Presentation Server Basic+ Interface

The Presentation Server exposes an object-based interface (i.e. properties, methods and events) to Basic+ that can be accessed by the following core Stored Procedures:

- Exec_Method
- Forward_Event
- Get_EventStatus
- Get_Property
- Post_Event
- Send_Event
- Set_EventStatus
- Set_Property
- Set_Property_Only

In addition to this there are other supporting Stored Procedures that relate directly to the Presentation Server and allow other Basic+ routines to integrate smoothly with it:

- Create_Dialog
- Dialog_Box
- End_Dialog
- End_Window
- IsEventContext
- SetDebugger
- Start_MDIChild
- Start_Window
- Yield

Previous versions of OpenInsight also supported the following Stored Procedures that have been deprecated since version 10:

- Send_Message (replaced by Exec_Method)
- Utility (replaced by various SYSTEM and FILESYSTEM object methods)

All these functions (bar the deprecated ones) are described fully in this section.

Create_Dialog stored procedure

Description

Executes a specified OpenInsight window (form), as a modal or modeless dialog box.

In general terms, a dialog box is a form designed to collect information from the user, or to display a message that the user must acknowledge (commonly known as an "alert" window). In most cases a dialog box is launched in modal format, which means it exhibits the following behavior when used with Create_Dialog:

- It must have an owner form.
- The owner form is disabled for the lifetime of the dialog box (Optionally all other forms belonging to the owner form may be disabled).

(Note however that unlike a model dialog box executed via the Dialog_Box function, a modal dialog box executed with the Create_Dialog function *cannot return a value to its caller*.)

When executed in non-modal format it simple behaves as any other form that has an owner.

Syntax

```
RetVal = Create_Dialog( DialogID, OwnerID, Mode, CreateParam, Options )
```

Parameters

Name	Required	Description
DialogID	Yes	Name of the form to execute. Must be in upper-case.
OwnerID	Yes	Name of the owner form. Must be in upper-case. This form is disabled until the dialog box is destroyed if the Mode is FALSE$ (i.e. non-modal) Note that this may be the ID of a "docked" form, or a child form like an MDI child. In this case the parent "top-level" form will be disabled.
Mode	No	Set to TRUE$ to create a non-modal dialog,or FALSE$ (the default) to create a modal one.
CreateParam	No	Data to pass to the form's CREATE event. This data is passed as the "CreateParam" argument when the CREATE event is triggered. It must *not* contain any @Rm system delimiter characters.

Options	No	A dynamic array of extra options for launching the form:
		
		`<1> If TRUE$ then disable ALL other forms owned by the OwnerID form, not just the OwnerID form itself when modal.`
		
		`<2> GETPARENTFORM override. The Create_Dialog stored procedure uses the WINDOW object GETPARENTFORM method internally. This option allows it to be tweaked as desired.`

Returns

The Instance ID of the newly executed dialog box is returned if successful. Null is returned if the dialog box fails to start. The instance ID is usually the same as the passed DialogID, but if the window is flagged as multi-instance then the PS can append a unique number (delimited with an "*" character) to the returned ID to ensure that there are no conflicts with existing windows.

Errors

Error information may be obtained via the Get_Status function.

Remarks

This function can be considered deprecated for launching modal dialog boxes. The Dialog_Box stored procedure should be used instead.

The End_Dialog subroutine or the WINDOW CLOSE method can be used to close a dialog box executed with Create_Dialog.

Example

```
// Example - launch a non-modal dialog

CurrVal  = Get_Property( CtrlEntID, "TEXT" )
DlgOpt   = ""

DialogID = Create_Dialog( "MY_DIALOG_BOX", @Window, TRUE$, CurrVal, DlgOpt )
```

See also

Dialog_Box stored procedure, End_Dialog stored procedure, Start_Window stored procedure, ENDDIALOG event, WINDOW CREATE event, WINDOW CLOSE method, WINDOW CLOSE event, WINDOW GETPARENTFORM property.

Dialog_Box stored procedure

Description

Executes a specified OpenInsight window (form), as a modal dialog box.

In general terms, a dialog box is a form designed to collect information from the user, or to display a message that the user must acknowledge (commonly known as an "alert" form). In the majority of cases a dialog box is launched in modal format, which means it exhibits the following behavior:

- It must have an owner form.
- The owner form is disabled for the lifetime of the dialog box (Optionally all other windows belonging to the owner form may be disabled).
- The dialog box can return information direct to the calling program (Synchronous Mode) or via the ENDDIALOG event (Asynchronous Mode).

Syntax

```
RetVal = Dialog_Box( DialogID, OwnerID, CreateParam, Options, AsyncParams )
```

Parameters

Name	Required	Description
DialogID	Yes	Name of the form to execute. Must be in upper-case.
OwnerID	Yes	Name of the owner form. Must be in upper-case. This form is disabled until the dialog box is destroyed. Note that this may be the ID of a "docked" form, or a child form like an MDI child. In this case the parent "top-level" form will be disabled.
CreateParam	No	Data to pass to the form's CREATE event. This data is passed as the "CreateParam" argument when the CREATE event is triggered. It must *not* contain any @Rm system delimiter characters.

Options	No	A dynamic array of extra options for launching the form:
		
		`<1> If TRUE$ then disable ALL other forms owned by the OwnerID form, not just the OwnerID form itself.`
		
		`<2> GETPARENTFORM override. The Dialog_Box stored procedure uses the WINDOW object GETPARENTFORM method internally. This option allows it to be tweaked as desired.`

AsyncParams	No	A dynamic array of options that control Asynchronous mode. When executed in this mode the returned data is not passed directly back to the calling stored procedure – rather it is passed back via the OwnerID's ENDDIALOG event.
		
		`<1> If TRUE$ then the dialog will be executed in Asynchronous mode.`
		
		`<2> Contains a string argument passed back to the OwnerID's ENDDIALOG event. Useful for identifying the returned data.`

Returns

If the dialog box is executed in synchronous mode the return value is the data passed back from an End_Dialog call.

If the dialog is executed in asynchronous mode the return value will be the instance ID of the created dialog box. In this case the data passed back from an End_Dialog call will passed as an argument to the owner form's ENDDIALOG event.

If an error occurs the return value will be null.

Errors

Error information may be obtained via the Get_Status function.

Remarks

In previous version of OpenInsight all modal dialogs were executed in Synchronous mode, which made programming them very easy but imposed some constraints on the system if multiple nested Dialog_Box calls were made - a scenario that is quite possible in a modern multi-monitor, multi-window system. When this happens, the system must “stack” these calls in the order that they are made, which may not always match the order in which the user dismisses them, and this can lead to some confusion with the user interface.

To alleviate this OpenInsight 10 has introduced Asynchronous mode, which means that calling programs never wait for a direct answer from the dialog box so there is no need to form a stack of waiting programs. Instead any returned data is passed to the caller’s ENDDIALOG event when triggered by an End_Dialog call. The downside of this mode is that it is more difficult to program as the flow of the code must be broken up into a calling and receiving phase, rather than running in a linear sequence. Despite this drawback Asynchronous mode is the preferred method and will result in a better runtime performance when implemented.

Example

For the purposes of this example we assume that we are going to launch a simple form called "MY_DIALOG_BOX" using the Dialog_Box function. The form contains a single EDITLINE control called "EDL_NAME", and a button called "BTN_OK". When BTN_OK is clicked it will get the text from EDL_NAME and return it to the owner with an End_Dialog call like so:

```
// CLICK event script for MY_DIALOG_BOX.BTN_OK - Get the data the user entered
Name = Get_Property( @Window : ".EDL_NAME", "TEXT" )

// Return it to the owner form
Call End_Dialog( @Window, Name )
```

Synchronous Mode Example

To launch MY_DIALOG_BOX in synchronous fashion, do the following from an event on the owner form:

```
// Launches MY_DIALOG_BOX as a modal Dialog_Box in synchronous fashion passing
// it the contents of a variable called CurrName as the CreateParam.
NewName = Dialog_Box( "MY_DIALOG_BOX", @Window, CurrName )

If BLen( NewName ) Then
   // The user entered a new name so process it
   Call Do_Something_With_This_Name( NewName )
End
```

In this mode the calling program will halt at the Dialog_Box call and wait for the user to close it, after which the text returned from the End_Dialog call on MY_DIALOG_BOX.BTN_OK will be placed in the NewName variable for subsequent processing.

Asynchronous Mode Example

To launch MY_DIALOG_BOX in asynchronous fashion, do the following from an event on the owner form:

```
// Launches MY_DIALOG_BOX as a modal Dialog_Box in asynchronous fashion
// passing it the contents of a variable called CurrName as the CreateParam.
AsyncParams = ""
AsyncParams<1> = TRUE$       ; // Async mode
AsyncParams<2> = "GetName"   ; // Optional "AsyncID" param for ENDDIALOG

// This code does not halt here - anything the user selects in the dialog
// will be passed back in the ENDDIALOG event
DlgID = Dialog_Box( "MY_DIALOG_BOX", @Window, CurrName, "", AsyncParams )
```

In this mode the calling program will NOT halt at the Dialog_Box call and wait for the user to close it. Instead any data returned from the End_Dialog call on MY_DIALOG_BOX.BTN_OK will be passed as an argument to the calling window's ENDDIALOG event.

The ENDDIALOG event would look something like this:

```
 Function ENDDIALOG( CtrlEntID, CtrlClassID, DialogID, DialogValue, AsyncID )

  // This is an ENDDIALOG event that will be triggered by the End_Dialog
  // call on MY_DIALOG_BOX.BTN_OK when launched in asynchronous mode,
  //
  // ENDDIALOG is passed three parameters:
  //
  //   DialogID
  //   DialogValue
  //   AsyncID

  Begin Case
     Case ( DialogID = "MY_DIALOG_BOX" )
        // This is optional but we can check AsyncID if we wanted to have
        //  more fine grain control over how this event is processed.
        If ( AsyncID == "GetName" ) Then
           Call Do_Something_With_This_Name( NewName )
        End
  End Case

Return FALSE$
```

See also

Create_Dialog stored procedure, End_Dialog stored procedure, Start_Window stored procedure, WINDOW GETPARENTFORM property, WINDOW CLOSE method, WINDOW SHOWDIALOH method, WINDOW CREATE event, WINDOW ENDDIALOG event.

End_Dialog stored procedure

Description

This stored procedure closes a modal dialog box and returns a value back to the caller.

Syntax

```
Call End_Dialog( DialogID, RetVal )
```

Parameters

Name	Required	Description
DialogID	Yes	Name of the dialog box to close. Must be in upper-case.
RetVal	No	Value to return to the caller.

Returns

N/a (subroutine).

Errors

N/a.

Remarks

This stored procedure is used to close a modal dialog box and return a value to the caller. If the dialog box was launched in synchronous mode the return value is returned directly from the originating Dialog_Box call, otherwise the return value is passed to the caller's ENDDIALOG event instead. See the description of the Dialog_Box stored procedure above for more details.

This stored procedure should only be used with modal dialog boxes that have been executed via a call to Dialog_Box. It can be used with non-modal dialog boxes but there is little point as a value cannot be returned to a caller.

Note that The WINDOW CLOSE method can also be used with a modal dialog – it is the equivalent of calling End_Dialog and passing a null RetVal parameter.

Example

See the Dialog_Box function above for an example of how to use End_Dialog.

See also
WINDOW CLOSE method Create_Dialog stored procedure, Dialog_Box stored procedure.

End_Window stored procedure

Description

This stored procedure destroys a form and sets focus to the indicated object or to the form's owner (if any).

Syntax

```
Call End_Window( FormID, FocusID )
```

Parameters

Name	Required	Description
FormID	Yes	Name of the form to destroy. Must be in upper-case.
FocusID	No	Name of a PS GUI object to move the focus to when the form is destroyed. Must be in upper-case. If this parameter is not specified the focus is set to the form's owner (if possible).

Returns

N/a (subroutine).

Errors

N/a.

Remarks

Using this stored procedure means that the specified form is destroyed unconditionally along with its "Window Common Area" (i.e. the Basic+ common variables that contain the form's synthetic property and semantic data). No attempt is made to release any locks or perform any validation checks.

The preferred way of destroying a form is to call its CLOSE method (which in turn calls End_Window internally). End_Window should only be used as last resort when the CLOSE method fails.

Example

```
// Example - Terminate a form unconditionally and look to see if it's
// handle is still valid.

Call End_Window( FormID )

If Get_Property( FormID, "HANDLE" ) Else
   // Handle invalid - The form was destroyed
End
```

See also

WINDOW CLOSE method, SYSTEM DESTROY method, HANDLE property, Start_Window stored procedure.

Exec_Method stored procedure

Description

Executes a method for a specified object.

Syntax

```
Value = Exec_Method( Object, Method, Param1, Param2, Param3, … Param12 )
```

Parameters

Name	Required	Description
Object	Yes	Identifier of the object to access. Must be in upper-case.
Method	Yes	Name of the method to execute. Must be in upper-case.
Param1 - Param12	No	Parameters to be passed to the method. Up to 12 parameters may be used.

Returns

The return value is method dependent. If an invalid object or method name is specified an empty string (null) is returned. No error condition is flagged.

Errors

N/a.

Remarks

Methods can also be invoked from an object's EXECMETHOD property.

Example

```
// Call the SYSTEM DESTROY method to destroy  the EDL_NAME control
Call Exec_Method( "SYSTEM", "DESTROY", @Window : ".EDL_NAME" )

// Call the LISTBOX INSERT method to append an item to a list box called LST_ITEMS
Call Exec_Method( @Window : ".LST_ITEMS", "INSERT", -1, "Item Twelve" )
```

See also

EXECMETHOD property.

Forward_Event stored procedure

Description

Suspends execution of the current event handler and transfers to the next handler in the event chain.

Syntax

```
Call Forward_Event(Param1, Param2, Param3, …, Param20 )
```

Parameters

Name	Required	Description
Param1 – Param20	No	Event-specific parameters to be forwarded to the next handler. Note that all events are passed two common parameters – CtrlEntID and CtrlClassID. These should *not* be passed to Forward_Event, only the parameters that are specific to the event need to be passed.

Returns

N/a (subroutine).

Errors

Error information is returned in the same format as for Get_EventStatus.

Remarks

This stored procedure is normally called from an event script to hand control over to some lower level system code (e.g. during a READ event to perform the actual system READ process) before executing some post processing.

The Get_EventStatus stored procedure should be used to check the status of the forwarded event for any errors.

Note: You should always return FALSE$ from your event script if you have called Forward_Event, otherwise the next handler will be called again.

Example

```
// Example - A POSCHANGED event script on an EditTable that uses Forward_Event
//           to validate the cell contents first before going on to make any
//           subsequent changes.
//
// POSCHANGED is passed four arguments:
//
//   CtrlEntID   - ID of the control that triggered the event
//   CtrlClassID - Class of the control that triggered the event
//   NextColumn  - Column index of the cell that is now current
//   NextRow     - Row index of the cell that is now current

function PosChanged( CtrlEntID, CtrlClassID, NextColumn, NextRow )

   Declare Function Get_EventStatus
   $Insert RTI_SSP_Equates
   $Insert Logical

   evError = ""

   // Ensure we have a clean slate
   Call Set_EventStatus( SETSTAT_OK$ )

   // Forward the event to perform validation - the system-level POSCHANGED
   // handler does this.
   Call Forward_Event( NextColumn, NextRow )

   // Check for errors
   If Get_EventStatus( evError ) Then
      // We failed validation so do nothing here - the focus will have been
      // moved back to the offending cell
      Null
   End Else
      // We passed - so do some post processing ...
      Call Set_Property_Only( @Window, "TEXT", NextColumn : " - " : NextRow )
   End

Return FALSE$
```

See also

Get_EventStatus stored procedure, Set_EventStatus stored procedure.

Get_EventStatus stored procedure

Description

Retrieves the status of a forwarded event.

Syntax

```
IsError = Get_EventStatus( EventStatus )
```

Parameters

Name	Required	Description
EventStatus	No	If passed then Get_EventStatus will return error information in this parameter.

Returns

If an error has been flagged for an event this function returns TRUE$, and places the error information in the EventStatus parameter (if passed). The EventStatus is an @Fm-delimited list of errors, each with the following format:

```
<0,1>          Event Error Code
<0,2> to <0,n> Error arguments associated with the code.
```

If no error has occurred then this function returns FALSE$.

Errors

N/a.

Remarks

Equates for event error codes can be found in the EVERRORS insert record. Error information may be translated into a textual format by using the RTI_ErrorText function (see example below).

Example

```
Declare Function Get_EventStatus, RTI_ErrorText
$Insert Logical

// Example - assume this is an Event Script for a CLEAR event on a
// window and we want to execute some post-CLEAR event code if the
// form was reset correctly.

// First we forward the CLEAR event, which should ensure that the
// data has been saved correctly if changed - To check if this is
// successful we use Get_EventStatus.

// Clear the form, passing the CLEAR event arguments:
//
//    bSaveKey,
//    bSuppressWarning
//    bMaintainFocus

Call Set_EventStatus( FALSE$ )
EvErrors = ""

Call Forward_Event( bSaveKey, bSuppressWarning, bMaintainFocus )

If Get_EventStatus( EvErrors ) Then
   // We have some error codes - make sure they are formatted
   // into text messages and display
   ErrCount = FieldCount( EvErrors, @Fm )
   ErrText  = ""
   For ErrIdx = 1 To ErrCount
      ErrText := RTI_ErrorText( "EV", EvErrors<ErrIdx> ) : @Tm
   Next
   ErrText[-1,1] = ""
   Call Msg( @Window, ErrText )
End Else
   // Do post-clear processing ...
End

// Because we've used Forward_Event we MUST return 0 from the
// event to stop the event chain from being repeated.
Return 0
```

See also

Common Object SENDEVENT method, Forward_Event stored procedure, RTI_ErrorText stored procedure, Set_EventStatus stored procedure.

Get_Property stored procedure

Description

Returns the current value of a specified property for an object.

Syntax

```
Value = Get_Property( Object, Property, Index )
```

Parameters

Name	Required	Description
Object	Yes	Identifier of the object to access. Must be in upper-case.
Property	Yes	Name of the property to access. Must be in upper-case.

Index	No	If the requested property supports indexing then this parameter specifies the index value(s). Indexed properties can have one or two dimensions – if the latter then the dimensions are @fm-delimited.
		`<1> First dimension`
		`<2> Second dimension`
		The actual index values themselves are specific to the property in question. In most cases they will be numeric, but some properties can handle quoted text values as well.

Returns

Returns the current property value. If an invalid object or property name is specified an empty string (null) is returned. No error condition is flagged.

Errors

N/a.

Remarks

Get_Property supports the property concatenation interface described in Appendix A – Concatenating Properties.

Example

```
// Get the text of the current window
WinText = Get_Property( @Window, "TEXT" )

// Get the current value of the RBN_GENDER radiobutton group on the current
// window
Gender = Get_Property( @Window : ".RBN_GENDER", "VALUE" )

// Get the text of the third tab for TAB_MAIN using a numeric index value
TabText = Get_Property( @Window : ".TAB_MAIN.TABS", "TEXT", 3 )

// Get the value of the "Events" tab for TAB_MAIN using a literal index value
TabText = Get_Property( @Window : ".TAB_MAIN.TABS", "VALUE", "Events" )
```

See also

Set_Property stored procedure, Set_Property_Only stored procedure, Appendix A – Concatenating Properties.

IsEventContext stored procedure

Description

Determines if the currently executing Basic+ code is being run in response to an event originating from the Presentation Server.

Syntax

```
IsEvent = IsEventContext()
```

Parameters

N/a.

Returns

Returns TRUE$ if the currently executing code is being called in response to an event, or FALSE$ otherwise.

Errors

N/a.

Remarks

Basic+ code can be executed from a variety of sources outside of the Presentation Server, such as from a web-server via OECGI, from a .NET client app via NetOI and so on, i.e. they can be executed in a different *context*.

When sharing code that is called from these different sources it is sometimes necessary to know this context so that the programs behave appropriately - some common functions, like Msg and Popup, will only operate properly when called from the Presentation Server. The IsEventContext stored procedure can be used to determine this and protect context-sensitive code sections.

Example

```
Declare Function IsEventContext
$Insert Logical

If IsEventContext() Then
   // Use the Msg() function to display an error message
   Call Msg( @Window, ErrorText )
End Else
   // report the error via other means
End
```

See also

N/a.

Post_Event stored procedure

Description

Posts an event onto the event queue. The posted event will not be executed until the current event chain is completed or the Yield stored procedure is called.

Syntax

```
SuccessFlag = Post_Event( Object, Event, Param1, Param2, Param3, …, Param20 )
```

Parameters

Name	Required	Description
Object	Yes	Identifier of the object to access. Must be in upper-case.
Event	Yes	Name of the event to execute. Must be in upper-case.
Param1 – Param20	No	Parameters to be passed to the event. Up to 20 parameters may be used.

Returns

A boolean value – TRUE$ if the event was posted successfully, or FALSE$ otherwise.

Errors

N/a.

Remarks

N/a.

Example

```
$Insert Logical

// Post a CLICK event to the BTN_OK button
Call Post_Event( @Window : ".BTN_OK", "CLICK" )

// Post a CLEAR event to the current window, clearing the key prompt
// and ignoring SAVEWARN warnings
Call Post_Event( @Window, "CLEAR", FALSE$, TRUE$ )
```

See also

Send_Event stored procedure, Yield stored procedure.

Send_Event stored procedure

Description

Executes an event for the specified object. The event is executed immediately.

Syntax

```
EventStatus = Send_Event( Object, Event, Param1, Param2, Param3, …, Param20 )
```

Parameters

Name	Required	Description
Object	Yes	Identifier of the object to access. Must be in upper-case.
Event	Yes	Name of the event to execute. Must be in upper-case.
Param1 – Param20	No	Parameters to be passed to the event. Up to 20 parameters may be used.

Returns

Null if the event was executed immediately, otherwise the event error code is returned, as described in the Get_EventStatus stored procedure.

Errors

Error information is returned in the same format as for Get_EventStatus.

Remarks

N/a.

Example

```
Declare Function Send_Event, RTI_ErrorText
$Insert Logical

// Example - Send a CLEAR event to the current window, clearing the key prompt and
// ignoring any changed data.  If there are any errors then report them to the user.
EvErrors = Send_Event( @Window, "CLEAR", FALSE$, TRUE$ )

If Len( EvErrors ) Then
   // We have some error codes - make sure they are formatted
   // into text messages and display
   ErrCount = FieldCount( EvErrors, @Fm )
   ErrText  = ""
   For ErrIdx = 1 To ErrCount
      ErrText := RTI_ErrorText( "EV", EvErrors<ErrIdx> ) : @Tm
   Next
   ErrText[-1,1] = ""
   Call Msg( @Window, ErrText )
End Else
   // Do post-clear processing ...
End
```

See also

Common Object SENDEVENT method, Common Object POSTEVENT method, Post_Event stored procedure, Get_EventStatus stored procedure, Set_ stored procedure function.

Set_EventStatus stored procedure

Description

Sets the status of the currently executing event.

Syntax

```
Call Get_EventStatus( Status, ErrorCode, ErrorArgs )
```

Parameters

Name	Required	Description

Status	Yes	Status of the event. Can be one of the following:
		`"0"  : Success, no error`
		`"1"  : Failure – error has occurred`
		`"-1" : Failure – append error to existing status`
		
		(Equated constants for these values may be found in the RTI_SSP_Equates insert record).

ErrorCode	Depends	Contains a code indicating the exact nature of the error. This is usually a number prefixed by the string "EV". Required if Status indicates a failure (1 or -1), ignored otherwise.
ErrorArgs	No	Contains an @Fm or @Vm delimited list of arguments associated with ErrorCode. They are used to replace numeric placeholder tokens in error text associated with the ErrorCode.

Returns

N/a (subroutinre).

Errors

N/a.

Remarks

Equates for event error codes can be found in the EVERRORS insert record. Error information may be translated into a textual format by using the RTI_ErrorText stored procedure.

In some circumstances the Event Status is used to return data to a calling stored procedure and does *NOT* actually indicate an error condition. This can happen for the following "EV" ErrorCodes:

- "EV200": Used to return a result to the Presentation Server from a synchronous WINMSG event.
- "EV999": Used to return a value directly from Send_Event to a calling stored procedure.

Example

```
  $Insert EvErrors
  $Insert RTI_SSP_Equates

  // Example WRITE event handler - check to see if we have all the
  // data that we need and that it makes sense.

  OK       = TRUE$
  ValidErr = ""

  GoSub ValidateFormData; // (Assume this sets OK and ValidErr)

  If OK Then
     // All good - let the system write the data record
     Call Forward_Event()
  End Else
     // Failed - set an EventStatus to let the caller know that
     // the was a problem. We can use one of the predefined EV
     // errors or create our own.
     //
     // For this we'll use the EV_VALIDERR$ and also use the SYSMSG
     // event to display a message.
     Call Set_EventStatus( SETSTAT_ERR$, EV_VALIDERR$, "" )
     Call Send_Event( @Window, "VALIDERR", ValidErr )

  End

  // Because we called Forward_Event we stop the event chain
  // here by returning FALSE$

 Return FALSE$
```

See also

Forward_Event stored procedure, Get_EventStatus stored procedure.

Set_Property stored procedure

Description

Updates the value of a specified property for an object and returns the current value.

Syntax

```
OrigValue = Set_Property( Object, Property, NewValue, Index )
```

Parameters

Name	Required	Description
Object	Yes	Identifier of the object to access. Must be in upper-case.
Property	Yes	Name of the property to access. Must be in upper-case.
NewValue	Yes	New value for the property. Can be null.

Index	No	If the property supports indexing then this parameter specifies the index value(s). Indexed properties can have one or two dimensions – if the latter then the dimensions are @Fm-delimited.
		
		`<1> First dimension`
		`<2> Second dimension`
		
		The actual index value themselves are specific to the property in question. In most cases they will be numeric, but some properties can handle quoted text values as well.

Returns

Returns the current property value. If an invalid object or property name is specified then no update occurs and an empty string (null) is returned. No error condition is flagged.

Errors

N/a.

Remarks

Set_Property supports the property concatenation interface described in Appendix A – Concatenating Properties.

Example

```
// Set the caption of the current form
OrigText = Set_Property( @Window, "TEXT", NewText )

// Set the current value of the RBN_GENDER radiobutton group on the
// current window
OrigGender = Set_Property( @Window : ".RBN_GENDER", "VALUE", "M" )

// Set the text of the third tab for TAB_MAIN using a numeric index
OrigText = Set_Property( @Window : ".TAB_MAIN.TABS", "TEXT", "Events", 3 )

// Set the value of the "Events" tab for TAB_MAIN using a literal index
OrigText = Set_Property( @Window : ".TAB_MAIN.TABS", "VALUE", "E", "Events" )
```

See also

Get_Property stored procedure, Set_Property_Only stored procedure, Appendix A – Concatenating Properties.

Set_Property_Only stored procedure

Description

Updates the value of a specified property for an object without returning the original value.

Syntax

```
Call Set_Property( Object, Property, NewValue, Index )
```

Parameters

Name	Required	Description
Object	Yes	Identifier of the object to access. Must be in upper-case.
Property	Yes	Name of the property to access. Must be in upper-case.
NewValue	Yes	New value for the property. Can be null.

Index	No	If the property supports indexing then this parameter specifies the index value(s). Indexed properties can have one or two dimensions – if the latter then the dimensions are @fm-delimited.
		
		`<1> First dimension`
		`<2> Second dimension`
		
		The actual index value themselves are specific to the property in question. In most cases they will be numeric, but some properties can handle quoted text values as well.

Returns

N/a (subroutine).

Errors

N/a.

Remarks

One of the problems with the normal Set_Property function is that it also performs an implicit Get_Property operation to return the previous contents of the property to the caller. In many cases this information is simply discarded after the “Set” making the

"Get" call actually unnecessary, and this inefficiency usually goes unnoticed as the quantity of information retrieved during the "Get" is often small. However, in some cases the effect is noticeable: For example, setting the ARRAY property of an EditTable that contains several thousand rows to null (i.e. clearing it), will produce a noticeable delay while the ARRAY contents are accessed and returned. Using Set_Property_Only alleviates this issue, resulting in better performance.

Set_Property_Only supports the property concatenation interface described in Appendix A – Concatenating Properties.

If an invalid object or property name is specified no update occurs. No error condition is flagged.

Example

```
// Set the caption of the current form
Call Set_Property_Only( @Window, "TEXT", NewText )

// Set the current value of the RBN_GENDER RadioButton group on the
// current window
Call Set_Property_Only( @Window : ".RBN_GENDER", "VALUE", "M" )

// Set the text of the third tab for TAB_MAIN using a numeric index value
Call Set_Property_Only( @Window : ".TAB_MAIN.TABS", "TEXT", "Events", 3 )

// Set the value of the "Events" tab for TAB_MAIN using a literal index value
Call Set_Property_Only( @Window : ".TAB_MAIN.TABS", "VALUE", "E", "Events" )
```

See also

Get_Property stored procedure, Set_Property stored procedure, Appendix A – Concatenating Properties.

SetDebugger stored procedure

Description

Controls various debugging tools for use with the Presentation Server.

Syntax

```
RetVal = SetDebugger( ToolID, Param1, Param2, Param3 )
```

Parameters

Name	Required	Description

ToolID	Yes	Specifies the debugging tool to use. Can be one of the following strings:
		
		`"SPY"      : Turns the PS event spy on or off.`
		`"ASSERT"   : Turns Assertion messages on or off`
		`"SAVEWARN" : Turns SAVEWARN tracing on or off.`

Param1	Optional	This can be a boolean value to turn the specified tool on or off. It may be null to simple return the status of the tool.

Param2	Optional	Depends on the method:
		
		`"SPY"      : An @fm-delimited list of forms to trace,`
		`           : or leave null to trace all forms.`
		`"ASSERT"   : N/a`
		`"SAVEWARN" : N/a`

Param3	Optional	Depends on the method:
		
		`"SPY"      : Set to TRUE$ to send all output to the`
		`           : System Monitor rather than the Event Spy`
		`           : channel on the IDE Output panel.`
		`"ASSERT"   : N/a`
		`"SAVEWARN" : N/a`

Returns

Returns the status of the tool before any updates are made (TRUE$ if the tool is active, or FALSE$ otherwise).

Errors

N/a.

Remarks

SPY tool – This tool is used to monitor events triggered by the system. Both the event resolution process and the firing of the event itself can be displayed in the System Monitor or the IDE's "Event Spy" channel on the Output Panel (the default output option).

ASSERT tool – This tool turns assertion messages on or off.

SAVEWARN tool – This tool is used to monitor the status of the SAVEWARN property for forms in the system. When set to TRUE$ any updates to a form's SAVEWARN property will be displayed in the System Monitor along with the control whose data was modified.

Although the SetDebugger function can be used programmatically, it is normally used directly from the System Monitor as per the following examples:

Example

```
To turn on the Event Spy for all forms execute the following in the System Monitor:

    setdebugger spy 1

To turn on the Event Spy for specific forms execute the following in the System Monitor:

   setdebugger spy 1 [myfirstformname,mysecondformname]

To turn off the Event Spy execute the following in the System Monitor:

    setdebugger spy 0

To check the status of the Event Spy execute the following in the System Monitor:

    setdebugger spy

To turn off Assertion messages execute the following in the System Monitor:

    setdebugger assert 0

To turn on Savewarn Tracing execute the following in the System Monitor:

    setdebugger savewarn 1
```

See also

System Monitor, SYSTEMMONITOR object, WINDOW SAVEWARN property, Basic+ $Assert statement.

Start_MDIChild function

Description

Executes a specified OpenInsight window (form) as an "MDI Child" form. MDI Child form must have an "MDI frame" as a parent and appears in its MDI Client Area at runtime.

Syntax

```
InstanceID = Start_MDIChild( WindowID,
                             FrameID,
                             CreateParam,
                             Reserved,
                             Title,
                             AppearanceMode,
                             InitX,
                             InitY,
                             InitWinStruct )
```

Parameters

Name	Required	Description
WindowID	Yes	Name of the child form to execute. Must be in upper-case.
FrameID	Yes	Name of the MDI Frame form that "owns" the new child. Must be in upper-case.
CreateParam	No	Data to pass to the child form's CREATE event. This data is passed as the "CreateParam" argument when the CREATE event is triggered.
Reserved	No	N/a.
Title	No	The title to appear in the child form's title bar. Defaults to the title of the child form as it was designed.

AppearanceMode	No	Specifies how the child form should be displayed. Can be one of the following values:
		`0 : Displays in the same way as the currently`
		`  : active child (this is the default)`
		`1 : Normal`
		`2 : Minimized`
		`3 : Maximized`

InitX	No	The initial X position of the child in the parent frame's MDI Client area.

InitY	No	The initial Y position of the child in the parent frame's MDI Client area.
InitWinStruct	No	Contains the form structure to use to create the child, overriding the structure identified by the WindowID parameter.

Returns

The Instance ID of the newly created form is returned if successful. Null is returned if the form fails to start. The instance ID is usually the same as the passed WindowID, but if the form is flagged as multi-instance then the PS can append a unique number (delimited with an "*" character) to the returned ID to ensure that there are no conflicts with existing forms.

Errors

In the event of an error the Start_MDIChild stored procedure returns null. Error information may be obtained via the Get_Status stored procedure.

Remarks

The basic structure for a form object is described in the PS_EQUATES and PS_WINDOW_EQUATES insert records.

The form's CREATE event (if any) will be triggered before Start_MDIChild returns.

Example

```
// Example - Start a maximized MDI Child form in an MDI Frame, passing
//           an ID to load in the child's CREATE event.

Declare Function Start_MDIChild
$Insert MSWin_ShowWindow_Equates

CustID  = "A12345"
ChildID = Start_MDIChild( "CUSTOMER_ENTRY", "MAIN_MDI", CustID, "", "",    |
                          SW_SHOWMAXIMIZED$, "", "", "" )
```

See also

Start_Window stored procedure, SYSTEM CREATE method, MDIFRAME property, WINDOW object, WINDOW CREATE event.

Start_Window stored procedure

Description

Executes a specified OpenInsight window (form) or returns its structure as a dynamic array so it can be modified before execution.

Syntax

```
InstanceID = Start_Window( WindowID,
                           OwnerID,
                           CreateParam,
                           GetStructureFlag,
                           Reserved )
```

Parameters

Name	Required	Description
WindowID	Yes	Name of the form to execute. Must be in upper-case.
OwnerID	No	Name of the form that "owns" the new form. Must be in upper-case. Windows that have an owner form always appear in front of their owner and are minimized when the owner is. They are also automatically destroyed when the owner is destroyed.
CreateParam	No	Data to pass to the form's CREATE event. This data is passed as the "CreateParam" argument when the CREATE event is triggered.
GetStructureFlag	No	If TRUE$ then the form is not executed. Rather its structure is returned as a dynamic array.

Returns

If GetStructureFlag is FALSE$ or omitted then the Instance ID of the newly executed form is returned if successful. Null is returned if the form fails to start. The instance ID is usually the same as the passed WindowID, but if the form is flagged as multi-instance then the PS can append a unique number (delimited with an "*" character) to the returned ID to ensure that there are no conflicts with existing forms.

If GetStructureFlag is TRUE$ then a dynamic array containing the form structure (along with the structure for any child controls) is returned instead. This structure may be adjusted and then passed to the SYSTEM CREATE method to execute the form instead.

Errors

In the event of an error the Start_Window stored procedure returns null. Error information may be obtained via the Get_Status stored procedure.

Remarks

The basic structure for a form object is described in the PS_EQUATES and PS_WINDOW_EQUATES insert records.

The form's CREATE event (if any) will be triggered before Start_Window returns.

Do not use this stored procedure to create an MDI child form – use the Start_MDIChild stored procedure instead.

Example

```
// Start a standalone form on the desktop passing it a record
// ID to process in its CREATE event.
RowID      = "X12W"
InstanceID = Start_Window( "MYFORM", "", RowID, FALSE$ )
If BLen( InstanceID ) Then
   // The form executed successfully
End Else
   ErrorText = ""
   If Get_Status( ErrorText ) Then
      // Handle the error
   End
End
```

```
// Start a standalone form on the desktop with the current form
// as the owner
InstanceID = Start_Window( "MYFORM", @Window, "", FALSE$ )
If BLen( InstanceID ) Then
   // The form executed successfully
End Else
   ErrorText = ""
   If Get_Status( ErrorText ) Then
      // Handle the error
   End
End
```

```
// Get the structure of the MYFORM form, modify it, and then create
// it "manually"
FormStruct = Start_Window( "MYFORM", "", "", TRUE$ )
If BLen( FormStruct ) Then
   // We have the structure - process it as needed and then execute

   // ... processing ...

   // Now execute
   CreateVal = Exec_Method( "SYSTEM", "CREATE", FormStruct )

End Else
   ErrorText = ""
   If Get_Status( ErrorText ) Then
      // Handle the error
   End
End
```

See also

Dialog_Box stored procedure, End_Window stored procedure, Start_MDIChild stored procedure, SYSTEM CREATE method, WINDOW OWNER property, WINDOW object, WINDOW CREATE event.

Yield stored procedure

Description

Allows the Presentation Server to check and execute all pending messages in the Windows message queue, and its own event queue, returning when the queue is empty. This allows the system to remain responsive during a long running process.

Syntax

```
Call Yield( SaveVars )
```

Parameters

Name	Required	Description
SaveVars	No	If TRUE$ (1) then the following system variables will be saved and then restored after the Yield operation: • @Dict • @Record • @ID • @RecCount • @Rn_Counter The state of Select Cursor 0 will also be saved (via the Push_Select/Pop_Select functions)

Returns

N/a.

Errors

N/a.

Remarks

Calling the Yield stored procedure during a long-running process is important for several reasons:

1. It allows screen updates, so that changes to controls can be displayed.
2. It allows other events to fire so that the user interface remains responsive.
3. It allows Windows to determine that the process is not "dead" so it will refrain from displaying a "ghost" window with the "Not Responding" caption.

The Yield stored procedure is designed to be used from within "Event Context". If called outside of this context it calls the WinYield stored procedure instead.

Example

```
$Insert Logical

Ctr = 0
Eof = FALSE$

Loop
   ReadNext ID Else Eof = TRUE$
Until Eof

   Ctr += 1

   // Update the progress bar
   Call Set_Property_Only( @Window : ".PRB_GASGAUGE", "VALUE", Ctr )

   // Allow the progress bar to update and save the
   // select state while we do this
   Call Yield( TRUE$ )

   // Check that the window is still up because the
   // user could have closed it during the Yield()

While ( Get_Property( @Window, "HANDLE" ) )

   // Do processing etc ...

Repeat
```

See also

SYSTEM WINDOWGHOSTING property, SYSTEM PROCESSEVENTS method, SYSTEM PROCESSWINMSGS method, Get_Property stored procedure, Set_Property stored procedure, Appendix A – Concatenating Properties.