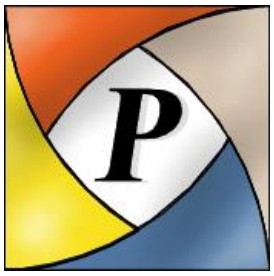
Institute for Energy Technology
OECD Halden Reactor Project



ProcSee

Graphical User Interface Management System

User Interface Controls

User's Guide
Reference Manual

1.3

This document will be subjected to revisions in the future as the development of ProcSee continues. New versions will be issued at new releases of the ProcSee system.

Published by : Institute for Energy Technology, OECD Halden Reactor Project
Date : June 2015
Version : 1.3

Contents

Introduction

This manual describes the User Interface Controls library for ProcSee. The library gives ProcSee designers an edge when developing applications using the powerful features offered by the controls and components in this library.

The User Interface Controls library is a set of graphical user interface components available on the Microsoft Windows platform. Examples of components available in this library are buttons, list boxes, scrollbars, and message boxes. These components are an official part of the ProcSee distribution and can freely be used in application development and distributed as part of system deliveries. Using these components in application development will create graphical interfaces with the familiar Microsoft Windows look and feel.

What is covered in this manual?

This manual is divided into two main parts. Part one, the user's guide, gives a detailed discussion and examples of using the different controls in ProcSee applications. It presents simple examples as well as more complex examples where performance is the main issue.

Part two is the reference manual. It gives a detailed description of all components available in the User Interface Controls library. The reference manual is divided into five chapters describing general reference manual information, .NET controls, .NET components, common interfaces and enums, respectively.

.NET framework

The User Interface Controls library uses the .NET framework from Microsoft to realize its COM components and ActiveX controls. The components in this library depend on version 2.0 or higher of the .NET framework.

All screenshots throughout this manual uses the .NET framework 2.0 from Microsoft.

Prerequisites

The User Interface Controls library requires Microsoft .NET framework 2.0. This applies to computers used at design time as well as run time.

The .NET framework can freely be downloaded from Microsoft (www.microsoft.com) and installed on your computer. Installing the .NET framework after installing ProcSee will require some steps before the components in the User Interface Controls library can be used. For more information see, *Installing the User Interface Controls library* below.

Notice that the components in this library are available only in the Microsoft Windows version of ProcSee.

Installing the User Interface Controls library

When installing ProcSee, the User Interface Controls library is normally installed and registered without errors. However, there can be situations where the .NET components fail to register, for instance if the .NET framework is missing. This section will cover the steps necessary to register the components if the initial ProcSee installation fails to install them. Alternatively, you can uninstall and re-install ProcSee.

The User Interface Controls library is created using the .NET framework. However, ProcSee does not support native .NET components. Therefore, the components must be registered as COM or ActiveX components. There are two utility tools available in the .NET framework to register .NET components as COM/ActiveX components. These tools are *regasm* and *gacutil*.

The name of the User Interface Controls library is ProcSeeControls.dll and it is located in:

%PROCSEE_DIR%\bin\winArch\components\R1.0

The tool regasm registers the .NET components in the Microsoft Windows registry and creates the type library file ProcSeeControls.tlb. This type library file is required to read the type information provided by the COM/ActiveX components. Type the following command to register the ProcSeeControls library.

```
regasm /tlb ProcSeeControls.dll
```

If running Windows 2000, the *User Interface Controls* must be registered in Microsoft Windows global assembly cache. To register the .NET components in this cache, type the following command.

```
gacutil /i ProcSeeControls.dll
```

Part I

User's Guide

2

Inserting the COM library

Developers who are using ProcSee as a tool for creating graphical user interfaces should be familiar with the COM concepts in ProcSee. This User's Guide will therefore not be a tutorial in basic COM usage in ProcSee, but instead focus on the User Interface Controls library and how these components and controls can be utilized in ProcSee application development.

The concepts and basic ideas behind using COM components and ActiveX controls in ProcSee are described in ProcSee's User's Guide and Reference Manual. Refer to these manuals for descriptions and examples of using COM in ProcSee.

ProcSee is a general and versatile tool for creating advanced graphical user interface systems. ProcSee has a lot of functionality for building complex graphical applications. However, it can be an advantage to add COM components or ActiveX controls from third-party vendors to extend functionality or decrease development time. Computers running Microsoft Windows normally have a lot of COM components registered, components that can be utilized when developing ProcSee applications. However, the COM libraries in which these components reside must be inserted into the ProcSee RTM before the components can be used. The reason for this is that ProcSee must build COM class definitions in the symbol table from the type information associated with the components.

The graphics editor, GED, is used to insert COM libraries. Right-click the *ComLibraries* node in the tree structure, and select *New* from the pop-up menu. The dialog *New COM Library* supports browsing for components and libraries registered on the computer. When components are registered, they are normally registered in one or several categories. GED has support for the category concept, which makes it easier to find correct components. Inserting the User Interface Controls library is easier than inserting other COM libraries; the next section describes systematically how to do it.

Inserting the User Interface Controls library

As the User Interface Controls library is part of the ProcSee distribution, GED offers a shortcut to insert the library. Right click on the *ComLibraries* node in the tree structure to bring up a popup menu. Select *Add* and *ProcSee UI Controls*, ref Figure 1. The user interface controls are now ready to be used in application development.

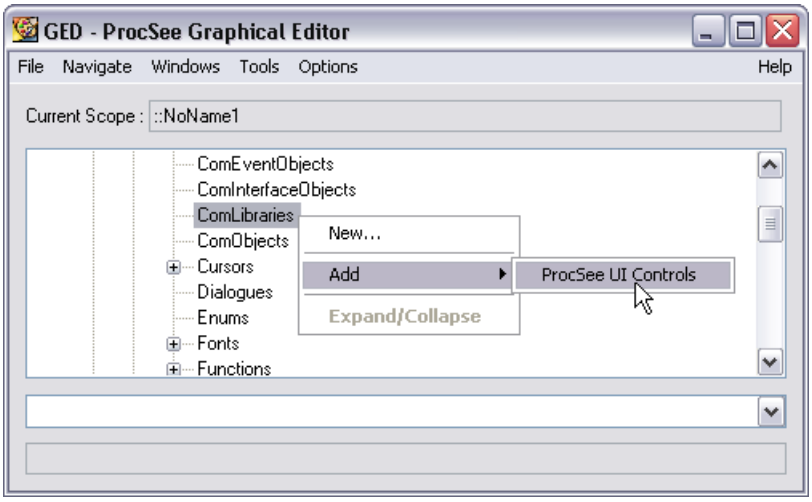


Figure 1 *Inserting the User Interface Controls library*

When inserting the library, a dialog box is displayed where the developer is asked whether to add the Utility component or not in the application, ref Figure 2. The Utility component provides functionality for file selection boxes and message boxes of different categories and is described in chapter 4. By default, the name of this component is UIUtility, but another name can be specified from the dialog. Click on the *Yes* button to add the Utility component or *No* to reject it. Note that the User Interface Controls library is inserted even if the *No* button is clicked, it is only the Utility component, which is not inserted.

Figure 2 *The dialog box displayed when inserting the ProcSee UI Controls library*

If the Utility component was not inserted as part of inserting the User Interface Controls library, it can be inserted later. To insert the Utilties component select the *ComObject* node in GED's tree structure. Right-click to bring up a pop-up menu and select *New Object...*. The *New ComObject* dialog, which lists all components added to the application or library, is displayed, ref Figure 3.

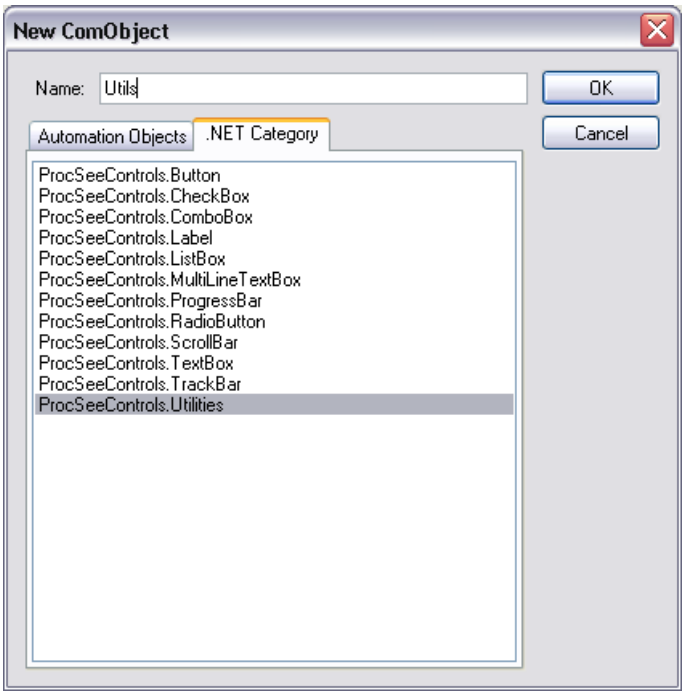


Figure 3 *The New ComObject dialog where the Utilities component is selected.*

In the *New ComObject* dialog, pick the *.NET Categories* tab and select the *ProcSeeControls.Utilities* component. In the *Name* field, type the name of the component and click the *OK* button. An instance of the Utiltities component is now inserted into the application or library and can freely be used anywhere in the application.

3

Controls

This chapter focuses on how to utilize the controls in the User Interface Controls library in application development. It starts by giving a brief introduction how to insert controls from the User Interface Controls library into a ProcSee picture or class. Next, a complete example is given, describing some basic use of these controls. The last section focuses on more advanced use.

A control, as ProcSee sees it, is a COM component with a visual appearance, also known as an ActiveX control. Such controls can be used in pictures or classes. The User Interface Controls library contains basic controls like buttons, radio buttons, check boxes, list boxes, scroll bars, track bars, etc. The full set of components is listed in chapter 6.

When finished reading this chapter the reader should be able to use the controls from the User Interface Controls library in application development. This chapter provides the reader with enough information in such a way that no deeper knowledge about using COM in ProcSee should be necessary.

Inserting User Interface controls in pictures or classes

Before inserting any control, ensure that the User Interface Controls library have been inserted to your application. Refer to section *Inserting the User Interface Controls library* on page 11 for details.

The controls from the User Interface Controls library are available from the *New ComShape* dialog in the drawing editor. The *New ComShape* dialog box is displayed when clicking on the *ComShape* icon in the shape toolbar. This toolbar is normally in a docking state to the left in the drawing area, see Figure 4.

The *New ComShape* dialog is displayed with the User Interface Controls listed in the *.NET Categories* tab. When opened, this tab is by default the first tab displayed in the dialog. To insert for instance a multi-line text input field, select *ProcSeeControls.MultiLineTextBox* from the list and click at a position in the picture or class. This action will insert the control and display it at the selected position.

When the first control in the User Interface Controls library is inserted, the library will be loaded from file and the .NET framework will initialize the COM callable wrapper (CCW) which is used by this library. This may take some time, depending on the computer's performance, but our experience is that it takes at least a few seconds.

At this point, a warning is appropriate. In ProcSee, it is possible to control where the configuration data for the ComShape is saved. The ComShape function *persist()* is used for this purpose. The argument to this function can be set to the values 0, 1 and 2. The value 0 means do not save configuration data for the ComShape. The value 1 means save the configuration data together with the ComShape as a binary stream. This is the default value for this attribute.

Setting the persist property to the value 2 will save the configuration data in a separate COM file. When the data is saved in a separate file, make sure that the ComShape and all its parent objects have names. Do not leave it anonymous, as the settings for anonymous ComShapes can not be saved to file

and reloaded. If the ComShape is part of a composite shape, like Group or Instance, no parent should be anonymous either, for the same reason. Thus, always leave the *Anonymous* check mark in the *New ComShape* off. ProcSee will automatically generate default names for the controls when this check mark is off.

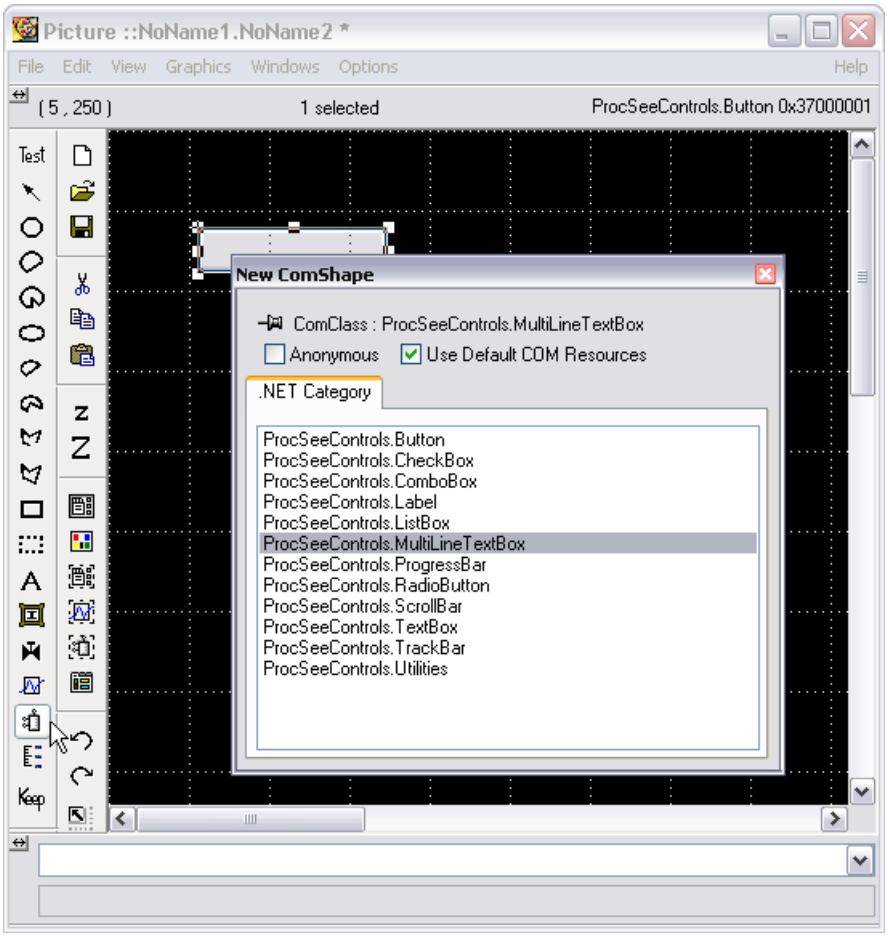


Figure 4 *The New ComShape dialog, displayed when inserting a new ComShape into a picture*

Basic use of User Interface Controls

This section shows how to use the controls in the User Interface Controls library by thoroughly describing an example using *ProcSeeControls.Button* and *ProcSeeControls.ListBox*. The full details of these controls can be found the Reference Manual part of this document, page 32 and 51. Figure 5 illustrates the example picture that will be developed in this section.

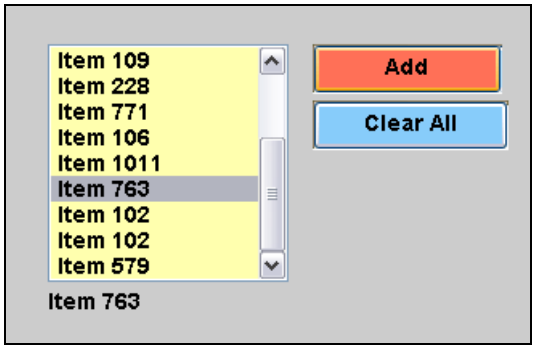


Figure 5 *The example picture developed in this section*

The picture consists of three ComShapes and one Text shape. The *Add* button is a ComShape of class *ProcSeeControls.Button*. When clicked, a new item is added to the list box. The *Clear All* button clears

the contents of the list box. When the user selects an item in the list box, the item text is displayed in the Text shape at the bottom.

Before starting on the source code example, it is necessary to clarify some basic issues concerning the ComShape implementation in ProcSee. A ComShape has one or several interfaces and normally one event interface. The interface is by default named *di* and the event interface is named *dei*. The *di* interface is used when modifying the ComShape's state or properties. Modifying a button's text or adding an item to a list box are examples of operations supported by the *di* interface. However, some of the more general properties of a ComShape, like foregroundColour, backgroundColour, width, and height, are controlled by the ComShape's attributes. Thus, common properties can be modified using GED's *Graphic Attributes* and *Selected Properties* windows. The event interface, *dei*, is used when the ComShape should call a pTALK function as a result of an operator action like clicking on a button or changing the selected item in a list box.

Now, it is time to start creating the picture. Open the *New ComShape* dialog, ref Figure 4, and insert two buttons and a listbox as shown in Figure 6. Open the *Selected ComShape* and *Selected Properties* windows and rename the ComShapes just created. Use the names csADD, csCLEARALL and csLIST, respectively. The next thing to do is to enter the texts to be displayed in the buttons. Make sure that the picture is in edit mode. Double click on the button and enter the text *Add* and *Clear All*, respectively. Click anywhere outside the button to complete entering the text.

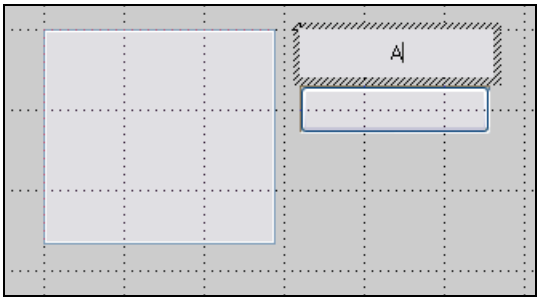


Figure 6 *The three ComShapes inserted in the picture. Entering the text of the first button*

Select the three ComShapes using the mouse and open GED's *Graphics Attributes* window. Now, press the font button in the *Graphics Attributes* window and select a font from the font list. The fonts of all three ComShapes will be changed, but for the list box the change is not visible as no items are currently added. Now, select one ComShape at a time and change their foreground and background colours from the *Graphics Attributes* window. Only the attributes foregroundColour and backgroundColour are used, not the foregroundFillColour or backgroundFillColour.

First, we create the callback function for the csADD button. The Button's event interface provides a callback function *OnClickCB* which is invoked when the operator clicks on the button (Button features are described in detail on page 32 and onwards). Select the csADD button and open the *Selected ComShape* and *Selected Properties* windows. In the *Selected ComShape* window, select *ComEventObjects* from the drop down list and select *dei* in the list box. Move the cursor to the *Selected Properties* window, press the *Functions* tab and then the *New* button, ref Figure 7. From the drop down list, select the *OnClickCB* item, as in Figure 8.

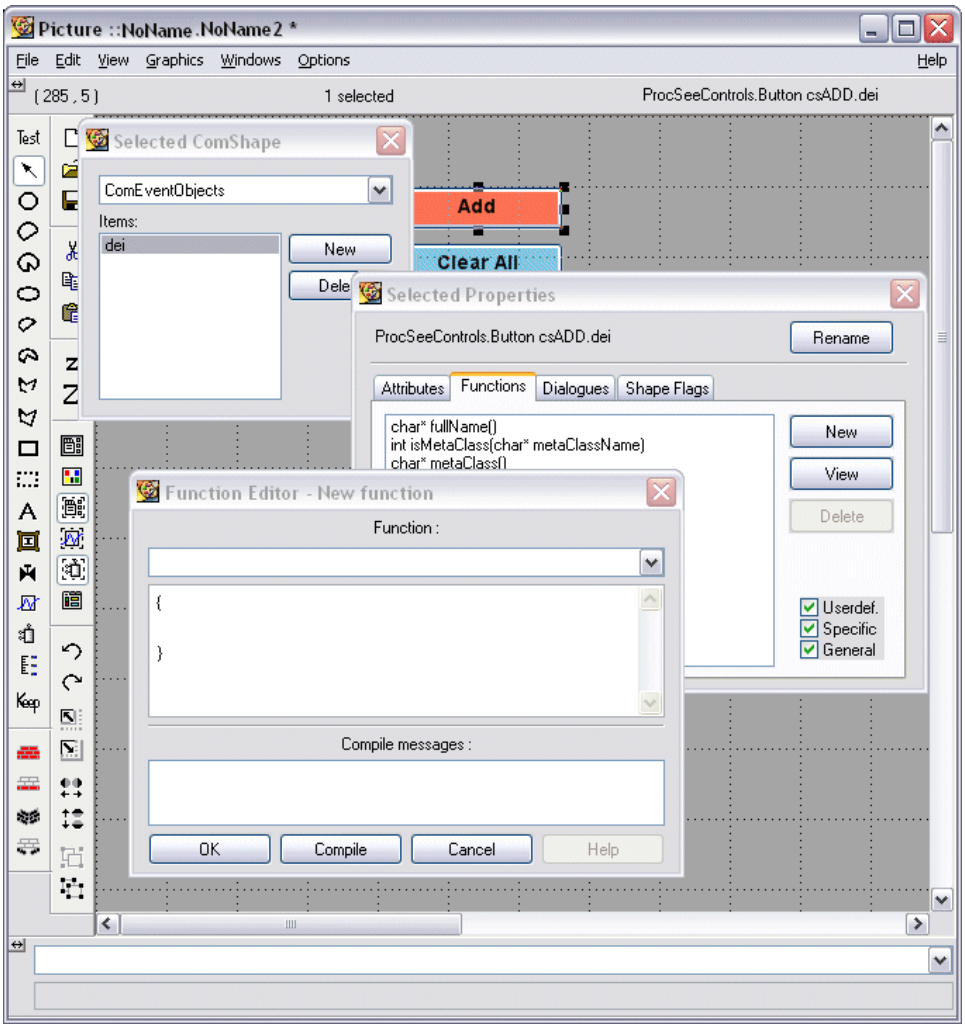


Figure 7 *This figure illustrates the dialogues Selected ComShape, Selected Properties and Function Editor used when adding callback methods.*

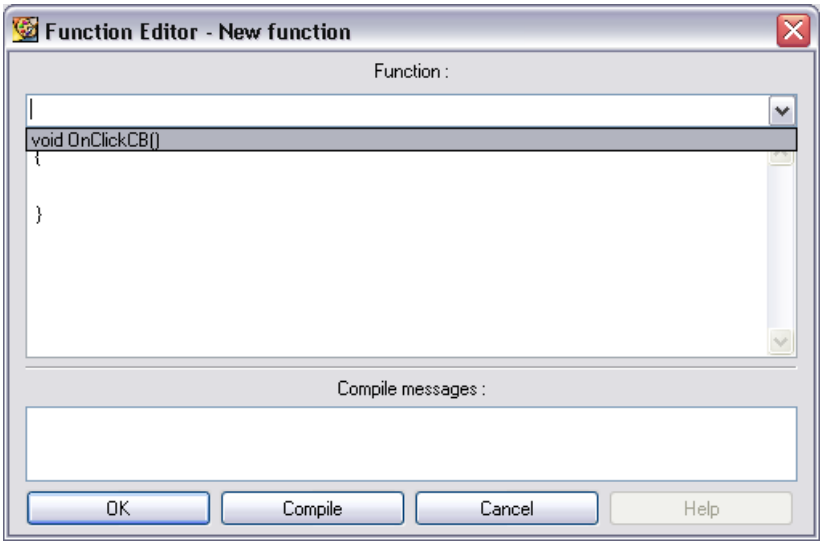


Figure 8 *The function editor used to create callback functions*

In the function body insert the following source code and press the *Compile* and *OK* button.

```
void OnClickCB()
```

```
{
   char* theItemText = newCharArray( 64 );

   sprintf( theItemText, "Item %d", random( 1024 ) );
   csLIST.di.AddItem( theItemText );
}
```

This source code uses the member function *AddItem* to add random text items to the list box, csLIST. The ListBox's methods and properties are listed in the *Selected Properties* window, and all ListBox features are described in detail on page 51 and onwards in this manual.

Follow the same steps as described above to create the callback function for the *Clear All* button. But, add the following code instead. This code will clear all items in the list box.

```
void OnClickCB()
{
    csLIST.di.Clear();
}
```

In this example, a Text shape below the list box displays the selected item. Insert a Text shape and call it tsSELECTED. Set the attribute *theText* equal to an empty string, "", to indicate that no item has been selected.

In order to get the desired functionality, a callback function must be implemented for the list box when the operator selects an item from the list. For the csLIST, follow the same steps as described when creating callback functions for the buttons, but OnSelectChangedCB instead of OnClickCB. Add the following source code in the callback function body. This source code updates the Text shape tsSELECTED with the currently selected item in the list box.

```
void OnSelectChangedCB()
{
    tsSELECTED.theText = di.SelectedItem( 0 );
    tsSELECTED.updateShape();
}
```

Finally, to finish the example the text displaying the selected item should be cleared when the *Clear All* button is clicked. Therefore, append the following code in the body of the csCLEARALL callback function *OnClickCB*.

```
…
   tsSELECTED.theText = "";
   tsSELECTED.updateShape();
…
```

Advanced use of User Interface Controls

More advanced use of the controls in the User Interface Controls library is covered in this section. It starts with describing how to use radio button groups, continues with enumrations and performance, and ends with a description of how to move input focus automatically when using text boxes.

RadioButton groups

Figure 9 illustrates two logical groups, *English Soccer Teams* and *Italian Soccer Teams*. The logical groups work independent of each other. For instance, clicking on the radio button Chelsea will leave the radio buttons in the *Italian Soccer Teams* unaffected, but will unselect the radio button Liverpool. This section describes how to implement this kind of behaviour using RadioButton controls. There are two different methods, which can be applied to create radio button groups. Which method to use is up to you? The first method uses ProcSee's Group shape and the second method uses the IRadioButton property GroupId.

Figure 9 *Two separate RadioButton groups.*

Start by inserting several RadioButton controls into the picture using the *New ComShape* dialog. Arrange the radio buttons as illustrated in Figure 9, where *English Soccer Teams* and *Italian Soccer Teams* are visually divided in two groups. Double click on each radio button in turn and type a team-name for each. Switch to test mode for the picture and test the behaviour of the radio buttons. The buttons do not behave as expected, they act as if they were one single group.

Turn the picture back into edit mode. It is time to separate the radio buttons into two logical groups. This paragraph demonstrates the first method where the Group shape is used to separate the radio buttons. Use the mouse to select the radio buttons belonging to the first group. From the menu bar, select *Edit* and then *Group* to create a Group shape. Note that the persist property should be left as is. If this property is set to the value 2 with the function *property()*, then give the Group shape a name using the *Rename* button in the *Selected Properties* window. It is very important to name each parent where ComShape objects are used when the persist property is 2. The object settings will otherwise be lost when reloaded from file. The same procedure must be applied to the other radio buttons. The picture contains two groups where each group contains several radio buttons. Set the picture in test mode and click on the radio buttons. The radio buttons now behave as expected because the radio buttons are separated into two logical groups.

The second method uses the RadioButton property *GroupId* to create logical groups. The prerequisite is that the radio buttons are in the same scope, for instance at the picture level. The default value of the *GroupId* property is zero. To continue with the example above, ungroup the radio buttons to cancel the effect achived using the Group shapes. Now, for all radio buttons in *English Soccer Teams* set the property *GroupId* to 1 for the *English Soccer Teams*, and value 2 for the *Italian Soccer Teams.* Set the picture in test mode and click on the radio buttons. It works as expected.

Enumerations

Some of the controls in the User Interface Controls library offer collections of items, numbers, etc. These collections are very often returned in an interface called IEnumAny. This is an enumeration interface, which provides functionality to enumerate through items in a collection. The IEnumAny interface is predefined and is always available at the global scope.

It is necessary to pinpoint the importance of the cast operator when methods return IUnknown interfaces. This is due to some COM issues which are outside the scope of this manual. The next source code snippet will compile but fail at run-time because the interface will not be of the expected type.

```
::IEnumAny* ea = LB.di.Items();  // FAILS AT RUN-TIME
```

The following source code illustrates how to obtain an interface of the expected type that will both compile and run without errors. Note the case operator (::IEnumAny *).

```
::IEnumAny* ea = (::IEnumAny *)LB.di.Items();
```

The following example uses the IEnumAny interface to enumerate through the selected items in the argument *listBox*. The IEnumAny interface method *next* returns the next item in the list. The argument to this function is of type *any**, which means that this function can return any simple data type, i.e. int, float, char*, etc. In the following example, the type of the argument is *char**, which is the data type returned in the selected items collection. The return value from this pTALK function is a List object containing the selected items.

```
::List* GetSelectedItems( ::ProcSeeControls.ListBox* listBox )
{
   ::List* result = newList();
   ::IEnumAny* ea = (::IEnumAny *)listBox->di.SelectedItems();

   char* name;
   while ( ( ea->next( &name ) ) != 0 )
      result->add( name );

   return result;
}
```

ListBox performance

The ListBox offers two different methods to boost performance when adding a large number of items. These methods are examined thoroughly in this section. When performance is important, it can be worth the effort to get familiar with both these methods, and to be aware of the pros and cons of using them. Note that this explanation only applies when a large number of items are added to the control. Use the IListBox methods *AddItem* or *InsertItem* as is if only adding a few items.

The first method explained is the simplest and the easiest to use. Before entering the for loop, where the items are added, call the IListBox method *BeginUpdate*. When items are added, this method will prevent the ListBox control from being redrawn. The redrawing of the control will be suspended until the *EndUpdate* is called. The source code example below uses this strategy when adding the items found in the ::List collection. These two method calls will boost performance when adding a large number of items compared to the situation where they are skipped. Hovewer, remember to call the *EndUpdate* method. Forgetting to call *EndUpdate* will forever prevent the control from being refreshed. This is the main disadvantage of using this method.

```
void AddItems( ::List* items )
{
    csLIST.di.BeginUpdate();
    for ( int i = 0; i < items->length(); i++ )
    {
         csLIST.di.AddItem( items->get( i ) );
    }
    csLIST.di.EndUpdate();
}
```

The next method uses the collection interface IItemCollection to add items to the ListBox control. This method has proven to give best performance, but it is more difficult to use compared to the method described above. First, before entering the for loop a new IItemCollection object must be created. The IItemCollection member function NewCollection creates and returns a pointer to a new collection. Note that a cast operator must be applied to convert the interface to the correct type. The for loop simply adds the items in the List object to the IItemCollection object. When all items have been added to the collection, the IListBox method AddCollection is called where the argument is the IItemCollection object. This function will copy all items in the collection object into the ListBox control.

```
void AddItems( ::List* items )
{
   ProcSeeControls.IItemCollection* c;
   c = (ProcSeeControls.IItemCollection *)csLIST.di.NewCollection();

   for ( int i = 0; i < items->length(); i++ )
   {
      c->AddItem( items->get( i ) );
   }

   csLIST.di.AddCollection( c );
}
```

Automatically switching keyboard focus to the next text input field

Automatically switching keyboard focus to the next input field when pressing the tab key is a common behaviour when using applications targeted for the Microsoft Windows operating systems. It is much faster for a user to move the input focus to the next input field using the keyboard than using the mouse. This kind of functionality has therefore been added to some of the controls in the User Interface Controls library.

In the example demonstrated here the input focus is automatically moved to the next text input field when the tab key is pressed. The example created contains four text boxes as shown in Figure 10. The names of these text boxes are csX, csY, csWIDTH and csHEIGHT, respectively. When the tab key is pressed, the input focus moves from csX to csY, from csY to csWIDTH and so on.

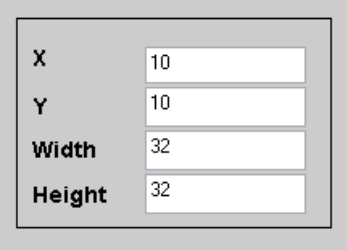


Figure 10 *Four TextBox controls where the tab key can be used to move keyboard focus to the next control*

First, insert the four text boxes by selecting TextBox in the *New ComShape* window in the drawing editor. Open the *Selected Properties* window and the *Selected ComShape* window. Select the ComEventObject from the drop down list in the *Selected ComShape* window and pick the *Functions* tab in the *Selected Properties* window. Next, give the four TextBox controls the names stated above. Start with the first TextBox shape called csX. Select this shape in the picture. In the *Selected ComShape* window select the dei interface. The *Selected Properties* window will then automatically be updated.

Now, click on the New button in the *Selected Properties* window. Press the drop down list and select the callback function OnTabCB. Add the following source code in the body of this callback function. This code will automatically set the input focus to the next TextBox control, in this case the control called csY.

```
void OnTabCB()
{
    csY.di.SetFocus();
}
```

Follow the steps above to add the OnTabCB callback functions for the remaining TextBox controls. But remember to replace the name of the TextBox control to jump to. csX is moved to csY, csY is moved to csWIDTH, csWIDTH is moved to csHEIGHT and finally csHEIGHT is moved to csX.

4

Utilities

The Utilities component is a collection of miscellaneous functionality, like message boxes, file selection boxes, etc. Normally, this component is used to provide the user with dialogues, which the user must respond to before the application can continue executing pTALK code, for instance how to act to an error situation. An example could be to display an error message dialog if a set point value entered was outside a legal value range. These kind of interactive message boxes are called modal dialogs in Microsoft Windows terminology.

The functionality in the Utilities component can be separated into two main categories, simple and advanced use. The simple use is covered in the section, *Basic use of the Utilities* component, while the more general and advanced use is covered in the sections *Using the general MessageBox* interface and *Using the general file selection interface*.

Basic use of the Utilities component

Before using the Utilities component, ensure that the User Interface Controls library and the Utilities component have been inserted to your application. Refer to section *Inserting the User Interface Controls library* on page 11 for details.

It is very easy to use the basic functionality offered by the Utilities component. To display an error message box for instance, call the instance method *ShowErrorMessageBox*. This method accepts two arguments, the error message and the message box title. This function call will display an error message box on the screen where the user must click the OK button to continue.

To be more specific, all basic functions in the Utilities component have the prefix Show, like ShowInformationMessageBox and ShowFileOpenDialog. These functions display the dialogs directly and are easy to use. The properties needed to customize the dialogs are provided as function arguments.

The following example demonstrates how to use the ShowErrorMessageBox, which displays an error message when the user enters an illegal value. All examples in this chapter use the name UIUtility for the Utilities component. When inserted from GED, the component will be given this name by default.

```
void DisplayError( char* errorMessage )
{
   UIUtility.di.ShowErrorMessageBox( errorMessage, "Error" );
}

…
   if ( inputError )
   {
     DisplayError( "The value must be in the range [0-100]" );
   }
…
```

All the basic message box functions can be called in the same way as illustrated in the source code snippet above. These *Show…* functions take two arguments, the message to display and the text to display in the caption bar of the message box. The following is a list of the basic message box functions provided by the Utilities component.

- ShowMessageBox
- ShowErrorMessageBox
- ShowInformationMessageBox
- ShowQuestionMessageBox
- ShowWarningMessageBox

The Utilities component also provides basic functionality to display file open and file save dialogues. These functions are called ShowFileOpenDialog and ShowFileSaveDialog, respectively. These functions accept the same number and the same type of arguments, which are the initial directory and the file filter. The return value from these functions is the name of the selected file if the OK button was clicked or the null value if the Cancel button was clicked.

The following example illustrates how a user may use the file open dialog to select image files of type bmp, tiff and png. Notice the special format of the filter argument. A single filter is composed of two strings separated by vertical bars, '|'. To view for instance files of type bitmap, bmp, the text to the left of the first vertical bar is the text appearing in the *file of types* combo box in the file dialog. The text to the right of the vertical bar is the actual filter string used by the file dialog to list files of a given type. In the following example is the first filter string *.bmp. Further, notice the use of the functions fileNameHost and fileNameNormal. The fileNameHost function ensures that the directory argument to ShowFileOpenDialog is in Microsoft Windows format, and the fileNameNormal function ensures that the image shape's fileName-attribute is in ProcSee's internal format.

The file dialog returns the selected file if the user clicks the OK button. In the example the selected image file will be displayed in an Image shape, in this source code called IS.

```
void SelectFile( char* directory )
{
   char* filter =
"Bitmap(*.bmp)|*.bmp|Tiff(*.tiff;*.tif)|*.tiff;*.tif|Ping(*.png)|*.p
ng"

   char* dh = fileNameHost( directory );
   char* fn = UIUtility.di.ShowFileOpenDialog( dh, filter );

   if ( fn )
   {
      IS.fileName = fileNameNormal( fn );
      IS.updateShape();
   }
}
```

Using the general MessageBox interface

The general message box interface can be used where the simple *Show…* message box functions does not provide the necessary functionality. The interface offers more functionality compared to the Show methods, but is more difficult to use. The Utilities member function MessageBox returns a pointer to an interface named IMessageBox. This interface has methods which offer functionality to tailor-made the message box to the application.

The general message box can theoretically be divided into four main parts which can be set individually by methods of the IMessageBox interface, see Figure 11. The main parts are:

- Caption text (the text to display in the caption bar of the message box)

- Message text (the text to display in the message box. Multi line text are supported, using '\n')
- Icon (the icon to display or no icon)
- Button row (controls the number of buttons and the default button)

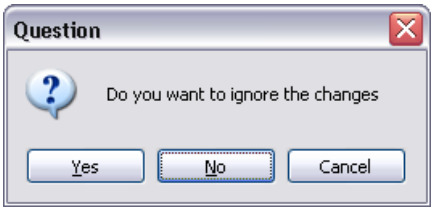


Figure 11 *This figure shows the general message box*

The source code below illustrates an example where the message box in Figure 11 is displayed. The first source code line noticeable is that the interface returned from the Utilities member method MessageBox must be converted to the correct interface type, IMessageBox, using the cast operator. It must be converted to the correct type because of some COM issues, which are outside the scope of this manual.

The next source code lines uses the enum constants defined in the User Interface Controls library to control the icon, buttons and default button, respectively. The next lines of source code put the text to display to the user in the caption bar and the message to display to the user. Finally, the last line in the function DisplayIgnoreQuestion displays the message box and returns the status of which button was clicked by the user.

The result of the DisplayIgnoreQuestion function call is defined in the enum constants *ProcSeeControls.DialogResult*. See the example in the source code below where the switch statement is used to determine which of the buttons were clicked.

```
int DisplayIgnoreQuestion()
{
   // Get the message box interface.
   ProcSeeControls.IMessageBox* mb;
   mb = (ProcSeeControls.IMessageBox *)UIUtility.di.MessageBox();

   // Set the icon, button and default button.
   mb->Icon    = ProcSeeControls.MessageBoxIcon.Question;
   mb->Buttons = ProcSeeControls.MessageBoxButtons.YesNoCancel;
   mb->DefaultButton =
                 ProcSeeControls.MessageBoxDefaultButton.Button2;

   // Set the text to display to the user.
   mb->Caption = "Question";
   mb->Text    = "Do you want to ignore the changes";

   return mb->ShowDialog();
}

…

   switch ( DisplayIgnoreQuestion() )
   {
     case ProcSeeControls.DialogResult.Yes:    … break;
     case ProcSeeControls.DialogResult.No:     … break;
     case ProcSeeControls.DialogResult.Cancel: … break;
   }
…
```

Using the general file selection interfaces

The Utilities component provides two interfaces for selecting files, IFileOpen and IFileSave. As the names states, the IFileOpen interface is used to open a file open dialog and IFileSave a file save dialog. These interfaces offers more advanced features than the ShowFileOpenDialog and ShowFileSaveDialog methods can provide. For instance, selecting multiple file names is a feature offered by the IFileOpen interface, which is not part of the ShowFileOpenDialog method.

The drawback when using the IFileOpen and IFileSave interfaces is that these interfaces are more difficult to use compared to the *Show…* file dialog functions. The Utilities member methods FileSave and FileOpen return the IFileSave and IFileOpen, respectively. Notice that these functions return pointers to the actual interfaces, but due to some COM issues, which are outside the scope of this manual, a cast operator must be applied. See the source code examples below how to obtain interfaces to the file open and file save dialogs, respectively.

```
…
   // Get a pointer to the file open dialog
   ProcSeeControls.IFileOpen* fo;
   fo = (ProcSeeControls.IFileOpen *)UIUtility.di.FileOpen();
…
```

```
…
   // Get a pointer to the file save dialog
   ProcSeeControls.IFileSave* fs;
   fs = (ProcSeeControls.IFileSave *)UIUtility.di.FileSave();
…
```

As the interfaces for opening and saving files are quite equal in functionality, this section mainly focuses on the file open dialog. The source code example presented here uses the file open dialog. Users familiarized with the IFileOpen interface should have no problem using the IFileSave interface. For a detailed description of the methods and properties available in the IFileSave interface, see description of the IFileSave interface on page 102.

The remainder of this section focuses on an example where the final result is a function displaying a file open dialog where the user can select multiple files of type png (Ping image files). Figure 12 shows the file open dialog developed here where multiple ping image files have been selected.

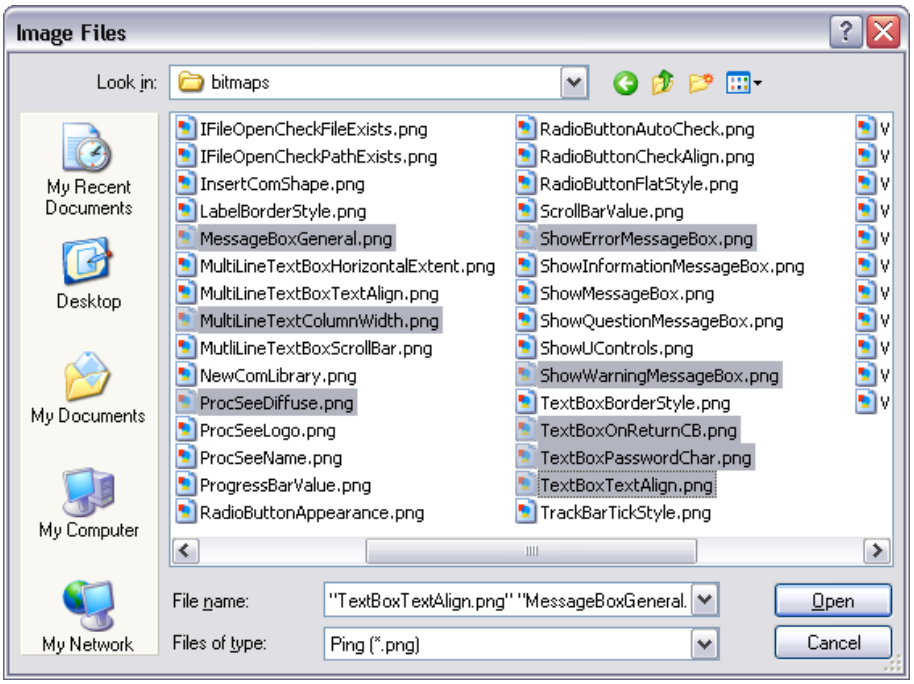


Figure 12 *The file open dialog created in this section where the user has selected multiple image files of type ping*

The file names selected in the file open dialog are returned from the pTALK function in a ::List object. The complete source code of the pTALK function is presented at the end of this section. The example starts with the prototype of the pTALK function.

```
::List* GetFiles( char* dir, char* filter, char* ext, char* title )
{
  ...
}
```

, where the parameter dir is the initial directory of the files to list. The parameter filter is the type of files to view. The parameter ext is the default extension added to the file if the extension is missing. The parameter title is the title to display in the caption bar of the file dialog.

Now, it is time to study the contents of the GetFiles function. First, this function can profitably be separated into four parts. Each source code part will be explained thoroughly in this section. The main parts are:

- Obtaining a pointer to an IFileOpen interface.
- Configuration of the IFileOpen interface.
- Displaying the file open dialog.
- Getting the files selected in the file open dialog.

The first part for obtaining a pointer to an IFileOpen interface has already been explained in detail. Next, is how to configure the IFileOpen interface before it is displayed? It is worth notice that the property MultiSelect is set to true. This property will allow multiple selections of file names in the file open dialog. The Title, DefaultExt, Filter and InitialDirectory are initialized with the parameters passed in the arguments to the GetFiles function. Notice the use of the fileNameHost function. For an explanation of the properties listed in the source code snippet below, see the section IFileOpen on page 102.

```
...

   fo->AddExtension     = 1;
```

```
   fo->CheckFileExists  = 1;
   fo->CheckPathExists  = 1;
   fo->MultiSelect      = 1;
   fo->Title            = title;
   fo->DefaultExt       = ext;
   fo->Filter           = filter;
   fo->InitialDirectory = fileNameHost( dir );
…
```

To display the file open dialog the IFileOpen member function ShowDialog is used. Calling this function will display the file open dialog. The source code for displaying the dialog is illustrated next. The file open dialog is displayed with two buttons, OK and Cancel. The ShowDialog function returns which button were clicked. An enum is defined in the User Interface Controls library which defines the constants returned by all dialogs. In this example a check is made to ensure that the OK button was clicked. If not the GetFiles function returns the value null.

```
…
   if ( fo->ShowDialog() == ProcSeeControls.DialogResult.Cancel )
      return 0;
…
```

Finally, to obtain the files selected in the file dialog an enumeration must be created. The interface ::IEnumAny should be used to enumerate through the selected files.

To continue with the source code example, a ::List object is created with the pTALK function newList. This ::List object is used as a container for the file names selected in the file open dialog. This ::List object will be the return value from the GetFiles function. The next line of source code obtains the ::IEnumAny interface from the IFileOpen interface method FileNames. Notice the cast operator used when returning the file names. The next two lines of code gets the filenames from the file open dialog and puts the filenames into the ::List object.

```
   char*       fileName;
   ::List*     fileList = newList();
   ::IEnumAny* fileEnum = (::IEnumAny *)fo->FileNames();

    while ( ( fileEnum->next( &fileName ) ) != 0 )
         fileList->add( fileNameNormal(fileName) );
```

All the main parts of the GetFiles source code example have been explained. It is now time to present the complete source code example of this pTALK function.

```
::List* GetFiles( char* dir, char* filter, char* ext, char* title )
{
   // Get interface pointer
   ProcSeeControls.IFileOpen* fo;
   fo = (ProcSeeControls.IFileOpen *)UIUtility.di.FileOpen();

   // Configure the dialog
   fo->AddExtension     = 1;
   fo->CheckFileExists  = 1;
   fo->CheckPathExists  = 1;
   fo->MultiSelect      = 1;
   fo->Title            = title;
   fo->DefaultExt       = ext;
   fo->Filter           = filter;
   fo->InitialDirectory = fileNameHost( dir );

   // Display dialog. Return if the user clicks Cancel
   if ( fo->ShowDialog() == ProcSeeControls.DialogResult.Cancel )
      return 0;
```

```
   // Obtain the selected files
   char*       fileName;
   ::List*     fileList = newList();
   ::IEnumAny* fileEnum = (::IEnumAny *)fo->FileNames();
    while ( ( fileEnum->next( &fileName ) ) != 0 )
         fileList->add( fileNameNormal(fileName) );

   return fileList;
}
```

Part II

Reference Manual

5

About the Reference Manual

The reference manual is divided into four chapters. Each chapter describes different parts of the reference manual. The chapters describe the following topics:

- .NET controls
- .NET components
- .NET common interfaces
- .NET enums.

Components available as ComShapes are in this context called .NET Controls. These controls have a visual appearance and can be inserted into and used in pictures just like any other shape. The controls included in the library are described in the chapter .NET Controls.

.NET components are available in ProcSee as ComObjects. These components do not have a visual appearance. The components are described in the chapter, .NET Components.

The functionality provided by a component is available through the interfaces it supports. Each interface is described together with the component it is associated with. The interfaces may have both properties and functions. Properties are just like attributes in ProcSee, i.e. they can be located at the left and the right side of the assignment operator. Notice however that some properties are read or write only. There are some interfaces, which are used by several components. These interfaces are described in a separate chapter, .NET Common Interfaces.

Most of the properties and functions supported by a control or component have almost unrestricted value range. For instance, the minimum and the maximum properties of a TrackBar can be set to any legal integer value. However, some properties and functions support only a limited set of legal values. In the User Interface Controls library, the legal values are provided as enums. These enums are described in the chapter .NET Enums.

6

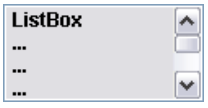
.NET Controls

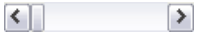
Components in the User Interface Controls library with a visual appearance are called .NET controls. These controls can be inserted as ComShapes in classes and pictures. Each control will be described in detail in this chapter. Source code examples are provided to demonstrate and illustrate functionality.

Almost every control in the User Interface Controls library supports interfaces of two types, 1) interfaces used to manipulate the control and 2) event interfaces. The latter interface are also called callback interfaces or source interfaces. Note that in the User Interface Controls library all event interfaces are postfixed with the name Event. In ProcSee the default interfaces are created automatically when a control is inserted in a picture. The names of these interfaces will be *di* for the interface, and *dei* for the event interface.

An example will clarify what an interface is, why using it and how to use it. Imagine a picture containing a Button control from the User Interface Controls library. The Button control provides two interfaces, IButton and IButtonEvent. The interface IButton is used to change the visual appearance of or modifying the Button control. This interface contains methods and properties. For instance, the property Text is used to set the label on the Button control. An event is generated when clicking with the mouse on the Button control. When fired, this event will call a designer-implemented pTALK function, called OnClickCB. Refer to the section *Basic use of User Interface Controls* on page 15 for a detailed example on the basic use.

The following table lists the .NET controls available in the User Interface Controls library. Each of these controls is described later in this chapter.

.NET Control	Description	Preview
Button	This control represents a standard button which the user can click to perform an action.	
CheckBox	This control represents a standard check box. The check box can be in two or three states, checked, indeterminate or unchecked.	
ComboBox	This control represents a standard combo box where the user can select among a set of predefined items from a drop down list.	
Label	This control represents a standard label. The label can be left, center or right aligned.	
ListBox	This control represents a standard list box. The list box can be in single or multi select mode where the user can select items in the list.	

MultiLineTextBox	This control represents a standard multi line text input field.	
ProgressBar	This control represents a standard progress bar. It is normally used to indicate a lengthy operation, for instance in a for loop in a pTALK function.	
RadioButton	This control represents a standard radio button. Several radio buttons can be grouped together where only one radio button in the group can be selected.	
ScrollBar	This control represents a standard scroll bar where the user can select values between a minimum and maximum value by clicking on the up- or down arrow, or moving the slider.	
TextBox	This control represents a standard single line text input field. It supports password text input.	
TrackBar	This control represents a standard track bar. The user can select values between a minimum and maximum value by moving a slider.	

Button

The Button control represents a standard Microsoft Windows button.

The Button control supports two interfaces, IButton and IButtonEvents. The IButton interface is by default named *di*, and is used to modify the contents of or change the visual appearance of the Button control. The event interface, IButtonEvents, is by default named *dei*.

Note that the IButton interface does not provide properties or methods to change the Button's font or colours. This functionality instead supported by the ComShape attributes theFont, foregroundColour and backgroundColour. The position and size of the Button are controlled by the ComShape attributes x, y, width and height, respectively.

Double clicking on a Button control while the picture is in edit mode will put the control into user input mode. In this mode the Button control will display a text input field where the user can enter the button text. Clicking outside the control will put the Button back to normal mode and the text entered will be displayed in the control.

IButton

The IButton interface has the following properties.

Property	Description
Enabled	Whether the button can be clicked or not.
FlatStyle	This property controls the style of the button.
Text	The text displayed in the button.
TextAlign	Controls the alignment of text in the button.

P r o p e r t i e s :

int IButton::Enabled

This property must be set to true if the Button control should respond to user interaction, otherwise false. To restrict the Button control from being used, set this property value to false.

The default value for this property is true.

---

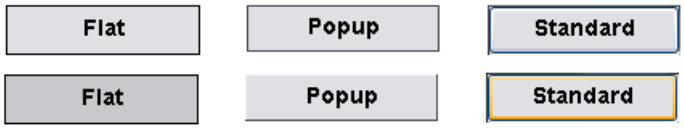
int IButton::FlatStyle

This property determines the visual appearance of the Button control. It can be set to the values flat, popup or standard. These values are defined in the enum FlatStyle.

The default value for this property is Standard.

Example

The first row illustrates the different FlatStyle enum constants used when the buttons do not have focus. The second row illustrates the same buttons with focus.



See Also

FlatStyle enum constants are described on page 114.

---

char* IButton::Text

This property gets or sets the text in the Button control.

Example

The following example changes the text in a Button shape. The name of the shape is bClear. The text in the control depends on the value of the attribute aClearAll.

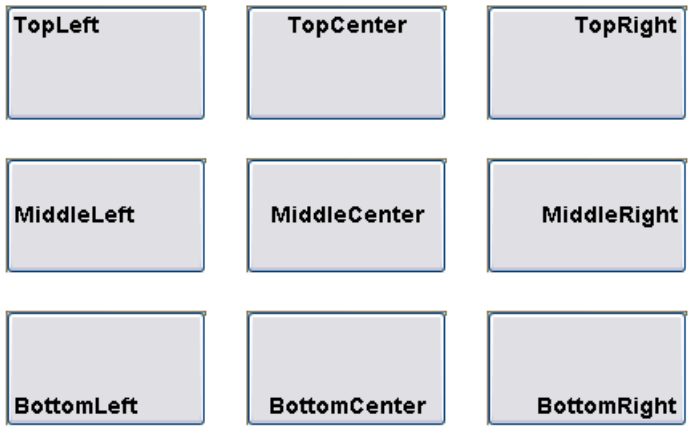
```
…
  if ( aClearAll ) bClear.di.Text = "Clear All";
  else             bClear.di.Text = "Clear";
…
```

---

int IButton::TextAlign

This property controls the alignment of the text in the Button. There are nine different positions in the Button where the text can be aligned. These positions are defined in the enum constant ContentAlignment. The default value for this property is MiddleCenter.

Example



See Also

ContentAlignment enum constants are described on page 114.

IButtonEvents

This is the source interface for the Button control. The IButtonEvent interface has the following methods.

Function	Description
OnClickCB	Fired when the Button is clicked

F u n c t i o n s :

---

void IButtonEvents::OnClickCB()

This call-back function is fired when the Button is being clicked.

Example

The following example displays the picture picCORE in window winMAIN when the button is being clicked.

```
void OnClickCB()
{
     displayPicture( "winMAIN", "pictures/picCORE.ppic" );
}
```

CheckBox

The CheckBox control represents a standard Microsoft Windows check box. A check box is a graphical component that can be in two or three different states. When clicked the check box changes state from checked to unchecked, or vice versa. A three state check box also changes state to an indeterminate state between checked and unchecked state.

The CheckBox control supports two interfaces, ICheckBox and ICheckBoxEvents. The ICheckBox interface, by default named *di*, is used to modify the contents of or change the visual appearance of the CheckBox control. The event interface, ICheckBoxEvents, is by default named *dei*.

Note that the ICheckBox interface does not provide properties or methods to change the CheckBox's font or colours. This functionality instead supported by the ComShape attributes theFont, foregroundColour and backgroundColour. The position and size of the CheckBox are controlled by the ComShape attributes x, y, width and height, respectively.

Double clicking on a CheckBox control while the picture is in edit mode will put the control into user input mode. In this mode the CheckBox control will display a check box mark together with a text input field. The user can type the check box text in the text input field. To set the initial state of the check box, click on the check box mark. Clicking outside the control will put the CheckBox back to normal. The text entered in the input field and the check box mark will be displayed in the control.

ICheckBox

The ICheckBox interface has the following properties.

Property	Description
Appearance	Controls the appearance of the CheckBox.
AutoCheck	Determines whether the CheckBox automatically changes state or not.
CheckAlign	Controls the alignment of the check box on the CheckBox control.
CheckState	Determines the state of the CheckBox, checked, indeterminate or unchecked.
Enabled	Whether the control can be clicked or not.
FlatStyle	Controls the style of the CheckBox.
Text	The text displayed in the CheckBox.
TextAlign	Controls the alignment of the text on the CheckBox control.
ThreeState	Determines whether the CheckBox allows two or three states.

P r o p e r t i e s :

int ICheckBox::Appearance

This property controls the appearance of the CheckBox. A CheckBox can visually be displayed like a button or as a normal check box. The Appearance enum constant controls the appearance of the CheckBox.

The default value for this property is Normal.

Example

See Also

Appearance enum constants are described on page 113.

int ICheckBox::AutoCheck

This property determines whether the CheckBox automatically changes the checked state when it is clicked. If this property is false the CheckBox state must be set manually. The event function OnClickCB is fired when CheckBox is clicked. To manually change the state of a CheckBox, use the property CheckState in the OnClickCB call-back function.

The default value for this property is true.

Example

The following example demonstrates how to change a CheckBox's state when the AutoCheck property is false. This example implements the OnClickCB event function. The CheckBox's state is changed only if the variable Running is true.

```
void OnClickCB()
{
   if ( Running )
   {
      if ( di.CheckState == ProcSeeControls.CheckState.Checked )
         di.CheckState = ProcSeeControls.CheckState.Unchecked;
      else
         di.CheckState = ProcSeeControls.CheckState.Checked;
   }
}
```

See Also

ICheckBox::CheckState, ICheckBoxEvents::OnClickCB

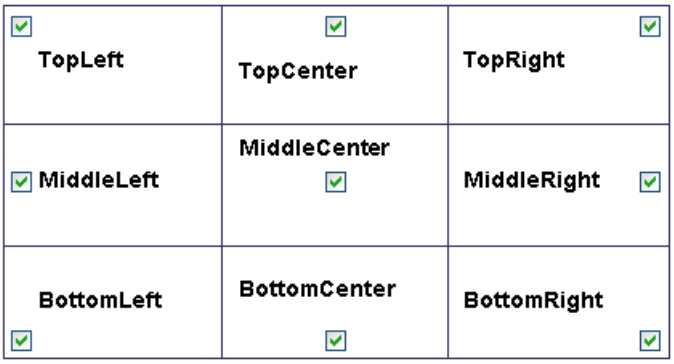
---

int ICheckBox::CheckAlign

This property controls the vertical and horizontal alignment of the check box mark on the CheckBox control. It can be aligned in nine different positions inside the control. To set this property use the constants defined in enum ContentAlignment.

The default value for this property is MiddleLeft.

Example



See Also

The enum ContentAlignment is described on page 114.

---

int ICheckBox::CheckState

This property sets or gets the state of the CheckBox. The constants in the enum CheckState defines the legal values this property can have. If the property ThreeState is true the check box can be in three states, checked, indeterminate and unchecked.

The default value is Unchecked.

See Also

The enum CheckState is described on page 114.

ICheckBox::ThreeState, ICheckBox::AutoCheck

---

int ICheckBox::Enabled

This property must be set to true if the CheckBox control should respond to user interaction, otherwise false. The default is true. To restrict the CheckBox control from being used, set this property value to false.

---

int ICheckBox::FlatStyle

This property controls the flat style of the CheckBox control. The constants in the enum FlatStyle define the legal values for this property.

The default value for this property is Standard.

Example

See Also

FlatStyle enum constants are described on page 115.

---

char* ICheckBox::Text

This property sets or gets the text displayed in the CheckBox control.

---

int ICheckBox::TextAlign

This property controls the alignment of the text in the CheckBox control. The text can be aligned in nine different positions inside the CheckBox. The legal values for this property are defined in the enum ContentAlignment.

The default value for this property is MiddleLeft.

Example

To view an example of ContentAlignment, refer to the examples in the description of IButton::TextAlign and ICheckBox::CheckAlign.

See Also

The enum constants ContentAlignment is described on page 114.

ICheckBox::CheckAlign, IButton::TextAlign.

---

int ICheckBox::ThreeState

This boolean property determines whether the CheckBox is in a three state or two state mode. Setting this property value to true will set the CheckBox to three state mode. The CheckBox can be set to an indeterminate state in addition to checked and unchecked states when this property is set to true.

The default value for this property is false.

Example

ICheckBoxEvents

The ICheckBoxEvents event interface has the following functions.

Function	Description
OnCheckStateChangedCB	Fired when the CheckBox changes state.
OnClickCB	Fired when the CheckBox is clicked.

F u n c t i o n s :

void ICheckBoxEvents::OnCheckStateChangedCB(int state)

This callback function is fired when the state of the CheckBox control changes state. The *state* argument has the value of one of the constants defined in the enum CheckState.

Parameters

state – Current state of the CheckBox control.

Example

The following example changes the foreground fill colour of the rectangle shape called R. The colour depends on the state of the CheckBox.

```
void OnCheckStateChangedCB(int state)
{
   switch ( state )
   {
       case ProcSeeControls.CheckState.Checked:
           R.foregroundFillColour = `red`;    break;
       case ProcSeeControls.CheckState.Unchecked:
           R.foregroundFillColour = `green`; break;
       case ProcSeeControls.CheckState.Indeterminate:
           R.foregroundFillColour = `blue`;   break;
   }
   R.updateShape();
}
```

void ICheckBoxEvents::OnClickCB)

This call-back function is fired when the CheckBox is being clicked.

See Also

ICheckBox::AutoCheck

ComboBox

The ComboBox control represents a standard Microsoft Windows combo box. A combo box is a graphical component displaying a text box combined with a drop down list box. In a combo box, the text can be typed into the text input field, or selected from the drop down list. The drop down list is displayed when the arrow next to the input field is clicked. To select an item in the list, click on the item and the selected item is automatically displayed in the text input field.

The ComboBox control supports two interfaces, IComboBox and IComboBoxEvents. The IComboBox interface, by default named *di*, is used to modify the contents of or change the visual appearance of the ComboBox control. The event interface, IComboBoxEvents, is by default named *dei*.

Note that the IComboBox interface does not provide properties or methods to change the ComboBox's font or colours. This functionality instead supported by the ComShape attributes theFont, foregroundColour and backgroundColour. The position and size of the ComboBox are controlled by the ComShape attributes x, y, width and height, respectively.

IComboBox

The IComboBox interface has the following properties and functions.

Property	Description
Count	Returns the number of items in the ComboBox control.
DropDownStyle	The style of the ComboBox control.
DropDownWidth	The width of the drop down portion of the ComboBox control.
DroppedDown	Whether the drop down portion of the ComboBox is visible or not.
Enabled	Whether the ComboBox reacts to user input or not.
MaxDropDownItems	Maximum number of items in the drop down portion of the ComboBox.
MaxLength	Maximum number of characters in the text input field of the ComboBox.
PersistContents	Whether the contents of the ComboBox should be persisted to file or not.
SelectedIndex	The index of the selected item in the ComboBox.
SelectedItem	The name of the selected item in the ComboBox.
SelectedText	The selected text in the input field of the ComboBox.
SelectionLength	Number of characters selected in the editable portion of the ComboBox.
SelectionStart	Where the selection starts in the input field of the ComboBox.
Sorted	Whether the drop down list of the ComboBox is sorted or not.
Text	The text in the input field of the ComboBox.

Function	Description
AddCollection	This function adds an IItemCollection object to the ComboBox control.
AddItem	This function adds a single item to the drop down list of the ComboBox.
BeginUpdate	Suspends redrawing of the control when adding new items to the ComboBox.
Clear	This function clears the drop down list in the ComboBox.
EndUpdate	Resumes redrawing of the ComboBox when items have been added.
FindItem	Finds an item in the drop down list matching the search string.
FindNextItem	Finds the next item in the drop down list matching the search string.
GetItem	Returns the item in the drop down list at the specified index.
InsertItem	This function inserts an item in the drop down list of the ComboBox.
Items	This function returns the items in the drop down list.
ModifyItem	This function modifies the item at the specified index.
NewCollection	This function creates a new collection whose items added to the collection can be added to the drop down list of the ComboBox.
RemoveAt	This function removes the items at the specified index.
RemoveItem	This function removes the item specified as argument.
Select	This function selects part of the text in the input field of the ComboBox.
SelectAll	This function selects all text in the input field of the ComboBox.
SetFocus	This function sets the keyboard focus to the ComboBox control.

P r o p e r t i e s :

IComboBox::Count

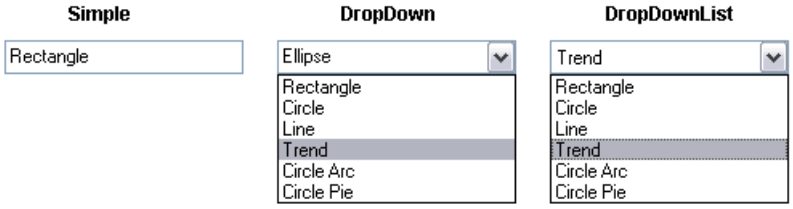
This property returns the number of items in the ComboBox control.

IComboBox::DropDownStyle

This property controls the style of the ComBox control. This property can be set to one of the constant values defined in the enum DropDownStyle.

The default value for this property is DropDown.

Example

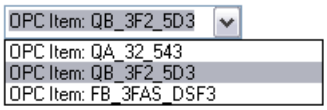


---

int IComboBox::DropDownWidth

This property controls the width of the drop down portion of the ComboBox control. It is not possible to set this property to a value smaller than the size of the ComboBox. The width of the ComboBox is controlled by the ComShape property width.

Example



See Also

IComboBox::MaxDropDownItems

---

int IComboBox::DroppedDown

This property controls whether the drop down portion of the ComboBox is displayed or not.

---

int IComboBox::Enabled

This boolean property value controls whether the ComboBox should respond to user interaction or not.

The default value of this property is true.

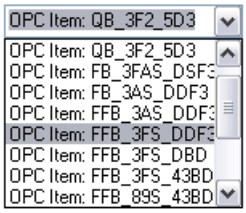
---

int IComboBox::MaxDropDownItems

This property controls the maximum number of items to display in the drop down portion of the ComboBox control. A scroll bar will appear to the right of the dropped down list if more items are added to the drop down list than specified with the MaxDropDownItems property

The default value of this property is 8.

Example



See Also

IComboBox::DropDownWidth

int IComboBox::MaxLength

This property controls the maximum number of characters allowed in the input field of the ComboBox control.

The default value of this property is the maximum short integer value.

int IComboBox::PersistContents

When this boolean property value is set to true the contents of the ComboBox control will be persisted to file. All items in the ComboBox will be saved.

int IComboBox::SelectedIndex

This function sets or gets the index of the currently selected item in the ComboBox control. When setting this property the item at the selected index will be displayed in the text input field of the ComboBox.

See Also

IComboBox::SelectedItem

char* IComboBox::SelectedItem

This function sets or gets the selected item in the ComboBox control. When setting this property to select an item in the ComboBox, the value must match one of the items in the drop down list. If an item is selected in the list, the item will be displayed in the text input field.

See Also

IComboBox::SelectedIndex

char* IComboBox::SelectedText

This property sets or gets the selected text in the input field of the ComboBox control. If the input field does not have a selected text, the empty string “” is returned.

See Also

IComboBox::SelectionLength, IComboBox::SelectionStart, IComboBox::Select, IComboBox::SelectAll

int IComboBox::SelectionLength

This property sets or gets the length of the selected text in the input field of the ComboBox control.

See Also

IComboBox::SelectedText, IComboBox::SelectionStart, IComboBox::Select, IComboBox::SelectAll

int IComboBox::SelectionStart

This integer property controls the start of the selected text in the input field of the ComboBox control. This property is zero indexed. The first character in the text input field has the index 0.

See Also

IComboBox::SelectedText, IComboBox::SelectionLength, IComboBox::Select, IComboBox::SelectAll

---

int IComboBox::Sorted

This property determines whether the items in the ComboBox are sorted in ascending order or not.

The default value of this property is false.

---

char* IComboBox::Text

This property gets or sets the text in the input field of the ComboBox control.

F u n c t i o n s :

---

int IComboBox::AddCollection(ProcSeeControls.IItemCollection* collection)

This function adds a collection of new items to the ComboBox control. This is the preferred method to add a large number of items when performance is an issue.

Parameters

collection – Pointer to an IItemCollection interface to add to the ComboBox control.

Return Value

This function returns true if the contents of the IItemCollection interface was successfully added to the ComboBox, otherwise false.

Example

The following example demonstrates the use of the IItemCollection object when adding items to the ComboBox. The function creates a new IItemCollection object, named ic. The for loop iterates over the items in the ::List object and add these objects to the collection object using the member function AddItem. Finally, the items in the collection object are added to the ComboBox by calling the function AddCollection.

```
void InitComboBox( ::List* list )
{
  ProcSeeControls.IItemCollection* ic;
  ic = (ProcSeeControls.IItemCollection)cb.di.NewCollection();

  int maxNum = list->length();
  for ( int j = 0; j < maxNum; j++ )
  {
     char* item = (char *)list->get( j );
     ic->AddItem( item );
  }

   cb.di.AddCollection( ic );
}
```

See Also

See page 108 for a description of the interface IItemCollection.

IComboBox::NewCollection

---

int IComboBox::AddItem(char* theItem)

This function appends a single item to the ComboBox control.

Parameters

theItem – The item to add to the ComboBox control.

Return Value

This function returns true if the item was successfully appended to the ComboBox control, otherwise false.

See Also

IComboBox::InsertItem, IComboBox::BeginUpdate, IComboBox::EndUpdate, IComboBox::AddCollection

---

void IComboBox::BeginUpdate()

This function is used to boost performance when adding a large number of items to the ComboBox control. This function should be called before calling the member functions AddItem or InsertItem. When BeginUpdate has been called the control is suspended from being redrawn until the function EndUpdate is called.

Preventing to call EndUpdate after calling BeginUpdate will freeze the ComboBox control.

Example

The following example demonstrates the use of the functions BeginUpdate and EndUpdate when adding items to the ComboBox.

```
…
CB.di.BeginUpdate();  // Suspend redrawing while adding items
CB.di.AddItem( "…" );
CB.di.AddItem( "…" );
…
CB.di.EndUpdate();
```

See Also

IComboBox::EndUpdate, IComboBox::AddItem, IComboBox::InsertItem

---

void IComboBox::Clear()

This function removes all items in the ComboBox control.

See Also

IComboBox::Remove, IComboBox::RemoveAt

---

void IComboBox::EndUpdate()

This function must be called to resume painting of the ComboBox control after painting is suspended by the BeginUpdate method. Using the functions BeginUpdate and EndUpdate to suspend redrawing of the control will boost performance when adding new items.

See Also

IComboBox::BeginUpdate, IComboBox::AddItem, IComboBox::InsertItem

---

int IComboBox::FindItem(char* searchValue, int exact)

This function finds and returns the index of the first item in the list matching the compare string. If the second argument exact is true the compare string must have an exact match to return the index of an item in the ComboBox. If the argument is false the search string must match the first n characters of the item in the ComboBox, where n is the length of the search string. When exact is false the compare algorithm is case insensitive.

The function does not support a regular expression search string.

Parameters

searchValue – The item to search for in the ComBox control.

exact – True if searching for an exact match, otherwise false.

Return Value

This function returns the index of the first item matching the search string in the ComboBox control. The value negative one (-1) is returned if this function failed to find the search string in the ComboBox.

See Also

IComboBox::FindNextItem

---

int IComboBox::FindNextItem(char* searchValue, int index, int exact)

This function finds and returns the index of the next item in the ComboBox matching the search string. The search will start from the specified index in the ComboBox. Passing the value -1 in the argument index will start the search from the first item in the list, that is, identical to the function FindItem. For more information about this function, see FindItem.

Parameters

searchValue – The item to search for in the ComBox control.

index – The index in the ComboBox where to begin the search.

exact – True if searching for an exact match, otherwise false.

Return Value

This function returns the index of the next item matching the search string in the ComboBox control. The value negative one (-1) is returned if this function failed to find the search string in the ComboBox.

See Also

IComboBox::Find

---

char* IComboBox::GetItem(int index)

This function returns the item at the specified index in the ComboBox control.

Parameters

index – The zero based index of the item to return in the ComboBox.

Return Value

This function returns the item at the specified index, otherwise a null pointer.

---

int IComboBox::InsertItem(int index, char* theValue)

This function inserts a single item to the ComboBox control at the specified index.

Parameters

index – The zero based index in the ComboBox to insert the item.

theValue- The item to insert in the ComboBox control.

Return Value

Returns true if the item was successfully inserted in the ComboBox, otherwise false.

See Also

IComboBox::AddItem, IComboBox::BeginUpdate, IComboBox::EndUpdate, IComboBox::AddCollection

---

::IEnumAny* IComboBox::Items()

This function returns a pointer to an interface which can be used to iterate through the items in the ComboBox. Enumerating over all items in the ComboBox is faster using this method compared to using the GetItem member function.

Return Value

This function returns a pointer to an IEnumAny interface containing the items in the ComboBox.

Notice

This function actually returns an IEnumAny interface pointer, but due to some COM implementation issues, which are outside the scope of this manual, all interfaces are returned as IUnknown pointers. Therefore, an explicit cast operator is required as shown below. The following code will compile OK but will fail at run-time because the interface is not of the expected IEnumAny type.

```
::IEnumAny* ea = L.di.Items();              // THIS LINE FAILS
::IEnumAny* ea = (::IEnumAny*)L.di.Items(); // OK
```

Example

The following source code example uses the Items member function to get a pointer to the items enumerator interface. All the items in the ComboBox are enumerated using the IEnumAny member function next. This function returns the item name. In the while loop a fictitious function AppendTo is called passing the item name as argument.

```
void GetItems()
{
   ::IEnumAny* items = (::IEnumAny*)L.di.Items();

   char* name;
   while ( ( items->next( &name ) ) != 0 )
```

```
   {
      AppendTo( name );
   }
}
```

See Also

IComboBox::GetItem

---

int IComboBox::ModifyItem(int index, char* theValue)

This function modifies the item in the ComboBox at the specified index.

Parameters

index – The zero based index of the item to modify.

theValue – The new value of the item to modify.

Return Value

This function returns true if the item was successfully modified, otherwise false.

See Also

IComboBox::AddItem, IComboBox::InsertItem

::ProcSeeControls.IItemCollection* IComboBox::NewCollection()

This function creates a new and returns a pointer to a created IItemCollection object. This IItemCollection object is normally used to improve performance of the ComboBox control. Items can be added, modified and deleted from the IItemCollection object. Use the function AddCollection to add the items in the collection object to the ComboBox.

Return Value

This function returns a pointer to a new IItemCollection object.

Notice

This function actually returns an IItemCollection interface pointer, but due to some COM implementation issues, which are outside the scope of this manual, all interfaces are returned as IUnknown pointers. Therefore, an explicit cast operator is required as shown below.

```
::ProcSeeControls.IItemCollection* ic;
ic = L.di.Items();                                  // FAILS
ic = (ProcSeeControls.IItemCollection*)L.di.Items(); // OK
```

See Also

IListBox::AddCollection

---

int IComboBox::RemoveAt(int index)

This function removes the item at the specified index from the ComboBox control.

Parameters

index – The zero based index of the item to remove.

Return Value

This function returns true if the item at the specified index was successfully removed, otherwise false.

See Also

IComboBox::RemoveItem, IComboBox::Clear

---

int IComboBox::RemoveItem(char* searchValue, int exact, int numToRemove)

This function removes one or several items from the ComboBox matching the search string passed as argument.

If the second argument exact is true the compare string must have an exact match to remove an item from the ComboBox. If the argument exact is false the search string must match the first n characters of the item in the ComboBox, where n is the length of the search string. When exact is false the compare algorithm is case insensitive.

The last argument to this function determines how many items in the ComboBox matching the search string to remove from the control. The value zero means, remove all items matching the search string. A value greater than zero means, remove this many number of items from the ComboBox matching the search string. To remove one item, call this function with the last argument set to the value 1.

Parameters

searchValue – The items to search for in the ComboBox.

exact – True if searching for an exact match, otherwise false.

numToRemove – Number of items to remove from the ComboBox. The value zero means, remove all matching items.

Return Value

This function returns true if items was successfully removed from the ComboBox, otherwise false.

See Also

IComboBox::Remove, IComboBox::Clear

---

void IComboBox::Select(int start, int length)

This function selects a range of text in the text input field of the ComboBox. The arguments to this function specify the start position and the length of the text to select.

Parameters

start – The start index of the text to select. The first character has the index 0.

length – Number of characters to select.

See Also

IComboBox::SelectionLength, IComboBox::SelectionStart, IComboBox::SelectedText, IComboBox::SelectAll

void IComboBox::SelectAll()

This function selects the entire text in the input field of the ComboBox.

See Also

IComboBox::SelectionLength, IComboBox::SelectionStart, IComboBox::SelectedText, IComboBox::Select

int IComboBox::SetFocus()

This function sets the keyboard focus to the ComboBox control.

Return Value

This function returns true if the keyboard focus was successfully set to the ComboBox control, otherwise false.

IComboBoxEvents

The IComboBoxEvents has the following functions.

Function	Description
OnSelectChangedCB	This callback fires when the selected item in the drop down list changes.
OnKeyPressCB	This callback fires when a character is typed in the ComboBox.
OnReturnCB	This callback fires when the return key is pressed in the ComboBox.
OnTabCB	This callback fires when the tab key is pressed in the ComboBox.
OnTextChangedCB	This callback fires when the text in the text input field changes.

F u n c t i o n s :

void IComboBoxEvents::OnSelectChangedCB

This callback function fires when the selected item in the drop down list changes.

Example

The following example demonstrates how to implement a selection changed call-back function. This function gets the selected item with the property SelectedItem and calls the user defined pTALK function ShapeTypeChanged.

```
void OnSelectChangedCB()
{
   char* item = di.SelectedItem;
   ShapeTypeChanged( item );
}
```

void IComboBoxEvents::OnKeyPressCB(unsigned short int c)

This callback function fires when a character is pressed in the ComboBox.

Parameters

c – The character typed in the ComboBox.

See Also

IComboBoxEvents::OnReturnCB, IComboBoxEvents::OnTabCB,
IComboBoxEvents::OnTextChangedCB

---

void IComboBoxEvents::OnReturnCB()

This callback function fires when the return key is pressed in the ComboBox.

See Also

IComboBoxEvents::OnKeyPressCB, IComboBoxEvents::OnTabCB,
IComboBoxEvents::OnTextChangedCB

---

void IComboBoxEvents::OnTabCB()

This callback function fires when the tab key is pressed in the ComboBox.

See Also

IComboBoxEvents::OnKeyPressCB, IComboBoxEvents::OnReturnCB,
IComboBoxEvents::OnTextChangedCB

---

void IComboBoxEvents::OnTextChangedCB(char* string)

This callback function fires when the text in the text input field changes.

See Also

IComboBoxEvents::OnKeyPressCB, IComboBoxEvents::OnTabCB,
IComboBoxEvents::OnReturnCB

Label

The Label control represents a standard Microsoft Windows label. A label is a graphical component displaying text. The text can be aligned at different positions, for instance right or centre aligned text.

Notice that the text in the Label control cannot be selected.

The Label control supports one interface ILabel. The ILabel interface, by default named *di*, is used to change the visual appearance of the Label control.

Note that the ILabel interface does not provide properties and methods to change the Label's font or colours. This functionality is instead supported by the ComShape attributes theFont, foregroundColour and backgroundColour. The position and size of the Label are controlled by the ComShape attributes x, y, width and height, respectively.

Double clicking on a Label control while the picture is in edit mode will put the control into user input mode. In this mode, the Label control will display a text input field where the user can type the label text. Clicking outside the control will put the Label back to normal mode and the text entered will be displayed in the control.

ILabel

The ILabel interface has the following properties.

Property	Description
BorderStyle	This property controls the border style.
Enabled	Whether the label is enabled or not.

PersistContents	Controls whether the label text is saved or not.
Text	The text displayed in the label.
TextAlign	Controls the alignment of the text in the label.

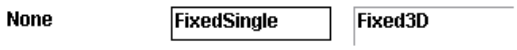
P r o p e r t i e s :

int ILabel::BorderStyle

This property controls the border style for the Label control. The border styles are defined in the enum constant BorderStyle.

The default value for this property is None.

Example



int ILabel::Enabled

When this property is false the text on the Label control is displayed using a grey colour.

int ILabel::PersistContents

This property controls whether the text on the Label control is persisted to file or not.

This property has the default value true.

See Also

ILabel::Text

char* ILabel::Text

This property gets or sets the text in the Label control.

int ILabel::TextAlign

This property controls the alignment of the text in the Label. There are nine different positions in the Label where the text can be aligned. These positions are defined in the enum constant ContentAlignment.

The default value for this property is MiddleLeft.

Example

See IButton::TextAlign on page 32 for an example of alignment of text in a control.

See Also

ContentAlignment enum constants are described on page 114.

ListBox

The ListBox control represents a standard Microsoft Windows list box. A list box is a graphical component displaying a list of items, one per line, which the user can select by clicking. If the number of items put into the list has more items than can be displayed in the list box view, a scroll bar will appear. A list box provides functionality like multi column view, multi selection mode, search commands etc.

The ListBox control supports two interfaces IListBox and IListBoxEvents. The IListBox interface, by default named *di*, is used to modify the contents of or change the visual appearance of the ListBox control. The event interface, IListBoxEvents, is by default named *dei*.

Note that the IListBox interface does not provide properties or methods to change the ListBox's font or colours. This functionality is instead supported by the ComShape attributes theFont, foregroundColour and backgroundColour. The position and size of the ListBox are controlled by the ComShape attributes x, y, width and height, respectively.

IListBox

The IListBox interface has the following properties and functions.

Property	Description
ColumnWidth	If multi column is enabled, this property controls the width of a column.
Count	Number of items in the ListBox control.
Enabled	Whether the ListBox control is enabled or disabled.
HorizontalExtent	Controls the extent of the ListBox control and enables horizontal scroll bar.
PersistContents	This property controls whether the ListBox items are persisted or not.
ScrollAlwaysVisible	This property controls whether the scroll bars are always visible or not.
SelectedCount	Number of items selected in the ListBox control.
SelectionMode	Type of selection mode, single select, multi select, etc.
Sorted	Controls whether the items in the ListBox control are sorted or not.
TopIndex	Index of the first visible item in the ListBox.
UseTabStops	Indicatets whether the ListBox can recognize and expand tab characters.

Function	Description
AddCollection	This function adds a new collection to the ListBox.
AddItem	This function adds an item to the ListBox.
BeginUpdate	This function suspends redrawing of the ListBox until EndUpdate is called.
Clear	This function removes all the items from the ListBox.
EndUpdate	This function resumes drawing of the ListBox after BeginUpdate is called.
FindItem	Finds an item in the ListBox matching the input string.
FindNextItem	Finds the next item in the ListBox matching the input string.
GetItem	Returns the item at the specified index.
InsertItem	This function inserts an item to the ListBox.
IsSelected	This function checks if the item at the specified index is selected.
Items	Returns the items in the list box.
ModifyItem	Modifies the item at the specified index.
NewCollection	Creates a new collection whose items added to the collection can be added to the ListBox.
RemoveAt	This function removes the item at the specified index.
RemoveItem	This function removes the item matching the name specified as argument.
Select	This function selects the item at the specified index.
SelectedIndex	Returns the index of the item in the ListBox specified by the zero based index in the selected items list.
SelectedIndices	This function returns a collection of selected indices.
SelectedItem	Returns the item at the index in the ListBox specified by the zero based index

	in the selected items list.
SelectedItems	This function returns a collection of selected items.
Unselect	This function unselects the item at the specified index.
UnselectAll	This function unselects all items in the ListBox.

P r o p e r t i e s :

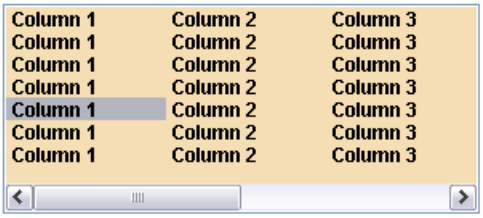
int IListBox::ColumnWidth

This property controls the column width and multi column mode of the ListBox control. To enable multi column mode set this value to a value greater than 0. Setting this value to 0 will disable multi column mode.

The default value of this property is 0.

Example

The following figure illustrates the use of a multi column ListBox control where the property ColumnWidth is set to the value 100.



int IListBox::Count

This read only property returns the number of items in the ListBox.

int IListBox::Enabled

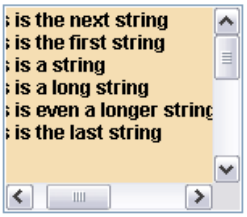
This boolean property value controls whether the ListBox should respond to user interaction or not. The default value of this property is true.

int IListBox::HorizontalExtent

This property controls the width by which the horizontal scroll bar of the ListBox can scroll. The scroll bar will automatically be visible when this property is set to a value greater than the ListBox width.

Example

The following example illustrates the use of the property HorizontalExtent where the width of the ListBox is 150 and the property HorizontalExtent is set to 300.



See Also

IListBox::ScrollAlwaysVisible

---

int IListBox::PersistContents

When this boolen property value is set to true the contents of the ListBox control will be persisted to file. All items in the list box will be saved.

---

int IListBox:: ScrollAlwaysVisible

This boolean property value controls whether the scroll bars should always be visible or not.

See Also

IListBox::HorizontalExtent

---

int IListBox::SelectedCount

This read only property returns the number of selected items in the ListBox control.

See Also

IListBox::SelectionMode, IListBox::Select, IListBox::Unselect, IListBox::SelectedIndex, IListBox::SelectedIndices, IListBox::SelectedItem, IListBox::SelectedItems, IListBox::IsSelected

---

int IListBox::SelectionMode

This property determines the method in which items are selected in the ListBox control. This property can be set to one of the constant values defined in the enum SelectionMode.

The default value of this property is One.

See Also

IListBox::SelectedCount, IListBox::Select, IListBox::Unselect, IListBox::SelectedIndex, IListBox::SelectedIndices, IListBox::SelectedItem, IListBox::SelectedItems, IListBox::IsSelected

---

int IListBox::Sorted

This property controls whether the items in the ListBox are sorted in ascending order or not.

The default value of this property is false.

---

int IListBox::TopIndex

This property determines the first visible item in the ListBox control.

---

int IListBox::UseTabStops

This property indicates whether the ListBox control can recognize and expand tab characters.

F u n c t i o n s :

int IListBox::AddCollection(::ProcSeeControls.IItemCollection* collection)

This function adds a collection of new items to the ListBox control. This is the preferred method to add a large number of items when performance is an issue.

Parameters

collection – Pointer to an IItemCollection interface to add to the ListBox control.

Return Value

This function returns true if the contents of the IItemCollection interface was successfully added to the ListBox, otherwise false.

Example

The source code snippet below uses the IItemCollection interface to add items to the ListBox control. The IListBox function NewCollection returns a pointer to a new IItemCollection object. The for loop iterates through the items in the ::List object and adds the string items to the collection object using the member function AddItem. Finally, when all the items are added to the collection the items in the IItemCollection object are added to the ListBox control using the IListBox member function AddCollection.

```
void AddNewItems( ::List* theList )
{
  ProcSeeControls.IItemCollection* items;
  Items = (ProcSeeControls.IItemCollection*)L.di.NewCollection();

  int maxN = theList->length();
  for ( int j = 0; j < maxN; j++ )
    items->AddItem( theList->get( j ) );

  L.di.AddCollection( items );
}
```

See Also

See page 108 for a description of the interface IItemCollection.

IListBox::NewCollection, IListBox::AddItem, IListBox::BeginUpdate, IListBox::EndUpdate

int IListBox::AddItem(char* theValue)

This function appends a single item to the ListBox control. If the property Sorted is true the item will be put in the list at the correct position in the alphabetic sorted list, otherwise last.

Parameters

theValue – The string value to append to the ListBox control.

Return Value

Returns true if the item was successfully appended to the ListBox control, otherwise false.

See Also

IListBox::InsertItem, IListBox::NewCollection, IListBox::AddCollection, IListBox::BeginUpdate, IListBox::EndUpdate

void IListBox::BeginUpdate()

This function is used to boost performance when adding a large number of items to the ListBox control. This function should be called before calling the member function AddItem or InsertItem. When BeginUpdate has been called, the control will suspend itself from being redrawn until the function EndUpdate is called.

Preventing to call EndUpdate after calling BeginUpdate will freeze the ListBox control.

Example

The source code below uses the function BeginUpdate and EndUpdate to improve the performance when a large number of items are added to a ListBox. The for loop iterates through the items in the ::List object and adds the string items to the ListBox object using the member function AddItem. Since redrawing has been suspended when new items are added, the time to finish the for loop will be dramatically reduced. Finally, when all the items are added the member function EndUpdate is called to redraw the items.

```
void AddNewItems( ::List* theList )
{
   L.di.BeginUpdate();

   int maxN = theList->length();
   for ( int j = 0; j < maxN; j++ )
      L.di.AddItem( theList->get( j ) );

    L.di.EndUpdate();
}
```

See Also

IListBox::EndUpdate, IListBox::NewCollection, IListBox::AddCollection, IListBox::AddItem, IListBox::InsertItem

---

void IListBox::Clear()

This function removes all the items in the ListBox.

See Also

IListBox::Remove, IListBox::RemoveItem

---

void IListBox::EndUpdate()

This function must be called to resume painting of the ListBox control after painting has been suspended by the BeginUpdate method. Using BeginUpdate and EndUpdate to suspend redrawing when items are added will boost performance.

See Also

IListBox::BeginUpdate, IListBox::AddCollection, IListBox::AddItem, IListBox::InsertItem

---

int IListBox::FindItem(char* searchValue, int exact)

This function finds and returns the index of the first item in the list matching the compare string. If the second argument exact is true the compare string must have an exact match to return the index of an item in the ListBox. If the argument exact is false the search string must match the first n

characters of the item in the ListBox, where n is the length of the search string. When exact is false the compare algorithm is case insensitive.

This function does not support a regular expression search string.

Parameters

searchValue – The item to search for in the ListBox.

exact – True if searching for an exact match, otherwise false.

Return Value

This function returns the index of the first item matching the search string in the ListBox control. The value negative one, -1, is returned if this function fails to find the search string in the ListBox

See Also

IListBox::FindNextItem

---

int IListBox::FindNextItem(char* searchValue, int index, int exact)

This function finds and returns the index of the next item in the list matching the search string. The search will start from the specified index in the ListBox. Passing the value -1 in the search string will start the search from the first item in the list, that is, identical to the function FindItem. For more information about this function, see FindItem.

Parameters

searchValue – The item to search for in the ListBox.

index – The index in the ListBox where to begin the search.

exact – True if searching for an exact match, otherwise false.

Return Value

This function returns the index of the next item matching the search string in the ListBox control. The value negative one, -1, is returned if this function fails to find the search string in the ListBox

Example

This following source code example adds all items in the ListBox matching the search string passed as argument to the ::List object.

```
::List* GetMatching( char* searchString )
{
  ::List* l = newList();

  int ix = -1;
  while ( ( ix = L.di.FindNextItem( searchString, ix, 0 ) ) > -1 )
  {
       l->add( ix );
  }

  return l;
}
```

See Also

IListBox::FindNext

char* IListBox::GetItem(int index)

This function returns the item at the specified index in the ListBox control.

Parameters

index – The zero indexed item to return in the ListBox.

Return Value

This function returns the item at the specified index, otherwise a null pointer.

int IListBox::InsertItem(int index, char* theValue)

This function inserts a single item to the ListBox control at the specified index.

Notice that if the property Sorted is true, the item will be put in the list at the correct position in the alphabetic sorted list, otherwise at the specified index.

Parameters

index – The zero based index in the ListBox to insert the item.

theValue – The string value to insert in the ListBox control.

Return Value

Returns true if the item was successfully inserted in the ListBox control, otherwise false.

See Also

IListBox::AddItem, IListBox::NewCollection, IListBox::AddCollection, IListBox::BeginUpdate, IListBox::EndUpdate

int IListBox::IsSelected(int index)

This function checks if the item at the specified index in the ListBox is selected or not.

Parameters

Index – The zero based index in the ListBox to check.

Return Value

This function returns true if the item at the specified index is selected, otherwise false.

See Also

IListBox::Select, IListBox::Unselect, IListBox::UnselectAll, IListBox::SelectedIndex, IListBox::SelectedIndices, IListBox::SelectedItem, IListBox::SelectedItems, IListBox::SelectionMode

::IEnumAny* IListBox::Items()

This function returns a pointer to an interface which can be used to iterate through the items in the ListBox. Enumerating over all items in the ListBox is faster using this method compared to using the GetItem member function.

Return Value

This function returns a pointer to an IEnumAny interface containing the items in the ListBox.

Notice

This function actually returns an IEnumAny interface pointer, but due to some COM implementation issues, which are outside the scope of this manual, all interfaces are returned as IUnknown pointers. Therefore, an explicit cast operator is required as shown below.

```
::IEnumAny* ea = L.di.Items();              // THIS LINE FAILS
::IEnumAny* ea = (::IEnumAny*)L.di.Items(); // OK
```

Example

The following source code example uses the Items member function to get a pointer to the items enumerator interface. All the items in the ListBox are enumerated using the IEnumAny member function next. This function returns the item name. In the while loop is a fictitious ConnectTo pTALK method called passing the item name as argument.

```
void GetItems()
{
   ::IEnumAny* items = (::IEnumAny *)L.di.Items();

   char* name;
   while ( ( items->next( &name ) ) != 0 )
   {
      ConnectTo( name );
   }
}
```

See Also

IListBox::GetItem

---

int IListBox::ModifyItem(int index, char* theValue)

This function modifies the item in the ListBox at the specified index.

Parameters

index – The zero based index of the item to modify.

theValue – The new value of the item to modify.

Return Value

This function returns true if the item was successfully modified, otherwise false.

See Also

IListBox::AddItem, IListBox::InsertItem

::ProcSeeControls.IItemCollection* IListBox::NewCollection()

This function creates a new and returns a pointer to the created IItemCollection object. This IItemCollection object is normally used to improve performance of the ListBox control. Items can be added, modified and deleted from the IItemCollection object. Use the function AddCollection to add the items in the collection object to the ListBox.

Return Value

This function returns a pointer to new IItemCollection object.

Notice

This function actually returns an IItemCollection interface pointer, but due to some COM implementation issues, which are outside the scope of this manual, all interfaces are returned as IUnknown pointers. Therefore, an explicit cast operator is requires as shown below.

```
::ProcSeeControls.IItemCollection* ic;
ic = L.di.Items();                                  // FAILS
ic = (ProcSeeControls.IItemCollection*)L.di.Items(); // OK
```

See Also

IListBox::AddCollection

---

int IListBox::RemoveAt(int index)

This function removes the item at the specified index from the ListBox control.

Parameters

index – The zero based index of the item to remove.

Return Value

This function returns true if the item at the specified index was successfully removed, otherwise false.

See Also

IListBox::RemoveItem, IListBox::Clear

---

int IListBox::RemoveItem(char* searchValue, int exact, int numToRemove)

This function removes one or several items from the ListBox matching the search string passed as argument.

If the second argument exact is true the compare string must have an exact match to remove an item from the ListBox. If the argument exact is false the search string must match the first n characters of the item in the ListBox, where n is the length of the search string. When exact is false, the compare algorithm is case insensitive.

The last argument to this function determines how many items in the ListBox matching the search string to remove from the control. The value zero means, remove all items matching the search string. A value greater than zero means, remove this many number of items from the ListBox matching the search string. To remove one item, call this function with the last argument set to the value 1.

Parameters

searchValue – The items to search for in the ListBox.

exact – True if searching for an exact match, otherwise false.

numToRemove – Number of items to remove from the ListBox. The value zero means, remove all matching items.

Return Value

This function returns true if items was successfully removed from the ListBox, otherwise false.

See Also

IListBox::Remove, IListBox::Clear

---

int IListBox::Select(int index)

This function selects the item in the ListBox at the specified index.

Parameters

index – The zero based index of the item to select.

Return Value

Returns true if the item in the ListBox was successfully selected, otherwise false.

See Also

IListBox::IsSelected, IListBox::Unselect, IListBox::UnselectAll, IListBox::SelectedIndex, IListBox::SelectedIndices, IListBox::SelectedItem, IListBox::SelectedItems, IListBox::SelectionMode

---

int IListBox::SelectedIndex(int index)

This function returns the zero based index in the ListBox of the nth selected item. The argument index is the nth index of the selected items in the ListBox.

Parameters

index – The zero based index of the selected item in the ListBox.

Return Value

This function returns the zero based index in the ListBox of the nth selected item. If the argument index is out of range or no item is selected a negative one, -1, is returned.

Example

The following example demonstrates how to return a ::List object with the names of all selected items in the ListBox. The property SelectedCount and the function SelectedIndex are used to get the number of selected items and the index of the selected item in the ListBox, respectively.

```
::List* GetSelected()
{
  int numSelected = L.di.SelectedCount;
  ::List* theList = newList();

  for ( int i = 0; i < numSelected; i++ )
  {
       int ix = L.di.SelectedIndex( i );
       theList->add( L.di.GetItem( ix ) );
  }

  return theList;
}
```

See Also

IListBox::IsSelected, IListBox::Unselect, IListBox::UnselectAll, IListBox::Select, IListBox::SelectedIndices, IListBox::SelectedItem, IListBox::SelectedItems, IListBox::SelectionMode

---

::IEnumAny* IListBox::SelectedIndices()

This function returns a pointer to an IEnumAny interface, which can be used to iterate through the selected indices in the ListBox. Note that the argument to use in the method IEnumAny::next is integer.

Return Value

This function returns a pointer to an IEnumAny interface containing the selected indices in the ListBox.

Notice

This function actually returns an IEnumAny interface pointer, but due to some COM implementation issues, which are outside the scope of this manual, all interfaces are returned as IUnknown pointers. Therefore, an explicit cast operator is required as shown below.

```
::IEnumAny* ea = L.di.Items();              // THIS LINE FAILS
::IEnumAny* ea = (::IEnumAny*)L.di.Items(); // OK
```

Example

The following example demonstrates the use of the function SelectedIndices to enumerate over all selected indices in the ListBox.

```
void GetSelectedIndices()
{
   ::IEnumAny* selind = (::IEnumAny*)L.di.SelectedIndices();

   int ix;
   while ( ( selind->next( &ix ) ) != 0 )
   {
       SelectedObject( ix );
   }
}
```

See Also

IListBox::IsSelected, IListBox::Unselect, IListBox::UnselectAll, IListBox::SelectedIndex, IListBox::Select, IListBox::SelectedItem, IListBox::SelectedItems, IListBox::SelectionMode

---

char* IListBox::SelectedItem(int index)

This function returns the item in the ListBox of the nth selected item. The argument index is the nth index of the selected items in the ListBox.

Parameters

index – The zero based index of the selected item in the ListBox.

Return Value

This function returns the item in the ListBox of the nth selected item. If the argument index is out of range or no item is selected a null pointer is returned.

Example

The following example demonstrates how to return a ::List object with the names of all selected items in the ListBox. The property SelectedCount and function SelectedItem are used to get the number of selected items and the selected item in the ListBox, respectively.

```
::List* GetSelected()
{
  int numSelected = L.di.SelectedCount;
  ::List* theList = newList();

  for ( int i = 0; i < numSelected; i++ )
       theList->add( L.di.SelectedItem( ix ) );

  return theList;
}
```

See Also

IListBox::IsSelected, IListBox::Unselect, IListBox::UnselectAll, IListBox::SelectedIndex, IListBox::SelectedIndices, IListBox::Select, IListBox::SelectedItems, IListBox::SelectionMode

---

::IEnumAny* IListBox::SelectedItems()

This function returns a pointer to an IEnumAny interface, which can be used to iterate through the selected items in the ListBox. Note that the argument to the method IEnumAny::next is string, char*.

Return Value

This function returns a pointer to an IEnumAny interface containing the selected items in the ListBox.

Notice

This function actually returns an IEnumAny interface pointer, but due to some COM implementation issues, which are outside the scope of this manual, all interfaces are returned as IUnknown pointers. Therefore, an explicit cast operator is required as shown below.

```
::IEnumAny* ea = L.di.Items();              // THIS LINE FAILS
::IEnumAny* ea = (::IEnumAny*)L.di.Items(); // OK
```

Example

The following example demonstrates the use of the function SelectedIndices to enumerate over all selected items in the ListBox.

```
void GetSelectedIndices()
{
   ::IEnumAny* selitems = (::IEnumAny *)L.di.SelectedItems();

   char* item;
   while ( ( selitems->next( &item) ) != 0 )
   {
       SelectedItem( item );
   }
}
```

See Also

IListBox::IsSelected, IListBox::Unselect, IListBox::UnselectAll, IListBox::SelectedIndex, IListBox::SelectedIndices, IListBox::SelectedItem, IListBox::Select, IListBox::SelectionMode

---

int IListBox::Unselect(int index)

This function unselects the item in the ListBox at the specified index.

Parameters

index – The zero based index of the item to unselect.

Return Value

Returns true if the item in the ListBox was successfully unselected, otherwise false.

See Also

IListBox::IsSelected, IListBox::Select, IListBox::UnselectAll, IListBox::SelectedIndex, IListBox::SelectedIndices, IListBox::SelectedItem, IListBox::SelectedItems, IListBox::SelectionMode

---

void IListBox::UnselectAll()

This function unselects all selected items in the ListBox.

See Also

IListBox::IsSelected, IListBox::Unselect, IListBox::Select, IListBox::SelectedIndex, IListBox::SelectedIndices, IListBox::SelectedItem, IListBox::SelectedItems, IListBox::SelectionMode

---

IListBoxEvents

The IListBoxEvents event interface has the following functions.

Function	Description
OnClickCB	This function is fired when the mouse is clicked on an item in the ListBox.
OnDoubleClickCB	This function is fired when the mouse is double clicked on an item in the ListBox.
OnSelectChangedCB	This function is fired when a new item is selected in the ListBox.

F u n c t i o n s :

---

void IListBoxEvents::OnClickCB(int index)

This call-back function is fired when clicking on an item in the ListBox control.

Parameters

index – The zero based index of the item clicked in the ListBox control.

See Also

IListBoxEvents::OnDoubleClickCB, IListBoxEvents::OnSelectChangedCB

void IListBoxEvents::OnDoubleClickCB(int index)

This call-back function is fired when double clicking on an item in the ListBox control.

Parameters

index – The zero based index of the item double clicked in the ListBox control.

See Also

IListBoxEvents::OnClickCB, IListBoxEvents::OnSelectChangedCB

void IListBoxEvents::OnSelectChangedCB()

This call-back function is fired when current selection of items in the ListBox has changed.

Example

The following example demonstrates how to implement an OnSelectChangedCB when the ListBox control's property SelectionMode is set to either MultiSimple or MultiExtended. When the selection change, all selected items are enumerated using the IEnumAny interface and the ListBox member function SelectedItems. The fictitious functions ClearSelected and AddSelected in the source code example are user implemented pTALK functions.

```
void OnSelectChangedCB()
{
   ::IEnumAny* selix = (::IEnumAny *)L.di.SelectedItems();

   char* item;
   ClearSelected();
   while ( selix->next( &item ) != 0 )
      AddSelected( item );
}
```

See Also

IListBoxEvents::OnClickCB, IListBoxEvents::OnDoubleClickCB

MultiLineTextBox

The MultiLineTextBox control is a standard Microsoft Windows multi line text field. Its content can be edited by the end-user as well as the designer.

The MultiLineTextBox control supports two interfaces, IMultiLineTextBox and IMultiLineTextBoxEvents. The IMultiLineTextBox interface, by default named *di*, is used to modify the contents of or change the visual appearance of the MultiLineTextBox control. The event interface, IMultiLineTextBoxEvents, is by default named *dei*.

Note that the IMultiLineTextBox interface does not provide properties or methods to change the MultiLineTextBox's font or colours. This functionality is instead supported by the ComShape attributes theFont, foregroundColour and backgroundColour. The position and size of the MultiLineTextBox are controlled by the ComShape attributes x, y, width and height, respectively.

IMultiLineTextBox

The IMultiLineTextBox interface has the following properties and functions.

Property	Description

BorderStyle	This property controls the border style.
CharacterCasing	Controls whether the character casing is modified or not when typing.
Enabled	Controls whether the MultiLineTextBox is enabled or not.
MaxLength	Maximum length of the text in the MultiLineTextBox control.
PersistContents	This property controls whether the text is persisted or not.
ReadOnly	Controls whether the MultiLineTextBox is read only or not.
ScrollBars	Controls which scroll bars to appear in the MultiLineTextBox control.
SelectedText	The selected text in the MultiLineTextBox.
SelectionLength	The length of the selected text in the MultiLineTextBox.
SelectionStart	The zero indexed start position of the selected text in the MultiLineTextBox.
Text	The text displayed in the MultiLineTextBox control.
TexAlign	The alignment of the text in the MultiLineTextBox.
TextLength	The length of the text in the MultiLineTextBox.
WordWrap	If true, it automatically wraps words to the next line if necessary.

Function	Description
AppendText	This function appends the text to the end of the text input field.
Clear	This function clears the text input field.
ClearUndo	This function clears the most recent undo information from the undo buffer.
Copy	This function copies the selected text to the clipboard.
Cut	This function clears current selection and copies the selected text to the clipboard.
Paste	This function pasts the text on the clipboard into the MultiLineTextBox.
Select	This function selects part of the text in the MultiLineTextBox control.
SelectAll	This function selects all the text in the MultiLineTextBox control.
SetFocus	This function sets the keyboard focus to the MultiLineTextBox control.
Undo	This function undoes the last edit operation in the MultiLineTextBox control.

P r o p e r t i e s :

---

int IMultiLineTextBox::BorderStyle

This property controls the border style of the MultiLineTextBox control. It can be set to one of the values defined in the constant enum BorderStyle.

The default value of this property is Fixed3D.

Example

The BorderStyle property is illustrated in ITextBox::BorderStyle.

See Also

ITextBox::BorderStyle

---

int IMultiLineTextBox::CharacterCasing

This property controls whether the MultiLineTextBox modifies the character casing as they are typed or not. The text typed into the text input field is automatically changed to upper- or lowercase characters if this property is set to the CharacterCasing enum constants Upper or Lower, respectively.

The default value of this property is Normal, that is, the characters typed into the text input field will be left unmodified.

int IMultiLineTextBox::Enabled

This boolean property value is used to control whether the MultiLineTextBox should respond to user interaction or not.

The default value of this property is true.

int IMultiLineTextBox::MaxLength

This function controls the maximum length of the characters allowed in the MultiLineTextBox control.

The default value of this property is the maximum short integer value.

int IMultiLineTextBox::PersistContents

Setting this boolean value to true will persist the contents of the text input field to file.

The default value of this property is true.

int IMultiLineTextBox::ReadOnly

This boolean property value controls whether the MultiLineTextBox is read only or not. When the MultiLineTextBox control is read only, the text input field is disabled for keyboard input.

The default value of this property is false.

Notice that the mouse can be used to select text in a read only MultiLineTextBox control. A read only MultiLineTextBox can be used to present information to the user.

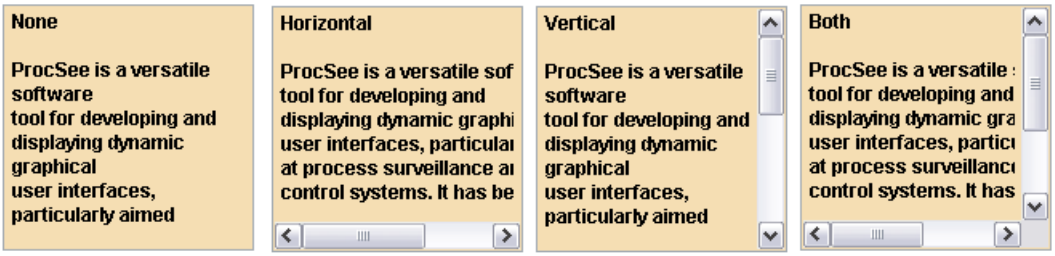
int IMultiLineTextBox::ScrollBars

This property controls which of the scroll bars to display in the MultiLineTextBox control. A MultiLineTextBox can have none, vertical, horizontal or both scroll bars visible. This property can be set to one of the constants values defined in the enum ScrollBars.

The default value of this property is None.

Notice that the property WordWrap must be false if a horizontal scroll bar should appear in the MultiLineTextBox control.

Example



See Also

IMultiLineTextBox::WordWrap

char* IMultiLineTextBox::SelectedText

This property sets or gets the selected text in the MultiLineTextBox control. The empty string “” is returned if the text input field does not have a selected text.

See Also

MultiLineTextBox::SelectionLength, MultiLineTextBox::SelectionStart, MultiLineTextBox::Select, MultiLineTextBox::SelectAll

int IMultiLineTextBox::SelectionLength

This property sets or gets the length of the selected text in the MultiLineTextBox control.

See Also

MultiLineTextBox::SelectedText, MultiLineTextBox::SelectionStart, MultiLineTextBox::Select, MultiLineTextBox::SelectAll

int IMultiLineTextBox::SelectionStart

This integer property controls the start of the selected text in the MultiLineTextBox control. This property is zero indexed. The first character in the text box has the index 0.

See Also

MultiLineTextBox::SelectedText, MultiLineTextBox::SelectionLength, MultiLineTextBox::Select, MultiLineTextBox::SelectAll

char* IMultiLineTextBox::Text

This property gets or sets the text in the MultiLineTextBox control.

See Also

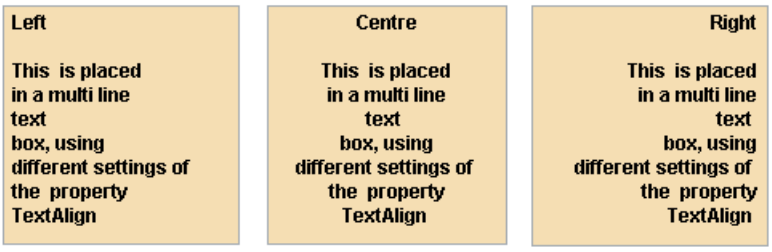
MultiLineTextBox::TextAlign, MultiLineTextBox::TextLength

int IMultiLineTextBox::TextAlign

This property controls the alignment of the text in the MultiLineTextBox control. The text can be aligned in three different positions, left, centre and right. These positions are defined in the enum HorizontalAlignment.

The default value of this property value is Left.

Example



See Also

MultiLineTextBox::Text

---

int IMultiLineTextBox::TextLength

This read only property returns the length of the text in the MultiLineTextBox control.

See Also

MultiLineTextBox::Text

---

int IMultiLineTextBox::WordWrap

This boolean property controls whether the MultiLineTextBox control automatically wraps words to the beginning of the next line when necessary or not.

The default value of this property is true.

If a horizontal scroll bar should appear in the MultiLineTextBox control, this property must be set to false and the property ScrollBar must be set to Horizontal or Both.

See Also

IMultiLineTextBox::ScrollBar

F u n c t i o n s :

---

void IMultiLineTextBox::AppendText(char* theText)

This function appends the string passed as argument to the end of the text in the MultiLineTextBox.

Parameters

theText – The string to append.

See Also

MultiLineTextBox::Text

---

void IMultiLineTextBox::Clear()

This function clears the text in the MultiLineTextBox control.

---

void IMultiLineTextBox::ClearUndo()

This function clears the most recent undo information in the MultiLineTextBox control.

See Also

MultiLineTextBox::Undo

---

void IMultiLineTextBox::Copy()

This function copies the currently selected text in the MultiLineTextBox control to the clipboard.

See Also

MultiLineTextBox::Cut, MultiLineTextBox::Paste

---

void IMultiLineTextBox::Cut()

This function copies the currently selected text to the clipboard and deletes the selected text from the MultiLineTextBox control.

See Also

MultiLineTextBox::Copy, MultiLineTextBox::Paste

---

void IMultiLineTextBox::Paste()

This function pastes the text on the clipboard to the MultiLineTextBox control replacing the selected text. The text is pasted at the insertion point if there is no selection in the MultiLineTextBox control.

See Also

MultiLineTextBox::Copy, MultiLineTextBox::Cut

---

void IMultiLineTextBox::Select(int start, int length)

This function selects a range of text in the MultiLineTextBox control in a single operation. The arguments to this function specify the start position and the length of the text to select.

Parameters

start – The zero indexed start position of the text to select.

length – Number of characters to select in the text.

See Also

MultiLineTextBox::SelectionStart, MultiLineTextBox::SelectionLength, MultiLineTextBox::SelectAll

---

void IMultiLineTextBox::SelectAll()

This function selects the entire text in the MultiLineTextBox control.

See Also

MultiLineTextBox::SelectionStart, MultiLineTextBox::SelectionLength, MultiLineTextBox::Select

---

int IMultiLineTextBox::SetFocus()

This function sets the keyboard focus to the MultiLineTextBox control.

See Also

MultiLineTextBox::ClearUndo

---

void IMultiLineTextBox::Undo()

This function undoes the last edit operation in the MultiLineTextBox control.

See Also

MultiLineTextBox::ClearUndo

IMultiLineTextBoxEvents

The IMultiLineTextBoxEvents event interface has the following members.

Function	Description
OnMouseDownCB	This callback fires when one of the mouse buttons are pressed in the text box.
OnMouseUpCB	This callback fires when one of the mouse buttons are released in the text box.
OnKeyPressCB	This function fires when a character is typed in the MultiLineTextBox control.
OnTextChangedCB	This callback fires when the text changes.

F u n c t i o n s :

---

void IMultiLineTextBoxEvents:: OnMouseDownCB(int button)

This callback function fires when one of the mouse buttons is pressed in the text box.

See Also

IMultiLineTextBoxEvents::OnMouseUpCB

---

void IMultiLineTextBoxEvents:: OnMouseUpCB(int button)

This callback function fires when one of the mouse buttons is released in the text box.

See Also

IMultiLineTextBoxEvents::OnMouseDownCB

---

void IMultiLineTextBoxEvents::OnKeyPressCB(unsigned short int c)

This function is fired when a character is typed in the MultiLineTextBox control.

Parameters

c – The character typed in the MultiLineTextBox

See Also

IMultiLineTextBoxEvents:: OnTextChangedCB

---

void IMultiLineTextBoxEvents:: OnTextChangedCB(char* string)

This callback function fires when the text changes.

See Also

IMultiLineTextBoxEvents::OnKeyPressCB

ProgressBar

The ProgressBar control represents a standard Microsoft Windows progress bar. A progress bar is a graphical component commonly used to indicate the progress of a long running operation, like a calculation. As the long running process continues, the progress bar visually indicates the progress of the operation by filling an appropriate number of rectangles arranged in a horizontal bar. When the action is complete, the bar is filled.

The ProgressBar control has a range from a minimum to a maximum value.

The ProgressBar control supports two interfaces, IProgressBar and IProgressBarEvents. The IProgressBar interface, by default named *di*, is used to modify the contents of or change the visual appearance of the ProgressBar control. The event interface, IProgressBarEvents, is by default named *dei*.

Note that the colours of the ProgressBar can not be changed. The colours are operating system dependent. The position and size of the ProgressBar is controlled by the ComShape properties x, y, width and height, respectively.

IProgressBar

The IProgressBar interface has the following properties and functions.

Property	Description
Enabled	This property controls whether the ProgressBar responds to user interaction or not.
Maximum	The upper limit of the ProgressBar's range.
Minimum	The lower limit of the ProgressBar's range.
Step	Step value to add to the property Value when the function PerformStep is called.
Value	Current value of the ProgressBar control.

Function	Description
Increment	Increments the property Value by the value passed as argument.
PerformStep	Increments the property Value by the property Step value.

P r o p e r t i e s :

int IProgressBar::Enabled

This property controls whether the ProgressBar responds to user interaction or not.

The default value for this property is true.

int IProgressBar::Maximum

This property controls the upper limit of the ProgressBar.

The default value for this property is 100.

See Also

IProgressBar::Value, IProgressBar::Minimum, IProgressBar::Step, IProgressBar::Increment, IProgressBar::PerformStep

int IProgressBar::Minimum

This property controls the lower limit of the ProgressBar.

The default value for this property is 0.

See Also

IProgressBar::Value, IProgressBar::Maximum, IProgressBar::Step, IProgressBar::Increment, IProgressBar::PerformStep

int IProgressBar::Step

This property controls the step value to add to the current value position when the function PerformStep is called.

The default value for this property is 10.

See Also

IProgressBar::Value, IProgressBar::Minimum, IProgressBar::Maximum, IProgressBar::Increment, IProgressBar::PerformStep

int IProgressBar::Value

This property represents current position in the ProgressBar control.

Example

See Also

IProgressBar::Maximum, IProgressBar::Minimum, IProgressBar::Step, IProgressBar::Increment, IProgressBar::PerformStep

F u n c t i o n s :

void IProgressBar::Increment(int amount)

This function increments current position of the ProgressBar control by the amount specified in the argument amount.

Compared to the function PerformStep is this function more flexible because the increment value can be specified.

Parameters

amount – The amount to increment current position of the ProgressBar.

See Also

IProgressBar::Maximum, IProgressBar::Minimum, IProgressBar::Step, IProgressBar::Value, IProgressBar::PerformStep

---

void IProgressBar::PerformStep()

This function increments current position of the ProgressBar control by the amount specified in the property Step.

See Also

IProgressBar::Maximum, IProgressBar::Minimum, IProgressBar::Step, IProgressBar::Value, IProgressBar::Increment

---

RadioButton

The RadioButton control represents a standard radio button in Microsoft Windows. Radio button controls are normally used for settings states or modes. Radio buttons are typically grouped together where each button represents a mutually exclusive selection. Grouping radio buttons are normally done using the GroupId property.

A radio button is typically displayed with a check box and an associated text, but can also appear as a toggle button.

The RadioButton control supports two interfaces, IRadioButton and IRadioButtonEvents. The IRadioButton interface, by default named *di*, is used to modify the contents of or change the visual appearance of the RadioButton control. The event interface, IRadioButtonEvents, is by default named *dei*.

Note that the IRadioButton interface does not provide properties and methods to change the RadioButton's font or colours. This functionality is instead supported by the ComShape attributes theFont, foregroundColour and backgroundColour. The position and size of the RadioButton are controlled by the ComShape attributes x, y, width and height, respectively.

Double clicking on a RadioButton control while the picture is in edit mode will put the control into user input mode. In this mode the RadioButton control will display a radio button mark together with a text input field. The user can type the radio button text in the text input field. To set the initial state of the radio button, click on the radio button mark. Clicking outside the control will put the RadioButton back to normal mode.The text entered in the input field and the radio button mark will be displayed in the control.

IRadioButton

The IRadioButton interface has the following properties.

Property	Description
Appearance	Controls the appearance of the RadioButton.
AutoCheck	Determines whether the RadioButton automatically changes state or not.
CheckAlign	Controls the alignment of the check box on the RadioButton control.
Checked	Determines the state of the RadioButton, checked or unchecked.
Enabled	Whether the control can be clicked or not.
FlatStyle	Controls the style of the RadioButton.
GroupId	The group identifier associated with the RadioButton.
Text	The text displayed in the RadioButton.
TextAlign	Controls the alignment of the text on the RadioButton control.

P r o p e r t i e s :

int IRadioButton::Appearance

This property controls the appearance of the RadioButton. A RadioButton can visually be displayed like a toggle button or as a standard radio button. The Appearance enum constants control the appearance of the RadioButton.

The default for this property is Normal.

Example

See Also

Appearance enum constants are described on page 113.

int IRadioButton::AutoCheck

This property determines whether the RadioButton automatically changes the checked state when it is clicked or not.

When the AutoCheck property is set to true the radio buttons in the same group will automatically be checked and unchecked. A group in this context is either RadioButton ComShapes in the same scope or radio buttons with same GroupId property. Examples of radio button groups in different scopes are RadioButtons added to Group entries, Instance entries or Picture entries.

If this property is false the RadioButton state must be set manually. The event function OnClickCB is fired when the RadioButton is clicked. To manually change the state of a RadioButton, use the property Checked in the OnClickCB callback function. The purpose with radio buttons is to have one and only one selected radio button in a group. Note that each of the radio buttons in the logical group must manually be set to checked and unchecked states when this property is false.

The default value for this property is true.

Example

The following example demonstrates three radio buttons in a group where the property AutoCheck is false. When one of the radio buttons is clicked its checked state must be set to true and the two other radio buttons in the group must be set to unchecked state. The example illustrates the OnClickCB call-back function for the Date radio button.


```
void OnClickCB()
{
    if ( !rbDate.di.Checked )
    {
       rbDate.di.Checked       = 1;
       rbAscending.di.Checked  = 0;
       rbDescending.di.Checked = 0;
    }
}
```

See Also

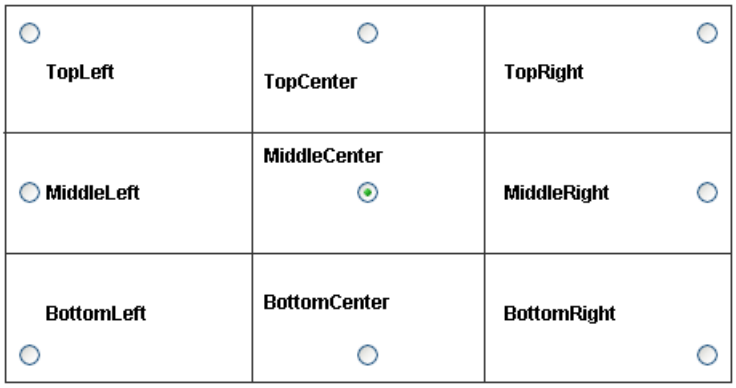
IRadioButtonEvents::OnClickCB

int IRadioButton::CheckAlign

This property controls the vertical and horizontal alignment of the check box on the RadioButton control. It can be aligned on nine different positions inside the control. To set this property use the constants defined in the enum ContentAlignment.

The default value for this property is MiddleLeft.

Example



See Also

ContentAlignment enum constants are described on page 114.

int IRadioButton::Checked

This property gets or sets the checked state of the RadioButton control.

The default value is false.

See Also

IRadioButton::AutoCheck, IRadioButtonEvents::OnClickCB

int IRadioButton::Enabled

This property must be set to true if the RadioButton control should respond to user interaction, otherwise false. The default value is true. To restrict the RadioButton from being used, set this property value to false.

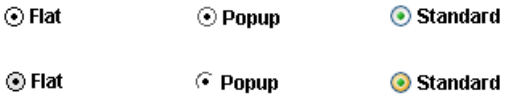
int IRadioButton::FlatStyle

This property controls the flat style of the RadioButton control. The constants in the enum FlatStyle define the legal values of this property.

The default value for this property is Standard.

Example

This example illustrates the different flat styles which can be used on RadioButtons. The radio buttons in the last row displays the same flat styles as in the first row, but the controls in the last row shows how the controls appear when they have focus.



See Also

FlatStyle enum constants are described on page 115.

---

int IRadioButton::GroupId

This property controls the group identifier associated with the RadioButton control. This property only has a purpose when the property AutoCheck is true.

Setting the property GroupId to the same value for radio buttons belonging together in a logical group will connect these radio buttons together. RadioButton controls with same GroupId will not affect RadioButton controls with another GroupId value. In this way several groups can be created in the same scope. The scope can be a Group entry, Instance entry or Picture entry.

See Also

IRadioButton::AutoCheck

---

char* IRadioButton::Text

This property sets or gets the text displayed in the RadioButton control.

---

int IRadioButton::TextAlign

This property controls the alignment of the text in the RadioButton control. The text can be aligned in nine different positions inside the RadioButton. The legal values for this property are defined in the enum ContentAlignment.

The default value for this property is MiddleLeft.

Example

To view an example of ContentAlignment refer to the examples in the description of IButton::TextAlign and IRadioButton::CheckAlign.

See Also

ContentAlignment enum constants are described on page 114.

IRadioButtonEvents

The IRadioButtonEvents interface has the following properties.

Function	Description
OnCheckedChangedCB	Fired when the state of the radio button change.
OnClickCB	Fired when the radio button is clicked.

F u n c t i o n s :

void IRadioButtonEvents::OnCheckedChangedCB(int isChecked)

This event function is fired when the RadioButton changes state either automatically or programmatically from pTALK.

Parameters

isChecked – Current value of the RadioButton control.

Example

The following example demonstrates how to implement a checked changed event function in the pTALK language. This example checks whether the parameter isChecked is true and if so calls the eventList instance function SortByDate.

```
void OnCheckedChangedCB(int isChecked)
{
   if ( isChecked )
   {
      eventList.SortByDate();
   }
}
```

See Also

IRadioButtonEvents::OnClickCB

void IRadioButtonEvents::OnClickCB()

This event function is fired when the RadioButton is clicked.

See Also

IRadioButton::AutoCheck, IRadioButtonEvents::OnCheckedChangedCB

ScrollBar

The ScrollBar control represents a standard Microsoft Windows scroll bar. The ScrollBar has a range from a minimum to a maximum value where the user can change values within the range minimum to maximum minus page size plus 1.

The ScrollBar control supports two interfaces, IScrollBar and IScrollBarEvents. The IScrollBar interface, by default named *di*, is used to modify the contents of or change the visual appearance of the ScrollBar control. The event interface, IScrollBarEvents, is by default named *dei*.

Note that the colours of the ScrollBar can not be changed. The colours are operating system dependent. The position and size of the ScrollBar are controlled by the ComShape attributes x, y, width and height, respectively.

IScrollBar

The IScrollBar interface has the following properties and functions.

Property	Description
Enabled	This property controls whether the ScrollBar responds to user interaction or not.
LargeChange	The value changes according to this value when clicking on either side of the

	scroll box. This value is also known as page size.
Maximum	The upper limit of the ScrollBar range.
Minimum	The lower limit of the ScrollBar range.
Orientation	This property controls the orientation, vertical or horizontal.
SmallChange	The value changes according to this value when the arrows are clicked or when the scroll box is moved a small distance.
Value	Current position in the ScrollBar.

Function	Description
SetRange	Sets upper and lower limits of the ScrollBar range.

P r o p e r t i e s :

int IScrollBar::Enabled

This property controls whether the ScrollBar responds to user interaction or not.

The default value for this property is true.

int IScrollBar::LargeChange

The property Value changes according to this value when the mouse is clicked on either side of the scroll box (in the trough). This property is also known as page size.

The default value for this property is 10.

See Also

IScrollBar::SmallChange

int IScrollBar::Maximum

This property controls the upper limit of the ScrollBar range.

The default value for this property is 100.

See Also

IScrollBar::Minimum, IScrollBar::SetRange

int IScrollBar::Minimum

This property controls the lower limit of the ScrollBar range.

The default value for this property is 0.

See Also

IScrollBar::Maximum, IScrollBar::SetRange

int IScrollBar::Orientation

This property specifies the orientation of the ScrollBar. The ScrollBar can be vertical or horizontal. The constant values for this property are defined in the enum Orientation.

The default value for this property is Vertical.

See Also

Orientation enum is described on page 116.

---

int IScrollBar::SmallChange

The property Value changes according to this value either when moving the scroll box a small distance or when clicking on the up or down buttons on the ScrollBar.

The default value for this property is 1.

See Also

IScrollBar::LargeChange

---

int IScrollBar::Value

This property represents current position of the scroll box in the ScrollBar. The value must be in the legal range between the properties Minimum and Maximum.

Example

See Also

IScrollBar::Minimum, IScrollBar::Maximum

F u n c t i o n s :

---

void IScrollBar::SetRange(int minValue, int maxValue)

This function sets the minimum and maximum limits for the ScrollBar.

Parameters

minValue – This ScrollBar's lower limit (property Minimum) is set to this value.

maxValue – This ScrollBar's upper limit (property Maximum) is set to this value.

See Also

IScrollBar::Minimum, IScrollBar::Maximum

IScrollBarEvents

The IScrollBarEvents interface has the following event functions.

Function	Description
OnValueChangedCB	This event is fired when the Value property changes value.

F u n c t i o n s :

void IScrollBarEvents::OnValueChangedCB(int theValue, int theType)

This event is fired when the scroll box in the ScrollBar has been changed either by the mouse or programmatically from pTALK.

The last argument to this function is the type of ScrollBar event that occurred. This argument can be used to improve the performance of the system. If the ScrollBar is associated with a list, the list can for instance be updated when the parameter theType is unequal to the constant enum value ThumbTrack. The ThumbTrack events are fired when moving the scroll box with the mouse.

The event types are defined in the enum ScrollBarEventType.

Notice that this event is fired with the theType parameter set to the constant enum value EndScroll, when the mouse is released after moving the scroll box with the mouse.

Parameters

theValue – The new position of the scroll box on the ScrollBar.

theType – This parameter specifies the type of event that occurred.

Range

The legal value range for the parameter theValue is the property value Minimum to the property value Maximum minus the property LargeChange plus 1. See definition next.

```
Minimum to ( Maximum – LargeChange + 1 ).
```

Example

The following example demonstrates how to optimize the performance of the system when scrolling through an alarm list. The alarm list is not updated when the scroll box is moved by the mouse.

```
void OnValueChangedCB(int theValue, int theType)
{
  if ( theType != ProcSeeControls.ScrollBarEventType.ThumbTrack )
   {
       alarmListTop = theValue;
       alarmList.updateShape();
   }
}
```

See Also

enum constants are defined on page 116.

TextBox

The TextBox control represents a standard Microsoft Windows single-line text input field. A TextBox is a graphical component that is normally used to allow users to input text information to the ProcSee application.

The TextBox control supports two interfaces, ITextBox and ITextBoxEvents. The ITextBox interface, by default named *di*, is used to modify the contents of or change the visual appearance of the TextBox control. The event interface, ITextBoxEvents, is by default named *dei*.

Note that the ITextBox interface does not provide properties and methods to change the TextBox's font or colours. This functionality is instead supported by the ComShape attributes theFont, foregroundColour and backgroundColour. The position and size of the TextBox are controlled by the ComShape attributes x, y, width and height, respectively.

ITextBox

The ITextBox interface has the following properties and functions.

Property	Description
BorderStyle	This property controls the border style.
CharacterCasing	Controls whether the TextBox modifies the character casing or not.
Enabled	Controls whether the TextBox is enabled or not.
MaxLength	Maximum length of the text in the TextBox control.
PasswordChar	This property controls the character used to mask the character of a password.
PersistContents	This property controls whether the input text is persisted or not.
ReadOnly	Controls whether the TextBox is read only.
SelectedText	The selected text in the TextBox.
SelectionLength	The length of the selected text in the TextBox.
SelectionStart	The zero indexed start position of the selected text in the TextBox.
Text	The text displayed in the TextBox control.
TextAlign	The alignment of the text in the TextBox.
TextLength	The length of the text in the TextBox.

Function	Description
AppendText	This function appends the text to the end of the text input field.
Clear	This function clears the text input field.
ClearUndo	This function clears the most recent undo information from the undo buffer.
Copy	This function copies the selected text to the clipboard.
Cut	This function clears current selection and copies the selected text to the clipboard.
Paste	This function pasts the text on the clipboard into the TextBox control.
Select	The function selects part of the text in the TextBox control.
SelectAll	The function selects all the text in the TextBox control.
SetFocus	This function sets the keyboard focus to the TextBox control.
Undo	This function undoes the last edit operation in the TextBox control.

P r o p e r t i e s :

int ITextBox::BorderStyle

This property controls the border style of the TextBox control. It can be set to one of the values defined in the constant enum BorderStyle.

The default value for this property is Fixed3D.

Example

---

int ITextBox::CharacterCasing

This property controls whether the TextBox modifies the characters casing as they are typed or not. The text typed into the text input field will automatically be changed to upper- or lowercase characters if this property is set to the CharacterCasing enum constants Upper or Lower, respectively.

The default value for this property is Normal, that is, the characters typed into the text input field will be left unmodified.

---

int ITextBox::Enabled

This boolean property value is used to control whether the TextBox should respond to user interaction or not. The default value of this property is true.

---

int ITextBox::MaxLength

This function controls the maximum length of characters allowed in the TextBox control. The default value of this property is the maximum short integer value.

---

unsigned ITextBox::PasswordChar

This property controls the character used to mask characters of a password when typed into the TextBox control.

The default value of this property is 0. When this property value is set to a value different from 0, the text input field will display this character instead of the character typed.

When used as a password text input field, the password character is normally the asterisk, '*'.

Example

---

int ITextBox::PersistContents

Setting this boolean value to true will persist the contents of the text input field of the TextBox to the saved file.

---

int ITextBox::ReadOnly

This boolean property value controls whether the TextBox is read only or not. When the TextBox control is read only, the text input field is disabled from keyboard input.

The default value of this property is false.

Notice that the mouse can be used to select text in a read only TextBox control.

---

char* ITextBox::SelectedText

This property sets or gets the selected text in the TextBox control. If the text input field does not have a selected text, the empty string "" is returned.

See Also

ITextBox::SelectionLength, ITextBox::SelectionStart

---

int ITextBox::SelectionLength

This property sets or gets the length of the selected text in the TextBox control.

See Also

ITextBox::SelectedText, ITextBox::SelectionStart, ITextBox::Select, ITextBox::SelectAll

---

int ITextBox::SelectionStart

This integer property controls the start of the selected text in the TextBox control. This property is zero indexed. The first character in the text box has the index 0.

Example

The following example selects the text from the first character after ':' to the last character before the next ':'. The name of the instance of the TextBox control in this example is TB.

```
…
char* t = TB.di.Text;

char* sStart = strchr( t, ':' );
char* sField = strfield( t, 1, ':' );
if ( !sStart || !sField ) return;

TB.di.SelectionStart  = strlen( t ) - strlen( sStart ) + 1;
TB.di.SelectionLength = strlen( sField );
…
```

See Also

ITextBox::SelectedText, ITextBox::SelectionLength, ITextBox::Select, ITextBox::SelectAll

---

char* ITextBox::Text

This property gets or sets the text in the TextBox control.

See Also

ITextBox::TextAlign, ITextBox::TextLength

---

int ITextBox::TextAlign

This property controls the alignment of the text in the TextBox control. The text can be aligned in three different positions, left, centre and right. These positions are defined in the enum HorizontalAlignment.

The default value of this property is Left.

Example

See Also

ITextBox::Text

---

int ITextBox::TextLength

This read only property returns the length of the text in the TextBox control.

See Also

ITextBox::Text

F u n c t i o n s :

---

void ITextBox::AppendText(char* theText)

This function appends the string passed as argument to the end of the text in the TextBox.

Parameters

theText – The string to append.

See Also

ITextBox::Text

---

void ITextBox::Clear()

This function clears the text in the TextBox control.

---

void ITextBox::ClearUndo()

This function clears the most recent undo information from the undo buffer in the TextBox control.

See Also

ITextBox::Undo

---

void ITextBox::Copy()

This function copies the currently selected text in the TextBox control to the clipboard.

See Also

ITextBox::Cut, ITextBox::Paste

---

void ITextBox::Cut()

This function copies the currently selected text to the clipboard and deletes the selected text from the TextBox control.

See Also

ITextBox::Copy, ITextBox::Paste

void ITextBox::Paste()

This function pastes the text on the clipboard to the TextBox control replacing the selected text. The text is pasted at the insertion point if there is no selection in the TextBox control.

See Also

ITextBox::Copy, ITextBox::Cut

void ITextBox::Select(int start, int length)

This function selects a range of text in the TextBox control. The arguments to this function specify the start position and the length of the text to select.

Parameters

start – The start index of the text to select. The first character has the index 0.

length – Number of characters to select.

See Also

ITextBox::SelectionStart, ITextBox::SelectionLength, ITextBox::SelectAll

void ITextBox::SelectAll()

This function selects the entire text in the TextBox control.

See Also

ITextBox::SelectionStart, ITextBox::SelectionLength, ITextBox::Select

int ITextBox::SetFocus()

This function sets the keyboard focus to the TextBox control.

Return Value

If the keyboard focus request was successful, true is returned, otherwise false.

void ITextBox::Undo()

This function undoes the last edit operation in the TextBox control.

See Also

ITextBox::ClearUndo

ITextBoxEvents

The ITextBoxEvents event interface has the following functions.

Function	Description
OnKeyPressCB	This function is fired when a character is typed in the TextBox control.
OnMouseDownCB	This callback fires when one of the mouse buttons are pressed in the text box.

OnMouseUpCB	This callback fires when one of the mouse buttons are released in the text box.
OnReturnCB	This function is fired when the return key is pressed in the TextBox control.
OnTabCB	This function is fired when the tab key is pressed in the TextBox control.
OnTextChangedCB	This callback fires when the text changes.

F u n c t i o n s :

void ITextBoxEvents::OnKeyPressCB(unsigned short int c)

This callback function fires when a character is typed in the TextBox.

Parameters

c – The character typed in the TextBox.

See Also

ITextBoxEvents::OnReturnCB, ITextBoxEvents::OnTabCB, ITextBoxEvents::OnTextChangedCB

void ITextBoxEvents::OnMouseDownCB(int button)

This callback function fires when one of the mouse buttons is pressed in the text box.

See Also

IMultiLineTextBoxEvents::OnMouseUpCB

void ITextBoxEvents::OnMouseUpCB(int button)

This callback function fires when one of the mouse buttons is released in the text box.

See Also

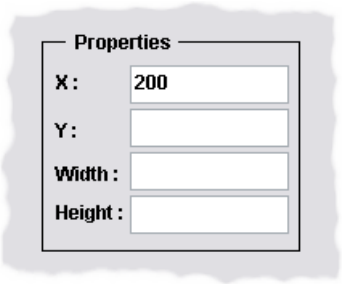
IMultiLineTextBoxEvents::OnMouseDownCB

void ITextBoxEvents::OnReturnCB()

This callback function is fires when the return key is pressed in the TextBox.

Example

A feature that most Microsoft Windows applications support is the possibility to automatically jump to the next text input field when the tab character or return key is pressed. The following example illustrates the implementation of the OnReturnCB call-back function for the first TextBox control in the text input group, see the figure below. The source code that follows automatically moves the keyboard focus to the next TextBox control, named yPropTextInput, when the return key is pressed. The code example is for the first text input field in the group. A similar code must be implemented for the other text input fields in the group as well.




```
void OnReturnCB()
{
  yPropTextInput.di.SetFocus();
}
```

See Also

ITextBoxEvents::OnKeyPressCB, ITextBoxEvents::OnTabCB, ITextBox::SetFocus, ITextBoxEvents::OnTextChangedCB

---

void ITextBoxEvents::OnTabCB()

This call-back function is fired when the tab key is pressed in the TextBox.

See Also

ITextBoxEvents::OnKeyPressCB, ITextBoxEvents::OnReturnCB, ITextBoxEvents::OnTextChangedCB

---

void ITextBoxEvents::OnTextChangedCB(char* string)

This callback function fires when the text changes.

See Also

ITextBoxEvents::OnKeyPressCB, ITextBoxEvents::OnReturnCB, ITextBoxEvents::OnTabCB

TrackBar

The TrackBar control represents a standard track bar in Microsoft Windows. A track bar control is a graphical component that both looks like and behaves rather like a scroll bar. The track bar has a fixed sized slider, which moves only in fixed increments, as opposed to a scroll bar control, which can be moved continuously. The track bar has a scale of tick marks displayed above, below or to one side of the control.

The TrackBar control supports two interfaces, ITrackBar and ITrackBarEvents. The ITrackBar interface, by default named *di*, is used to modify the contents of or change the visual appearance of the TrackBar control. The event interface, ITrackBarEvents, is by default named *dei*.

Note that the ITrackBar interface does not provide properties or methods to change the background colour of the TrackBar control. This functionality is instead supported by the ComShape attribute backgroundColour. The position and size of the TrackBar are controlled by the ComShape attributes x, y, width and height, respectively.

ITrackBar

The ITrackBar interface supports the following properties and functions.

Property	Description
Enabled	This property controls whether the TrackBar responds to user interaction or not.
LargeChange	The value changes according to this value when clicking on either side of the slider.
Maximum	The upper limit of the TrackBar.
Minimum	The lower limit of the TrackBar.
Orientation	This property controls whether the TrackBar is horizontal or vertical.
SmallChange	The value changes according to this value when the slider is moved a small distance.
TickFrequency	The delta between ticks on the TrackBar.
TickStyle	How to display the tick marks on the TrackBar.
Value	Current position in the TrackBar.

Function	Description
SetRange	Sets upper and lower limit of the TrackBar.

P r o p e r t i e s :

int ITrackBar::Enabled

This property controls whether the TrackBar responds to user interaction or not.

The default value for this property is true.

int ITrackBar::LargeChange

The property Value changes according to this value when the mouse is clicked on either side of the slider.

The default value for this property is 5.

See Also

ITrackBar::Value, ITrackBar::Minimum, ITrackBar::Maximum, ITrackBar::SetRange, ITrackBar::SmallChange

int ITrackBar::Maximum

This property controls the upper limit of the TrackBar.

The default value for this property is 10.

See Also

ITrackBar::Value, ITrackBar::Minimum, ITrackBar::SetRange

int ITrackBar::Minimum

This property controls the lower limit of the TrackBar.

The default value for this property is 0.

See Also

ITrackBar::Value, ITrackBar::Maximum, ITrackBar::SetRange

int ITrackBar::Orientation

This property specifies the orientation of the TrackBar. The TrackBar can be vertical or horizontal. The constant values for this property are defined in the enum Orientation.

The default value for this property is Horizontal.

See Also

Orientation enum is described on page 116.

int ITrackBar::SmallChange

The property Value changes according to this value when moving the slider a small distance.

The default value for this property is 1.

See Also

ITrackBar::Value, ITrackBar::Minimum, ITrackBar::Maximum, ITrackBar::SetRange, ITrackBar::SmallChange

int ITrackBar::TickFrequency

This property specifies the delta between the tick marks on the slider.

The default value for this property is 1.

See Also

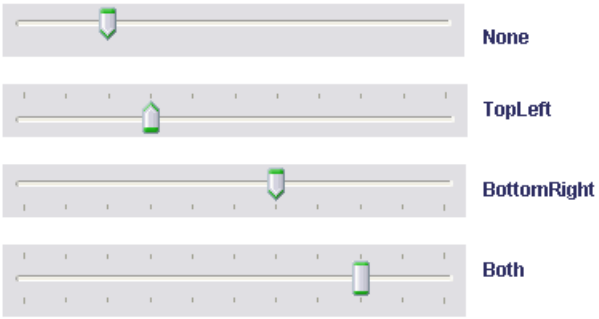
ITtrackBar::TickStyle

int ITrackBar::TickStyle

The property TickStyle controls how to display the tick marks on the TrackBar. This property can be set equal to the constant values defined in the enum TickStyle.

The default value for this property is BottomRight.

Example



See Also

TickStyle enum is described on page 116.

ITrackBar::TickFrequency

int ITrackBar::Value

This property represents current position in the TrackBar. The value must be in the legal range between the properties Minimum and Maximum.

See Also

ITrackBar::Minimum, ITrackBar::Maximum

F u n c t i o n s :

void ITrackBar::SetRange(int minValue, int maxValue)

This function sets the minimum and the maximum limits for the TrackBar.

Parameters

minValue – This TrackBar's lower limit (property Minimum) is set to this value.

maxValue – The TrackBar's upper limit (property Maximum) is set to this value.

See Also

ITrackBar::Minimum, ITrackBar::Maximum

ITrackBarEvents

The ITrackBarEvents interface supports the following functions.

Function	Description
OnScrollCB	This event is fired when the slider changes position.

F u n c t i o n s :

void ITrackBarEvents::OnScrollCB(int theValue)

This event is fired when the slider changes position in the TrackBar.

Parameters

theValue – The new position of the slider on the TrackBar.

Example

The following example changes the text shape tValue according to the current value of the TrackBar.

```
void OnScrollCB(int theValue)
{
    tValue.theValue = theValue;
    tValue.updateShape();
}
```

7

.NET Components

Utilities

The Utility component is a placeholder for miscellaneous functionality offered by the User Interface Controls library. Through this utility component, functionality like message boxes, file selection boxes etc. can be used in ProcSee applications.

The Utility component offers a set of simple functions, which are easy to use. These functions have the prefix Show, like ShowErrorMessageBox or ShowFileOpenDialog.

The utility component also provides functionality to customize message boxes and file dialogues. In situations where the simple functions does not offer the required functionality, interfaces to the message box and file selection dialogues are provided. The interfaces give the user a lot of options where the dialogues can be tailor-made to the ProcSee application.

Refer to *Basic use of the Utilities* component on page 22 and *Using the general file selection interfaces* on page 25 for examples on how to use the utility component.

IUtilities

The IUtilities interface supports the following functions.

Function	Description
FileOpen	Returns an interface to a customizable IFileOpen interface.
FileSave	Returns an interface to a customizable IFileSave interface.
MessageBox	Returns an interface to a customizable IMessageBox interface.
ShowErrorMessageBox	Displays an error message box.
ShowFileOpenDialog	Displays an open file selection box.
ShowFileSaveDialog	Displays a save file selection box.
ShowInformationMessageBox	Displays an information message box.
ShowMessageBox	Displays a general message box without icon.
ShowQuestionMessageBox	Displays a question message box with yes and no buttons.
ShowWarningMessageBox	Displays a warning message box.
VersionNumber	Returns the version number of the ProcSeeControls.
VersionString	Returns the version of the ProcSeeControls as a string.

F u n c t i o n s :

::ProcSeeControls.IFileOpen* IUtilities::FileOpen()

This function returns a pointer to an IFileOpen interface. The IFileOpen interface provides functionality to tailor-make the file open dialogues. For instance, a customized file open dialogue allows the user to select and open one or several files.

Return Value

This function returns a pointer to an IFileOpen interface.

Example

```
…
ProcSeeControls.IFileOpen* fopen;
fopen = (ProcSeeControls.IFileOpen *)UIUtility.FileOpen();
…
fopen->ShowDialog();
```

See Also

The IFileOpen interface is described on page 97.

IUtility::ShowFileOpenDialog

::ProcSeeControls.IFileSave* IUtilities::FileSave()

This function returns a pointer to an IFileSave interface. The IFileSave interface provides functionality to tailor-make the file save dialogues. For instance, a customized file save dialogue can display a warning if the user specifies a file that does not exist or prompt the user for permission to create the file.

Return Value

This function returns a pointer to an IFileSave interface.

Example

```
…
ProcSeeControls.IFileSave* fsave;
fsave = (ProcSeeControls.IFileSave *)UIUtility.FileSave();
…
fsave->ShowDialog();
```

See Also

The IFileSave interface is described on page 102.

IUtility::ShowFileSaveDialog

::ProcSeeControls.IMessageBox* IUtilities::MessageBox()

This function returns a pointer to an IMessageBox interface. This function provides the user interface designer with message box functionality that the simple functions do not offer. When using the IMessageBox interface, the message box icon, the buttons, default button, etc. can be customized.

Return Value

This function returns a pointer to an IMessageBox interface.

Example

```
…
ProcSeeControls.IMessageBox* msgBox;
msgBox = (ProcSeeControls.IMessageBox*)UIUtility.MessageBox();
…
msgBox->ShowDialog();
```

See Also

The IMessageBox interface is described on page 105.

IUtility::ShowErrorMessageBox, IUtility::ShowInformationMessageBox, IUtility::ShowMessageBox, IUtility::ShowQuestionMessageBox, IUtility::ShowWarningMessageBox.

---

int IUtilities::ShowErrorMessageBox(char* text, char* caption)

This function opens a modal error message box. The message box displays an error icon and an OK button. The user must respond to the error message box before the application can continue executing pTALK code.

Parameters

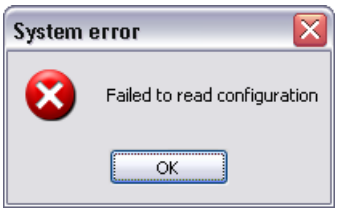
text – The text to display in the error message box.

caption – The text to display in the caption bar of the message box.

Return Value

This function returns the enum DialogResult constant value OK.

Example



See Also

IUtility:: MessageBox, IUtility::ShowInformationMessageBox, IUtility::ShowMessageBox, IUtility::ShowQuestionMessageBox, IUtility::ShowWarningMessageBox.

---

char* IUtilities::ShowFileOpenDialog(char* initialDirectory, char* filter)

This function displays the open file dialogue where the user can select a file to open. The directory to view is specified in the first argument to this function. The last argument filter specifies the file types to view. For a description of the filter argument, see IFileOpen on page 97.

Parameters

initialDirector – The location of the initial directory to view.

filter – The filter pattern, specifies the type of files to view.

Return Value

This function returns the name of the selected file. The name includes the directory name and extension.

See Also

The interface IFileOpen is described on page 97.

IUtilities::ShowFileSaveDialog

---

char* IUtilities::ShowFileSaveDialog(char* initialDirectory, char* filter)

This function displays the save file dialogue where the user can select or enter a name of the file to save. The directory to view is specified in the first argument to this function. The last argument filter specifies the file types to view. For a description of the filter argument, see IFileOpen on page 97.

Parameters

initialDirector – The location of the initial directory to view.

filter – The filter pattern, specifies the type of files to view.

Return Value

This function returns the name of the selected file. The name includes the directory name and extension.

See Also

The interface is described IFileSave on page 102.

IUtilities::ShowFileOpenDialog

---

int IUtilities::ShowInformationMessageBox(char* text, char* caption)

This function opens a modal information message box. The message box displays an information icon and an OK button. The user must respond to the information message box before the application can continue executing pTALK code.

Parameters

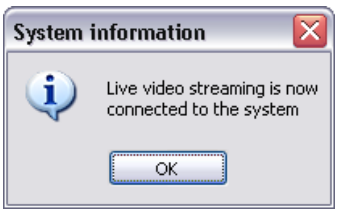
text – The text to display in the information message box.

caption – The text to display in the caption bar of the message box.

Return Value

This function returns the enum DialogResult constant value OK.

Example



See Also

IUtility:: MessageBox, IUtility::ShowErrorMessageBox, IUtility::ShowMessageBox, IUtility::ShowQuestionMessageBox, IUtility::ShowWarningMessageBox.

---

int IUtilities::ShowMessageBox(char* text, char* caption)

This function opens the general modal message box. The message box does not contain an icon. The user must click the OK button before the application can continue executing pTALK code.

Parameters

text – The text to display in the message box.

caption – The text to display in the caption bar of the message box.

Return Value

This function returns the enum DialogResult constant value OK.

Example

See Also

IUtility:: MessageBox, IUtility::ShowErrorMessageBox, IUtility::ShowMessageBox, IUtility::ShowQuestionMessageBox, IUtility::ShowWarningMessageBox.

---

int IUtilities::ShowQuestionMessageBox(char* text, char* caption)

This function opens a modal question message box. The message box displays a question icon and Yes and No buttons. The user must respond to the question message box before the application can continue executing pTALK code.

Parameters

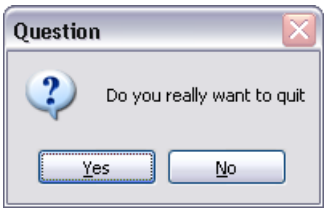
text – The text to display in the question message box.

caption – The text to display in the caption bar of the message box.

Return Value

This function returns either the enum DialogResult constant value Yes if the Yes button was pressed or No if the No button was pressed.

Example



See Also

IUtility:: MessageBox, IUtility::ShowErrorMessageBox, IUtility::ShowMessageBox, IUtility::ShowInformationMessageBox, IUtility::ShowWarningMessageBox.

---

int IUtilities::ShowWarningMessageBox(char* Text, char* Caption)

This function opens a modal warning message box. The message box displays a warning icon and an OK button. The user must respond to the error message box before the application can continue executing pTALK code.

Parameters

text – The text to display in the error message box.

caption – The text to display in the caption bar of the message box.

Return Value

This function returns the enum DialogResult constant value OK.

Example

See Also

IUtility:: MessageBox, IUtility::ShowInformationMessageBox, IUtility::ShowMessageBox, IUtility::ShowQuestionMessageBox, IUtility::ShowWarningMessageBox.

---

int IUtilities::VersionNumber()

This function returns the version number of the ProcSeeControls. This number is an integer consisting of a major and a minor version number. The format is XXYY, where XX is the major version number, and YY is the minor version number.

Return Value

This function returns the version number.

See Also

IUtility::VersionString.

---

char* IUtilities::VersionString()

This function returns the version of the ProcSeeControls including the ProcSee release information.

Return Value

This function returns the version as a string.

See Also

IUtility::VersionNumber.

IFileOpen

The IFileOpen interface provides functionality to customize the file open dialogue box. The IFileOpen interface can offer more advanced features compared to the Utility::ShowFileOpenDialog function. However, in most ProcSee applications the simple file open dialogue function is sufficient.

The IFileOpen interface provides functionality like, multi file selections, show read only files, validation of file names, checking file existence and path existence etc.

The interface IFileOpen cannot be created as a stand-alone component or control. The member function Utility::FileOpen creates and returns a pointer to an interface of type IFileOpen.

The example that follows demonstrates how to create an IFileOpen object, initialize it, display it, and finally return the selected file.

```
char* GetSelectedFile()
{
  ProcSeeControls.IFileOpen* fo;
  fo = (ProcSeeControls.IFileOpen *)UIUtility.di.FileOpen();

  fo->Title = "ProcSee Open";
  fo->Filter = "Config Files (*.Tdoc)|*.Tdoc";
  fo->DefaultExt = "Tdoc";

  if ( fo->ShowDialog() == ProcSeeControls.DialogResult.OK )
    return fo->FileName;
  else
    return 0;
}
```

Property	Description
AddExtension	Controls whether the file dialog adds an extension to the file name or not.
CheckFileExists	Whether a warning message is issued if the specified file does not exist.
CheckPathExists	Whether a warning message is issued if the specified path does not exist.
DefaultExt	Specifies the name of the default file name extension.
DereferenceLinks	Whether to dereference links or not.
FileName	The name of the selected file.
Filter	Controls the type of files to view in the file dialog.
FilterIndex	Specifies the index to the default file extension.
InitialDirectory	The name of the initial directory to browse.
MultiSelect	Whether multi file selection mode is enabled or not.
ReadOnlyChecked	Whether the read only check box is checked or unchecked.
RestoreDirectory	Whether to restore the directory before closing or not.
ShowReadOnly	Whether to show the read only check box in the file dialog or not.
Title	The title to display in the caption bar of the file dialog.
ValidateNames	Whether the file dialog only accepts Win32 file names or not.

Function	Description
FileNames	This function returns the file names selected in the file dialog.
Reset	This function resets the properties to its default values.
ShowDialog	This function displays the file open dialog box.

P r o p e r t i e s :

int IFileOpen::AddExtension

This boolean property controls whether the file open dialogue automatically adds an extension to the file name if the user omits it or not.

The default value of this property is true.

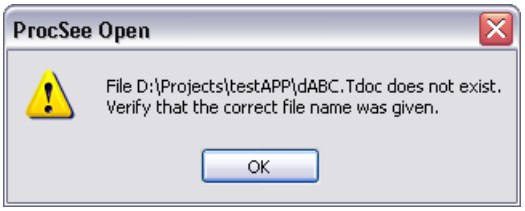
int IFileOpen::CheckFileExists

If this boolean property value is true a warning message is issued if the file specified does not exist.

The default value of this property is true.

Example

The following dialog will appear on the screen if the property CheckFileExists is true and the specified file does not exit.



int IFileOpen::CheckPathExists

If this boolean property value is true a warning message is issued if the file specified contains a path that does not exit.

The default value of this property is true.

Example

char* IFileOpen::DefaultExt

This property sets or gets the default file name extension.

The default value of this property is “”.

int IFileOpen::DereferenceLinks

This property indicates whether the file dialogue returns the location of the file referenced by the shortcut or it returns the location of the shortcut.

The default value of this property is true.

char* IFileOpen::FileName

This property sets or gets the name of the filename selected in the file selection dialogue.

char* IFileOpen::Filter

The property Filter controls what type of files to view in the file open dialog. Several file extensions can be specified separated by the vertical bar '|'. Each filter contains a description of the filter, followed by the vertical bar and the filter pattern. The strings for the different file types to view in a filter pattern, are separated by the semicolon.

Example

The following is an example of a filter string, which lists all Tdoc and plib files.

```
fo->Filter = "ProcSee text files (*.Tdoc;*.plib)|*.Tdoc;*.plib".
```

To view all files use the filter pattern,

```
fo->Filter = "All files (*.*)|*.*".
```

The following is an example of a filter, which lists mage files, using three patterns, plus all files. Supported file types are BMP, TIFF, TIF, PNG.

```
fo->Filter = "Bitmap (*.bmp)|*.bmp|
              Tiff(*.tiff;*.tif)|*.tiff;*.tif|
              Ping (*.png)|*.png|
              All files (*.*)|*.*"
```

See Also

IFileOpen::FilterIndex

int IFileOpen::FilterIndex

The property FilterIndex is a zero based index into the property Filter which specifies the default file extension(s).

The default value of this property is 0.

See Also

IFileOpen::Filter

char* IFileOpen::InitialDirectory

The property InitialDirectory specifies the initial location of the directory to browse when opening the file open dialog box.

int IFileOpen::MultiSelect

The boolean property MultiSelect controls whether multiple files can be selected in the file open dialog box or not. Use the function FileNames to get the selected file names from the file open dialog when this property is true.

The default value of this property is false.

See Also

IFileOpen::FileNames

---

int IFileOpen::ReadOnlyChecked

The property ReadOnlyChecked controls whether the read only check box is true or false in the file open dialog. This property only has an effect when the property ShowReadOnly is true.

The default value of this property is false.

See Also

IFileOpen::ShowReadOnly

---

int IFileOpen::RestoreDirectory

The boolean property RestoreDirectory controls whether the initial directory is restored before closing or not.

The default value of this property is false.

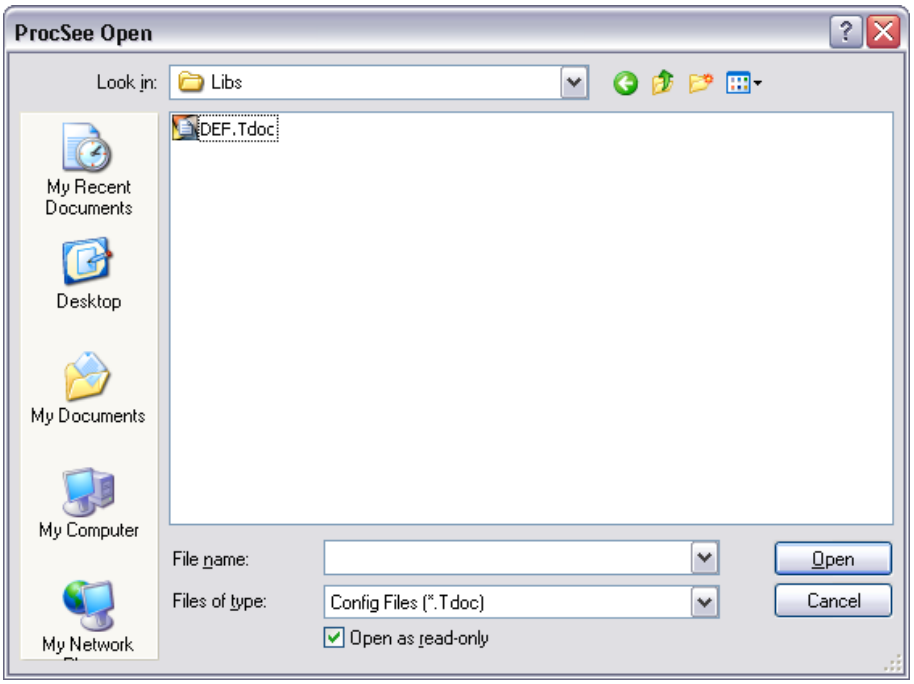
---

int IFileOpen::ShowReadOnly

The property ShowReadOnly displays a read only check box below the input fields in the file open dialog box. See the example below. To set the initial state of this check box, use the property ReadOnlyChecked.

The default value of this property is false.

Example



See Also

IFileOpen::ReadOnlyChecked

char* IFileOpen::Title

The property Title sets or gets the title in the caption bar of the file open dialog box.

int IFileOpen::ValidateNames

The property ValidateNames controls whether the file open dialog box only accepts Win32 file names or not.

The default value of this property is true.

F u n c t i o n s :

::IEnumAny* IFileOpen::FileNames()

This function returns a pointer to an IEnumAny interface which can be used to iterate through the filenames selected in the file open dialog box. This function is normally used when the property MultiSelect is true.

Return Value

This function returns a pointer to an IEnumAny interface containing the filenames selected in the file open dialog box.

Notice

This function actually returns an IEnumAny interface pointer, but due to some COM implementation issues, which are outside the scope of this manual, all interfaces are returned as IUnknown pointers. Therefore, an explicit cast operator is required as shown below.

```
::IEnumAny* ea = fo->FileNames();              // THIS LINE FAILS
::IEnumAny* ea = (::IEnumAny*)fo->FileNames(); // OK
```

Example

The following example demonstrates the use of the function FileNames to enumerate over the file names selected in the file open dialog box. This function returns a ::List object with all selected file names.

```
::List* GetFileNames()
{
   ProcSeeControls.IFileOpen* fo;
   fo = (ProcSeeControls.IFileOpen *)UIUtility.di.FileOpen();

   fo->MultiSelect = 1;
   fo->Title = "ProcSee Open";
   fo->Filter = "Config Files (*.Tdoc)|*.Tdoc";
   fo->DefaultExt = "Tdoc";

   ::List* fileNamesList = newList();
   if ( fo->ShowDialog() == ProcSeeControls.DialogResult.OK )
   {
     ::IEnumAny* fn = (::IEnumAny *)fo->FileNames();
     char* fileName;
     while ( fn->next( &fileName ) != 0 )
       fileNamesList->add( fileName );
   }

  return fileNamesList;
```

```
}
```

See Also

IFileOpen::FileName

---

void IFileOpen::Reset()

This function resets the file open dialog to its default values.

---

int IFileOpen::ShowDialog()

This function displays the file open dialog box.

Return Value

This function returns the value DialogResult.OK if the Open button was pressed, and DialogResult.Cancel if the Cancel button was pressed. The enum DialogResult is described on page 114.

IFileSave

The IFileSave interface provides functionality to customize the file save dialogue box. The IFileSave interface can offer more advanced features compared to the Utiltity::ShowFileSaveDialog function. However, in most ProcSee applications the simple file save dialogue function is sufficient.

The IFileSave interface provides functionality like overwrite prompt, validation of file names, checking file existence, path existence etc.

The interface IFileSave cannot be created as a stand-alone component or control. The member function Utility::FileSave creates and returns a pointer to an interface of type IFileSave.

The example that follows demonstrates how to create an IFileSave object, initialize it, display it, and finally return the selected file.

```
char* GetSelectedFile()
{
  ProcSeeControls.IFileSave* fs;
  fs = (ProcSeeControls.IFileSave *)UIUtility.di.FileSave();

  fs->Title = "ProcSee Save";
  fs->Filter = "Config Files (*.pcfg)|*.pcfg";
  fs->DefaultExt = "pcfg";

  if ( fs->ShowDialog() == ProcSeeControls.DialogResult.OK )
    return fs->FileName;
  else
    return 0;
}
```

Property	Description
AddExtension	Controls whether the file dialog adds an extension to the file name.
CheckFileExists	Whether a warning message is issued if the specified file does not exist.
CheckPathExists	Whether a warning message is issued if the specified path does not exist.
CreatePrompt	Whether to display a question dialog if the specified file does not exist.
DefaultExt	Specifies the name of the default file name extension.

DereferenceLinks	Whether to dereference links or not.
FileName	The name of the selected file.
Filter	Controls the type of files to view in the file save dialog.
FilterIndex	Specifies the index of the default file extension.
InitialDirectory	The name of the initial directory to browse.
OverwritePrompt	Whether to display a question dialog if the specified file exist or not.
RestoreDirectory	Whether to restore the directory when closing the dialog box.
Title	The title displayed in the caption bar of the file save dialog.
ValidateNames	Whether the file dialog only accepts Win32 file names or not.

Function	Description
Reset	This function resets the properties to its default values.
ShowDialog	This function displays the file save dialog box.

P r o p e r t i e s :

int IFileSave::AddExtension

Setting this boolean property AddExtension to true will automatically add the file name extension, specified with the property DefaultExt, to the file name.

The default value of this property is true.

See Also

IFileSave::DefaultExt

int IFileSave::CheckFileExists

If this boolean property value is true a warning message is issued if the specified file does not exist.

The default value of this property is false.

int IFileSave::CheckPathExists

If this boolean property value is true a warning message is issued if the specified path does not exist.

The default value of this property is true.

int IFileSave::CreatePrompt

The boolean property CreatePrompt controls whether the file save dialog prompts the user for permission to create the file if the file specified does not exist.

The default value of this property is false.

char* IFileSave::DefaultExt

This property sets or gets the default file name extension.

See Also

IFileSave::AddExtension

int IFileSave::DereferenceLinks

This property indicates whether the file save dialog box returns the location of the file referenced by the shortcut or it returns the location of the shortcut.

The default value of this property is true.

char* IFileSave::FileName

This property sets or gets the name of the filename selected in the file selection dialog.

char* IFileSave::Filter

The property Filter controls what type of files to view in the file save dialog. Several file extensions can be specified separated by the vertical bar '|'. Each filter contains a description of the filter, followed by the vertical bar and the filter pattern. The strings for the different file types to view in a filter pattern, are separated by the semicolon.

Example

The following is an example of a filter string, which lists all Tdoc and plib files.

```
fo->Filter = "ProcSee text files "(*.Tdoc;*.plib)|*.Tdoc;*.plib".
```

To view all files use the filter pattern,

```
fo->Filter = "All files (*.*)|*.*".
```

The following is an example of a filter, which lists mage files, using three patterns, plus all files. Supported file types are BMP, TIFF, TIF, PNG.

```
fo->Filter = "Bitmap (*.bmp)|*.bmp|
              Tiff(*.tiff;*.tif)|*.tiff;*.tif|
              Ping (*.png)|*.png|
              All files (*.*)|*.*"
```

See Also

IFileSave::FilterIndex

int IFileSave::FilterIndex

The property FilterIndex is a zero based index into the property Filter which specifies the default file extension(s).

The default value of this property is 0.

See Also

IFileSave::Filter

char* IFileSave::InitialDirectory

The property InitialDirectory specifies the initial location of the directory to browse when opening the file save dialog box.

int IFileSave::OverwritePrompt

The boolean property OverwritePrompt controls whether to display a question dialog when the selected file exists.

The default value of this property is true.

int IFileSave::RestoreDirectory

The boolean property RestoreDirectory controls whether the initial directory is restored before closing or not.

The default value of this property is false.

char* IFileSave::Title

The property Title sets or gets the title in the caption bar of the file save dialog box.

int IFileSave::ValidateNames

The property ValidateNames controls whether the file save dialog box only accepts Win32 file names or not.

The default value of this property is true.

int IFileSave::ValidateNames

This property controls whether the file save dialog box only accepts Win32 file names or not.

The default value of this property is true.

F u n c t i o n s :

void IFileSave::Reset()

This function resets the file save dialog box to its default values.

int IFileSave::ShowDialog()

This function displays the file save dialog box.

Return Value

This function returns the value DialogResult.OK if the Open button was pressed, and DialogResult.Cancel if the Cancel button was pressed. The enum DialogResult is described on page 114.

IMessageBox

The IMessageBox interface provides functionality to customize the message box. The IMessageBox interface can offer more advanced features compared to the dedicated message box functions Utility::ShowErrorMessageBox, Utility::ShowQuestionMessageBox, etc. However, in most ProcSee applications these simple message boxes are sufficient.

The IMessageBox interface provides functionality for specifying buttons, icons and default button.

The interface IMessageBox can not be created as a stand alone component or control. The member function Utility::MessageBox creates and returns a pointer to an interface of type IMessageBox.

The example that follows demonstrates how to create an IMessageBox object, initialize it and display it.

```
int DisplayMessage( char* title, char* message )
{
  ProcSeeControls.IMessageBox* mb;
  mb = (ProcSeeControls.IMessageBox *)UIUtility.di.MessageBox();

  mb->Caption       = title;
  mb->Buttons       = ProcSeeControls.MessageBoxButtons.YesNoCancel;
  mb->DefaultButton = ProcSeeControls.MessageDefaultButton.Button2;
  mb->Icon          = ProcSeeControls.MessageBoxIcon.Hand;
  mb->Text          = message;

  return mb->ShowDialog();
}
```

Property	Description
Buttons	Controls the number of buttons to display and the names of these buttons.
Caption	The message box title.
DefaultButton	Controls which of the buttons in the message box are the default button.
Icon	Controls the message button icon.
Text	The text to display in the message box.

Function	Description
Reset	This function resets the message box properties to its default values.
ShowDialog	This function displays the message box.

P r o p e r t i e s :

int IMessageBox::Buttons

The property Buttons controls which and the number of buttons to display in the message box. The constants values in the enum MessageBoxButtons define the legal value for this property.

char* IMessageBox::Caption

The property Caption sets the title in the message box.

int IMessageBox::DefaultButton

This property control, which button in the message box, is the default button. This property must be set to one of the constant values defined in the enum MessageBoxDefaultButton described on page 115.

int IMessageBox::Icon

This property controls the icon to display in the message box. This property must be set to one of the constant values defined in the enum MessageBoxIcon described on page 115.

char* IMessageBox::Text

This property sets or gets the text to display in the message box. To insert line breaks in the string use the escape sequence character '\n'.

Example

The following example demonstrates how to display a message on three separate lines in the message box.

```
…
mb->Text = "This text\nis displayed\non three separate lines";
mb->ShowDialog();
…
```

F u n c t i o n s :

void IMessageBox::Reset()

This function resets the properties used by the message box to their default values.

int IMessageBox::ShowDialog()

This function displays the message box.

Return Value

The return value from this method depends on the property Buttons. The legal retrun values are defined in the enum constant MessageBoxResult.

8

.NET Common Interfaces

This chapter describes the interfaces used by several .NET controls and .NET components.

The following interface types are documented:

- Collections

Refer to *ListBox* performance on page 20 for an example on using Collections.

Collections

A collection is a way of grouping and managing related objects. In its simplest form, objects can be added, modified and removed from a collection. A collection object can be used to improve the performance when adding many objects for instance to a ListBox control.

The following collection interfaces are available:

- ItemCollection

IItemCollection

The IItemCollection interface is a collection of character string items. It is implemented as a dynamic array, i.e. the array will automatically grow when the limit is reached.

An IItemCollection interface cannot be created as a stand-alone interface but is created and returned from methods in other interfaces, like IListBox, IComboBox, etc.

Property	Description
Capacity	The number of elements the collection can contain.
Count	Number of items in the collection.

Function	Description
AddCollection	Adds another IItemCollection to this collection.
AddItem	Adds a string item to the collection.
Clear	Removes all items from the collection.
Contains	Determines whether an item is within the collection or not.
GetItem	Returns the item at the specified index.
IndexOf	Returns the index of the specified string item in the collection.

InsertItem	Inserts an item at a specified index in the collection.
Remove	Removes an item in the collection with a specified name.
RemoveAt	Removes an item in the collection at a specified index.
Sort	Sorts the items in the collection in ascending order.

P r o p e r t i e s :

int IItemCollection::Capacity

This property controls the number of elements that the ItemCollection can contain. The collection will automatically be increased when exceeding the capacity.

int IItemCollection::Count

The read only property returns the actual number of items in the ItemCollection.

F u n c t i o n s :

int IItemCollection::AddCollection(::IItemCollection* collection)

This function adds the items in the IItemCollection passed as argument to this collection object. The items in the collection passed as argument will be put last in this collection.

Parameters

collection – The IItemCollection to add to this collection object.

Return Value

True is returned if the IItemCollection object passed as argument to this function was successfully added to this collection, otherwise false.

int IItemCollection::AddItem(char* theValue)

This function adds the character string passed as argument last in the collection.

Parameters

theValue – The character string to add to the collection.

Return Value

True is returned if the character string was successfully added to the collection, otherwise false.

See Also

IItemCollection::InsertItem

int IItemCollection::Clear()

This function clears all items from the IItemCollection object. The collection will be empty.

Return Value

Returns true if the collection was cleared, otherwise false.

See Also

IItemCollection::Remove, IItemCollection::RemoveAt

---

int IItemCollection::Contains(char* item)

This function determines whether the character string passed as argument is within the IItemCollection object or not.

Parameters

item – The name of the item to search for within the collection.

Return Value

This function returns true if the character string passed as argument was found in the collection, otherwise false.

---

char* IItemCollection::GetItem(int index)

This function returns the character string at the specified index in the collection.

Parameters

index – The index in the collection whose character string to return.

Return Value

This function returns the item at the specified index. The value 0 is returned if the index is outside the array collection bounds.

---

int IItemCollection::IndexOf(char* item)

This function returns the index of the string item to search for in the IItemCollection.

Parameters

item – The name of the item whose index in the collection to return.

Return Value

This function returns the index of the item in the collection. The value negative one, -1, is returned if the item is not found in the collection.

---

int IItemCollection::InsertItem(int index, char* theValue)

This function inserts a string item at the specified index in the IItemCollection.

Parameters

inex – The index in the collection where the item is inserted.

theValue – The name of the item to insert in the collection.

Return Value

This function returns true if the item was successfully inserted into the collection, otherwise false.

Example

The following example inserts an item first in the collection.

```
…
    IItemCollection* ic = …
…
    srintf( buffer, "%s_%d", … );
    ic->InsertItem( 0, buffer );
…
```

See Also

IItemCollection::AddItem

---

int IItemCollection::Remove(char* theItem)

This function removes the item in the IItemCollection with the specified name.

Parameters

theItem – Name of the item to remove from the collection.

Return Value

This function returns true if the item was successfully removed, otherwise false.

See Also

IItemCollection::Clear, IItemCollection::RemoveAt

---

int IItemCollection::RemoveAt(int index)

This function removes the item in the IItemCollection at the specified index.

Parameters

index – The index of the item to remove from the collection.

Return Value

This function returns true if the item was successfully removed, otherwise false.

See Also

IItemCollection::Remove, IItemCollection::Clear

---

int IItemCollection::Sort()

This function sorts the items in the IItemCollection in ascending order.

Return Value

This function returns true if the items in the collection was successfully sorted, otherwise false.

9

.NET Enums

This chapter describes the enums defined in the User Interface Controls library. The enums are used in method calls and properties where the legal value range is restricted. The following enums are defined:

Enum	Description
Appearance	The controls appearance.
BorderStyle	Defines the style of the border around the control.
CharacterCasing	Constants defining character casing or not.
CheckState	Check state used by the CheckBox control.
ComboBoxStyle	The style of the ComboBox.
ContentAlignment	Alignment of text within controls like Button, CheckBox, etc.
DialogResult	The result returned from a dialog box.
FlatStyle	Changes the visual appearance of the control.
HorizontalAlignment	The alignment of text within a text box.
MessageBoxButtons	Controls the buttons in a message box.
MessageBoxDefaultButton	Specifies the default button in a message box.
MessageBoxIcon	Specifies the icon to use in the message box.
Orientation	The orientation of some of the controls.
ScrollBars	Defines the scrollbars to display.
ScrollEventType	Defines the type of scrollbar event that occurred.
SelectionMode	The selection mode used by the ListBox control.
TickStyle	The tick style used by the TrackBar control.

Appearance

This enum controls the visual appearance of the control.

Enum	Value	Description
Button	1	The control appears as a button.
Normal	0	The control has the normal appearance.

BorderStyle

This enum controls the style of the border around the control.

Enum	Value	Description
None	0	No border around the control.
FixedSingle	1	A single fixed line is drawn around the control.

Fixed3D	2	A 3D style line is drawn around the control.

CharacterCasing

This enum controls whether character casing is active or not when using text input fields.

Enum	Value	Description
Normal	0	No character casing.
Upper	1	All characters will be converted to upper-case letters.
Lower	2	All characters will be converted to lower-case letters.

CheckState

This enum is used by the CheckBox to set and get the checked state of the conrol.

Enum	Value	Description
Unchecked	0	The check-box is unchecked.
Checked	1	The check-box is checked.
Indeterminate	2	The check-box is in an indeterminate state.

ComboBoxStyle

This enum determines the ComboBox visual style and functionality.

Enum	Value	Description
Simple	0	The combo-box does not have a drop down list.
DropDown	1	The combo-box displays a drop down list and an editable text input field.
DropDownList	2	The combo-box displays a drop down list.

ContentAlignment

The ContentAlignment enum defines constants used to align text within controls like Button, CheckBox and RadioButton.

Enum	Value	Description
TopLeft	1	The text is vertically aligned at the top, and horizontally to the left.
TopCenter	2	The text is vertically aligned at the top, and horizontally in the centre.
TopRight	4	The text is vertically aligned at the top, and horizontally to the right.
MiddleLeft	16	The text is vertically aligned in the middle, and horizontally to the left.
MiddleCenter	32	The text is vertically aligned in the middle, and horizontally in the centre.
MiddleRight	64	The text is vertically aligned in the middle, and horizontally to the right.
BottomLeft	256	The text is vertically aligned at the bottom, and horizontally to the left.
BottomCenter	512	The text is vertically aligned at the bottom, and horizontally in the centre.
BottomRight	1024	The text is vertically aligned at the bottom, and horizontally to the right.

DialogResult

This enum defines the values returned from the MessageBox.

Enum	Value	Description
None	0	No button was pressed.

OK	1	The OK button was pressed.
Cancel	2	The Cancel button was pressed.
Abort	3	The Abort button was pressed.
Retry	4	The Retry button was pressed.
Ignore	5	The Ignore button was pressed.
Yes	6	The Yes button was pressed.
No	7	The No button was pressed.

FlatStyle

The FlatStyle enum defines constants used to change the visual appearance of a control.

Enum	Value	Description
Flat	0	The control appears flat.
Popup	1	The control appears flat until the mouse is moved above the control, then it appears three-dimensional.
Standard	2	The control appears three dimensional

HorizontalAlignment

This enum controls the horizontal alignment of text in TextBox and MultiLineTextBox.

Enum	Value	Description
Left	0	The text is left aligned.
Right	1	The text is right aligned.
Center	2	The text is centered.

MessageBoxButtons

This enum controls the buttons to display in the general MessageBox dialog.

Enum	Value	Description
OK	0	Only add OK button.
OKCancel	1	Add OK and Cancel buttons.
AbortRetryIgnore	2	Add Abort, Retry and Ignore buttons.
YesNoCancel	3	Add Yes, No and Cancel buttons.
YesNo	4	Add Yes and No button.
RetryCancel	5	Add Retry and Cancel button.

MessageBoxDefaultButton

This enum controls which of the buttons is the default button in the general MessageBox dialog.

Enum	Value	Description
Button1	0	The first button in the button row is the default button.
Button2	256	The second button in the button row is the default button.
Button3	512	The third button in the button row is the default button.

MessageBoxIcon

The enum controls the icon displayed in the general MessageBoxdialog.

Enum	Value	Description
None	0	No icon.
Error	16	Error icon.
Hand	16	Hand icon.
Stop	16	Stop icon.

Question	32	Question icon.
Warning	48	Warning icon.
Exclamation	48	Exclamation icon.
Asterisk	64	Asterisk icon.
Information	64	Information icon.

Orientation

This enum controls the orientation of some of the controls in the User Interface Controls library.

Enum	Value	Description
Horizontal	0	The control is displayed horizontal.
Vertical	1	The control is displayed vertical.

ScrollBars

This enum controls which scrollbars to display.

Enum	Value	Description
None	0	No scrollbar is wanted.
Horizontal	1	Only horizontal scrollbar is wanted.
Vertical	2	Only vertical scrollbar is wanted.
Both	3	Both scrollbars are wanted.

ScrollBarEventType

This enum specifies the type of event that occurred in the scrollbar callback method OnValueChanged.

Enum	Value	Description
ValueChanged	-1	The Value property has changed programmatically.
SmallDecrement	0	A small decrement in the property Value.
SmallIncrement	1	A small increment in the property Value.
LargeDecrement	2	A large decrement in the property Value.
LargeIncrement	3	A large increment in the property Value.
ThumbPosition	4	The scroll thumb has been moved.
ThumbTrack	5	The scroll thumb is currently being moved.
First	6	The thumb is moved to the minimum position.
Last	7	The thumb is moved to the maximum position.
EndScroll	8	The scroll thumb has stopped moving.

SelectionMode

This enum is used by the ListBox control. The constants determines the selection mode.

Enum	Value	Description
None	0	Cannot select items in the list.
One	1	Only one item can be selected in the list.
MultiSimple	2	Possible to select several items in the list using the control key.
MultiExtended	3	Possible to select several items in the list using the mouse.

TickStyle

The enum TickStyle is used by the TrackBar control.

Enum	Value	Description
None	0	No ticks are wanted.
TopLeft	1	Tick to the left or to the top is wanted. It depends on the orientation.
BottomRight	2	Ticks to the right or to the bottom is wanted. It depends on the orientation.

Both	3	The ticks are wanted on both sides of the control.

Index

I

L

M

N

O

P

Q

R

S

T

U

V

W

Y