



# Building Script Projects

---

## Summary

Tutorial

TU0125 (v1.0) November 30, 2004

This tutorial covers the general aspects of the Scripting System and the use of Borland Delphi Objects and DXP Object Models in scripts.

A Netlister, Board Outline Copier and Query Expression scripts are developed using the DXP Object Models to illustrate the power of the scripting system in the DXP application.

---

This tutorial builds on the **Getting Started with Scripting** tutorial and this tutorial is broken up into several sections; **Scripting Fundamentals** which cover the concepts you need to understand before you can script successfully, **Executing Scripts**, **Debugging Scripts**, **Using the Script Completion features** which illustrates the ways you can use to help you write scripts faster and more efficiently, **Using Borland Objects in your scripts**, **Using DXP Object Models in your scripts**, **Building a Netlister project**, **Building the Board Outline Copier project** and finally **Using Query Expressions** in scripts.

You will have a look at existing real world script examples. A netlist generator using Workspace Manager Object Model which generates a Protel netlist format. This netlister script is written using DelphiScript. A Board Outline Copier script which is written in VisualBasic Script, or VBScript for short, utilizes the PCB object model which copies the PCB board outline as tracks and arcs onto a specified layer on the PCB document.

Finally you will get the chance to experiment with query expression scripts that can be executed in the **Filter** panel that can search for specified objects on a PCB or Schematic document.

In this tutorial, the DelphiScript and VBScript language sets are used. The DelphiScript language set is based on Borland Delphi and VBScript is based on Microsoft Scripting technology. For information on DelphiScript and Object Pascal (used in Borland Delphi), please refer to the **Differences between DelphiScript and Object Pascal** in the DelphiScript section of the Scripting online help.



For more information on VBScript please refer to the Scripting documentation by Microsoft at <http://www.msdn.microsoft.com/library/default.asp?url=/library/en-us/script56/html/vtoriMicrosoftWindowsScriptTechnologies.asp>.

## Scripting Fundamentals

---

Before we can start writing or debugging scripts successfully, we need to define terms that are related to the scripting world within DXP.

- There are two Run Time Libraries in DXP which are used by the Scripting System - The DXP Run Time Library and its DXP Object Models and a subset of the Borland Delphi Run Time Library.
- DXP Object Interfaces are interfaces representing actual design objects in DXP which are part of the DXP Object model. You can use DXP Object Model to extract and modify data from design documents open in DXP from your scripts in DelphiScript, JScript, VBScript etc. The DXP Object Model is composed of Schematic Object Model, PCB Object Model, Workspace Manager Object Model and Client Object Model.
- You need to have the design document open first before you wish to run a script that uses DXP object models which is targetted towards a particular design document by a menu, key or toolbar button or via the *Run Process* dialog. For example to deal with PCB documents and objects, you need to use the PCB Object Model, and to deal with Projects and documents in general, you will need to use the Workspace Manager Object Model.
- Object Interfaces consist of methods and properties (but have no variables).
- You need to check the validity of object interfaces by either checking whether they are not Nil or using the Assigned function. For example `If Assigned(Board) = True Then // do something with the board which is a IPCB_Board type.`

## Script Units and Forms

- A script form has two views – the script **code** view and the script **form** view. The script code view contains event handlers and standalone functions. The script form view represents a dialog form (of different types) and have controls and their associated event handlers on it.
- Components are visual or non visual control objects on the *Toolbox* panel (which is a subset of the Borland Delphi™ Component Palette) that you can manipulate on a script form at design time using the *Object Inspector* panel. A component is a control that accepts the user's input such as a mouse click or a sequence of characters entered.
- A component has methods, properties, and events. Methods are the actions an object can perform. Properties represent the data contained in the object interface and properties can be accessed or modified. Events are conditions a component on a script form can react to. There are event handlers in a script form that process captured conditions.
- Components (although they are developed in Borland Delphi™) can be used in any script form in any language set supported by the DXP Scripting system. Therefore all the script forms end with a DFM file extension which denotes they are based on the Borland Delphi™ forms.



You can refer to the **DXP RTL Reference** online help and the **Supported Borland Delphi Units** section within for more information on DXP Run Time Libraries and the supported Borland Delphi Run Time Library for the scripting system.

## Executing scripts

A project helps you manage your scripts much easier and there are two script types; unit and form scripts, please refer to the **Getting Started with Scripting** tutorial and the **Scripting Online Help** for more information. Any script (using the same language set) within a project have access to global variables and procedures thus it is very important to have unique procedure and global variable names.

Use unique names for procedures/routines and global variables for scripts in the same project.

Basically to run a script that needs access to DXP object interfaces, you need to run the script on the currently focussed document in DXP. For example, if your script has calls to PCB design objects and you attempt to run the script in the text editor, you will get undeclared identifier errors. Thus you need to run the script on an opened PCB document. It is imperative to check for the presence of a PCB document first before proceeding with PCB calls from a script.

You can install script projects in the **Installed Projects** list, so that these projects are seen as global to DXP which means the global variables and procedures from these scripts are available to use in other script projects open in DXP. From the **DXP » Preferences** menu, and drill down to **Scripting System** folder within the DXP System on the left pane of the *Preferences* dialog to install scripts in the *Installed Projects* list.



You can refer to the script examples, mostly in DelphiScript, in the `\Examples\Scripts\` folder which are available for experimentation.

### Running a script from the RunProcess dialog

To run a script from the *Run Process* dialog (**DXP » Run Process**) There are two parameters in this case: the `ProjectName` and the `ProcName`. For the `ProcName` parameter, you need to specify the script filename and the main procedure in this script. So the format is as follows: `ProcName = ScriptFileName>ProcedureName`. Note the **GreaterThan** symbol used between the script file name and the procedure name.

To execute a script form the Run Process dialog;

**Process:** `ScriptingSystem:RunScript`

**Parameters:** `ProjectName = PathToProject | ProcName = ScriptFileName > ProcedureName`

Where the `ProjectName` parameter contains the full path to a project with a `PrjScr` extension and the `ProcName` parameter contains two parts separated by the greater than symbol; the `ScriptFileName > ProcedureName`

Note (Parameter = Value) blocks are separated by a Pipe (Vertical Bar) symbol eg `Param1 = Val1 | Param2 = Val2`

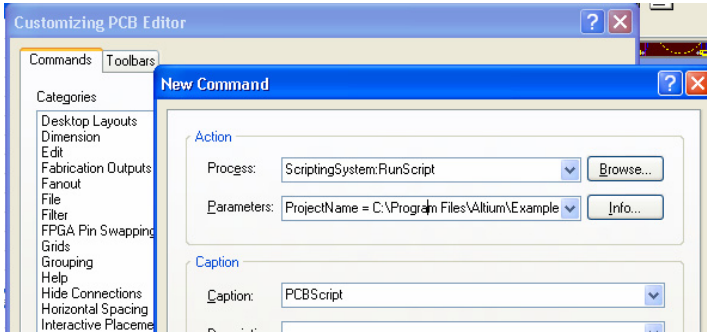
### Running a script from a DXP resource (key, toolbar or menu)

You have the ability to assign a script to a server menu, toolbar or hot key in DXP which makes it possible for you to run the script over a current PCB document for example.

1. Double click on the menu and the *Customizing ServerName Editor* dialog appears.
2. Click on the **New** button of the *Customizing ServerName Editor* dialog. The New Command dialog appears.

## Building Script Projects

3. Choose **ScriptingSystem:RunScript** process in the **Process:** field of the *Customizing ServerName Editor* dialog.
4. Define the `ProjectName` and `ProcName` parameters in the **Parameters:** field of the *Customizing ServerName Editor* dialog.



5. You will need to give a name to this new command and assign a new icon in the **Caption:** field of this dialog. The new commands appear in the **[Custom]** category of the **Categories** list. Click on the **[Custom]** entry from the **Categories** list. The **new** command appears in the **Commands** list of this dialog.
6. You then need to drag the new command onto the menu from the *Customizing ServerName Editor* dialog. The command appears on the menu. You can then click on this new command and the HelloWorld form appears.



You can refer to the **Getting Started with Scripting** tutorial which is available in the Scripting Online help in DXP and in the \Help folder of the DXP installation directory.

## Debugging a script

---

There are various tools in the scripting system to help you debug your scripts; using break points in your script, using the **Watches List** panel to monitor the variable values, using the bookmarks in the script to jump around more efficiently, and using step in or step out facilities to step through the scripts. There are a few tips below.

Basically when a script contains an error, the script is stopped by the debugger. Stop the script using the stop button in the debugger and make changes to the script. Save this script and then you can re-start the script.

To check the values of the variables in your scripts, you can utilise the following features in the scripting system;

- Modal dialogs (such as ShowMessage and MsgBox functions)
- Messages panel
- Breakpoints
- Expression Evaluation tooltips
- Symbol Insight tool tips
- Evaluate dialog



You can refer to the **A Tour of the Scripting System** section within the Scripting Online help for more details.

### ShowMessage (DelphiScript) and MsgBox (VBScript) functions

The **ShowMessage ShowInfo and ShowError** procedures from DXP Run Time Library (part of DXP application and exposed through the Scripting System) can be used as a simple tool for displaying data values or to track the sequence of events of your script in any language set you use. VBScript has built in functions that you can use in your VBScript projects and one of the common functions is the **MsgBox** function.

#### Example

```
Procedure HelloWorld;
Begin
    ShowMessage('Hello world!');
End;
```

### The Messages Panel (Workspace Manager object)

The *Messages* panel is a very useful debugging tool for scripting. You can use the *Messages* panel as a debugging dialog that shows states of variables and properties from your script. An example of using the *Message* Panel in your script to capture data values or states is shown below.

## Building Script Projects

### Example

```
Var
    WSM : IWorkspace;
Begin
    WSM := GetWorkspace;
    If WSM = Nil Then Exit;
    // Clear out messages and initialize the Message panel
    WSM.DM_ClearMessages;
    WSM.DM_MessageViewBeginUpdate;
    WSM.DM_ShowMessageView;

    // Display states or values of data with DM_AddMessage procedures
    WSM.DM_AddMessage('MessageClass 1', 'MessageText 1',
        'DXP Message 1', '1', '', '', 3, False);
    WSM.DM_AddMessage('MessageClass 2', 'MessageText 2',
        'DXP Message 2', '2', '', '', 3, False);
    WSM.DM_MessageViewEndUpdate;
    WSM.DM_ShowMessageView;
End;
```

## Trapping errors in your DelphiScript scripts

The use of **Try Finally End** and **Try Except End** blocks are useful to catch exceptions in your DelphiScript scripts. The major limitation of a **Try Except End** block is that you cannot define custom exceptions but only use a text string for the **Raise** statement.

### Example

```
Try
    X := Y/Z;
Except
    Raise('A divide by zero error!');
End;
```

The **Raise** keyword is related to the **Try** keyword. The **Raise** keyword can be used without parameters to re-raise the last exception. It can also be used with a string parameter to raise an exception using a specific message. An example is

```
Raise(Format('Invalid Value Entered : %d', [Height]));
```

Note, the **On** keyword is not supported, thus you cannot use **Exception** objects as in Borland Delphi.

## Trapping errors in your VB scripts

The use of On Error statement can be used in your VB Scripts. Consult the Scripting documentation from the MSDN website ([www.msdn.com](http://www.msdn.com)) for more information on trapping errors.

## Using the Script Completion features

The Scripting System supports the Script Completion facility. Server scripting panels include the Code Explorer, etc. The Code Explorer panel which displays the variables, methods and objects is used by the script in the tree-like panel.

### Script Completion

The Script completion feature utilizes a script pop up window which shows available methods and objects once you have entered the **period** character (the dot/fullstop character) after the variable name in the script.

```

var
  Board : IPCB_Board;
begin
  If PCBServer.
end;

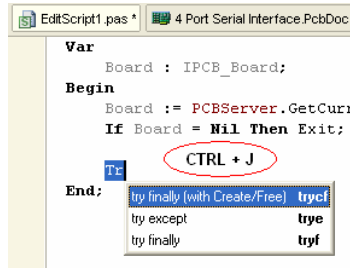
```

CTRL + SPACE



### Script Templates

You can use the script template feature to automatically generate a complete statement for you when you click on CTRL + J after you have typed the first few letters of the statement.



```

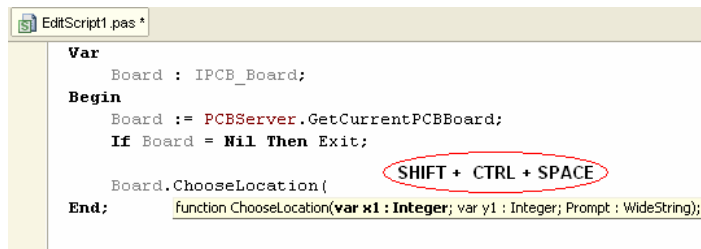
EditScript1.pas * 4 Port Serial Interface.PcbDoc
var
  Board : IPCB_Board;
begin
  Board := PCBServer.GetCurrentPCBBoard;
  If Board = Nil Then Exit;
end;

```

CTRL + J

### Script Parameters

You can use the Script Parameters feature to inspect which parameters are used for a particular function/procedure/method of an object interface. Use the SHIFT + CTRL + SPACE keys to display the parameters pop up message.



```

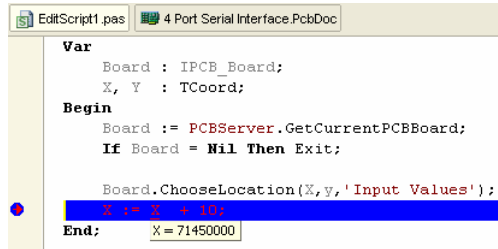
EditScript1.pas *
var
  Board : IPCB_Board;
begin
  Board := PCBServer.GetCurrentPCBBoard;
  If Board = Nil Then Exit;
end;

```

SHIFT + CTRL + SPACE

### Expression Evaluation Tooltip

Finally the ToolTips feature displays data values for the variable that the mouse cursor is hovering on.



The screenshot shows a code editor with two tabs: 'EditScript1.pas' and '4 Port Serial Interface.PcbDoc'. The code is as follows:

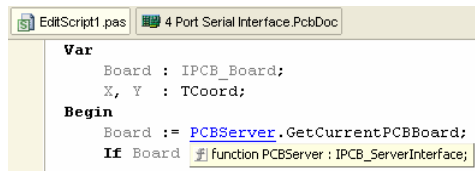
```
Var
  Board : IPCB_Board;
  X, Y  : TCoord;
Begin
  Board := PCBServer.GetCurrentPCBBoard;
  If Board = Nil Then Exit;

  Board.ChooseLocation(X, Y, 'Input Values');
  X := X + 10;
End;
      X = 71450000
```

A tooltip is displayed over the line `X := X + 10;`, showing the value `X = 71450000`.

### Symbol Insight Tooltip

The Tooltip feature shows what type the variable is when the mouse cursor is hovering on as well as pressing the CTRL key.



The screenshot shows the same code editor as above. The code is:

```
Var
  Board : IPCB_Board;
  X, Y  : TCoord;
Begin
  Board := PCBServer.GetCurrentPCBBoard;
  If Board = Nil Then Exit;
End;
```

A tooltip is displayed over the line `If Board = Nil Then Exit;`, showing the type `function PCBServer : IPCB_ServerInterface;`.



Consult the Scripting online help for more information on debugging scripts and script panels.

## Using Borland Delphi objects in your scripts

---

The scripting system handles two types of components for script forms: Visual and Nonvisual components. The visual components (Button, Edit and Memo components for example) are the ones you use to build the user interface, and the nonvisual components are used for different tasks such as these Timer, OpenFileDialog and MainMenu components. You use the Timer nonvisual component to activate specific code at scheduled intervals and it is never seen by the user. Components are also called controls. All script forms have one or more components. Any combination of components can be placed on a form, and while your script is running a user can interact with any component on a form, it is your task, to decide what happens when a user clicks a button or changes a text in an Edit box.

Both types of components appear at design time, but non visual components are not visible at runtime.

### Dropping components onto a script form

The Scripting system supplies a number of components for your scripts. You can find all the components you can place on a form from the *Toolbox* palette. To place a component on a form, locate its icon on the *Tool Palette* panel and double-click it. This action places a component on the active form. When you drop a component onto a form, the Scripting system automatically generates code necessary to use the component and updates the script form, thus you do not need to create or destroy these objects, the script form does them for you automatically.

You only need to set properties, put code in event handlers and use methods as necessary to get the component on the script form working in DXP. You use the Object Inspector to manipulate the properties of a component on the script form.

### Creating components from the script form

You can also directly create and destroy components in a script – normally you don't need to pass in the handle of the form because the script form takes care of it automatically for you, thus you just normally pass a Nil parameter to the Constructor of a component.

#### Example of creating a TOpenDialog component

```
Function LoadAFile : String;
Var
    OpenFileDialog : TOpenDialog;
begin
    Result := '';

    OpenDialog := TOpenDialog.Create(nil);
    OpenFileDialog.InitialDir := 'C:\';
    // Display the OpenFileDialog component
    OpenFileDialog.Execute;

    // Obtain the file name of the selected file.
    Result := OpenFileDialog.FileName;
    OpenDialog.Free;
End;
```

## Building Script Projects

Note, you can also use the **CreateObject** function as provided by the DelphiScript language. The **CreateObject** function can take in multiple parameters for example

```
Procedure DemoCreateObjectFn;
Var
    OpenDialog : TOpenDialog;
Begin
    // the CreateObject function takes in two parameters -
    // the TOpenDialog class and a Nil parameter
    OpenDialog := CreateObject(TOpenDialog, nil);
    OpenDialog.Execute;
End;
```

## Instantiating Borland Delphi Classes

For other Borland Delphi objects such as the **TStringList** object, you can use the classes from the Borland Delphi Run Time Library to create them during the duration of the script. Normally in scripts you can use these classes to create objects and store information in these objects and free them when they are no longer in use. It is your responsibility as a coder to create and free objects where appropriate.

```
Procedure Demo_TStringList_Class;
Var
    AList : TStringList;
    SaveDialog : TSaveDialog;
Begin
    Try
        AList := TStringList.Create;
        AList.Add('Line 1');
        AList.Add('Line 2');
        AList.SaveToFile(FileName);
    Finally
        AList.Free;
    End;
End;
```

You can use Borland Delphi Classes in your scripts whether it is DelphiScript, VB or JScript. The scripting system supports a sub set of the Borland Delphi 6 Run Time Library.



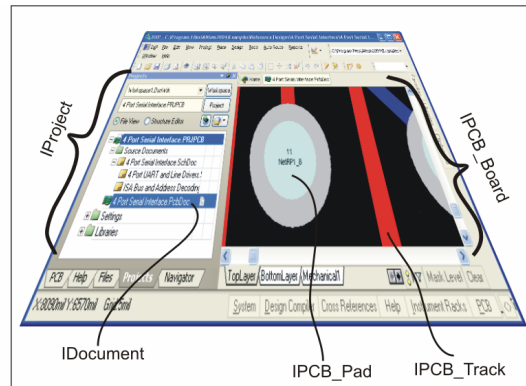
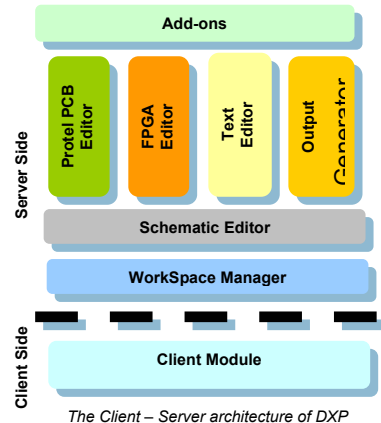
You can check out the Supported Borland Delphi Units in the Scripting Online Help to find which Borland units are exposed in the scripting system.

## Using DXP Object Models in your scripts

The DXP application is a large system which is composed of a single Client executable along with plugged in servers. The Client deals with actions generated by the user of the DXP application. The servers provide specialized functionality depending upon the task requested by the user.

The scripting system has access to the DXP Object Models and DXP Run Time Library functionality as well as a subset of Borland Delphi Run Time Library. You do not need to worry about where the functions or objects come from – they are exposed to be used in the scripts. The DXP Object model consists of the multiple object models such as PCB Object Model, Schematic Object Model and the WorkSpace Manager Object Model which are the major object models.

From the diagram below, a Design project is represented by the **IProject** interface as part of the WorkSpace Manager Object model. The PCB document is represented by the **IPCBoard** interface from the PCB Object Model.



In this tutorial, the Workspace Manager Object Model will be used to retrieve the connectivity information of the schematic project to generate a netlist. The PCB Object model is used to fetch PCB objects from a PCB document. The Query Expressions are used in query scripts to execute a custom query in the *Filter* panel for the PCB or Schematic document.



The Scripting Online Help contains the DXP RTL reference which covers the Object Models and the Run Time Library. You can use the Script Completion feature to find out what are the properties or methods for a DXP object interface or a Borland Delphi object in a script.



You can refer to the script examples, mostly in DelphiScript, in the **\Examples\Scripts\** folder which are available for experimentation.

# Building a Netlister project

---

The aim of this Netlister in this tutorial is to generate a Protel netlist (two formats called Versions 1 or 2) for the DXP Project that contains schematics. Basically a flat netlist of a schematic project is separated into two sections – component designators and the information associated with each component, and net names and the information associated with each net name along with pin connections (pins of a component).

From the WorkSpace Manager Object Model, there are interfaces that represent the project, and its constituents – the documents, the components and its pins, and the nets. The WorkSpace Manager is a system extensions server coupled tightly with the Client module and deals with projects and their associated documents. This server provides compiling, multi sheet design support, connectivity navigation tools, multi-channel support, multiple implementation documents and so on. To retrieve the WorkSpace Manager interface, invoke the **GetWorkspace** function which yields you the **IWorkspace** interface.

In this tutorial, we need to concern ourselves with the **IWorkspace**, **IProject**, **IDocument**, **IComponent**, **IPin** and **INet** interfaces. You may notice that some of the interfaces, especially the design object interfaces correspond directly to Schematic Object interfaces.

For example, **IPin** and **INet** correspond directly to **ISch\_Pin** and **ISch\_Net** interfaces respectively. It is because the logical documents in a project are schematic documents with connectivity information and the workspace manager is dealing with these documents. In fact, we can use the Schematic Object model instead, but the Workspace Manager provides us the functionality to compile a project, extract documents of a project as well as retrieving data from schematic objects.

Thus, let's start building the netlister script!

## Main parts of a netlister script

The main parts of the script are

- a global TargetFileName string which is the file name of the netlist.
- a global Netlist TStringList collection object which contains the data of the netlist.
- WriteComponent and its WriteComponent\_Version1 and WriteComponent\_Version2 routines
- WriteNet and its WriteNet\_Version1 and WriteNet\_Version2 routines
- ConvertElectricToString which converts a pin's electrical property to a string
- GenerateNetlist procedure

## The functionality of a Netlister

The netlister is based on the `ScripterProtelNetlist.pas` script from the `..\Examples\Scripts\DelphiScript Scripts\WSM\Protel Netlister\` folder. We will go through the netlister script step by step.

1. The two parameter-less procedures with **GenerateProtelV1FormatNetlist** and **GenerateProtelV2FormatNetlist** names will appear in the *Select Item to Run* dialog. So the user has the choice of generating a Protel V1 format netlist OR a Protel V2 format netlist. These procedures will call the **GenerateNetlist** procedure.

### The two procedures in the Select Item to Run dialog

```

Procedure GenerateProtelV1FormatNetlist;
Var
    Version : Integer;
Begin
    // Protel 1 Netlist format, pass 0
    GenerateNetlist(0);
End;
Procedure GenerateProtelV2FormatNetlist;
Var
    Version : Integer;
Begin
    // Protel 2 Netlist format, pass 1
    GenerateNetlist(1);
End;

```

2. The code inside the **GenerateNetList** procedure gets the workspace interface so that project interface can be extracted for the current project open in DXP. The project needs to be compiled first before we can extract nets for the project, because the compile process builds the connectivity information for the project in memory. The Project interface's **DM\_Compile** method does that as shown in the code snippet below.

```

WS := GetWorkspace;
If WS = Nil Then Exit;
Prj := WS.DM_FocusedProject;
If Prj = Nil Then Exit;

// Compile the project to fetch the connectivity
// information for the design.
Prj.DM_Compile;

```

3. The component and net information is stored in the Netlist object of **TStringList** type which is used later to generate a formatted netlist text file. The **TStringList** object is a Borland Delphi class part of the **Classes** unit of the Borland Delphi Run Time Library which is available to use in scripts.
4. Since a netlist is broken up into two sections – so we need two procedures to write component data and net data separately. For nets, at least two nodes in a net will be written out to a netlist, anything less than two nodes is discarded. For each net, the net name is based on the Net's **DM\_CalculatedNetName** method which extracts the name from the connectivity information generated by the compile process in DXP). Two code snippets for the Components and Nets sections of a netlist are shown below for Protel Version 1 format. Remember that component and net data are stored in the **NetList** object (which is a **TStringList** object).
5. The **Generate** procedure is a heavy duty procedure, it obtains the output path of a project for the netlist file to go in. Then with all the schematic documents in a project, each document is checked for nets and components and they are then extracted to the Netlist object.

## Building Script Projects

6. When a netlist is generated, it is composed of two sections; the component information section and the net information section.

### Components Section

In this section, for each component found from the Design Project, it is checked if it is an actual component and then the physical designator, footprint and part type are extracted.

```
If Component <> Nil Then
Begin
    NetList.Add(' ');
    NetList.Add(Component.DM_PhysicalDesignator);
    NetList.Add(Component.DM_FootPrint);
    NetList.Add(Component.DM_PartType);
    NetList.Add(' ');
    NetList.Add(' ');
    NetList.Add(' ');
    NetList.Add(']');
End;
```

### Nets section

In this section, if a net has two pins or more, then the NetName and the designators are extracted.

```
If Net.DM_PinCount >= 2 Then
Begin
    NetList.Add(' (');
    NetList.Add(Net.DM_CalculatedNetName);

    For i := 0 To Net.DM_PinCount - 1 Do
    Begin
        Pin := Net.DM_Pins(i);
        PinDsgn := Pin.DM_PhysicalPartDesignator;
        PinNo := Pin.DM_PinNumber;
        NetList.Add(PinDsgn + '-' + PinNo);
    End;
    NetList.Add(')');
```

7. Save the script and then execute the **RunScript** command from DXP system menu. The *Select Item to Run* dialog appears. Choose either the **GenerateProteI1FormatNetlist** procedure or **GenerateProteI2FormatNetlist** procedure and a netlist will be generated and opened in DXP automatically.

You can open the `Scripter_ProtelNetlist.PrjScr` project and execute the `ScripterProtelNetlist.pas` script from the `\Examples\Scripts\DelphiScript Scripts\WSM\Protel Netlister\` folder.

## Building the Board Outline Copier project

---

The aim of this Board Outline Copier in this tutorial is to copy an existing board outline from the PCB document to a layer in the same document. Basically, with the PCB Object model and the PCB interfaces, we can extract objects of a board outline and copy these objects onto a specified layer.

We will be using a script form so that user can input the width of the board outline and select the layer from a drop down menu. Thus, let's start dissecting the board outline copier script! The Board Outline Copier example can be found in `\Examples\Scripts\VB Scripts`, `\Examples\Scripts\JScript Scripts` and `\Examples\Scripts\DelphiScript Scripts\PCB` folders.

### Main parts of a Board Outline Copier script

The main parts of the script are

- a global `PCB_Board` (of `IPCB_Board` type) variable
- `CopyBoardOutline` sub routine with `AWidth` and `ALayer` parameters
- `bCancelClick` event handler which closes the Board Outline script form.
- `bOkClick` event handler which obtains the width and layer values from the script form and then executes the `CopyBoardOutline` subroutine.

### The functionality of a Board Outline Copier script

Since we are using a script form, therefore we need to have event handlers to capture the mouse clicks of individual controls such as the Cancel and Ok buttons. The Ok Click event handler obtains the Width of the outline in Internal coordinate values from the `StringToCoordUnit` function and the Layer enumerated value from the `String2Layer` function. Both functions come from the DXP Run Time Library which is available to use in scripts. Load the `CopyBoardOutlinePRJ.PRJSCR` project and check out the `CopyBoardOutlineForm` Script form.

#### IPCB\_Board Interface

A board outline is obtained from the `IPCB_Board` interface via the `PCBServer` function and it needs to be initialized first before proceeding with copying and creating a new board outline.

#### Board outline has arc and track segments

The board outline is represented by the `IPCB_BoardOutline` interface and this interface is inherited from the `IPCB_Group` interface. A `IPCB_Group` interface represents a group object that can store child objects. An example of `IPCB_Group` interface is a polygon or a board outline because they can store arcs and tracks.

A board outline object stores two different type of segments – `ePolySegmentLine` and `ePolySegmentArc` which represent a track or arc object respectively. The number of segments is determined by the `PointCount` method from the `IPCB_BoardOutline` interface.

Each segment of this board outline need to be checked for tracks and arcs with the `If PCB_Board.BoardOutline.Segments(I).Kind = ePolySegmentLine Then` statement. For each segment found, depending on the segment kind, a new track or arc is created.

#### PCBObjectFactory and AddPCBObject methods

## ***Building Script Projects***

Creating a PCB object employs the **PCBObjectFactory** method from the **IPCB\_ServerInterface** interface. In this case, the **PCBServer.PCBObjectFactory(eArcObject, eNoDimension, eCreate\_Default)** statement creates a new Arc object. A new copy of an arc object is added onto a specified layer of the PCB document with the **PCB\_Board.AddPCBObject(NewObject)** statement.

### **PreProcess and PostProcess methods**

For each PCB object creation, you need to apply **PreProcess** method from the **IPCB\_ServerInterface**, ie **PCBServer.PreProcess** before creating a PCB object, and then after creating this PCB object you need to apply the **PostProcess** method from the **IPCB\_ServerInterface** interface. The **PreProcess** and **PostProcess** methods keep the Undo system synchronized in the PCB editor.

### **Setting the new PCB layer**

When objects are added to a layer that has not been displayed in the PCB document, we need to force this layer to be visible. This **PCB\_Board.LayerIsDisplayed(ALayer) = True** statement does this job.

### **Refreshing the PCB document.**

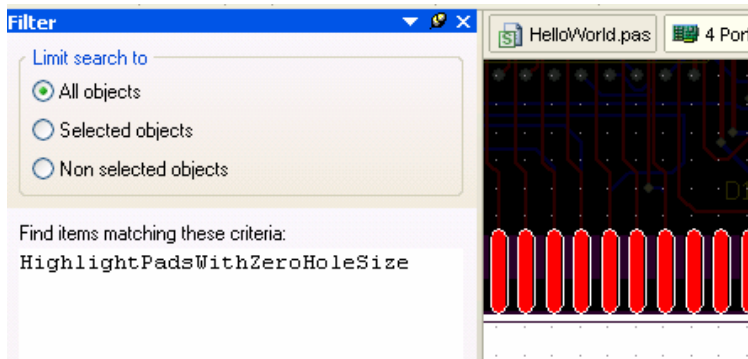
Finally the PCB document is refreshed with the new board outline on a new layer with the **PCB:Zoom** command and using the **Action = Redraw** parameters.

## Using Query Expressions in scripts

The *Filter* panel in Schematic and PCB has the ability to execute scripts that have query expressions. To be able to execute query scripts, these script projects must be installed in the **Installed Projects** list, so that they are made global to DXP. From the **DXP » Preferences** menu, and drill down to **Scripting System** folder within the DXP System on the left pane of the *Preferences* dialog to install scripts in the Installed Projects list.

A query script must have a function name that returns a boolean value. Inside the function block contains the query expressions. The function name is entered in the Filter panel's expression window as shown below.

Query Expression scripts need to be installed in the **Installed Projects** list in the Scripting System page under the DXP System folder on the **Preferences** dialog.



An example will be demonstrated how a query expression incorporated in a script can be executed in the Filter query box on the *Filter* panel. A simple query expression that looks for components with pads that have zero hole sizes. This script can only be executed on a PCB document.

```
Function HighlightPadsWithZeroHoleSize : Boolean;
Begin
    Result := False;
    // Check if PCB document exists, if not, exit.
    If UpperCase(Client.CurrentView.OwnerDocument.Kind) <> 'PCB' Then Exit;
    Result := IsComponentPad and (Holesize = 0);
End;
```



You can refer to the Query Expression Script examples which can be found in **\Examples\Scripts\Query Scripts** folder.



Consult the **Query System** online help in DXP to obtain more information on PCB and Schematic query expressions.

## **Conclusion**

---

You have learnt how to create a script project and dissected three functional scripts that uses Workspace Manager Object Interfaces, PCB Object Interfaces and Query Expressions.

You can consult the Online Help in DXP for the scripting system, query language reference and the various scripting languages used for the scripting system.

Finally, you can explore various script examples in the `\Examples\Scripts` folder.

## Revision History

---

Date	Version No.	Revision
30-Nov-2004	1.0	New product release

Software, hardware, documentation and related materials:

Copyright © 2004 Altium Limited.

All rights reserved. You are permitted to print this document provided that (1) the use of such is for personal use only and will not be copied or posted on any network computer or broadcast in any media, and (2) no modifications of the document is made. Unauthorized duplication, in whole or part, of this document by any means, mechanical or electronic, including translation into another language, except for brief excerpts in published reviews, is prohibited without the express written permission of Altium Limited. Unauthorized duplication of this work may also be prohibited by local statute. Violators may be subject to both criminal and civil penalties, including fines and/or imprisonment. Altium, CAMtastic, Design Explorer, DXP, LiveDesign, NanoBoard, NanoTalk, Nexar, nVisage, CircuitStudio, P-CAD, Protel, Situs, TASKING, and Topological Autorouting and their respective logos are trademarks or registered trademarks of Altium Limited or its subsidiaries. All other registered or unregistered trademarks referenced herein are the property of their respective owners and no trademark rights to the same are claimed.