PL/SQL Developer Plug-In interface Documentation

Version 2.10 of PL/SQL Developer introduced a Plug-In interface. The purpose of this interface is easy external addition of new functionality to PL/SQL Developer. Plug-Ins should be used to add functions to PL/SQL Developer that are not very well suited as built-in functions. Reasons to build a Plug-In could be to add some company or product specific functions. You could also consider to build commercial Plug-Ins (no license fee required). We will distribute some Plug-Ins of our own on our web site (several interfaces to version control systems), we can also promote yours (commercial or not) if you wish.

A Plug-In is a DLL, so if you are using a programming language that can create DLL's, you can create PL/SQL Developer Plug-Ins. On startup PL/SQL Developer will check certain directories for *.dll files which will be loaded if certain key functions are available. If one or more Plug-Ins are found, the configuration menu item is enabled which allows an end-user to activate or de-activate Plug-Ins.

The interface is built in C++ style. This means that if you are using a non C++ language (like Delphi) you should make sure that you force all calls (export functions and callbacks) to the C++ calling convention. To prevent programming language incompatibilities we limited the number of different parameters to three, Boolean (32 bit), Integer (32 bit) and zero terminated strings.

If you should find a bug or if you have wishes for additional functions, just let us know and we will try to improve the interface. We'll make sure however that any modifications to the interface will be upward compatible.

Basic functions

There are ten functions that can be exported from the DLL. Three of these functions are required for PL/SQL Developer to recognize a DLL as a Plug-In. Below are the basic functions to create a functional Plug-In.

NOTE:
Starting in version 500, only the IdentifyPlugIn function is required as a necessary function for PL/SQL Developer to recognize the Plug-In.

Plug-In Primary functions	
IdentifyPlugIn	*C++* char* IdentifyPlugIn(int ID) *Delphi* function IdentifyPlugIn(ID: Integer): PChar This function receives a Plug-In ID from PL/SQL Developer and should return a description for the Plug-In. The returned description should be unique for your Plug-In and will be displayed in the Plug-In configuration dialog. The ID identifies your Plug-In and can be used in other callback functions.
CreateMenuItem	*C++* char* CreateMenuItem(int Index) *Delphi* function CreatMenuItem(Index: Integer): PChar This function will be called with an Index ranging from 1 to 99. For every Index you can return a string that creates a new menu-item in PL/SQL Developer.
OnMenuClick	*C++* void OnMenuClick(int Index) *Delphi* procedure OnMenuClick(Index: Integer) This function is called when a user selected a menu-item created with the CreateMenuItem function and the Index parameter has the value (1 to 99) it is related to.

A simple Delphi Plug-In built with these functions could look like this:

```
var   PlugInID: Integer;
const Desc = 'Test Plug-In';

function IdentifyPlugIn(ID: Integer): PChar;  cdecl;
begin
  PlugInID := ID;
  Result := Desc;
end;
```

```
function CreateMenuItem(Index: Integer): PChar;  cdecl;
begin
  Result := '';
  case Index of
    10 : Result := 'Tools / -';
    11 : Result := 'Tools / Say &Hello...';
    12 : Result := 'Tools / Say &Goodbye...';
  end;
end;

procedure OnMenuClick(Index: Integer);  cdecl;
begin
  case Index of
    11 : ShowMessage('Hello');
    12 : ShowMessage('Goodbye');
  end;
end;

exports
  IdentifyPlugIn,
  CreateMenuItem,
  OnMenuClick;
```

In this example a menu separator and two menu items will be created which will display a message when selected.

As mentioned, the CreatMenuItem function is called with Index values ranging from 1 to 99. In the example three values are returned for Index 10, 11 and 12. To create a menu simply return the menu structure where the menu items are separated by a slash. If, for example, you wanted to create a new menu item in PL/SQL Developers File menu, the return value could look like this:

```
Result := 'File / My menu item'
```

You can create a menu separator if you specify a – as menu item like this:

```
Result := 'File / -'
```

If you wanted add a menu that should appear in the File – Open submenu, you could return this:

```
Result := 'File / Open / My menu item'
```

Spaces around the slash are optional and you can add a & to create keyboard shortcuts, so the last example could also be:

```
Result := 'File/Open/&My menu item'
```

If a menu item does not exist, it will be created. This means that you can even create new main menu and submenu items.
The new items in the previous examples will all be created as the last item. This is not always acceptable, sometimes you want to create a new menu item in the middle of an existing menu. If you wanted to create a new save method, you probably want it near the existing PL/SQL Developer save menu items. You can insert a new menu item by first specifying an existing menu followed by a << or >> (to insert before or after), followed by your new menu:

```
Result := 'File / Save all >> &My save'
```

To return to the example, the three return values will result in three menu items at the end of the existing PL/SQL Developer Tools menu.

```
10 : Result := 'Tools / -';
11 : Result := 'Tools / Say &Hello...';
12 : Result := 'Tools / Say &Goodbye...';
```

Since a menu separator can not be selected, the OnMenuClick function only has to check for values 11 and 12, which will display a simple message dialog.

Event functions

You can build some more intelligence in your Plug-In with the following functions. These functions are events that get called when something changes in PL/SQL Developer. One important thing you can do with these is enable/disable the menu-item(s) your Plug-In created.

Plug-In Event functions	
OnCreate	*C++* void OnCreate() *Delphi* procedure OnCreate This function is called when the Plug-In is loaded into memory. You can use it to do some one-time initialization. PL/SQL Developer is not logged on yet and you can't use the callback functions, so you are limited in the things you can do.
OnActivate	*C++* void OnActivate() *Delphi* procedure OnActivate OnActivate gets called after OnCreate. However, when OnActivate is called PL/SQL Developer and the Plug-In are fully initialized. This function is also called when the Plug-In is enabled in the configuration dialog. A good point to enable/disable menus.
OnDeactivate ***Available in version 300***	*C++* void OnDeactivate() *Delphi* procedure OnDeactivate This is the counterpart of the OnActivate. It is called when the Plug-In is de-activated in the configuration dialog.
OnDestroy	*C++* void OnDestroy() *Delphi* procedure OnDestroy This is the counterpart of the OnCreate. You can dispose of anything you created in the OnCreate.
CanClose ***Available in version 700***	*C++* BOOL CanClose() *Delphi* function CanClose: Bool This will be called when PL/SQL Developer is about to close. If your PlugIn is not ready to close, you can show a message and return False.
AfterStart ***Available in version 710***	*C++* void AfterStart() *Delphi* procedure AfterStart Called after all Plug-Ins are loaded and PL/SQL Developer is finished starting.
OnBrowserChange	*C++* void OnBrowserChange() *Delphi* procedure OnBrowserChange If your Plug-In depends on a selected item in the Browser, you can use this function to enable/disable menu-items. This function is called on every change in the Browser. You can use the IDE_GetBrowserInfo callback function to determine if the selected item is of interest to you.
OnWindowChange	*C++* void OnWindowChange() *Delphi* procedure OnWindowChange This function is called if PL/SQL Developer child windows change focus. You can use the IDE_GetWindowType callback to determine the active child window type.
OnWindowCreate ***Available in version 502***	*C++* void OnWindowCreate(int WindowType) *Delphi* procedure OnWindowCreate(WindowType: Integer) This function is called directly after a new window is created.
OnWindowCreated ***Available in version 514***	*C++* void OnWindowCreated(int WindowType) *Delphi* procedure OnWindowCreated(WindowType: Integer) This function is called after a new window is created. The difference with the "Create" function is that the Window is now completely initialized.
OnWindowClose ***Available in version 502***	*C++* int OnWindowClose(int WindowType, BOOL Changed) *Delphi* function OnWindowClose(WindowType: Integer; Changed: BOOL): Integer

	This function allows you to take some action before a window is closed. You can influence the closing of the window with the following return values: 0 = Default behavior 1 = Ask the user for confirmation (like the contents was changed) 2 = Don't ask, allow to close without confirmation The Changed Boolean indicates the current status of the window.
BeforeExecuteWindow ***Available in version 714***	*C++* BOOL BeforeExecuteWindowe(int WindowType) *Delphi* function BeforeExecuteWindow(WindowType: Integer): Bool This function is called before a Window is executed. Nothing is actually executed yet, and you can cancel execution by returning false. When you do return false, please give some feedback to the user why execution was cancelled.
AfterExecuteWindow ***Available in version 714***	*C++* void AfterExecuteWindow (int WindowType, int Result) *Delphi* procedure AfterExecuteWindow(WindowType, Result: Integer) When execution is finished, this function is called. The return parameter will indicate how execution finished: 0 = Finished with error 1 = Finished with the option to continue (like "next page" in the SQL Window) 2 = Finished successfully
OnConnetionChange	*C++* void OnConnectionChange() *Delphi* procedure OnConnectionChange This function is called when the user logs on to a different database or logs off. You can use the IDE_Connected and IDE_GetConnectionInfo callback to get information about the current connection.
OnPopup ***Available in version 300***	*C++* void OnPopup(char *ObjectType, char *ObjectName) *Delphi* procedure OnPopup(ObjectType, ObjectName: PChar) This function is called when a context sensitive popup is about to be displayed. It gives you the opportunity to do something with the menus you have created with the IDE_CreatePopupMenuItem callback.
OnMainMenu ***Available in version 401***	*C++* void OnMainMenu(char *MenuName) *Delphi* procedure OnMainMenu(MenuName: PChar) This function is called when a main menu is selected (when it drops down). You can use this event to activate your Plug-In menu(s) if none of the other events are appropriate. The MenuName parameter is the name of the main menu item that was selected.
OnTemplate ***Available in version 702***	*C++* BOOL OnTemplate(char *Filename, char **Data) *Delphi* function OnTemplate(Filename: PChar; var Data: PChar): Bool This function is called before a template is executed. This gives you a chance to modify the contents in the Data parameter. If you return false, the template is cancelled.
OnFileLoaded ***Available in version 514***	*C++* void OnFileLoaded(int WindowType, int Mode) *Delphi* procedure OnFileLoaded(WindowType, Mode: Integer) Called after a file is loaded. The mode parameter can identify the following: 1: recovery file (from a crash) 2: backup file (normal file backup with a ~ extension)
OnFileSaved ***Available in version 514***	*C++* void OnFileSaved(int WindowType, int Mode) *Delphi* procedure OnFileSaved(WindowType, Mode: Integer) Called after a file is saved. The mode parameter can identify the following: 1: recovery file (from a crash) 2: backup file (normal file backup with a ~ extension)
About ***Available in version 400***	*C++* char* About() *Delphi* function About: PChar This function allows you to display an about dialog. You can decide to display a dialog yourself (in which case you should return an empty text) or just return the about text.

	In PL/SQL Developer 3.1 there is an about button in the Plug-In configuration dialog.
Configure ***Available in version 400***	*C++* void Configure() *Delphi* procedure Configure If the Plug-In has a configure dialog you could use this function to activate it. This will allow a user to configure your Plug-In using the configure button in the Plug-In configuration dialog.
CommandLine ***Available in version 513***	*C++* void CommandLine(int FeedbackHandle, char *Command, char *Params) *Delphi* procedure CommandLine(FeedbackHandle: Integer; Command, Params: PChar) You can use this function if you want the Plug-In to be able to accept commands from the command window. See IDE_CommandFeedback for how to return messages to the command window.
Plug-In naming functions	
PlugInName ***Available in version 700***	*C++* char* PlugInName () *Delphi* function PlugInName: PChar The PlugIn name (if defined) will be used for online updates, and as name for command window PlugIn commands. If you want your PlugIn to be handled by online updates, please contact support. If this function is not defined, the PlugInName will be the dll filename.
PlugInSubName ***Available in version 700***	*C++* char* PlugInSubName () *Delphi* function PlugInSubName: PChar The subname will be added to the PlugInName. Possible values are ‘Trial’ or ‘Beta’.
PlugInShortName ***Available in version 700***	*C++* char* PlugInShortName () *Delphi* function PlugInShortName: PChar The short name is specifically for command window PlugIn commands. This allows you to specify a name that can be entered quickly.
Plug-In External FileSystem functions	
RegisterFileSystem ***Available in version 400***	*C++* char* RegisterFileSystem() *Delphi* function RegisterFileSystem: PChar Use this function if you want your Plug-In to load/save files somewhere ‘external’. If you use this function you should return a description that identifies your filesystem (like FTP for the FTP Plug-in). See the chapter about External File Systems.
DirectFileLoad ***Available in version 400***	*C++* char* DirectFileLoad() *Delphi* function DirectFileLoad: function(var Tag, Filename: PChar; WindowType: Integer): PChar This function will get called when a file will be directly loaded without a file dialog. This is needed if a user selects a file from the recent used files list. The Parameters indicate the file that you have to load and the return value is the file data.
DirectFileSave ***Available in version 400***	*C++* BOOL DirectFileSave() *Delphi* function DirectFileSave(var Tag, Filename: PChar; Data: PChar; WindowType: Integer): Bool This function will be called when ‘File Save’ is selected (not ‘File Save As…). You should save the data as specified in the parameters and return True if everything was successful.
Plug-In Export functions	
RegisterExport ***Available in version 500***	*C++* char* RegisterExport() *Delphi* function RegisterExport: PChar Use this function if you want to add an export option for (result) grids. The name you return will be the name that is displayed in the popup menus (next to html, xml, …).

	See the chapter about adding export options.
ExportInit ***Available in version 500***	*C++* BOOL ExportInit() *Delphi* function ExportInit: Boolean First call after an export request. You can ask the user for a filename and/or initialize variables. Return False if you want to cancel the export.
ExportFinished ***Available in version 500***	*C++* void ExportFinished() *Delphi* procedure ExportFinished; The export has finished.
ExportPrepare ***Available in version 500***	*C++* BOOL ExportPrepare() *Delphi* function ExportPrepare: Boolean This function allows you to prepare for the actual data. All values received with Exportdata before this function is called are column headers, and all values received after ExportPrepare is data. The return value allows you to signal if the prepare was processed correctly.
ExportData ***Available in version 500***	*C++* BOOL ExportData(char *Value) *Delphi* function ExportData(Value: PChar): Boolean One cell of data, this can be the column description or the actual data.

If you need one or more of these functions, export them from the Plug-In DLL. When a function is exported, PL/SQL Developer will call it. All these functions are very straightforward, the description should give you enough information.

Callback functions

Callback functions are functions within PL/SQL Developer that you can use in your Plug-In. They need to be “activated” by the RegisterCallback function, so you need to export this function from your Plug-In DLL.

Plug-In Callback function	
RegisterCallback	*C++* void RegisterCallback(int Index, void *Addr) *Delphi* procedure RegisterCallback(Index: Integer; Addr: Pointer) There are several functions in PL/SQL Developer that you can use from your Plug-In. With this function you can get access to the callback functions you need. The Index is related to a specific callback function while the Addr parameter holds the address to this function.

If you want to use PL/SQL Developer callback functions you need to create a declaration of these functions and assign them an address via the RegisterCallback function. RegisterCallback is called for every callback function, identified by a unique index, and passes the corresponding function address. In Delphi this would look like this:

```
Var
  IDE_MenuState: procedure(ID, Index: Integer; Enabled: Bool); cdecl;
  IDE_Connected: function: Bool; cdecl;
  IDE_GetConnectionInfo: procedure(var Username, Password, Database: PChar); cdecl;
  IDE_GetBrowserInfo: procedure(var ObjectType, ObjectOwner, ObjectName: PChar); cdecl;

procedure RegisterCallback(Index: Integer; Addr: Pointer); cdecl;
begin
  case Index of
    10 : @IDE_MenuState := Addr;
    11 : @IDE_Connected := Addr;
    12 : @IDE_GetConnectionInfo := Addr;
    13 : @IDE_GetBrowserInfo := Addr;
  end;
end;
```

In C++ this would look like this:

```
void  (*IDE_MenuState)(int ID, int Index, BOOL Enabled);
BOOL  (*IDE_Connected)();
void  (*IDE_GetConnectionInfo)(char **Username, char **Password, char **Database);
void  (*IDE_GetBrowserInfo)(char **ObjectType, char **ObjectOwner, char **ObjectName);

void RegisterCallback(int Index, void *Addr)
{
  switch (Index)
  {
    case 10 :
      (void *)IDE_MenuState = Addr;
     break;
    case 11 :
      (void *)IDE_Connected = Addr;
     break;
    case 12 :
      (void *)IDE_GetConnectionInfo = Addr;
     break;
    case 13 :
      (void *)IDE_GetBrowserInfo = Addr;
     break;
  }
}
```

The previous example only defined four callback functions. Below you will find the complete list of all callback functions, with index, name and a brief description:

SYSTEM Info functions		
1	**SYS_Version**	*C++* int SYS_Version() *Delphi* function SYS_Version: Integer Returns the PL/SQL Developer main and subversion, for example 210 for version 2.1.0. This might be useful if you want to use functions that are not available in all versions.
2	**SYS_Registry**	*C++* char* SYS_Registry() *Delphi* function SYS_Registry: PChar Returns the registry root name of PL/SQL Developer in HKEY_CURRENT_USER (usually “Software\PL/SQL Developer”). If you want to save your settings in the registry, you can create a section within the PL/SQL Developer section. Note: In PL/SQL Developer 3.1, the registry section is moved to: (“Software\Allround Automations\PL/SQL Developer”)
3	**SYS_RootDir**	*C++* char* SYS_RootDir() *Delphi* function SYS_RootDir: PChar The directory where PL/SQL Developer is installed, for example “C:\Program Files\PLSQL Developer”.
4	**SYS_OracleHome**	*C++* char* SYS_OracleHome() *Delphi* function SYS_OracleHome: PChar The Oracle directory, for example “C:\Orawin95”
5	**SYS_OCIDLL** ***Available in version 300***	*C++* char* SYS_OCIDLL() *Delphi* function SYS_OCIDLL: PChar Returns the path of the OCI DLL that is used by PL/SQL Developer. If you want to initialize a new session, you might want to use this value if you want to make sure you’re using the same OCI version.
6	**SYS_OCI8Mode** ***Available in version 300***	*C++* BOOL* SYS_OCI8Mode() *Delphi* function SYS_OCI8Mode: Bool Returns True if PL/SQL Developer is currently connected in OCI8 Mode

		(Net8).
7	**SYS_XPStyle** ***Available in version 700***	*C++* BOOL* SYS_XPStyle() *Delphi* function SYS_XPStyle: Bool Returns if PL/SQL Developer is currently using the visual XP style.

8	**SYS_TNSNAMES**
	Available in version 700
		C++ char* SYS_TNSNAMES (char *Param)
		Delphi function SYS_TNSNAMES(Param: PChar): PChar
		
		If Param is empty, the function will return the full tnsnames filename.
		If Param has a value, the connection details of the alias as specified by Param is returned. If Param is *, the connection details of the current connection are returned). The return value can look like:
		`TEST =`
		`  (DESCRIPTION =`
		`    (ADDRESS_LIST =`
		`      (ADDRESS = (PROTOCOL = TCP)(HOST = p2800)(PORT = 1521))`
		`    )`
		`    (CONNECT_DATA =`
		`      (SERVER = DEDICATED)`
		`      (SERVICE_NAME = AAA)`
		`    )`
		`  )`

9	**SYS_DelphiVersion** ***Available in version 702***	*C++* int SYS_DelphiVersion() *Delphi* function SYS_DelphiVersion: Integer Returns the Delphi version used to build PL/SQL Developer. Only useful for very specific functions.
IDE functions		
10	**IDE_MenuState**	*C++* void IDE_MenuState(int ID, int Index, BOOL Enabled) *Delphi* procedure IDE_MenuState(ID, Index: Integer; Enabled: Bool) Use this function to enable or disable a menu. The ID is the Plug-In ID, which is given by the IdentifyPlugIn function. The Index is the menu index, which the menu was related to by the CreateMenuItem function. The Enabled boolean determines if the menu item is enabled or grayed.
11	**IDE_Connected**	*C++* BOOL IDE_Connected() *Delphi* function IDE_Connected: Bool Returns a boolean that indicates if PL/SQL Developer is currently connected to a database.
12	**IDE_GetConnectionInfo**	*C++* void IDE_GetConnectionInfo(char **Username, char **Password, char **Database) *Delphi* procedure IDE_GetConnectionInfo(var Username, Password, Database: PChar) Returns the username, password and database of the current connection.
13	**IDE_GetBrowserInfo**	*C++* void IDE_GetBrowserInfo(char **ObjectType, char **ObjectOwner, char **ObjectName); *Delphi* procedure IDE_GetBrowserInfo(var ObjectType, ObjectOwner, ObjectName: PChar) Returns information about the selected item in the Browser. If no item is selected, all items are empty.
14	**IDE_GetWindowType**	*C++* int IDE_GetWindowType() *Delphi* function IDE_GetWindowType: Integer Returns the type of the current window. 1 = SQL Window 2 = Test Window 3 = Procedure Window 4 = Command Window 5 = Plan Window

		6 = Report Window 0 = None of the above
15	**IDE_GetAppHandle**	*C++* int IDE_GetAppHandle() *Delphi* function IDE_GetAppHandle: Integer Returns the Application handle of PL/SQL Developer
16	**IDE_GetWindowHandle**	*C++* int IDE_GetWindowHandle() *Delphi* function IDE_GetWindowHandle: Integer Returns the handle of PL/SQL Developers main window
17	**IDE_GetClientHandle**	*C++* int IDE_GetClientHandle() *Delphi* function IDE_GetClientHandle: Integer Returns the handle of PL/SQL Developers client window
18	**IDE_GetChildHandle**	*C++* int IDE_GetChildHandle() *Delphi* function IDE_GetChildHandle: Integer Returns the handle of the active child form
19	**IDE_Refresh** ***Available in version 213***	*C++* void IDE_Refresh() *Delphi* procedure IDE_Refresh Resets the state of the menus, buttons and the active window. You can call this function if you made some changes that affect the state of a menu or window which are unnoticed by PL/SQL Developer.
20	**IDE_CreateWindow**	*C++* void IDE_CreateWindow(int WindowType, char *Text, BOOL Execute) *Delphi* procedure IDE_CreateWindow(WindowType: Integer; Text: PChar; Execute: Bool) Creates a new window. The Text parameter contains text that is placed in the window. If the Execute Boolean is true, the Window will be executed. WindowType can be one of the following values: 1 = SQL Window 2 = Test Window 3 = Procedure Window 4 = Command Window 5 = Plan Window 6 = Report Window ***Version 800 and higher*** 7 = HTML Window
21	**IDE_OpenFile**	*C++* BOOL IDE_OpenFile(int WindowType, char *Filename) *Delphi* function IDE_OpenFile(WindowType: Integer; Filename: PChar): Bool Creates a window of type WindowType and loads the specified file. WindowType can be one of the following values: 1 = SQL Window 2 = Test Window 3 = Procedure Window 4 = Command Window The function returns True if successful. ***Version 301 and higher*** If you pass 0 as WindowType, PL/SQL Developer will try to determine the actual WindowType on the extension of the filename. ***Version 800 and higher*** 5 = Plan Window 6 = Report Window 7 = HTML Window
22	**IDE_SaveFile**	*C++* BOOL IDE_SaveFile() *Delphi* function IDE_SaveFile: Bool

		This function saves the current window. It returns True if successful.
23	**IDE_Filename**	*C++* char* IDE_Filename() *Delphi* function IDE_Filename: PChar Return the filename of the current child window. *See also IDE_SetFilename()*
24	**IDE_CloseFile**	*C++* void IDE_CloseFile() *Delphi* procedure IDE_CloseFile Closes the current child window
25	**IDE_SetReadOnly**	*C++* void IDE_SetReadOnly(BOOL ReadOnly) *Delphi* procedure IDE_SetReadOnly(ReadOnly: Bool) Set the ReadOnly status of the current Window
26	**IDE_GetReadOnly** ***Available in version 213***	*C++* BOOL IDE_GetReadOnly *Delphi* function IDE_GetReadOnly: Bool Get the ReadOnly status of the current Window
27	**IDE_ ExecuteSQLReport** ***Available in version 300***	*C++* BOOL IDE_ExecuteSQLReport(char *SQL, Char *Title, BOOL: Updateable) *Delphi* function IDE_ExecuteSQLReport(SQL: PChar; Title: PChar; Updateable: Bool): Bool This function will execute a query (SQL parameter) and display the result in a 'result only' SQL Window. Title will be used as the window name and the Updateable parameter determines if the results are updateable.
28	**IDE_ReloadFile** ***Available in version 301***	*C++* BOOL IDE_ReloadFile *Delphi* function IDE_ReloadFile: Bool Forces the active child window to reload its file from disk. Note: In PL/SQL Developer 4 there will no longer be a warning message when modifications were made.
29	**IDE_SetFilename** ***Available in version 303***	*C++* void IDE_SetFilename(char *Filename) *Delphi* procedure IDE_SetFilename(Filename: PChar) Set the filename of the active child window. The filename should contain a valid path, but the file does not need to exist. The new filename will be used when the file is saved. If the Filename parameter is an empty string, the Window will behave as a new created Window.
30	**IDE_GetText**	*C++* char* IDE_GetText() *Delphi* function IDE_GetText: PChar Retrieves the text from the current child window.
31	**IDE_GetSelectedText**	*C++* char* IDE_GetSelectedText() *Delphi* function IDE_GetSelectedText: PChar Retrieves the selected text from the current child window.
32	**IDE_GetCursorWord**	*C++* char* IDE_GetCursorWord() *Delphi* function IDE_GetCursorWord: PChar Retrieves the word the cursor is on in the current child window.
33	**IDE_GetEditorHandle**	*C++* int IDE_GetEditorHandle() *Delphi* function IDE_GetEditorHandle: Integer Returns the handle of the editor of the current child window.
34	**IDE_SetText** ***Available in version 213***	*C++* BOOL IDE_SetText(char *Text) *Delphi* function IDE_SetText(Text: PChar): Bool Sets the text in the editor of current window. If this failed for some reason (ReadOnly?), the function returns false.

35	**IDE_SetStatusMessage** ***Available in version 213***	*C++* BOOL IDE_SetStatusMessage(char *Text) *Delphi* function IDE_SetStatusMessage(Text: PChar): Bool Places a message in the status bar of the current window, returns false if the window did not have a status bar.
36	**IDE_SetErrorPosition** ***Available in version 213***	*C++* BOOL IDE_SetErrorPosition(int Line, int Col) *Delphi* function IDE_SetErrorPosition(Line, Col: Integer): Bool Highlights the given line and places the cursor at the given position. This will only work when the active window is a procedure window, if not, the function returns false.
37	**IDE_ClearErrorPositions** ***Available in version 213***	*C++* void IDE_ClearErrorPositions() *Delphi* procedure IDE_ClearErrorPositions Resets the highlighted lines.
38	**IDE_GetCursorWordPosition** ***Available in version 400***	*C++* int IDE_GetCursorWordPosition() *Delphi* function IDE_GetCursorWordPosition: Integer This function returns the location of the cursor in the word after a call to IDE_GetCursorWord. Possible return values: 0: Unknown 1: Cursor was at start of word 2: Cursor was somewhere in the middle 3: Cursor was at the end
39	**IDE_Perform** ***Available in version 400***	*C++* BOOL IDE_Perform(int Param) *Delphi* function IDE_Perform(Param Integer): Bool This function allows you to perform a specific action as if the menu item as specified in Param was selected. The following values are supported: 1: Execute 2: Break 3: Kill 4: Commit 5: Rollback 6: Print
60	**IDE_GetCustomKeywords** ***Available in version 300***	*C++* char* IDE_GetCustomKeywords() *Delphi* function IDE_GetCustomKeywords: PChar Returns a list of all keywords as entered in the ‘custom keywords’ option in the Editor preference.
61	**IDE_SetCustomKeywords** ***Available in version 300***	*C++* void IDE_SetCustomKeywords(char *Keywords) *Delphi* procedure IDE_SetCustomKeywords(Keywords: PChar) Fills the custom keywords with the words in the Keywords parameter. Words should be separated by cr/lf. The currently used keywords will be overwritten.
62	**IDE_SetKeywords** ***Available in version 300***	*C++* void IDE_SetKeywords(int ID, int Style, char *Keywords) *Delphi* procedure IDE_SetKeywords(ID, Style: Integer; Keywords: PChar) Adds a number of keywords with a specific style. This function is more specific then IDE_SetCustomKeywords because this one can set multiple sets of keywords for different highlighting styles. ID should be the PlugIn ID as returned by the IdentifyPlugIn function. Style can be one of the following values: 10: Custom 11: Keywords 12: Comment 13: Strings 14: Numbers 15: Symbols Keywords is a cr/lf separated list of words. You can define one list per style.
63	**IDE_ActivateKeywords**	*C++* void IDE_ActivateKeywords()

	Available in version 300	*Delphi* procedure IDE_ActivateKeywords Activates the keywords as defined by the IDE_SetKeywords function.
64	**IDE_RefreshMenus** ***Available in version 300***	*C++* void IDE_RefreshMenus(int ID) *Delphi* procedure IDE_RefreshMenus(ID: Integer) When this function is called, all menus for this Plug-In are removed and CreateMenuItem will be called to build a new set of menus. This only makes sense if you supply a different set of menu-items.
65	**IDE_SetMenuName** ***Available in version 300***	*C++* void IDE_SetMenuName(int ID, int Index, char *Name) *Delphi* procedure IDE_SetMenuName(ID, Index: Integer; Name: PChar) This function allows you to rename a certain menu-item. ID is the Plug-In ID, Index is the Menu number and name is the new menu name.
66	**IDE_SetMenuCheck** ***Available in version 300***	*C++* void IDE_SetMenuCheck(int ID, int Index, BOOL Enabled) *Delphi* procedure IDE_SetMenuCheck(ID, Index: Integer; Enabled: Bool) You can display or remove a check mark for a menu-item.
67	**IDE_SetMenuVisible** ***Available in version 300***	*C++* void IDE_SetMenuVisible(int ID, int Index, BOOL Enabled) *Delphi* procedure IDE_SetMenuVisible(ID, Index: Integer; Enabled: Bool) With this function you can hide or show a specific menu. You can use this instead of IDE_MenuState.
68	**IDE_GetMenulayout** ***Available in version 300***	*C++* char* IDE_GetMenulayout() *Delphi* function IDE_GetMenulayout: PChar Returns a list of all standard PL/SQL Developer menu items. Items are separated by cr/lf and child menu level is indicated by a number of spaces. You can use this function to build an advanced user configuration dialog where the user could be able to select place where he wants to insert the Plug-In menus.
69	**IDE_CreatePopupItem** ***Available in version 300***	*C++* void* IDE_ CreatePopupItem(int ID, int Index, char *Name, char *ObjectType) *Delphi* procedure IDE_CreatePopupItem(ID, Index: Integer; Name, ObjectType: PChar) With this function you can add items to certain popup menus. The ID is the Plug-In ID and the index is the menu index. You can pass any number as the menu index, it can be an existing menu (as used by CreateMenuItem) or anything else. If the popup menu gets selected, OnMenuClick is called with the corresponding index. The Name is the menu name as it will be displayed. The ObjectType determines in which popup menus this item will be displayed. Some possible values are: ‘TABLE’, ‘VIEW’, ‘PACKAGE’, etc. ***Version 301 and higher*** If you pass one of the following values as ObjectType, you can add items to specific Windows. PROGRAMWINDOW SQLWINDOW TESTWINDOW COMMANDWINDOW ***Version 400 and higher*** You can add popup items to Object Browser items like Tables, Views, etc. by passing their name as ObjectType. ***Version 510 and higher***

		If you want to create popup menus for multiple selected items (of the same object type), you can add a + to the ObjectType parameter like ‘TABLE+’, ‘VIEW+’, etc. The OnMenuClick will be called for every selected item, and the GetPopupObject will return the correct details. ***Version 700 and higher*** Supports Popup for the Session Window with the SESSIONWINDOW ObjectType. (see also IDE_GetSessionValue) ***Version 712 and higher*** Supports Popup for result grids with SQLRESULT ***Version 800 and higher*** Supports Popup for file browser with FILE
70	**IDE_SetConnection** ***Available in version 301***	*C++* BOOL IDE_SetConnection(char *Username, char *Password, char *Database) *Delphi* function IDE_SetConnection(Username, Password, Database: PChar): Bool This function allows you to reconnect PL/SQL Developer as another user. The return value indicates if the connection was successful. The function will fail if there is a childwindow with an active query. Also see IDE_SetConnectionAs
71	**IDE_GetObjectInfo** ***Available in version 400***	*C++* int IDE_GetObjectInfo(char *AnObject, char **ObjectType, char **ObjectOwner, char **ObjectName, char **SubObject) *Delphi* procedure IDE_GetObjectInfo(AnObject: PChar; var ObjectType, ObjectOwner, ObjectName, SubObject: PChar) This function returns Oracle information about the item in the AnObject parameter. The SubObject returns the name of the procedure if the Object is a packaged procedure.
72	**IDE_GetBrowserItems** ***Available in version 400***	*C++* char* IDE_GetBrowserItems(char *Node, BOOL GetItems) *Delphi* function IDE_GetBrowserItems(Node: PChar; GetItems: Bool): PChar Returns a cr/lf separated list of items from the Object Browser. The Node parameter determines which items are returned. This can be one of the main items like TABLES, but you can also us a slash to get more specific items like TABLES/DEPT/COLUMNS. The GetItems boolean determines if PL/SQL Developer will fetch these values from the database if the item has not been opened yet in the Browser.
73	**IDE_RefreshBrowser** ***Available in version 400***	*C++* void IDE_RefreshBrowser(char *Node) *Delphi* procedure IDE_RefreshBrowser(Node: PChar) Force a refresh to the Object Browser. If Node is empty, all items are refreshed. To refresh a specific item you can enter the name in the Node parameter. ***Note:*** Version 500 allows you to pass a * to refresh the current selected browser item. ***Note:*** Version 600 allows you to pass a ** to refresh to parent of the current browser item, and you can pass *** to refresh to root item.
74	**IDE_GetPopupObject** ***Available in version 400***	*C++* int IDE_GetPopupObject(char **ObjectType, char **ObjectOwner, char **ObjectName, char **SubObject) *Delphi* procedure IDE_GetPopupObject(var ObjectType, ObjectOwner, ObjectName, SubObject: PChar) This function returns information about the item for which a popup menu

		(created with IDE_CreatePopupItem) was activated. If the item is a Browser folder, the name of the folder will be returned in ObjectName and ObjectType will return 'FOLDER'
75	**IDE_GetPopupBrowserRoot** ***Available in version 400***	*C++* char* IDE_GetPopupBrowserRoot() *Delphi* function IDE_GetPopupBrowserRoot: PChar This function returns the name of browser root item for which a popup menu (created with IDE_CreatePopupItem) was activated.
76	**IDE_RefreshObject** ***Available in version 400***	*C++* void IDE_RefreshObject (char *ObjectType, char *ObjectOwner, char *ObjectName, int Action) *Delphi* procedure IDE_RefreshObject(ObjectType, ObjectOwner, ObjectName: PChar; Action: Integer) If you modify database objects in your Plug-In and you want to update PL/SQL Developer to reflect these changes, you can do so by calling this function. You should pass the object type, owner, name and the action that you performed on the object. The action can be one of the following: 1 = Object created 2 = Object modified 3 = Object deleted PL/SQL Developer will update the browser and all windows that might use the object.
77	**IDE_FirstSelectedObject** ***Available in version 500***	*C++* BOOL IDE_FirstSelectedObject (char *ObjectType, char *ObjectOwner, char *ObjectName, char *SubObject) *Delphi* function IDE_FirstSelectedObject(var ObjectType, ObjectOwner, ObjectName, SubObject: PChar): Bool This function will return the details of the first selected in the Browser. The function will return false if no items are selected. Use in combination with IDE_NextSelectedObject to determine all selected items.
78	**IDE_NextSelectedObject** ***Available in version 500***	*C++* BOOL IDE_NextSelectedObject (char *ObjectType, char *ObjectOwner, char *ObjectName, char *SubObject) *Delphi* function IDE_NextSelectedObject(var ObjectType, ObjectOwner, ObjectName, SubObject: PChar): Bool This function can be called after a call to IDE_FirstSelectedObject to determine all selected objects. You can keep calling this function until it returns false.
79	**IDE_GetObjectSource** ***Available in version 511***	*C++* char* IDE_GetObjectSource(char *ObjectType, char *ObjectOwner, char *ObjectName) *Delphi* function IDE_GetObjectSource (ObjectType, ObjectOwner, ObjectName: PChar): PChar Returns the source for the specified object. This function will only return source for objects that actually have source (packages, views, …).
80	**IDE_GetWindowCount** ***Available in version 301***	*C++* int IDE_GetWindowCount() *Delphi* function IDE_GetWindowCount: Integer Returns the number of child windows in PL/SQL Developer. In combination with IDE_SelectWindow you can communicate with all child windows.
81	**IDE_SelectWindow** ***Available in version 301***	*C++* BOOL IDE_SelectWindow(int Index) *Delphi* function IDE_SelectWindow(Index: Integer): Bool This function will 'select' one of PL/SQL Developers child Windows. Index is the window number where 0 is the top child window. The return value will indicate if the window existed. Normally all window related functions communicate with the active child window. With this function you can select any window and all window-related IDE functions will refer to the selected window.

		Note: IDE_SelectWindow does not actually bring the window to front, you need IDE_ActivateWindow to do that.
82	**IDE_ActivateWindow** ***Available in version 301***	*C++* BOOL IDE_ActivateWindow(int Index) *Delphi* function IDE_ActivateWindow(Index: Integer): Bool Brings the Indexth child window with to front.
83	**IDE_IsWindowModified** ***Available in version 301***	*C++* BOOL IDE_WindowIsModified() *Delphi* function IDE_WindowIsModified: Bool Returns if the contents of the window is modified.
84	**IDE_IsWindowRunning** ***Available in version 301***	*C++* BOOL IDE_WindowIsRunning() *Delphi* function IDE_WindowIsRunning: Bool Returns if there is anything running in the current window.
90	**IDE_SplashCreate** ***Available in version 303***	*C++* void IDE_SplashCreate(int ProgressMax) *Delphi* procedure IDE_SplashCreate(ProgressMax: Integer) Creates an empty splash screen (the one you see when PL/SQL Developer is starting or printing) which allows you to show some kind of progress on lengthy operations. If the ProgressMax parameter is larger then 0, a progress bar is displayed which you can advance with the IDE_SplashProgress function. ***Note:*** There can only be one splash screen active at a time. If a splash screen is created while one was active, the first one will get re-used.
91	**IDE_SplashHide** ***Available in version 303***	*C++* void IDE_SplashHide() *Delphi* procedure IDE_SplashHide Hides the splash screen. This function will work on any splash screen, you can even hide the one created by PL/SQL Developer.
92	**IDE_SplashWrite** ***Available in version 303***	*C++* void IDE_SplashWrite(char *s) *Delphi* procedure IDE_SplashWrite(s: string) Add text to the splash screen.
93	**IDE_SplashWriteLn** ***Available in version 303***	*C++* void IDE_SplashWriteLn(char *s) *Delphi* procedure IDE_SplashWriteLn(s: string) Add text to the splash screen beginning on the next line.
94	**IDE_SplashProgress** ***Available in version 303***	*C++* void IDE_SplashProgress(int Progress) *Delphi* procedure IDE_SplashProgress(Progress: Integer) If the splash screen was created with a progress bar, you can indicate progress with this function.
95	**IDE_TemplatePath** ***Available in version 400***	*C++* char* IDE_TemplatePath() *Delphi* function IDE_TemplatePath: PChar This function returns the path where the templates are located.
96	**IDE_ExecuteTemplate** ***Available in version 400***	*C++* BOOL IDE_ExecuteTemplate(char *Template BOOL NewWindow) *Delphi* function IDE_ExecuteTemplate(Template: PChar; NewWindow: Bool): Bool If you want to execute a template from within your PlugIn you can do so with this function. The NewWindow parameter indicates if a new window should be created or that the result of the template should be pasted at the current cursor position in the active window. The template parameter should contain the template name. If the template is located in one or more folders, the folder name(s) should be prefixed to the template name separated by a backslash.
97	**IDE_GetConnectAs** ***Available in version 500***	*C++* char IDE_GetConnectAs() *Delphi* function IDE_GetConnectAs: PChar

		Use this function to determine if the current connection has a specific 'Connect As'. Possible return values are: '', 'SYSDBA' and 'SYSOPER'
98	**IDE_SetConnectionAs** ***Available in version 500***	*C++* BOOL IDE_SetConnectionAs(char *Username, char *Password, char *Database, char *ConnectAs) *Delphi* function IDE_SetConnectionAs(Username, Password, Database, ConnectAs: PChar): Bool Identical to IDE_SetConnection, but with an option to specify a ConnectAs parameter. You can pass 'SYSDBA' or 'SYSOPER', all other values will be handled as 'NORMAL'.
External FileSystem functions		
100	**IDE_GetFileOpenMenu** ***Available in version 400***	*C++* char* IDE_GetFileOpenMenu(int MenuIndex, int *WindowType) *Delphi* function IDE_GetFileOpenMenu(MenuIndex: Integer; var WindowType: Integer): PChar If you want to create a new 'File Open' menu with the same items as the standard menu, you can use this function to determine the standard items. You can call this function in a loop while incrementing MenuIndex (starting with 0) until the return value is an empty string. The return values are the menu names in the File Open menu and the WindowType is the corresponding window type.
101	**IDE_CanSaveWindow** ***Available in version 400***	*C++* BOOL IDE_CanSaveWindow() *Delphi* function IDE_CanSaveWindow: Bool Returns True if the active child window can be saved. (which are the SQL, Test, Program and Command windows).
102	**IDE_OpenFileExternal** ***Available in version 400***	*C++* void IDE_OpenFileExternal(int WindowType, char *Data, char *FileSystem, char *Tag, char *Filename) *Delphi* procedure IDE_OpenFileExternal(WindowType: Integer; Data, FileSystem, Tag, Filename: PChar) Creates a new Window (of type WindowType) for the specified (and registered) FileSystem, Tag and Filename.
103	**IDE_GetFileTypes** ***Available in version 400***	*C++* char* IDE_GetFileTypes(int WindowType) *Delphi* function IDE_GetFileTypes(WindowType: Integer): PChar Returns the defined filetypes for a specific WindowType.
104	**IDE_GetDefaultExtension** ***Available in version 400***	*C++* char* IDE_GetDefaultExtension(int WindowType) *Delphi* function IDE_GetDefaultExtension(WindowType: Integer): PChar Returns the default extension (without period) for a specific window type.
105	**IDE_GetFileData** ***Available in version 400***	*C++* char* IDE_GetFiledata() *Delphi* function IDE_GetFileData: PChar Returns the data of a window. You can use this function to get the data and save it.
106	**IDE_FileSaved** ***Available in version 400***	*C++* void IDE_FileSaved(char *FileSystem, char *FileTag, char *Filename) *Delphi* procedure IDE_FileSaved(FileSystem, FileTag, Filename: PChar) You can call this function when a file is saved successfully. The filename will be set in the Window caption and the status will display that the file is 'saved successfully'. FileSystem and FileTag can be nil.
107	**IDE_ShowHTML** ***Available in version 510***	*C++* BOOL IDE_ShowHTML(char *Url, char *Hash, char *Title, char *ID) *Delphi* function IDE_ShowHTML(Url, Hash, Title, ID: PChar): Bool

		This function displays a html file in a child window. The url parameter identifies the file and the hash parameter allows you to jump to a specific location. The title parameter will be used as window title. You can refresh the contents of an already opened window by specifying an ID. If ID is not empty, and a window exists with the same ID, this will be used, otherwise a new window will be created.
108	**IDE_RefreshHTML** ***Available in version 512***	*C++* BOOL IDE_RefreshHTML(char *Url, char *ID, BOOL BringToFront) *Delphi* function IDE_ShowHTML(Url, ID: PChar; BringToFront: Bool): Bool Refresh the contents of a HTML Window. You can pass an url to refresh all windows that show a specific url, or you can pass an ID to refresh a specific Window.
109	**IDE_GetProcEditExtension** ***Available in version 514***	*C++* char* IDE_GetProcEditExtension (char *oType) *Delphi* function IDE_GetProcEditExtension (oType: PChar): PChar Returns the define file extension of a specific object type. The oType parameter can hold one of the following valies: FUNCTION, PROCEDURE, TRIGGER, PACKAGE, PACKAGE BODY, PACKAGE SPEC AND BODY, TYPE, TYPE BODY, TYPE SPEC AND BODY, JAVA SOURCE
110	**IDE_GetWindowObject** ***Available in version 512***	*C++* BOOL IDE_GetWindowObject (char **ObjectType, char **ObjectOwner, char **ObjectName, char **SubObject) *Delphi* function IDE_GetWindowObject(var ObjectType, ObjectOwner, ObjectName, SubObject: PChar): Bool Get info about the object opened in a Window. This will only work for Program Windows.
111	**IDE_FirstSelectedFile** ***Available in version 800***	*C++* char* IDE_FirstSelectedFile(BOOL Files, BOOL Directories) *Delphi* function IDE_FirstSelectedFile(Files, Directories: Boolean): PChar; Returns the first selected item in the file browser. Use IDE_NextSelectedFile for multiple selected items. The *Files* and *Directories* parameters allow you to specify if you do or don't want selected files and/or directories.
112	**IDE_NextSelectedFile** ***Available in version 800***	*C++* char* IDE_NextSelectedFile() *Delphi* function IDE_NextSelectedFile: PChar Returns the next selected item. See the previous function. Returns empty value when no more items.
113	**IDE_RefreshFileBrowser** ***Available in version 800***	*C++* void IDE_RefreshFileBrowser() *Delphi* procedure IDE_RefreshFileBrowser Forces the file browser to refresh the contents. Normally the browser will autodetect changes.
120	**IDE_KeyPress** ***Available in version 510***	*C++* void IDE_KeyPress(int Key, int Shift) *Delphi* procedure IDE_KeyPress(Key, Shift: Integer) Simulates a key press. You can use this function to do the things you can also do with the keyboard. The Key parameter is the virtual key code of the key, and the Shift parameter holds the status of the Shift Ctrl and Alt keys. You can combine the following values: 1 = Shift 2 = Alt 3 = Ctrl
121	**IDE_GetMenuItem** ***Available in version 510***	*C++* int IDE_GetMenuItem(char *MenuName) *Delphi* function IDE_GetMenuItem(MenuName: PChar): Integer

		This function will return an ‘index’ of a specific menu item. The MenuName parameter must specify the menu path separated by a slash, for example ‘edit / selection / uppercase’. The menu name is not case sensitive. If the function returns zero, the menu did not exist. You can use the return value with IDE_SelectMenu
122	**IDE_SelectMenu** ***Available in version 510***	*C++* BOOL IDE_SelectMenu(int MenuItem) *Delphi* function IDE_SelectMenu(MenuItem: Integer): Bool You can execute a menu item with this function. The MenuItem parameter has to be determined by the IDE_SelectMenu function. If this function returns false, the menu did not exist, or it was disabled.
130	**IDE_TranslationFile** ***Available in version 510***	*C++* char* IDE_TranslationFile() *Delphi* function IDE_TranslationFile: PChar Returns the currently used translation file. If the return value is empty, no translation is used.
131	**IDE_TranslationLanguage** ***Available in version 510***	*C++* char* IDE_TranslationLanguage() *Delphi* function IDE_TranslationLanguage: PChar Returns the language of the currently used translation file. If the return value is empty, no translation is used.
132	**IDE_GetTranslatedMenuLayout** ***Available in version 510***	*C++* char* IDE_GetTranslatedMenuLayout() *Delphi* function IDE_GetTranslatedMenuLayout: PChar Returns a list of all standard PL/SQL Developer menu items like IDE_GetMenuLayout, but this function will return the translated menus.
133	**IDE_MainFont** ***Available in version 510***	*C++* char* IDE_MainFont() *Delphi* function IDE_MainFont: PChar Return the PL/SQL Developer main font in the format: “Name”, size, color, charset, “style”
134	**IDE_TranslateItems** ***Available in version 510***	*C++* char* IDE_TranslateItems(char *Group) *Delphi* function IDE_TranslateItems(Group: PChar): PChar Function for translating items.
135	**IDE_TranslateString** ***Available in version 510***	*C++* char* IDE_TranslateString(char *ID, char *Default, char Param1, char Param2) *Delphi* function IDE_TranslateString(ID, Default, Param1, Param2: PChar): PChar Function for translating items.
140	**IDE_SaveRecoveryFiles** ***Available in version 510***	*C++* BOOL IDE_SaveRecoveryFiles() *Delphi* function IDE_SaveRecoveryFiles: Bool PL/SQL Developer has a preference to save all opened files on a time interval, and/or when an Execute is performed. In case of a crash (from the system, Oracle or PL/SQL Dev), the user will be able to recover the edited files. If the Plug-In can do things that have a possible risk of causing a crash, you can call this function to protect the user’s work.
141	**IDE_GetCursorX** ***Available in version 510***	*C++* int IDE_GetCursorX() *Delphi* function IDE_GetCursorX: Integer Returns the (1 based) character position of the cursor in the current editor.
142	**IDE_GetCursorY** ***Available in version 510***	*C++* int IDE_GetCursorY() *Delphi* function IDE_GetCursorY: Integer Returns the (1 based) line position of the cursor in the current editor.
143	**IDE_SetCursor** ***Available in version 510***	*C++* void IDE_SetCursor(int X, int Y) *Delphi* procedure IDE_SetCursor(X, Y: Integer)

		Set the cursor in the current editor. If the X or Y parameter is 0, the position will not change. This function will also update the position display in the statusbar.
144	**IDE_SetBookmark** ***Available in version 510***	*C++* int IDE_SetBookmark(int Index, int X, int Y) *Delphi* function IDE_SetBookmark(Index, X, Y: Integer): Integer Create a bookmark at position X (character), Y (line). Index is the bookmark (0..9) you want to set. If you pass –1 as bookmark, the first free bookmark will be used. The returned value is the used bookmark. Normally, from within PL/SQL Developer. Bookmarks can only be used for windows with a gutter (Test window and Program editor), but the Plug-In interface allows you to use bookmarks for all windows.
145	**IDE_ClearBookmark** ***Available in version 510***	*C++* void IDE_ClearBookmark(int Index) *Delphi* procedure IDE_ClearBookmark(Index: Integer) Clears the specified bookmark
146	**IDE_GotoBookmark** ***Available in version 510***	*C++* void IDE_GotoBookmark(int Index) *Delphi* procedure IDE_GotoBookmark(Index: Integer) Jumps to a bookmark
147	**IDE_GetBookmark** ***Available in version 510***	*C++* BOOL IDE_GetBookmark(int Index, int X, int Y) *Delphi* function IDE_GetBookmark(Index: Integer; var X: Integer; var Y: Integer): Bool Get the cursor position for a specific bookmark
148	**IDE_TabInfo** ***Available in version 511***	*C++* char* IDE_TabInfo(int Index) *Delphi* function IDE_TabInfo(Index: Integer): PChar Returns the description tab page Index (zero based). The return value is empty if the tab page does not exist. This function allows you to determine which tab pages (if any) are available for the current window.
149	**IDE_TabIndex** ***Available in version 511***	*C++* int IDE_TabIndex(int Index) *Delphi* function IDE_TabIndex(Index: Integer): Integer This function allows you to read or set the active tab page. To set a specific page, pass a zero based value to the Index parameter. The return value is the actual selected page. To determine the active page (without setting it) pass a value of –1 to the Index parameter.
150	**IDE_CreateToolButton** ***Available in version 510***	*C++* void IDE_CreateToolButton(int ID, int Index, char *Name char *BitmapFile, int BitmapHandle) *Delphi* procedure IDE_CreateToolButton(ID, Index: Integer; Name: PChar; BitmapFile: PChar; BitmapHandle: Integer) This function allows you to add Toolbuttons to your Plug-In, similar to IDE_CreatePopupItem. The ID is the Plug-In ID and the index is the menu index. When a button is selected, OnMenuClick is called with the corresponding index. The Name will appear as hint for the button, and as name in the preferences dialog. The button can be enabled and disabled with IDE_MenuState. The image for the button can be set by passing a filename to a bmp file in the BitmapFile parameter, or as a handle to a bitmap in memory. The bmp image can have any number of colors, but should approximately be 20 x 20 pixels in size. The button will only be visible if it is selected in the Toolbar preference.
153	**IDE_WindowHasEditor** ***Available in version 710***	*C++* BOOL IDE_WindowHasEditor(BOOL CodeEditor) *Delphi* procedure IDE_WindowHasEditor(CodeEditor: Bool) Returns true if the current Window has an Editor. If the CodeEditor parameter is true, it returns false for editors like the output editor.

160	**IDE_BeautifierOptions** ***Available in version 510***	*C++* int IDE_BeautifierOptions() *Delphi* function IDE_BeautifierOptions: Integer Returns the PL/SQL Beautifier options. The result is a value where the following values are or-ed together: 1 AfterCreating enabled 2 AfterLoading enabled 4 BeforeCompiling enabled 8 BeforeSaving enabled You can use this to determine if you need to call the beautifier.
161	**IDE_BeautifyWindow** ***Available in version 510***	*C++* BOOL IDE_BeautifyWindow() *Delphi* function IDE_BeautifyWindow: Bool Calls the PL/SQL Beautifier for the current Window. The result indicates if the operations succeeded.
162	**IDE_BeautifyText** ***Available in version 510***	*C++* char* IDE_BeautifyText(char *S) *Delphi* function IDE_BeautifyText(S: PChar): PChar Calls the PL/SQL Beautifier to beautify the text in the S parameter. The result is the beautified text or it is empty if the function failed
165	**IDE_ObjectAction** ***Available in version 514***	*C++* BOOL IDE_ObjectAction(char *Action, char *ObjectType, char *ObjectOwner, char *ObjectName) *Delphi* IDE_ObjectAction(Action, ObjectType, ObjectOwner, ObjectName: PChar): Bool This function allows you to do a specific action for the object specified. The following actions are available: VIEW, VIEWSPECANDBODY, EDIT, EDITSPECANDBODY, EDITDATA, QUERYDATA, TEST
166	**IDE_ShowDialog** ***Available in version 700***	*C++* BOOL IDE_ShowDialog (char *Dialog, char *Param) *Delphi* function IDE_ShowDialog(Dialog, Param: PChar): Bool This allows you to start a specific PL/SQL Developer dialog. The following are supported: AUTHORIZATIONS PROJECTITEMS BREAKPOINTS PREFERENCES CONFIG PLUGINS CONFIG TOOLS CONFIG DOCUMENTS CONFIG REPORTS CONFIG MACROS CONFIG AUTOREFRESH The Param parameter is for future use.
173	**IDE_DebugLog** ***Available in version 700***	*C++* void IDE_DebugLog(char *Msg) *Delphi* procedure IDE_DebugLog(Msg: PChar) When debuggin is on, this function allows you to add messages in the debug.txt file generated.
174	**IDE_GetParamString** ***Available in version 700***	*C++* char* IDE_GetParamString(char *Name) *Delphi* function IDE_GetParamString(Name: PChar): PChar This function returns a command-line parameter, or a parameter specified in the params.ini file.
175	**IDE_GetParamBool** ***Available in version 700***	*C++* BOOL IDE_GetParamBool(char *Name) *Delphi* function IDE_GetParamBool(Name: PChar): Bool This function returns a command-line parameter, or a parameter specified in the params.ini file.
176	**IDE_GetBrowserFilter** ***Available in version 702***	*C++* void IDE_GetBrowserFilter(int Index, char **Name, char **WhereClause, char **OrderByClause, char **User, BOOL Active) *Delphi* procedure IDE_GetBrowserFilter(Index: Integer; var

		Name, WhereClause, OrderByClause, User: PChar; var Active: Bool) This function returns the defined browser filters. You can use this if the Plug-in has a similar requirement. Index = 0 and higher, and the returned values are empty if the filter does not exist.
180	**IDE_CommandFeedBack** ***Available in version 513***	*C++* void IDE_CommandFeedback(int FeedbackHandle char *S) *Delphi* procedure IDE_CommandFeedback(FeedBackHandle: Integer; S: PChar) This function allows you to return feedback to the command window. The description S will be displayed in the window identified by the FeedbackHandle. See the CommandLine Plug-In function for details.
190	**IDE_ResultGridRowCount** ***Available in version 516***	*C++* int IDE_ResultGridRowCount() *Delphi* function IDE_ResultGridRowCount: Integer Returns the number of rows in the result grid of a SQL or Test Window.
191	**IDE_ResultGridColCount** ***Available in version 516***	*C++* int IDE_ResultGridColCount() *Delphi* function IDE_ResultGridColCount: Integer Returns the number of cols in the result grid of a SQL or Test Window.
192	**IDE_ResultGridCell** ***Available in version 516***	*C++* char* IDE_ResultGridCell(int Col, int Row) *Delphi* function IDE_ResultGridCell(Col, Row: Integer): PChar This function allows you to access the results of a query in a SQL or Test Window. Use the above two functions to determine the number of rows and cols.
200	**IDE_Authorized** ***Available in version 600***	*C++* BOOL IDE_CommandFeedback(char * Category, char *Name, char *SubName) *Delphi* function IDE_Authorized(Category, Name, SubName: PChar): Bool In PL/SQL Developer 6 we introduced the concept of Authorization. You should test if a specific feature is allowed for the current user with this function. In the Category parameter you can specify one of the main categories (objects, menus, system). The name parameter specifies the item (session.kill or objects.drop). Some items have a subname, like objects.drop with the different objects.
201	**IDE_WindowAllowed** ***Available in version 600***	*C++* BOOL IDE_WindowAllowed(int WindowType, BOOL ShowErrorMessage) *Delphi* function IDE_WindowAllowed(WindowType: Integer; ShowErrorMessage: Bool): Bool For a quick check if authorization allows the Plug-In to create a specific function, you can use this function.
202	**IDE_Authorization** ***Available in version 600***	*C++* BOOL IDE_Authorization() *Delphi* function IDE_Authorization: Bool Returns if authorization is enabled or not.
203	**IDE_AuthorizationItems** ***Available in version 600***	*C++* char* IDE_AuthorizationItems(char *Category) *Delphi* function IDE_AuthorizationItems(Category: PChar): PChar If you want a list off all available authorization items, you can call this function. It will return a cr/lf separated list.
204	**IDE_AddAuthorizationItem** ***Available in version 600***	*C++* void IDE_AddAuthorizationItem(int PlugInID, char *Name) *Delphi* procedure IDE_AddAuthorizationItem(PlugInID: Integer; Name: PChar) If you want to add items to the authorization list to allow them to be

		managed through the authorization option, you can use this function. Pass the PlugInID to identify your Plug-In, and pass the Name parameter with the item you want to add. The name should be unique, so you should prefix it with the name the Plug-In, for example: *MyPlugIn.Create New Command* All items will be added in the PlugIns category, so if you want to test if this feature is allowed you should call: *IDE_Authorized('PlugIns ', ' MyPlugIn.Create New Command')*
210	**IDE_GetPersonalPrefSets** ***Available in version 600***	*C++* char* IDE_GetPersonalPrefSets() *Delphi* function IDE_GetPersonalPrefSets: PChar Returns a list of all personal preference sets. If you to have the Plug-In to use different preferences depending on the current connection, you can use this function to build a list of possible preference sets.
211	**IDE_GetDefaultPrefSets** ***Available in version 600***	*C++* char* IDE_GetDefaultPrefSets() *Delphi* function IDE_GetDefaultPrefSets: PChar Returns a list of all default preference sets.
212	**IDE_GetPrefAsString** ***Available in version 600***	*C++* char* IDE_GetPrefAsString(int PlugInID, char * PrefSet, char *Name, char *Default) *Delphi* function IDE_GetPrefAsString(PlugInID: Integer; PrefSet, Name: PChar; Default: PChar): PChar Read a Plug-In preference from the preferences. In PL/SQL Developer 6, personal preferences are stored in files, not in the registry. You can still use the registry, but if you want to store your preferences in a shared location, you can use this function. Pass the PlugInID you received with the IdentifyPlugIn call. The PrefSet parameter can be empty to retrieve default preferences, or you can specify one of the existing preference sets.
213	**IDE_GetPrefAsInteger** ***Available in version 600***	*C++* int IDE_GetPrefAsInteger(int PlugInID, char * PrefSet, char *Name, BOOL Default) *Delphi* function IDE_GetPrefAsInteger(PlugInID: Integer; PrefSet, Name: PChar; Default: Integer): Integer As IDE_GetPrefAsString, but for integers.
214	**IDE_GetPrefAsBool** ***Available in version 600***	*C++* BOOL IDE_GetPrefAsBool(int PlugInID, char * PrefSet, char *Name, BOOL Default) *Delphi* function IDE_GetPrefAsBool(PlugInID: Integer; PrefSet, Name: PChar; Default: Bool): Bool As IDE_GetPrefAsString, but for booleans.
215	**IDE_SetPrefAsString** ***Available in version 600***	*C++* BOOL IDE_SetPrefAsString(int PlugInID, char *PrefSet, char *Name, char *Value) *Delphi* function IDE_SetPrefAsString(PlugInID: Integer; PrefSet, Name: PChar; Value: PChar): Bool Set a Plug-In preference. Pass the PlugInID you received with the IdentifyPlugIn call. The PrefSet parameter can be empty to set default preferences, or you can specify one of the existing preference sets. The return value indicates if the function succeeded.
216	**IDE_SetPrefAsInteger** ***Available in version 600***	*C++* BOOL IDE_SetPrefAsInteger(int PlugInID, char *PrefSet, char *Name, int Value) *Delphi* function IDE_SetPrefAsInteger(PlugInID: Integer; PrefSet, Name: PChar; Value: Integer): Bool As IDE_SetPrefAsString, but for integers.
217	**IDE_SetPrefAsBool** ***Available in version 600***	*C++* BOOL IDE_SetPrefAsBool(int PlugInID, char *PrefSet, char *Name, BOOL Value) *Delphi* function IDE_SetPrefAsBool(PlugInID: Integer; PrefSet, Name: PChar; Value: Bool): Bool

		As IDE_SetPrefAsString, but for booleans.
218	**IDE_GetGeneralPref** ***Available in version 700***	*C++* char* IDE_GetGeneralPref(char *Name) *Delphi* function IDE_GetGeneralPref(Name: PChar): PChar Returns the value of a preference. The names can be found in the preference ini file under the [Preferences] section.
219	**IDE_PlugInSetting** ***Available in version 710***	*C++* BOOL IDE_PlugInSetting(int PlugInID char *Setting char *Value) *Delphi* function IDE_PlugInSetting(PlugInID: Integer; Setting, Value: PChar): Bool Make a Plug-In specific setting: NOFILEDATECHECK TRUE|FALSE Determines if PL/SQL Developer checks for changes in files (default true) CHARMODE ANSI|UTF8|UTF8BOM Determines how PChar parameters are passed through the Plug-In interface. Since version 7.1 supports editing of Unicode, but the interface only supports normal characters, you can choose to support UTF8 encoding. The UTF8BOM encoding will precede the characters with a BOM indicator when text contains Unicode.
220	**IDE_GetProcOverloadCount** ***Available in version 700***	*C++* int IDE_GetProcOverloadCount (char *Owner, char *PackageName, char *ProcedureName) *Delphi* IDE_GetProcOverloadCount(Owner, PackageName, ProcedureName: PChar): Integer Returns the number of overloads for a specific procedure. Result < 0 = Procedure doesn't exist Result > 0 = overload count
221	**IDE_SelectProcOverloading** ***Available in version 700***	*C++* int IDE_SelectProcOverloading (char *Owner, char *PackageName, char *ProcedureName) *Delphi* IDE_SelectProcOverloading(Owner, PackageName, ProcedureName: PChar): Integer Shows a dialog to allow the user to select an overloaded procedure. Result < 0 = Cancel Result 0 = No overloadings Result > 0 = Overload index
230	**IDE_GetSessionValue** ***Available in version 700***	*C++* char* IDE_GetSessionValue (char *Name) *Delphi* function IDE_GetSessionValue(Name: PChar): PChar This function will return one of the Session parameters as you see in the grid of the session tool. You will only get a result if the Session Window is active, so this will only work from a Popup menu created for the SESSIONWINDOW object.
231	**IDE_CheckDBVersion** ***Available in version 700***	*C++* BOOL IDE_CheckDBVersion(char *Version) *Delphi* function IDE_CheckDBVersion(Version: PChar): Boolean You can use this function to check if the database is equal or higher then the specified version. The parameter should be in the format aa.bb, like 09.02 or 10.00.
SQL functions		
40	**SQL_Execute**	*C++* int SQL_Execute(char *SQL) *Delphi* function SQL_Execute(SQL: PChar): Integer Executes the statement defined in the SQL parameter. The function returns 0 if successful, else the Oracle error number.
41	**SQL_FieldCount**	*C++* int SQL_FieldCount() *Delphi* function SQL_FieldCount: Integer Returns the number of fields after a SQL_Execute.
42	**SQL_Eof**	*C++* BOOL SQL_Eof()

		Delphi function SQL_Eof: Bool Returns if there are any more rows to fetch.
43	**SQL_Next**	*C++* int SQL_Next() *Delphi* function SQL_Next: Integer Returns the next row after a SQL_Execute. The function returns 0 if successful, else the Oracle error number.
44	**SQL_Field**	*C++* char* SQL_Field(int Field) *Delphi* function SQL_Field(Field: Integer): PChar Returns the field specified by the Field parameter.
45	**SQL_FieldName**	*C++* char* SQL_FieldName(int Field) *Delphi* function SQL_FieldName(Field: Integer): PChar Returns the fieldname specified by the Field parameter.
46	**SQL_FieldIndex**	*C++* int SQL_FieldIndex(char *Name) *Delphi* function SQL_FieldIndex(Name: PChar): Integer Converts a fieldname into an index, which can be used in the SQL_Field, SQL_FieldName and SQL_FieldType functions. If the field does not exist, the return value is -1.
47	**SQL_FieldType**	*C++* int SQL_FieldType(int Field) *Delphi* function SQL_FieldType(Field: Integer): Integer Return the fieldtype of a field. 3 = otInteger 4 = otFloat 5 = otString 8 = otLong 12 = otDate 24 = otLongRaw
48	**SQL_ErrorMessage** ***Available in version 301***	*C++* char* SQL_ ErrorMessage() *Delphi* function SQL_ErrorMessage: PChar This function will return the error message for any error that occurred during: SQL_Execute SQL_Eof SQL_Next IDE_SetConnection
50	**SQL_UsePlugInSession** ***Available in version 600***	*C++* BOOL SQL_UsePlugInSession(int PlugInID) *Delphi* function SQL_UsePlugInSession(PlugInID: Integer): Bool Normally, the SQL functions will use the main PL/SQL Developer Oracle session. If you want to make sure you don't interfere with other transactions, and you want the PlugIn to use a private session, call this function. The return value indicates if the function succeeded.
51	**SQL_UseDefaultSession** ***Available in version 600***	*C++* void SQL_UseDefaultSession(int PlugInID) *Delphi* procedure SQL_UseDefaultSession(PlugInID: Integer) This function will cancel the previous function and set the Oracle session back to default.
52	**SQL_CheckConnection** ***Available in version 700***	*C++* BOOL SQL_CheckConnection() *Delphi* function SQL_CheckConnection: Bool Forces PL/SQL Developer to check if the current connection to the database is still open (and tries a re-connect if necessary). The return value indicates if there is a connection.
53	**SQL_GetDBMSGetOutput** ***Available in version 700***	*C++* char* SQL_GetDBMSGetOutput() *Delphi* function SQL_GetDBMSGetOutput: PChar

		Returns sys.dbms_output for the current (PlugIn specific) session.
54	**SQL_SetVariable** ***Available in version 700***	*C++* void SQL_SetVariable (char *Name, char *Value) *Delphi* procedure SQL_SetVariable(Name, Value: PChar) This function declares a variable. Call this for al variables you use in the statement you pass in SQL_Execute.
55	**SQL_GetVariable** ***Available in version 700***	*C++* char* SQL_GetVariable (char *Name) *Delphi* function SQL_GetVariable(Name: PChar): PChar This function will return the value of a variable.
56	**SQL_ClearVariables** ***Available in version 700***	*C++* void SQL_ClearVariables () *Delphi* procedure SQL_ClearVariables Clear all declared variables. If you are finished doing a query it is a good idea to call this function to prevent errors for the next execute.

The callback functions are divided into three groups, SYS functions (returning system information), IDE functions (for interaction with the PL/SQL Developer IDE) and SQL functions.

The SYS functions return PL/SQL Developer and Oracle information. You might need these to locate or store information.

The IDE functions allow you to communicate with the PL/SQL Developer IDE. Some functions return information of the current state of PL/SQL Developer. This allows your Plug-In to be context sensitive. If you want to send messages to a window or an Editor, you can use the handle functions to get hold of any handle you might need.

The SQL functions can be used to execute any kind of SQL statement. If, for example, you wanted to query all existing tables you could use the SQL functions like this:

```
SQL_Execute('Select * from all_tables');
index = SQL_FieldIndex('TABLE_NAME');
while not SQL_Eof do
begin
  FieldName := SQL_Field(index);
  // Do something with Fieldname
  SQL_Next;
end;
```

Note that you can’t nest queries. You should also be aware that the Oracle session used for the query is the same session that is used internally by PL/SQL Developer for compilations and other DDL statements. If the Session Mode preference is set to Dual Session or Multi Session, a different session is used for all SQL Windows, Test Windows and Command Windows.

All returned string values (like the value from SQL_Field) are returned as a pointer to an array of zero terminated characters. PL/SQL Developer allocates memory for this array but you should copy the value if you are going to use it because the same buffer will be used again for the next function that returns a string.

Developing your Plug-In

While developing your Plug-In it might be handy to configure PL/SQL Developer to pick up the Plug-In in your development directory. Simply set the Plug-Ins directory in the preferences dialog to your development directory. The default Plug-In directory will always be checked so any other Plug-Ins will still be loaded.

Most programming languages allow you to define a “host” application while developing a DLL. If you define PL/SQL Developer as host application you can “run” your Plug-In while actually PL/SQL Developer is started which (if configured properly) will load your Plug-In. This allows you to quickly test any modifications.

Note that PL/SQL Developer will only load a Plug-In if the description is unique. If you have Plug-Ins with identical descriptions, only the first one is loaded.

You should also be aware that C++ programming languages will modify exported function names. This has something to do with method overloading, but it will cause PL/SQL Developer to ignore the Plug-In because the expected exported functions were not found. Use extern "c" to prevent function names from being mangled in C++ programs, like this:

```
extern "C"
{
  __declspec(dllexport) char* IdentifyPlugIn(int);
  __declspec(dllexport) char* CreateMenuItem(int);
  __declspec(dllexport) void  RegisterCallback(int, void *);
  __declspec(dllexport) void  OnMenuClick(int);
}
```

It might be a good idea to start with one of the supplied demos. We have included demos in C++Builder (version 3 and upwards) and Delphi (2 and upwards) format.

Debugging a Plug-In

In version 7.1, PL/SQL Developer has a new commandline option DEBUGPLUGINS, which will write debuglines to debug.txt like Plug-In initialization info and the functions called.

Plug-In External FileSystem

The External file system functions allows you to add open and save functions to store files wherever you want. Our FTP Plug-In is an example of this. You need to create a unique name for your “filesystem” and export a RegisterFileSystem function like this:

```
const FileSystem = 'FTP';

function RegisterFileSystem: PChar; cdecl;
begin
  Result := FileSystem;
end;
```

Next you probably want to add open and save menu items so you can actually handle files. The FTP Plug-In does something like this:

```
var WindowType: Array[0..9] of Integer;

function CreateMenuItem(Index: Integer): PChar;  cdecl;
var S: string;
    wt: Integer;
begin
  MenuString := '';
  case Index of
    2 : MenuString := PChar('File / Save As... >> FTP Save As...');
    3 : MenuString := PChar('File / Open >> FTP Open');
   10 ..
   19 : begin
          S := IDE_GetFileOpenMenu(Index - 10, wt);
          if wt <> wtNone then
          begin
            WindowType[Index - 10] := wt;
            MenuString := PChar('File / FTP Open / ' + S);
          end;
        end;
  end;
  Result := PChar(MenuString);
end;
```

Item 2 adds a “save as” menu, and item 3 adds an “open” group where items 10 to 19 add a menu for all existing window types. The OnMenuClick can look like this:

```
procedure OnMenuClick(Index: Integer);  cdecl;
begin
  case Index of
     2 : FileSave;
    10 ..
    19 : FileOpen(WindowType[Index - 10]);
  end;
end;

procedure FileSave;
var w: Integer;
    sProfileName, sFileName, E: string;
    oStream: TStringStream;
begin
  w := IDE_GetWindowType;
  E := IDE_GetFileTypes(w);
  FTP.DefaultFileExt := IDE_GetDefaultExtension(wtNone);
  FTP.CurrentWindowType := w;
  oStream := TStringStream.Create(IDE_GetFileData);
  try
    sProfileName := '';
    sFileName := '';
    if FTP.SaveFile(sProfileName, sFileName, E, oStream) then
      IDE_FileSaved(FileSystem, PChar(sProfileName), PChar(sFileName));
  finally
    oStream.Free;
  end;
end;

procedure FileOpen(w: Integer);
var sProfileName, sFileName, E: string;
    oStream: TStringStream;
```

```
begin
  E := IDE_GetFileTypes(w);
  FTP.DefaultFileExt := IDE_GetDefaultExtension(wtNone);
  FTP.CurrentWindowType := w;
  oStream := TStringStream.Create('');
  try
    sProfileName := '';
    sFileName := '';
    if FTP.OpenFile(sProfileName, sFileName, E, oStream) then
      IDE_OpenFileExternal(w, PChar(oStream.DataString), FileSystem, PChar(sProfileName),
PChar(sFileName));
  finally
    oStream.Free;
  end;
end;
```

Above is the general code as used in our FTP Plug-in.

The IDE_FileSaved and IDE_OpenFileExternal functions have a filesystem and tag parameter. The first is the name you declared in RegisterFileSystem, the second (tag) parameter can be used for your own use. In the case of the FTP Plug-In it holds the profile name, which is the name that references a defined connection.

In addition to the above you'll also need to add a load and save function to bypass the file dialog. This is required for when you open a file from the recently used file list, or if you select “save file” and not “save as”, or when loading/saving the application desktop. For this you need to add the following two exported functions.

```
function DirectFileLoad(var Tag, Filename: PChar; WindowType: Integer): PChar; cdecl;
function DirectFileSave(var Tag, Filename: PChar; Data: PChar; WindowType: Integer): Bool; cdecl;
```

Again, the tag is passed as a parameter, together with a filename, the windowtype and the actual data. The tag, filesystem and filename is stored with every window.

Plug-In Export functions

Not all functions related to export functions are described yet. If you want to create your own data export module, just let us know and we will give you some additional information.

There is a Delphi RTF Export demo you can use as a reference.

Distributing your Plug-In

Installing your Plug-In basically means copying it to PL/SQL Developers Plug-In directory. If you want to build an installer, you can determine the PL/SQL Developer directory by reading the following registry value:

HKEY_CLASSES_ROOT\PL/SQL Developer\Shell\Open\Command

Which will return something like:

"C:\Program Files\PLSQL Developer\PLSQLDev.exe"

If you remove the executable name and add “PlugIns”, you have the destination path.

You can make Plug-Ins user specific by placing them in an additional “username” directory.
PL/SQL Developer uses the following sequence to look for Plug-Ins:

1 Load Plug-Ins from Plug-Ins preference setting
2 Load Plug-Ins from PlugIns\Username\*.dll
3 Load Plug-Ins from PlugIns*.dll

New Plug-Ins will be active when PL/SQL Developer starts.

Notes for MS Visual C++

If you want to build a Plug-In with Microsoft Visual C++, please note that the registration of the callback functions is slightly different from the Borland C++Builder examples. The RegisterCallback function for MS Visual C++ would look like this:

```
void RegisterCallback(int Index, void *Addr)
{
  switch (Index)
  {
    case 10 :
      void* IDE_MenuState = Addr;
      break;
    case 11 :
      void * IDE_Connected = Addr;
      break;

  }
}
```

Notice the difference in the void* declaration.

Contacting us

If you want to contact us with questions or remarks about the Plug-In interface or PL/SQL Developer in general, just send an email to:

Allround Automations
support@allroundautomations.com
http://www.allroundautomations.com/plsqldev.html