_LIMITTEXT, NL_EXPRESSION - 1, 0);

        /* Get properties and initialize dialog fields  */
        WizProp_GetDWord (editChunk, editObj,
            PROP_RRECTA_ONCOLOR, &dwData, DEF_ON_COLOR);
        WWDlg_RegisterColorCtrl (hDlg, ID_RRECTA_ONCOLOR,
            dwData);
        WizProp_GetExpr (editChunk, editObj,
            PROP_RRECTA_BLINKEXPR, NL_EXPRESSION, text,
            defBlinkExprName);
        SetDlgItemText (hDlg, ID_RRECTA_BLINKEXPR, text);

        return FALSE;

    case WM_COMMAND:
        switch (LOWORD (wParam)) {
        case IDCANCEL:
            /* Unregister the color controls  */
            WWDlg_UnregisterColorCtrl (hDlg,
               ID_RRECTA_ONCOLOR, &dwData);

            EndDialog (hDlg, 0);
            return (TRUE);
```

continued

```
            case IDOK:
                /* Unregister controls, save properties */
                dwData = DEF_OFF_COLOR;
                WWDlg_UnregisterColorCtrl (hDlg,
                    ID_RRECTA_ONCOLOR, &dwData);
                WizProp_SetDWord (editChunk, editObj,
                    PROP_RRECTA_ONCOLOR, dwData);
                GetDlgItemText (hDlg, ID_RRECT_BLINKEXPR, text,
                    NL_EXPRESSION);
                text[NL_EXPRESSION - 1] = '\0';
                WizProp_SetExpr (editChunk, editObj,
                    PROP_RRECTA_BLINKEXPR, text);
                EndDialog (hDlg, 0);
                return (TRUE);

            default:
                return FALSE;
            }
            return TRUE;

        default:
            return FALSE;
        }
        return (TRUE);
}
```

The new property for the color, PROP_RECTA_ONCOLOR is retrieved during WM_INITDIALOG using **WizProp_GetDWord**, then sent back to the Wizard once the InTouch application developer clicks **OK**, using **WizProp_SetDWord**. (Since the color value is an RGB value, the property type that we use is DWORD.)

The last step is changing the **RRect01New** function to use the new color property. Since the color fill for the blink link is one of the parameters in our **BlinkLink_New** function, the "hard-coded" color value should be replaced by the value saved in our property. The **WizProp_GetDWord** function is used to retrieve the DWORD value in the property named PROP_RECTA_ONCOLOR, which is then stored in the LONG variable *onColor*.

Notice that the fill color parameter has been changed from the hard-coded RGB color to the color in *onColor*:

```
BlinkLnk_New( hChunk, whObj, parseBuf, FALSE,
    RGB( 0x00, 0x00, 0x00 ),  onColor,
    RGB( 0x00, 0x00, 0x00 ), BLINK_MEDIUM );
```

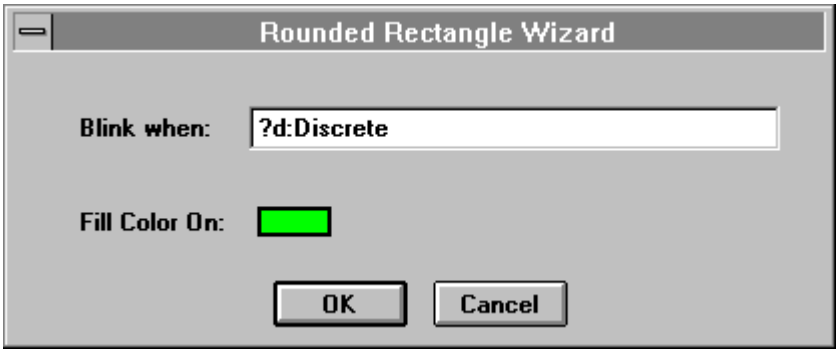
The new **RRect01New** function becomes:

```
int WINAPI
RRect01New (
    HCHUNK    hChunk,
    int       index,
    int       left,
    int       top,
    int       right,
    int       bottom,
    LPSTR     dllName,
    WHMEM     whData,
    int       mode,
    RECT      prevRect,
    WHMEM     FAR * pwhWizard)
{
    WHMEM     whHLObj;
    WHMEM     whObj;
    RECT      oldRect;
    RECT      newRect;
    DWORD     dwData;
    char      blinkExpr[NL_EXPRESSION];
    LONG      onColor;

    SetRect (&oldRect, 419, 19, 581, 91);
    if (left == right && top == bottom) {
        right = left + oldRect.right - oldRect.left;
        bottom = top + oldRect.bottom - oldRect.top;
    }

    SetRect (&newRect, left, top, right, bottom);
    /* Initialize any properties  */

    WizProp_GetDWord (hChunk, whData,
        PROP_RRECTA_ONCOLOR, &dwData, DEF_ON_COLOR);
    onColor = (LONG) dwData;
    WizProp_GetExpr (hChunk, whData,
        PROP_RRECTA_BLINKEXPR, NL_EXPRESSION, blinkExpr,
        defBlinkExprName);

    whHLObj = WizardObj_New (hChunk, (WHMEM) 0,
        left, top, right, bottom, dllName, index, whData);

    wizPen.lopnStyle   = 0;
    wizPen.lopnWidth.x = 1;
    wizPen.lopnWidth.y = 1;
    wizPen.lopnColor   = RGB (0x00, 0x00, 0x00);
    WWKit_SetPen (&wizPen);

    wizBrush.lbStyle   = 0;
    wizBrush.lbColor   = RGB (0xff, 0xff, 0xff);
    wizBrush.lbHatch   = 4;
    WWKit_SetBrush (&wizBrush);

    SetRect (&tmpRect1, 420, 20, 580, 90);
    Rect_Scale (&tmpRect1, &oldRect, &newRect, 0);
```

continued

```
whObj = RRectangleObj_New (hChunk, whHLObj,
        tmpRect1.left, tmpRect1.top,
        tmpRect1.right, tmpRect1.bottom, 20, 20);

    wsprintf (parseBuf, "%s", (LPSTR) blinkExpr);

    BlinkLnk_New (hChunk, whObj, parseBuf, FALSE,
        RGB (0x00, 0x00, 0x00), onColor,
        RGB (0x00, 0x00, 0x00), BLINK_MEDIUM);

    *pwhWizard = whHLObj;
    return 0;
}
```

Now that we have completed these steps, when the InTouch application developer places our simple Wizard into a window and double-clicks on it, the following configuration dialog box will appear:



The InTouch application developer now has the ability to change both the Blink Expression and the Color for the blink "on" condition.

Wizard Toolkit Dialog Functions

The Wizard API also provides several functions that are worth mentioning at this point. Some of the functions provide standard InTouch user interfaces such as a script editor or key-equivalent handling. Other functions provide error checking and can alleviate some of the tedious error checking of values that should occur in the dialog procedures.

Function	Description
WWDlg_CheckExprCtrl	Validates the item using the standard InTouch validation of Script expressions. Error messages are automatically displayed when an error is detected.
WWDlg_CheckTagCtrl	Validates the dialog item using the standard InTouch validation of database tagnames. Error messages are automatically displayed when an error is detected.
WWDlg_GetDoubleCtrl	Validates the dialog item using standard InTouch validation rules for floating point values. The resulting value is returned.
WWDlg_ProcessKeyCtrl	Processes messages to the key-equivalent handling controls. Displays standard InTouch key-equivalent dialog if requested.
WWDlg_RegisterColorCtrl	Registers a dialog item to use the standard InTouch color choice dialog.
WWDlg_RegisterKeyCtrl	Registers a set of dialog items to obtain key-equivalent handling information for the Wizard.
WWDlg_RegisterTagnameCtrl	Registers a dialog item to respond to a double-click by displaying the standard tagname selection dialog.
WWDlg_ScriptEdit	Displays a generic script editing dialog.
WWDlg_SetDoubleCtrl	Sets the dialog item with the character representation for the floating point value specified.
WWDlg_UnregisterColorCtrl	Unregisters a dialog item that was registered using **WWDlg_RegisterColorCtrl**. Any memory used is freed and the current color selection is returned.

Function	Description
WWDlg_UnregisterKeyCtrl	Unregisters the set of dialog items to that were registered in **WWDlg_RegisterKeyCtrl**. Frees any memory associated with the dialog mapping.
WWDlg_UnregisterTagnameCtrl	Unregisters a dialog box item that was registered using **WWDlg_RegisterTagnameCtrl**. Any memory is freed and the control takes on its standard Windows capabilities.

CHAPTER 3

Wizard Toolkit Functions

This chapter describes the Wizard Toolkit Application Programming Interface (API) functions that are used to create and manipulate InTouch objects. The functions in the Wizard API are used by the Wizard Developer to implement the **Wizard_New** and **Wizard_Edit** functions for each wizard. A simple wizard may only need to access a few of these functions while a complex wizard may need to access many of them. The following briefly describes the categories of Wizard API functions and their primary purpose:

Wizard DLL:

Standard Functions	Required to build the wizard DLL and called by WindowMaker to access wizard functionality.

Wizard API Functions:

General Functions	Provide toolkit initialization, error checking and other routines that are typically used by all wizards.
Object Functions	Manipulate window objects such as, text, simple objects and complex trending and alarm objects.
Link Functions	Define animation, input/output functionality for window objects.
Utility Functions	Used for scaling fonts, points and rectangles.
Wizard Property Functions	Define the properties for configurable wizards.
User Interface Functions	Used in wizard user interfaces to provide consistent error checking and a common set of user interface controls.
Database Tagname Functions	Create, find and manipulate InTouch database entries, Access Names and Alarm Groups.

Contents

Wizard DLL Standard Functions

As a wizard developer you must create a Windows DLL for each set of wizards that you distribute. One or more wizards can be contained in a single DLL. A wizard DLL must support a standard set of functions. These functions are called by WindowMaker to access wizard functionality.

Function	Description
Wizard_New	Creates a new wizard. This function is called to place a new wizard, resize an existing wizard, or after the user has modified the wizard through the wizard's dialog.
Wizard_GetInfo	Returns information about a particular wizard for integration into the WindowMaker Wizard Set.
WizardLib_GetInfo	Returns information to WindowMaker about a Wizard Library.
Wizard_Edit	Brings up dialog box to edit the Wizard's configuration. You must supply a Windows dialog box to handle changing the wizard's configuration. Modify or edit wizard properties as appropriate and save if user clicks **OK**. Abandon changes if user selects **Cancel**.
Wizard_DoCommand	Allows the wizard to execute a process for command wizards. Required only for command wizards. For more information, see the "Command Wizards" section of Chapter 4.

Each of the Wizard_ functions, for example, **Wizard_New**, that you must supply for a wizard DLL take a parameter called *index*. This parameter is used when developing a wizard DLL library that contains more than one wizard. The *index* is a unique number that you use to identify each wizard in the wizard DLL. This *index* does not have to be unique across different wizard DLLs. *Index* must be a positive number greater than 0.

Note We recommend that once you have distributed a wizard DLL and assigned a unique number to each wizard, that you maintain the same numbers for the wizards in your DLL. Also, if you later discontinue the support of a wizard, do not reassign the wizard's unique number to another wizard or new wizard in your wizard DLL.

Wizard API Functions

The functions described in this section are used by the Wizard developer to implement the **Wizard_New** and **Wizard_Edit** functions for each wizard.

General Functions

General functions provide toolkit initialization, error checking, and other routines typically used by all wizards.

Function	Description
WWKit_GetKeyStatus	Retrieves the current status of the Wonderware hardware key.
WWKit_GetLastError	Returns the error status of the most recent call to the Wizard Toolkit.
WWKit_GetSerialNumber	Retrieves the serial number of the Wonderware hardware key.
WWKit_Init	Initializes the wizard toolkit if not previously done so. This call must be done at least once per wizard DLL. Normally this is done in Wizard_New.
WWKit_SetBrush	Sets the brush used when manipulating objects that have a brush associated with them.
WWKit_SetFont	Sets the font used when manipulating objects that have a font associated with them.
WWKit_SetPen	Sets the pen used when manipulating objects that have a pen associated with them.
WWKit_SetTextBrush	Sets the text brush used when manipulating objects that have a text brush associated with them.
WWKit_SetTextPen	Sets the text pen used when manipulating objects that have a text pen associated with them.

Object Functions

Object functions are used to manipulate window objects such as text, simple drawing objects, and the more complex trending and alarm objects. Using object functions the developer can create useful application specific objects and general purpose objects.

These functions create new window objects.

Function	Description
AlarmObj_New	Creates an alarm object at the specified location in the current application window.
BitmapObj_New	Creates a bitmap object at the specified location in the current application window.
ButtonObj_New	Creates a button object at the specified location in the current application window. Labels the button with the specified text.
DllObj_New	Creates a DLL object at the specified location in the current application window. Currently available DLL objects include the InTouch SPC pareto, histogram, and control chart objects.
EllipseObj_New	Creates an ellipse object at the specified location in the current application window.
GroupObj_New	Creates a group (cell) object at the specified location in the current application window. The Wizard can then populate the group with other objects by using this object's handle as the parent handle.
HistTrendObj_New	Creates a historical trend object at the specified location in the current application window.
LineObj_New	Creates a line object at the specified location in the current application window.
PolygonObj_New	Creates a polygon object at the specified location in the current application window.
PolylineObj_New	Creates a polyline object at the specified location in the current application window.
RealTrendObj_New	Creates a real time trend object at the specified location in the current application window.
RectangleObj_New	Creates a rectangle object at the specified location in the current application window.
RRectangleObj_New	Creates a rounded corner rectangle object at the specified location in the current application window.

Function	Description
SymbolObj_New	Creates a symbol object at the specified location in the current application window. The Wizard can then populate the symbol with other objects by using this object's handle as the parent handle.
TextObj_New	Creates a text object at the specified location in the current application window.
WizardObj_New	Creates a wizard object at the specified location in the current application window. Populate the wizard with other objects by using this object's handle as the parent handle. The following functions manipulate existing window objects.
Obj_Delete	Deletes the specified window object.
TrendObj_SetItem	Configures an item within the specified historical or real time trend object. Each item corresponds to a pen in the trend.
TrendObj_SetTimeInfo	Configures the time axis settings within the specified historical or real time trend object.
TrendObj_SetValueInfo	Configures the value axis settings within the specified historical or real time trend object.

Utility Functions

A set of utility functions is provided for scaling fonts, points, and rectangles.

Function	Description
Font_Scale	Linearly scales the logical font supplied using the old and new rectangles and the string specified.
PointReal_Scale	Linearly scales the point defined as floating point numbers supplied using the old and new rectangles specified.
PointRealArray_Scale	Linearly scales the array of points defined as floating point numbers supplied using the old and new rectangles specified.
Point_Scale	Linearly scales the point supplied using the old and new rectangles specified.
PointArray_Scale	Linearly scales the array of points supplied using the old and new rectangles specified.
Rect_Scale	Linearly scales the rectangle supplied using the old and new rectangles specified.
RectReal_Scale	Linearly scales the rectangle with REAL coordinates.
Text_GetExtent	Returns the width and height of the text in pixels, based upon the logical font specified. This function should be used instead of the Windows **GetTextExtent** function when calculating the metrics of text to be used in creating InTouch objects.

Link Functions

Link functions define animation and input/output functionality for window objects. These objects have been created using one of the functions in the Object Functions group.

Function	Description
AnlgAlarmLnk_New	Connects an analog alarm link to the object specified.
AnlgColorLnk_New	Connects an analog fill, text, or line link to the object specified.
AnlgInputLnk_New	Connects an analog input link to the object specified.
AnlgOutputLnk_New	Connects an analog output link to the object specified.
BlinkLnk_New	Connects a blink link to the object specified.
DisableLnk_New	Connects a disable link to the object specified.
DiscAlarmLnk_New	Connects a discrete alarm link to the object specified.
DiscColorLnk_New	Connects a discrete fill, text, or line link to the object specified.
DiscInputLnk_New	Connects a discrete input link to the object specified.
DiscOutputLnk_New	Connects a discrete output link to the object specified.
DiscTouchLnk_New	Connects a discrete touch link to the object specified.
LocationLnk_New	Connects a horizontal or vertical location link to the object specified.
OrientationLnk_New	Connects an orientation link that defines the specified object's angle of rotation.
PctFillLnk_New	Connects a horizontal or vertical percent fill link to the object specified.
SizeLnk_New	Connects a horizontal or vertical size link to the object specified.
SliderLnk_New	Connects a horizontal or vertical slider touch link to the object specified.

Function	Description
StmtTouchLnk_New	Connects an action touch link to the object specified. Statements can be associated with the up, down, and while down conditions to the object.
StrInputLnk_New	Connects a string (message) input link to the object specified.
StrOutputLnk_New	Connects a string (message) output link to the object specified.
VisibilityLnk_New	Connects a visibility link to the object specified.

The following function manipulates objects used in links.

Function	Description
Stmt_New	Creates and validates a block of statements and returns a handle to the validated statement.

Wizard Property Functions

Wizards are like "smart cells." These wizards contain all of the window objects necessary to define configurable and sizable cells. A wizard may have properties that define the smart cell's current configuration. When a user selects the smart cell for editing, the wizard that created the smart cell displays a dialog that allows the user to configure the smart cell. The configuration data is stored with the smart cell as wizard properties.

The following functions are used to retrieve and store wizard properties:

Function	Description
WizProp_Delete	Deletes the named wizard property.
WizProp_Find	Returns a handle to the named wizard property.
WizProp_GetBlock	Returns the data for a named wizard property that contains a block of data.
WizProp_GetDouble	Returns a floating point value for the named wizard property.
WizProp_GetDWord	Returns a double word (32-bit) value for the named wizard property.
WizProp_GetExpr	Returns the data for a named wizard property that contains an expression.

Function	Description
WizProp_GetFont	Returns a logical font (Windows standard) structure for the named wizard property. This function provides a platform-independent method to retrieve the logical font structure.
WizProp_GetStmt	Returns the data for a named wizard property that contains a statement.
WizProp_GetString	Returns a NULL terminated string for the named wizard property.
WizProp_New	Creates a wizard property with the name and type specified.
WizProp_SetBlock	Sets the data for a named wizard property that contains a block of data.
WizProp_SetDouble	Sets a floating point value for the named wizard property.
WizProp_SetDWord	Sets a double word (32-bit) value for the named wizard property.
WizProp_SetExpr	Sets the data for a named wizard property that contains an expression.
WizProp_SetFont	Sets a logical font (Windows standard) structure for the named wizard property. This function provides a platform-independent method to save the logical font structure.
WizProp_SetStmt	Sets the data for a named wizard property that contains a statement.
WizProp_SetString	Sets a NULL terminated string for the named wizard property.

User Interface Functions

A set of functions is provided for use in wizard user interfaces to provide consistent error checking and a common set of user interface controls.

Function	Description
WWDlg_CheckExprCtrl	Validates the item using the standard InTouch validation of QuickScript expressions. Error messages are automatically displayed when an error is detected.
WWDlg_CheckTagCtrl	Validates the dialog item using the standard InTouch validation of database tagnames. Error messages are automatically displayed when an error is detected.
WWDlg_GetDoubleCtrl	Validates the dialog item using standard InTouch validation rules for floating point values. The resulting value is returned.
WWDlg_ProcessKeyCtrl	Processes messages to the key-equivalent handling controls. Displays standard InTouch key-equivalent dialog if requested.
WWDlg_RegisterColorCtrl	Registers a dialog item to use the standard InTouch color choice dialog.
WWDlg_RegisterKeyCtrl	Registers a set of dialog items to obtain key-equivalent handling information for the wizard.
WWDlg_RegisterTagnameCtrl	Registers a dialog item to respond to a double-click by displaying the standard tagname selection dialog.
WWDlg_ScriptEdit	Displays a generic script editing dialog.
WWDlg_SetDoubleCtrl	Sets the dialog item with the character representation for the floating point value specified.
WWDlg_UnregisterColorCtrl	Unregisters a dialog item that was registered using **WWDlg_RegisterColorCtrl**. Any memory used is freed and the current color selection is returned.
WWDlg_UnregisterKeyCtrl	Unregisters the set of dialog items that were registered in **WWDlg_RegisterKeyCtrl**. Frees any memory associated with the mapping of dialog items.
WWDlg_UnregisterTagnameCtrl	Unregisters a dialog item that was registered using **WWDlgRegisterTagnameCtrl**. Any memory is freed and the control takes on its standard Windows capabilities.

Database Tag Functions

Database tag functions are used to create, find, and manipulate InTouch database entries, access names and alarm groups.

The following functions create, find and delete database tags.

Function	Description
Tag_Find	Returns the handle of the database tagname with the given name.
Tag_FindApplTopicItem	Returns the handle of the database tagname with the specified application (server), topic, and item description.
Tag_New	Creates a database tagname with the specified name, type, and comment.

The following functions set and retrieve information for a database tag.

Function	Description
Tag_GetAccessInfo	Returns general access information for a database tagname.
Tag_GetGroup	Returns the group handle for the database tagname with the given handle.
Tag_GetInfo	Returns general information for a database tagname.
Tag_GetRetentiveInfo	Returns retentive information for a database tagname.
Tag_GetUniqueName	Returns a unique database name from supplied base name.
Tag_GetValueAlarm	Returns value alarm fields for a database tagname.
Tag_SetAccessInfo	Set general access information for a database tagname.
Tag_SetDeviationAlarm	Sets deviation alarm fields in a given tagname.
Tag_SetDiscAlarm	Sets discrete alarm fields for a database tagname.
Tag_SetEventInfo	Sets event logging information for a database tagname.
Tag_SetGroup	Sets the group handle for the database tagname with the given handle.
Tag_SetInfo	Sets general information for a database tagname.
Tag_SetRateOfChangeAlarm	Sets rate of change alarm fields in a database tagname.

Function	Description
Tag_SetRetentiveInfo	Sets retentive information for a database tagname.
Tag_SetScalingInfo	Sets scaling information for a database tagname.
Tag_SetValueAlarm	Sets value alarm fields for a database tagname.

The following functions set and return information about tagnames that are type-specific.

Function	Description
AnlgTag_GetInfo	Returns the initial value of an analog tagname.
AnlgTag_SetInfo	Sets the initial value of an analog tagname.
DiscTag_GetInfo	Returns the initial value of an discrete tagname.
DiscTag_SetInfo	Sets the initial value of an discrete tagname.
StrTag_SetInfo	Sets the initial value of a string tagname.

The following functions manipulate access names required for DDE access name tags.

Function	Description
AccessName_Find	Returns the handle of the Access Name with the given name.
AccessName_FindApplTopic	Returns the handle of the Access Name with the given application and topic.
AccessName_GetInfo	Returns access information and the given access name ID (accID).
AccessName_GetName	Returns a pointer to the Access Name via lpSourceName.
AccessName_GetUniqueName	Returns a unique Access Name in the string specified by Access Name, based on string specified in basename.
AccessName_New	Creates an Access Name with the specified name and settings.
AccessName_SetInfo	Sets access information into a I/OSOURCE.
AccessName_SetName	Sets Access Name specified in the lpSourceName.

CHAPTER 4

User Supplied Wizard Functions

This chapter describes in detail, the user supplied functions that are required for the proper integration and execution of InTouch wizards. The functions' purpose, syntax, parameters, and return values are also described.

Contents

Functions Required to Create and Configure Wizards

There are two functions that must be supplied in order for InTouch to place and configure wizards: **Wizard_New** and **Wizard_Edit**. **Wizard_New** is called when the wizard is initially placed, resized, or edited. **Wizard_Edit** is called when the InTouch application developer double-clicks on a wizard and provides the Wizard developer with the opportunity to bring up a dialog and solicit input from the InTouch application developer for configuration.

Wizard_New

```
int

Wizard_New(  HCHUNK hChunk,
             int index,
             int left,
             int top,
             int right,
             int bottom,
             LPSTR dllName,
             WHMEM whData,
             int mode,
             RECT prevRect,
             WHMEM FAR *pwhWizard)
```

Description

Called when a user places, resizes, or edits a wizard.

Parameter	Description
index	Refers to the index number of a particular wizard in a library.
left	Left Coordinate of wizard.
top	Top Coordinate of wizard.
right	Right Coordinate of wizard.
bottom	Bottom Coordinate of wizard.
dllName	Name of DLL containing wizard code.
whData	Pointer to wizard properties.
mode	The reason Wizard_New was called.

Value	Meaning
MODE_NEW	Wizard initially placed
MODE_SIZE	Wizard sized
MODE_EDIT	Wizard edited
MODE_RESTORE	An undo was issued

Parameter	Description
prevRect	Previous rectangle coordinates for **Wizard_New** call.
pwhWizard	Pointer to Wizard Object.

Return Value Error Code for configuration. Otherwise, 0 if there were no errors.

Comments **Wizard_New** is called when the user initially places a wizard, resizes a wizard, or configures a wizard. The *mode* parameter indicates the condition that caused the **Wizard_New** to be called.

Wizard_Edit

```
int
Wizard_Edit( int index,
             WHMEM whData,
             HCHUNK hChunk)
```

Description Allows the user to configure the wizard.

Parameter	Description
index	Refers to the index number of a particular wizard in a library.
whData	Pointer to wizard properties.
hChunk	Memory allocation handle

Return Value Error Code for configuration. Otherwise, 0 if there were no errors.

Comments **Wizard_Edit** is called when the user double-clicks on a wizard. Typically a dialog box will appear that will allow the user to configure specific properties. The wizard developer will bring up this dialog whenever **Wizard_Edit** is called by WindowMaker If there is no **Wizard_Edit** routine, the Wizard is not configurable.

Functions Required to Integrate Wizards into InTouch

To fully integrate a wizard library into the WindowMaker environment two functions must be supplied: **Wizard_GetInfo** and **WizardLib_GetInfo**. **Wizard_GetInfo** should return information about a particular wizard and **WizardLib_GetInfo** should return information about a given Wizard Library

Wizard_GetInfo

```
BOOL

Wizard_GetInfo( int index,
                WORD wCommand,
                DWORD dwData,
                LPVOID lpInfo)
```

Description

Returns information about the wizard based on the value of *dwData.*

Parameter	Description
index	Refers to the index number of a particular wizard in a library.
wCommand	Specifies which property it is requesting information about.
dwData	Specifies more information used in retrieving information about a specific wizard.
lpInfo	Pointer to the returned information

Return Value

TRUE if information was returned. Otherwise, FALSE.

Comments

The values requested in *wCommand* and the required actions are listed as follows:

Command	Required Action
WIZ_BITMAP	Return HBITMAP for selection dialog.
WIZ_DESCRIPTION	Return wizard description string.
WIZ_FLAGS	Return general flags for wizard.

Value	Meaning
WIZFLAG_NONLITE	Set if NON-LITE mode wizard.
WIZFLAG_NOBREAK	Set if NOT breakable into components.
WIZFLAG_COMMAND	Set if command wizard.

Command	Required Action
WIZ_GROUPNAME	Return wizard's group descriptive name.
WIZ_HELPINFO	Return context sensitive help information.
WIZ_SIZEMODE	Return size mode flags for wizard.
WIZSIZE_ASPECT	Size X and Y retaining aspect (default).
WIZSIZE_FULL	Unrestricted sizing.
WIZSIZE_NONE	Not Supported.
WIZ_TBOXBITMAP	Return HBITMAP for toolbar.

The WIZ_BITMAP, and WIZ_TBOXBITMAP commands will use the default bitmaps if one is not supplied. To supply bitmaps for the wizard, there are three bitmaps needed:

- 64x64 bitmap for the Wizard Selection Dialog
- 16x16 bitmap for toolbar when the button is up
- 16x16 bitmap for toolbar when the button is pressed

Note The pair of bitmaps used in the toolbar must be identical, except the fill for the button up should be "buttonface" gray and the fill for the button down should be white. WindowMaker will automatically create the 3-D border for you. The bitmap you provide should not be shifted over and down. WindowMaker will handle the "pressed in" effect automatically when the button is depressed.

WizardLib_GetInfo

```
BOOL
WizardLib_GetInfo(   int index,
                     WORD wCommand,
                     DWORD dwData,
                     LPVOID lpInfo)
```

Description

Returns information about the wizard library based on the value of *dwData.*

Parameter	Description
index	Refers to the index number of a particular wizard in a library.
wCommand	Specifies which property it is requesting information about.
dwData	Specifies more information used in retrieving information about the wizard library.
lpInfo	Pointer to the returned information

Return Value

TRUE if information was returned. Otherwise, FALSE.

Comments

The values requested in *wCommand* and the required actions are listed in the following table.

Command	Required Action
WIZ_COMPANYNAME	Return company description string.
WIZ_LIBNAME	Return library descriptive name.
WIZ_NEXTWIZID	Return next wizard ID based on the current wizard ID passed in dwData. Return the first wizard ID, (generally 1) when dwData is 0. Return a 0 to indicate that dwData is the last wizard ID.
WIZ_VERSIONNUM	Return library version number.
WIZ_VERSIONSTR	Return library version description string.

Command Wizards

Command wizards are wizards that start processes or applications and do not place or edit objects. Command wizards are specified by returning WIZFLAG_COMMAND for the case WIZ_FLAGS in the function **Wizard_GetInfo**. If this flag is set, then the functions **Wizard_DoCommand** must be provided.

Wizard_DoCommand

```
int
Wizard_DoCommand(  int index)
```

Description

Allows the user to execute an application or run a process, such as converting a database.

Parameter	Description
index	Refers to the index number of a particular wizard in a library.

Return Value

Error Code for configuration. Otherwise, 0 if there were no errors.

Comments

Command wizards do not place objects on the window, or allow editing. They simply execute the code in **Wizard_DoCommand**. **Wizard_DoCommand** is called when the InTouch application developer selects a command wizard from the Wizard Selection dialog box via the wizard tool or clicks a command wizard button on the toolbar.

CHAPTER 5

Style Guide for Wizard Library Development

This chapter contains our recommended style guide for Wizard libraries. These guidelines are provided to help you implement good basic wizard development practices. Taking some time to learn these basic naming conventions now may save you a lot of time later!

Contents

Guidelines for Wizard Library Development

Each wizard library should be named with a unique mnemonic (referred to as <name> in this document) that identifies the type of wizards in the library. For example, SWITCH, METER, and so on. This short abbreviated name will be used throughout the wizard library to generate names for specific files, functions, defines and declarations.

Creating Libraries with Multiple Wizards

Some changes are required to build a wizard library with multiple wizards. If you follow these simple naming conventions, the routines become very generic and can easily handle wizard libraries with any number of wizards. The samples provided for **WizardGet_Info** and **WizardLib_GetInfo** use these naming conventions.

Assign each wizard in the library a two-digit ID. For example, 01 for the first wizard. The <name> and <id> will be combined in many places to name files, functions and defines. For example, METER01.C or Meter01New.

Use a sample wizard library to obtain standard file name examples, such as WZMAIN.C, WZSTUB.C. A two-letter prefix should be used to identify your wizards from any wizards produced by other sources. This will help eliminate wizard DLLs of the same name. All of the wizards built into InTouch use WZ as the two-letter identifier. Choose one that makes sense for you.

Wizard Library Directory

The naming convention for the directory should be:

<id prefix><name> (for example, WZMETER).

The naming convention for the makefile should be:

named <id prefix><name>.MAK (for example, WZMETER.MAK).

The makefile will cause the DLL <id prefix><name>.DLL to be built.

Wizard C Modules

Use the standard WZMAIN.C and WZSTUB.C files for the common C modules. The WZMAIN.C file will have a WZMAIN.H header file for the global declarations. Typically this H file will have the globals containing the default values and the string globals containing the property names.

Each wizard NEW function should be placed in its own C module named using the form <name><id> where <id> is a two digit ID for the wizard. (for example, METER02.C will have the wizard new function **Meter02New**).

Since it will be very common for a single dialog to be shared by multiple wizards, each unique wizard dialog should be placed in its own C module. This module should be named using the form <name><letter> where <letter> is a unique letter from A to Z assigned to each unique dialog in the wizard library. (for example, METERA.C will have the wizard dialog procedure **MeterADlgProc** for the dialog resource named METERADLG in the resource file.)

Function Names

<name><letter>DlgProc	Dialog procedure for a unique wizard dialog.
	For example: MeterADlgProc
<name><id>New	Wizard new function for a wizard.
	For example: Meter01New

WZMAIN.C

To create the **WZMAIN.C** file, modify one of a sample wizards that was supplied with the **Wizard Toolkit**. Make sure to replace the <name> for the sample library with your <name> for your wizard library.

Change the include wz<name>.h.

Add the necessary forward declarations for the wizard new functions (for example, Meter01New). Remove the ones for the original sample wizards.

Add the necessary forward declarations for the wizard dialog procedures (for example, MeterADlgProc). Remove the ones for the original sample wizards.

Add the cases to **Wizard_Edit** to call each of the unique dialogs for the wizards. For example:

```
case 1:
case 2:
    DialogBox( hDrawInst, "METERADLG", hDrawWnd,
        MeterADlgProc );
    break;
case 3:
    DialogBox( hDrawInst, "METERBDLG", hDrawWnd,
        MeterBDlgProc );
    break;
```

In this example, two wizards (IDs 1 and 2) use the same dialog "METERADLG."

Add the cases to **Wizard_New** to call each of the wizard new functions for each wizard. For example,

```
case 1:
    error = Meter01New( hChunk, index, left, top, right,
        bottom, dllName, whProperties, pwhWizard );
    break;
case 2:
    error = Meter02New( hChunk, index, left, top, right,
        bottom, dllName, whProperties, pwhWizard );
    break;
case 3:
    error = Meter03New( hChunk, index, left,
        top, right, bottom, dllName, whProperties, pwhWizard
);
    break;
```

Header File

Create a header file named <id prefix><name>.h (for example, WZMETER.H) for your wizard library file. It will contain control IDs for dialogs, defines for properties, and external definitions for globals containing property names and property defaults. These property name and default globals are declared in WZMAIN.H.

Dialog control IDs are associated with a unique dialog, not a unique wizard. Therefore, name your control IDs based on each unique dialog they belong to. Use the naming convention ID_<name><letter>_<description>. For example, the controls used by wizards of type MeterA would be named:

```
#define ID_METERA_EXPR                  101
#define ID_METERA_GAUGEGROUP            105
#define ID_METERA_GAUGETEXT             106
#define ID_METERA_GAUGEFILLCOLOR        107
#define ID_METERA_GAUGETEXTCOLOR        108

#define ID_METERA_RANGEGROUP            110
#define ID_METERA_MIN                   111
#define ID_METERA_MAX                   112

#define ID_METERA_TICKGROUP             115
#define ID_METERA_MAJORDIV              116
#define ID_METERA_MINORDIV              117

#define ID_METERA_LABELGROUP            120
#define ID_METERA_DISPLAYLABEL          121
#define ID_METERA_LABELTEXTCOLOR        122
#define ID_METERA_NUMFORMAT             123
```

Remember, property names are usually associated with a unique dialog, not a unique wizard. Therefore, name your property defines based on the corresponding unique dialog. Use the naming convention PROP_<name><letter>_<description>. For example, the properties used by wizards of type MeterA would be named:

```
#define PROP_METERA_EXPR              exprStr
#define PROP_METERA_GAUGETEXT         gaugeLabelStr
#define PROP_METERA_GAUGEFILLCOLOR    gaugeFillColorStr
#define PROP_METERA_GAUGETEXTCOLOR    gaugeTextColorStr
#define PROP_METERA_LABELTEXTCOLOR    labelTextColorStr
#define PROP_METERA_MIN               minValueStr
#define PROP_METERA_MAX               maxValueStr
#define PROP_METERA_MAJORDIV          majorDivStr
#define PROP_METERA_MINORDIV          minorDivStr
#define PROP_METERA_DISPLAYLABEL      displayLabelStr
```

It is very possible for more than one wizard to use the property defined in this example. There should be a dialog for the defined properties that is named METERADLG in the resource file.

Default property defines may either be defined as constants or references to globals declared in WZMAIN.H. For these defines use the naming convention DEF_<name><id>_<description>. For example, the default defines for button defaults would be named:

```
/* Button 5 */

#define DEF_BUTTN05_ON_COLOR     RGB( 0x00, 0xff, 0x00 )
#define DEF_BUTTN05_OFF_COLOR    RGB( 0xff, 0x00, 0x00 )

    /* Button 6 */

#define DEF_BUTTN06_ON_COLOR     RGB( 0xff, 0x00, 0x00 )
#define DEF_BUTTN06_OFF_COLOR    RGB( 0x80, 0x80, 0x80 )

    /* Button 7 */

#define DEF_BUTTN07_ON_COLOR     RGB( 0x00, 0xff, 0x00 )
#define DEF_BUTTN07_OFF_COLOR    RGB( 0xff, 0x00, 0x00 )
```

An example for the defaults for the wizards in the TEXT wizard library would be:

```
#define DEF_TEXT01_TAGNAME       tagnameStr
#define DEF_TEXT01_LABEL         "Title"
```

Definition (.DEF) File

Name the DEF file for the wizard library <id prefix><name>.DEF.

For example: WZMETER.DEF

The LIBRARY statement should name the library <id prefix><Name>

For example: WZMETER

Remember to export all dialog procedures in addition to the standard wizard DLL entry points. The standard wizard DLL entry points are:

- **Wizard_New**
- **Wizard_Edit**
- **Wizard_GetInfo**
- **WizardLib_GetInfo**

Resource (.RC) File

Name the resource file <id prefix><name>.RC.

For example: WZMETER.RC

Name all bitmap files using the following convention:

<name><id>M.BMP	for Wizard selection dialog box bitmap (for example, METER04M.BMP)
<name><id>T.BMP	for Toolbar bitmap (for example, METER04T.BMP)
<name><id>P.BMP	for Toolbar pushed in bitmap (for example, METER04P.BMP)

Name each bitmap resource the same as the filename, without the period (.). For example, name all bitmap resources using the following conventions:

<name><id>MBMP	for Wizard selection dialog box bitmap (for example, METER04M.BMP)
<name><id>TBMP	for Toolbar bitmap (for example, METER04TBMP)
<name><id>PBMP	for Toolbar pushed in bitmap (for example, METER04P.BMP)

For example:

```
METER02TBMP        BITMAP  MOVEABLE PURE   "METER02T.BMP"
```

Name each unique wizard dialog resource <name><letter>DLG where <letter> is a unique letter from A to Z assigned to each unique dialog in the wizard library.

For example: METERADLG

```
METERADLG DIALOG DISCARDABLE  9, 24, 250, 66
```

We recommend that all wizard descriptions reside in the STRINGTABLE for the wizard library. The string table identifier for each wizard is generated using the following formula:

Long Description <id> * 2 - 1

Short Description <id> * 2

```
STRINGTABLE
BEGIN
1,    "Wizard ID #1 Long Description"
2,    "Wizard ID #1 Short Description"
3,    "Wizard ID #2 Long Description"
4,    "Wizard ID #2 Short Description"
5,    "Wizard ID #3 Long Description"
6,    "Wizard ID #3 Short Description"
END
```

Put all wizard descriptions in the STRINGTABLE for the wizard library. The string table identifier for each wizard is generated using the following formula:

Long Description <id> * 2 - 1

Short Description <id> * 2

CHAPTER 6

Wizard API Function Reference

This chapter is a complete reference manual for the Wizard Toolkit Application Programming Interface (API). Wizards are implemented using functions in the Wizard API. All of the Wizard API functions are documented in this chapter. They are presented in alphabetic order. The purpose, syntax, parameters and possible return values for all functions are included.

Contents

AccessName_Find

```
I/OSOURCE
AccessName_Find( LPSTR srcName)
```

Description Returns the handle of the Access Name with the given name.

Parameter	Description
srcName	Points to a null-terminated string containing the Access Name.

Return Value Handle to the Access Name found. Otherwise, it is 0.

Comments None.

AccessName_FindApplTopic

```
I/OSOURCE
AccessName_FindApplTopic(  LPSTR application,
                           LPSTR topic)
```

Description Returns the handle of the Access Name with the given application and topic.

Parameter	Description
application	Points to a null-terminated string containing the application name (for example, "EXCEL"). The application can also include the node name (for example, "\\NODE\EXCEL").
topic	Points to a null-terminated string containing the I/O topic (for example, "SHEET1.XLS").

Return Value Handle to the Access Name found. Otherwise, it is 0.

Comments None.

AccessName_GetInfo

```
int
AccessName_GetInfo( I/OSOURCE accID,
                    LPACCESSINFO IpInfo)
```

Description Returns the access information in the given Access Name ID (*accID*).

Parameter	Description
accID	I/OSOURCE that specifies a particular Access Name.
lpInfo	Pointer to the access information structure.

Return Value Error code or 0 if successful.

Comments An error will be returned if the *accID* is an invalid I/O Source ID.

AccessName_GetName

```
int
AccessName_GetName( I/OSOURCE accID,
                    LPSTR IpSourceName)
```

Description

Returns a pointer to the Access Name via lpSourceName for the given Access Name ID (*accID*).

Parameter	Description
accID	I/OSOURCE that specifies a particular Access Name.
lpSouceName	Pointer to a string containing the Access Name associated with *accID*.

Return Value

Error code or 0 if successful.

Comments

An error will be returned if the *accID* is an invalid I/O Source ID.

AccessName_GetUniqueName

```
int
AccessName_GetName( LPSTR basename,
                    LPSTR accessname)
```

Description

Returns a unique I/O Access Name in the string specified by Access Name, based on the string specified in base name.

Parameter	Description
basename	Pointer to a string containing the base for the unique Access Name.
accessname	Pointer to a string containing the unique Access Name.

Return Value

Error code or 0 if successful.

Comments

An error will be returned if a unique name cannot be found.

AccessName_New

```
I/OSOURCE
AccessName_New( LPSTR srcName,
                LP_ACCESSNAMEINFO lpInfo)
```

Description Creates an Access Name with the specified name and settings.

Parameter	Description
srcName	Points to a null-terminated string containing the Access Name.
lpInfo	Points to an ACCESSNAMEINFO structure that contains the Access Name settings.

Return Value The return value is the handle of the Access Name if the function is successful. Otherwise, it is 0.

Comments This function will fail if the Access Name already exists.

AccessName_SetInfo

```
int
AccessName_SetInfo( I/OSOURCE accID,
                    LP_ACCESSNAME lpInfo)
```

Description Sets the Access Name information into the I/OSOURCE specified in *accID*.

Parameter	Description
accID	I/OSOURCE identifier.
lpInfo	Pointer to the Access Name information structure.

Return Value Error code.

Comments None.

AccessName_SetName

```
int

AccessName_SetName( I/OSOURCE accID,
                    LPSTR lpSourceName)
```

Description

Sets the Access Name specified in lpSourceName for the given Access Name ID (*accID*).

Parameter	Description
accID	I/OSOURCE that specifies a particular Access Name.
lpSourceName	Pointer to a string containing the Access Name associated with *accID*.

Return Value

Error code or 0 if successful.

Comments

An error will be returned if the *accID* is an invalid I/O Source ID.

AlarmObj_New

```
WHMEM

AlarmObj_New( HCHUNK hChunk,
              WHMEM whParent,
              int left,
              int top,
              int right,
              int bottom,
              WORD alarmType,
              WORD options,
              LONG windowColor,
              LONG borderColor,
              LONG titleBarColor,
              LONG titleTextColor,
              LONG unAckAlmColor,
              LONG ackColor,
              LONG rtnColor,
              LONG evtColor,
              LPSTR alarmFormat,
              LPSTR alarmGroup,
              LPSTR fromPriority,
              LPSTR toPriority,
              LPSTR prevPageTagname,
              LPSTR nextPageTagname)
```

Description Creates an alarm object at the specified location in the current application window.

Parameter	Description
hChunk	Handle to the memory section containing the object.
whParent	Handle to the parent object (symbol, group, or wizard) that will contain this object. 0 indicates there is no parent object.
left	Specifies the x-coordinate of the upper-left corner.
top	Specifies the y-coordinate of the upper-left corner.
right	Specifies the x-coordinate of the lower-right corner.
bottom	Specifies the y-coordinate of the lower-right corner.
alarmType	Specifies the flags that determine the alarm type. This parameter can be one of the following values:

Value	Meaning
ALARM_SUMMARY	Specifies an alarm summary object.
ALARM_HISTORY	Specifies an alarm history object.

options	Specifies the flags that determine the alarm object options. This parameter can be a combination of the following values:

Value	Meaning
ALARM_ TITLES	Specifies a title bar for the alarm object with labels for each column.
ALARM_ SERVER	Specifies the display of alarms/events collected by the server node. This is used in conjunction with the master/slave configuration.

windowColor	Specifies the alarm object's background color. Colors are specified in Windows standard RGB format.
borderColor	Specifies the color of the alarm object's border. Colors are specified in Windows standard RGB format.
titleBarColor	Specifies the color of the alarm object's title bar. Colors are specified in Windows standard RGB format.
titleTextColor	Specifies the color of the alarm object's column titles in the title bar. Colors are specified in Windows standard RGB format.

Parameter	Description
unAckAlmColor	Specifies the text color of the alarm object's unacknowledged alarms. Colors are specified in Windows standard RGB format.
ackColor	Specifies the text color of the alarm object's acknowledged alarms. Colors are specified in Windows standard RGB format.
rtnColor	Specifies the text color of the alarm object's return to normal alarms. Colors are specified in Windows standard RGB format. Not used for ALARM_SUMMARY type.
evtColor	Specifies the text color of the alarm object's events. Colors are specified in Windows standard RGB format. Not used for ALARM_SUMMARY type.
alarmFormat	Points to a null-terminated string containing the format specification for the alarm information.
alarmGroup	Points to a null-terminated string containing the alarm group tagname or group variable tagname to use for the link. For example, $**System** displays all alarms in all groups.
fromPriority	Points to a null-terminated string containing the specification for the highest alarm priority level for the range of priorities that will be displayed in the alarm object. An analog tagname or constant value can be specified. For example, "FromPri" or "1".
toPriority	Points to a null-terminated string containing the specification for the lowest alarm priority level for the range of priorities that will be displayed in the alarm object. An analog tagname or constant value can be specified. For example, "ToPri" or "999".
prevPageTagname	Points to a null-terminated string containing the discrete tagname used to start the display to page up. An empty string indicates no page up capability.
nextPageTagname	Points to a null-terminated string containing the discrete tagname used to start the display to page down. An empty string indicates no page down capability.

Return Value

The return value is the handle of the object if the function is successful. Otherwise, it is *whNull*.

Comments

None.

AnlgAlarmLnk_New

```
WHMEM

AnlgAlarmLnk_New( HCHUNK hChunk,
                  WHMEM whObj,
                  WORD linkType,
                  WORD alarmType,
                  LPSTR tagname,
                  LONG FAR colors[5])
```

Description

Creates an analog alarm link for the object specified.

Parameter	Description
hChunk	Handle to the memory section containing the object for which the link is being created.
whObj	Handle to the object for which the link is being created.
linkType	Specifies the flags that determine the link type. This parameter can be one of the following values:

Value	Meaning
LINE_LINK	Specifies a line color link based on alarm status.
TEXT_LINK	Specifies a text color link based on alarm status.
FILL_LINK	Specifies a fill color link based on alarm status.

alarmType	Specifies the flags that determine the alarm type. This parameter can be one of the following values:

Value	Meaning
VALUE_ALARM_LINK	Specifies a value alarm link. A value alarm can be in one of five states: LoLo, Lo, Normal, Hi, HiHi.
DEV_ALARM_LINK	Specifies a deviation alarm link. A deviation alarm can be in one of three states: Normal, Minor, Major.
ROC_ALARM_LINK	Specifies a rate of change (ROC) alarm link. A ROC alarm has two states: Normal, ROC.

Parameter	Description
tagname	Points to a null-terminated string containing the analog tagname to use for the link.
colors	Points to an array of five LONG types. Each LONG specifies an alarm state color. Colors are specified in Windows standard RGB format. The index for each item in the array is defined as follows for the VALUE_ALARM_LINK alarm type.

Value	Meaning
0	Specifies the color for the LoLo alarm status.
1	Specifies the color for the Lo alarm status.
2	Specifies the color for the Normal alarm status.
3	Specifies the color for the Hi alarm status.
4	Specifies the color for the HiHi alarm status.
	The index for each item in the array is defined as follows for the DEV_ALARM_LINK alarm type.
0	Specifies the color for the Normal alarm status.
1	Specifies the color for the Minor Deviation alarm status.
2	Specifies the color for the Major Deviation alarm status.

Value	Meaning
3,4	Unused, but must be passed.
	The index for each item in the array is defined as follows for the ROC_ALARM_LINK alarm type.
0	Specifies the color for the Normal alarm status.
1	Specifies the color for the ROC alarm status.
2,3,4	Unused, but must be passed.

Return Value The return value is the handle of the link if the function is successful. Otherwise, it is *whNull*.

Comments If a zero (0) is returned, check for invalid tagname, link type or alarm type.

AnlgColorLnk_New

```
WHMEM

AnlgColorLnk_New( HCHUNK hChunk,
                  WHMEM whObj,
                  WORD linkType,
                  LPSTR expression,
                  REAL FAR value[4],
                  LONG FAR colors[5])
```

Description

Creates an analog fill, text, or line link for the object specified. Five value ranges are defined by specifying four breakpoints. Five different colors can be selected which will be displayed as the value range changes.

Parameter	Description
hChunk	Handle to the memory section containing the object for which the link is being created.
whObj	Handle to the object for which the link is being created.
linkType	Specifies the flags that determine the link type. This parameter can be one of the following values:

Value	Meaning
LINE_LINK	Specifies a line color link based on alarm status.
TEXT_LINK	Specifies a text color link based on alarm status.
FILL_LINK	Specifies a fill color link based on alarm status.

expression	Points to a null-terminated string containing the analog expression or tagname to use for the link.
value	Points to an array of four REAL types. Each REAL specifies a value range breakpoint. Four breakpoints are used to define the five value ranges. Each item in the array must be greater than the previous item.
colors	Points to an array of five LONG types. Each LONG specifies a value range color. Index 0 of the array corresponds to the lowest value range. Index 4 of the array corresponds to the highest value range. Colors are specified in Windows standard RGB format.

Return Value

The return value is the handle of the link if the function is successful. Otherwise, it is *whNull*.

Comments

If a zero (0) is returned, check for invalid analog expression or link type.

AnlgInputLnk_New

```
WHMEM

AnlgInputLnk_New( HCHUNK hChunk,
                  WHMEM whObj,
                  LPSTR tagname,
                  LPSTR userMsg,
                  BOOL bUseKeypad,
                  BOOL bInputOnly,
                  BYTE cKeyFlags,
                  WORD wVirtKey,
                  REAL minValue,
                  REAL maxValue)
```

Description

Creates an analog input link for the object specified.

Parameter	Description
hChunk	Handle to the memory section containing the object for which the link is being created.
whObj	Handle to the object for which the link is being created.
tagname	Points to a null-terminated string containing the analog tagname to use for the link.
userMsg	Points to a null-terminated string containing the message or instruction to display if the *bUseKeypad* option is enabled.
bUseKeypad	Specifies the use of an on-screen keypad for entering new values if this value is non-zero.
bInputOnly	Specifies this link as input only, the value entered will not be displayed, if this value is non-zero. This setting only applies to objects that have text display associated with them (for example, a push button).
cKeyFlags	Specifies the flags used when a keyboard key is assigned to this link. This parameter can be a combination of the following values:

Value	Meaning
TOUCH_KS_SHIFT	Specifies the SHIFT key must be held down in addition to the key specified.
TOUCH_KS_CTRL	Specifies the CTRL key must be held down in addition to the key specified.
0	Specifies that only the specified key must be held down.

Parameter	Description
wVirtKey	Specifies the keyboard key equivalent assigned to this link. This value is 0 if there is no keyboard equivalent.
minValue	Specifies the minimum allowable input value.
maxValue	Specifies the maximum allowable input value.

Return Value The return value is the handle of the link if the function is successful. Otherwise, it is *whNull*.

Comments If a zero (0) is returned, check for invalid tagname or too long a *userMsg* variable.

AnlgOutputLnk_New

```
WHMEM
AnlgOutputLnk_New(  HCHUNK hChunk,
                    WHMEM whObj,
                    LPSTR expression)
```

Description Creates an analog output link for the object specified.

Parameter	Description
hChunk	Handle to the memory section containing the object for which the link is being created.
whObj	Handle to the object for which the link is being created.
expression	Points to a null-terminated string containing the analog expression or tagname to use for the link.

Return Value The return value is the handle of the link if the function is successful. Otherwise, it is *whNull*.

Comments If a zero (0) is returned, *expression* is NULL, too long or invalid.

AnlgTag_GetInfo

```
int
AnlgTag_GetInfo(  DBHND dbHnd,
                  LP_ANLGTAGINFO lpAnlgInfo)
```

Description Returns the analog tagname for the database tagname with the given handle.

Parameter	Description
dbHnd	Handle to the database tagname.
lpAccessInfo	Pointer to the analog tagname information structure.

Return Value Error code.

Comments None.

AnlgTag_SetInfo

```
int
AnlgTag_SetInfo(  DBHND dbHnd,
                  LP_ANLGTAGINFO lpAnlgInfo)
```

Description Sets the analog tagname information for a database tagname with the given handle.

Parameter	Description
dbHnd	Handle to the database tagname.
lpAnlgInfo	Pointer to the analog tagname information structure.

Return Value Error code.

Comments None.

BitmapObj_New

```
WHMEM
BitmapObj_New( HCHUNK hChunk,
               WHMEM whParent,
               int left,
               int top,
               int right,
               int bottom,
               HBITMAP hBitmap)
```

Description Creates a bitmap object at the specified location in the current application window.

Parameter	Description
hChunk	Handle to the memory section containing the object.
whParent	Handle to the parent object (symbol, group, or wizard) that will contain this object. 0 indicates there is no parent object.
left	Specifies the x-coordinate of the upper-left corner.
top	Specifies the y-coordinate of the upper-left corner.
right	Specifies the x-coordinate of the lower-right corner.
bottom	Specifies the y-coordinate of the lower-right corner.
hBitmap	Handle of the bitmap to "paste" into the object. The bitmap is a standard Windows bitmap.

Return Value The return value is the handle of the object if the function is successful. Otherwise, it is *whNull*.

Comments The object will be created if *hBitmap* is NULL.

BlinkLnk_New

```
WHMEM
BlinkLnk_New(   HCHUNK hChunk,
                WHMEM whObj,
                LPSTR expression,
                BOOL bInvisibleWhenBlinked,
                LONG lineColor,
                LONG fillColor,
                LONG textColor,
                int blinkSpeed)
```

Description

Creates a blink link for the object specified.

Parameter	Description
hChunk	Handle to the memory section containing the object for which the link is being created.
whObj	Handle to the object for which the link is being created.
expression	Points to a null-terminated string containing the analog expression or tagname to use for the link.
bInvisibleWhenBlinked	Specifies that the object will blink by disappearing and reappearing if this value is non-zero. Otherwise, the object will blink by changing the specified line, fill and text color attributes.
lineColor	Specifies the object's line color. Colors are specified in Windows standard RGB format.
fillColor	Specifies the object's fill color. Colors are specified in Windows standard RGB format.
textColor	Specifies the object's text color. Colors are specified in Windows standard RGB format.
blinkSpeed	Specifies the blinking speed for the object. The following values are defined to specify the blink speed.

Value	Meaning
BLINK_SLOW	Specifies the slow blink speed as defined in InTouch.
BLINK_MEDIUM	Specifies the medium blink speed as defined in InTouch.
BLINK_FAST	Specifies the fast blink speed as defined in InTouch.

Return Value

The return value is the handle of the link if the function is successful. Otherwise, it is *whNull*.

Comments

If a zero (0) is returned, if *expression* is NULL, too long or invalid.

ButtonObj_New

```
WHMEM

ButtonObj_New( HCHUNK hChunk,
               WHMEM whParent,
               int left,
               int top,
               int right,
               int bottom,
               LPSTR text)
```

Description

Creates a button object at the specified location in the current application window.

Parameter	Description
hChunk	Handle to the memory section containing the object.
whParent	Handle to the parent object (symbol, group, or wizard) that will contain this object. 0 indicates there is no parent object.
left	Specifies the x-coordinate of the upper-left corner.
top	Specifies the y-coordinate of the upper-left corner.
right	Specifies the x-coordinate of the lower-right corner.
bottom	Specifies the y-coordinate of the lower-right corner.
text	Points to a null-terminated string containing the button's text.

Return Value

The return value is the handle of the object if the function is successful. Otherwise, it is *whNull*.

Comments

The function will fail if *buttonbold* is greater than MAX_BUTTON_STRINGLEN.

DisableLnk_New

```
WHMEM

DisableLnk_New(  HCHUNK hChunk,
                 WHMEM whObj,
                 LPSTR expression,
                 BOOL disableWhenTrue)
```

Description

Creates a disable link for the object specified.

Parameter	Description
hChunk	Handle to the memory section containing the object for which the link is being created.
whObj	Handle to the object for which the link is being created.
expression	Points to a null-terminated string containing the analog expression or tagname to use for the link.
disableWhenTrue	If this value is non-zero, the object's touch capability is disabled when the disable expression evaluates to a non-zero value. Otherwise, the object's touch capability is disabled when the disable expression evaluates to zero.

Return Value

The return value is the handle of the link if the function is successful. Otherwise, it is *whNull*.

Comments

None.

DiscAlarmLnk_New

```
WHMEM
DiscAlarmLnk_New(  HCHUNK hChunk,
                   WHMEM whObj,
                   WORD linkType,
                   LPSTR tagname,
                   LONG alarmColor,
                   LONG normalColor)
```

Description

Creates a discrete alarm link for the object specified.

Parameter	Description
hChunk	Handle to the memory section containing the object for which the link is being created.
whObj	Handle to the object for which the link is being created.
linkType	Specifies the flags that determine the link type. This parameter can be one of the following values:

Value	Meaning
LINE_LINK	Specifies a line color link based on alarm status.
TEXT_LINK	Specifies a text color link based on alarm status.
FILL_LINK	Specifies a fill color link based on alarm status.

tagname	Points to a null-terminated string containing the discrete tagname to use for the link.
alarmColor	Specifies the color when the tagname is in alarm state. Colors are specified in Windows standard RGB format.
normalColor	Specifies the color when the tagname is not in alarm state. Colors are specified in Windows standard RGB format.

Return Value

The return value is the handle of the link if the function is successful. Otherwise, it is *whNull*.

Comments

If a zero (0) is returned, check for invalid *tagname* or *linkType*.

DiscColorLnk_New

```
WHMEM

DiscColorLnk_New(  HCHUNK hChunk,
                   WHMEM whObj,
                   WORD linkType,
                   LPSTR expression,
                   LONG onColor,
                   LONG offColor)
```

Description

Creates a discrete fill, text, or line link for the object specified.

Parameter	Description
hChunk	Handle to the memory section containing the object for which the link is being created.
whObj	Handle to the object for which the link is being created.
linkType	Specifies the flags that determine the link type. This parameter can be one of the following values:

Value	Meaning
LINE_LINK	Specifies a line color link based on alarm status.
TEXT_LINK	Specifies a text color link based on alarm status.
FILL_LINK	Specifies a fill color link based on alarm status.

expression	Points to a null-terminated string containing the discrete expression or tagname to use for the link.
onColor	Specifies the color when the expression evaluates to "on" (non-zero). Colors are specified in Windows standard RGB format.
offColor	Specifies the color when the expression evaluates to "off" (zero). Colors are specified in Windows standard RGB format.

Return Value

The return value is the handle of the link if the function is successful. Otherwise, it is *whNull*.

Comments

If a zero (0) is returned, *expression* is or invalid, or *linkType* is invalid.

DiscInputLnk_New

```
WHMEM

DiscInputLnk_New(  HCHUNK hChunk,
                   WHMEM whObj,
                   LPSTR tagname,
                   LPSTR userMsg,
                   LPSTR onMsg,
                   LPSTR offMsg,
                   LPSTR setMsg,
                   LPSTR resetMsg,
                   BOOL bInputOnly,
                   BYTE cKeyFlags,
                   WORD wVirtKey)
```

Description

Creates a discrete input link for the object specified.

Parameter	Description
hChunk	Handle to the memory section containing the object for which the link is being created.
whObj	Handle to the object for which the link is being created.
tagname	Points to a null-terminated string containing the discrete tagname to use for the link.
userMsg	Points to a null-terminated string containing the message or instruction to display when the link for this object is activated.
onMsg	Points to a null-terminated string containing the message to display when the tagname has an "on" (non-zero) value. This message is displayed only for objects that have a text field.
offMsg	Points to a null-terminated string containing the message to display when the tagname has an "off" (zero) value. This message is displayed only for objects that have a text field.
setMsg	Points to a null-terminated string containing the label for the "Set" button that appears when this link is activated. The default label, "Set" will be used if this parameter is the empty string ("").
resetMsg	Points to a null-terminated string containing the label for the "Reset" button that appears when this link is activated. The default label, "Reset" will be used if this parameter is the empty string ("").

Parameter	Description
bInputOnly	Specifies this link as input only, the value entered will not be displayed, if this value is non-zero. This setting only applies to objects that have text display associated with them (for example, a push button).
cKeyFlags	Specifies the flags used when a keyboard key is assigned to this link. This parameter can be a combination of the following values:

Value	Meaning
TOUCH_KS_SHIFT	Specifies the SHIFT key must be held down in addition to the key specified.
TOUCH_KS_CTRL	Specifies the CTRL key must be held down in addition to the key specified.
0	Specifies that only the specified key must be held down.

wVirtKey	Specifies the keyboard key equivalent assigned to this link. This value is 0 if there is no keyboard equivalent.

Return Value

The return value is the handle of the link if the function is successful. Otherwise, it is *whNull*.

Comments

If a zero (0) is returned, *tagname* is NULL, too long or invalid *userMsg*, *onMsg*, *offMsg*, *setMsg* is NULL or too long.

DiscOutputLnk_New

```
WHMEM
DiscOutputLnk_New( HCHUNK hChunk,
                   WHMEM whObj,
                   LPSTR expression,
                   LPSTR onMsg,
                   LPSTR offMsg)
```

Description Creates a discrete output link for the object specified.

Parameter	Description
hChunk	Handle to the memory section containing the object for which the link is being created.
whObj	Handle to the object for which the link is being created.
expression	Points to a null-terminated string containing the expression or tagname to use for the link.
onMsg	Points to a null-terminated string containing the message to display when the tagname has an "on" (non-zero) value. This message is displayed only for objects that have a text field.
offMsg	Points to a null-terminated string containing the message to display when the tagname has an "off" (zero) value. This message is displayed only for objects that have a text field.

Return Value The return value is the handle of the link if the function is successful. Otherwise, it is *whNull*.

Comments If a zero (0) is returned, *expression* is NULL, too long or invalid *onMsg*, *offMsg*, is NULL or too long.

DiscTag_GetInfo

```
int
DiscTag_GetInfo( DBHND dbHnd,
                 LP_DISCTAGINFO lpDiscInfo)
```

Description Returns the discrete tagname information for the database tagname with the given handle.

Parameter	Description
dbHnd	Handle to the database tagname.
lpDiscInfo	Pointer to the discrete tagname information structure.

Return Value Error code.

Comments None.

DiscTag_SetInfo

```
int
DiscTag_SetInfo(  DBHND dbHnd,
                  LP_DISCTAGINFO lpDiscInfo)
```

Description Sets the discrete tagname information for a database tagname with the given handle.

Parameter	Description
dbHnd	Handle to the database tagname.
lpDiscInfo	Pointer to the discrete tagname information structure.

Return Value Error code.

Comments None.

DiscTouchLnk_New

```
WHMEM
DiscTouchLnk_New(  HCHUNK hChunk,
                   WHMEM whObj,
                   LPSTR tagname,
                   WORD actionType,
                   BYTE cKeyFlags,
                   WORD wVirtKey)
```

Description Creates a discrete touch link for the object specified.

Parameter	Description
hChunk	Handle to the memory section containing the object for which the link is being created.
whObj	Handle to the object for which the link is being created.
tagname	Points to a null-terminated string containing the discrete tagname to use for the link.
actionType	Specifies the flags that determine the action type for the touch link. This parameter can be one of the following values:

Value	Meaning
ACTION_DIRECT	Specifies the direct action that will set the value equal to 1 when the button is pressed and held down. The value automatically resets to 0 when the button is released.

Value	Meaning
ACTION_REVERSE	Specifies the reverse action that will set the value equal to 0 when the button is pressed and held down. The value automatically resets to 1 when the button is released.
ACTION_TOGGLE	Specifies the toggle action that will set the value to 1 if it is currently 0 and 0 if the value is currently 1 when the button is pressed.
ACTION_RESET	Specifies the reset action that will set the value to 0 when the button is pressed.
ACTION_SET	Specifies the set action that will set the value to 1 when the button is pressed.

cKeyFlags Specifies the flags used when a keyboard key is assigned to this link. This parameter can be a combination of the following values:

Value	Meaning
TOUCH_KS_SHIFT	Specifies the SHIFT key must be held down in addition to the key specified.
TOUCH_KS_CTRL	Specifies the CTRL key must be held down in addition to the key specified.
0	Specifies that only the specified key must be held down.

Parameter	Description
wVirtKey	Specifies the keyboard key equivalent assigned to this link. This value is 0 if there is no keyboard equivalent.

Return Value The return value is the handle of the link if the function is successful. Otherwise, it is *whNull*.

Comments If a zero (0) is returned, *tagname* is NULL or too long; or *actionType* is invalid.

DllObj_New

```
WHMEM
DllObj_New( HCHUNK hChunk,
            WHMEM whParent,
            int left,
            int top,
            int right,
            int bottom,
            LPSTR dllName,
            DWORD dllData)
```

Description Creates a DLL object at the specified location in the current application window. Currently available DLL objects include the InTouch SPC pareto, histogram, and control chart objects.

Parameter	Description
hChunk	Handle to the memory section containing the object.
whParent	Handle to the parent object (symbol, group, or wizard) that will contain this object. 0 indicates there is no parent object.
left	Specifies the x-coordinate of the upper-left corner.
top	Specifies the y-coordinate of the upper-left corner.
right	Specifies the x-coordinate of the lower-right corner.
bottom	Specifies the y-coordinate of the lower-right corner.
dllData	Specifies data unique to each DLL that initializes the object.
dllName	Points to a null-terminated string containing the name of the DLL capable of creating the object. "SPCDLL":

Value	Meaning
SPC_CONTROL	Create an SPC control chart.
SPC_HISTOGRAM	Create an SPC histogram chart.
SPC_PARETO	Create an SPC pareto chart.

Return Value The return value is the handle of the object if the function is successful. Otherwise, it is *whNull*.

Comments None.

EllipseObj_New

```
WHMEM

EllipseObj_New(  HCHUNK hChunk,
                 WHMEM whParent,
                 int left,
                 int top,
                 int right,
                 int bottom)
```

Description

Creates an ellipse object at the specified location in the current application window.

Parameter	Description
hChunk	Handle to the memory section containing the object.
whParent	Handle to the parent object (symbol, group, or wizard) that will contain this object. 0 indicates there is no parent object.
left	Specifies the x-coordinate of the upper-left corner.
top	Specifies the y-coordinate of the upper-left corner.
right	Specifies the x-coordinate of the lower-right corner.
bottom	Specifies the y-coordinate of the lower-right corner.

Return Value

The return value is the handle of the object if the function is successful. Otherwise, it is *whNull*.

Comments

None.

Font_Scale

```
VOID

Font_Scale( HWND hWnd,
            LPLOGFONT lFnt,
            LPRECT old,
            LPRECT new,
            int len,
            LPSTR text)
```

Description

Linearly scales the logical font supplied using the old and new rectangles and the string specified.

Parameter	Description
hWnd	NULL must be passed for this parameter.
lFnt	Points to a LOGFONT structure that defines the characteristics of the logical font used to display the text in the rectangle specified by the old parameter. The scaled font will be returned in this parameter. LOGFONT is a Windows structure.
old	Points to a RECT structure that defines the original rectangle that was used to display the text specified by the text parameter.
new	Points to a RECT structure that defines the new rectangle that will be used to display the text specified by the text parameter.
len	Specifies the number of bytes in the string.
text	Points to the character string used to scale the logical font within the rectangle specified.

Return Value

None.

Comments

The old and new RECT structures set the proportion to change the text.

GroupObj_New

```
WHMEM
GroupObj_New( HCHUNK hChunk,
              WHMEM whParent,
              int left,
              int top,
              int right,
              int bottom)
```

Description

Creates a group (cell) object at the specified location in the current application window. Populate the group with other objects by using this object's handle as the parent handle.

Parameter	Description
hChunk	Handle to the memory section containing the object.
whParent	Handle to the parent object (symbol, group, or wizard) that will contain this object. 0 indicates there is no parent object.
left	Specifies the x-coordinate of the upper-left corner.
top	Specifies the y-coordinate of the upper-left corner.
right	Specifies the x-coordinate of the lower-right corner.
bottom	Specifies the y-coordinate of the lower-right corner.

Return Value

The return value is the handle of the object if the function is successful. Otherwise, it is *whNull*.

Comments

None.

HistTrendObj_New

```
WHMEM

HistTrendObj_New(  HCHUNK hChunk,
                   WHMEM whParent,
                   int left,
                   int top,
                   int right,
                   int bottom,
                   LPSTR tagname,
                   LONG chartColor,
                   LONG borderColor,
                   WORD spanUnits,
                   DWORD spanTime,
                   WORD displayMode,
                   WORD options)
```

Description

Creates a historical trend object at the specified location in the current application window.

Parameter	Description
hChunk	Handle to the memory section containing the object.
whParent	Handle to the parent object (symbol, group, or wizard) that will contain this object. 0 indicates there is no parent object.
left	Specifies the x-coordinate of the upper-left corner.
top	Specifies the y-coordinate of the upper-left corner.
right	Specifies the x-coordinate of the lower-right corner.
bottom	Specifies the y-coordinate of the lower-right corner.
tagname	Points to a null-terminated string containing a historical trend tagname.
chartColor	Specifies the color of the chart's background. Colors are specified in Windows standard RGB format.
borderColor	Specifies the color of the chart's border. Colors are specified in Windows standard RGB format.
spanUnits	Specifies the flags that determine the measurement units for the spanTime parameter. This parameter can be one of the following values:

Value	Meaning
TIME_SEC	Specifies units as seconds.
TIME_MIN	Specifies units as minutes.

Value	Meaning
TIME_HR	Specifies units as hours.
TIME_DAY	Specifies units as days.

spanTime Specifies the historical trend object's time span in units specified by the spanUnits parameter. This parameter can be a value from 1 to 999.

displayMode Specifies the flags that determine the initial display mode. This parameter can be one of the following values:

Value	Meaning
HTREND_MODE_AVE	Specifies average value data display.
HTREND_MODE_MINMAX	Specifies minimum and maximum data display.

options Specifies the flags that determine the trend options. This parameter can be a combination of the following values:

Value	Meaning
TREND_RT_CHANGES	Specifies that run-time changes to the historical trend are allowed.

Return Value The return value is the handle of the object if the function is successful. Otherwise, it is *whNull*.

Comments Possible error conditions include, invalid *spanUnits*, *spanTime*, *displayMode*, *tagname* (not a historical tagname type).

Defaults are taken for colors, labels, division information and pen colors.

If the specified tagname does not exist, one will be created.

LineObj_New

```
WHMEM

LineObj_New( HCHUNK hChunk,
             WHMEM whParent,
             int x1,
             int y1,
             int x2,
             int y2)
```

Description

Creates a line object at the specified location in the current application window.

Parameter	Description
hChunk	Handle to the memory section containing the object.
whParent	Handle to the parent object (symbol, group, or wizard) that will contain this object. 0 indicates there is no parent object.
x1	Specifies the x-coordinate of the beginning of the line.
y1	Specifies the y-coordinate of the beginning of the line.
x2	Specifies the x-coordinate of the end of the line.
y2	Specifies the y-coordinate of the end of the line.

Return Value

The return value is the handle of the object if the function is successful. Otherwise, it is *whNull*.

Comments

None.

LocationLnk_New

```
WHMEM

LocationLnk_New( HCHUNK hChunk,
                 WHMEM whObj,
                 WORD linkType,
                 WORD referenceType,
                 LPSTR expression,
                 REAL minValue,
                 REAL maxValue,
                 int minPosition,
                 int maxPosition)
```

Description Creates a horizontal or vertical location link for the object specified.

Parameter	Description
hChunk	Handle to the memory section containing the object for which the link is being created.
whObj	Handle to the object for which the link is being created.
linkType	Specifies the flags that determine the link type. This parameter can be one of the following values:

Value	Meaning
VERT_LINK	Specifies a vertical link.
HORIZ_LINK	Specifies a horizontal link.

referenceType Specifies the flags that determine the reference type. This parameter can be one of the following values:

Value	Meaning
LINK_LEFT	Specifies that the object will be moved using the object's left side as its origin. This value is valid for a HORIZ_LINK.
LINK_MIDDLE	Specifies that the object will be moved using the object's horizontal midpoint as its origin. This value is valid for a HORIZ_LINK.

Value	Meaning
LINK_RIGHT	Specifies that the object will be moved using the object's right side as its origin. This value is valid for a HORIZ_LINK.
LINK_TOP	Specifies that the object will be moved using the object's top as its origin. This value is valid for a VERT_LINK.
LINK_CENTER	Specifies that the object will be moved using the object's vertical midpoint as its origin. This value is valid for a VERT_LINK.
LINK_BOTTOM	Specifies that the object will be moved using the object's bottom as its origin. This value is valid for a VERT_LINK.

expression Points to a null-terminated string containing the analog expression or tagname to use for the link.

minValue Specifies the value when the object is at its highest (VERT_LINK) or leftmost (HORIZ_LINK) position.

maxValue Specifies the value when the object is at its lowest (VERT_LINK) or rightmost (HORIZ_LINK) position.

minPosition Specifies the number of pixels the object will move up (VERT_LINK) or left (HORIZ_LINK) from its current position in relation to the values specified for the minValue and maxValue parameters.

Parameter	Description
maxPosition	Specifies the number of pixels the object will move down (VERT_LINK) or right (HORIZ_LINK) from its current position in relation to the values specified for the minValue and maxValue parameters.

Return Value The return value is the handle of the link if the function is successful. Otherwise, it is *whNull*.

Comments If a zero (0) is returned, *expression* is NULL, too long or invalid or, *linkType* or *referenceType* is incorrect.

Obj_Delete

```
BOOL
Obj_Delete( HCHUNK hChunk,
            WHMEM whObj)
```

Description Deletes the specified window object.

Parameter	Description
hChunk	Handle to the memory section containing the object.
whObj	Handle to the object to delete.

Return Value The return value is TRUE if the function is successful. Otherwise, it is FALSE.

Comments None.

OrientationLnk_New

```
WHMEM

OrientationLnk_New( HCHUNK hChunk,
                    WHMEM whObj,
                    LPSTR expression,
                    REAL CCWmaxValue,
                    REAL CWmaxValue,
                    REAL CCWmaxDegrees,
                    REAL CWmaxDegrees,
                    REAL xOffset,
                    REAL yOffset)
```

Description

Creates an orientation link that defines the specified object's angle of rotation.

Parameter	Description
hChunk	Handle to the memory section containing the object for which the link is being created.
whObj	Handle to the object for which the link is being created.
expression	Points to a null-terminated string containing the analog expression or tagname to use for the link.
CCWmaxValue	Specifies the value when the object is rotated to its maximum counter-clockwise position.
CWmaxValue	Specifies the value when the object is rotated to its maximum clockwise position.
CCWmaxDegrees	Specifies the maximum number of degrees the object will rotate counter-clockwise (starting at 12:00) from its current position in relation to the values specified for the *CCWmaxValue* and *CWmaxValue* parameters.
CWmaxDegrees	Specifies the maximum number of degrees the object will rotate clockwise (starting at 12:00) from its current position in relation to the values specified for the *CCWmaxValue* and *CWmaxValue* parameters.
xOffset	Specifies the number of pixels the object's rotation centerpoint is moved horizontally from the centerpoint of the object (positive values are right).
yOffset	Specifies the number of pixels the object's rotation centerpoint is moved vertically from the centerpoint of the object (positive values are down).

Return Value

The return value is the handle of the link if the function is successful. Otherwise, it is *whNull*.

Comments

If a zero (0) is returned, *expression* is NULL, too long or invalid.

PctFillLnk_New

```
WHMEM

PctFillLnk_New( HCHUNK hChunk,
                WHMEM whObj,
                WORD linkType,
                WORD directionType,
                LPSTR expression,
                REAL minValue,
                REAL maxValue,
                int minPercent,
                int maxPercent,
                LONG fillColor)
```

Description

Creates a horizontal or vertical percent fill link for the object specified.

Parameter	Description
hChunk	Handle to the memory section containing the object for which the link is being created.
whObj	Handle to the object for which the link is being created.
linkType	Specifies the flags that determine the link type. This parameter can be one of the following values:

Value	Meaning
VERT_LINK	Specifies a vertical link.
HORIZ_LINK	Specifies a horizontal link.
directionType	Specifies the flags that determine the direction type for the fill's origin. This parameter can be one of the following values:
LINK_LEFT	Specifies that the object will be filled from the object's left side. This value is valid for a HORIZ_LINK.
LINK_MIDDLE	Specifies that the object will be filled from the object's horizontal midpoint. This value is valid for a HORIZ_LINK.

Value	Meaning
LINK_RIGHT	Specifies that the object will be filled from the object's right side. This value is valid for a HORIZ_LINK.
LINK_TOP	Specifies that the object will be filled from the object's top. This value is valid for a VERT_LINK.
LINK_CENTER	Specifies that the object will be filled from the object's vertical midpoint. This value is valid for a VERT_LINK.
LINK_BOTTOM	Specifies that the object will be filled from the object's bottom. This value is valid for a VERT_LINK.

expression Points to a null-terminated string containing the analog expression or tagname to use for the link.

minValue Specifies the value when the fill reaches its smallest amount.

maxValue Specifies the value when the fill reaches its largest amount.

minPercent Specifies the percentage, from 0 to 100, of the object's defined width (HORIZ_LINK) or height (VERT_LINK) that the object will filled when the expression is at the minValue.

maxPercent Specifies the percentage, from 0 to 100, of the object's defined width (HORIZ_LINK) or height (VERT_LINK) that the object will filled when the expression is at the maxValue.

fillColor Specifies the fill color. Colors are specified in Windows standard RGB format.

Return Value

The return value is the handle of the link if the function is successful. Otherwise, it is *whNull*.

Comments

If a zero (0) is returned, expression is NULL, too long or invalid or, *linkType*, *directionType* are invalid.

Point_Scale

```
VOID

Point_Scale( LPPOINT dest,
             LPRECT old,
             LPRECT new,
             int flags)
```

Description

Linearly scales the point supplied using the old and new rectangles specified.

Parameter	Description
dest	Points to a POINT structure that defines the original point that was based upon the original rectangle. The scaled point will be returned in this parameter.
old	Points to a RECT structure that defines the original rectangle.
new	Points to a RECT structure that defines the new rectangle.
flags	Specifies the flags that determine the scaling mode. This parameter can be a combination of the following values:

Value	Meaning
0	SCALE_X1 | SCALE_Y1. Specifies scaling of all coordinates. The coordinates are absolute.
SCALE_X1	Specifies linear scaling of the x-coordinate.
LOFFSET_X1	Specifies the x-coordinate should retain the same offset from the left edge of the rectangle.
ROFFSET_X1	Specifies the x-coordinate should retain the same offset from the right edge of the rectangle.
OFFSET_X1	The same as LOFFSET_X1.

Value	Meaning
SCALE_Y1	Specifies linear scaling of the y-coordinate.
TOFFSET_Y1	Specifies the y-coordinate should retain the same offset from the top edge of the rectangle.
BOFFSET_Y1	Specifies the y-coordinate should retain the same offset from the bottom edge of the rectangle.
OFFSET_Y1	The same as LOFFSET_Y1.
SCALE_REL	Specifies that the coordinates are relative to the origin of the rectangle. If this flag is not set then the coordinates are assumed to be absolute.

Return Value None.

Comments If the scaling mode SCALE_REL is set then the coordinates will be unaffected when SCALE_X1, or SCALE_Y1 are used. The _X1 flags cannot be combined. Likewise for the _Y1 flags.

PointArray_Scale

```
VOID
PointArray_Scale( int nPts,
                  LPPOINT dest,
                  LPRECT old,
                  LPRECT new,
                  int flags)
```

Description

Linearly scales the array of points supplied using the old and new rectangles specified.

Parameter	Description
nPts	Specifies the number of points in the array.
dest	Points to an array of POINT structures. Each structure in the array specifies a point that defines an original point that was based upon the original rectangle. Each point in the array will be scaled and returned in this parameter.
old	Points to a RECT structure that defines the original rectangle.
new	Points to a RECT structure that defines the new rectangle.
flags	Specifies the flags that determine the scaling mode. This parameter can be a combination of the following values:

Value	Meaning
0	SCALE_X1 | SCALE_Y1. Specifies scaling of all coordinates. The coordinates are absolute.
SCALE_X1	Specifies linear scaling of the x-coordinate.
LOFFSET_X1	Specifies the x-coordinate should retain the same offset from the left edge of the rectangle.
ROFFSET_X1	Specifies the x-coordinate should retain the same offset from the right edge of the rectangle.

Value	Meaning
OFFSET_X1	The same as LOFFSET_X1.
SCALE_Y1	Specifies linear scaling of the y-coordinate.
TOFFSET_Y1	Specifies the y-coordinate should retain the same offset from the top edge of the rectangle.
BOFFSET_Y1	Specifies the y-coordinate should retain the same offset from the bottom edge of the rectangle.
OFFSET_Y1	The same as LOFFSET_Y1.
SCALE_REL	Specifies that the coordinates are relative to the origin of the rectangle. If this flag is not set then the coordinates are assumed to be absolute.

Return Value None.

Comments If the scaling mode SCALE_REL is set then the coordinates will be unaffected when SCALE_X1, or SCALE_Y1 are used. The _X1 flags cannot be combined. Likewise for the _Y1 flags.

PointReal_Scale

```
VOID
PointReal_Scale( LPPOINTREAL dest,
                 LPRECT old,
                 LPRECT new,
                 int flags)
```

Description

Linearly scales the point, with REAL coordinates, using the old and new rectangles specified.

Parameter	Description
dest	Points to a POINTREAL structure that defines the original point that was based upon the original rectangle. The scaled point will be returned.
old	Points to a RECT structure that defines the original rectangle.
new	Points to a RECT structure that defines the new rectangle.
flags	Specifies the flags that determine the scaling mode. Can be a combination of the following values:

Value	Meaning
0	SCALE_X1 | SCALE_Y1. Specifies scaling of all coordinates. The coordinates are absolute.
SCALE_X1	Specifies linear scaling of the x-coordinate.
LOFFSET_X1	Specifies the x-coordinate should retain the same offset from the left edge of the rectangle.
ROFFSET_X1	Specifies the x-coordinate should retain the same offset from the right edge of the rectangle.

Value	Meaning
OFFSET_X1	The same as LOFFSET_X1.
SCALE_Y1	Specifies linear scaling of the y-coordinate.
TOFFSET_Y1	Specifies the y-coordinate should retain the same offset from the top edge of the rectangle.
BOFFSET_Y1	Specifies the y-coordinate should retain the same offset from the bottom edge of the rectangle.
OFFSET_Y1	The same as LOFFSET_Y1.
SCALE_REL	Specifies that the coordinates are relative to the origin of the rectangle. If this flag is not set then the coordinates are assumed to be absolute.

Return Value None.

Comments If the scaling mode SCALE_REL is set then the coordinates will be unaffected when SCALE_X1, or SCALE_Y1 are used. The _X1 flags cannot be combined. Likewise for the _Y1 flags.

PointRealArray_Scale

```
VOID
PointRealArray_Scale( int nPts,
                      LPPOINTREAL dest,
                      LPRECT old,
                      LPRECT new,
                      int flags)
```

Description

Linearly scales the array of points, with REAL coordinates, using the old and new rectangles specified.

Parameter	Description
nPts	Specifies the number of points in the array.
dest	Points to an array of POINTREAL structures. Each structure in the array specifies a point that defines an original point that was based upon the original rectangle. Each point in the array will be scaled and returned in this parameter.
old	Points to a RECT structure that defines the original rectangle.
new	Points to a RECT structure that defines the new rectangle.
flags	Specifies the flags that determine the scaling mode. This parameter can be a combination of the following values:

Value	Meaning
0	SCALE_X1 | SCALE_Y1. Specifies scaling of all coordinates. The coordinates are absolute.
SCALE_X1	Specifies linear scaling of the x-coordinate.
LOFFSET_X1	Specifies the x-coordinate should retain the same offset from the left edge of the rectangle.
ROFFSET_X1	Specifies the x-coordinate should retain the same offset from the right edge of the rectangle.

Value	Meaning
OFFSET_X1	The same as LOFFSET_X1.
SCALE_Y1	Specifies linear scaling of the y-coordinate.
TOFFSET_Y1	Specifies the y-coordinate should retain the same offset from the top edge of the rectangle.
BOFFSET_Y1	Specifies the y-coordinate should retain the same offset from the bottom edge of the rectangle.
OFFSET_Y1	The same as LOFFSET_Y1.
SCALE_REL	Specifies that the coordinates are relative to the origin of the rectangle. If this flag is not set then the coordinates are assumed to be absolute.

Return Value None.

Comments If the scaling mode SCALE_REL is set then the coordinates will be unaffected when SCALE_X1, or SCALE_Y1 are used. The _X1 flags cannot be combined. Likewise for the _Y1 flags.

PolygonObj_New

```
WHMEM
PolygonObj_New( HCHUNK hChunk,
                WHMEM whParent,
                int nPts,
                LPPOINT lpPoints)
```

Description Creates a polygon object at the specified location in the current application window.

Parameter	Description
hChunk	Handle to the memory section containing the object.
whParent	Handle to the parent object (symbol, group, or wizard) that will contain this object. 0 indicates there is no parent object.
nPts	Specifies the number of points in the array. This value must be at least 2.
lpPoints	Points to an array of POINT structures. Each structure in the array specifies a point.

Return Value The return value is the handle of the object if the function is successful. Otherwise, it is *whNull*.

Comments None.

PolylineObj_New

```
WHMEM
PolylineObj_New( HCHUNK hChunk,
                 WHMEM whParent,
                 int nPts,
                 LPPOINT lpPoints)
```

Description Creates a polyline object at the specified location in the current application window.

Parameter	Description
hChunk	Handle to the memory section containing the object.
whParent	Handle to the parent object (symbol, group, or wizard) that will contain this object. 0 indicates there is no parent object.
nPts	Specifies the number of points in the array. This value must be at least 2.
lpPoints	Points to an array of POINT structures. Each structure in the array specifies a point.

Return Value The return value is the handle of the object if the function is successful. Otherwise, it is *whNull*.

Comments None.

RealTrendObj_New

```
WHMEM

RealTrendObj_New( HCHUNK hChunk,
                  WHMEM whParent,
                  int left,
                  int top,
                  int right,
                  int bottom,
                  LPSTR comment,
                  LONG chartColor,
                  LONG borderColor,
                  WORD sampleUnits,
                  DWORD sampleTime,
                  DWORD samples,
                  WORD options)
```

Description

Creates a real time trend object at the specified location in the current application window.

Parameter	Description
hChunk	Handle to the memory section containing the object.
whParent	Handle to the parent object (symbol, group, or wizard) that will contain this object. 0 indicates there is no parent object.
left	Specifies the x-coordinate of the upper-left corner.
top	Specifies the y-coordinate of the upper-left corner.
right	Specifies the x-coordinate of the lower-right corner.
bottom	Specifies the y-coordinate of the lower-right corner.
comment	Points to a null-terminated string containing a comment for the real time trend object.
chartColor	Specifies the color of the chart's background. Colors are specified in Windows standard RGB format.
borderColor	Specifies the color of the chart's border. Colors are specified in Windows standard RGB format.
sampleUnits	Specifies the flags that determine the measurement units for the *sampleTime* parameter. This parameter can be one of the following values:

Value	Meaning
TIME_MSEC	Specifies units as milliseconds.
TIME_SEC	Specifies units as seconds.

Value	Meaning
TIME_MIN	Specifies units as minutes.
TIME_HR	Specifies units as hours.

sampleTime — Specifies the time between samples in units specified by the *sampleUnits* parameter. This parameter can be a value from 1 to 999.

samples — Specifies the number of samples displayed in the chart. This value must be at least 2 and no greater than 1024.

options — Specifies the flags that determine the trend options. This parameter can be a combination of the following values:

Value	Meaning
TREND_MEM_UPDATE	Specifies trend updates only when the trend is in memory.

Return Value

The return value is the handle of the object if the function is successful. Otherwise, it is *whNull*.

Comments

If a zero (0) is returned, invalid *sampleUnits* (less than 2 or greater than 1024).

Defaults are used for labels, colors and pens.

Rect_Scale

```
VOID

Rect_Scale( LPRECT dest,
            LPRECT old,
            LPRECT new,
            int flags)
```

Description

Linearly scales the rectangle supplied using the old and new rectangles specified.

Parameter	Description
dest	Points to a RECT structure that defines the original rectangle that was based upon the original rectangle. The scaled rectangle will be returned in this parameter.
old	Points to a RECT structure that defines the original rectangle.
new	Points to a RECT structure that defines the new rectangle.
flags	Specifies the flags that determine the scaling mode. This parameter can be a combination of the following values:

Value	Meaning
0	SCALE_X1 | SCALE_X2 | SCALE_Y1 | SCALE_Y2. Specifies scaling of all coordinates. The coordinates are absolute.
SCALE_X1	Specifies linear scaling of the x-coordinate of the top-left corner.
LOFFSET_X1	Specifies the x-coordinate of the top-left corner should retain the same offset from the left edge of the rectangle.
ROFFSET_X1	Specifies the x-coordinate of the top-left corner should retain the same offset from the right edge of the rectangle.

Value	Meaning
OFFSET_X1	The same as LOFFSET_X1.
SCALE_X2	Specifies linear scaling of the x-coordinate of the lower-right corner.
LOFFSET_X2	Specifies the x-coordinate of the lower-right corner should retain the same offset from the left edge of the rectangle.
ROFFSET_X2	Specifies the x-coordinate of the lower-right corner should retain the same offset from the right edge of the rectangle.
OFFSET_X2	The same as ROFFSET_X2.
SCALE_Y1	Specifies linear scaling of the y-coordinate of the top-left corner.
TOFFSET_Y1	Specifies the y-coordinate of the top-left corner should retain the same offset from the top edge of the rectangle.
BOFFSET_Y1	Specifies the y-coordinate of the top-left corner should retain the same offset from the bottom edge of the rectangle.

Value	Meaning
OFFSET_Y1	The same as LOFFSET_Y1.
SCALE_Y2	Specifies linear scaling of the y-coordinate of the lower-right corner.
TOFFSET_Y2	Specifies the y-coordinate of the lower-right corner should retain the same offset from the top edge of the rectangle.
BOFFSET_Y2	Specifies the y-coordinate of the lower-right corner should retain the same offset from the bottom edge of the rectangle.
OFFSET_Y2	The same as BOFFSET_Y2.
SCALE_REL	Specifies that the coordinates are relative to the origin of the rectangle. If this flag is not set then the coordinates are assumed to be absolute.

Return Value None.

Comments If the scaling mode SCALE_REL is set then the coordinates will be unaffected when SCALE_X1, SCALE_X2, SCALE_Y1,or SCALE_Y2 are used. The _X1 flags cannot be combined. Likewise for the _X2, _Y1, and _Y2 flags.

RectangleObj_New

```
WHMEM

RectangleObj_New( HCHUNK hChunk,
                  WHMEM whParent,
                  int left,
                  int top,
                  int right,
                  int bottom)
```

Description

Creates a rectangle object at the specified location in the current application window.

Parameter	Description
hChunk	Handle to the memory section containing the object.
whParent	Handle to the parent object (symbol, group, or wizard) that will contain this object. 0 indicates there is no parent object.
left	Specifies the x-coordinate of the upper-left corner.
top	Specifies the y-coordinate of the upper-left corner.
right	Specifies the x-coordinate of the lower-right corner.
bottom	Specifies the y-coordinate of the lower-right corner.

Return Value

The return value is the handle of the object if the function is successful. Otherwise, it is *whNull*.

Comments

None.

RectReal_Scale

```
VOID

RectReal_Scale( LPRECTREAL dest,
                LPRECT old,
                LPRECT new,
                int flags)
```

Description

Linearly scales the rectangle, with REAL coordinates, using the old and new rectangles specified.

Parameter	Description
dest	Points to a RECTREAL structure that defines the original rectangle that was based upon the original rectangle. The scaled rectangle will be returned in this parameter.
old	Points to a RECT structure that defines the original rectangle.
new	Points to a RECT structure that defines the new rectangle.
flags	Specifies the flags that determine the scaling mode. This parameter can be a combination of the following values:

Value	Meaning
0	SCALE_X1 | SCALE_X2 | SCALE_Y1 | SCALE_Y2. Specifies scaling of all coordinates. The coordinates are absolute.
SCALE_X1	Specifies linear scaling of the x-coordinate of the top-left corner.
LOFFSET_X1	Specifies the x-coordinate of the top-left corner should retain the same offset from the left edge of the rectangle.
ROFFSET_X1	Specifies the x-coordinate of the top-left corner should retain the same offset from the right edge of the rectangle.

Value	Meaning
OFFSET_X1	The same as LOFFSET_X1.
SCALE_X2	Specifies linear scaling of the x-coordinate of the lower-right corner.
LOFFSET_X2	Specifies the x-coordinate of the lower-right corner should retain the same offset from the left edge of the rectangle.
ROFFSET_X2	Specifies the x-coordinate of the lower-right corner should retain the same offset from the right edge of the rectangle.
OFFSET_X2	The same as ROFFSET_X2.
SCALE_Y1	Specifies linear scaling of the y-coordinate of the top-left corner.
TOFFSET_Y1	Specifies the y-coordinate of the top-left corner should retain the same offset from the top edge of the rectangle.
BOFFSET_Y1	Specifies the y-coordinate of the top-left corner should retain the same offset from the bottom edge of the rectangle.
OFFSET_Y1	The same as LOFFSET_Y1.

Value	Meaning
SCALE_Y2	Specifies linear scaling of the y-coordinate of the lower-right corner.
TOFFSET_Y2	Specifies the y-coordinate of the lower-right corner should retain the same offset from the top edge of the rectangle.
BOFFSET_Y2	Specifies the y-coordinate of the lower-right corner should retain the same offset from the bottom edge of the rectangle.
OFFSET_Y2	The same as BOFFSET_Y2.
SCALE_REL	Specifies that the coordinates are relative to the origin of the rectangle. If this flag is not set then the coordinates are assumed to be absolute.

Return Value None.

Comments If the scaling mode SCALE_REL is set then the coordinates will be unaffected when SCALE_X1, SCALE_X2, SCALE_Y1,or SCALE_Y2 are used. The _X1 flags cannot be combined. Likewise for the _X2, _Y1, and _Y2 flags.

RRectangleObj_New

```
WHMEM

RRectangleObj_New( HCHUNK hChunk,
                   WHMEM whParent,
                   int left,
                   int top,
                   int right,
                   int bottom,
                   int rrWidth,
                   int rrHeight)
```

Description

Creates a rounded corner rectangle object at the specified location in the current application window.

Parameter	Description
hChunk	Handle to the memory section containing the object.
whParent	Handle to the parent object (symbol, group, or wizard) that will contain this object. 0 indicates there is no parent object.
left	Specifies the x-coordinate of the upper-left corner.
top	Specifies the y-coordinate of the upper-left corner.
right	Specifies the x-coordinate of the lower-right corner.
bottom	Specifies the y-coordinate of the lower-right corner.
rrWidth	Specifies the width of the ellipse used to draw the rounded corners.
rrHeight	Specifies the height of the ellipse used to draw the rounded corners.

Return Value

The return value is the handle of the object if the function is successful. Otherwise, it is *whNull*.

Comments

None.

SizeLnk_New

```
WHMEM

SizeLnk_New( HCHUNK hChunk,
             WHMEM whObj,
             WORD linkType,
             WORD anchorType,
             LPSTR expression,
             REAL minValue,
             REAL maxValue,
             int minPercent,
             int maxPercent)
```

Description

Creates a horizontal or vertical size link for the object specified.

Parameter	Description
hChunk	Handle to the memory section containing the object for which the link is being created.
whObj	Handle to the object for which the link is being created.
linkType	Specifies the flags that determine the link type. This parameter can be one of the following values:

Value	Meaning
VERT_LINK	Specifies a vertical link.
HORIZ_LINK	Specifies a horizontal link.

anchorType — Specifies the flags that determine the anchor type. This parameter can be one of the following values:

Value	Meaning
LINK_LEFT	Specifies that the object will be anchored at the object's left side. This value is valid for a HORIZ_LINK.
LINK_MIDDLE	Specifies that the object will be anchored at the object's horizontal midpoint. This value is valid for a HORIZ_LINK.
LINK_RIGHT	Specifies that the object will be anchored at the object's right side. This value is valid for a HORIZ_LINK.

Value	Meaning
LINK_TOP	Specifies that the object will be anchored at the object's top. This value is valid for a VERT_LINK.
LINK_CENTER	Specifies that the object will be anchored at the object's vertical midpoint. This value is valid for a VERT_LINK.
LINK_BOTTOM	Specifies that the object will be anchored at the object's bottom. This value is valid for a VERT_LINK.

expression Points to a null-terminated string containing the analog expression or tagname to use for the link.

minValue Specifies the value when the object is its smallest size.

maxValue Specifies the value when the object is its largest size.

minPercent Specifies the percentage, from 0 to 100, of the object's defined width (HORIZ_LINK) or height (VERT_LINK) that the object will be when the expression is at the minValue.

maxPercent Specifies the percentage, from 0 to 100, of the object's defined width (HORIZ_LINK) or height (VERT_LINK) that the object will be when the expression is at the maxValue.

Return Value

The return value is the handle of the link if the function is successful. Otherwise, it is *whNull*.

Comments

If a zero (0) is returned, *expression* is NULL, too long or invalid *linkType*, invalid *anchorType*.

SliderLnk_New

```
WHMEM

SliderLnk_New( HCHUNK hChunk,
               WHMEM whObj,
               WORD linkType,
               WORD referenceType,
               LPSTR tagname,
               REAL minValue,
               REAL maxValue,
               int minPosition,
               int maxPosition)
```

Description

Creates a horizontal or vertical slider touch link for the object specified.

Parameter	Description
hChunk	Handle to the memory section containing the object for which the link is being created.
whObj	Handle to the object for which the link is being created.
linkType	Specifies the flags that determine the link type. This parameter can be one of the following values:

Value	Meaning
VERT_LINK	Specifies a vertical link.
HORIZ_LINK	Specifies a horizontal link.

referenceType	Specifies the flags that determine the reference type. This parameter can be one of the following values:

Value	Meaning
LINK_LEFT	Specifies that the object will be moved using the object's left side as its origin. This value is valid for a HORIZ_LINK.
LINK_MIDDLE	Specifies that the object will be moved using the object's horizontal midpoint as its origin. This value is valid for a HORIZ_LINK.

Value	Meaning
LINK_RIGHT	Specifies that the object will be moved using the object's right side as its origin. This value is valid for a HORIZ_LINK.
LINK_TOP	Specifies that the object will be moved using the object's top as its origin. This value is valid for a VERT_LINK.
LINK_CENTER	Specifies that the object will be moved using the object's vertical midpoint as its origin. This value is valid for a VERT_LINK.
LINK_BOTTOM	Specifies that the object will be moved using the object's bottom as its origin. This value is valid for a VERT_LINK.

tagname Points to a null-terminated string containing the analog tagname to use for the link.

minValue Specifies the value when the object is at its highest (VERT_LINK) or leftmost (HORIZ_LINK) position.

maxValue Specifies the value when the object is at its lowest (VERT_LINK) or rightmost (HORIZ_LINK) position.

minPosition Specifies the number of pixels the object will move up (VERT_LINK) or left (HORIZ_LINK) from its current position in relation to the values specified for the minValue and maxValue parameters.

maxPosition Specifies the number of pixels the object will move down (VERT_LINK) or right (HORIZ_LINK) from its current position in relation to the values specified for the minValue and maxValue parameters.

Return Value The return value is the handle of the link if the function is successful. Otherwise, it is *whNull*.

Comments If a zero (0) is returned, *tagname* is NULL, too long or invalid *linkType* or invalid *referenceType*.

Stmt_New

EXPR

```
Stmt_New( HCHUNK hChunk,
          LPSTR pStmtStr,
          int len,
          LPINT errorCol)
```

Description

Creates and validates a block of statements and returns a handle to the validated statement.

Parameter	Description
hChunk	Handle to the memory section containing the object for which the statement is being created.
pStmtStr	Points to a null-terminated string containing a block of statements. Each statement must be separated by a CR-LF pair ("\r\n").
len	Specifies the number of bytes in the string.
errorCol	Points to an integer variable that indicates the character position where a syntax error occurred in the block of statements. This parameter is returned when there is an error, as indicated when the return value is *whNull*.

Return Value

The return value is the handle of the statement block if the function is successful. Otherwise, it is *whNull*.

Comments

None.

StmtTouchLnk_New

```
WHMEM

StmtTouchLnk_New( HCHUNK hChunk,
                  WHMEM whObj,
                  EXPR stmtDown,
                  EXPR stmtUp,
                  EXPR stmtWhile,
                  DWORD dwWhileFreq,
                  BYTE cKeyFlags,
                  WORD wVirtKey)
```

Description

Creates an action touch link for the object specified. Statements can be associated with the up, down, and while down conditions for the object.

Parameter	Description
hChunk	Handle to the memory section containing the object for which the link is being created.
whObj	Handle to the object for which the link is being created.
stmtDown	Handle to the block of statements that execute when the touch link button for the object is initially pressed. This parameter should be *whNull* if no statements are desired.
stmtUp	Handle to the block of statements that execute when the touch link button for the object is released. This parameter should be *whNull* if no statements are desired.
stmtWhile	Handle to the block of statements that execute while the touch link button for the object is held down. This parameter should be *whNull* if no statements are desired.
dwWhileFreq	Specifies the frequency (in milliseconds) that the stmtWhile block of statements will execute.
cKeyFlags	Specifies the flags used when a keyboard key is assigned to this link. This parameter can be a combination of the following values:

Value	Meaning
TOUCH_KS_SHIFT	Specifies the SHIFT key must be held down in addition to the key specified.
TOUCH_KS_CTRL	Specifies the CTRL key must be held down in addition to the key specified.
0	Specifies that only the specified key must be held down.

Parameter	Description
wVirtKey	Specifies the keyboard key equivalent assigned to this link. This value is 0 if there is no keyboard equivalent.

Return Value The return value is the handle of the link if the function is successful. Otherwise, it is *whNull*.

Comments None.

StrInputLnk_New

```
WHMEM
StrInputLnk_New( HCHUNK hChunk,
                 WHMEM whObj,
                 LPSTR tagname,
                 LPSTR userMsg,
                 BOOL bUseKeypad,
                 BOOL bEchoChars,
                 BOOL bInputOnly,
                 BYTE cKeyFlags,
                 WORD wVirtKey)
```

Description Creates a string (message) input link for the object specified.

Parameter	Description
hChunk	Handle to the memory section containing the object for which the link is being created.
whObj	Handle to the object for which the link is being created.
tagname	Points to a null-terminated string containing the string tagname to use for the link.
userMsg	Points to a null-terminated string containing the message or instruction to display if the bUseKeypad option is enabled.
bUseKeypad	Specifies the use of an on-screen keypad for entering new values if this value is non-zero.
bEchoChars	Specifies if the characters being input will be displayed as they are input. A value of FALSE will prevent sensitive data, such as a password, from being displayed on the screen.
bInputOnly	Specifies this link as input only, the value entered will not be displayed, if this value is non-zero. This setting only applies to objects that have text display associated with them (for example, a push button).

Parameter	Description
cKeyFlags	Specifies the flags used when a keyboard key is assigned to this link. This parameter can be a combination of the following values:

Value	Meaning
TOUCH_KS_SHIFT	Specifies the SHIFT key must be held down in addition to the key specified.
TOUCH_KS_CTRL	Specifies the CTRL key must be held down in addition to the key specified.
0	Specifies that only the specified key must be held down.

wVirtKey	Specifies the keyboard key equivalent assigned to this link. This value is 0 if there is no keyboard equivalent.

Return Value The return value is the handle of the link if the function is successful. Otherwise, it is *whNull*.

Comments If a zero (0) is returned, *tagname* is NULL, too long or invalid or, *userMsg* is NULL or too long.

StrOutputLnk_New

```
WHMEM
StrOutputLnk_New( HCHUNK hChunk,
                  WHMEM whObj,
                  LPSTR expression)
```

Description Creates a string (message) output link for the object specified.

Parameter	Description
hChunk	Handle to the memory section containing the object for which the link is being created.
whObj	Handle to the object for which the link is being created.
expression	Points to a null-terminated string containing the string expression or tagname to use for the link.

Return Value The return value is the handle of the link if the function is successful. Otherwise, it is *whNull*.

Comments If a zero (0) is returned, *expression* is NULL, too long or invalid.

StrTag_SetInfo

```
int
StrTag_SetInfo( DBHND dbHnd,
                LP_STRTAGINFO lpStrInfo)
```

Description Sets the string information for a database tagname with the given handle.

Parameter	Description
dbHnd	Handle to the database tagname.
lpStrInfo	Pointer to the string information structure.

Return Value Error code.

Comments None.

SymbolObj_New

```
WHMEM
SymbolObj_New( HCHUNK hChunk,
               WHMEM whParent,
               int left,
               int top,
               int right,
               int bottom)
```

Description Creates a symbol object at the specified location in the current application window. Populate the symbol with other objects by using this object's handle as the parent handle.

Parameter	Description
hChunk	Handle to the memory section containing the object.
whParent	Handle to the parent object (symbol, group, or wizard) that will contain this object. 0 indicates there is no parent object.
left	Specifies the x-coordinate of the upper-left corner.
top	Specifies the y-coordinate of the upper-left corner.
right	Specifies the x-coordinate of the lower-right corner.
bottom	Specifies the y-coordinate of the lower-right corner.

Return Value The return value is the handle of the object if the function is successful. Otherwise, it is *whNull*.

Comments None.

Tag_Find

```
DBHND
Tag_Find( LPSTR tagname)
```

Description Returns the handle of the database tagname with the given name.

Parameter	Description
tagname	Points to a null-terminated string containing the database tagname.

Return Value Handle to the tagname found. Otherwise, it is 0.

Comments None.

Tag_FindApplTopicItem

```
DBHND
Tag_FindApplTopicItem( LPSTR application,
                       LPSTR topic,
                       LPSTR item)
```

Description Returns the handle of the database tagname with the given I/O application, topic and item.

Parameter	Description
application	Points to a null-terminated string containing the application name (for example, "EXCEL"). The application can also include the node name (for example, "\\NODE\EXCEL").
topic	Points to a null-terminated string containing the I/O topic (for example, "SHEET1.XLS").
item	Points to a null-terminated string containing the I/O item (for example, "R1C1").

Return Value Handle to the tagname found.

Comments If the tagname is not found, 0 is returned.

Tag_GetAccessInfo

```
int
Tag_GetAccessInfo( DBHND dbHnd,
                   LP_TAGACCESSINFO lpAccessInfo)
```

Description Returns the access information for the database tagname with the given handle.

Parameter	Description
dbHnd	Handle to the database tagname.
lpAccessInfo	Pointer to the access information structure.

Return Value Error code.

Comments None.

Tag_GetGroup

```
DBHND
Tag_GetGroup( DBHND dbHnd)
```

Description Returns the group handle for the database tagname with the given handle.

Parameter	Description
dbHnd	Handle to the database tagname.

Return Value Handle to the group handle for the database tagname. A value of 0 indicates that the database tagname belongs to the system group.

Comments If *dbHnd* is NULL, a zero (0) is returned.

Tag_GetInfo

```
int
Tag_GetInfo( DBHND dbHnd,
             LP_TAGINFO lpTagInfo)
```

Description Returns the general information for the database tagname with the given handle.

Parameter	Description
lpTagInfo	Points to a TAGINFO structure that returns general information for the database tagname.

Return Value The return value is the error status of the function. 0 indicates success. A non-zero value is an error status.

Comments None.

Tag_GetRetentiveInfo

```
int
Tag_GetRetentiveInfo( DBHND dbHnd,
                      LP_TAGRETENTIVEINFO lpRetentiveInfo)
```

Description Returns the retentive information for a database tagname with the given handle.

Parameter	Description
dbHnd	Handle to the database tagname.
lpRetentiveInfo	Pointer to the retentive information structure.

Return Value Error code.

Comments None.

Tag_GetUniqueName

```
int
Tag_GetUniqueName( LPSTR basename,
                   LPSTR tagname)
```

Description Returns a unique tagname derived from the basename supplied.

Parameter	Description
basename	Points to a null-terminated string containing the basename for the database tagname. If the basename is not unique, a unique name is generated by indexing the basename.
tagname	Points to a null-terminated string returning the unique database tagname.

Return Value The return value is the error status of the function. 0 indicates success. A non-zero value is an error status.

Comments The area of memory specified by *tagname* must be large enough to store a full tagname (NL_TAGNAME).

Tag_GetValueAlarm

```
int
Tag_GetValueAlarm( DBHND dbHnd,
                   LP_VALALARMINFO lpValAlarm)
```

Description Returns the value alarm information for a database tagname with the given handle.

Parameter	Description
dbHnd	Handle to the database tagname.
lpValAlarm	Pointer to the value alarm tagname information structure.

Return Value Error code.

Comments None.

Tag_New

```
DBHND

Tag_New( LPSTR tagname,
         WORD tagType,
         WORD accessType,
         LPSTR comment,
         LP_TAGACCESSINFO lpAccessInfo)
```

Description

Creates a database tagname with the specified name, type, and comment.

Parameter	Description
tagname	Points to a null-terminated string containing the database tagname.
tagType	Specifies the flags that determine the tagname type. This parameter can be one of the following values:

Value	Meaning
TYPE_DISCRETE	Specifies a discrete tagname.
TYPE_INTEGER	Specifies an integer tagname.
TYPE_REAL	Specifies a floating point tagname.
TYPE_STRING	Specifies a string (message) tagname.
TYPE_ANALOG	Specifies an analog tagname. This is the same as TYPE_REAL.
TYPE_ALMGRP	Specifies an indirect alarm group reference. The *accessType* parameter is not used for this type.
TYPE_HIST	Specifies a historical trend tagname. The *accessType* parameter is not used for this type.
TYPE_TAGID	Specifies a tagname ID type. The *accessType* parameter is not used for this type.

Parameter	Description
accessType	Specifies the flags that determine the access mode for the tagname. This parameter can be one of the following values:

Value	Meaning
ACCESS_MEM	Specifies a memory access tagname.
ACCESS_DDE	Specifies an I/O access tagname.
ACCESS_IND	Specifies an indirect access tagname.

comment	Points to a null-terminated string containing the comment for the database tagname.
lpAccessInfo	Points to an TAGACCESSINFO structure that contains the I/O access information if the access type ACCESS_DDE is requested. This parameter should be NULL for all other access types.

Return Value The return value is the handle of the database tagname if the function is successful. Otherwise, it is 0.

Comments This function will fail if the database tagname already exists or if *tagType* or *accessType* are invalid.

Tag_SetAccessInfo

```
int
Tag_SetAccessInfo( DBHND dbHnd,
                   LP_TAGACCESSINFO lpAccessInfo)
```

Description Sets the access information for the database tagname with the given handle.

Parameter	Description
dbHnd	Handle to the database tagname.
lpAccessInfo	Pointer to the access information structure.

Return Value Error code.

Comments None.

Tag_SetDeviationAlarm

```
int
Tag_SetDeviationAlarm( DBHND dbHnd,
                       LP_DEVALARMINFO lpDevAlarm)
```

Description Sets the deviation alarm information for a database tagname with the given handle.

Parameter	Description
dbHnd	Handle to the database tagname.
lpDevAlarm	Pointer to the deviation alarm information structure.

Return Value Error code.

Comments None.

Tag_SetDiscAlarm

```
int
Tag_SetDiscAlarm( DBHND dbHnd,
                  LP_DISCALARMINFO lpDiscAlarm)
```

Description Sets the discrete alarm information for a database tagname with the given handle.

Parameter	Description
dbHnd	Handle to the database tagname.
lpDiscAlarm	Pointer to the discrete alarm information structure.

Return Value Error code.

Comments None.

Tag_SetEventInfo

```
int
Tag_SetEventInfo( DBHND dbHnd,
                  LP_TAGEVENTINFO lpEventInfo)
```

Description Sets the event information for the database tagname with the given handle.

Parameter	Description
dbHnd	Handle to the database tagname.
lpEventInfo	Pointer to the event information structure.

Return Value Error code.

Comments None.

Tag_SetGroup

```
int
Tag_SetGroup( DBHND dbHnd,
              DBHND dbGroup)
```

Description Sets the group handle for the database tagname with the given handle.

Parameter	Description
dbHnd	Handle to the database tagname.
dbGroup	Handle to the group to set for the database tagname.

Return Value The return value is the error status of the function. 0 indicates success. A non-zero value is an error status.

Comments None.

Tag_SetInfo

```
int
Tag_SetInfo( DBHND dbHnd,
             LP_TAGINFO lpTagInfo)
```

Description Sets the information specified in the *lpTagInfo* structure into the tagname specified by *dbHnd*.

Parameter	Description
dbHnd	Handle to the tagname.
lpTagInfo	Pointer to the TAGINFO structure.

Return Value Error code, 0 if successful.

Comments None.

Tag_SetRateOfChangeAlarm

```
int
Tag_SetRateOfChangeAlarm( DBHND dbHnd,
                          LP_ROCALARMINFO lpRocAlarm)
```

Description Sets the rate of change alarm information for a database tagname with the given handle.

Parameter	Description
dbHnd	Handle to the database tagname.
lpRocAlarm	Pointer to the rate of change alarm information structure.

Return Value Error code.

Comments None.

Tag_SetRetentiveInfo

```
int
Tag_SetRetentiveInfo( DBHND dbHnd,
                      LP_TAGRETENTIVEINFO lpRetentiveInfo)
```

Description Sets the retentive information for a database tagname with the given handle.

Parameter	Description
dbHnd	Handle to the database tagname.
lpRetentiveInfo	Pointer to the retentive information structure.

Return Value Error code.

Comments None.

Tag_SetScalingInfo

```
int
Tag_SetScalingInfo( DBHND dbHnd,
                    LP_TAGSCALEINFO lpScaleInfo)
```

Description Sets the scaling information for a database tagname with the given handle.

Parameter	Description
dbHnd	Handle to the database tagname.
lpScaleInfo	Pointer to the scaling information structure.

Return Value Error code.

Comments None.

Tag_SetValueAlarm

```
int
Tag_SetValueAlarm( DBHND dbHnd,
                   LP_VALALARMINFO lpValAlarm)
```

Description Sets the value alarm information for a database tagname with the given handle.

Parameter	Description
dbHnd	Handle to the database tagname.
lpValAlarm	Pointer to the value alarm information structure.

Return Value Error code.

Comments None.

Text_GetExtent

```
VOID
Text_GetExtent( LPLOGFONT lFnt,
                LPINT width
                LPINT height
                int len,
                LPSTR text)
```

Description

Returns the width and height of the text in pixels, based upon the logical font specified. This function should be used instead of the Windows GetTextExtent function when calculating the metrics of text to be used in InTouch objects.

Parameter	Description
lFnt	Logical font structure.
width	Returned width value of text in pixels.
height	Returned height value of text in pixels.
len	Length of string.
text	Text string for which extent is being requested.

Return Value

None.

Comments

None.

TextObj_New

```
WHMEM

TextObj_New( HCHUNK hChunk,
             WHMEM whParent,
             int left,
             int top,
             int right,
             int bottom,
             LPSTR text,
             WORD options)
```

Description

Creates a text object at the specified location in the current application window.

Parameter	Description
hChunk	Handle to the memory section containing the object.
whParent	Handle to the parent object (symbol, group, or wizard) that will contain this object. 0 indicates there is no parent object.
left	Specifies the x-coordinate of the upper-left corner.
top	Specifies the y-coordinate of the upper-left corner.
right	Specifies the x-coordinate of the lower-right corner.
bottom	Specifies the y-coordinate of the lower-right corner.
text	Points to a null-terminated string containing the text to display.
options	Specifies the flags that determine how to draw the text. This parameter can be one of the following values:

Value	Meaning
TEXT_CENTER	Centers text horizontally.
TEXT_LEFT	Left-aligns text.
TEXT_RIGHT	Right-aligns text.

Return Value

The return value is the handle of the object if the function is successful. Otherwise, it is *whNull*.

Comments

None.

TrendObj_SetItem

```
BOOL

TrendObj_SetItem( HCHUNK hChunk,
                  WHMEM whObj,
                  int index,
                  LPSTR expression,
                  LONG penColor,
                  int penWidth)
```

Description

Configures an item within the specified historical or real time trend object. Each item corresponds to a pen in the trend.

Parameter	Description
hChunk	Handle to the memory section containing the real time or historical trend object.
whObj	Handle to the real time or historical trend object.
index	Specifies the pen being modified. This value must be at least 1 and no greater than the allowed number of pens for the trend object.
expression	Points to a null-terminated string containing the analog expression or tagname to use for the pen.
penColor	Specifies the pen color. Colors are specified in Windows standard RGB format.
penWidth	Specifies the pen width.

Return Value

The return value is TRUE if the function is successful. Otherwise, it is FALSE.

Comments

A zero (0) is returned, if *whObj* is invalid or if length of expression is greater than MAX_EXPR_STRLEN.

TrendObj_SetTimeInfo

```
BOOL
TrendObj_SetTimeInfo( HCHUNK hChunk,
                      WHMEM whObj,
                      int nMajorDiv,
                      LONG majorDivColor,
                      int nMinorDiv,
                      LONG minorDivColor,
                      int nLabelDiv,
                      LONG labelDivColor,
                      WORD options,
                      LPSTR timeFormat)
```

Description

Configures time axis settings for the specified historical or real time trend object.

Parameter	Description
hChunk	Handle to the memory section containing the real time or historical trend object.
whObj	Handle to the real time or historical trend object.
nMajorDiv	Specifies the number of major division lines. This value must be an even multiple of the number of the 'nLabelDiv' parameter.
majorDivColor	Specifies the major division line color. Colors are specified in Windows standard RGB format.
nMinorDiv	Specifies the number of minor division lines between major division lines.
minorDivColor	Specifies the minor division line color. Colors are specified in Windows standard RGB format.
nLabelDiv	Specifies the number of major divisions per time label.
labelDivColor	Specifies the label text color. Colors are specified in Windows standard RGB format.
options	Specifies the flags that determine the time axis options for the trend object. This parameter can be a combination of the following values:

Value	Meaning
TREND_BOTTOM_LABELS	Specifies time labels at the bottom.
TREND_TOP_LABELS	Specifies time labels at the top.

timeFormat	Points to a null-terminated string containing the format specification for the time labels.

Return Value

The return value is TRUE if the function is successful. Otherwise, it is FALSE.

Comments

A zero (0) is returned, if *nMajorDiv*, *nMinorDiv*, *nLabelDiv* is less than 0 or greater than 9999 or *whObj* is invalid.

TrendObj_SetValueInfo

```
BOOL

TrendObj_SetValueInfo( HCHUNK hChunk,
                       WHMEM whObj,
                       int nMajorDiv,
                       LONG majorDivColor,
                       int nMinorDiv,
                       LONG minorDivColor,
                       int nLabelDiv,
                       LONG labelDivColor,
                       WORD options,
                       REAL minValue,
                       REAL maxValue)
```

Description

Configures the value axis settings within the specified historical or real time trend object.

Parameter	Description
hChunk	Handle to the memory section containing the real time or historical trend object.
whObj	Handle to the real time or historical trend object.
nMajorDiv	Specifies the number of major division lines. This value must be an even multiple of the number of the 'nLabelDiv' parameter.
majorDivColor	Specifies the major division line color. Colors are specified in Windows standard RGB format.
nMinorDiv	Specifies the number of minor division lines between major division lines.
minorDivColor	Specifies the minor division line color. Colors are specified in Windows standard RGB format.
nLabelDiv	Specifies the number of major divisions per value label.
labelDivColor	Specifies the label text color. Colors are specified in Windows standard RGB format.
options	Specifies the flags that determine the value axis options for the trend object. This parameter can be a combination of the following values:

Value	Meaning
TREND_LEFT_LABELS	Specifies value labels at the left.
TREND_RIGHT_LABELS	Specifies value labels at the right.

minValue	Specifies the minimum value for the range of values to be displayed.

Parameter	Description
maxValue	Specifies the maximum value for the range of values to be displayed.

Return Value The return value is TRUE if the function is successful. Otherwise, it is FALSE.

Comments A zero (0) is returned, if *nMajorDiv*, *nMinorDiv*, *nLabelDiv* is less than 0 or greater than 9999 or *minValue* is greater than *maxValue*.

VisibilityLnk_New

```
WHMEM
VisibilityLnk_New( HCHUNK hChunk,
                   WHMEM whObj,
                   LPSTR expression,
                   BOOL onOff)
```

Description Creates a visibility link for the object specified.

Parameter	Description
hChunk	Handle to the memory section containing the object for which the link is being created.
whObj	Handle to the object for which the link is being created.
expression	Points to a null-terminated string containing the analog expression or tagname to use for the link.
onOff	Specifies the desired visibility state. A value of TRUE will cause the object to be visible when the expression evaluates to TRUE.

Return Value The return value is the handle of the link if the function is successful. Otherwise, it is *whNull*.

Comments A zero (0) is returned if *expression* is NULL, too long or invalid.

WizardObj_New

```
WHMEM

WizardObj_New( HCHUNK hChunk,
               WHMEM whParent,
               int left,
               int top,
               int right,
               int bottom,
               LPSTR dllName,
               int dllIndex,
               WHMEM whData)
```

Description

Creates a wizard object at the specified location in the current application window. Populate the wizard with other objects by using this object's handle as the parent handle.

Parameter	Description
hChunk	Handle to the memory section containing the object. This parameter should be the value of the hChunk parameter passed to the **Wizard_New** function.
whParent	Handle to the parent object (symbol, group, or wizard) that will contain this object. 0 indicates there is no parent object. This parameter is normally 0.
left	Specifies the x-coordinate of the upper-left corner.
top	Specifies the y-coordinate of the upper-left corner.
right	Specifies the x-coordinate of the lower-right corner.
bottom	Specifies the y-coordinate of the lower-right corner.
dllName	Points to a null-terminated string containing the name of the DLL capable of creating the wizard. This parameter should be the value of the dllName parameter passed to the **Wizard_New** function.
dllIndex	Specifies the unique identifier for the wizard in the DLL. This parameter should be the value of the index parameter passed to the **Wizard_New** function.
whData	Handle to the data for the wizard being created. This parameter should be the value of the *whData* parameter passed to the **Wizard_New** function.

Return Value

The return value is the handle of the object if the function is successful. Otherwise, it is *whNull*.

Comments

A zero (0) is returned if length of *dllname* is greater than 32 characters.

WizProp_Delete

```
int
WizProp_Delete(  HCHUNK hChunk,
                 WHMEM whData,
                 LPSTR name)
```

Description Deletes the named wizard property.

Parameter	Description
hChunk	Handle to the memory section containing the wizard data.
whData	Handle to the wizard's data. This is the *whData* parameter passed to **Wizard_New** or the *whData* parameter passed to **Wizard_Edit**.
name	Points to a null-terminated string containing the property name.

Return Value The return value is the error status of the function. 0 indicates success. A non-zero value is an error status.

Comments Fails if the property *name* is not found.

WizProp_Find

```
int
WizProp_Find( HCHUNK hChunk,
              WHMEM whData,
              LPSTR name,
              WHMEM FAR *whProperty)
```

Description Returns a handle to the named wizard property.

Parameter	Description
hChunk	Handle to the memory section containing the wizard data.
whData	Handle to the wizard's data. This is the *whData* parameter passed to **Wizard_New** or the *whData* parameter passed to **Wizard_Edit**.
name	Points to a null-terminated string containing the property name.
whProperty	Points to a handle to the wizard property found. This parameter will return *whNull* if no property is found in the wizard matching the name.

Return Value The return value is the error status of the function. 0 indicates success. A non-zero value is an error status.

Comments Fails if the property *name* is not found.

WizProp_GetBlock

```
int
WizProp_GetBlock( HCHUNK hChunk,
                  WHMEM whData,
                  LPSTR name,
                  DWORD dwMax,
                  LPVOID data,
                  LPDWORD dwSize)
```

Description

Returns the data for a named wizard property that contains a block of data.

Parameter	Description
hChunk	Handle to the memory section containing the wizard data.
whData	Handle to the wizard's data. This is the *whData* parameter passed to **Wizard_New** or the *whData* parameter passed to **Wizard_Edit**.
name	Points to a null-terminated string containing the property name.
dwMax	Specifies the maximum number of bytes to return.
data	Points to the buffer to receive the property data.
dwSize	Points to a DWORD that indicates the number of bytes actually stored in the data buffer.

Return Value

The return value is the error status of the function. 0 indicates success. A non-zero value is an error status.

Comments

Fails if property *name* is not found. Only returns property if the property specified by *name* is of type block.

WizProp_GetDouble

```
int
WizProp_GetDouble( HCHUNK hChunk,
                   WHMEM whData,
                   LPSTR name,
                   double FAR * data,
                   double dataDef)
```

Description

Returns a floating point value for the named wizard property.

Parameter	Description
hChunk	Handle to the memory section containing the wizard data.
whData	Handle to the wizard's data. This is the *whData* parameter passed to **Wizard_New** or the *whData* parameter passed to **Wizard_Edit**.
name	Points to a null-terminated string containing the property name.
data	Points to a double that will receive the value of the property.
dataDef	Specifies the default value to return in the data if the property is not found.

Return Value

The return value is the error status of the function. 0 indicates success. A non-zero value is an error status.

Comments

Fails if property *name* is not found or the property type is not DOUBLE.

WizProp_GetDWord

```
int
WizProp_GetDWord( HCHUNK hChunk,
                  WHMEM whData,
                  LPSTR name,
                  LPDWORD data,
                  DWORD dataDef)
```

Description

Returns a double word (32-bit) value for the named wizard property.

Parameter	Description
hChunk	Handle to the memory section containing the wizard data.
whData	Handle to the wizard's data. This is the *whData* parameter passed to **Wizard_New** or the *whData* parameter passed to **Wizard_Edit**.
name	Points to a null-terminated string containing the property name.
data	Points to a DWORD that will receive the value of the property.
dataDef	Specifies the default value to return in the data if the property is not found.

Return Value

The return value is the error status of the function. 0 indicates success. A non-zero value is an error status.

Comments

Fails if property *name* is not found or the property type is not DWORD.

WizProp_GetExpr

```
int
WizProp_GetExpr(  HCHUNK hChunk,
                  WHMEM whData,
                  LPSTR name,
                  DWORD dwMax,
                  LPSTR data,
                  LPSTR dataDef)
```

Description

Returns the data for a named wizard property that contains an expression.

Parameter	Description
hChunk	Handle to the memory section containing the wizard data.
whData	Handle to the wizard's data. This is the *whData* parameter passed to the **Wizard_New** or the *whData* parameter passed to the **Wizard_Edit**.
name	Points to a null-terminated string containing the property name.
dwMax	Specifies the maximum number of bytes to return.
data	Points to the buffer to receive the property data.
dataDef	Specifies the default value to return in the data if the property is not found.

Return Value

The return value is the error status of the function. 0 indicates success. A non-zero value is an error status.

Comments

Fails if property *name* is not found or the property type is not EXPR. Use this property type for any expression, or tagname property. This will expose the property to InTouch as an expression and provide the standard support for expressions, such as substitute tags, and automatic placeholder generation.

WizProp_GetFont

```
int
WizProp_GetFont( HCHUNK hChunk,
                 WHMEM whData,
                 LPSTR name,
                 LPLOGFONT logFont)
```

Description

Returns the logical font data for the named wizard property.

Parameter	Description
hChunk	Handle to the memory section containing the wizard data.
whData	Handle to the wizard's data. This is the *whData* parameter passed to **Wizard_New** or the *whData* parameter passed to **Wizard_Edit**.
name	Points to a null-terminated string containing the property name.
logFont	Points to a LOGFONT structure that returns the contents of the logical font property. LOGFONT is a Windows structure.

Return Value

The return value is the error status of the function. 0 indicates success. A non-zero value is an error status.

Comments

This function should be used instead of attempting to retrieve font settings using the **WizProp_GetBlock** function. This function will isolate your code from differences in how the LOGFONT structure is saved on all Windows and Windows NT platforms. Also, if you are attempting to use the standard font property, 'ww_font', to enable toolbar font operations on your wizard, you must use this function to retrieve the font property.

Fails if property *name* is not found or the property type is not TEXT or BLOCK.

WizProp_GetStmt

```
int
WizProp_GetStmt( HCHUNK hChunk,
                 WHMEM whData,
                 LPSTR name,
                 DWORD dwMax,
                 LPSTR data,
                 LPSTR dataDef)
```

Description

Returns the data for a named wizard property that contains a statement.

Parameter	Description
hChunk	Handle to the memory section containing the wizard data.
whData	Handle to the wizard's data. This is the *whData* parameter passed to the **Wizard_New** or the *whData* parameter passed to the **Wizard_Edit**.
dwMax	Specifies the maximum number of bytes to return.
data	Points to the buffer to receive the property data.
dataDef	Specifies the default value to return in the data if the property is not found.

Return Value

The return value is the error status of the function. 0 indicates success. A non-zero value is an error status.

Comments

Fails if property *name* is not found or the property type is not STMT. Use this property type for any script property. This will expose the property to InTouch as a script and provide standard support for scripts, such as substitute tags and automatic placeholder generation.

WizProp_GetString

```
int
WizProp_GetString( HCHUNK hChunk,
                   WHMEM whData,
                   LPSTR name,
                   DWORD dwMax,
                   LPSTR data,
                   LPSTR dataDef)
```

Description

Returns a NULL terminated string for the named wizard property.

Parameter	Description
hChunk	Handle to the memory section containing the wizard data.
whData	Handle to the wizard's data. This is the *whData* parameter passed to **Wizard_New** or the *whData* parameter passed to **Wizard_Edit**.
name	Points to a null-terminated string containing the property name.
dwMax	Specifies the maximum number of bytes to return, including the null character.
data	Points to the character array to receive the property data.
dataDef	Specifies the default value to return in the data if the property is not found.

Return Value

The return value is the error status of the function. 0 indicates success. A non-zero value is an error status.

Comments

Fails if property *name* is not found or the property type is not STRING. Do not use this property type for expressions, tagnames or scripts. See **WizProp_GetExpr** or **WizProp_GetStmt** for this purpose.

WizProp_New

```
int
WizProp_New( HCHUNK hChunk,
             WHMEM whData,
             LPSTR name,
             char type,
             WHMEM FAR *whProperty)
```

Description

Creates a wizard property with the name and type specified.

Parameter	Description
hChunk	Handle to the memory section containing the wizard data.
whData	Handle to the wizard's data. This is the *whData* parameter passed to **Wizard_New** or the *whData* parameter passed to **Wizard_Edit**.
name	Points to a null-terminated string containing the property name.
type	Specifies the flags that determine the property type. This parameter can be one of the following values:

Value	Meaning
WIZPROP_TYPE_DWORD	Specifies a DWORD data type.
WIZPROP_TYPE_REAL	Specifies a floating point data type.
WIZPROP_TYPE_STRING	Specifies a string data type.
WIZPROP_TYPE_BLOCK	Specifies a block data type.
WIZPROP_TYPE_FONT	Specifies a font data type.
WIZPROP_TYPE_EXPR	Specifies an expression or tagname type.
WIZPROP_TYPE_STMT	Specifies a script type. A script is a series of InTouch statements.

whProperty	Points to a handle to the wizard property created. This parameter will return *whNull* if the property could not be created.

Return Value

The return value is the error status of the function. 0 indicates success. A non-zero value is an error status.

Comments

Fails if property *name* exists.

WizProp_SetBlock

```
int
WizProp_SetBlock( HCHUNK hChunk,
                  WHMEM whData,
                  LPSTR name,
                  DWORD dwSize,
                  LPVOID data)
```

Description Sets the data for a named wizard property that contains a block of data.

Parameter	Description
hChunk	Handle to the memory section containing the wizard data.
whData	Handle to the wizard's data. This is the *whData* parameter passed to **Wizard_New** or the *whData* parameter passed to **Wizard_Edit**.
name	Points to a null-terminated string containing the property name.
dwSize	Specifies the number of bytes in the data buffer.
data	Points to the buffer containing the property data.

Return Value The return value is the error status of the function. 0 indicates success. A non-zero value is an error status.

Comments None.

WizProp_SetDouble

```
int
WizProp_SetDouble( HCHUNK hChunk,
                   WHMEM whData,
                   LPSTR name,
                   double data)
```

Description Sets a floating point value for the named wizard property.

Parameter	Description
hChunk	Handle to the memory section containing the wizard data.
whData	Handle to the wizard's data. This is the *whData* parameter passed to **Wizard_New** or the *whData* parameter passed to **Wizard_Edit**.
name	Points to a null-terminated string containing the property name.
data	Specifies the property value.

Return Value The return value is the error status of the function. 0 indicates success. A non-zero value is an error status.

Comments None.

WizProp_SetDWord

```
int
WizProp_SetDWord( HCHUNK hChunk,
                  WHMEM whData,
                  LPSTR name,
                  DWORD data)
```

Description

Sets a double word (32-bit) value for the named wizard property.

Parameter	Description
hChunk	Handle to the memory section containing the wizard data.
whData	Handle to the wizard's data. This is the *whData* parameter passed to **Wizard_New** or the *whData* parameter passed to **Wizard_Edit**.
name	Points to a null-terminated string containing the property name.
data	Specifies the property value.

Return Value

The return value is the error status of the function. 0 indicates success. A non-zero value is an error status.

Comments

None.

WizProp_SetExpr

```
int
WizProp_SetExpr( HCHUNK hChunk,
                 WHMEM whData,
                 LPSTR name,
                 LPSTR data)
```

Description

Sets the data for a name wizard property that contains an expression.

Parameter	Description
hChunk	Handle to the memory section containing the wizard data.
whData	Handle to the wizard's data. This is the *whData* parameter passed to the **Wizard_New** or the *whData* parameter passed to the **Wizard_Edit**.
name	Points to a NULL-terminated string containing the property name.
data	Points to the buffer to receive the property data.

Return Value

The return value is the error status of the function. 0 indicates success. A non-zero value is an error status.

Comments

Use this property type for any expression, or tagname property. This will expose the property to InTouch as an expression and provide the standard support for expressions, such as substitute tags, and automatic placeholder generation.

WizProp_SetFont

```
int
WizProp_SetFont( HCHUNK hChunk,
                 WHMEM whData,
                 LPSTR name,
                 LPLOGFONT logFont)
```

Description

Sets the logical font data for the named wizard property.

Parameter	Description
hChunk	Handle to the memory section containing the wizard data.
whData	Handle to the wizard's data. This is the *whData* parameter passed to **Wizard_New** or the *whData* parameter passed to **Wizard_Edit**.
name	Points to a null-terminated string containing the property name.
logFont	Points to a LOGFONT structure that defines the characteristics of the logical font property. LOGFONT is a Windows structure.

Return Value

The return value is the error status of the function. 0 indicates success. A non-zero value is an error status.

Comments

This function should be used instead of attempting to store font settings using the **WizProp_SetBlock** function. This function will isolate your code from differences in how the LOGFONT structure is saved on all Windows and Windows NT platforms. Also, if you are attempting to use the standard font property, 'ww_font', to enable toolbar font operations on your wizard, you must use this function to store the font property.

WizProp_SetStmt

```
int
WizProp_SetStmt( HCHUNK hChunk,
                 WHMEM whData,
                 LPSTR name,
                 LPSTR data)
```

Description

Sets the data for a name wizard property that contains a statement.

Parameter	Description
hChunk	Handle to the memory section containing the wizard data.
whData	Handle to the wizard's data. This is the *whData* parameter passed to the **Wizard_New** or the *whData* parameter passed to the **Wizard_Edit**.
name	Points to a NULL-terminated string containing the property name.
data	Points to the buffer to receive the property data.

Return Value

The return value is the error status of the function. 0 indicates success. A non-zero value is an error status.

Comments

Use this property type for any script property. This will expose the property to InTouch as a script and provide standard support for scripts, such as substitute tags and automatic placeholder generation.

WizProp_SetString

```
int
WizProp_SetString( HCHUNK hChunk,
                   WHMEM whData,
                   LPSTR name,
                   LPSTR data)
```

Description

Sets a NULL terminated string for the named wizard property.

Parameter	Description
hChunk	Handle to the memory section containing the wizard data.
whData	Handle to the wizard's data. This is the *whData* parameter passed to **Wizard_New** or the *whData* parameter passed to **Wizard_Edit**.
name	Points to a null-terminated string containing the property name.
data	Points to a null-terminated string containing the property data.

Return Value

The return value is the error status of the function. 0 indicates success. A non-zero value is an error status.

Comments

Do not use this property type for expressions, tagnames or scripts. See **WizProp_SetExpr** or **WizProp_SetStmt** for this purpose.

WWDlg_CheckExprCtrl

```
int
WWDlg_CheckExprCtrl( HWND hDlg,
                     int ctrlID,
                     BYTE type)
```

Description

Validates the dialog item using the standard InTouch validation of script expressions. Error messages are automatically displayed when an error is detected.

Parameter	Description
hDlg	Identifies the dialog box.
ctrlID	Specifies the identifier of the dialog box control that contains text to validate as a script expression. The control must be a Windows standard edit box control.
type	Specifies the flags that determine the expression type. This parameter can be one of the following values:

Value	Meaning
TYPE_DISCRETE	Specifies a discrete expression.
TYPE_INTEGER	Specifies an integer expression.
TYPE_REAL	Specifies a floating point expression.
TYPE_STRING	Specifies a string (message) expression.
TYPE_ANALOG	Specifies an analog expression. An integer or floating point expression is allowed.

Return Value

The return value is the error status of the function. 0 indicates success. A non-zero value is an error status.

Comments

None.

WWDlg_CheckTagCtrl

```
int
WWDlg_CheckTagCtrl( HWND hDlg,
                    int ctrlID,
                    BYTE type)
```

Description

Validates the dialog item using the standard InTouch validation of database tagnames. Error messages are automatically displayed when an error is detected.

Parameter	Description
hDlg	Identifies the dialog box.
ctrlID	Specifies the identifier of the dialog box control that contains text to validate as a database tagname. The control must be a Windows standard edit box control.
type	Specifies the flags that determine the tagname type. This parameter can be one of the following values:

Value	Meaning
TYPE_DISCRETE	Specifies a discrete tagname.
TYPE_INTEGER	Specifies an integer tagname.
TYPE_REAL	Specifies a floating point tagname.
TYPE_STRING	Specifies a string (message) tagname.
TYPE_ANALOG	Specifies an analog tagname. An integer or floating point tagname is allowed.
TYPE_NUM	Specifies an integer, floating point or discrete tagname.
TYPE_ALMGRP	Specifies an indirect alarm group reference.
TYPE_HIST	Specifies a historical trend tagname.
TYPE_TAGID	Specifies a tagname ID type.
TYPE_ANY	Specifies any tagname type.

Return Value

The return value is the error status of the function. 0 indicates success. A non-zero value is an error status.

Comments

None.

WWDlg_GetDoubleCtrl

```
int
WWDlg_GetDoubleCtrl( HWND hDlg,
                     int ctrlID,
                     REAL FAR * value)
```

Description

Validates the dialog item using standard InTouch validation rules for floating point values. The resulting value is returned.

Parameter	Description
hDlg	Identifies the dialog box.
ctrlID	Specifies the identifier of the dialog box control that contains text to validate and convert to a floating point value. The control must be a Windows standard edit box control.
value	Points to a REAL that returns the converted floating point value.

Return Value

The return value is the error status of the function. 0 indicates success. A non-zero value is an error status.

Comments

None.

WWDlg_ProcessKeyCtrl

```
int
WWDlg_ProcessKeyCtrl( HWND hDlg,
                      int enableKeyCtrlID,
                      int ctrlID)
```

Description

Processes messages to the key-equivalent handling controls.

Parameter	Description
hDlg	Identifies the dialog box.
enableKeyCtrlID	Specifies the identifier of the dialog box control that enables or disables the key equivalent controls. It is used in the key equivalent combination and must be a check box control.
ctrlID	Specifies the identifier of the dialog box control to which the message has been sent.

Return Value

The return value is the error status of the function. 0 indicates success. A non-zero value is an error status.

Comments

WWDlg_ProcessKeyCtrl should be called for message sent to any of the key equivalent controls. It will process those messages appropriately.

WWDlg_RegisterColorCtrl

```
int
WWDlg_RegisterColorCtrl( HWND hDlg,
                         int ctrlID,
                         DWORD color)
```

Description

Registers a dialog item to use the standard InTouch color choice dialog.

Parameter	Description
hDlg	Identifies the dialog box.
ctrlID	Specifies the identifier of the dialog box control that uses the standard InTouch color choice dialog. The control must be a Windows standard list box control with LBS_NOTIFY enabled.
color	Specifies the initial color for the dialog box control. Colors are specified in Windows standard RGB format.

Return Value

The return value is the error status of the function. 0 indicates success. A non-zero value is an error status.

Comments

Make sure that **WWDlg_UnregisterColorCtrl** is called to free any associated memory.

WWDlg_RegisterKeyCtrl

```
int
WWDlg_RegisterKeyCtrl( HWND hDlg,
                       int enableKeyCtrlID,
                       int ctrlKeyCtrlID,
                       int shiftKeyCtrlID,
                       int selectKeyCtrlID,
                       int keyTextCtrlID,
                       DWORD dwPropKeyCode,
                       DWORD dwPropKeyFlags)
```

Description

Registers a set of dialog items to obtain key-equivalent handling information for the wizard.

Parameter	Description
hDlg	Identifies the dialog box.
enableKeyCtrlID	Specifies the identifier of the dialog box control that enables or disables the key equivalent controls. Is used in the key equivalent combination. Must be a check box control.
ctrlKeyCtrlID	Specifies the identifier of the dialog box control that specifies if the "ctrl" key is used in the key equivalent combination. Must be a check box control.
shiftKeyCtrlID	Specifies the identifier of the dialog box control that specifies if the "shift" key is used in the key equivalent combination. Must be a check box control.
selectKeyCtrlID	Specifies the identifier of the dialog box control that displays the standard InTouch key selection dialog. Must be a push button control.
KeyTextCtrlID	Specifies the identifier of the dialog box control that displays chosen key equivalent text. Must be a static text control.
dwPropKeyCode	Key code for the selected Key (zero means no key selected). Uses virtual key codes as defined in WINDOWS.H.
dwPropKeyFlags	Key flags values.

Value	Meaning
TOUCH_KS_SHIFT	Use shift key.
TOUCH_KS_CTRL	Use ctrl key.
0	No flags.

Return Value

The return value is the error status of the function. 0 indicates success. A non-zero value is an error status.

Comments

None.

WWDlg_RegisterTagNameCtrl

```
int
WWDlg_RegisterTagNameCtrl( HWND hDlg,
                           int ctrlID)
```

Description

Registers a dialog item to respond to a double-click by displaying the standard tagname selection dialog.

Parameter	Description
hDlg	Identifies the dialog box.
ctrlID	Specifies the identifier of the dialog box control that uses the standard Tagname Selection Dialog.

Return Value

The return value is the error status of the function. 0 indicates success. A non-zero value is an error status.

Comments

None.

WWDlg_ScriptEdit

```
VOID
WWDlg_ScriptEdit( HWND hDlg,
                  HCHUNK hChunk,
                  LPSTR lpString)
```

Description

Displays a generic script editing dialog.

Parameter	Description
hDlg	Identifies the dialog box.
hChunk	Memory handle.
lpString	Points to an area of memory that will receive the script text.

Return Value

The return value is the error status of the function. 0 indicates success. A non-zero value is an error status.

Comments

The caller of **WWDlg_ScriptEdit** is responsible for allocating adequate memory for the resulting script statement. The Define (STMT_STRLEN) has been provided for that purpose. The memory allocated for the script should be globally allocated and should be freed when no longer needed.

WWDlg_SetDoubleCtrl

```
int
WWDlg_SetDoubleCtrl( HWND hDlg,
                     int ctrlID,
                     REAL value)
```

Description

Sets the dialog item with the character representation for the floating point value specified.

Parameter	Description
hDlg	Identifies the dialog box.
ctrlID	Specifies the identifier of the dialog box control that contains text representing a floating point value. The control must be a Windows standard edit box control.
value	Specifies the floating point value to convert to text.

Return Value

The return value is the error status of the function. 0 indicates success. A non-zero value is an error status.

Comments

None.

WWDlg_UnregisterColorCtrl

```
int
WWDlg_UnregisterColorCtrl( HWND hDlg,
                           int ctrlID,
                           DWORD FAR *color)
```

Description

Unregisters a dialog item that was registered using **WWDlg_RegisterColorCtrl**. Any memory used is freed and the current color selection is returned.

Parameter	Description
hDlg	Identifies the dialog box.
ctrlID	Specifies the identifier of the dialog box control that uses the standard InTouch color choice dialog. The control must be a Windows standard list box control with LBS_NOTIFY enabled.
color	Points to a DWORD that returns the current color selection.

Return Value

The return value is the error status of the function. 0 indicates success. A non-zero value is an error status.

Comments

None.

WWDlg_UnregisterKeyCtrl

```
int
WWDlg_UnregisterKeyCtrl( HWND hDlg,
                         int enableKeyCtrlID,
                         DWORD FAR * dwPropKeyCode,
                         DWORD FAR * dwPropKeyFlags)
```

Description

Unregisters the set of dialog items to that were registered in **WWDlg_RegisterKeyCtrl**. Frees any memory associated with the mapping of dialog items.

Parameter	Description
hDlg	Identifies the dialog box.
enableKeyCtrlID	Specifies the identifier of the dialog box control that enables or disables the key equivalent controls. Is used in the key equivalent combination. Must be a check box control.
dwPropKeyCode	Key code for the selected Key (zero means no key selected). Uses virtual key codes as defined in windows.h.
dwPropKeyFlags	Key flags values:

Value	Meaning
TOUCH_KS_SHIFT	Use shift key.
TOUCH_KS_CTRL	Use ctrl key.
0	No flags.

Return Value

The return value is the error status of the function. 0 indicates success. A non-zero value is an error status.

Comments

The selected key code and key flags are returned in *dwPropKeyCode* and *dwPropKeyFlags*.

WWDlg_UnregisterTagNameCtrl

```
int
WWDlg_UnregisterTagNameCtrl( HWND hDlg,
                             int ctrlID)
```

Description

Unregisters a dialog item that was registered using **WWDlg_RegisterTagnameCtrl**. Any memory is freed and the control takes on its standard Windows capabilities.

Parameter	Description
hDlg	Identifies the dialog box.
ctrlID	Specifies the identifier of the dialog box control that uses the standard Tagname Selection Dialog.

Return Value

The return value is the error status of the function. 0 indicates success. A non-zero value is an error status.

Comments

None.

WWKit_GetKeyStatus

```
BOOL
WWKit_GetKeyStatus( VOID)
```

Description

Retrieves the current status of the Wonderware hardware key.

Return Value

The return value is the status of the key. TRUE indicates that the key is attached to the parallel port and is functioning normally. FALSE indicates either the key is not attached or is not functioning.

Comments

This function could be used by third-party Wizard developers to implement a copy-protection scheme for their product. For example, the developer of the wizard can choose to only make a certain subset of their wizard's functionality available when the key is not there (demo mode). See also the function **WWKit_GetSerialNumber**.

WWKit_GetLastError

```
int
WWKit_GetLastError( VOID)
```

Description

Returns the error status of the most recent call to the Wizard Toolkit.

Parameter	Description
Return Value	The return value, if non-zero, indicates the error status. 0 indicates the most recent call was successful.

Value	Meaning
1	unsupported function
2	out of memory
3	wizard read error
10	object null
11	object bad type
20	property not found
21	property exists
22	bad property type
23	property type mismatch
24	property name too long
30	invalid access data
31	invalid access ID
32	invalid item name
33	problem with tagname
34	problem with tagname type
35	problem with expression
36	problem with expression type
37	problem with application/topic

Value	Meaning
38	problem with Stmt expression
40	problem with text too long
41	problem with dllojb not found
42	problem with trend object - sample parameter
43	problem with trend object span units
44	problem with trend object span units
45	problem with trend object min/max relationship
46	bad time parameter
47	bad mode parameter
50	link type invalid
51	alarm type invalid
52	error from Lnk_New

Comments **WWKit_GetLastError** will clear the error condition.

WWKit_GetSerialNumber

```
DWORD
WWKit_GetSerialNumber( VOID)
```

Description Retrieves the serial number of the Wonderware hardware key.

Return Value The return value is the serial number of the key.

Comments This function could be used by third-party Wizard developers to implement a copy-protection scheme for their product. For example, the developer of the wizard can choose to only make their wizard function when certain numbered keys are attached. See also the function **WWKit_GetKeyStatus**.

WWKit_Init

```
VOID
WWKit_Init( VOID)
```

Description Initializes the wizard tool kit if not previously done so. This call must be done once per wizard DLL.

Return Value None.

Comments None.

WWKit_SetBrush

```
VOID
WWKit_SetBrush(  LPLOGBRUSH lpLogBrush)
```

Description Sets the brush used when manipulating objects that have a brush associated with them.

Parameter	Description
lpLogBrush	Points to a LOGBRUSH structure that defines the characteristics of the logical brush used for objects that require one. LOGBRUSH is a Windows structure.

Return Value None.

Comments The logical brush does not need to be modified until an object is created that requires a different logical brush. The logical brush is used for InTouch objects that allow fill color selection.

WWKit_SetFont

```
VOID
WWKit_SetFont( LPLOGFONT lpLogFont)
```

Description Sets the font used when manipulating objects that have a font associated with them.

Parameter	Description
lpLogFont	Points to a LOGFONT structure that defines the characteristics of the logical font used for objects that require one. LOGFONT is a Windows structure.

Return Value None.

Comments The logical font does not need to be modified until an object is created that requires a different logical font. The logical font is used for InTouch objects that allow font selection.

WWKit_SetPen

```
VOID
WWKit_SetPen( LPLOGPEN lpLogPen)
```

Description Sets the pen used when manipulating objects that have a pen associated with them.

Parameter	Description
lpLogPen	Points to a LOGPEN structure that defines the characteristics of the logical pen used for objects that require one. LOGPEN is a Windows structure.

Return Value None.

Comments The logical pen does not need to be modified until an object is created that requires a different logical pen. The logical pen is used for InTouch objects that allow line color and width selection.

WWKit_SetTextBrush

```
VOID
WWKit_SetTextBrush( LPLOGBRUSH lpLogBrush)
```

Description Sets the text brush used when manipulating objects that have a text brush associated with them.

Parameter	Description
lpLogBrush	Points to a LOGBRUSH structure that defines the characteristics of the logical brush used for objects that require a text logical brush. LOGBRUSH is a Windows structure.

Return Value None.

Comments The text logical brush does not need to be modified until an object is created that requires a different text logical brush. The text logical brush is used for InTouch objects that allow text background color selection.

WWKit_SetTextPen

```
VOID
WWKit_SetTextPen( LPLOGPEN lpLogPen)
```

Description Sets the text pen used when manipulating objects that have a text pen associated with them.

Parameter	Description
lpLogPen	Points to a LOGPEN structure that defines the characteristics of the logical pen used for objects that require a text logical pen. LOGPEN is a Windows structure.

Return Value None.

Comments The text logical pen does not need to be modified until an object is created that requires a different text logical pen. The text logical pen is used for InTouch objects that allow text color selection.

CHAPTER 7

Wizard API Structures

The Wizard Toolkit contains several structures associated with the Wizard API functions. These structures are defined in alphabetic order in this chapter.

Contents

ACCESSNAMEINFO

```
typedef struct {
                LPSTR application;
                LPSTR topic;
                WORD bRequestInitialData;
                WORD bAlwaysAdvise;
} ACCESSNAMEINFO, FAR* LP_ACCESSNAMEINFO;
```

Description The ACCESSNAMEINFO structure contains information used to define Access Names. Items in the structure are set and passed to **AccessName_New** and **AccessName_SetInfo**.

Element	Description
application	Points to a null-terminated string that specifies the application name
topic	Points to a null-terminated string that specifies the topic name
bRequestInitialData	If TRUE, set "Request initial data", if FALSE, "wait for change"
bAlwaysAdvise	If TRUE, set "advise all points", if FALSE advise only active points

See Also **AccessName_New**, **AccessName_SetInfo**

ANLGTAGINFO

```
typedef struct {
                double initValue;
} ANLGTAGINFO, FAR* LP_ANLGTAGINFO;
```

Description The ANLGTAGINFO structure contains information to set the initial value of an analog database tagname. Items in the structure are set and passed to the **AnlgTag_SetInfo** function. The **AnlgTag_GetInfo** function returns the information in this structure.

Element	Description
initValue	Specifies the initial value for the analog tagname

Comments The *initValue* field in the structure is CAST appropriately to REAL or INTG, depending on the type of the database tagname.

See Also **AnlgTag_GetInfo**, **AnlgTag_SetInfo**

DEVALARMINFO

```
typedef struct {
                WORD majorAlarmState;
                WORD minorAlarmState;
                WORD majorAlarmPriority;
                WORD minorAlarmPriority;
                REAL majorAlarmValue;
                REAL minorAlarmValue;
                REAL alarmDeadband;
                double devTarget;
} DEVALARMINFO, FAR* LP_DEVALARMINFO;
```

Description

The DEVALARMINFO structure contains information to set the deviation alarm fields in the database tagname. Items in the structure are set and passed to the **Tag_SetDeviationAlarm** function.

Element	Description
majorAlarmState	Specifies the major deviation alarm state
minorAlarmState	Specifies the minor deviation alarm state
majorAlarmPriority	Specifies the major deviation alarm priority
minorAlarmPriority	Specifies the minor deviation alarm priority
majorAlarmValue	Specifies the major deviation alarm value
minorAlarmValue	Specifies the minor deviation alarm value
alarmDeadband	Specifies the deviation alarm deadband
devTarget	Specifies the deviation target (CAST for type)

See Also

Tag_SetDeviationAlarm

DISCALARMINFO

```
typedef struct {
                WORD alarmState;
                WORD alarmPriority;
} DISCALARMINFO, FAR* LP_DISCALARMINFO;
```

Description The DISCALARMINFO structure contains information to set the discrete alarm fields in the database tagname. Items in the structure are set and passed to the **Tag_SetDiscAlarm** function.

Element	Description	
alarmState	Specifies the alarm state	
	Value	**Meaning**
	ALARMSTATE_NONE	No alarm state
	ALARMSTATE_OFF	Alarm is off
	ALARMSTATE_ON	Alarm is on
alarmPriority	Specifies the alarm priority	

See Also **Tag_SetDiscAlarm**

DISCTAGINFO

```
typedef struct {
                WORD initValue;
                LPSTR onMsg;
                LPSTR offMsg;
} DISCTAGINFO, FAR* LP_DISCTAGINFO;
```

Description The DISCTAGINFO structure contains information to set initial value and the On/Off message fields in a discrete database tagname. Items in the structure are set and passed to the **DiscTag_SetInfo** function. The **DiscTag_GetInfo** function returns the information in this structure.

Element	Description
initValue	Specifies the initial value (on or off)
onMsg	Points to a null-terminated string that specifies the on value message
offMsg	Points to a null-terminated string that specifies the off value message

See Also **DiscTag_GetInfo**, **DiscTag_SetInfo**

ROCALARMINFO

```
typedef struct {
                WORD alarmState;
                WORD alarmPriority;
                WORD rocUnits;
                REAL pctChange;
} ROCALARMINFO, FAR* LP_ROCALARMINFO;
```

Description

The ROCALARMINFO structure contains information to set the rate of change alarm fields in the database tagname. Items in the structure are set and passed to the **Tag_SetRateOfChangeAlarm** function.

Element	Description
alarmState	Specifies the alarm state
alarmPriority	Specifies the alarm priority
rocUnits	Specifies the rate of change units
pctChange	Specifies the percent change value

See Also

Tag_SetRateOfChangeAlarm

STRTAGINFO

```
typedef struct {
                WORD nMaxString;
                LPSTR initValue;
} STRTAGINFO, FAR* LP_STRTAGINFO;
```

Description

The STRTAGINFO structure contains information to set the initial value and maximum string length for a message tagname. Items in the structure are set and passed to the **StrTag_SetInfo** function.

Element	Description
nMaxString	Specifies the maximum string length
initValue	Points to a null-terminated string that specifies the initial value

See Also

StrTag_SetInfo

TAGACCESSINFO

```
typedef struct {
                DDESOURCE accessID;
                LPSTR itemName;
} TAGACCESSINFO, FAR* LP_TAGACCESSINFO;
```

Description

The TAGACCESSINFO structure contains information to set the retentive flags in a database tagname. The items in the structure are set and passed to the **Tag_New** and **Tag_SetAccessInfo** functions. The **Tag_GetAccessInfo** function returns the value in the TAGACCESSINFO structure.

Element	Description
accessID	Access ID, as returned from AccessName_Find, AccessName_FindApplTopic, or AccessName_New
itemName	Points to a null-terminated string that specifies the I/O point item name

See Also

Tag_GetAccessInfo, **Tag_New**, **Tag_SetAccessInfo**

TAGEVENTINFO

```
typedef struct {
                WORD bEnabled;
                WORD priority;
} TAGEVENTINFO, FAR* LP_TAGEVENTINFO;
```

Description

The TAGEVENTINFO structure contains information to set the event flags in a database tagname. The items in the structure are set and passed to the **Tag_SetEventInfo** function.

Element	Description
bEnabled	*bEnabled* is set to TRUE if event logging is enabled for this tagname
priority	Set to the event logging priority of this tagname

See Also

Tag_SetEventInfo

TAGINFO

```
typedef struct {
                WORD type;
                WORD accessType;
                WORD used;
                LPSTR name;
                LPSTR comment;
} TAGINFO, FAR* LP_TAGINFO;
```

Description

The TAGINFO structure contains information to set the basic information in a database tagname. The items in the structure are set and passed to the **Tag_SetInfo** function. The **Tag_GetInfo** function returns the data via the TAGINFO structure.

Element	Description
type	Specifies the type of tagname. See **Tag_New** for valid tagname types.
accessType	Access type, either memory, indirect or I/O

Value	Meaning
ACCESS_MEM	Tag is a memory tagname
ACCESS_DDE	Tag is a I/O type tagname
ACCESS_IND	Tag is an indirect tagname

used	Set to TRUE if used
name	Points to a null-terminated string that specifies the tagname
comment	Points to a null-terminated string that specifies the comment field

See Also **Tag_GetInfo**, **Tag_SetInfo**

TAGRETENTIVEINFO

```
typedef struct {
                WORD bValue;
                WORD bAlarmParams;
} TAGRETENTIVEINFO, FAR* LP_TAGRETENTIVEINFO;
```

Description

The TAGRETENTIVEINFO structure contains information to set the retentive flags in a database tagname. The items in the structure are set and passed to the **Tag_SetRetentiveInfo** function.

Element	Description
bValue	Set to TRUE if it's a retentive value
bAlarmParams	Set to TRUE if alarm parameters are saved

See Also **Tag_GetRetentiveInfo**, **Tag_SetRetentiveInfo**

TAGSCALEINFO

```
typedef struct {
                WORD inputConv;
                double minValue;
                double maxValue;
                double minRawValue;
                double maxRawValue;
                double deadband;
} TAGSCALEINFO, FAR* LP_TAGSCALEINFO;
```

Description

The TAGSCALEINFO structure contains information to set the scaling information in a database tab. The items in the structure are set and passed to the **Tag_SetScalingInfo** function.

Element	Description
inputConv	Specifies the input conversion
	For Discrete Tags
	DISC_CONVERT_REVERSE
	DISC_CONVERT_DIRECT
	For Analog Tags
	ANLG_CONVERT_LINEAR
	ANLG_CONVERT_SQRT
minValue	Specifies the minimum value
maxValue	Specifies the maximum value
minRawValue	Specifies the minimum raw value
maxRawValue	Specifies the maximum raw value
deadband	Specifies the value deadband

Comments

The *minValue*, *maxValue*, *minRawValue*, *maxRawValue* fields should be CAST appropriately to REAL or INTG, depending on the type of the tagname.

See Also

Tag_SetScalingInfo

VALALARMINFO

```
typedef struct {
                  WORD hiHiAlarmState;
                  WORD hiAlarmState;
                  WORD loAlarmState;
                  WORD loLoAlarmState;
                  WORD hiHiAlarmPriority;
                  WORD hiAlarmPriority;
                  WORD loAlarmPriority;
                  WORD loLoAlarmPriority;
                  REAL hiHiAlarmValue;
                  REAL hiAlarmValue;
                  REAL loAlarmValue;
                  REAL loLoAlarmValue;
                  REAL alarmDeadband;
} VALALARMINFO, FAR* LP_VALALARMINFO;
```

Description

The VALALARMINFO structure contains information for value alarm fields within a database tagname. Items in the structure are used to set fields in the database pertaining to alarms using the **Tag_SetValueAlarm**.

Element	Description
hiHiAlarmState	Specifies the HiHi alarm state
hiAlarmState	Specifies the Hi alarm state
loAlarmState	Specifies the Lo alarm state
loLoAlarmState	Specifies the LoLo alarm state
hiHiAlarmPriority	Specifies the HiHi alarm priority
HiAlarmPriority	Specifies the Hi alarm priority
loAlarmPriority	Specifies the LoLo alarm priority
loLoAlarmPriority	Specifies the Lo alarm priority
hiHiAlarmValue	Specifies the HiHi alarm value
hiAlarmValue	Specifies the Hi alarm value
loAlarmValue	Specifies the Lo alarm value
loLoAlarmValue	Specifies the LoLo alarm value
alarmDeadband	Specifies the deviation alarm deadband

See Also

Tag_SetValueAlarm

CHAPTER 8

Testing and Debugging Wizards

This chapter outlines the issues that should be considered when testing and debugging wizards. Even though the complexity of wizards can range from simple to extremely complex, there are several testing issues that need to be considered for every wizard. This chapter describes various tests that the Wizard developer should perform on all wizards and wizard DLLs once they are installed into WindowMaker. Performing these tests will ensure that your Wizards have been developed accurately and are completely functional.

This chapter also describes how to debug your Wizards using CodeView for Windows, Visual C++ Debugger, or by sending debug messages to the Wonderware Logger program.

Contents

Testing Guidelines for Wizards

We highly recommend that you perform the various tests described in this section to ensure that the wizards you have developed function properly after they are installed in WindowMaker.

Testing a Newly Installed Wizard

 After you have installed your new wizards in WindowMaker, perform the following test to ensure that they installed properly:

1. Click the Wizard tool on the **Wizard/ActiveX Toolbar** in WindowMaker. The **Wizard Selection** dialog box will appear.
2. Click on the name of the category for the new wizard(s). (The wizard should appear in area to the right of the category listing.)
3. Click on the new wizard(s) to verify that its description is correct.
4. Select each wizard and then, click **Add to toolbar** to add each wizard to the **Wizard/Active X Toolbar**.
5. Try adding wizards that are already in the toolbar. (If functioning properly, a message box will appear informing you that the wizard is already in the toolbar.)
6. Click **Cancel** to close the **Wizard Selection** dialog box and return to WindowMaker.
7. Check the **Wizard/ActiveX Toolbar** to verify that the newly added wizard(s) is there.
8. Click on the new wizard(s). Does its bitmap appear pushed in?
9. Move your mouse over the wizard in the toolbar. Does its tool tip description appear?
10. Undo and redo the creation of the wizard.
11. Remove combinations of wizards from the toolbar. In the **Wizard Selection** dialog box, select each wizard and then, click **Remove from toolbar**.
12. Close the **Wizard Selection** dialog box and return to WindowMaker to verify that the wizards selected for removal are no longer in the toolbar.

Testing Wizard Sizing

If the wizard was designed to be resized (most will be) the Wizard.DLL has to rebuild every object in the wizard based on the new size chosen by the user. To test a wizard's sizing, click on the wizard to select it then drag the selection handles to:

1. Size the wizard smaller in the horizontal direction.
2. Size the wizard larger in the horizontal direction.
3. Size the wizard smaller in the vertical direction.
4. Size the wizard larger in the vertical direction.
5. Size the wizard in a combination of these cases.
6. Size the wizard in a random sequence. You should be able to resize a wizard to an earlier size and have it achieve the same characteristics as the older size. Also, re-sizing a wizard should not cause the wizard to "enlarge" or "shrink" unexpectedly.
7. Verify that the aspect ratio of the wizard is maintained when the wizard is scaled.
8. When sizing the wizard, do text objects properly scale? Some text objects may be fixed or not related to the size of the wizard, while other text objects may be scaled as a ratio of the wizard.
9. Undo and redo the wizard's size and make sure the wizard properly refreshes. Pay special attention to any selection handles that should appear on the wizard. Select and move the wizard to assure that the handles and wizard have been properly refreshed.

Testing Wizard Editing Capabilities

Editing is allowed on wizards that have implemented the function **Wizard_Edit**. Editing a wizard simply changes its configuration properties. A "smart cell" rebuilds the wizard automatically based on the wizard's new configuration. Internal contents of a wizard may change based on the configuration of the wizard. To test the editing capabilities of a wizard, perform the following steps:

1. Edit the wizard by double-clicking on the wizard to display the wizard's configuration dialog box.
2. Use the TAB key to move from item-to-item in the dialog. Does the tabbing sequence follow Microsoft User Interface Guidelines?
3. Repeat step 2 using the arrow keys (especially within radio button groups). Make sure the arrow keys wrap around within the radio button group.
4. When the dialog box initially appears, is there an edit field or button that has properly been given focus? If the focus is on an edit field, is the entire contents pre-selected?
5. Does the dialog box have a proper title for the wizard being edited? Is it consistent? The style of the wizard dialog should be DS_MODALFRAME which will show up as a "fat" border. The wizard dialog should NOT have a system menu.
6. The dialog must be modal. The wizard dialog must not allow you to get to menus or windows of WindowMaker. Test this by trying to access WindowMaker. However, you should be able to switch to another application.
7. Verify the operation of the **Cancel** or similar operation. This operation should cause any modifications made in the dialog to have no effect.
8. Verify the operation of the **OK** or similar operation.
9. Undo and redo the wizard's editing and make sure the wizard properly refreshes. Pay special attention to any selection handles that should appear on the wizard. Select and move the wizard to insure that the handles and wizard have been properly refreshed.
10. Edit a wizard, but click only **OK**. Make sure the wizard size does not change. In general, clicking **OK** without making changes should not affect the wizard. The wizard should only change if a new version of the wizard has been installed. But once the wizard has changed based on the new version, it should not change if you reenter the edit dialog box and click **OK**.

Testing Wizard Configurations

For a wizard that supplies a configuration dialog box, you must verify the wizard in as many configuration combinations as possible. Test each of the following cases as applicable to the wizard:

1. Verify limit checking on any numeric entry field.
2. Verify maximum data entry in any text entry field.
3. Verify tagname checking on any field allowing tagnames. If the tagname field requires specific types, check to make sure only the proper types are allowed. When you double-click in the field does the tagname selection dialog appear?
4. Verify expression checking on any field allowing expressions. If the expression field requires specific types, check to make sure only the proper types are allowed. When you double-click in the field, does the proper WindowMaker dialog box for expression fields appear? The dialog box that appears may change based on the position of the cursor in relation to the text in the field. Enter real expressions.
5. Verify checking on numeric fields that are related to one another. For example, a minimum value field should not have a value larger than that of a maximum value field.
6. Verify that the color configurations appropriately effect objects that appear within the wizard. For example, a light might have an ON and OFF color. Make sure that the light draws properly after changing its configuration to either of the two colors. The light may actually work in WindowViewer, but in WindowMaker it may not appropriately show the right colors.
7. Verify that the color selection dialog box is used when changing colors. The color selection dialog box should be the WindowMaker standard dialog.
8. Test each configuration in WindowViewer. At the very least, verify that each field or selection in the dialog works.

Testing Toolbox Operations on a Wizard

A number of toolbar operations may be allowed for an individual wizard. These operations are optional for a wizard, but must not change based on modifications made to the wizard. These operations include:

Font Operations

- Bold, Italic, Underline
- Left Justify, Centered, Right Justify
- Reduce Font, Enlarge Font, Fonts. (Select Font)

Object Operations

- Line Color, Fill Color, Text Color

➢ **Do the following to test the toolbar operations for the wizard:**

1. Verify that the operations allowed for the wizard are properly enabled/disabled when the wizard is first placed.
2. Verify that the operations allowed for the wizard are properly enabled/disabled after the wizard has been resized.
3. Verify that the operations allowed for the wizard are properly enabled/disabled after the wizard has been edited.
4. Perform each of the Font operations on a wizard that has text objects upon which the Font operations can be applied.
5. Undo and redo the Font operations making sure the object properly refreshes.
6. Verify when performing Font operations, that all of the appropriate text objects change.
7. Check the "performance" of Font operations. In some cases the wizard may be unnecessarily performing internal text measurements on each text object within the wizard that cause the wizard to rebuild (refresh) slowly.
8. Perform each of the Object operations on a wizard that supports Object operations.
9. Undo and redo the Object operations making sure the object properly refreshes.

Special Wizard Tests

The following special tests should be performed as applicable for the wizard:

1. Verify the placeholders show for exported wizards that have tagnames, expression, or script properties.
2. Verify that none of the wizards "auto grow." This will happen if the wizard rectangle (passed to Wizard **Obj_New**) does not encompass all of the objects it contains (plus one pixel on each side).
3. Verify that the text in the wizard placeholders describes the correct type of tagname/expression that can be used with each Wizard. For example, ?d:discrete.
4. Verify that no memory leaks are present. Using the GDI reading, in ResLog, verify that the percentage of memory resources is not effected by repeated executions of the Window Open command.

Sending Debug Messages to the Wonderware Logger

The Wonderware Logger program is very useful for debugging and logging error condition messages, and so on. The Wizard developer can write debugging messages to the Wonderware Logger by using the WWDBG.LIB debug library supplied with the Wizard Toolkit.

To use the Wonderware Logger, the Wizard developer must add the debug.h file (located in the /INC directory of the Wizard Toolkit) and the WWDBG.LIB file to the project file.

To set the program name field in the Wonderware Logger window, the Wizard developer must call the initialization routing formatted as follows:

DebugInit (LPSTR library_name)

Note If you pass an invalid application name to DebugInit( ), the request will be ignored, and "UNKNOWN" will be placed into the program name field. No error message will be written to the Wonderware Logger. Valid application names are from 1 to 16 characters, and it is recommended that you do not use the / or % characters.

The string in library_name is truncated at 8 characters (length of space available in the logger window). (We recommend using the Wizard DLL name.) Place DebugInit in the DLL Startup routine (DLLMain for Windows NT), so that it is only executed once. We suggest placing it into the case for DLL_PROCESS_ATTACH.

```
int
WINAPI
DllMain( HANDLE hInstance, DWORD ul_reason_being_called,
LPVOID lpReserved )
{
    switch( ul_reason_being_called ) {
    case DLL_PROCESS_ATTACH:
        // Initialize needed globals
        hDrawInst = hInstance;
        hDrawWnd = FindWindow( "Wmak Class", NULL );
        DebugInit( (LPSTR) "WIZ1" );
        WWKit_Init();
        break;
    case DLL_THREAD_ATTACH:
        break;
    case DLL_PROCESS_DETACH:
        break;
    case DLL_THREAD_DETACH:
        break;
    default:
        break;
    }

    return 1;
}
```

To send a debug message to the logger, format the string as follows:

debug("format_string", variable list ...);

where: format_string is a format_string used by vsprintf.

For example:

```
debug("wizard index = %d", index);
debug("loaded bitmap %s", name);
```

Using CodeView to Debug the Wizard DLL

CodeView for Windows can also be used to debug a wizard DLL. When you use CodeView, be sure to remember to debug WindowMaker (WM.EXE) and load the DLL. Either load the DLL using the /l dll_name option on the cvw command line, or load it from the **Run** menu pull down under **Load**

There will be no symbols available for WindowMaker; however, there will be symbols for the wizard DLL. Breakpoints, source, watch variables, and so on, will be available just as if you were debugging an EXE. If there are no symbols for the wizard DLL, then check the build options on the project and ensure it was built for debug.

Using Visual C++ to Debug

Visual C++ includes a built-in debugger that makes it very simple to go straight from writing code to testing your wizards. Use the following steps to set up your Visual C++ environment for debugging.

1. Make sure you have built your Wizard DLL in 'Debug Mode'. To set this mode, on the **Options** menu select **Project** and then, select **Build Mode**. It is also important to set the switches. To do so, on the **Options** menu, select **Project** and then, select **Compiler**. A typical set of compiler switches to use for debugging is:

   /nologo /Gs /G2 /Zp1 /W4 /Z7 /AMw /Od /D "_DEBUG" /D "_WINDOWS" /D "_WINDLL" /FR /Gw

   A typical set of linker switches to use for debugging is:

   /NOLOGO /LIB:"libw" /LIB:"mdllcew" /LIB:"oldnames" /NOD /NOE /PACKC:61440 /SEG:256 /ALIGN:16 /ONERROR:NOEXE /CO /MAP:FULL

2. Since WindowMaker will be calling your Wizard DLL to use the wizard, you have to set up a 'Calling Program.' To do so, on the **Options** menu, select **Debug** and enter the full path to WindowMaker (for example, C:\Program Files\FactorySuite\INTOUCH\WM.EXE). Leave the **Debugging Mode** set to **Soft**.

3. Set a breakpoint in your code (a place to stop during execution) by placing your cursor on the desired line and selecting **Breakpoints**. Click **Add** to add the breakpoint to your list. You can set multiple breakpoints this way by adding to the list.

4. To start debugging, select **Go** on the **Debug** menu. Visual C++ will start WindowMaker normally. You want to open a window and use the Wizard (place it on the screen as you would normally). WindowMaker will call your Wizard DLL (and execute your wizard code). When the code execution gets to your breakpoint, Visual C++ will take the focus and the stop at that point. You can then step through the code using the options under the **Debug** menu (Step Into, Step Over, and so on)

When you are done debugging, select **Go** on the **Debug**menu and let the wizard DLL process to completion. It will then give control back to WindowMaker. You can then close WindowMaker normally.

CHAPTER 9

InTouch QuickScript Functions

Welcome to the InTouch QuickScript Functions Toolkit. By using the QuickScript Functions Toolkit, a proficient C programmer can develop and embed custom functions or subroutines which can be used either in InTouch QuickScripts or in expressions displayed on InTouch windows.

Powerful scripts can be developed within InTouch utilizing a rich set of conditional statements, functions, and data operators. For example, suppose an OEM wanted to provide an error message handling routine that maps an error code from a device to an error message. A simple C routine can be written to do the mapping and the InTouch application developer can call a function such as "ErrorMessage (errorNumber)" and retrieve an error message back. Furthermore, the script extension developer can write an interface to the Window Help function that would allow the InTouch user to bring up custom help screens by selecting the **Help** button and then select which function they desire help on.

Since the output of the QuickScript Functions Toolkit is a Windows DLL, anything that is callable through the standard Windows C interface is accessible through the QuickScript Extension Toolkit. In order to maximize the benefits that the Toolkit offers, it is important that you are capable of understanding and developing Window DLLs.

The QuickScript Functions Toolkit consists of documentation, samples and the necessary utilities to apply the InTouch QuickScripting feature.

Contents

Getting Started with the QuickScript Toolkit

WindowMaker recognizes the existence of a script DLL by reading files with a .WDF extension. This is an encrypted file that specifies the calling sequence of the function, help information, and paste link information.

Once the input definition file is created it needs to be converted from an input file to a .WDF file via the CRYPT utility program by using the following format:

CRYPT in file, out file /e.

For example:

CRYPT testfile.idf testfile.wdf /e

- Each function in the script DLL is specified on a separate line. These functions are called by WindowMaker and are integrated into WindowMaker in the same manner as other script functions.
- The function should follow this format:

  User Function name, Help file name, Help file index, Ignore Return Value, Script Flags, Paste Argument String, Paste Function, DLL Function name, DLL export name, Function Type, Return Type, Argument Types, and so on.
- A blank parameter must be separated by a space. For example: ", ," NOT ",,".
- The input definition file also contains a version information line. It appears on a separate line and must be of the format "Version=nnn". The version number is not case sensitive, however it is expected to be an integer number.
- Blank lines are allowed anywhere in the file *except for the last line*.
- Comment lines must begin with a semicolon ";".

The following lists the functions that are called by WindowMaker to access script functionality:

Parameter	Description
User Function name	Name of the function to the user.
Help file name	Name of .HLP file.
Help file index	Index in .HLP file.
Ignore Return Value	This function does not necessarily return a result. It only applies when the function is used in a script. If "Ignore Return Value" is set to 1, the result of the function does not need to be stored in a variable. If "Ignore Return Value" is set to 0, the result must be stored in a variable.
Script Flags	Flags For more information, see "Flags" later in this chapter.
Paste Argument String	Name of function to supply 'Paste Function and Arguments' string.
Paste Function	Name of DLL which supplies 'Paste Function and Arguments' function. If this is blank, it will default to the DLL that the function is in.
DLL Function name	Name of DLL function.
DLL export name	Name of export in DLL. This is the function name as defined in the C code.
Function Type	Calling type for function: PASCAL or C.
Return Type	Return parameter type, such as: INT LONG FLOAT DOUBLE STRING WORD DWORD VOID

Parameter	Description
Argument Types	The following parameter types are used when a constant or the value of a tag is passed to the function: INT LONG FLOAT DOUBLE STRING WORD DWORD The following types are pointer types. They are used when the function intends to pass back a result by changing one of the parameters. A tag must be provided when the user uses the function and is providing an entry for this argument: LP_INT LP_LONG LP_FLOAT LP_DOUBLE LP_STRING LP_WORD LP_DWORD No return is passed for the following parameter type: VOID This can only be used with C calling conventions: VARARG

Flags

The flag parameter is a hexadecimal DWORD, packed with information about a particular function. The lower word contains information about where and how a function can be used. The upper word contains sorting information for the script dialog procedures to categorize a function appropriately. The flags parameter is a combined value of all appropriate flags and types.

Functionality	Hex Value
Can be used in an expression or in QuickScripts	0x00000000
Use Only in QuickScripts	0x00000001
Display this function in Selection Box	0x00000008
This function is InTouch Standard	0x00000020

Type of Function	Hex Value
System Function	0x80000000
Recipe Function	0x40000000
SQL Access Function	0x20000000
SPC	0x10000000
Math Function	0x08000000
String	0x04000000
Miscellaneous Function	0x02000000

Note If you do not specify a function type, the function will not be displayed in the WindowMaker script editor dialog box.

Special Considerations

If your function causes the machine to be suspended for an extensive period of time, your InTouch application will also be suspended until the function has finished executing.

Also, do not use Windows API calls that contain model loops that dispatch messages or yields to other applications. (For example, MessageBox, DialogBox, DDEML calls, PeekMessage, DispatchMessage, and so on.) Doing so may result in WindowViewer processing a message in the middle of the execution of your script function.

Pasting Functions and Arguments

WindowMaker will automatically paste a "function help string" when the user selects it from a selection box. The DLL writer must supply a function or functions that return the syntax of the function and argument representations. WindowMaker will also highlight sections of the argument list for the user. The paste function must return character positions to mark the beginning and end of the section to highlight.

The function declaration would look like this:

```
/*
* DWORD ArgFunc( LPSTR funcName, LPSTR result);
* Note that the result buffer can only hold 100 characters.
*
* funcName is the name of the function that was chosen
* result is the result buffer
*
*/
```

The function file should always return a complete, syntactically correct string. This means a full and correct function name, beginning and ending parentheses if necessary, any arguments needed, and don't forget the semicolon, ';'.

For example:

SetDdeAppTopic(AccessName_Text, App_Text, Topic_Text);

These are some common keywords used in Paste Function Strings:

Window_Text would specify "window"

Tagname would be any tagname

Number would be any type of number.

Highlighting Replacement Values

The DWORD value returned by the Paste function consists of two values: the start and end position of a highlighted selection. Typically the developer of the DLL would highlight the first parameter of the function. The user is then set up to do the required editing after the paste. These values are zero based and do not include the end position.

For example:

ErrorMessage(errorNumber);

would return:

MAKELONG (14,25).

If the function does not require a parameter to be highlighted, return 0. If the values are the same, nothing will be highlighted but the cursor would be moved to that position.

Installing Your Script Extensions

Simply install the script function by copying the DLL and .WDF files to your InTouch directory. (The WDF file is the encrypted function definition file.) Anything can be done in the DLL that's allowable. Be warned that when the function is executed you have control of the processor and could tie up the system.

Sample Script

The following is a sample script function that returns an error message string for a given error message number.

```
LPSTR
WINAPI
ErrorMessage( int nErrorNumber )
{
    int    nLen;

    nLen = LoadString( hInst, nErrorNumber, (LPSTR) message,
        131);
    if (nLen <= 0){
        return ((LPSTR) "No message Found");
    } else {
        return (LPSTR) message;
    }
}
```

We will also supply a function for pasting the function name and arguments into the text of the WindowMaker script. This function is structured to accommodate pasting of strings for several functions.

```
DWORD
WINAPI
PasteBuiltInFuncs( LPSTR funcName, LPSTR result )
{
DWORD    hilite=0;
    int    i;

    lstrcpy( result, funcName );

    if( lstrcmpi(funcName, "ErrorMessage") == 0 ) {
        lstrcat( result, " ( errorNumber );" );
        hilite = MAKELONG(14, 27);
    }
    return( hilite );
}
```

WindowMaker will optionally call two routines, WWDllInit and WWDllFree, when it tries to load the DLL and before it frees the DLL. These routines do not have to be supplied but they can be useful if you need to have certain operations performed on startup or shutdown.

The function prototypes for the routines are as follows:

```
void WINAPI  WWDllInit (void);
void WINAPI  WWDllFree( void);
```

Our sample does not require any special initialization or shutdown treatment, so we'll just stub them out for completeness.

```
void
WINAPI
WWDllInit()
{
    /* do anything special that is required for the DLL  */
    return;
}


void
WINAPI
WWDllFree()
{
    /* clean up anything that needs to be cleaned up */
    return;
}
```

We will also need a DLL entry routine (DLLMain for 32-bit Windows). In this case, we need to save away our hInstance for use by LoadString.

```
int
WINAPI
DllMain( HANDLE hInstance, DWORD ul_reason_being_called,
LPVOID lpReserved )
{
    switch( ul_reason_being_called ) {
    case DLL_PROCESS_ATTACH:
        hInst = hInstance;      // save it for later
        break;
    case DLL_THREAD_ATTACH:
        break;
    case DLL_PROCESS_DETACH:
        break;
    case DLL_THREAD_DETACH:
        break;
    default:
        break;
    }

    return 1;
}
```

Of course, no DLL is complete without a DEF file. Since we only have one script function in this DLL, the DEF is very simple.

```
LIBRARY ERRMSG
DESCRIPTION 'Copyright Wonderware Software Corp, 2000'

EXETYPE WINDOWS

''
EXPORTS
    ErrorMessage        @1
    WWDllInit           @2
    WWDllFree           @3
    PasteBuiltInFuncs   @4
```

The RC file would contain the strings that would be returned for an individual error number.

```
#include    "windows.h"
STRINGTABLE
BEGIN
1,    "Low Battery"
2,    "Communication Failure"
3,    "Device mismatch"
4,    "Device I/O type mismatch"
5,    "Configuration mismatch"
6,    "System bus error"
7,    "Failed battery"
8,    "Stack is full or not there"
9,    "PLC watchdog timer timed out"
END
```

The IDF file (un-encrypted input definition file) would contain a version information line and one function definition line.

```
;
; ERRMSG.IDF
;
Version=1


ErrorMessage,errmsg.hlp,C030,1,0x20000028,PasteBuiltInFuncs,
        ERRMSG,ERRMSG,ErrorMessage,PASCAL,STRING,int
```

Combining the QuickScript Functions with IDEA

A powerful combination is to use the IDEA Toolkit API within the QuickScript Functions Toolkit. This is useful when reading large amounts of data from an external source, generic functionality, where the names of the tags are configured from an external source.

CHAPTER 10

IDEA Toolkit

Man-Machine Interface (MMI) applications perform some very specific computations involving data that is gathered and displayed by other MMI programs. InTouch has facilities to accomplish user-coded computations; however, the requirements of the application may preclude the use of InTouch built-in logic processing. Sometimes, a package of proven, proprietary algorithms must be incorporated into the overall MMI solution. For these situations, developers can use the InTouch Database External Access (IDEA) Toolkit.

For programmers using Visual Basic, an InTouch OCX is also included with the InTouch Extensibility Toolkit which greatly simplifies and streamlines the programming task. The OCX provides notification of data changes so that the programmer no longer needs to know the issues of polling in control applications.

Contents

Requirements

This toolkit is intended for use by experienced software developers. Mastery of the programming language and development tools is a prerequisite to the toolkit's use.

IDEA users who wish to develop a Windows program may work in Microsoft C/C++, Microsoft Visual C/C++, and Microsoft Visual Basic. Microsoft C users must have a Windows Software Development Kit if they are not using Microsoft Visual C++.

Note Users who wish to work in the 32-bit Windows environment must use the Microsoft Visual C++ development environment.

Summary of IDEA Options & Requirements

	Windows 95/98	Windows NT
Microsoft C/C++	Yes	Yes
Microsoft Visual Basic	Yes	Yes
*****Windows SDK Required	Yes	Yes
Type of Computer	Pentium	Pentium

***** Only if Microsoft Visual C++ is not used. Microsoft Visual C++ 6.0 Service Pack 3 or later is highly recommended.

IDEA Toolkit Contents

Once the IDEA Toolkit has been installed, the following files and directories are available:

\INTOUCH DIRECTORY ITEDIT.OCX	**\IDEA\SAMPLES\CIDEAAPP** Windows C++ 32-bit source code equivalent to Example #2 via CIdea.
\IDEA\LIB PTACC.LIB WWDBG.LIB WWHEAP.LIB	**\IDEA\SAMPLES\WINSIMPL** Windows C 32-bit source code for Example #2.
\IDEA\INC PTACC.H DEBUG.H WWHEAP.H	**\IDEA\SAMPLES\WINCMPLX** Windows C 32-bit source code for Example #3.
\IDEA\VB PTACC.BAS	**\IDEA\SAMPLES\ITAPP** Sample "*DOCALC*" InTouch application.
\IDEA\EXE Contains pre-built release-mode IDEA application samples.	**\IDEA\SAMPLES\VBSIMPLE\VB6** Visual Basic 6.0 32-bit example source for Window NT.
	\IDEA\SAMPLES\VBSIMPLE\VB6\ VB6ITAPP InTouch 7.1 sample to interface with Visual Basic 6.0 application.

Functional Description

An IDEA program that accesses InTouch data has the following structure:

1. Establish a connection (ACCID handle) with the InTouch runtime database by calling **PtAccInit**.
2. Prepare to access specific tagnames by calling **PtAccActivate** for each tagname to be accessed or changed. This creates HPT handles.
3. Read the specific tagnames that are needed as input for the algorithms by calling **PtAccReadD**, **PtAccReadI**, **PtAccReadR**, **PtAccReadA** or **PtAccReadM**.
4. Call **PtAccOK** to ensure that InTouch is still running and the tagnames that were read are still valid.
5. Perform computations as needed.
6. Write the results back to the InTouch tagname database by calling **PtAccWriteD**, **PtAccWriteI**, **PtAccWriteR**, **PtAccWriteA** or **PtAccWriteM**.
7. Repeat steps 3, 4, 5 and 6 as needed until time to shut down.
8. Shutdown by calling **PtAccShutdown** and exit.

Though the structure outlined in the previous steps is the simplest application of the IDEA Toolkit, it is probably adequate for the majority of users' needs. Other features are available to allow a more sophisticated program structure. Some of these are:

- A Windows program written with Microsoft Visual C++ can register with InTouch to be notified immediately when specific tagnames change values
- Refer to **PtAccActivateAndNotify** and **PtAccActivateAndSendNotify**

Special Data Types

Some special data types are used with the IDEA Toolkit functions. They are defined in the files appropriate for each supported language.

ACCID — This is an Access Id. It is used in most functions to identify a connection to the InTouch database. A program must have at least one Access Id to be able to access the database. The program may have more than Access Id. Each variable accessed is associated with an Access Id.

HPT — This is a Point Handle. When a program accesses a variable in the InTouch database, it must first convert the variable name (which is a character string containing the name of any valid tagname or field) into a handle for efficient use. Each point to be accessed must have a Point Handle. These handles are used throughout the program to read and write variables.

PTYPE — This is a Point Type. **PtAccType** returns a value of this type to indicate the type of InTouch variable. NULL is use to indicate an invalid or unknown type of point. There are four valid codes for PTYPE:

PT_DISCRETE

PT_INTEGER

PT_REAL

PT_STRING

```
#include "ptacc.h"

ACCID    AccId;
HPT      SecondsHandle;
PTYPE    SecondsType;
long     IntSeconds;
float    FloatSeconds;
char     MsgSeconds[132];

/*  The first thing needed is an ACCID. This is provided by
     the function PtAccInit.*/

AccId = PtAccInit( hWnd, 8 );
```

```
/* To be able to access an InTouch variable, one must supply
the variable's name so that it can be converted into a
shorthand reference. This is the HPT. PtAccActivate takes as
parameters an ACCID and a variable name and returns the HPT
(point handle.)  All other IDEA functions work with ACCID and
HPT.*/

SecondsHandle = PtAccActivate( AccId, "$Second" );

SecondsType = PtAccType( AccId, SecondsHandle );
switch( SecondsType ) {
    case PT_DISCRETE:
        /* $Second as a discrete?  Doesn't make sense */
        break;
    case PT_INTEGER:
        IntSeconds = PtAccReadI( AccId, SecondsHandle );
        break;
    case PT_REAL:
        RealSeconds = PtAccReadR( AccId, SecondsHandle );
        break;
    case PT_STRING:
        PtAccReadM( AccId, SecondsHandle, MsgSeconds,
        sizeof(MsgSeconds) );
        break;
}
```

Access ID Handles (ACCID)

The first step is to create one or more ACCIDs through calls to **PtAccInit**. This step is necessary because it allows the IDEA functions to determine whether InTouch is running on the computer and is accessible. It also makes it possible to run multiple programs, simultaneously accessing InTouch data. Every other function and each of the following steps depends on a unique ACCID.

If **PtAccInit** returns a NULL handle it means that InTouch is not accessible (not running).

Only one ACCID is required for a program to use InTouch variables. However, it may be useful to work with more than one ACCID. As described in the next section, each Point Handle (HPT) you create is connected to an ACCID. This provides a form of logical grouping of variables.

Point Handles (HPT)

The second step to access InTouch data is to set up Point Handles (HPTs) for all of the InTouch variables that the program must read or write. In setting up Point Handles, two important things are accomplished. 1) The InTouch variable or field name is sent to InTouch to be validated, 2) If the name is valid, a handle is established. The handle makes every access to the variable speedy by eliminating the need to look up the variable name and validate it each time. Successfully creating an HPT indicates that the variable is valid. The point must be activated before it can be read.

Activating Variables

The InTouch runtime database keeps track of which of its tagnames are in use. A tagname is considered to be in use when:

- It is used in an animation link for an object in a visible window
- It is trended
- It is alarmed
- It is used in InTouch logic
- It is used in an action script for an object in a visible window
- It has been the object of an Advise request from an I/O client
- It has been activated by a program developed with the IDEA Toolkit

Some InTouch variables originate with other programs in the system. For example, some come from external equipment through I/O Server programs. Other variables may come from a wide variety of Windows applications such as Microsoft Excel spreadsheets. When this type of variable is in use, InTouch requests the originating program to notify InTouch of every change in its value. When these variables are not in use, InTouch cancels the request. An IDEA program must activate each variable to ensure that when it reads a variable, it receives the current value. The IDEA functions **PtAccActivate** and **PtAccHandleActivate** activate variables.

If there are periods during which the program does not need some or all of the variables, it is good practice to notify InTouch that they are not in use by calling **PtAccHandleDeactivate** or **PtAccDeactivate**. This allows InTouch to notify any involved I/O servers that these variables are not needed and polling can be suspended. When these variables are, in fact, needed, they must be activated again by calling **PtAccHandleActivate**.

Note **PtAccActivate** should not be used to reactivate a point because it creates a new HPT as well as activating it.

InTouch Variable Types

The InTouch tagname database contains data of four basic types: discrete, integer, real (or floating point) and character strings. A discrete tagname has only two possible values, 0 (off or false) or 1 (on or true.) An integer tagname is a 32-bit signed number ranging between -2,147,483,648 and +2,147,483,647. Real (floating point) tagnames are 32-bit IEEE floating point numbers ranging between -3.4e+38 and +3.4e+38. Character string tagnames are null-terminated (C language standard) ASCII arrays up to 131 8-bit characters in length.

The function **PtAccType** allows the user to determine the type of a tagname. Knowing the type, the user can call the appropriate functions to read and write the tagname. Another pair of function calls, **PtAccReadA** and **PtAccWriteA**, can be used with discrete, integer or real tagnames. The value of the tagname is converted to double-precision floating point by **PtAccReadA** and converted from double-precision floating point to the appropriate type by **PtAccWriteA**.

Of course, the IDEA program can know ahead of time the type of each variable it uses. If so, the appropriate read and write functions can be called without checking **PtAccType**. If this is the case, it is good practice to verify the type of each variable when the HPT is created.

Reading InTouch Variables

After creating an ACCID, creating an HPT for each variable and activating each HPT, values can be read from InTouch. IDEA has five functions for reading InTouch data:

Function	Description
PtAccReadD	Reads an InTouch discrete, integer or real variable and returns it as a discrete value.
PtAccReadI	Reads an InTouch discrete, integer or real variable and returns it as an integer value.
PtAccReadR	Reads an InTouch discrete, integer or real variable and returns it as a 32-bit IEEE floating point value.
PtAccReadA	Reads an InTouch discrete, integer or real variable and returns it as a 64-bit IEEE floating point value.
PtAccReadM	Reads an InTouch string variable.

Writing InTouch Variables

After creating an ACCID and an HPT for each variable, values can be written to InTouch. IDEA has five functions for writing InTouch data:

Function	Description
PtAccWriteD	Writes a discrete value to InTouch discrete, integer or real variable.
PtAccWriteI	Writes an integer value to an InTouch discrete, integer or real variable.
PtAccWriteR	Writes a 32-bit IEEE floating point value to an InTouch discrete, integer or real variable.
PtAccWriteA	Writes a 64-bit IEEE floating point value an InTouch discrete, integer or real variable.
PtAccWriteM	Writes an InTouch string variable.

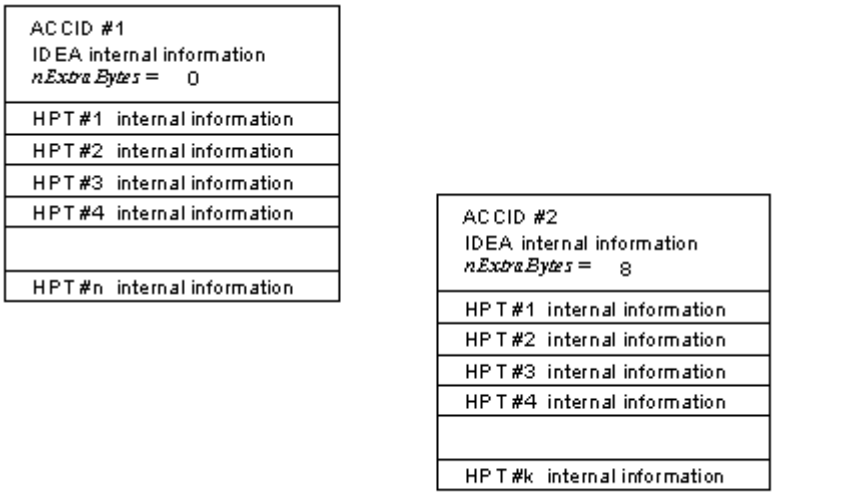
Detecting InTouch Exits

In most cases, the program must detect when InTouch exits. The function of the IDEA program may be invalid when InTouch is not running. This can be accomplished by calling **PtAccOK**. The recommended use if **PtAccOK** is to call it once after each group of variables is read before calculating and writing the results back to InTouch. This technique is shown in the sample programs.

Storing Program Data with Each HPT

IDEA programs can store and retrieve a virtually unlimited amount of extra information associated with each InTouch variable being accessed. For every variable being accessed, IDEA will allocate storage in the amount requested by the program. The user specifies the amount of storage needed in the call to **PtAccInit**. The functions **PtAccSetExtraInt**, **PtAccSetExtraLong**, **PtAccGetExtraInt** and **PtAccGetExtraLong** allow the program to store and retrieve information from that storage space.

Conceptually, IDEA maintains information about the ACCIDs and HPTs as follows:



In this case, when the user created the second ACCID, IDEA was specified to allocate 8 extra bytes for each HPT. The user can store and retrieve information in the 8 bytes allocated. It is up to the user to decide how to use the storage area. Example #3 in the "Program Examples" section of this manual shows how this feature can be used.

Differences Between 16 and 32-Bit Compilers

When using 16-bit compilers, the size of an *integer* is two bytes and a *long* is four bytes. When using a 32-bit compiler, the size of each data type is doubled. Hence, an *integer* is four bytes and a *long* is eight bytes. This is especially important when you are porting an IDEA application to 32-bit Windows 95/98 or 32-bit Windows NT. The following table shows the bytes required for each function under different compilers. Example #5 further clarifies the differences between compilers.

Note Version 7.1 (or later) of the IDEA Toolkit only supports 32-bit Windows development.

	Bytes Needed	
Function	**16-bit Windows**	**32-bit Windows**
PtAccSetExtraInt PtAccReadExtraInt	2	4
PtAccSetExtraLong PtAccReadExtraLong	4	8

Tag Handles and Memory Usage

This section describes how to best use tag handles so that their implementation does not cause an unnecessary strain on system resources. Certain implementations of using the IDEA Toolkit to keep the handles to a large number of tags active can cause WindowViewer to slow down, and to even lock up. You may see "RDB Command Not Processed" errors, Assertion errors, and even Access Violation errors, which can cause WindowViewer to terminate.

Note "RDB Command Not Processed" errors, Assertion errors, and Access Violation errors are often caused by other problems, such as a corrupted InTouch application. An IDEA application may not always be at fault.

A typical IDEA application has a program flow that resembles the following:

```
// Global variable declarations
ACCID g_accid= NULL;
HPT g_hpt[SomeNumber];
// Some function that is used for application initialization.
void InitializationRoutine()
{
UINT ui= 0;
wwHeap_Register(NULL, &ui);
g_accid= PtAccInit(NULL, 0);
for(int i= 0; i< SomeNumber; i++)
{
g_hpt[i]= PtAccActivate(g_accid, "SomeTagnames");
}
// Some other app initialization here.
}
// Some function that is used for processing data from View.
void DataProcessingRoutine()
{
for(int i= 0; i< SomeNumber; i++)
{
PtAccWriteI(g_accid, g_hpt[i], SomeValue);
}
}
// Some function that is used to clean up tag handles and
InTouch connection.
void ShutdownRoutine()
{
for(int i= 0; i< SomeNumber; i++)
{
PtAccDeactivate(g_accid, g_hpt[i]);
PtAccDelete(g_accid, g_hpt[i]);
}
PtAccShutdown(g_accid);
WwHeap_Unregister();
}
```

In some instances, PtAccHandleCreate, PtAccHandleActivate, PtAccHandleDeactivate, and PtAccHandleDelete may also be used, but the effect is the same.

This method of implementation is fine for a small amount of tags. However, it will not work for a large amount of tags due to memory allocation and access within the WWHeap.dll. Each time a handle for a tag is created (that is, calling **PtAccActivate**, **PtAccHandleCreate**, and so on), a chunk of memory is allocated in WWHeap for that handle. The more handles that are created, the more memory is allocated and the more messages that need to be processed by WWHeap.

At some point, a threshold will be reached where there are not enough system resources to process the messages and the application will hang or eventually crash. This threshold is difficult to define. It depends on a number of variables, including the CPU speed, available memory, number of tag handles requested, and InTouch application resources that are required (that is, the more scripts, animation, and open windows it has, the more resources it will need).

For example, you have an InTouch application with 11,000 Discrete tags that are being monitored for alarms and it uses a distributed alarm system. If all of the tags are being updated at the same time (such as in the For Next loop in the sample script), a problem may result. On the other hand, if an InTouch application is large (that is, there are hundreds of windows and scripts), then the problem may occur with as few as 1,000 tags being updated at once.

The problem is not the result from updating the tags. Instead, it usually occurs when there is an update to a number of tags and many tags are currently activated within the IDEA toolkit application. This is because only one process at a time can use WWHeap. Though many applications can connect to WWHeap and use it, WWHeap will only perform management functions for one process at a time. One or more of the following may add to the load encountered by WWHeap, and may render it useless until it has finished processing its message queue: many handles that are allocated for large or busy InTouch applications, lots of tag handle updates, or other InTouch processes, such as running scripts, opening windows, and using history or alarms.

The best way to avoid this situation is to keep the handles active only for tags that are processed frequently during the execution of the IDEA application. If a tag is only going to be used occasionally, create the handle and then delete it after processing for that tag is complete, as shown in the following script:

```
// Global variable declarations
ACCID g_accid= NULL;
HPT g_hpt= NULL;
// Some function that is used for application initialization.
void InitializationRoutine()
{
UINT ui= 0;
wwHeap_Register(NULL, &ui);
g_accid= PtAccInit(NULL, 0);
// Some other app initialization here.
}
// Some function that is used for processing data from View.
void DataProcessingRoutine()
{
for(int i= 0; i< SomeNumber; i++)
{
g_hpt= PtAccActivate(g_accid, "SomeTagnames");
PtAccWriteI(g_accid, g_hpt, SomeValue);
PtAccDeactivate(g_accid, g_hpt);
PtAccDelete(g_accid, g_hpt);
}
}
// Some function that is used to clean up tag handles and
InTouch connection.
void ShutdownRoutine()
{
PtAccShutdown(g_accid);
WwHeap_Unregister();
}
```

Accessing Remote Tags

When attempting to read a tag handle (obtained through a remote tag), sufficient time must be allowed in order for a current value to be obtained before the tag handle is deactivated and/or deleted. For example, it is common practice to create a handle, activate it, read it's value, deactivate it, then delete it all within the same API:

```
void ProcessTags()

{
HPT hpt= PtAccActivate(accid, "AccessName:Item");
SaveValue(PtAccReadI(accid, hpt));
PtAccDeactivate(accid, hpt);
PtAccDelete(accid, hpt);
}
```

If this type of script is executed for a remote tag, it is possible that the current value will not be retrieved, since the current value may not have been obtained from the I/O Server before the handle was deleted.

In order to ensure enough time to retrieve the value, you could insert a delay between the time that the tag handle is read, and the time it is deactivated/deleted. You could also create the tag handle and then let it remain active during the entire life of the application. However, this is not recommended since it has the potential to slow down WindowViewer.

Program Examples

The following set of examples is provided to illustrate how the IDEA Toolkit can be used. The following samples are written in no particular language. Their purpose is to illustrate the ideas behind the functions and various ways to use the functions.

In addition, some of the examples provided are complete and working. Sample #2 is provided for all supported environments: Windows C/C++ and Visual Basic. Sample #3 is provided for the Windows C/C++ environment. To demonstrate the working sample programs, we have also included a simple InTouch application.

Example #1

In this trivial example, we show an access to the InTouch database which just reads and displays the value of the message variable $TimeString.

```
accID = PtAccInit( 0, 0 );
if accID = 0  then
    { InTouch is not running }
    print "InTouch is not running"
else
    { InTouch is active ... try to activate $TimeString }
    hPtTime = PtAccActivate( accID, "$TimeString" );
    if hPtTime <> 0  then
        { InTouch recognizes $TimeString ... read the current
            value }
        PtAccReadM( accID, hPtTime, timeString );

        { Display the current value }
        print "Current time in InTouch is " timeString;
    endif

    { Shut down the connection to InTouch }
    PtAccDeactivate( accID, hPtTime );
    PtAccShutdown( accID );
endif
```

Example #2

In this example, we show realistic sample that reads three points and performs a calculation based on those points whenever InTouch signals the need to do the calculation. After calculating the result, it is returned to InTouch. When InTouch wants to do the calculation, it sets the variable *DoCalc* to TRUE and expects *DoCalc* to be set FALSE when the operation is complete. While this example uses a Boolean variable, *DoCalc*, to determine when to do the calculation, it could just as well have performed the calculation every n seconds, or when an analog value exceeded some limit, or as often as possible.

```
{ Initialization }

accID = PtAccInit( hWnd (NULL), 0 );
if accID = 0  then
    print "InTouch not active"
    stop;
endif

hPtInput1 = PtAccActivate( accID, "Input1" );   { real }
hPtInput2 = PtAccActivate( accID, "Input2" );   { real }
hPtInput3 = PtAccActivate( accID, "Input3" );   { real }
hPtResult = PtAccActivate( accID, "Result" );   { real }
hPtDoCalc = PtAccActivate( accID, "DoCalc" );   { discrete }

{ check that each of the hPtInput1 ... hPtDoCalc are non-NULL
If PtAccOK() returns FALSE, InTouch has been stopped and we
must shut down}
while( PtAccOK( accID ) )  do
    if PtAccReadD( accID, hPtDoCalc ) = TRUE  then
        { InTouch wants us to do the calculation }
        input1 = PtAccReadA( hPtInput1 );
        input2 = PtAccReadA( hPtInput2 );
        input3 = PtAccReadA( hPtInput3 );
        if PtAccOK( accID )  then
            { InTouch still running OK, do the calculation }
            result = input1 * input2 * input3;
            { store the result to InTouch }
            PtAccWriteA( accID, hPtResult, result );
            { tell InTouch that we're done}
            PtAccWriteD( accID, hPtDoCalc, FALSE );
        endif
    endif
endwhile
{ shut down the connection to InTouch }
PtAccShutdown( accID )
```

Example #3

This is a Windows example that only updates the screen when values change. It illustrates calls to **PtAccHandleCreate**, **PtAccHandleActivateAndNotify**, **PtAccType**, **PtAccSetExtraInt**, **PtAccGetExtraInt**, and **PtAccACCIDFromHPT**. Realize that this is not a trivial example, but for Windows development, it is important to understand the concepts applied in this example.

In this example, we assume that the user enters the names of two tagnames that he wishes to display. We will assume that these tagnames are in an array of structures called tagInfo[ ], see the following Declarations code. As an additional feature, if the user minimizes the program window, we will tell WindowViewer to deactivate the points, and will reactivate the points when the window is on the screen again.

It's worth reviewing the code that uses **PtAccSetExtraInt** and **PtAccGetExtraInt**. At initialization time, when we call **PtAccInit**, we tell it to allocate 2 extra bytes of information for each HPT. As we create handles using **PtAccHandleCreate**, we save the index into our array of tagInfo in the extra bytes allocated with HPT, using **PtAccSetExtraInt**. When IDEA notifies us of a change using the dbChgMsg message, it passes the HPT in the lParam of the message. First, we obtain the corresponding ACCID by a call to **PtAccACCIDFromHPT** and then obtain the array index by calling **PtAccGetExtraInt**. The result of this is that we ask IDEA to save our index for us so that we don't have to search for HPT in our data structures. In this example, with only 2 points, it would not have been a problem, but in a large system, it's easy to see the time savings.

```
{ Declarations }

struct {
    char     ti_tagname[100];
    HPT      ti_hPt;
    PTYPE    ti_ptType;
    char     ti_valueString[132];
} TAGINFO;

TAGINFO    tagInfo[ 2 ];
ACCID      accID;

{ Initialization }

accID = PtAccInit( hWndParent, 2 );
{Request 2 extra bytes per hPt}
if accID = 0  then
    print "InTouch not active"
    stop;
endif

{ Register the "DBCHGMSG" defined in ptacc.h with Windows }
dbChgMsg = RegisterWindowMessage( DBCHGMSG );

{ Get the tagnames for the 2 tagInfo structures;  Use WIN.INI
for simplicity in this example}
GetProfileString( "Sample", "Tag1", tagInfo[0].ti_tagname, ...
    "$TimeString");
GetProfileString( "Sample", "Tag2", tagInfo[1].ti_tagname, ...
    "$Second");
```

```
{ Create handles for the 2 tagnames }
for i=0 to 1  do
    tagInfo[i].ti_hPt = PtAccHandleCreate( accID,
        tagInfo[i].ti_tagname );
    if tagInfo[i].ti_hPt = 0   then
        ErrorMsg( "Cannot find tagname", tagInfo[i].ti_tagname
);
        stop
    else
        { remember the point type }
        tagInfo[i].ti_ptType = PtAccType( accID,
            tagInfo[i].ti_hPt);

        {Activate the point [assumes initial window is not
            iconic] }
        PtAccHandleActivateAndNotify( accID, tagInfo[i].ti_hPt
);

        { Remember the index in the "extra" bytes for the
            point }
        PtAccSetExtraInt( accID, tagInfo[i].ti_hPt, 0, i );

        { Get the initial value }
        UpdatePointValue( i );    { code shown later }
    endif
endfor

{ Support Routines }

UpdatePointValue( n ):
    { This code is called at initialization time to get an
        initial value and is also called each time we're
        notified that the point value has changed }

switch( tagInfo[ n ].ti_ptType )  {
case PT_DISCRETE:
    { Put result of PtAccReadD() into the string
        tagInfo[n].ti_valueString }
    sprintf( tagInfo[ n ].ti_valueString, "%d",
        PtAccReadD( accID, tagInfo[ n ].ti_hPt ) );
    break;
case PT_INTEGER:
    { Put result of PtAccReadI() into the string
        tagInfo[n].ti_valueString }
    sprintf( tagInfo[ n ].ti_valueString, "%ld",
    PtAccReadI( accID, tagInfo[ n ].ti_hPt ) );
    break;
case PT_REAL:
    { Put result of PtAccReadR() into the string
        tagInfo[n].ti_valueString }
    sprintf( tagInfo[ n ].ti_valueString, "%f",
    PtAccReadR( accID, tagInfo[ n ].ti_hPt ) );
    break;
case PT_STRING:
    { Put result of PtAccReadM() into the string
        tagInfo[n].ti_valueString }
    PtAccReadM( accID, tagInfo[n].ti_hPt,
    tagInfo[n].ti_valueString, 132);
    break;
endswitch

RepaintScreen(); { code shown later }

RepaintScreen():
for i=0 to 1 do
    ShowText( line #i, tagInfo[i].ti_valueString );
endfor
```

```
{ Windows Message Processing }

WM_PAINT message:
    RepaintScreen();

dbChgMsg message:
    { lParam of the dbChgMsg contains the hPt that was changed
}
    hPt = lParam;

    { Ask IDEA for the accID that corresponds to this hPt }
    accID = PtAccACCIDFromHPT( hPt );

    { Now that we have the accID and hPt, we need to know the
        index into our array of points. We could search our
        tagInfo[] array. However, when we initialized, we
        asked IDEA to remember the index in each hPt by
        calling PtAccSetExtraInt(). Now, by calling
        PtAccGetExtraInt() we can retrieve the index without
        any lookup}
        n = PtAccGetExtraInt( accID, hPt, 0 );

    { Update the value and the screen }
        UpdatePointValue( n );

WM_SIZE message:
if  wParam is SIZENORMAL or SIZEFULLSCREEN and the windows is
    ICONIC then
    { User is looking at screen again ... activate all the
       points}
    for i=0 to 1 do
        PtAccHandleActivateAndNotify( accID, tagInfo[i].ti_hPt
);
    endfor
endif

if  wParam is SIZEICONIC  then
    { User is not looking at our points ... deactivate all
      the points }
    for i=0 to 1 do
        PtAccHandleDeactivate( accID, tagInfo[i].ti_hPt );
    endfor
endif
```

Example #4

For C++ programmers there is a class titled CIdea that encapsulates programming in the IDEA Toolkit. It is the core of implementation for the ITEdit.OCX. An example application highlighting its correct usage is found in the CIDEAAPP sample. This sample application is the equivalent of WINSIMPL functionally, but is implemented via the CIdea C++ class.

Example #5

Example #5 shows a test sample using one integer and one long under 32-bit.

```
   // test sample using one int and one long under 32-bit
   // initialize (need 4 bytes for the int and 8 for
   // the long)
   AccID = PtAccInit( hWnd, 4+8 ); // sizeof(int) +
sizeof(long)
   // register a tag
   hPt = PtAccActivate( AccID, "TestTag" );
   // write 15 to the extra int (first data offset 0)
   PtAccSetExtraInt( AccID, hPt, 0, 0xf );
   // write 15 to the extra long (make room for the int)
   PtAccSetExtraLong( AccID, hPt, 4, 0xf );
```

IDEA Programs in the Windows NT Environment

Writing IDEA programs for 32-bit Windows is the same as writing IDEA programs for 16-bit Windows with the following exception: IDEA (PTACC.DLL) uses the Wonderware heap management facility (wwHEAP). If you are writing a standalone application that uses the IDEA interface, before calling **PtAccInit**, you must register your application with the Wonderware heap manager using the **wwHeap_Register** function. You must also unregister your program before it exits using the **wwHeap_Unregister** function. The function prototypes are:

```
// call at the beginning of the C or C++ program
// before any other calls are made to PtAcc
#ifdef_DEBUG
wwHeap_RegisterEx( hWnd, pwMsgNotify,_FILE_,_LINE_);
#else
wwHeap_Register(hWnd, pwMsgNotify);
#endif
// All your other InTouch and PtAcc code goes here…
// Call at the end of the C or C++ program
#ifdef_DEBUG
wwHeap_UnregisterEx(_FILE_,_LINE_);
#else
wwHeap_Unregister();
#endif
```

Function prototypes:

```
BOOL  WINAPI wwHeap_Register( HWND hWnd, UINT *wMsgNotify);
BOOL  WINAPI wwHeap_Unregister();
BOOL  WINAPI wwHeap_RegisterEx( HWND hWnd, UINT *wMsgNotify,
             LPCTSTR szFile, int iLine);
BOOL  WINAPI wwHeap_UnregisterEx(LPCTSTR szFile, int iLine);
```

The function prototypes are defined in #include wwheap.h. All 32-bit IDEA Samples use the **wwHeap_Register** and **wwHeap_Unregister** functions.

Note If you are using IDEA in an InTouch Script or Wizard DLL, it is not necessary to use the **wwHeap_Register** and **wwHeap_Unregister** functions.

Note Under 32-bit Windows, DLLs cannot reside in multiple places on the hard disk. You must provide a search path to the InTouch directory so your IDEA application can locate it. If a duplicate PTACC.DLL or WWHEAP.DLL resides on your system, 32-bit Windows will attempt to create another instance of this DLL and your IDEA application will generate Assertion Errors in WWHEAP.

InTouch Notification of Tag Changes

The following example scenario shows the difference between the **PtAccActivateAndNotify**, **PtAccHandleActivateAndNotify** and **PtAccActivateAndSendNotify PtAccHandleActivateAndSndNotify** functions and how they affect InTouch performance.

PtAccActivateAndNotify and PtAccHandleActivateAndNotify

These functions cause InTouch to send notification via PostMessage. InTouch will continue to post messages into your IDEA application's window message queue until it exits it's message loop (releases the system processor). You may receive multiple notifications before gaining control of the processor to handle them.

Example:

InTouch user clicks a button assigned directly to a discrete tagname:

InTouch enters message loop

Tag goes High (1) and then Low (0)

InTouch posts notifications of both changes to IDEA Application

InTouch exits message loop

IDEA App enters message loop

IDEA App processes first notification and reads value, it will get 0

IDEA App exits message loop

IDEA App enters message loop

IDEA App processes second notification and reads value, it will get 0

IDEA App exits message loop

This occurs because InTouch has updated the tagname before the IDEA application can read the value. If the values contained in the tags remain static until the IDEA app processes them, then this is a good function to use. If your IDEA application requires real-time update of data, use the function **PtAccActivateAndSendNotify** or **PtAccHandleActivateAndSndNotify**.

PtAccActivateAndSendNotify and PtAccHandleActivateAndSndNotify

These functions cause InTouch to send notification via the Windows API SendMessage function. InTouch will post a message at the beginning of your IDEA application's window message queue and then wait the IDEA application processes the message. Once this happens, your application will have full control of the processor until you exit your message loop. The example scenario describes this in further detail.

Example:

InTouch user clicks a button assigned directly to a discrete tagname.

InTouch enters message loop

Tag goes High (1)

InTouch posts notification of change to IDEA App

InTouch goes to sleep

IDEA App enters message loop

IDEA App processes notification and reads value, it will get 1

IDEA App exits message loop

InTouch wakes up

Tag goes Low (0)

InTouch posts notification of change to IDEA App

InTouch goes to sleep

IDEA App enters message loop

IDEA App processes notification and reads value, it will get 0

IDEA App exits message loop

InTouch wakes up

InTouch exits message loop

InTouch gives the IDEA application control after notification, then you are ensured that the value read from the tagname is the value you received.

Anything requiring a long time-slice (large table look-ups, database request waits, file I/O, and so on), should not be done during this cycle. This will cause erratic performance of InTouch (waiting for your application to exit its message loop). If you must perform these types of functions, post a USER message or registered windows message into your IDEA application's message queue (PostMessage) as a response to the InTouch notification and perform this operation when your application's private message has been received.

Note The use of these functions with tags that change frequently could cause your application's buffer to overflow. If this occurs, a message will be posted in the Wonderware Logger each time the event occurs. It is possible to increase your application's buffer to 120 using the following code segment placed in your **WinMain** function:

```
{
    int MsgQuesize = 120;

    while( MsgQueSize && !SetMessageQueue(MsgQueSize) )
        MsgQueSize -= 8;
    If( !MsgQueSize ) return(FALSE);
}
```

If the overflow error still occurs, set an alarm state and using the *<MYTAG>.Alarm* field.

Use of Environment Variables

Compilers and most software development tools are able to use environment variables to find the files that are needed. When installing the toolkit, keep the following in mind:

PATH	This environment variable is used by Windows NT to locate executable files (programs) when the file is not found in the current working directory. The PATH variable contains a sequence of directories to search. Your PATH should contain both the Windows NT directory and the InTouch directory (usually C:\INTOUCH.) Throughout this section, executable files will be copied to the C:\INTOUCH directory. If you wish to use a different directory, you must be certain that the directory is in your PATH.
INCLUDE	This environment variable is used by most compilers and assemblers to locate files accessed through INCLUDE directives. Throughout this section, INCLUDE files will be copied to C:\INCLUDE. If you wish to use a different directory, be sure it is named in the INCLUDE environment variable. For example, in AUTOEXEC.BAT: SET INCLUDE=C:\INCLUDE
LIB	The object code linker uses the LIB environment variable to locate object code libraries that are not found in the current working directory on the default disk drive. Throughout this section, library files will be copied to C:\LIB. If you wish to use a different directory, be sure it is named in the LIB environment variable. For example, in AUTOEXEC.BAT: SET LIB=C:\LIB

To proceed with installation, skip to the sections that address your target environment.

Installing for Microsoft C in a Windows NT Environment

Requirements

- Microsoft Visual C++ 6.0 Service Pack 3
- Microsoft Windows Platform SDK
- Microsoft Windows NT
- Wonderware InTouch 7.1 (or later)
- Pentium machines supported

Support

Windows NT for the Intel Platforms is supported

Files Required for Development

PTACC.LIB	must be in LIB path (for example, C:\LIB)
PTACC.H	must be in INCLUDE path (for example, C:\INCLUDE)

Samples

\SAMPLES\WINSIMPL	Sample #2
\SAMPLES\WINCMPLX	Sample #3
\SAMPLES\CIDEAAPP	Sample #4

Microsoft Visual C++ MFC CIdea C++ sample which functions the same as Sample #2. The paths in the INCLUDE (C:\INTOUCH\SCRIPT\INC) and LIBRARY (C:\INTOUCH\SCRIPT\LIB) environment variables must be set before being built. Once done, all dependencies must be regenerated (both release and debug). After building the application, it must be copied to the InTouch directory to run correctly.

Running IDEA Toolkit Samples

Before proceeding with the samples, ensure that you have installed InTouch 7.1 or later and have installed the IDEA Toolkit.

Step 1

For any of the samples delivered on the IDEA Toolkit, you must run InTouch WindowViewer with the application supplied. Run View using this application before starting any of the samples.

Windows C 32-bit Simple Sample (\IDEA\SAMPLES\WINSIMPL)

To run this sample, simply run \IDEA\EXE\WINSIMPL.EXE after completing Step 1. Position the sample on the left side of your screen. Now, in InTouch, press the button labeled "Do Calculation". This button sets the InTouch variable *DoCalc*, which the sample notices, performs the calculation, and InTouch updates the screen with the result generated by WINSIMPL.EXE. Experiment with the sample source code, rebuild the sample and re-test. For example, change the formula to multiply the 3 inputs then multiply by 2 and see if the results in InTouch correspond.

Windows C 32-bit Complex Sample (\IDEA\SAMPLES\WINCMPLX)

To run this sample, simply run \IDEA\EXE\WINCMPLX.EXE after completing Step 1. Position the sample on the left side of your screen. You should see the current value of $TimeString and $Second displayed on the screen.

Windows C++ 32-bit Simple Sample (\IDEA\SAMPLES\CIDEAAPP)

To run this sample, simply run \IDEA\EXE\CIDEAAPP.EXE after completing Step 1. Position the sample on the left side of your screen. Now, in InTouch, press the button labeled "Do Calculation". This button sets the InTouch variable *DoCalc*, which the sample notices, performs the calculation, and InTouch updates the screen with the result generated by CIDEAAPP.EXE. Experiment with the sample source code, rebuild the sample and re-test. For example, change the formula to multiply the 3 inputs then multiply by 2 and see if the results in InTouch correspond.

To run this sample, copy \IDEA\EXE\CIDEAAPP.EXE to the InTouch directory. It can be run after completing Step 1.

Running Windows NT Samples

To run the NT samples, make sure it's in your InTouch directory or InTouch is in your PATH.

Visual Basic 6.0 32-Bit Sample (IDEA\SAMPLES\VBSIMPLE\VB6)

To run this sample, you must first run InTouch. Select the "\…\VB6ITAPP" application and click the WindowViewer Icon. Run the IDEA\EXE\VB5APP.EXE. The sample gives you the capability to read and write values to and from WindowViewer. To update values in Visual Basic, Click **Update/Read All** from the Visual Basic application menu.

Function Reference

This section contains a description of each function in the InTouch Database External Access (IDEA) Toolkit. The parameters passed to each function and the returned value are described.

Function Summary

Initialization Functions

ACCID	**PtAccInit**(*hWnd*, *nExtraBytes*)
HPT	**PtAccActivate**(*accID*, *Name*)
HPT	**PtAccHandleCreate**(*accID*, *Name*)
int	**PtAccHandleActivate**(*accID*, *hPt*)
PTYPE	**PtAccType**(*accID*, *hPt*)
HPT	**PtAccActivateAndNotify**(*accID*, *Name*)
HPT	**PtAccActivateAndSendNotify**(*accID*, *Name*)
HPT	**PtAccHandleActivateAndNotify**(*accID*, *hPt*)
HPT	**PtAccHandleActivateAndSndNotify**(*accID*, *hPt*)

Data Read Functions

int	**PtAccReadD**(*accID*, *hPt*)
long	**PtAccReadI**(*accID*, *hPt*)
float	**PtAccReadR**(*accID*, *hPt*)
double	**PtAccReadA**(*accID*, *hPt*)
void	**PtAccReadM**(*accID*, *hPt*, *StringBuffer*, *StringBufferLength*)

Data Write Functions

int	**PtAccWriteD**(*accID*, *hPt*, *value*)
int	**PtAccWriteI**(*accID*, *hPt*, *value*)
int	**PtAccWriteR**(*accID*, *hPt*, *value*)
int	**PtAccWriteA**(*accID*, *hPt*, *value*)
int	**PtAccWriteM**(*accID*, *hPt*, *value*)

Shutdown Functions

int **PtAccDeactivate**(*accID, hPt*)

int **PtAccDelete**(*accID, hPt*)

int **PtAccHandleDeactivate**(*accID, hPt*)

int **PtAccHandleDelete**(*accID, hPt*)

int **PtAccShutdown**(*accID*)

void **PtAccShutdownAllAssociated**(*hWnd*)

Miscellaneous Functions

int **PtAccOK**(*accID*)

int **PtAccSetExtraInt**(*accID, hPt, offset, value*)

int **PtAccGetExtraInt**(*accID, hPt, offset*)

long **PtAccSetExtraLong**(*accID, hPt, offset, value*)

long **PtAccGetExtraLong**(*accID, hPt, offset*)

ACCID **PtAccACCIDFromHPT**(*hPt*)

PtAccACCIDFromHPT

```
ACCID

PtAccACCIDFromHPT(      HPT hPt )
```

Description

This function returns the Access Id for a given Point Handle. This is useful when processing DbChgMsg messages. The DbChgMsg message provides the HPT of the variable that changed. You must have the ACCID before you can call any of the PtAccRead functions.

Parameter	Description
hPt	This is the Point Handle created by a call to **PtAccHandleCreate** or **PtAccActivate** that identifies a variable that is temporarily not needed.

Returned Value

The returned value is an Access Id (of type ACCID) that was used when the *hPt* was created.

Example

```
HPT     hPtSeconds;
double    SecondsValue;

if( message == dbChgMsg ) {

    /*  Notification of point change... */
    /*  lParam has the HPT of the variable that changed. */

    hPtSeconds = (HPT)lParam;
    accID = PtAccACCIDFromHPT( hPtSeconds );
    SecondsValue = PtAccReadA( accID, hPtSeconds );

} else switch( message ) {
    /* Usual Windows message processing here */
}
```

PtAccActivate

```
HPT
PtAccActivate( ACCID accID,
               LPSTR lpszName )
```

Description

PtAccActivate performs two functions that must be done before it is possible to access a variable in InTouch. It converts the variable's name to a handle that allows faster access, and it notifies InTouch to activate the variable. This ensures that InTouch has the up-to-date value of the variable.

Parameter	Description
accID	This is a handle of type ACCID returned by a previous call to **PtAccInit**.
lpszName	This is the variable name (tagname or field name) that will be accessed. The name must be a far pointer to a null-terminated ASCII string.

Return Value

The returned value is a handle of the type HPT that can be used in subsequent function calls to read or write the tagname named by *lpszName*. A NULL returned value indicates a failure; probably the *lpszName* is not known to InTouch.

Comments

PtAccActivate function is equivalent to a call to **PtAccHandleCreate** and **PtAccHandleActivate**. When activating and deactivating points, **PtAccHandleActivate** and **PtAccHandleDeactivate** should be used. **PtAccActivate** creates a point handle each time, which is unnecessary.

Note If a program makes multiple calls to **PtAccActivate** for the same variable name, multiple independent HPTs will be created.

Example

```
HPT    hPtSeconds;

hPtSeconds = PtAccActivate( accID, "$Second" ) );
if( hPtSeconds != NULL ) {
 /* $Second can be read/written using accID and hPtSeconds */
}
```

PtAccActivateAndNotify

Note This function is not available in Visual Basic.

```
HPT
PtAccActivateAndNotify( ACCID accID,
                        LPSTR lpszName )
```

Description

This function is similar to PtAccActivate. In addition to creating a Point Handle and activating it, it requests the runtime database to post (using PostMessage) a dbChgMsg message to the window whose handle was used in the PtAccInit() call that created the *accID* when the value changed. This function is not available to Visual Basic applications.

Parameter	Description
accID	This is a handle of type ACCID returned by a previous call to **PtAccInit**. The window whose handle was used in the call to **PtAccInit** will receive the change notification messages.
lpszTagname	This is the variable name (tagname or field name) that will be accessed. The name must be a far pointer to a null-terminated ASCII string.

Return Value

The returned value is a handle of the type HPT that can be used in subsequent function calls to read or write the tagname named by *lpszName*. A NULL returned value indicates a failure; probably the *lpszName* is not known to InTouch.

Comments

For a description of point activation, see "Functional Description" earlier in this chapter.

Note A program that is using this feature must register the DBCHGMSG message with Windows and must check for this message in the main message processing loop. **Example #3** shows how this is done.

Example

```
HPT    hPtSeconds;
hPtSeconds = PtAccActivateAndNotify( accID, "$Second" ) );
if( hPtSeconds != NULL ) {
 /* $Second can be read/written using accID and hPtSeconds */
}
```

PtAccActivateAndSendNotify

Note This function is not available in Visual Basic.

```
HPT

PtAccActivateAndSendNotify( ACCID accID,
                            LPSTR lpszName )
```

Description

This function is similar to **PtAccActivate**. In addition to creating a Point Handle and activating it, it requests the runtime database to send (using SendMessage) a dbChgMsg message to the window whose handle was used in the **PtAccInit()** call that created the *accID* when the value changed. This function is not available to Visual Basic applications.

Parameter	Description
accID	This is a handle of type ACCID returned by a previous call to **PtAccInit**. The window whose handle was used in the call to **PtAccInit** will receive the change notification messages.
lpszName	This is the variable name (tagname or field name) that will be accessed. The name must be a far pointer to a null-terminated ASCII string.

Return Value

The returned value is a handle of the type HPT that can be used in subsequent function calls to read or write the tagname named by *lpszName*.

Comments

For a description of point activation, see "Functional Description" earlier in this chapter.

Note A program that is using this feature must register the DBCHGMSG message with Windows and must check for this message in the main message processing loop. Example #3 shows how this is done.

Example

```
HPT    hPtSeconds;

hPtSeconds = PtAccActivateAndSendNotify( accID, "$Second" ) );
if( hPtSeconds != NULL ) {
 /* $Second can be read/written using accID and hPtSeconds */
}
```

PtAccDeactivate

```
int

PtAccDeactivate( ACCID accID,
                 HPT hPt )
```

Description

PtAccDeactivate notifies InTouch that a variable is no longer needed. This makes it possible to stop polling an item that originates in a I/O server program. The Point Handle remains valid and can later be activated by a call to **PtAccHandleActivate**. While a Point Handle is in the deactivated state, the program should not read its value, as the returned value may not be current.

Parameter	Description
accID	This is a handle of type ACCID returned by a previous call to **PtAccInit**.
hPt	This is the *Point Handle* created by a call to **PtAccHandleCreate** or **PtAccActivate** that identifies a variable that is temporarily not needed.

Return Value

The returned value is 1 if successful or 0 if the function failed.

Example

```
if( PtAccDeactivate( accID, hPtSeconds ) ) {
/* hPtSeconds not "in use", not being polled */
} else {
    /* Error, accID or hPtSeconds is invalid */
}
```

PtAccDelete

```
int

PtAccDelete( ACCID accID,
             HPT hPt )
```

Description

PtAccDelete deletes a Point Handle. After calling **PtAccDelete** the Point Handle must not be used. If the point was activated, it must be deactivated by an explicit call to **PtAccDeactivate** before **PtAccDelete** is called.

Parameter	Description
accID	This is a handle of type ACCID returned by a previous call to **PtAccInit**.
hPt	This is the Point Handle created by a call to **PtAccHandleCreate** or **PtAccActivate** that identifies a variable that is no longer needed.

Return Value

The returned value is 1 if successful or 0 if the function failed.

Example

```
if( PtAccDelete( accID, hPtSeconds ) ) {
    hPtSeconds = NULL;
    /* hPtSeconds deleted */
} else {
    /* Error, accID or hPtSeconds is invalid */
}
```

PtAccGetExtraInt

```
int

PtAccGetExtraInt( ACCID accID,
                  HPT hPt,
                  int nOffset )
```

Description

PtAccGetExtraInt retrieves a value from a 4-byte (32-bit) field at the specified offset within the extra storage area allocated for the specified HPT. If this feature is to be used, IDEA must have been told to allocate extra storage for each HPT. This is done with the *nExtraBytes* argument specified in the call to **PtAccInit** that created the ACCID.

Parameter	Description
accID	This is a handle of type ACCID returned by a previous call to **PtAccInit**.
hPt	This is the Point Handle created by a call to **PtAccHandleCreate** or **PtAccActivate** that identifies the variable whose extra storage area is to be retrieved.
nOffset	The offset in the extra storage area from which the 4-byte value is to be retrieved. The offset must be between 0 and the size of the area minus 4 (32-bit). The size of the area is determined by the *nExtraBytes* argument to **PtAccInit**.

Return Value

The returned value is the integer data that was stored in the two bytes of extra storage area at the offset specified by *nOffset*.

Example

```
ACCID    accID;
HPT      hPtTag;
int      ValueFromOffset12;

accID = PtAccInit( hWnd, sizeof(int));
hPtTag = PtAccActivate( accID, "VariableName" );
PtAccSetExtraInt( accID, hPtTag, 0, DataToSave );

/* later I can retrieve the DataToSave */
ValueFromOffset12 = PtAccGetExtraInt( accID, hPtTag, 12 );
```

PtAccGetExtraLong

```
long

PtAccGetExtraLong( ACCID accID,
                   HPT hPt,
                   int nOffset,
                   long lValue )
```

Description

PtAccGetExtraLong retrieves a value from an 8-byte (32-bit) field at the specified offset within the extra storage area allocated for the specified HPT. If this feature is to be used, IDEA must have been told to allocate extra storage for each HPT. This is done with the *nExtraBytes* argument specified in the call to PtAccInit that created the ACCID.

Parameter	Description
accID	This is a handle of type ACCID returned by a previous call to **PtAccInit**.
hPt	This is the Point Handle created by a call to **PtAccHandleCreate** or **PtAccActivate** that identifies the variable whose extra storage area is to be retrieved.
nOffset	The offset in the extra storage area from which the 8-byte value is to be retrieved. The offset must be between 0 and the size of the area minus 8 (32-bit). The size of the area is determined by the *nExtraBytes* argument to **PtAccInit**.
lValue	This is the new 4-byte value to be stored.

Return Value

The returned value is the long integer data that was stored in the four bytes of extra storage area at the offset specified by *nOffset*.

Example

```
ACCID    accID;
HPT      hPtTag;
long     ValueFromOffset0;

accID = PtAccInit( hWnd, sizeof(long));
hPtTag = PtAccActivate( accID, "VariableName" );
PtAccSetExtraLong( accID, hPtTag, 0, LongDataToSave );

/* later I can retrieve the LongDataToSave */
ValueFromOffset0 = PtAccGetExtraLong( accID, hPtTag, 0 );
```

PtAccHandleActivate

```
int

PtAccHandleActivate( ACCID accID,
                     HPT hPt )
```

Description

This function requests InTouch to activate the variable associated with the Point Handle. This causes InTouch to consider the point to be in use so that its current value will be available.

Parameter	Description
accID	This is a handle of type ACCID returned by a previous call to **PtAccInit**.
hPt	This is the Point Handle created by a call to **PtAccHandleCreate** or **PtAccActivate** that identifies a variable that will be used.

Return Value

The returned value is 1 if successful or 0 if the function failed.

Example

```
hPtSeconds = PtAccHandleCreate( accID, "$Second" );
if( (hPtSeconds != NULL) &&
    PtAccHandleActivate( accID, hPtSeconds ) ) {
 /* $Second can be read/written using accID and hPtSeconds */
}
```

PtAccHandleActivateAndNotify

Note This function is not available in Visual Basic.

```
int
PtAccHandleActivateAndNotify( ACCID accID,
                              HPT hPt )
```

Description

This function is similar to **PtAccHandleActivate**. In addition to activating a Point Handle, it requests the runtime database to post (using PostMessage) a dbChgMsg message to the window whose handle was used in the **PtAccInit()** call that created the *accID* when the value changed. This function is not available to Visual Basic applications.

Parameter	Description
accID	This is a handle of type ACCID returned by a previous call to **PtAccInit**. The window whose handle was used in the call to **PtAccInit** will receive the change notification messages.
hPt	This is the Point Handle created by a call to **PtAccHandleCreate** or **PtAccActivate** that identifies a variable that will be used.

Return Value

The returned value is 1 if successful or 0 if the function failed. If this call fails in a Windows Environment, make sure that the *hWnd* parameter on the **PtAccInit** is correct.

Comments

For a description of point activation, see "Functional Description" earlier in this chapter.

Note A program that is using this feature must register the DBCHGMSG message with Windows and must check for this message in the main message processing loop. Example #3 (described earlier in this chapter) shows how this is done.

Example

```
HPT    hPtSeconds;

hPtSeconds = PtAccHandleCreate( accID, "$Second" ) );
if( hPtSeconds != NULL ) {

    /* $Second can be read/written using accID and hPtSeconds
*/

    PtAccHandleActivateAndNotify( accID, hPtSeconds );

    /*  Now, when $Second changes, InTouch will send a message
*/
    /*  Refer to Example #3 for more detail.
*/
}
```

PtAccHandleActivateAndSndNotify

Note This function is not available in Visual Basic.

```
int
PtAccHandleActivateAndSndNotify( ACCID accID,
                                 HPT hPt )
```

Description

This function is similar to PtAccHandleActivate. In addition to activating a Point Handle, it requests the runtime database to send (using SendMessage) a dbChgMsg message to the window whose handle was used in the **PtAccInit()** call that created the *accID*. This function is not available to Visual Basic applications.

Parameter	Description
accID	This is a handle of type ACCID returned by a previous call to **PtAccInit**. The window whose handle was used in the call to **PtAccInit** will receive the change notification messages.
hPt	This is the Point Handle created by a call to **PtAccHandleCreate** or **PtAccActivate** that identifies a variable that will be used.

Return Value

The returned value is 1 if successful or 0 if the function failed. If this call fails in a Windows Environment, make sure that the *hWnd* parameter on the **PtAccInit** is correct.

Comments

For a description of point activation, see "Functional Description" earlier in this chapter.

Note A program that is using this feature must register the DBCHGMSG message with Windows and must check for this message in the main message processing loop. Example #3 (described earlier in this chapter) shows how this is done.

Example

```
HPT    hPtSeconds;

hPtSeconds = PtAccHandleCreate( accID, "$Second" ) );
if( hPtSeconds != NULL ) {

    /* $Second can be read/written using accID and hPtSeconds
*/
    PtAccHandleActivateAndSndNotify( accID, hPtSeconds );

    /*  Now, when $Second changes, InTouch will send a message
*/
    /*  Refer to Example #3 for more detail.
*/
}
```

PtAccHandleCreate

```
HPT

PtAccHandleCreate( ACCID accID,
                   LPSTR lpszName )
```

Description

Before a tagname (variable) from the InTouch database can be read or written, the program must call **PtAccHandleCreate** or **PtAccActivate** specifying the name of the variable so that it can be verified and a Point Handle (HPT) can be set up. Once the Point Handle is set up, reads and writes of the variable can be done very quickly because the lengthy name comparisons are no longer needed.

Parameter	Description
accID	This is a handle of type ACCID returned by a previous call to **PtAccInit**.
lpszName	This is the variable name (tagname or field name) that will be accessed. The name must be a far pointer to a null-terminated ASCII string.

Return Value

The returned value is a handle of the type HPT that can be used in subsequent function calls to read or write the tagname named by *lpszName*. A NULL returned value indicates a failure; probably the *lpszName* is not known to InTouch.

Example

```
ACCID    accID;
HPT      hPtSeconds;

hPtSeconds = PtAccHandleCreate( accID, "$Second" ) );
if( hPtSeconds != NULL ) {
    if( PtAccHandleActivate( accID, hPtSeconds ) ) {
    /* $Second can be read/written using accID and hPtSeconds
*/
    }
}
```

PtAccHandleDeactivate

```
int
PtAccHandleDeactivate( ACCID accID,
                       HPT hPt )
```

Description

This function is equivalent to a call to **PtAccDeactivate**. It causes InTouch to consider the variable to be not in use. The Point Handle remains valid and can later be activated by a call to **PtAccHandleActivate**. While a Point Handle is in the deactivated state, the program should not read its value. The returned value may not be current.

Parameter	Description
accID	This is a handle of type ACCID returned by a previous call to **PtAccInit**.
hPt	This is the Point Handle created by a call to **PtAccHandleCreate** or **PtAccActivate** that identifies a variable that is temporarily not needed.

Return Value

The returned value is 1 if successful or 0 if the function failed.

Example

```
if( PtAccHandleDeactivate( accID, hPtSeconds ) ) {
    /* hPtSeconds not "in use", not being polled */
}else {
    /* Error, accID or hPtSeconds is invalid */
}
```

PtAccHandleDelete

```
int
PtAccHandleDelete( ACCID accID,
                   HPT hPt )
```

Description

This function is equivalent to a call to **PtAccDelete**. It deletes a Point Handle. After calling **PtAccHandleDelete** the Point Handle must not be used. If the point was activated, it must be deactivated by an explicit call to **PtAccHandleDeactivate** before **PtAccHandleDelete** is called.

Parameter	Description
accID	This is a handle of type ACCID returned by a previous call to **PtAccInit**.
hPt	This is the Point Handle created by a call to **PtAccHandleCreate** or **PtAccActivate** that identifies a variable that is no longer needed.

Return Value

The returned value is 1 if successful or 0 if the function failed.

Example

```
if( PtAccHandleDelete( accID, hPtSeconds ) ) {
    hPtSeconds = NULL;
    /* hPtSeconds deleted */
}else {
    /* Error, accID or hPtSeconds is invalid */
}
```

PtAccInit

```
ACCID

PtAccInit( HWND hWnd,
           int nExtraBytes )
```

Description

The first step required to access information from the InTouch runtime database is to call **PtAccInit**. It verifies that the InTouch runtime database is accessible and prepares for the other InTouch access functions that will be called. If **PtAccInit** returns a NULL handle, it indicates that InTouch is not running. The program may repeat the call until a non-NULL return value is received which indicates that InTouch has become ready.

Parameter	Description
hWnd	In a Windows program, this is the window handle of the window that will receive notifications of value changes for data items. Typically, this will be the program's parent window. It may be NULL if change notifications are not needed. In a Windows program, if this argument is NULL or 0, then no error will occur until the **PtAccHandleActivateAndNotify**.
nExtraBytes	A program may specify that some extra storage be allocated for each data item that is being accessed. If *nExtraBytes* is non-zero, it indicates the number of bytes of extra storage to be allocated for each data item. The storage is allocated when **PtAccHandleCreate** or **PtAccActivate** is called and the extra storage is associated with the Point Handle returned. The storage area can be accessed by **PtAccSetExtraInt**, **PtAccSetExtraLong**, **PtAccGetExtraInt** and **PtAccGetExtraLong**.

Return Value

The return value is an Access Id handle (of type ACCID) that identifies a connection to the InTouch runtime database. It must be saved and passed to other functions that access the database. If NULL is returned, it indicates that InTouch is not available.

Example

```
#include "ptacc.h"

ACCID          accID;
extern HWND    hWndMain;
do {
    accID = PtAccInit( hWndMain, 8 );
    /* 8 bytes extra storage */

} while( accID == NULL );
```

PtAccOK

```
int

PtAccOK(  ACCID accID )
```

Description

After reading all of the tagnames to be used in a computation, the program should call **PtAccOK** to ensure that InTouch is still accessible and the tagname values that were read are valid. If **PtAccOK** returns FALSE, it indicates that InTouch has been shut down. Values read should not be used and the program should call **PtAccShutdown** for each ACCID handle that was created by calls to **PtAccInit**. After that, the program can wait for InTouch to be restarted by periodically calling **PtAccInit** until a non-NULL value is returned.

Parameter	Description
accID	This is a handle of type ACCID returned by a previous call to **PtAccInit**.

Return Value

The returned value is 1 if InTouch is still accessible and the tagnames read since the last call are valid. The return value is 0 otherwise.

Example

```
ACCID    accID;

if( !PtAccOK( accID ) ) {
    PtAccShutdown( accID );
    accID = NULL;
    /*Need to go back and wait for PtAccInit to return non-
NULL
    */
}
```

PtAccReadA

```
double

PtAccReadA( ACCID accID,
            HPT hPt )
```

Description

PtAccReadA returns the current value of an InTouch discrete, integer or floating point variable.

Parameter	Description
accID	This is a handle of type ACCID returned by a previous call to **PtAccInit**.
hPt	This is the Point Handle created by a call to **PtAccHandleCreate** or **PtAccActivate** that identifies the variable whose value to return.

Return Value

The returned value is the value of the variable. A variable of discrete, integer or floating point type is converted to a 64-bit IEEE floating point number. Discretes are represented by 1.0 or 0.0.

Example

```
double    TagValue;
HPT       hPtTag;

hPtTag = PtAccActivate
    ( accID, "DiscreteIntegerOrRealTagName" );
if( hPtTag != NULL ) {
    TagValue = PtAccReadA( accID, hPtTag );
}
```

PtAccReadD

```
int
PtAccReadD( ACCID accID,
            HPT hPt )
```

Description

PtAccReadD returns the current value of an InTouch discrete variable.

Parameter	Description
accID	This is a handle of type ACCID returned by a previous call to **PtAccInit**.
hPt	This is the Point Handle created by a call to **PtAccHandleCreate** or **PtAccActivate** that identifies the variable whose value to return.

Return Value

The returned value is 1 if the discrete variable is ON, 0 if the discrete variable is OFF.

Comments

If the type of the variable (*hPt*) being read is other than discrete, it will be converted to a discrete value as follows:

Integer	If variable is 0, the result is 0. Otherwise the result is 1.
Real	If variable is 0.0, the result is 0. Otherwise the result is 1.
String	This is an error condition, the returned value is always 0.

Example

```
int TagValue;
HPT hPtTag;

hPtTag = PtAccActivate( accID, "DiscreteTagName" );
if( hPtTag != NULL ) {
    TagValue = PtAccReadD( accID, hPtTag );
}
```

PtAccReadI

```
long

PtAccReadI( ACCID accID,
            HPT hPt )
```

Description

PtAccReadI returns the current value of an InTouch integer variable.

Parameter	Description
accID	This is a handle of type ACCID returned by a previous call to **PtAccInit**.
hPt	This is the *Point Handle* created by a call to **PtAccHandleCreate** or **PtAccActivate** that identifies the variable whose value to return.

Return Value

The returned value is the value of the variable. It is a 32-bit signed integer.

Comments

If the type of the variable (*hPt*) being read is other than integer, it will be converted to an integer value as follows:

Discrete	If variable is off or 0, the result is 0. Otherwise the result is 1.
Real	If variable is less than -2,147,483,648 the result is -2,147,483,648. If variable is greater than +2,147,483,647, the result is +2,147,483,647. Otherwise the result is the nearest integer value to the floating point value.
String	This is an error condition, the returned value is always the maximum long integer value (2,147,483,647.)

Example

```
long    TagValue;
HPT hPtTag;

hPtTag = PtAccActivate( accID, "IntegerTagName" );
if( hPtTag != NULL ) {
    TagValue = PtAccReadI( accID, hPtTag );
}
```

PtAccReadM

```
void

PtAccReadM( ACCID accID,
            HPT hPt,
            LPSTR lpszVal
            int nMax )
```

Description

PtAccReadM returns the current value of an InTouch string variable.

Parameter	Description
accID	This is a handle of type ACCID returned by a previous call to **PtAccInit**.
hPt	This is the Point Handle created by a call to **PtAccHandleCreate** or **PtAccActivate** that identifies the variable whose value to return.
lpszVal	A far pointer to a the string buffer where the variable's current value is to be returned. This buffer must be at least 132 bytes in length to accommodate the maximum length InTouch string.
nMax	The length of the string buffer *lpszVal*. If the InTouch string is longer than *nMax*, it will be truncated to fit the string buffer.

Return Value

Void, none.

Comments

None.

Example

```
char    TagValue[132];
HPT hPtTag;

hPtTag = PtAccActivate( accID, "StringTagName" );
if( hPtTag != NULL ) {
    PtAccReadM( accID, hPtTag, TagValue, sizeof(TagValue) );
}
```

PtAccReadR

```
float

PtAccReadR( ACCID accID,
            HPT hPt )
```

Description

PtAccReadR returns the current value of an InTouch floating point variable.

Parameter	Description
accID	This is a handle of type ACCID returned by a previous call to **PtAccInit**.
hPt	This is the Point Handle created by a call to **PtAccHandleCreate** or **PtAccActivate** that identifies the variable whose value to return.

Return Value

The returned value is the value of the variable. It is a 32-bit IEEE floating point number.

Comments

If the type of the variable (*hPt*) being read is other than real, it will be converted to a real value as follows:

Discrete	If variable is off or 0, the result is 0.0. Otherwise the result is 1.0.
Integer	The 32-bit signed integer is converted to 32-bit IEEE floating point format. There is potential for loss of significant digits.
String	This is an error condition, the returned value is always the largest positive floating point value, approximately 3.4 e 38.

Example

```
float    TagValue;
HPT hPtTag;

hPtTag = PtAccActivate( accID, "FloatingPointTagName" );
if( hPtTag != NULL ) {
    TagValue = PtAccReadR( accID, hPtTag );
}
```

PtAccSetExtraInt

```
int

PtAccSetExtraInt( ACCID accID,
                  HPT hPt,
                  int nOffset,
                  int nValue )
```

Description

PtAccSetExtraInt writes a value to a 4-byte (32-bit) field at the specified offset within the extra storage area allocated for the specified HPT. If this feature is used, IDEA must have been told to allocate extra storage for each HPT. This is done with the *nExtraBytes* argument specified in the call to **PtAccInit** that created the ACCID.

Parameter	Description
accID	This is a handle of type ACCID returned by a previous call to **PtAccInit**.
hPt	This is the Point Handle created by a call to **PtAccHandleCreate** or **PtAccActivate** that identifies the variable whose extra storage area is to be modified.
nOffset	The offset in the extra storage area where the 4-byte value is to be stored. The offset must be between 0 and the size of the area - 4. The size of the area is determined by the *nExtraBytes* argument to **PtAccInit**.
nValue	This is the new 4-byte value to be stored.

Return Value

The value returned is the previous contents of the 4-byte field.

Example

```
int    OldValue;

OldValue = PtAccSetExtraInt( accID, hPtTag, 0, NewValue );
```

PtAccSetExtraLong

```
long

PtAccSetExtraLong( ACCID accID,
                   HPT hPt,
                   int nOffset,
                   long lValue )
```

Description

PtAccSetExtraLong writes a value to a 4-byte (32-bit) field at the specified offset within the extra storage area allocated for the specified HPT. If this feature is to be used, IDEA must have been told to allocate extra storage for each HPT. This is done with the *nExtraBytes* argument specified in the call to **PtAccInit** that created the ACCID.

Parameter	Description
accID	This is a handle of type ACCID returned by a previous call to **PtAccInit**.
hPt	This is the Point Handle created by a call to **PtAccHandleCreate** or **PtAccActivate** that identifies the variable whose extra storage area is to be modified.
nOffset	The offset in the extra storage area where the 8-byte value is to be stored. The offset must be between 0 and the size of the area - 4. The size of the area is determined by the *nExtraBytes* argument to **PtAccInit**.
lValue	This is the new 4-byte value to be stored.

Return Value

The value returned is the previous contents of the 4-byte field.

Example

```
long    OldValue;

OldValue = PtAccSetExtraLong( accID, hPtTag, 0, NewValue );
```

PtAccShutdown

```
int
PtAccShutdown(   ACCID accID )
```

Description

PtAccShutdown cleans up and shuts down a connection to InTouch (represented by an ACCID.)

Parameter	Description
accID	This is a handle of type ACCID returned by a previous call to **PtAccInit**.

Return Value

The returned value is 1 if successful or 0 if the function failed.

Example

```
if( PtAccShutdown( accID ) ) {
    /* accID is no longer valid */
    accID = NULL;

} else {
    /*  FALSE return indicates that the accID was invalid */
}
```

PtAccShutdownAllAssociated

```
void
PtAccShutdownAllAssociated(    HWND hWnd )
```

Description

PtAccShutdownAllAssociated cleans up and shuts down every connection to InTouch (ACCID) that was created in association with the specified *hWnd*.

Parameter	Description
hWnd	This is a window handle that was used in one or more calls to **PtAccInit**. Every associated ACCID will be shut down.

Return Value

Void, none.

Example

```
switch( message ) {

case WM_ENDSESSION:
case WM_DESTROY:

    if( !PtAccShutdownAllAssociated( hWnd ) ) {
        /* Error */
    }
}
```

PtAccType

```
PTYPE

PtAccType( ACCID accID,
           HPT hPt )
```

Description

PtAccType returns a code that indicates the type of an InTouch variable.

Parameter	Description
accID	This is a handle of type ACCID returned by a previous call to **PtAccInit**.
hPt	This is the Point Handle created by a call to **PtAccHandleCreate** or **PtAccActivate** that identifies the variable whose type is to be returned.

Return Value

The returned value is a code of the type PTYPE that indicates the type of the variable:

PT_DISCRETE

PT_INTEGER

PT_REAL

PT_STRING

NULL is returned in the case of an invalid ACCID or HPT.

Example

```
int SecondsType;

SecondsType = PtAccType( accID, hPtSeconds );
switch( SecondsType ) {
    case PT_INTEGER:
        Seconds = PtAccReadI( accID, hPtSeconds );
        break;
    case PT_REAL:
        Seconds = PtAccReadR( accID, hPtSeconds );
        break;
    default:
        break;
}
```

PtAccWriteA

```
int
PtAccWriteA( ACCID accID,
             HPT hPt,
             double dValue )
```

Description

PtAccWriteA sets a new value into an InTouch discrete, integer or floating point variable.

Parameter	Description
accID	This is a handle of type ACCID returned by a previous call to **PtAccInit**.
hPt	This is the Point Handle created by a call to **PtAccHandleCreate** or **PtAccActivate** that identifies the variable whose value to set.
dValue	This is the new 64-bit IEEE value to set into the InTouch variable.

Return Value

The returned value is 1 if successful or 0 if the function failed.

Comments

The value being written will be converted to the appropriate type according to the type of the InTouch variable as follows:

Discrete	If *dValue* is 0.0, 0 is written to the InTouch variable. Otherwise 1 is written.
Integer	If *dValue* is greater than the maximum 32-bit signed integer value (2,147,483,647) the InTouch variable will be set to the maximum. If *dValue* is less than the minimum 32-bit signed integer value (-2,147,483,648) the InTouch variable will be set to the minimum. Otherwise, *dValue* is converted to a 32-bit signed integer.
Real	If *dValue* is greater than the maximum 32-bit IEEE floating point value (3.4 e 38) the InTouch variable will be set to the maximum. If *dValue* is less than the minimum 32-bit IEEE floating point value (-3.4 e 38) the InTouch variable will be set to the minimum. Otherwise, *dValue* is converted to a 32-bit IEEE floating point value.
String	This is an error condition, no write takes place.

Example

```
if( !PtAccWriteA( accID, hPtTag, NewDoubleValue ) ) {
    /* Error, accID or hPtTag invalid */
}
```

PtAccWriteD

```
int
PtAccWriteD( ACCID accID,
             HPT hPt,
             int bValue )
```

Description

PtAccWriteD sets a new value into an InTouch discrete variable.

Parameter	Description
accID	This is a handle of type ACCID returned by a previous call to **PtAccInit**.
hPt	This is the Point Handle created by a call to **PtAccHandleCreate** or **PtAccActivate** that identifies the variable whose value to set.
bValue	The new value for the variable. If *bValue* is 0, the InTouch discrete variable is set to OFF, otherwise it is set to ON.

Return Value

The returned value is 1 if successful or 0 if the function failed.

Comments

If the type of the variable (*hPt*) being written is other than discrete, it will be converted to the appropriate value as follows:

Integer — If *bValue* is zero, 0 is written to the InTouch variable. Otherwise 1 is written.

Real — If *bValue* is zero, 0.0 is written to the InTouch variable. Otherwise 1.0 is written.

String — This is an error condition, no write takes place.

Example

```
if( !PtAccWriteD( accID, hPtTag, NewDiscreteValue ) ) {
/* Error, accID or hPtTag invalid */
}
```

PtAccWriteI

```
int
PtAccWriteI( ACCID accID,
             HPT hPt,
             long lValue )
```

Description

PtAccWriteI sets a new value into an InTouch integer variable.

Parameter	Description
accID	This is a handle of type ACCID returned by a previous call to **PtAccInit**.
hPt	This is the Point Handle created by a call to **PtAccHandleCreate** or **PtAccActivate** that identifies the variable whose value to set.
lValue	This is the new value to be set into the InTouch variable.

Return Value

The returned value is 1 if successful or 0 if the function failed.

Comments

If the type of the variable (*hPt*) being written is other than integer, it will be converted to the appropriate value as follows:

Discrete	If *lValue* is zero, 0 is written to the InTouch variable. Otherwise 1 is written.
Real	*lValue* is converted to a 32-bit IEEE floating number. There is potential for loss of significant digits.
String	This is an error condition, no write takes place.

Example

```
if( !PtAccWriteI( accID, hPtTag, NewIntegerValue ) ) {
    /* Error, accID or hPtTag invalid */
}
```

PtAccWriteM

```
int

PtAccWriteM( ACCID accID,
             HPT hPt,
             LPSTR lpszValue )
```

Description

PtAccWriteM sets a new value into an InTouch string variable.

Parameter	Description
accID	This is a handle of type ACCID returned by a previous call to **PtAccInit**.
hPt	This is the Point Handle created by a call to **PtAccHandleCreate** or **PtAccActivate** that identifies the variable whose value to set.
lpszValue	A far pointer to a the new string value to set into the InTouch variable. This string must be at most 131 bytes in length. If the new string is longer than 131 characters, it will be truncated to 131.

Return Value

The returned value is 1 if successful or 0 if the function failed.

Example

```
if( !PtAccWriteM( accID, hPtTag, NewStringValue ) ) {
    /* Error, accID or hPtTag invalid */
}
```

PtAccWriteR

```
int
PtAccWriteR( ACCID accID,
             HPT hPt,
             float fValue )
```

Description

PtAccWriteR sets a new value into an InTouch floating point variable.

Parameter	Description
accID	This is a handle of type ACCID returned by a previous call to **PtAccInit**.
hPt	This is the Point Handle created by a call to **PtAccHandleCreate** or **PtAccActivate** that identifies the variable whose value to set.
fValue	This is the new IEEE 32-bit floating point value to be set into the InTouch variable.

Return Value

The returned value is 1 if successful or 0 if the function failed.

Comments

If the type of the variable (*hPt*) being written is other than real, it will be converted to the appropriate value as follows:

Discrete	If *fValue* is 0.0, 0 is written to the InTouch variable. Otherwise 1 is written.
Integer	If *fValue* is greater than the maximum 32-bit signed integer value (2,147,483,647) the InTouch variable will be set to the maximum. If *fValue* is less than the minimum 32-bit signed integer value (-2,147,483,648) the InTouch variable will be set to the minimum. Otherwise, *fValue* is converted to a 32-bit signed integer.
String	This is an error condition, no write takes place.

Example

```
if( !PtAccWriteR( accID, hPtTag, NewFloatingPointValue ) ) {
    /* Error, accID or hPtTag invalid */
}
```

CHAPTER 11

ITEdit.OCX

ITEdit is an OLE control specifically designed to access InTouch (6.0 or greater) database from any OLE container that provides support for OLE controls. It can be configured to respond to changes in InTouch tagname values in addition to changes in InTouch run status. Some features are:

- ITEdit provides an interface, via OLE Automation, that enables the user to programmatically alter its functionality
- ITEdit provides a property sheet for configuration purposes
- ITEdit is a replacement for the VBIT custom control for Visual Basic within the 32 bit environment
- ITEdit functionality is provided via the newly created class CIdea, which is a C++ wrapper around the functionality provided within PTACC DLL. It allows the user to access these features without having to learn the intricacies of PTACC.

Contents

ITEdit Overview

ITEdit is an OLE control specifically designed to access the InTouch database from any OLE container that provides support for OLE controls. It can be configured to respond to changes in InTouch tagname values and InTouch run status. ITEdit provides an interface, via OLE automation, that enables you to programmatically alter its functionality. In addition, ITEdit provides a property sheet for configuration purposes. ITEdit is a replacement for the VBIT custom control for Visual Basic within the 32 bit environment.

ITEdit is a subclassed edit control and therefore inherits all properties of this type of control. ITEdit can be configured to reflect the value of any valid InTouch tagname. ITEdit displays the current value of the tagname and uses this information to change the tag's value. When the control has focus, all updates are suspended. You can now change the value of the tagname. The new value is written to the tagname when focus is lost or when Enter is pressed. Updates to the tag's value are reflected in the control only after it has lost focus.

There are many ways to display the value of a tagname within ITEdit. ITEdit can be configured to display different strings for different values of a "*Discrete*" tagname. For tagnames of numeric types, you can use the familiar formatting capabilities of InTouch to achieve desired output. For tagnames that are of type "*Message*", the value displayed is independent of the format string specified.

Registering ITEdit.OCX

To register ITEdit control you must run the registration utility REGSVR32.EXE. The command line is "REGSVR32 *path*\ITEDIT.OCX" (where *path* is the fully qualified path to ITEDIT.OCX) and is entered in the directory where the control is located.

The registration process results in information about the control being placed into the system registry. ITEdit is automatically registered once it is installed. Once registered, the control can be used by any application. However, ITEdit must be re-registered anytime you change its location or copy it to a different computer.

➢ **To register ITEdit:**

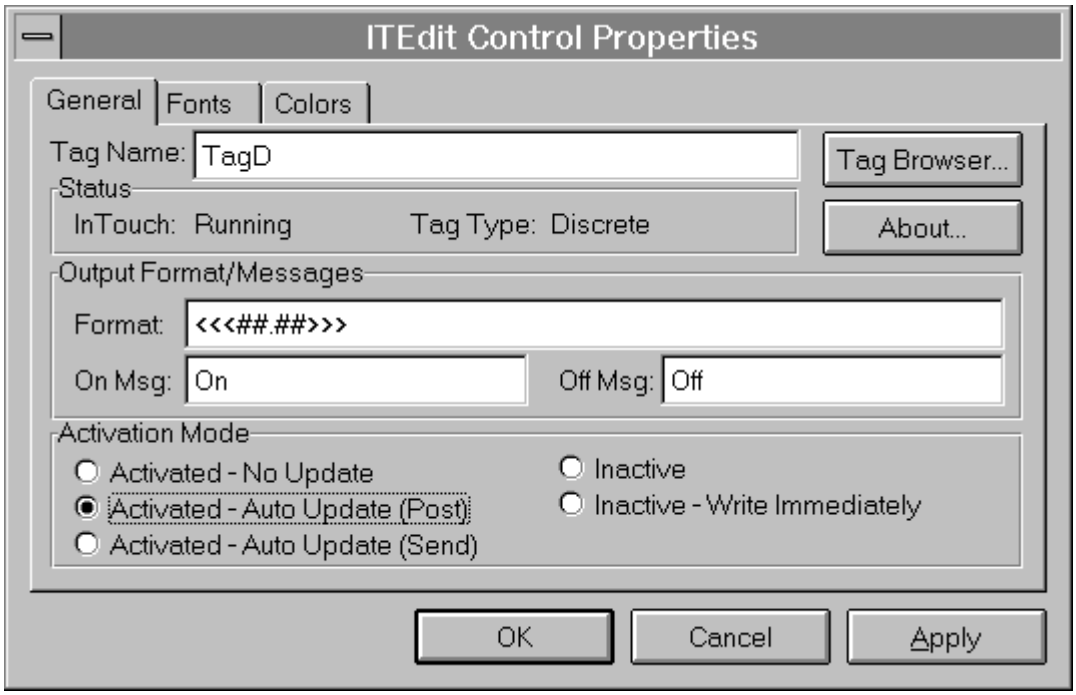
1. On the Windows Taskbar, click **Start** and then, click **Run**. The **Run** dialog box will appear.
2. In the **Open** box, type REGSVR32.EXE.
3. Type the path where ITEdit.OCX is located including the file name ITEDIT.OCX.
4. Click **OK**.

Installing ITEdit.OCX

Install ITEdit.OCX into your development environment as you would with any new control. The icon appears in your application.

Configuring ITEdit.OCX

The ITEdit Control Properties dialog box is used to configure ITEdit.OCX. The dialog box is accessed via the Properties menu item on the Context menu. Right-click on the control to display the ITEdit Control context menu, then select the Properties item to open the dialog box. A listing of each field and its description are as follows:



General Tab

The **General Tab** is used to configure custom properties of the ITEdit control. **Fonts** and **Colors** tabs are common property sheets provided by Microsoft. They are used to configure stock properties and are utilized by many other OLE Controls.

Field	Description
Tagname	Enter the tagname associated with the control in this field.
Tag Browser	Click **Tag Browser** to select a tagname. All tagnames in the Tagname Data Dictionary are shown.

Field	Description
Status	Displays the status of InTouch and the tagname type associated with the control. **InTouch**: Running or Not Running. **Tag Type**: Discrete, Real, Integer, or Message.
Output Format/Messages	Contains three fields that configure the format for the tag's value. **Format** Used to define the output format for numeric values in a text field. It allows for other tests to be used around the actual format specification. For example: *Welcome #.# InTouch* yields *Welcome 3.2 InTouch*. **On Msg:** **On -** If you defined the tagname as a discrete, when the tagname's value is equal to **1**, any message you enter here will be displayed in the alarm window's value/limit field. **Off Msg:** **Off -** If you defined the tagname as a discrete, when the tagname's value is equal to **0**, any message you enter here will be displayed in the alarm window's value/limit field.
Activation Mode	Determines the interaction between the tagname and the control. Select on the appropriate option to configure the tag's activation property as follows: Mode 0 **Inactive-Write Immediately** Mode 1 **Inactive** Mode 2 **Activated-No update** Mode 3 **Activated-Auto Update (Post)** Mode 4 **Activated-Auto Update (Send)** For more information, refer to *ITActivationMode* Property.

ITEdit Properties

The following is a list of **ITEdit** stock and custom properties.

Stock Properties	Custom Properties
Appearance	ITActivationMode
BackColor	ITDataIsValid
BorderStyle	ITFormat
Enabled	ITOffMessage
Font	ITOnMessage
ForeColor	ITRunning
	ITTagName
	ITTagType
	ITValue
	ITValueQuality

Stock Properties

Stock Property	Description
Appearance	Determines if the control is flat or 3D. Use (0) zero for flat and (1) one for 3D.
BackColor	Specifies the color of the interior of the control (in RGB values).
BorderStyle	Specifies if the control is drawn with or without a border. Use (0) zero for no border and (1) one for a border.
Enabled	Specifies if the control can receive focus. Use (0) zero for disabled and (1) one for enabled.
Font	Specifies the current font for the control. An OLEFONT structure is required to utilize this property.
ForeColor	Specifies the color for the display of the text and graphics in the control (in RGB values).

Custom Properties

This section describes ITEdit custom properties.

ITActivationMode Property Windows NT

Description

A read/write property that defines the mode of displayed tagname associated with the control. Setting this property determines the interaction between the tagname and the control. Interaction can range from none to the tagname being activated and notification of changes being received.

Example

```
TagMode = ITEditCtrl.ITActivationMode
ITEditCtrl.ITActivationMode = "4"
```

Valid Activation Modes are:

Mode	Mode Description
0	**Inactive.** In this mode, no values will be read from or written to the WindowViewer application. Internally, no tag handle is created within the control for the assigned tag. This is a great mode to switch your control to when it will be idle in the Visual Basic application. This will help reduce the amount of overhead between your Visual Basic application and WindowViewer.
1	**Inactive: Write Immediately.** A value may be written to the tagname without waiting for the setup period for I/O Server based tagnames. In this mode, values may be written, but no values may be read from the WindowViewer application. Internally, no tag handle is initially created; however, when a write is performed a tag handle is created, activated, the value is written, and then the tag handle is deactivated and deleted. This also helps reduce the amount of overhead between your Visual Basic application and WindowViewer. This mode is perfect for controls that will only perform writes to tags but no reads.

Mode	Mode Description
2	**Activated, No Update.** Updates to the tag's value are not received by the control. The tag's value is updated when an ITValue property is being retrieved or set. In this mode, values may be read from or written to the WindowViewer application. Internally, the tag handle is created initially, then the handle is activated and will stay that way as long as the control is in this mode. This mode requires a little more overhead since the tag handles are always active. This means that even your WindowViewer application has the tags active in its database. This mode is good for tags that must remain active so reads and writes can be performed "on the fly;" however, most of the time switching between this mode and mode 0 (zero) is your best bet. It will help cut down on the overhead induced by keeping tag handles active.
3	**Activated Auto Update: Post.** The value of information displayed within the control is updated whenever a PTACC notification message is received. The notifications are achieved by using "PostMessage". In this mode, values may be written at any time to WindowViewer, and when the value changes in WindowViewer, it will automatically be reflected in the control. Internally, the tag handle is created and activated with the automatic notification method using the **PostMessage Windows API**. This is very handy for controls that will monitor things, like heartbeats, and for controls where you're waiting for a change in the value from WindowViewer to perform some action within your Visual Basic application. This mode requires more overhead than the previous activation modes and should only be used when necessary.
4	**Activated Auto Update: Send.** The value of information displayed within the control is updated whenever a PTACC notification message is received. The notifications are achieved by using "SendMessage. This mode is almost identical to Mode 3, but it uses the **SendMessage Windows API**, rather than the PostMessage API. The drawback of this is that it now runs synchronously with WindowViewer rather than asynchronously. This can be a major drawback to your Visual Basic application as it may slow it down considerably, especially if the WindowViewer application is busy (that is, many scripts, windows, tags, an so on).

Remarks

Many users and developers simply drop the ITEdit control onto their Visual Basic form and then switch to Mode 3 or 4. In some cases this is the best configuration. However, there are other instances in which this isn't desirable.

With very large, very busy InTouch applications, care must be taken when selecting the activation modes. It's important to note that it's acceptable to switch from one mode to another during runtime to achieve the desired affect. Switching modes doesn't really add much overhead and could actually reduce overhead in the long run.

Example

Let's say we have a Visual Basic application with four instances of the ITEdit ActiveX control in it. Each of them will be used for a different purpose while communicating with a WindowViewer application. The following is a description of what the controls will do:

- **Control1** - Used to monitor a heartbeat tag
- **Control2** - Used to monitor WindowViewer for a specific event
- **Control3** - Used to loop through a list of tags and write values to them
- **Control4** - Used to loop through a list of tags and read values from them

The mode for **Control1** should be set to 3. We need to have WindowViewer notify us every time the heart beat changes so we don't have to ask WindowViewer for a new value every second or so.

For **Control2**, we should also set the mode to 3 whenever we need to monitor WindowViewer for the event. There may be times when we don't need to monitor the WindowViewer application for the change in this tag, so we would set the mode to 0 (zero) during those periods to cut down on overhead.

For **Control3**, we're going to loop through a list of tags and write to them. In this case, we'll never perform any reads so we'll set the mode to 1. This will only create and activate tag handles around the write, but at no other time, adding to the efficiency of the application.

At various times we'll be looping through a list of tags and reading from them with **Control4**. During the read times, we'll set the mode to 2. There's no point in having WindowViewer send notification messages when the tag value changes since we'll be reading it with the ITValue property (and looping through several tags in the first place). Therefore, Modes 3 and 4 would be pointless and would add too much overhead to the application. During the times we're not reading from the tags, we should set the mode to 0 (zero) to further reduce overhead.

So, when should Mode 4 be used? First, you should understand the difference between the APIs that are used with Modes 3 and 4. When the **PostMessage Windows API** is used with Mode 3, an application will send a message to another application, continue its processing, and assume the application that is the recipient of the message did in fact receive the message. On the other hand, when the **SendMessage Windows API** is used with Mode 4, the sending application will actually wait for the receiving application to acknowledge receiving the message. It's important to note that the return value the sender receives only tells the sender that the intended application received the message, not that it was actually processed properly.

Thus, waiting for the receiving application to acknowledge that it received a message can cause problems, particularly if the recipient is busy when the message is sent. This will cause the sender to hang until a return value is sent by the recipient application.

There may be some cases when a developer may decide to use the SendMessage approach (Mode 4) because it better suits their needs and they will choose it over Mode 3. Regardless of which mode you choose, remember that it will always help your Visual Basic and WindowViewer applications to run better if you can "turn off" the automatic notification messaging and set the mode to zero as often as possible.

ITDataIsValid Property Windows NT

Description A read only property that determines whether data retrieved via the ITValue property is valid. This property determines if InTouch is running and if the data point is valid.

Example

```
ValidData = ITEditCtrl.ITDataIsValid
```

ITFormat Property Windows NT

Description A read/write property that defines the format of displayed value of the tagname. You can embed any InTouch type format specifiers within this field.

Example

```
TagFormat = ITEditCtrl.ITTagFormat
ITEditCtrl.ITTagFormat = "Total count of widgets is ##.##"
```

ITOffMessage Property Windows NT

Description A read/write property that defines the value displayed for a Discrete tagname when the value is FALSE. This will override any format string specified.

Example

```
TagOffMessageString = ITEditCtrl.ITOffMessage
ITEditCtrl.ITOffMessage = "False"
```

ITOnMessage Property Windows NT

Description A read/write property that defines the value displayed for a Discrete tagname when the value is TRUE. This will override any format string specified.

Example

```
TagOnMessageString = ITEditCtrl.ITOnMessage
ITEditCtrl.ITOnMessage = "True"
```

ITRunning Property Windows NT

Description Checks status of InTouch. If InTouch is running, value *True* is returned, else *False*. Read-only at runtime. Not available at design time.

Example

```
InTouchRunning = ITEditCtrl.ITRunning
```

ITTagName Property Windows NT

Description Defines name of the InTouch tagname ITEdit control is attached to.

Example

```
ITEditCtrl.ITTagName = "ReactorTag1"
TagString = ITEditCtrl.ITTagName
```

ITTagType Property Windows NT

Description Returns type code of InTouch tagname which is attached to ITEdit control. Read only at runtime. Not available at design time.

Example

```
TagType = ITEditCtrl.ITTagType
```

Return values

1 = discrete tagname

2 = integer tagname

3 = real number tagname

4 = message tagname

ITValue Property Windows NT

Description A read/write property used to either get or set tagname values associated with the control. The type of parameters needed to get or set the value varies depending on the tagname type. This is the default property for this control. Setting the value through this property results in a tagname database write. Retrieving the value through this property results in a database read.

Example

```
ITEditCtrl.ITValue = 78
ITEditCtrl = 89
Count = ITEditCtrl.ITValue
Count = ITEditCtrl
```

ITValueQuality Property Windows NT

Description A read-only property that contains the InTouch tag data quality value. The data type is a long integer (32-bit). Descriptions of InTouch quality values can be found in the Chapter 5, "Protocols," in your *FactorySuite System Administrator's Guide*.

Example For an example using this property, see "Using ITNotifyValue and ITNotifyQuality" later in this chapter.

Events

This section describes ITEdit events.

ITNotifyValue Event Windows NT

Description Fires when the value of a particular InTouch tag changes or is initialized.

For more information, see "Using ITNotifyValue and ITNotifyQuality" later in this section.

ITNotifyQuality Event Windows NT

Description Fires when the data quality of a particular InTouch tag changes or is initialized.

For more information, see "Using ITNotifyValue and ITNotifyQuality" later in this section.

Using ITNotifyValue and ITNotifyQuality

If both value and quality for a tag changes, the events will be triggered in the following order: 1) **ITNotifyValue**, 2) **ITNotifyQuality,** and finally 3) **ITNotify**.

The **ITValue** and **ITValueQuality** properties are updated prior to the triggering of these events.

The following programming example illustrates how to use the **ITNotifyQuality** and **ITNotifyValue** events.

```
Dim MinTag, MaxTag As Integer

Private Sub ITEdit1_ITNotify()
  Rem This function can be used for backward compatibility
End Sub


Private Sub ITEdit1_ITNotifyQuality()
If ITEdit1.ITValueQuality <> &HC0 Then
  Rem Value Quality not GOOD
  ITEdit1.Enabled = False
    If (ITEdit1.ITValueQuality = &H56) Or
      (ITEdit1.ITValueQuality = &H55)
```

Continued

```
Then
      Rem Value Quality is CLAMPED HIGH or CLAMPED LOW
      Label1.Caption = "TagName value must be changed"
    Else
      Rem Other bad Value Quality
      Label1.Caption = ""
    End If
  Else
    Rem Value Quality is GOOD
    ITEdit1.Enabled = True
  End If

End Sub

Private Sub ITEdit1_ITNotifyValue()

If ITEdit1.ITValueQuality = &HC0 Then
  Rem Value Quality is GOOD
  If ITEdit1.ITValue > MaxTag Then
    MaxTag = ITEdit1.ITValue
  End If

  If ITEdit1.ITValue < MinTag Then
    MinTag = ITEdit1.ITValue
  End If
End If

End Sub
```

Error Dialog Box

Displays when ITEdit.OCX is used and WindowViewer is not running.

CHAPTER 12

Tag Access

This chapter provides instructions and documentation for using the Wonderware Tag Access ActiveX objects for InTouch. The Tag Access objects consist of a DataChange ActiveX control and a TagLink ActiveX object. The TagBrowser ActiveX control is also installed. These components provide developers using Visual Basic's rapid application development (RAD) tools with the ability to quickly deploy applications that link to the InTouch real-time database. Because they utilize standard ActiveX technologies, these components are also useful in any of the Microsoft Office applications as they can be used from within Visual Basic for Applications (VBA) to expose an InTouch tag database object model.

Contents

Tag Access ActiveX Objects for InTouch

The Tag Access ActiveX objects can be used to develop extensions to InTouch in a variety of ways:

- They can be used to develop standalone applications that integrate with InTouch, such as custom data loggers, setpoint downloading, statistical/advanced numerical analysis, custom InTrack clients, and more.
- They can be used to create ActiveX servers that can be called from within the InTouch scripting environment, allowing Visual Basic to be used for the bulk of application scripting, leveraging the speed, functionality, and extensibility of Visual Basic.
- These components can be embedded into other ActiveX controls, enabling them to be used in the creation of custom ActiveX controls such as special types of animations, charts, or user interface objects that can be used in InTouch or Visual Basic and are bound to data in the InTouch tagname dictionary.

The Tag Access ActiveX objects also allow "encapsulation" of business logic or process expertise in high-performance compiled objects that cannot be inadvertently changed or "borrowed" by others. Updates to the business logic can be deployed without requiring modifications to the InTouch application. This is an extremely powerful capability to virtually any type of FactorySuite™ user. OEMs can protect their proprietary value-added to the FactorySuite, systems integrators can develop vertical-industry application components that they can reuse in their projects, and end users can develop "standard objects" that can be deployed throughout their plant or enterprise.

The TagLink object provides access to all relevant dot fields of an InTouch tag, including both read and write access where appropriate, and the DataChange ActiveX control provides a means to monitor one or more InTouch tags and to receive notification whenever the value, alarm state, or acknowledge state changes. The TagBrowser ActiveX control provides a means to browse the tagname database for any InTouch application, remote or local. Extensive capabilities to filter and control the appearance of the browser are provided.

A number of sample applications have been provided that allow you to see "real-world examples" of how these components can be applied to implement innovative solutions to your manufacturing challenges.

For more information, see "Sample Applications" later in this chapter.

Requirements

Requirements for using the Tag Access objects are as follows:

- In order to use these controls with Visual Basic, you must be using Visual Basic 6.0 with FactorySuite 2000, Version 7.1 (or later).
- InTouch 7.1 or later must be installed, and an InTouch runtime license is required at each node that will be using the Tag Access ActiveX objects. We strongly suggest installing all applicable Windows NT and FactorySuite Service Packs.
- To use the ActiveX objects with Microsoft Office and Microsoft Visual Basic, it is essential that Office Service Release 2 or later be installed. Office Service Release 2 fixes a number of critical bugs in VBA, particularly when using VBA within Microsoft Excel.

Note This release does not support remote referencing of InTouch data using ACCESSNAME:TagName.Field syntax.

Deployment Information

After building your applications on your Visual Basic development computer, it is strongly suggested that you develop a setup/installation program for deploying your application/component on other computers. We have had good experiences with InstallShield and Wise Installation. Be sure to install the following files in your application as needed:

Installation Files

InTouchCOM.DLL (for the TagLink and DataChange objects)

LHTagBrowser.OCX (for the TagBrowser ActiveX control)

Registration

Both files are self-registering. Prior to registering InTouchCOM.DLL, be certain that the InTouch installation directory is in the PATH environment variable (typically C:\Program Files\FactorySuite\InTouch). This is critical for accessing certain InTouch DLLs that are used by the InTouchCOM.DLL component.

Dependencies

- Both files require the MFC DLL (MFC42.DLL) and the Microsoft C Runtime library (MSVCRT.DLL).
- LHTagBrowser.OCX also requires the Visual Basic Runtime (MSVBVM50.DLL) and the Common Controls component (COMCTL32.OCX).
- InTouchCOM.DLL requires that InTouch 7.0 or newer must be installed and an InTouch runtime license is required at each node that will be using these ActiveX controls and objects. We strongly suggest installing all applicable Windows NT and FactorySuite service packs.

DataChange ActiveX Control

The DataChange ActiveX control is extremely useful for implementing "event-based" rather than "polling-based" applications that will need to integrate with the InTouch real-time tag database. This ActiveX control allows the developer to monitor the value, alarm status, and acknowledge status of any InTouch tag and to receive ActiveX events whenever any of these values change.

Each DataChange ActiveX control can monitor up to 512 InTouch tags. The ActiveX control maintains an "active watch list" of tags that are of interest to the container application. The **AddWatch** and **RemoveWatch** methods are used to add or remove tags from this list. In general, it is more efficient to have a single instance of the DataChange ActiveX control monitor multiple tags than to create multiple instances of the ActiveX control on a form.

The application developer has control over which events are generated for each monitored tag. This is determined by passing the appropriate flags when the **AddWatch** method is called to add a tagname to the active list. Separate flags are provided to enable/disable the **ValueChanged**, **AlarmStatusChanged**, and **AckStatusChanged** events.

Another useful feature is the ability to associate a user-determined value (a Variant value) with each entry in the active tag list. This parameter is passed (along with the tagname and the changed value) to the event handler routine for each of the three types of events, providing another means besides the tagname for associating the event with some code or other item in the container application. A common example might be a color chart application monitoring an array of tags, such as the ten temperature readings in a baking oven, and associating a value identifying the oven heating zone with each monitoring point. When any of the zone temperatures changes or changes alarm/ack state, the container application can simply use the UserData parameters as an index to quickly redraw the display.

Since the UserData parameter is a "Variant" data type, it can also be used to store references to other objects, such as a reference to a TagLink object that corresponds to the same tag. In this manner, whenever an event is received, you could easily access any of the tag’s dot fields at the same time you are handling the event.

Events

The DataChange ActiveX control can generate up to three events for each tag that it monitors. These events correspond to the most commonly accessed attributes of an InTouch tag, those being its value, its alarm status, and its acknowledge status. Whenever the value of any of these attributes changes for a tag being monitored, the associated event will be fired.

Note Whenever a tag is added to the active "watch list" using the **AddWatch** method, a set of Boolean flags are used to indicate which events are of interest to the application, allowing specific events to be monitored or suppressed on a "per tag" basis.

AckStatusChanged

This event is fired whenever the .Ack field for a tag being monitored changes state. The event handler format (shown as it might appear in Visual Basic) is:

```
Private Sub DataChange1_AckStatusChanged(ByVal TagName As
String, ByVal AckStatus As Long, ByVal UserData As Long)
       Debug.Print "Ack Changed For " + TagName + " To " +
       Format$(AckStatus)
End Sub
```

The UserData parameter is the same user-supplied value that was associated with the tag when the **AddWatch** method was called and can be used as an alternative to the tagname as a means to determine which tag has changed.

AlarmStatusChanged

This event is fired whenever the .Alarm field for a tag being monitored changes state. Note that an event will not be fired when the tag changes from one alarm state to another, such as HI to HIHI, but only from Normal to Alarm or Alarm to Normal. The event handler format (shown as it might appear in Visual Basic) is:

```
Private Sub DataChange1_AlarmStatusChanged(ByVal TagName As
String, ByVal AlarmStatus As Long, ByVal UserData As Long)
       Debug.Print "Alarm Changed For " + TagName + " To " +
       Format$(AlarmStatus)
End Sub
```

The UserData parameter is the same user-supplied value that was associated with the tag when the **AddWatch** method was called and can be used as an alternative to the tagname as a means to determine which tag has changed.

ValueChanged

This event is fired whenever the value of a tag being monitored changes. Note that the data type for the value is a VARIANT, since the actual data type is dependent on the InTouch tag type. The event handler format (shown as it might appear in Visual Basic) is:

```
Private Sub DataChange1_ValueChanged(ByVal TagName As String,
ByVal Value As Variant, ByVal UserData As Long)
      Debug.Print "Value Changed For " + TagName + " To " +
      Format$(vData)
End Sub
```

The corresponding data types for each InTouch tag type are:

InTouch Tag Type	VARIANT data type
Discrete	Boolean (VT_BOOL)
Integer	Long Integer (VT_I4)
Real	Float (VT_R4)
Message	String (VT_BSTR)

The UserData parameter is the same user-supplied value that was associated with the tag when the **AddWatch** method was called and can be used as an alternative to the tagname as a means to determine which tag has changed.

Methods

Each DataChange ActiveX control can monitor up to 512 InTouch tags. The ActiveX control maintains an "active watch list" of tags that are of interest to the container application. The **AddWatch** and **RemoveWatch** methods are used to add or remove tags from this list.

Note Each of the methods can potentially raise ActiveX errors if invalid tagnames are passed as parameters. As such, normal Visual Basic (or other) client error handling can be used to trap and handle these errors in the calling application.

AddWatch

This method is used to add a tag to the "active watch list" for a specific instance of the DataChange control. If the tagname is not valid, the maximum number of active tags is exceeded, or if communications cannot be established to InTouch, an ActiveX error will be raised. Conversely, the **RemoveWatch** method removes a tag from the active watch list. The syntax for the method, shown in Visual Basic form, is as follows:

```
Call DataChangeControl.AddWatch(ByVal TagName As String, ByVal
Boolean bNotifyValue, ByVal Boolean bNotifyAlarms, ByVal
Boolean bNotifyAcks, ByVal Variant UserData)
```

Note that the Boolean and Long are similar from an ActiveX perspective, thus the type library will indicate a "Long" data type for BOOL parameters.

Parameter	Description
TagName	A string value containing the InTouch tagname to be monitored
bNotifyValue	A Boolean flag indicating whether value change events should be generated for this tag
bNotifyAlarms	A Boolean flag indicating whether alarm status change events should be generated for this tag
bNotifyAcks	A Boolean flag indicating whether ack status change events should be generated for this tag
UserData	A Variant value that represents a user-supplied value which will be sent with any notification events

Return Type

This method does not return a value.

Example

The following code fragment demonstrates this method. In this example, we add ten tags to the active watch list and ask the control to notify us if either the value, alarm, or ack state changes (all three event flag parameters are "true"). We also use the UserData parameter to store the index into an array, which we could use later in the event handler. Of course, this value could be simply set to zero as well.

```
Const NUMTAGS = 10

Dim I As Integer

 ‘ Use "inline" error handling

On Error Resume Next

For I=1 to NUMTAGS

       Call DataChange1.AddWatch("Analog"+Format$(I),true,
       true,true,I)

       If Err.Number <> 0 Then

          MsgBox Err.Description

End If

Next
```

For information and example code on how the UserData parameter can be used to enhance the power of the DataChange ActiveX control and to integrate the control with the TagLink object, see "DataChange ActiveX Control."

RemoveWatch

This method is used to remove tag from the "active watch list" for a specific instance of the DataChange control. If the tagname is not found in the active list, an ActiveX error will be raised (and can be handled by the caller). Conversely, the **AddWatch** method adds a tag to the active watch list. The syntax for the method, shown in Visual Basic form, is as follows:

```
Call DataChangeControl.RemoveWatch(ByVal TagName As String)
```

Parameter	Description
TagName	A string value containing the InTouch tagname to be removed from the active list

Return Type This method does not return a value.

Example The following code demonstrates how this method might be used in an application.

```
 ‘ Trap the possible error if the tagname is not valid
On Error Resume Next
Call DataChange1.RemoveWatch("MyTagName")
If Err.Number <> 0 Then
       MsgBox "Tag Was Not Being Monitored" + Err.Description
End If
```

Trappable Errors

The following trappable errors are generated by the DataChange ActiveX control:

Value	Description
8193	"Connection To InTouch Failed"
8194	"Invalid Tag Name"
8200	"Exceeded maximum number of tags that can be active for a single control (512)"
8201	"Specified tagname was not in the active list"

TagLink Object

The TagLink object provides an ActiveX/COM object model wrapper around the InTouch tag database, allowing full access to the various attributes and dot fields of an InTouch tag using properties of the TagLink object. Many of these properties can be written to in addition to read, allowing application to perform advanced tasks as changing process values, adjusting alarm limits, enabling/disabling alarms, acknowledging alarms, and many other functions.

Users with special data handling needs, such as event-based collection of "record-oriented" data to a relational database, recipe handling, specialized reporting, complex algorithms, and so on can leverage the power and speed of Visual Basic while maintaining live connections to InTouch data and freeing themselves from the limitations of the InTouch scripting language for more advanced applications.

InTrack users who have chosen to use Visual Basic as their GUI development environment can leverage the power of InTouch for communicating to shop floor data, while bypassing the drawbacks of DDE (single dot field at-a-time access, extra configuration, etc). Additionally, with the ActiveX automation support in the InTouch scripting language provided with InTrack, developers can perform virtually all of their scripting in a higher-level language such as Visual Basic, compile these scripts as ActiveX automation servers, and call them from InTouch.

There are double benefits to InTrack users, in that they get not only the advantages of the Visual Basic programming language (structures, arrays, rich set of functions, and so on), but also higher InTrack transaction throughput, since Visual Basic performs "early binding" against the InTrack engine versus the "late binding" that InTouch’s scripting performs.

OEMs who have specialized application requirements that are cumbersome to implement inside the InTouch GUI environment are that are awkward and time-consuming to implement using InTouch scripting can also exploit the Tag Access objects to maximize the value of their offering integrated with the FactorySuite, and can hide and embed their own proprietary content without fear of it being copied by a competitor, which is a distinct possibility if standard scripting is used.

When developing ActiveX controls that will be used within InTouch, one advantage of using the TagLink object approach versus the property binding provided by InTouch is that tag associations can be changed "on the fly" by the application and that access to multiple dot fields can be accomplished with a single property assignment. (InTouch's property binding requires that each dot field be bound to a separate property.)

A good example would be creating a PID faceplate ActiveX control, where it would be desirable to access 15-20 dot fields in the object, such as engineering units and scaling information, alarm ranges, alarm states, and so on. Using the Tag Access objects, a single "LoopID" property could be used externally and bound to an InTouch message tag, and internally a TagLink object could be used to access all of the necessary dot fields. Conversely, using the InTouch property binding technique, 20 separate properties would need to be exposed by the ActiveX control, and tedious configuration performed to map each dot field to its associated property.

Another powerful benefit of the TagLink object is the ability to integrate it with Microsoft's Visual Basic for Applications (VBA) environment, which is built into the Microsoft Office suite of applications and other third-party products. Using the Tag Access objects allows a high-performance means for linking products such as Microsoft Excel or Word to the InTouch tag database for advanced reporting, charting, or numerical analysis.

Properties

The TagLink object exposes a number of properties that correspond to the many dot fields or attributes that comprise an InTouch tag. Many of these properties are write-able as well as readable. Some properties are useful only for specific InTouch tag types (for example, the **HiLimit** property is meaningless for a discrete tag).

The **TagName** property is the means by which a TagLink object is associated with an InTouch tag, the **TagType** property indicates the data type of the tag, and the Valid property is used to determine whether a TagLink object is currently connected to an InTouch tag.

TagName

The *TagName* property is the most important of the properties, as it is used to associate an instance of a TagLink object with a specific InTouch tag. Setting the **TagName** property causes the ActiveX server to attempt to connect to InTouch and establish a link to the specified tagname.

If the TagLink object was already active and linked to another InTouch tag, that link will be disconnected first. To "disconnect" a TagLink object without assigning a new tagname, simply assign a null string (not a null pointer) to the *TagName* property. The ActiveX server will automatically disconnect the tag link to InTouch when the TagLink object falls out of scope (for example, a TagLink object created and used within a Visual Basic subroutine).

If the tagname is not valid, or if communications cannot be established to InTouch, a trappable ActiveX error will be raised which can be handled by the container application.

Note The tagname string can be expressed as a remote reference, using the same syntax as when using remote references in InTouch animation links.

Setting the TagName Property: Example 1

The following example demonstrates how to set the **TagName** property in Visual Basic. This example also demonstrates one technique for trapping errors, such as invalid tagnames or InTouch not being active by using a specific error hander:

```
 ‘ Declare a TagLink object

Dim MyTag As TagLink

 ‘ Create the TagLink object

Set MyTag = new TagLink

 ‘ Trap any errors by jumping to our error handling code

On Error GoTo OurErrorHandler

 ‘ Attach to the tagname "Analog1"

MyTag.TagName = "Analog1"

MsgBox "The Value of "+MyTag.TagName + " is
"+Format$(MyTag.Value,"0.00")

 ‘ Attach to the tagname "Analog2"

MyTag.TagName = "Analog2"

MsgBox "The Value of "+MyTag.TagName + " is
"+Format$(MyTag.Value,"0.00")

 ‘ Deactivate the link

MyTag.TagName = ""

OurErrorHandler:
       MsgBox Err.Description
```

Setting the TagName Property: Example 2

You can create arrays of tag links to simulate array tag types that reads the values from ten analog tags, named "Analog1" to "Analog10" as shown in the following example. This example demonstrates an alternative error handling mechanism, which looks at the error code after each call that could potentially generate a trappable error.

```
Const NUMTAGS = 10

Dim AnalogTags As TagLink(NUMTAGS)

Dim I As Integer

 ‘ Use "inline" error handling

On Error Resume Next

For I=1 to NUMTAGS
       Set AnalogTags(I) = New TagLink
       AnalogTags(I).TagName = "Analog"+Format$(I)
       If Err.Number <> 0 Then
           MsgBox Err.Description
End If
```

```
Next
 ‘ Now write the values to the debug output
For I=1 to NUMTAGS
       If AnalogTags(I).Valid Then
       Debug.Print AnalogTags(I).Value
       End If
Next
```

TagType

The **TagType** property is an integer value that indicates the data type for the value of the tag specified . An enumeration is provided in the type library for the TagLink object that can be used to interpret the **TagType** property. The following table lists the possible values:

PT_UNKNOWN	=	0
PT_DISCRETE	=	1
PT_INTEGER	=	2
PT_REAL	=	3
PT_MESSAGE	=	4
PT_GROUP	=	7

In addition to describing the data type for the value of the tag, the tag type also determines the data type for certain dot fields, such as alarm limits, raw and engineering unit ranges, and others. The dot field reference table lists which elements vary depending on the tag data type.

For more information, see "Dot Field Properties" later in this chapter.

Valid

The **Valid** property is a Boolean value that indicates whether or not a link was successfully established with InTouch for the specified tag. It can be checked anytime after the **TagName** property has been set.

Dot Field Properties

This section describes some important attributes of the properties that correspond to the various dot fields supported by InTouch tags. Note that not all tag types support all of the dot fields. If application attempts to access a dot field that is not valid for the associated tag, a trappable ActiveX error will be raised.

The following table displays the list of supported fields, along with their data type and whether the property is read only or read/write. Note that the data types of certain dot fields are dependent on the tag type. Notably, the two types of analog tags, Integer and Real, use different data types for handling their value, alarm limits, and other characteristics. For Integer tags, these fields are handled as long integer values (4 byte), and for Real tags they are handled as float values (4 byte).

Note Refer to the InTouch reference information for detailed descriptions of the purpose and functionality of each supported dot field. The InTouch documentation will indicate which dot fields are support for the various tag types. Also, the TagList object does not support HistoricalTrend tag dot fields or PenID dot fields.

Dot Field Name	Data Type	Access Mode
Ack	Boolean	R/W
Alarm	Boolean	R
AlarmDevDeadband	Float	R/W
AlarmEnabled	Boolean	R/W
AlarmValDeadband	Variant (depends on tag type)	R/W
Comment	String	R
DevTarget	Variant (depends on tag type)	R/W
EngUnits	String	R
HiHiLimit	Variant (depends on tag type)	R/W
HiHiSet	Boolean	R
HiHiStatus	Boolean	R
HiLimit	Variant (depends on tag type)	R/W
HiSet	Boolean	R
HiStatus	Boolean	R
LoLimit	Variant (depends on tag type)	R/W
LoLoLimit	Variant (depends on tag type)	R/W
LoLoSet	Boolean	R
LoLoStatus	Boolean	R
LoSet	Boolean	R
LoStatus	Boolean	R
MajorDevPct	Float	R/W
MajorDevSet	Boolean	R

Dot Field Name	Data Type	Access Mode
MajorDevStatus	Boolean	R
MaxEU	Variant (depends on tag type)	R
MaxRaw	Variant (depends on tag type)	R
MinEU	Variant (depends on tag type)	R
MinorDevPct	Float	R/W
MinorDevSet	Boolean	R
MinorDevStatus	Boolean	R
MinRaw	Variant (depends on tag type)	R
Normal	Boolean	R
OffMsg	String	R
OnMsg	String	R
Quality	Long integer	R
QualityLimit	Long integer	R
QualityLimitString	String	R
QualityStatus	Long integer	R
QualityStatusString	String	R
QualitySubstatus	Long integer	R
QualitySubstatusString	String	R
RawValue	Variant (depends on tag type)	R
Reference	String	R/W
ReferenceComplete	Boolean	R
ROCPct	Long Integer	R/W
ROCSet	Boolean	R
ROCStatus	Boolean	R
SPCStatus	Boolean	R
TagType	Short Integer	R
TimeDate	Long integer	R
TimeDateString	String	R
TimeDateTime	Float	R
TimeDay	Long integer	R
TimeHour	Long integer	R

Dot Field Name	Data Type	Access Mode
TimeMinute	Long integer	R
TimeMonth	Long integer	R
TimeMSec	Long integer	R
TimeSecond	Long integer	R
TimeTime	Long integer	R
TimeTimeString	String	R
TimeYear	Long integer	R
Unack	Boolean	R
Value	Variant (depends on tag type)	R/W

Trappable Errors

The following trappable errors are generated by the TagLink object:

Value	Description
8193	"Connection To InTouch Failed"
8194	"Invalid Tag Name"
8195	"Invalid Field Name"
8196	"Read Failed"
8197	"Write Failed"
8198	"Wrong Data Type"
8199	"Tag link is not active – cannot access field"

Sample Applications

Note These sample applications have been provided as learning tools only. As such, they are not extensively documented nor is complete error handling implemented. These are not supported products, and we regret that we cannot provide technical assistance on using/modifying these sample applications. Use at your own risk.

A number of sample applications are installed with the product. The sample applications provided include:

SAMPLES\TESTAPP

InTouch Application for use with the Visual Basic Demo Applications.

SAMPLES\TAGACCESSVBADEMO

Excel Spreadsheet Accessing InTouch via Visual Basic for Applications (VBA).

SAMPLES\CIRCCHART

Visual Basic ActiveX Control that simulates a Circular Chart Recorder. This control is not automatically registered upon installation. You will need to either explicitly register it using REGSVR32 or rebuild the project using Visual Basic.

SAMPLES\LEDTEST

Visual Basic Application Demonstrating Simple Use of TagLink and DataChange Objects.

SAMPLES\WWDEBUG

General-Purpose Visual Basic Application For Interactively Viewing/Manipulating InTouch Realtime Tag Database Information – Uses TagLink, DataChange, and LHTagBrowser Objects.

Combining the DataChange Control and TagLink Object: An Example

The following simple example demonstrates how you might combine the two objects. This code monitors two tags for alarm state changes and displays a detailed message on each alarm event.

Note Displaying a message box as shown in this example should never be done in a real application, as your event handlers should do their processing and return as soon as possible, rather than performing a modal action.

```
 ‘ Declare Two TagLink objects
Dim MyTag1 As TagLink
Dim MyTag2 As TagLink
 ‘ This subroutine sets up the necessary links
Sub SetupLinks
‘ Create the TagLink objects
Set MyTag1 = new TagLink
Set MyTag2 = new TagLink
 ‘ Trap any errors by jumping to our error handling code
On Error GoTo OurErrorHandler
 ‘ Attach to the tagname "Analog1"
MyTag1.TagName = "Analog1"
 ‘ Attach to the tagname "Analog2"
MyTag2.TagName = "Analog2"
 ‘ Place these tags "on watch", ignoring value and ack status
changes
Call DataChange1.AddWatch("Analog1",false,true,false,MyTag1)
Call DataChange1.AddWatch("Analog2",false,true,false,MyTag2)
OurErrorHandler:
       MsgBox Err.Description
End Sub
```

continued

```
 ‘ This event handler responds to alarm change events
Private Sub DataChange1_AlarmStatusChanged(ByVal TagName As
String, ByVal AlarmStatus As Long, ByVal UserData As Variant)
Dim szAlarmStatus As String
SzAlarmStatus = "TagName "+ TagName + " Changed To "
If UserData.Normal Then
       szAlarmStatus = szAlarmStatus + "Normal"
Else
       If UserData.HiHiStatus Then
          szAlarmStatus = szAlarmStatus + "HiHi"
       End If
       If UserData.HiStatus Then
          szAlarmStatus = szAlarmStatus + "Hi"
       End If
       If UserData.LoStatus Then
          szAlarmStatus = szAlarmStatus + "Lo"
       End If
       If UserData.LoLoStatus Then
          szAlarmStatus = szAlarmStatus + "LoLo"
       End If
End If
 ‘ Tag on the current value
SzAlarmStatus = szAlarmStatus + " At A Value Of " +
Format$(UserData.Value)
 ‘ Display a message box
MsgBox szAlarmStatus
End Sub
```

Note It is recommended, but not required, that when the DataChange control is added to a Visual Basic form, the "Visible" property provided by Visual Basic for this control should be set to "false", since this control provides no run-time user interface.

Note It is important to note that the **AlarmStatusChanged** event is generated only when the .Alarm field of an InTouch tag changes state, such as going from any alarm condition back to normal, or from normal to any alarm condition. This event is not generated when the tag changes from one alarm state to another, such as from HI to HIHI. However, by combining the TagLink object with the DataChange object's **ValueChanged** event, full access to any of the dot fields can be achieved, allowing determination of individual alarm status, as described above.

TagBrowser ActiveX Control

The following are a few tips/suggestions for using the TagBrowser ActiveX control:

- For reference purposes, the actual name (ProgID) of the TagBrowser ActiveX control is LHTagBrowser.TagDisplay
- Be certain to size the TagBrowser ActiveX control large enough to display all of the browser components.
- Remember to call the **UpdateView** method whenever new filter expressions have been set to update the list of tags.

Properties

The following sections describe the properties of the TagBrowser ActiveX control.

AccessNameFilter

This property is a String value that, when set to anything other than an empty string (""), will filter the list of tags based on standard pattern matching rules against the AccessName assigned to the tag. For example, to display all tags in the AccessName called "ABPLC99", this property could be set using:

```
TagDisplay1.AccessNameFilter = "ABPLC99"

TagDisplay1.UpdateView
```

For more information on how to define filter strings, see "Filter Expressions" later in this chapter.

Remember to call the **UpdateView** method to refresh the display after changing filter properties.

AlarmGroupFilter

This property is a String value that, when set to anything other than an empty string (""), will filter the list of tags based on standard pattern matching rules against the AlarmGroup assigned to the tag. For example, to display all tags in AlarmGroups that begin the word "Unit" and end with the word "Utilities", this property could be set using:

```
TagDisplay1.AlarmGroupFilter = "Unit*Utilities"

TagDisplay1.UpdateView
```

For more information on how to define filter strings, see "Filter Expressions" later in this chapter.

Remember to call the **UpdateView** method to refresh the display after changing filter properties.

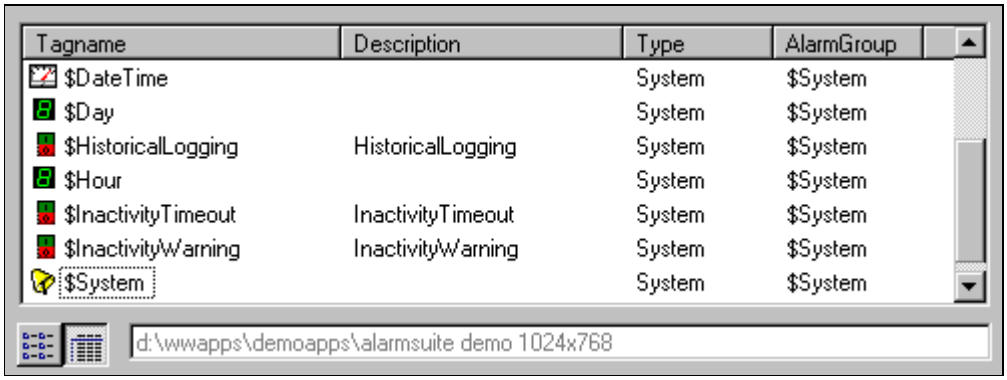
AllowBrowsing

This is a Boolean (true/false) property that, when set to true, will allow the user to double-click on the application path and browse to other InTouch application directories.

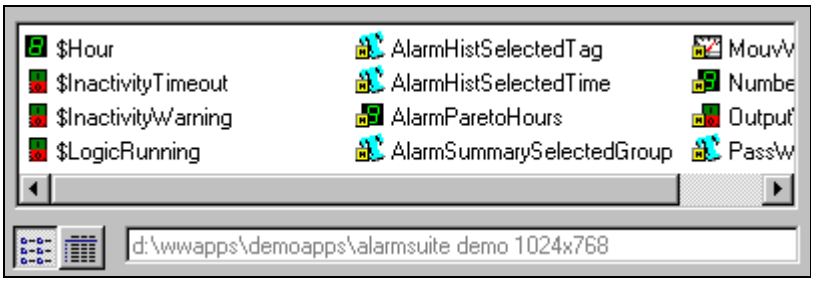
AllowViewChanges

This is a Boolean (true/false) property that, when set to true, will allow the user to click on the Report/List view icons at the lower-left corner of the control. This will switch the view between a Report view (detailed data displayed in columns) or a List view (small icons only).

The Report view is shown as follows:



The List view is shown as follows:



AppPath

This is a String property used to specify the InTouch application directory from which the tag browser will lookup tag information. For example:

```
TagDisplay1.AppPath = "D:\WWAPPS\MYAPPLICATION"
```

You can also use the **GetCurrentAppPath** method to set this value automatically based on the last InTouch application that was edited in WindowMaker or run in WindowViewer on the current computer.

If the AutoRefresh property is enabled, the **UpdateView** method will automatically be called each time the application path is changed.

AutoRefresh

This is a Boolean (true/false) property that determines whether or not the display will be automatically updated whenever the application path is changed. If the **AutoRefresh** property is enabled, the **UpdateView** method will automatically be called each time the application path is changed.

HistoricallyLoggedOnly

This is a Boolean (true/false) property that, if set to true, will display only tags that are configured to be historically logged to InTouch's standard historical logging system (for example, the **Log Data** option is selected in WindowMaker's tag editor).

LogEventsOnly

This is a Boolean (true/false) property that, if set to true, will display only tags that are configured to have events logged to InTouch’s alarm system (for example, the **Log Events** option is selected in WindowMaker's tag editor).

RetentiveOnly

This is a Boolean (true/false) property that, if set to true, will display only tags that are configured to be retentively stored.

SelectedTag

This is a String property corresponding to the currently selected tagname. You can also cause a tagname to be selected programmatically by assigning a value to this property.

SelectedTagAccessName

This is a String property corresponding to the Access Name for the currently selected tag.

SelectedTagAlarmGroup

This is a String property corresponding to the Alarm Group for the currently selected tag.

SelectedTagDescription

This is a String property corresponding to the Tag Description for the currently selected tag.

SelectedTagMode

This is a String property corresponding to the tag mode for the currently selected tag.

Valid Tag Modes: "System", "Memory", "I/O", or "?????"

SelectedTagType

This property is a read-only integer value that corresponds to the tag type for the currently selected tag.

For more information, see "Valid Tag Types" later in this chapter.

ShowAccessNames

This is a Boolean (true/false) property that, if set to true, will display the Access Names for each tag when Report View is active.

ShowAppPath

This is a Boolean (true/false) property that, if set to true, will display the currently selected application path at the bottom of the control.

TagNameFilter

This property is a String value that, when set to anything other than an empty string (""), will filter the list of tags based on standard pattern matching rules. For example, to display all tags beginning with the letters "TIC," this property could be set using:

```
TagDisplay1.TagNameFilter = "TIC*"

TagDisplay1.UpdateView
```

For more information on how to define filter strings, see "Filter Expressions" later in this chapter.

Remember to call the UpdateView method to refresh the display after changing filter properties.

TagTypeFilter

This property is an integer value that, when set to a non-zero value, will display only tags matching the specified type. To display all tags, set this property to zero (0).

For more information, see "Valid Tag Types" later in this chapter.

Valid Tag Types

The following list describes the various constants that correspond to the various InTouch tag types:

```
Public Enum enumTagType
       tagTypeAll = 0
       tagTypeIODiscReadOnly = 201
       tagTypeIODisc = 202
       tagTypeIOIntReadOnly = 203
       tagTypeIOInt = 204
       tagtypeIORealReadOnly = 205
       tagtypeIOReal = 206
       tagtypeIOMsgReadOnly = 207
       tagtypeIOMsg = 208
       tagtypeMemoryDisc = 209
       tagtypeMemoryInt = 211
       tagtypeMemoryReal = 213
       tagtypeMemoryMsg = 215
       tagtypeAlarmGroup = 217
       tagtypeGroupVar = 218
       tagtypeHistTrend = 222
       tagtypeTagID = 223
       tagtypeIndirectDisc = 224
       tagtypeIndirectAnalog = 225
       tagtypeIndirectMsg = 226
End Enum
```

Filter Expressions

The general syntax for search expressions is as follows:

String comparisons are based on a case-insensitive, textual sort order determined by your system's locale. The pattern-matching features allow you to use wildcard characters, character lists, or character ranges, in any combination, to match strings. The following table shows the characters allowed in pattern and what they match:

Characters in pattern	Matches in string
?	Any single character.
*	Zero or more characters.
#	Any single digit (0–9).
[charlist]	Any single character in charlist.
[!charlist]	Any single character not in charlist.

A group of one or more characters (charlist) enclosed in brackets ([ ]) can be used to match any single character in string and can include almost any character code, including digits.

Note To match the special characters left bracket ([), question mark (?), number sign (#), and asterisk (*), enclose them in brackets. The right bracket (]) can't be used within a group to match itself, but it can be used outside a group as an individual character.

By using a hyphen (–) to separate the upper and lower bounds of the range, charlist can specify a range of characters. For example, [A-Z] results in a match if the corresponding character position in string contains any uppercase letters in the range A–Z. Multiple ranges are included within the brackets without delimiters.

Other important rules for pattern matching include the following:

- An exclamation point (!) at the beginning of charlist means that a match is made if any character except the characters in charlist is found in string.
- When used outside brackets, the exclamation point matches itself.
- A hyphen (–) can appear either at the beginning (after an exclamation point if one is used) or at the end of charlist to match itself. In any other location, the hyphen is used to identify a range of characters.
- When a range of characters is specified, they must appear in ascending sort order (from lowest to highest). [A-Z] is a valid pattern, but [Z-A] is not.
- The character sequence [] is considered a zero-length string ("").

Methods

The following sections describe the methods of the TagBrowser ActiveX control.

GetCurrentAppPath

You can use the **GetCurrentAppPath** method to set the **AppPath** property automatically based on the last InTouch application that was edited in WindowMaker or run in WindowViewer on the current computer.

UpdateView

Calling the **UpdateView** method will refresh the display, applying any changes to the application path or filter properties that have been assigned since the last update.

Events

The following sections describe the events of the TagBrowser ActiveX control.

ApplicationChanged

This event is fired whenever the application path is changed, either programmatically or via the user selecting a new application directory.

DblClick

This event is fired whenever the user double-clicks on the control.

SelectionChanged

This event is fired whenever the user selects a new tag in the control. The selected tagname is passed as a parameter to this event. The selected tag (and other properties) are also available via the **SeletedTag**, **SelectedTagAccessName**, **SelectedTagAlarmGroup**, **SelectedTagType**, **SelectedTagDescription**, and **SelectedTagMode** properties.

Index

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W