Tech Note

embarcadero®

Extending the Delphi IDE

Bruno Fierens

March 2011

Americas Headquarters
100 California Street, 12th Floor
San Francisco, California 94111

EMEA Headquarters
York House
18 York Road
Maidenhead, Berkshire
SL6 1SF, United Kingdom

Asia-Pacific Headquarters
L7. 313 La Trobe Street
Melbourne VIC 3000
Australia

INTRODUCTION

Embarcadero Delphi® offers a rich API that enables developers to customize and extend the IDE in many ways. The goal of this whitepaper is to introduce a number of these APIs to you and provide samples on how they can be used. Among these samples are also a number of free IDE extensions provided by TMS software. The APIs available to extend the IDE are under the umbrella "OTAPI", which is an acronym for Open Tools API. This whitepaper and the samples have been written for and tested with Delphi XE.

ADDITIONAL INFORMATION

Other than this whitepaper, a lot of information on OTAPI can be found in the source code that is provided with Delphi XE. The API is declared in the unit TOOLSAPI.PAS. For Delphi XE, this unit can be found, by default, in the folder:

C:\Program Files\Embarcadero\RAD Studio\8.0\source\ToolsAPI

It not only contains the definitions of the interfaces but also has a lot of useful comments in the source code that can help in learning the purpose of the interfaces. You will regularly want to refer to this valuable source of information..

Another interesting source of information is the GExperts source code and OTAPI FAQ page which you can find at: http://www.gexperts.org/otafaq.html

BASIC ARCHITECTURE

The API is heavily based on interfaces. The interfaces typically start with the prefix IOTA or INTA. The IDE exposes a lot of interfaces that can be called from the plugin; conversely, the IDE itself can also call code from the plugin when a specific action is triggered in the IDE. To inform the IDE that the plugin has a handler for these actions, in most cases, this is done by writing a class descending from TNotifierObject that implements an interface and register the class with the IDE. As a plugin writer, you will find yourself mostly writing code that calls the IDE interfaces and write classes that implement interfaces that will be called from the IDE.

AREAS OF THE IDE THAT CAN BE EXTENDED

The Delphi IDE can be extended in many ways. This is a brief overview of the most common areas of extending the IDE:

- Create and add custom docking panels

It is possible to add custom docking panels like the component palette panel, the object inspector panel etc...

- Interact with the code editor
  Interfaces are offered to programmatically manipulate the Delphi IDE code editor; for example, to insert snippets of code, replace text, handle special key sequences, add custom syntax highlighters and more...

- Interact with Code Insight
  Code Insight in the editor can be customized as well, offering custom help texts on specific constructs in the code.

- Interact with the Project Manager
  The IDE allows you to have custom context menus to projects and files in the IDE Project Manager tool panel.

- Add custom wizards, items to the repository
  It is possible to add custom items or start custom wizards from items added to the Delphi repository. From these wizards, new project types, specific form types, or data modules can be created.

- Interact with ToDo items
  An API is also available to interact with ToDo items in code from a Delphi IDE extension.

- Interact with debugger, create custom debugger visualizers
  In newer Delphi versions, an IDE extension can be added that provides a custom display of a specific data type while debugging.

- Interact with the form designer
  From a Delphi plugin, an API is available to interact with the form designer as well.

- Splash screen notifications
  An interface is provided to add custom text on the splash screen during the startup of the IDE.

In this whitepaper, various parts of the OTAPI will be covered such as creating custom docking panels, accessing the editor, the project manager, the repository, and extending various menus.

HISTORY OF THE OPEN TOOLS API

As Delphi has grown through the years, so has the API to extend the IDE. In Delphi XE, existing APIs to customize the IDE have been extended, and new APIs have been added. As the OTAPI is mainly interface based, new capabilities are generally offered via new interfaces. To keep things organized, the Delphi team adopts the naming convention for additional interfaces that descend from an earlier interface by giving the new interface the name of the original interface with a suffix using the IDE version. Please note that the IDE version is different from the compiler version. For example, the interface to extend the repository was originally IOTARepositoryWizard and in later IDE versions, it was extended and named IOTARepositoryWizard60 and later to IOTARepositoryWizard80.

If a plugin wants to take advantage of some new capabilities exposed in later versions of the IDE, it should provide a class that implements the newest interface. At the same time, a plugin that was created for an older IDE version and that implements for example the original IOTARepositoryWizard interface will keep working.

For example:

- IOTARepositoryWizard = interface(IOTAWizard)
  Base interface for extending the repository.

- IOTARepositoryWizard60 = interface(IOTARepositoryWizard)
  Extended interface offered from Delphi 7 to provide a way to differentiate between VCL and CLX projects.

- IOTARepositoryWizard80 = interface(IOTARepositoryWizard60)
  New interface introduced in Delphi 2005 to provide access to the new gallery hierarchical structure of the repository and access to multiple personalities the IDE can host, in the case of Delphi 2005 being VCL and VCL.NET.

This is the list of IDE version numbers that are being used in various OTAPI interfaces:

60 = Delphi 7
80 = Delphi 8
90 = Delphi 2005
100 = Delphi 2006
110 = Delphi 2007
120 = Delphi 2009
140 = Delphi 2010
145 = Delphi XE

ACCESSING THE IDE

While the IDE offers various interfaces for implementation that will be called when a particular part of the IDE is accessed, in many cases, it is necessary to directly call some IDE functionality from a plugin. To allow this, Delphi exposes many interfaces. When the IDE starts it creates a global variable BorlandIDEServices that implements various interfaces. When the unit ToolsAPI is in the uses list, this variable BorlandIDEServices is accessible and can be used to query the interface needed.

For example:

```
  // check if the BorlandIDEServices global variable is assigned
  if Assigned(BorlandIDEServices) then
  begin
    // access the IOTAModuleServices interface implemented in
BorlandIDEServices and call CloseALL to close all modules
    (BorlandIDEServices as IOTAModuleServices).CloseAll;
  end;
```

The following interfaces are exposed by the Delphi XE IDE:

- INTAServices
  Interface to access IDE toolbars, menu, imagelists, actions

- IOTAActionServices
  Interface to open, close, save files

- IOTACodeInsightServices
  Interface gives access to the code insight managers and their interface IOTACodeInsightManager

- IOTADebuggerServices
  Interface gives access to debugger related functions such as breakpoints, event logging, processes

- IOTAEditorServices
  Interface gives access to the IDE editor, the edit buffer, edit view, options

- IOTAEditorViewServices
  Interface gives access to the IDE editor view

- INTAEnvironmentOptionsServices

Interface to add options in the IDE Tools, Options menu

- IOTAKeyBindingServices
  Interface to access shortcut key bindings

- IOTAKeyboardServices
  Interface to give access to IDE macro recording & playback

- IOTAMessageServices
  Interface to access the message view, allows adding custom messages to the IDE message view, clear messages

- IOTAModuleServices
  Interface to access the modules (e.g. projects, source files, form files, etc.) opened in the IDE

- IOTAPackageServices
  Interface to access the list of installed packages & components in the IDE

- IOTAServices
  Interface exposing general IDE info like product identifier, application folder, bin folder, installed languages

- IOTAToDoServices
  Interface to access the ToDo items for a module

- IOTAWizardServices
  Interface to access and extend the IDE repository

- IOTAHighlightServices
  Interface to access the syntax highlighters and to add custom syntax highlighter interfaces IOTAHighlighter

- IOTAPersonalityServices
  Interface to access installed IDE personalities, file extension associations with personalities

- IOTACompileServices
  Interface to the compiler allow to start and stop compilation of projects and handle notifications when compiling is done.

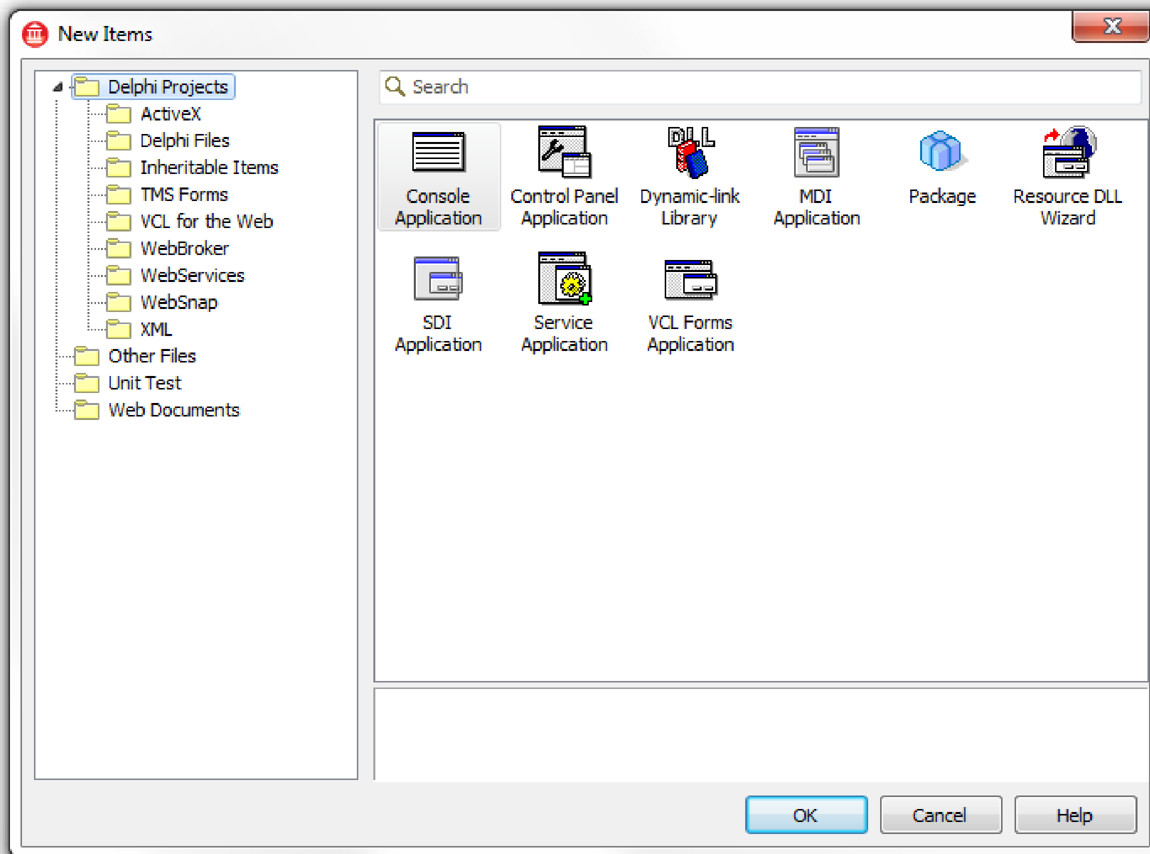
GETTING STARTED TO CREATE AN IDE PLUGIN

IDE plugins are compiled packages installed into the IDE. This means the IDE will load the package during startup. The package can perform its initialization both with code executed in the Initialization section of the units in the package as well as providing a Register method from where classes are registered with the IDE. This is very similar to a package that installs a component in the IDE where the Register procedure is called by the IDE from where the RegisterComponents() call registers the new components in the IDE. To install an IDE plugin or extension, the minimum that needs to be done is creating a class of the type TNotifierObject that implements the IOTAWizard interface or one of its descendents and register it:

```
unit MyIDEPlugin;

interface

uses
  Classes, ToolsAPI;


  TMyIDEWizard = class(TNotifierObject, IOTAWizard)
  // implement interfaces here
  end;


implementation

//register with the IDE:

procedure Register;
begin
  RegisterPackageWizard(TMyIDEWizard.Create);
end;
```

It is the IDE that will then call the appropriate interface methods when needed.

EXTENDING THE IDE WITH CUSTOM REPOSITORY WIZARDS



Our first in depth look at extending the IDE is about creating custom repository wizards. To create a plugin that will extend the repository, it is necessary to write a class that implements the IOTAWizard interface and the IOTARepositoryWizard interface. The IOTARepositoryWizard interface has 3 versions. It was extended in Delphi 7 to enable differentiating between repository items for VCL or CLX projects. It was extended again in Delphi 2005 where hierarchical categories of repository items were introduced. Note that it is only necessary to implement the latest IOTARepositoryWizard80 interface if you want to take advantage of the new features. The original IOTARepositoryWizard interface works fine in Delphi XE. For this example, we will take advantage of the categories exposed in the IOTARepositoryWizard80 interface. As such, we will write a class descending from TNotifierObject that implements the IOTAWizard interface and the IOTARepositoryWizard.

```
TMyProjectWizard = class(TNotifierObject, IOTAWizard,
IOTARepositoryWizard80)
  // implement interfaces here
  end;
```

This class will then be registered with the IDE:

```
procedure Register;
begin
  RegisterPackageWizard(TMyProjectWizard.Create);
end;
```

The next step is to implement the interfaces. The IOTAWizard interface informs the IDE of the plugin name, ID and provide a method to execute the plugin. The IOTARepositoryWizard interface provides the IDE with information about the repository item, the author, its glyph and also the category and personality for which the repository item is provided.

The class definition becomes:

```
 TMyProjectWizard = class(TNotifierObject, IOTAWizard,
IOTARepositoryWizard80)
  public
    // IOTAWizard
    function GetIDString: string;
    function GetName: string;
    function GetState: TWizardState;
    procedure Execute;

    // IOTARepositoryWizard
    function GetAuthor : string;
    function GetComment : string;
    function GetPage : string;
    function GetGlyph: Cardinal;

    // IOTARepositoryWizard80
    function GetGalleryCategory: IOTAGalleryCategory;
    function GetPersonality: string;
    function GetDesigner: string;
```

```
  end;
```

The full implementation of the wizard class methods can be found in sample 1.

One part of the interface informs the IDE about the new repository item, the Execute method will be called when the user clicks that repository item in the IDE. It is in this Execute method that the plugin needs to take the necessary steps to create the source files for the selected repository item. In the Execute method, we grab the BorlandIDEServices global variable and use the IOTAModuleServices interface to query the default unit, class, and filename that the IDE proposes for creating a new instance of the repository item. With this unit, class and filename, the IOTAModuleServices CreateModule method is then used to create the actual new module that is added to the active project.

The Execute method

```
procedure TTMSFormWizard.Execute;
var
  LProj: IOTAProject;
begin
  if Assigned(BorlandIDEServices) then
  begin
    // use the IOTAModuleServices interface to query a new default
unit, class & filename
    (BorlandIDEServices as
IOTAModuleServices).GetNewModuleAndClassName('', FUnitIdent,
FClassName, FFileName);

    FClassName := SFormName + Copy(FUnitIdent, 5,
Length(FUnitIdent));
    LProj := GetActiveProject;
    if LProj <> nil then
    begin
      (BorlandIDEServices as
IOTAModuleServices).CreateModule(TMyUnitCreator.Create(LProj,
FUnitIdent, FClassName, FFileName));
    end;
  end;
end;
```

The module creator is a class that implements the IOTACreator, IOTAModuleCreator interfaces. Via these interfaces, the actual source code file and form file can be created.

```
  TMyUnitCreator = class (TNotifierObject, IOTACreator,
IOTAModuleCreator)
  public
    // IOTACreator
    function GetCreatorType: string;
    function GetExisting: Boolean;
    function GetFileSystem: string;
    function GetOwner: IOTAModule;
    function GetUnnamed: Boolean;

    // IOTAModuleCreator
    function GetAncestorName: string;
    function GetImplFileName: string;
    function GetIntfFileName: string;
    function GetFormName: string;
    function GetMainForm: Boolean;
    function GetShowForm: Boolean;
    function GetShowSource: Boolean;
    function NewFormFile(const FormIdent, AncestorIdent: string):
IOTAFile;
    function NewImplSource(const ModuleIdent, FormIdent,
AncestorIdent: string): IOTAFile;
    function NewIntfSource(const ModuleIdent, FormIdent,
AncestorIdent: string): IOTAFile;
    procedure FormCreated(const FormEditor: IOTAFormEditor);
```

The most important methods in this interface are the NewFormFile and the NewImplSource methods. These functions should return a class implementing the IOTAFile interface that will return the actual code for the .DFM form file and the .PAS source code file. The NewIntfSource function is not used for Delphi forms or projects, only for C++ header files.

The IOTAFile interface is an interface that simply returns the content of the .DFM or .PAS file as a string and the file age as TDateTime.

```
  TCodeFile = class(TInterfacedObject, IOTAFile)
  protected
    function GetSource: string;
```

```
    function GetAge: TDateTime;
  end;
```

Note that it is important that the .DFM and .PAS file returned contain valid code. The IDE will try to parse the form file and source code and should the parsing fail, the IDE will not create the file.

In the sample, the form unit & DFM file are stored in a resource file created with BRCC32 from the .RC file:

TCompanyFormSRC 10 " Unit1.pas"
TCompanyFormFRM 10 " Unit1.DFM"

and the IOTAFile implementing class retrieves the source code + DFM file from the resource, replaces the form name, form class, unit name with the new name and returns it via the GetSource function:

```
function TUnitFile.GetSource: string;
var
  Text, ResName: AnsiString;
  ResInstance: THandle;
  HRes: HRSRC;
begin
  Resname := AnsiString(SCodeResName);
  ResInstance := FindResourceHInstance(HInstance);
  HRes := FindResourceA(ResInstance, PAnsiChar(ResName),
PAnsiChar(10));
  Text := PAnsiChar(LockResource(LoadResource(ResInstance,
HRes)));
  SetLength(Text, SizeOfResource(ResInstance, HRes));
  Result := Format(string(Text), [FModuleName, FFormName,
FAncestorName]);
end;
```

EXTENDING THE IDE WITH CUSTOM DOCKING PANELS

Creating a custom dock panel for the IDE is a different type of IDE extension and we do not need to register a IOTAWizard interface implementing class with the IDE. Instead, the

plugin should create an instance of a form, provide a way for the user to show or hide the docking panel, persist its state with the IDE desktop, and, finally, destroy the docking panel upon exiting the IDE. The class used for the IDE docking panel is implemented in the units DockForm, DockToolForm that are part of the DesignIDE package. Two different types exist: TDockableForm and TDockableToolBarForm. We will need to write a Register procedure in a unit of the plugin package. This Register procedure will be called by the IDE after loading the package. From this Register procedure, we can create the docking panel and insert a new menu item in the IDE to show/hide the panel. The code in the finalization section of the unit will be called when the IDE is closed and thus, the panel can be destroyed in this phase.

The skeleton for the unit to create the custom docking panel and destroying it is as such:

```
unit MyIDEDockPanel;

interface

procedure Register;

implementation

uses
  MyDockForm;

var
  MyIDEDockForm: TMyDockForm;

procedure Register;
begin
  if MyIDEDockForm = nil then
  begin
    MyIDEDockForm := TMyIDEDockForm.Create(nil);
  end;
end;


finalization
  MyIDEDockForm.Free;

end.
```

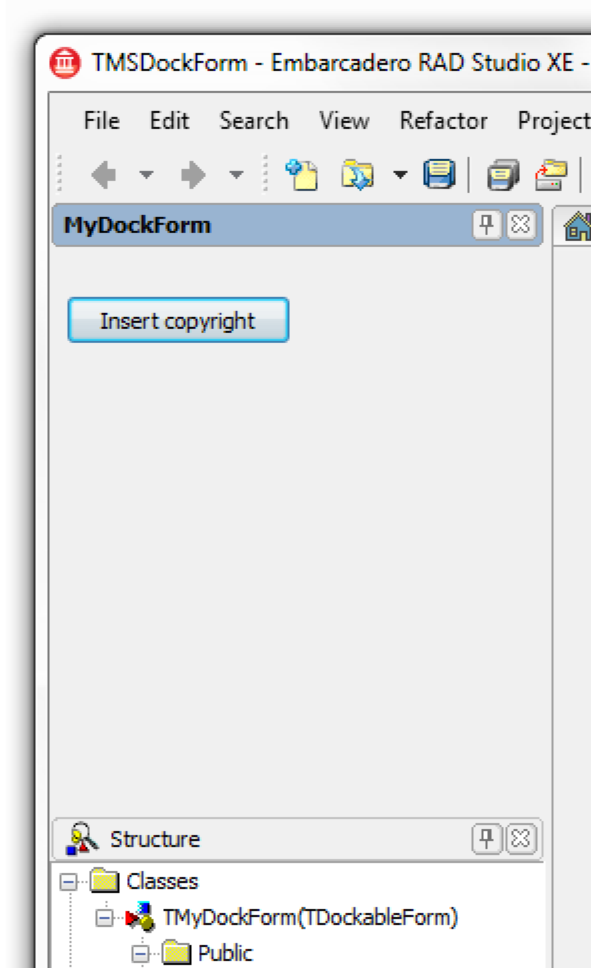
TMyDockForm is a class that descends either from TDockableForm and TDockableToolBarForm. To play well with the IDE desktop persisting functionality, it is required to initialize the DeskSection, AutoSave and SaveStateNecessary properties of the base class:

```
constructor TMyIDEDockForm.Create(AOwner: TComponent);
begin
  inherited;
  DeskSection := Name;
  AutoSave := True;
  SaveStateNecessary := True;
end;

destructor TMyIDEDockForm.Destroy;
begin
  SaveStateNecessary := True;
  inherited;
end;
```

In addition, the custom docking panel class should be registered with the IDE desktop with the code:

```
RegisterDesktopFormClass(TMyIDEDockForm,
MyIDEDockForm.Name,MyIDEDockForm.Name);

if @RegisterFieldAddress <> nil then
  RegisterFieldAddress(MyIDEDockForm.Name, @MyIDEDockForm);
```

The result of the docking panel plugin is:



Full code of the docking panel plugin can be found in the code download in the folder DockForm.

Accessing the Delphi code editor

Access to the IDE code editor is made possible via IOTAEditorServices interface that is implemented in the BorlandIDEServices global variable. The IOTAEditorServices provides access to the active editor view via IOTAEditorServices.TopView: IOTAEditView. The IOTAEditView interface provides access to editor bookmarks, cursor position, scrolling, etc. In turn, this IOTAEditView provides access to the editor buffer. The buffer exposes the interface IOTAEditBuffer. This IOTAEditBuffer interface allows manipulation of text, for example insertion and deletion.

This code snippet will grab the IOTAEditorServices from BorlandIDEServices, get the IOTAEditView interface and access the IOTAEditBuffer to insert text at the top of the source code file in the active editor window:

```
var
  EditorServices: IOTAEditorServices;
  EditView: IOTAEditView;
  copyright: string;

  copyright := 'Copyright © 2011 by tmssoftware.com';

  EditorServices := BorlandIDEServices as IOTAEditorServices;

  EditView := EditorServices.TopView;

  if Assigned(EditView) then
  begin
    // position cursor at 1,1
    EditView.Buffer.EditPosition.Move(1,1);
    // insert copyright notice on top
    EditView.Buffer.EditPosition.InsertText(copyright);
  end;
```

Extending the Delphi IDE menu

The Delphi IDE provides the IOTAMenuWizard interface to add items the Delphi IDE main menu. The limitation of this interface is that all menu items added this way will be organized under the help menu. We will offer an alternative method to insert new menu items anywhere in the existing Delphi IDE main menu.

To add a new menu item via IOTAMenuWizard, create a class that descends from TNotifierObject and implements IOTAWizard, IOTAMenuWizard:

```
 TMyMenuItem = class(TNotifierObject, IOTAWizard, IOTAMenuWizard)
    function GetIDString: string;
    function GetName: string;
    function GetState: TWizardState;
    procedure Execute;
    function GetMenuText: string;
  end;
```

and register this class with the IDE via the unit Register procedure:

```
RegisterPackageWizard(TMyMenuItem.Create);
```

When the menu item is clicked in the IDE, Delphi will call the Execute method from where your custom action can be performed.

Alternatively, it is possible to get access to the Delphi IDE main menu as a TMainMenu class and use the common TMainMenu methods to insert a TMenuItem instance in this menu.

To do this, use the INTAServices40 interface implemented in BorlandIDEServices and call its function MainMenu that returns a TMainMenu instance.

```
var
  NTAServices : INTAServices40;
  mnuitem: TMenuItem;
  mnuitem := TMenuItem.Create(nil);
  mnuitem.Caption := 'New item';

  NTAServices := BorlandIDEServices as INTAServices40;
  NTAServices.MainMenu.Items.Add(mnuitem);
```

When the first technique with the IOTAMenuWizard is used, the IDE will automatically remove the menu item when the plugin is uninstalled. Using the second technique, we'll need to remove the menu item in code. As the interface is based on TMainMenu, call Items.Remove( ) to remove the item from the menu and destroy it as well as the object that handles the menu click. The full code to add and remove the menu item becomes:

```
procedure AddIDEMenu;
```

```
var
  NTAServices: INTAServices40;

begin
  NTAServices := BorlandIDEServices as INTAServices40;

  // avoid inserting twice
  if NTAServices.MainMenu.Items[5].Find('INTAServices40Menu') =
nil then
 begin
    CustomMenuHandler := TCustomMenuHandler.Create;
    mnuitem := TMenuItem.Create(nil);
    mnuitem.Caption := 'INTAServices40Menu';
    mnuitem.OnClick := CustomMenuHandler.HandleClick;
    NTAServices.MainMenu.Items[5].Add(mnuitem)
  end;
end;



procedure RemoveIDEMenu;
var
  NTAServices: INTAServices40;

begin
  if Assigned(mnuitem) then
  begin
    NTAServices := BorlandIDEServices as INTAServices40;
    NTAServices.MainMenu.Items[5].Remove(mnuitem);
    mnuitem.Free;
    if Assigned(CustomMenuHandler) then
      CustomMenuHandler.Free;
  end;
end;
```

The full sample for extending the IDE main menu can be found in the code download in the IDEMenu folder.

EXTENDING THE DELPHI PROJECT MANAGER CONTEXT MENU

In this section, we will access to the IDE project manager, its context menu, and the projects & files opened in the project manager. Furthermore, we will extend the context menu with custom actions. To start, we will make use of the IOTAProjectManager interface, available in the BorlandIDEServices global variable. The IOTAProjectManager interface exposes the function AddMenuItemCreatorNotifier that needs to be called to pass an instance of a class descending from TNotifierObject and implementing the IOTAProjectMenuItemCreatorNotifier interface. Basically, this informs the IDE that before it shows the project manager context menu, that it should query our plugin if one or more custom context menu items should be added.

The class for the context menu item creator is:

```
 TMyProjectContextMenu = class(TNotifierObject,
IOTAProjectMenuItemCreatorNotifier)

procedure AddMenu(const Project: IOTAProject; const IdentList:
TStrings; const ProjectManagerMenuList: IInterfaceList;
IsMultiSelect: Boolean);
end;
```

Via the parameter IOTAProject, the code can determine for which project the context menu is shown and via the parameter ProjectManagerMenuList, instances of a TMyProjectContextMenuLocal class can be added. In this sample code snippet, a context menu item is unconditionally added, regardless of of which item is right-clicked in the project manager:

```
procedure TMyProjectContextMenu.AddMenu(const Project:
IOTAProject; const IdentList: TStrings; const
ProjectManagerMenuList: IInterfaceList; IsMultiSelect: Boolean);
var
  MnuItem: TMyProjectContextMenuLocal;
begin
  MnuItem := TMyProjectContextMenuLocal.Create;
  ProjectManagerMenuList.Add(MnuItem)
end;
```

The IdentList is a stringlist holding string identifiers of what item type is right clicked. This is declared in TOOLSAPI.PAS and can be:

```
  sBaseContainer = 'BaseContainer';
  sFileContainer = 'FileContainer';
  sProjectContainer = 'ProjectContainer';
  sProjectGroupContainer = 'ProjectGroupContainer';
  sCategoryContainer = 'CategoryContainer';
  sDirectoryContainer = 'DirectoryContainer';
  sReferencesContainer = 'References';
  sContainsContainer = 'Contains';
  sRequiresContainer = 'Requires';
```

If the context menu item should only appear when the project group is right-clicked, the code would be:

```
procedure TMyProjectContextMenu.AddMenu(const Project:
IOTAProject; const IdentList: TStrings; const
ProjectManagerMenuList: IInterfaceList; IsMultiSelect: Boolean);
var
  MnuItem: TMyProjectContextMenuLocal;
begin
  if (IdentList.IndexOf(sProjectGroupContainer) <> -1) then
  begin
    MnuItem := TMyProjectContextMenuLocal.Create;
    // Set menu item properties here
    MnuItem.OnExecute := MenuClickHandler;
    ProjectManagerMenuList.Add(MnuItem)
  end;
end;
```

The TMyProjectContextMenuLocal is a class descending from TProjectContextMenuLocal and should implement the interfaces IOTALocalMenu and IOTAProjectManagerMenu. This interface consists of methods:

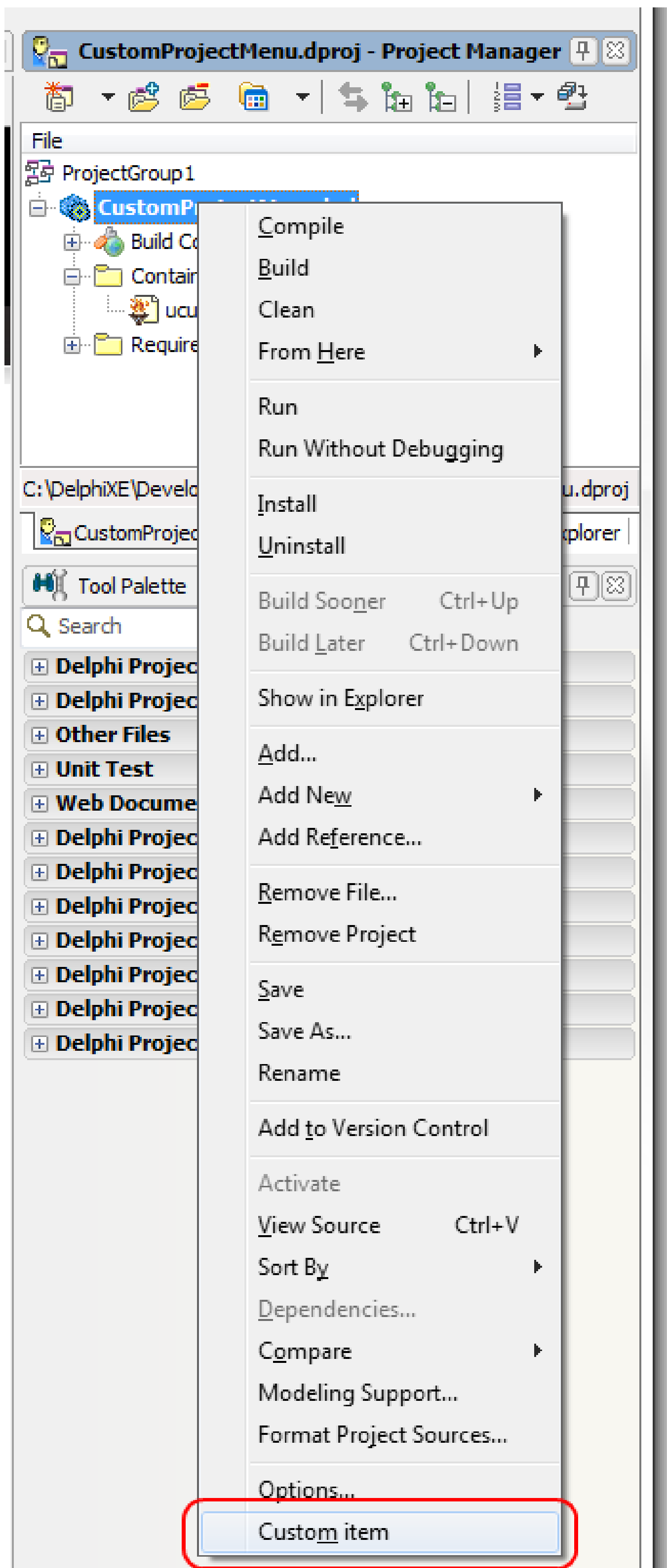
```
  TMyProjectContextMenuItem = class(TProjectContextMenuLocal,
IOTALocalMenu, IOTAProjectManagerMenu)
  public
    // IOTALocalMenu
```

```
    function GetCaption: string;
    function GetChecked: Boolean;
    function GetEnabled: Boolean;
    // IOTAProjectManagerMenu interface
  function GetIsMultiSelectable: Boolean;
    procedure SetIsMultiSelectable(Value: Boolean);
    procedure Execute(const MenuContextList: IInterfaceList);
overload;
    function PreExecute(const MenuContextList: IInterfaceList):
Boolean;
    function PostExecute(const MenuContextList: IInterfaceList):
Boolean;
    property IsMultiSelectable: Boolean read GetIsMultiSelectable
write SetIsMultiSelectable;
  end;
```

With the IOTALocalMenu, we can set the caption, checked state, and enabled state of the menu item. With the IOTAProjectManagerMenu interface, we can define whether or not the context menu item supports executing on multiple selected items in the project manager. The methods PreExecute, Execute, PostExecute are called, respectively, before actual execution of the Execute, when the item is selected, and after the actual execution of Execute. The parameter of the Execute methods is the MenuContextList that is a list of items selected in the project manager.

From the Execute method, the project opened for which the context menu item was selected is retrieved with following code:

```
procedure TMyProjectContextMenuLocal.Execute(const
MenuContextList: IInterfaceList);
var
  MenuContext: IOTAProjectMenuContext;
  Project: IOTAProject;
begin
  MenuContext := MenuContextList.Items[0] as
IOTAProjectMenuContext;
  Project := MenuContext.Project;
end;
```


CustomProjectMenu.dproj - Project Manager
File
ProjectGroup1
Compile
Build
Clean
From Here
Run
Run Without Debugging
Install
Uninstall
Build Sooner Ctrl+Up
Build Later Ctrl+Down
Show in Explorer
Add...
Add New
Add Reference...
Remove File...
Remove Project
Save
Save As...
Rename
Add to Version Control
Activate
View Source Ctrl+V
Sort By
Dependencies...
Compare
Modeling Support...
Format Project Sources...
Options...
Custom item
Tool Palette
Search
Other Files
Unit Test

The full sample for adding a project manager context menu item can be found in the code download in the ProjMenu folder.

ADDING AN IDE EDITOR CONTEXT MENU ITEM AND GRABBING THE SELECTED TEXT

If your plugin wants to offer some processing on selected text in the editor, you may want to add a custom item in the IDE editor context menu that, when clicked, grabs the selected text and processes it. To add such a context menu, we need to get access to the menu, which, in turn, first need access to the editor. Please note that the editor is not immediately available upon startup of the IDE, therefore we can't get access to the editor in the plugin initialization code. What we needed is to register a class that implements the INTAEditServicesNotifier interface. The IDE calls this interface when the editor is activated in the IDE. At that point, the plugin code can be sure that the editor instance exists. The INTAEditServicesNotifier interface offers several methods of which only EditorViewActivated is of interest. The interface of this INTAEditServicesNotifier implementing class is:

```
TEditNotifierHelper = class(TNotifierObject, IOTANotifier,
INTAEditServicesNotifier)

    procedure WindowShow(const EditWindow: INTAEditWindow; Show,
LoadedFromDesktop: Boolean);

    procedure WindowNotification(const EditWindow: INTAEditWindow;
Operation: TOperation);

    procedure WindowActivated(const EditWindow: INTAEditWindow);

    procedure WindowCommand(const EditWindow: INTAEditWindow;
Command, Param: Integer; var Handled: Boolean);

    procedure EditorViewActivated(const EditWindow:
INTAEditWindow; const EditView: IOTAEditView);

    procedure EditorViewModified(const EditWindow: INTAEditWindow;
const EditView: IOTAEditView);

    procedure DockFormVisibleChanged(const EditWindow:
INTAEditWindow; DockForm: TDockableForm);

    procedure DockFormUpdated(const EditWindow: INTAEditWindow;
DockForm: TDockableForm);
```

```
    procedure DockFormRefresh(const EditWindow: INTAEditWindow;
DockForm: TDockableForm);

  end;
```

This notifier class is registered with the IDE via the code:

```
procedure Register;
var
  Services: IOTAEditorServices;
begin
  Services := BorlandIDEServices as IOTAEditorServices;
  NotifierIndex :=
Services.AddNotifier(TEditNotifierHelper.Create);
end;
```

The AddNotifier function returns a unique index with which the class is registered. This NotifierIndex variable in the unit needs to be used to unregister the notifier class again when the plugin is uninstalled. As such, we need to perform this unregister in the finalization section of the unit:

```
procedure RemoveNotifier;
var
  Services: IOTAEditorServices;

begin
  if NotifierIndex <> -1 then
  begin
    Services := BorlandIDEServices as IOTAEditorServices;
    Services.RemoveNotifier(NotifierIndex);
    NotifierIndex := -1;
  end;
end;

finalization
  RemoveNotifier;
end.
```

With this notifier installed, it is now possible to get access to the IDE editor context menu when the editor first becomes active and install our custom menu item. The code used to do this is:

```
var
  custommenu: TMenuItem;

procedure TEditNotifierHelper.EditorViewActivated(const
EditWindow: INTAEditWindow; const EditView: IOTAEditView);
begin
  if not Assigned(custommenu) then
  begin
    AddEditContextMenu;
  end;
end;

procedure AddEditContextMenu;
var
  editview: IOTAEditView140;
  popupmenu: TPopupMenu;

begin
  editview := (BorlandIDEServices as IOTAEditorServices).TopView;
  popupmenu :=
editview.GetEditWindow.Form.FindComponent('EditorLocalMenu') as
TPopupMenu;
  custommenu := TMenuItem.Create(nil);
  custommenu.Caption := 'Custom context menu item';
  custommenu.OnClick := mnuHandler.MenuHandler;
  popupmenu.Items.Add(custommenu);
end;


initialization
  custommenu := nil;

finalization
  custommenu.Free;
end.
```

To handle the click on the editor context menu item, a class with the method MenuHandler is created:

```
  TMenuHandler = class(TComponent)
    procedure MenuHandler(Sender: TObject);
  end;
```

and this MenuHandler() method can get access to the editor view and get the selected text with:

```
procedure TMenuHandler.MenuHandler(Sender: TObject);
var
  editview: IOTAEditView140;
  editblock: IOTAEditBlock;

begin
  editview := (BorlandIDEServices as IOTAEditorServices).TopView;

  // get the selected text in the edit view
 editblock := editview.GetBlock;

  ShowMessage('Context menu click: ' +
inttostr(editblock.StartingColumn)+':'+inttostr(editblock.Starting
Row) + ' - ' +
inttostr(editblock.EndingColumn)+':'+inttostr(editblock.EndingRow)
);

  // if there is a selection of text, get it via editblock.Text

  if (editblock.StartingColumn <> editblock.EndingColumn) or
(editblock.StartingRow <> editblock.EndingRow) then
    ShowMessage('Selected text: ' + editblock.Text);
end;
```

The full sample for adding an editor context menu item can be found in the code download in the EditorContextMenu folder.

Adding information on the Delphi splash screen

To give the IDE plugin we create a finishing touch and to inform the user it is properly installed in the IDE, we can add some information to the splash screen during the startup of the IDE. The TOOLSAPI unit exposes the SplashScreenServices IOTASplashScreenServices interface. This interface offers the method AddPluginBitmap:

```
  procedure AddPluginBitmap(const ACaption: string; ABitmap:
HBITMAP; AIsUnRegistered: Boolean = False; const ALicenseStatus:
string = ''; const ASKUName: string = '');
```

The parameters are:

- ACaption : text to appear on the splash screen
- ABItmap: bitmap handle to show in the splash screen associated with the plugin for a 24x24 bitmap
- AIsUnRegisterd: this is a boolean parameter indicating whether the text should appear in red font for unregistered products or regular white text for registered products.
- ALicenseStatus: this is a text that can display for example 'Trial' or 'Registered'.
- ASKUName: this text can show the name of the SKU if different SKUs exist for the plugin.

In the initialization section of a unit within a package loaded by the IDE, we can use this interface to add custom information to the splash screen while the IDE is loading:

```
procedure AddSplashText;

var
  bmp: TBitmap;
begin
  bmp := TBitmap.Create;
  bmp.LoadFromResourceName(HInstance, 'PLUGINBITMAPRESOURCE');
  SplashScreenServices.AddPluginBitmap('Plugin product XYZ © 2011
by MyCompany',bmp.Handle,false,'Registered','');
  bmp.Free;
end;


initialization
   AddSplashText;
end.
```

The full sample for adding an entry in the IDE splash screen can be found in the code download in the SplashScreen folder.

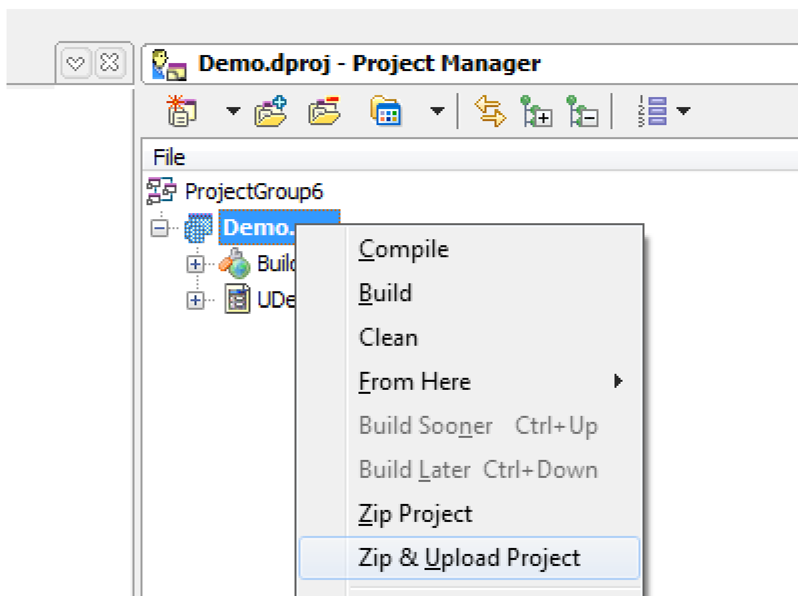
FREE TMS IDE PLUGINS FOR DELPHI XE

Based on the techniques presented in this whitepaper, we offer a couple of free IDE plugins for Delphi XE. The free plugins can be found in the folder "TMS plugins for Delphi XE" in the code download.

TMS PROJECT MANAGER PLUGIN

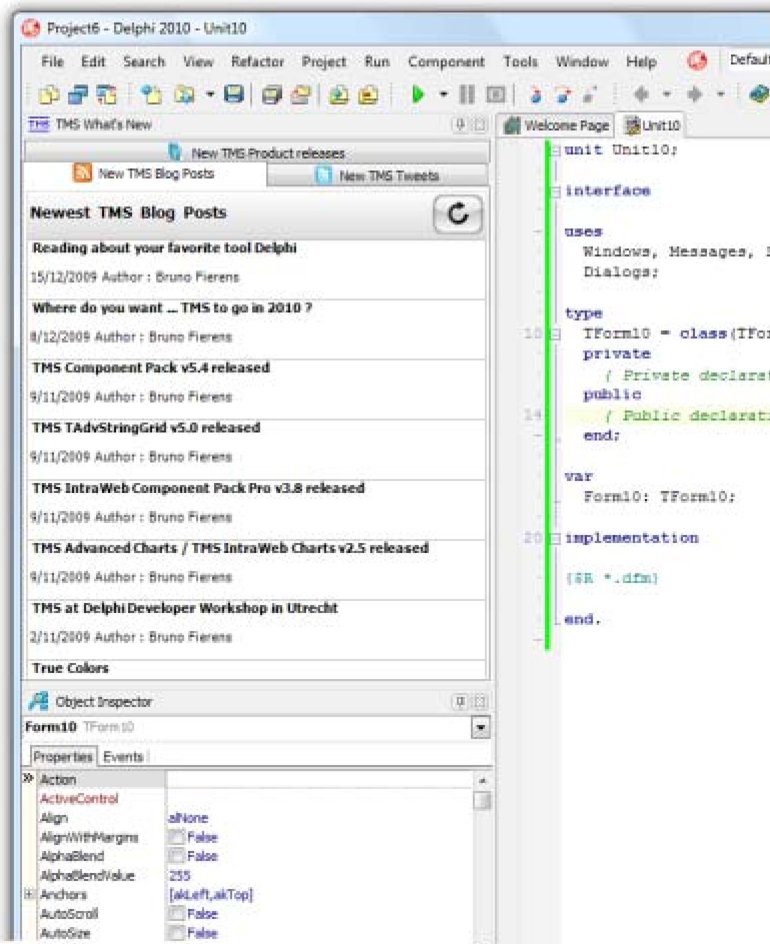
This plugin uses the project manager context menu extension and the IDE menu extension to offer access to the plugin options. It adds a context menu to the Project Manager with 3 new options:

1) ZIP project: this allows to create a ZIP file containing the project files. In the plugin options, it can be configured what file types to include in a project.
2) ZIP & Email project: this option will try to use the default email client to send the project ZIP file by email.
3) ZIP & Upload project: this option will try to upload the project ZIP file to an FTP server defined in the plugin options



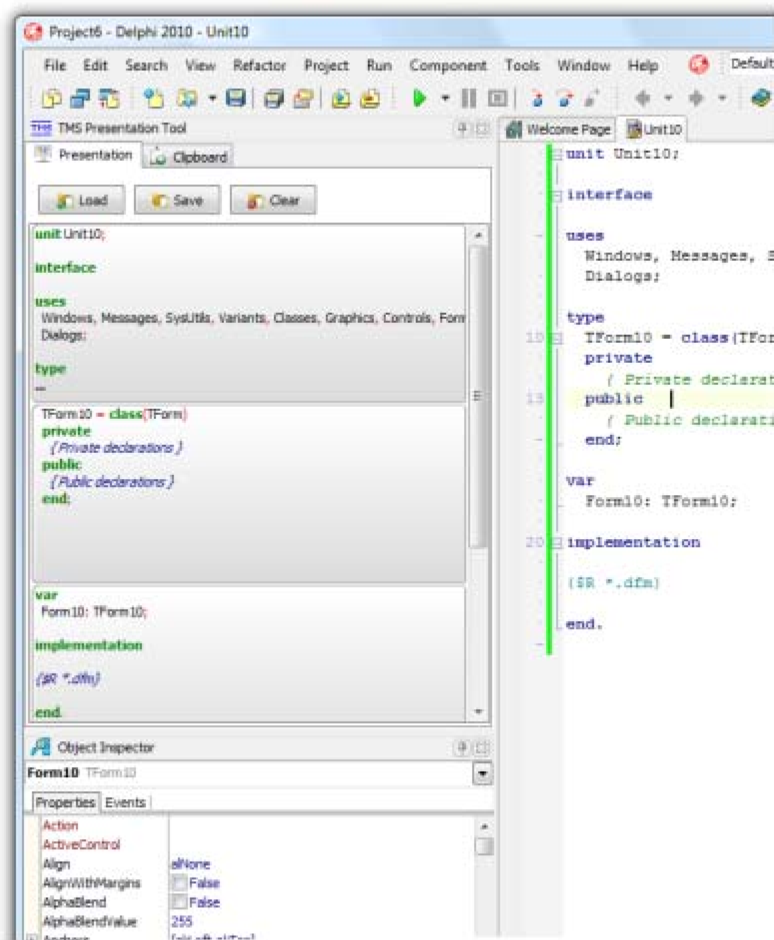
TMS WHAT'S NEW PLUGIN

This plugin builds on the technique to create a custom IDE docking form. This docking form has three tabs. It displays the latest component releases from TMS software, the blog feed as well as the tweets from TMS software.



TMS PRESENTATION TOOL PLUGIN

The TMS Presentation tool plugin is also based on the custom IDE docking form capability as well as interface with the IDE editor. It offers two tab. In the first tab is a clipboard monitor. This shows all text that was put on the clipboard. Direct drag & drop from the docking form to the IDE editor is possible. The second tab is a list of saved code snippets. These code snippets can be used for example when giving presentation.



About the Author

Bruno studied civil electronic engineering at university of Ghent and started a career as R&D digital hardware engineer at Barco Graphics in Belgium. He founded TMS software in 1996, developing VCL components starting with Delphi 1. TMS software became Borland Technology Partner in 1998 and developed the Delphi Informant award-winning grid & scheduling components. In 2001, he started development on IntraWeb component and in 2003, ASP.NET components. Currently, Bruno is developing and managing VCL, Silverlight, ASP.NET and IntraWeb component development projects, as well as consulting and custom project development and management on Windows, Web and iPad with Delphi and XCode. His special areas of interest are user interfaces & hardware.

Embarcadero Technologies, Inc. is the leading provider of software tools that empower application developers and data management professionals to design, build, and run applications and databases more efficiently in heterogeneous IT environments. Over 90 of the Fortune 100 and an active community of more than three million users worldwide rely on Embarcadero's award-winning products to optimize costs, streamline compliance, and accelerate development and innovation. Founded in 1993, Embarcadero is headquartered in San Francisco with offices located around the world. Embarcadero is online at www.embarcadero.com.

Copyright © 2011 TMS software