SIEMENS EDA

Getting Started with Layout Automation

Release X-ENTP VX.2.13
Document Revision 10

SIEMENS

Unpublished work. © 2023 Siemens

This Documentation contains trade secrets or otherwise confidential information owned by Siemens Industry Software Inc. or its affiliates (collectively, “Siemens”), or its licensors. Access to and use of this Documentation is strictly limited as set forth in Customer’s applicable agreement(s) with Siemens. This Documentation may not be copied, distributed, or otherwise disclosed by Customer without the express written permission of Siemens, and may not be used in any way not expressly authorized by Siemens.

This Documentation is for information and instruction purposes. Siemens reserves the right to make changes in specifications and other information contained in this Documentation without prior notice, and the reader should, in all cases, consult Siemens to determine whether any changes have been made.

No representation or other affirmation of fact contained in this Documentation shall be deemed to be a warranty or give rise to any liability of Siemens whatsoever.

If you have a signed license agreement with Siemens for the product with which this Documentation will be used, your use of this Documentation is subject to the scope of license and the software protection and security provisions of that agreement. If you do not have such a signed license agreement, your use is subject to the Siemens Universal Customer Agreement, which may be viewed at https://www.sw.siemens.com/en-US/sw-terms/base/uca/, as supplemented by the product specific terms which may be viewed at https://www.sw.siemens.com/en-US/sw-terms/supplements/.

SIEMENS MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS DOCUMENTATION INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT OF INTELLECTUAL PROPERTY. SIEMENS SHALL NOT BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, CONSEQUENTIAL OR PUNITIVE DAMAGES, LOST DATA OR PROFITS, EVEN IF SUCH DAMAGES WERE FORESEEABLE, ARISING OUT OF OR RELATED TO THIS DOCUMENTATION OR THE INFORMATION CONTAINED IN IT, EVEN IF SIEMENS HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

TRADEMARKS: The trademarks, logos, and service marks (collectively, "Marks") used herein are the property of Siemens or other parties. No one is permitted to use these Marks without the prior written consent of Siemens or the owner of the Marks, as applicable. The use herein of third party Marks is not an attempt to indicate Siemens as a source of a product, but is intended to indicate a product from, or associated with, a particular third party. A list of Siemens' Marks may be viewed at: www.plm.automation.siemens.com/global/en/legal/trademarks.html. The registered trademark Linux® is used pursuant to a sublicense from LMI, the exclusive licensee of Linus Torvalds, owner of the mark on a world-wide basis.

About Siemens Digital Industries Software

Siemens Digital Industries Software is a leading global provider of product life cycle management (PLM) software and services with 7 million licensed seats and 71,000 customers worldwide. Headquartered in Plano, Texas, Siemens Digital Industries Software works collaboratively with companies to deliver open solutions that help them turn more ideas into successful products. For more information on Siemens Digital Industries Software products and services, visit www.siemens.com/plm.

Support Center: support.sw.siemens.com
Send Feedback on Documentation: support.sw.siemens.com/doc_feedback_form

Revision History ISO-26262

Revision	Changes	Status/Date
10	Modifications to title page to reflect the latest product version supported. Approved by Kevin Chupp. No technical enhancements, changes, and fixes listed in the *Xpedition Enterprise Flow Release Notes* for this product are reflected in this document. Approved by Mike Bare.	Released March 2023
9	Modifications to title page to reflect the latest product version supported. Approved by Kevin Chupp. No technical enhancements, changes, and fixes listed in the *Xpedition Enterprise Flow Release Notes* for this product are reflected in this document. Approved by Mike Bare.	Released September 2022
8	Modifications to title page to reflect the latest product version supported. Approved by Kevin Chupp. All technical enhancements, changes, and fixes listed in the *Xpedition Enterprise Flow Release Notes* for this product are reflected in this document. Approved by Mike Bare.	Released March 2022
7	Modifications to title page to reflect the latest product version supported. Approved by Kevin Chupp. All technical enhancements, changes, and fixes listed in the *Xpedition Enterprise Flow Release Notes* for this product are reflected in this document. Approved by Mike Bare.	Released September 2021

Author: In-house procedures and working practices require multiple authors for documents. All associated authors for each topic within this document are tracked within the Siemens documentation source. For specific topic authors, contact the Siemens Digital Industries Software documentation department.

Revision History: Released documents include a revision history of up to four revisions. For earlier revision history, refer to earlier releases of documentation on Support Center.

Table of Contents

Chapter 1
Setting up the Tutorial Data 11
Tutorial Topics..... 11
Tutorial Design Data..... 13
Installing the Tutorial Design Data..... 13

Chapter 2
Introduction 15
COM Automation..... 15
Documentation Conventions..... 16
Font Usage..... 16
Mouse Usage..... 16
Automation Terminology..... 17
Client and Server..... 17
Enumerate..... 18
Object..... 19
Property..... 19
Method..... 20
Collection..... 20
Event..... 21
Type Library..... 21
Prog ID..... 21
Script..... 22
Script Host..... 22
Scripting Language..... 22
Accelerator Key Binding..... 23
Menu Item..... 23
Pulldown and Popup Menu..... 23
In-Process Client..... 24
Out-of-Process Client..... 24

Chapter 3
VBScript Primer..... 27
Scripting Languages..... 27
VBScript Language..... 29
General-Purpose Statements..... 30
Option Explicit..... 30
Dim—Dimension a Variable..... 30
Dim(x,y,z,…)—Dimension an Array of Variables..... 31
ReDim(x,y,z,…)—Re-dimension an Array of Variables..... 32
UBound—Return the Upper Bound..... 33
Set—Setting an Object to a Variable..... 34
Call—Calling a Method or Function..... 35
Arithmetic, Comparison, and Logical Operators..... 36

Arithmetic Operators....36
Comparison Operators....37
Logical Operators....38
String Operators....38
Control Statements....40
If … Then … End If....40
If … Then … Else … End If....40
For … To … Next....41
For Each … In … Next....42
While … Wend....42
Select Case … End Select....42
Functions and Subroutines....44
Function and End Function....44
Sub and End Sub....45
User Interaction Functions....46
InputBox()....46
MsgBox()....46
Conversion and Formatting Functions....48
Conversion Functions....49
CInt()....49
CDbl()....49
CStr()....49
String Manipulation Functions....50
Len()....50
Left() and Right()....50
Replace()....51
Special Characters....51
Format Number Function....51
Date and Time Functions....52
Error Handling....54

Chapter 4
Running Your First Script....59
Creating and Running the Hello World Script....60
Creating the Hello World Script....60
Running the Hello World Script....60
Creating and Running the Via Count Script....61
Understanding the Data Hierarchy....62
Creating the Via Count Script....62
Running the Via Count Script....65
An Alternative Technique....66

Chapter 5
Creating Forms and Dialogs Using the IDE....69
Creating a Script With the IDE....69
Handling Errors in the IDE....74
Creating a Dialog With the IDE....76
Launching the IDE in Form-editing Mode....76

Adding Buttons to the Form....78
Adding Code "Behind" the Cancel Button....81
Adding Code "Behind" the Count All Button....84
Adding Code "Behind" the Count Selected Button....93
Removing Duplicate Code....95
Assigning a Title....96
Going into Production....98

Chapter 6
Adding Accelerator Keys....103
Adding an Accelerator Key....103

Chapter 7
Adding Menus and Menu Items....107
Adding a Menu Item With Two Scripts....107
Adding a Menu Item With One Script....114
ViaCountMenuSingle.vbs....116
Adding a New Top-Level Menu....121
ViaCountMenuTop.vbs....122

Chapter 8
Adding Toolbar Buttons....127
Adding a New Button....127
Viewing the Utilities Toolbar....127

Chapter 9
Running Scripts from the Command Line....133
Running Command Line Scripts....135
Running Command Line Scripts—With No Arguments....135
Displaying Command Line Arguments....138
Using Command Line Arguments....143
Adding Additional Error Handling....146
Working With Applications and Documents....149
Attaching to an Existing Application—With an Open Document....149
Attaching to an Existing Application—With No Open Document....151
Creating an Invisible Application....155
Creating a Visible Application....158

Chapter 10
Running Scripts from an Application....161
Using the WDIR Environment Variable....162
Creating and Modifying the WDIR Environment Variable....163

Chapter 11
Running Startup Scripts....171
Using scripts.ini Files....171

Creating Application Startup Scripts..... 172
Exercise 1: Creating an Xpedition Layout Startup Script..... 173
Creating Document Startup Scripts..... 175
Exercise 2: Creating an Xpedition Layout Document Startup Script..... 175
Debugging Startup Scripts..... 176

Chapter 12
Introducing Events and the Scripting Object 179
Events and In-Process Scripts..... 179
Events and Out-of-Process Scripts Using mgcscript.exe..... 183

Chapter 13
Basic Building-Block Examples..... 187
Creating a Skeleton Script..... 187
Traversing Objects..... 189
Exercise 1: Traversing Nets..... 189
Exercise 2: Traversing Components..... 192
Exercise 3: Traversing Pins..... 197
Task #1: Selecting all pins with the name “1”..... 199
Task #2: Selecting all of the pins on a user-specified net..... 200
Task #3: Selecting all of the pins on a specific component..... 201
Filtering Collections..... 208
Exercise 4: Finding Unplaced Components..... 208
Exercise 5: Deleting Traces..... 211
Exercise 6: Filtering Padstack Objects..... 214
Using the Display Control..... 218
Exercise 7: Toggling the Display of Placement Outlines..... 218
Exercise 8: Changing the Layer Color..... 219
Exercise 9: Loading Scheme..... 221
Running Menu Commands..... 225
Exercise 10: Running the Fit Board Command..... 225
Exercise 11: Running the Route Command..... 227
Creating Objects with Geometries..... 232
Exercise 12: Adding Traces..... 235
Exercise 13: Adding User Layer Graphics..... 243
Creating Interactive Commands..... 248
Exercise 14: Placing Vias With the Mouse..... 248
Creating Input/Output (I/O) Files..... 256
Exercise 15: Creating a Net Report..... 256
Exercise 16: Reading a Points Array From a File..... 259
Running Executables..... 268
Exercise 17: Viewing a Net Report With NotePad..... 268
Exercise 18: Compressing a File and E-mailing the Result..... 270
Reading, Analyzing, and Updating a Design..... 276
Exercise 19: Generating Via Caps..... 276

Chapter 14
Xpedition Layout Examples....283
Exercise 1: Displaying a Single Routing Layer.... 283
Exercise 2: Documenting the Layer Stackup.... 296
Exercise 3: Performing Assembly DRC 316
Exercise 4: Removing Unconnected Pads from Pins and Vias....335
Exercise 5: Integration with Excel....358
Exercise 6: Generating a Component CSV File....374
Exercise 7: Finalizing a Design 383
Exercise 8: Routing by Layer 401

Chapter 15
Xpedition FabLink Examples....419
Introducing Xpedition FabLink 419
Exercise 1: Generating a BOM....421
Exercise 2: Generating a Panel Side View.... 442
Exercise 3: Creating a Via Mask.... 458
Exercise 4: Generating Manufacturing Output Files 473
Exercise 5: Automated Board Placement on a Panel.... 491

Chapter 16
Recommended Scripting Practices 519
Naming Conventions.... 520
Variable Names....520
Function and Subroutine Names.... 520
Method and Property Names....521
Event-Handler Function Names....521
Object Names in IDE Forms.... 521
File Names for Scripts and Forms.... 522
Option Explicit Statement....522
Comments.... 523
Parentheses....523
Space Characters....523
Dim and Dim()....523

Chapter 17
Recommended Script Architecting Process 525
Creating a Template....525
Re-using Functions and Subroutines 526
Internal and External Scripts....526
Calling Other Scripts.... 526
Script to Script Communication.... 526
Delivering Scripts.... 527

Chapter 18
Creating and Using a Library 529
Creating a Library..... 529
Calling Routines From a Library.....530

Chapter 19
VBScript References and Tools 535
General References..... 535
Text and Scripting Editors..... 535
Object Browsers..... 536

Third-Party Information

Chapter 1
Setting up the Tutorial Data

Welcome to the PCB Automation Tutorial. This section covers the following topics:

Tutorial Topics
Tutorial Design Data
Installing the Tutorial Design Data

Tutorial Topics

The PCB Automation Tutorial assumes no prior knowledge of Automation and familiarity with the Expedition layout environment.

Table 1: Tutorial Topic Summary summarizes the major topics that tutorial contains.

Note:
Reference topics and exercises that do not specify scope context support both X-ENTP VX.x, and PADS-Pro VX.x flows.

Table 1. Tutorial Topic Summary

Topic/Page	Description
“Introduction” on page 15	This section describes COM Automation, Documentation Conventions, and Automation Terminology.
“VBScript Primer” on page 27	This section provides an overview of Scripting Languages, and the VBScript Language.
“Running Your First Script” on page 59	This section guides you through the process of running a simple "Hello World" script, “Creating and Running the Hello World Script”. Next, you create and run a script that counts the number of vias in a design, “Creating and Running the Via Count Script”. This script is the basis for scripts throughout this tutorial.
“Creating Forms and Dialogs Using the IDE” on page 69	This section provides several exercises for creating forms and scripts using the Integrated Development Environment (IDE). The exercises include: “Creating a Script With the IDE”, “Handling Errors in the IDE”, and “Creating a Dialog With the IDE”.
“Adding Accelerator Keys” on page 103	This section describes how to associate a shortcut (accelerator) key with a script using the Key Bind Server. The exercise is “Adding an Accelerator Key”.
“Adding Menus and Menu Items” on page 107	This section describes how to add menus and menu items to an application’s menu set using the Command Bar Server.

Table 1. Tutorial Topic Summary (continued)

Topic/Page	Description
“Adding Toolbar Buttons” on page 127	This section describes how to add toolbar buttons (icons) to an application’s toolbar using the Command Bar Server.
“Running Scripts from the Command Line” on page 133	This section provides several exercises for running scripts from the command line and include—passing arguments, attaching to existing applications, and creating invisible applications.
“Running Scripts from an Application” on page 161	This section shows you how to create and modify the WDIR environment variable, which facilitates multiple ways to run a script.
“Running Startup Scripts” on page 171	This section contains exercises that show how to create application and document (design) startup scripts, and how to debug startup scripts.
“Introducing Events and the Scripting Object” on page 179	This section introduces events and event handling functions. The exercises include handling events in scripts that run within the scope of the application, “Events and In-Process Scripts” and handling events in scripts that run out-of-process, “Events and Out-of-Process Scripts Using mgcscript.exe”.
“Basic Building-Block Examples” on page 187	This section presents a number of basic “building-block” automation examples. Each of these scripts perform a fundamental task in isolation. You can combine these tasks in different ways to form more scripts. The categories of exercises include: “Traversing Objects”, “Filtering Collections", "Using the Display Control", "Running Menu Commands", "Creating Objects with Geometries", "Creating Interactive Commands", Creating Input/Output (I/O) Files", "Running Executables", and "Reading, Analyzing, and Updating a Design”.
“Xpedition Layout Examples” on page 283	This section presents several Xpedition Layout script examples.
“Xpedition FabLink Examples” on page 419	This section introduces Xpedition FabLink and presents several exercises focused on automating panel design and producing manufacturing outputs and documentation.
“Recommended Scripting Practices” on page 519	This section presents recommended scripting practices including “Naming Conventions", "and use of “Option Explicit Statement", "Comments", "Parentheses", "Space Characters", and "Dim and Dim()”.
“Recommended Script Architecting Process” on page 525	This section presents a number of tricks and techniques to make the script easy to understand, use, re-use, and maintain.
“Creating and Using a Library” on page 529	This section shows you how to create and re-use a library of help functions.

Table 1. Tutorial Topic Summary (continued)

Topic/Page	Description
"VBScript References and Tools" on page 535	This section presents a list of scripting references and scripting tools.

Tutorial Design Data

The tutorial data can be found in a downloadable *PCB_Automation_Tutorial.zip* file from your local install at *<path_to_install_target>\<release>\docs\data\PCB_Automation_Tutorial.zip.* For example:

```
C:\MentorGraphics\EEVX.2.10\docs\data\PCB_Automation_Tutorial.zip
```

The tutorial data contains the following:

- **.\updated_files\Automation Tutorial Sample Scripts.zip** — contains all of the "finished" scripts for this tutorial.
- **.\updated_files\Automation Tutorial Sample Expedition Design.zip** — contains the Xpedition Layout design named *Candy*.
- **Automation Tutorial Sample FabLink XE Design.zip** — contains the Xpedition FabLink panel design named *CandyPanel*.

Installing the Tutorial Design Data

Install the data.

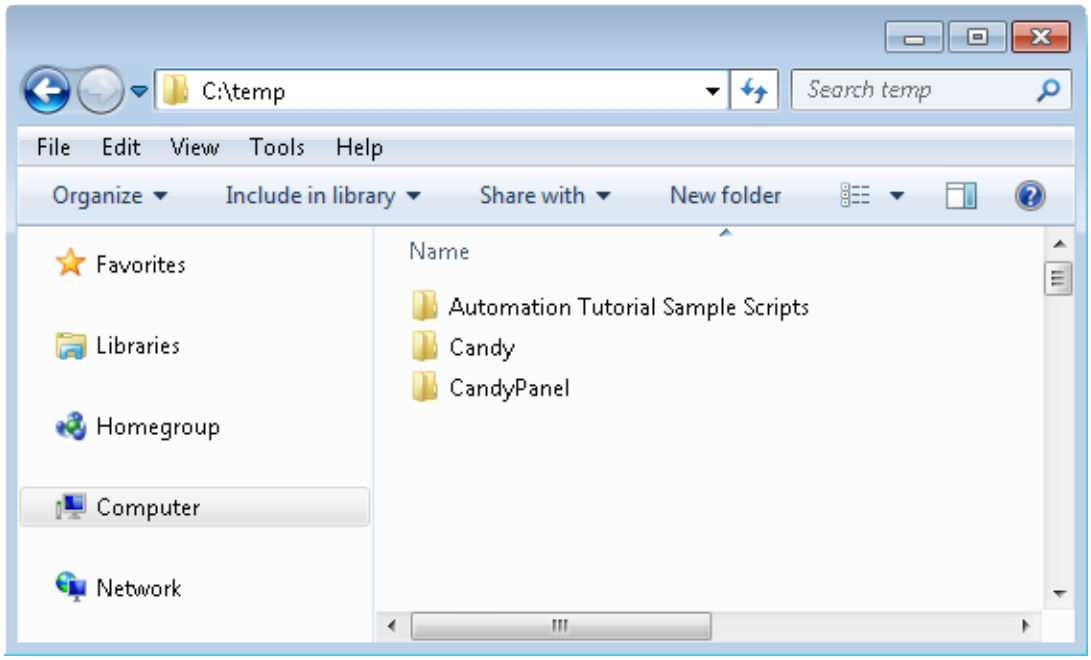
Prerequisites

- Your Operating system is Microsoft® Windows. This tutorial is not supported on UNIX or Linux.
- Installed Xpedition Layout
- Installed Xpedition FabLink

Procedure

Double-click each zip file and extract to the C:\temp location on your machine. Refer to Figure 1: Extracting the Design Data for details.

Figure 1. Extracting the Design Data



Results

The C:\temp directory contains three folders:

- Automation Tutorial Sample Scripts
- Candy
- CandyPanel

Chapter 2
Introduction

This section describes COM Automation, Documentation Conventions, and Automation Terminology.

COM Automation
Documentation Conventions
Automation Terminology

COM Automation

Xpedition Layout is Mentor Graphics' state-of-the-art PCB layout application. In the context of this document, the term automation refers to the ability for customers (and Mentor Graphics personnel) to augment, customize, and drive the capabilities of Xpedition Layout and related application engines by means of a scripting language and associated support utilities. (The reasoning behind the automation terminology comes from the perspective of the end-user having the ability to automate a series of actions that are available in the application.)

Xpedition Layout supports *Component Object Model* (COM) automation. Developed by Microsoft®, you can consider COM as both a framework and an *Application Programming Interface* (API) that provides the ability to access, control, and manipulate data inside applications. Note that automation in a non-Mentor Graphics application does not necessarily infer COM technology. Different applications may offer various types of automation as follows:

- **Data-Centric Automation** — In this case, the automation scripts have access to the application's data and it is not necessary for the end-user to pre-select any data before running an automation script. (In this case, “access to the application's data,” does not imply that the automation scripts directly access and modify core data structures internal to the application; instead, the scripts access and interact with this data by means of methods, objects, and properties.)

- **Function-Centric Automation** — This form of automation relies on a set of built-in application functions. These functions operate on whatever data is pre-selected. It is possible for the end-user to create new functions, but these can only run a series of existing functions. This limits the functionality to whatever is built-in the application by its architects. The Mentor Graphics Falcon Framework and applications such as Board Station are an example of this automation. The Falcon Framework provides the function-centric automation, while the scripting language AMPLE drives the automation.

- **Flow-Centric Automation** — This form of automation can augment and customize individual applications; it can also control data flow through multiple applications in a design process. (Different data and automation are available at different points in the design flow.)

The automation Xpedition Layout provides is predominantly data-centric, but significant amounts of function-centric and flow-centric capabilities also exist.

Documentation Conventions

This section describes the documentation conventions for this tutorial.

Font Usage
Mouse Usage

Font Usage

The following font conventions are applied in this manual.

- *Italics* font is used to emphasize text. It is also used for book, document, and for path and file names.
- **Bold** is used to emphasize text, highlight menu items, and denote the titles of (and fields in) dialog boxes.
- **Menu > Command** identifies the path to select a menu using the application GUI.
- **Menu– > Command** identifies the path to select a menu in a script.
- Courier font is used for program and script listings and for any text messages that the software displays on the screen.

Mouse Usage

The following mouse usage conventions apply in this manual.

- “Click” or “left-click” means click the left mouse button on the indicated item.
- “Right-click” means click the right mouse button on the indicated item.
- “Double-click” means click twice consecutively with the left mouse button.
- “Drag-and-drop” means press and hold the left mouse button on the indicated item, then move the cursor to the destination, release the button.
- “Select” is the same as “click.”

Automation Terminology

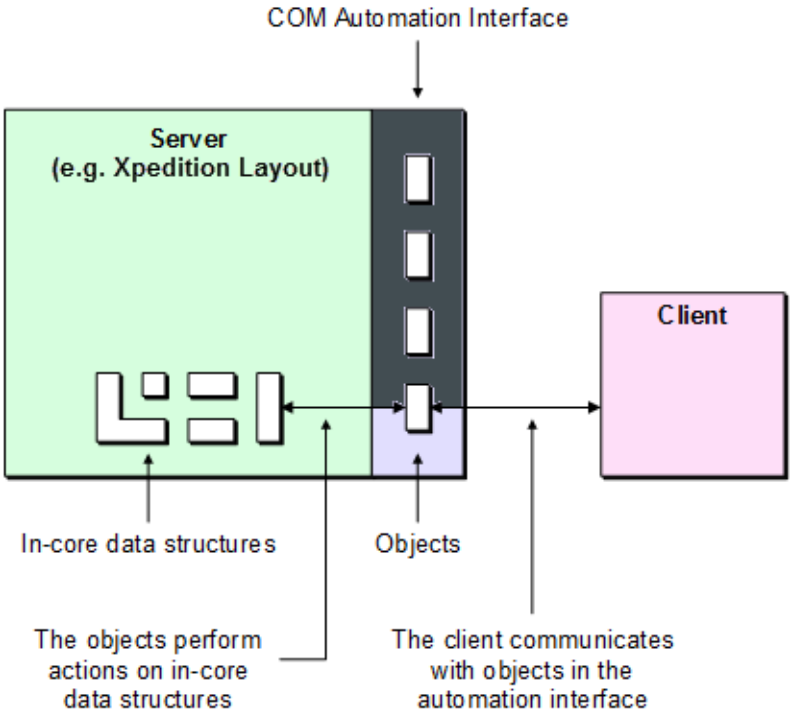
Automation involves a large number of concepts and terms. This section briefly defines fundamental terminology.

Client and Server
Enumerate
Object
Property
Method
Collection
Event
Type Library
Prog ID
Script
Script Host
Scripting Language
Accelerator Key Binding
Menu Item
Pulldown and Popup Menu
In-Process Client
Out-of-Process Client

Client and Server

A *server* is a software module or application that satisfies requests from a client. A *client* is a software module or application that makes requests to a server (or servers). As an example, consider the following illustration:

Figure 2. Client-Server Based Automation



In Figure 2: Client-Server Based Automation Xpedition Layout is in the role of the server. The external (out-of-process) client communicates with objects in Xpedition Layout's automation interface. These objects perform actions on Xpedition Layout's in-core data structures. All of this is largely transparent to the user. From the user's perspective, it appears that actions are performed directly on the in-core data structures; indeed, things should be visualized as working this way.

Note:
Multiple clients can communicate with a single server, a single client can communicate with multiple servers, or multiple clients can communicate with multiple servers. For details, refer to In-Process Client and Out-of-Process Client.

Enumerate

An *enumerate* may be regarded as a “descriptive number.” For example, one property of a *Trace* object is *Anchor*. The application represents the anchor type as an integer:

0 = None

1 = Semi-fixed

2 = Fixed

3 = Locked

To set the anchor type to *Fixed*, you can enter the following statement:

```
traceObj.Anchor = 2
```

However, this is not intuitive and it makes this statement difficult to parse and understand. For this reason, Xpedition Layout has an enumerate named *epcbAnchorType*, which has four possible assignments:

```
epcbAnchorNone        =  0
epcbAnchorSemiFixed   =  1
epcbAnchorFixed       =  2
epcbAnchorLocked      =  3
```

To set the anchor type to *Fixed* using an enumerate, enter the following statement:

```
traceObj.anchor = epcbAnchorFixed
```

Object

An *object* is an entity (in memory) containing methods and properties. Objects perform actions on data or *Graphical User Interface* (GUI) elements in the server. For example, a *Trace object* is a data object that operates on or provides information about a single trace in the server. A *Button object* is a GUI object that operates on a button in a dialog.

Rather than directly access design data, you assign objects to variables. You can think of the variable as a pointer or a reference that links to the actual database object.

To assign an object to a variable, you use the *Set* statement as follows:

```
Set netObj = documentObj.FindNet("GND")
```

Note:
The Set statement is discussed in “Set—Setting an Object to a Variable” on page 34.

Property

A *property* is an attribute associated with an object. For example, *width* is one attribute of the *Trace* object.

Properties are set and modified by means of the “=” assignment operator. For example, you can set the width of a trace to a specific value using the following statement:

```
traceObj.width = 10
```

Alternatively, if an integer variable named *widthInt* has a value of 10, you can set the width using the following statement:

```
traceObj.width = widthInt
```

Method

A *method* is a functional operation or action that is performed on an object. For example, Xpedition Layout supports a method named *MoveRelative* that performs a relative move on objects such as *Components*.

You can move a component relative to its original location using the following statement:

```
Call cmpObj.MoveRelative(10,15)
```

Note:
The *Call* statement in the example is discussed in "Call—Calling a Method or Function" on page 35.

Alternatively, if two integer variables named *xInt* and *yInt* have values of 10 and 15, respectively, you can enter the following statement to perform the relative move:

```
Call cmpObj.MoveRelative(xInt,yInt)
```

Note the difference between *applying* a method and *assigning* a value to a property. Often it is easy to understand the difference between properties and methods. Other times the line is somewhat complicated. Consider the following statement, which finds a net named *Fred*:

```
Set netObj = documentObj.FindNet("Fred")
```

This appears similar to manipulating a property because the statement returns an object. However, this is performing an action on the document (*finding* an object). Therefore, in this example the statement is applying a method.

Note:
The Set statement in the example is discussed in "Set—Setting an Object to a Variable" on page 34.

Collection

A *collection* is a container for multiple objects. Furthermore, you handle a collection as any another object. This means that scripts assign collections to variables using the *Set* statement. The following locates all placed components in the design document and returns them in the collection:

```
Set placedCompsColl = documentObj.Components(epcbSelectPlaced)
```

In this case, *epcbSelectPlaced* is a pre-defined option associated with an enumerate.

Note:
Every collection has an associated *Item* method that allows you to locate and perform actions on, a single item in the collection.

Event

When using properties and methods, the client initiates communication with the server. However, in many cases it is useful for the server to initiate communication, for example, when something occurs. In such a case, *events* provide a mechanism for the server to notify one or more clients of selected occurrences.

To think of this another way, in this tutorial the client is the script and the server is the application (e.g. Xpedition Layout). Events provide a mechanism by which clients can react to server changes and modify the server behavior. Yet another way of thinking about this is that the client (script) defines a function and it then allows the server (application) to call that function.

For example, assume that a script includes an *OnClose* function, and that this function must run after the document closes. When the application closes the document, the application calls the *OnClose* function in the script. The event causes the script to react to a change in the application; that is, the script should not attempt to perform any additional operations on the document because the document is closed.

Note:
The action of the server (application) calling the function in the client (script) is the *event*; the function itself is the *event handler*.

An example of the client using an event to modify the behavior of the server is a *PreClose* function. Assume a script has a *PreClose* function and that the end-user attempts to close a document. When this occurs, the application issues an event to the script that indicates that the application wants to close the document, but that this action has not yet occurred. The script's *PreClose* event handler can then return a value of *True* (allowing the application to close the document) or a value of *False* (preventing the application from closing the document), thereby modifying the behavior of the application.

Type Library

A *type library* defines the *enumerates*, *objects*, *methods*, *properties*, and *events* associated with and supported by a particular server.

Refer to “Creating the Via Count Script” on page 62 for an example of adding a type library to a script.

Prog ID

A *Prog ID* (program identifier) is a predefined string that uniquely identifies a server. The identifier is used to launch a server or to attach a client to an already running server. Furthermore, a *Prog ID* is required to access and extract the enumerates from a server’s type library.

Refer to “Creating the Via Count Script” on page 62 for an example of this usage.

Each automation-enabled application has its own *Prog ID*. A list of the *Prog IDs* associated with each AutoActive tool is as follows:

```
MGCPCB.ExpeditionPCBApplication
MGCPCB.BoardStationREApplication
MGCPCB.BoardStationXEApplication
MGCPCB.DrawingEditorApplication
```

```
MGCPCB.FablinkXEApplication
MGCPCB.PlannerPCBApplication
MGCPCB.SFXREApplication
MGCPCB.ViewerPCBApplication
```

The following lists additional *Prog IDs* of interest:

Note:
Xpedition Layout and Xpedition Designer each use different command bar servers. Scripts must call the correct command bar server for the respective product. Use MGCSDD.CommandBarsEx for Xpedition Layout menus; use Viewlogic.CommandBarsEx for Xpedition Designer menus.

```
MGCSDD.CommandBarsEx
MGCSDD.KeyBindings
```

```
Scripting.FileSystemObject
```

```
MGCPCBEngines.DFFDRC
MGCPCBEngines.DRC
MGCPCBEngines.GeneralInterfaceEngine
MGCPCBEngines.Gerber
MGCPCBEngines.IDFExport
MGCPCBEngines.ManufacturingOutputValidation
MGCPCBEngines.MaskEngine
MGCPCBEngines.NCDrill
MGCPCBEngines.NeutralFileExport
MGCPCBEngines.ODBPPOutputEngine
MGCPCBEngines.PDFOutput
MGCPCBEngines.SilkscreenGen
```

```
MGCPCBLibraries.MaterialsDB
MGCPCBLibraries.PadstackEditorDlg
MGCPCBLibraries.PartsEditorDlg
```

Script

A *script* is a set of ASCII (text) statements that a *script host* processes.

Script Host

A *script host* (sometimes called a *script engine*) is an application that runs a *script*.

Scripting Language

A *scripting language* is a programming language used to create scripts.

There are multiple scripting languages available; "Scripting Languages" on page 27 provides more details. This tutorial uses *Visual Basic Script* (also known as *VBScript* or *VBS*).

Accelerator Key Binding

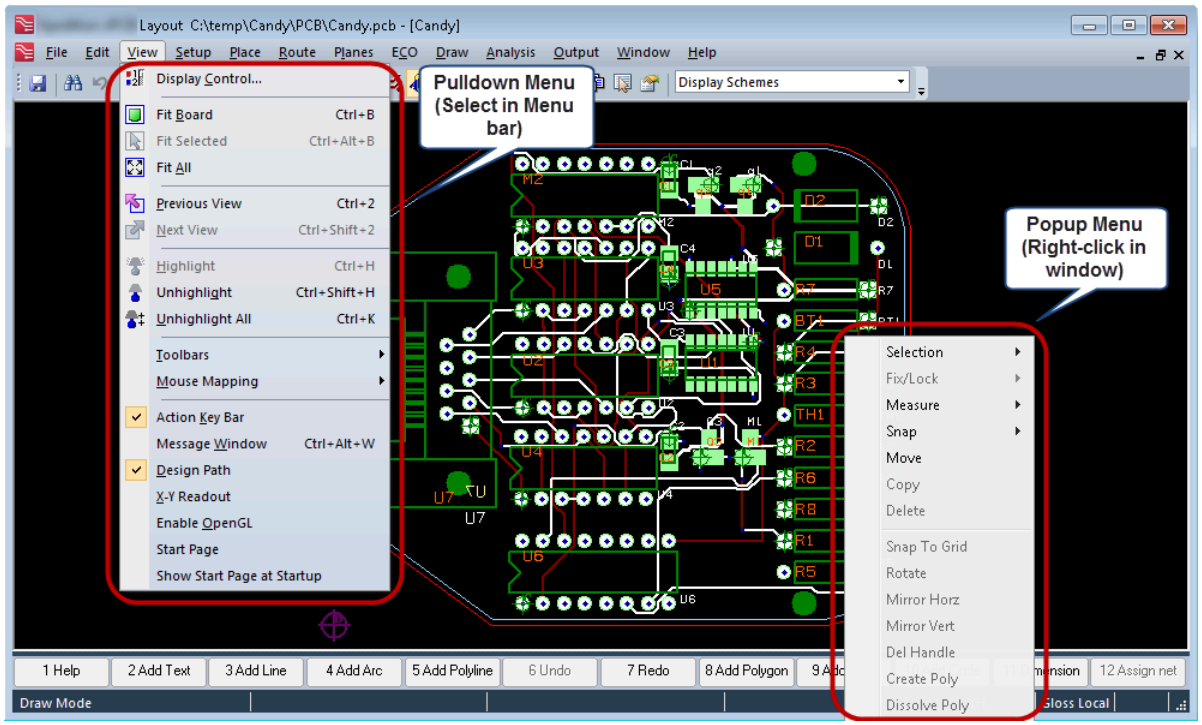
The term *accelerator* refers to a shortcut key on the keyboard that executes functionality such as performing an action or a series of actions. In the context of automation, the term *accelerator* refers to a shortcut key that launches a script.

Menu Item

The term *menu item* or *menu button* refers to a single item in a menu or on the menu bar.

For example, in Figure 3: Xpedition Layout Menus, **View, Display Control**, **Fit Board**, **Fit Selected**, and**Fit All** are menu items.

Figure 3. Xpedition Layout Menus



Pulldown and Popup Menu

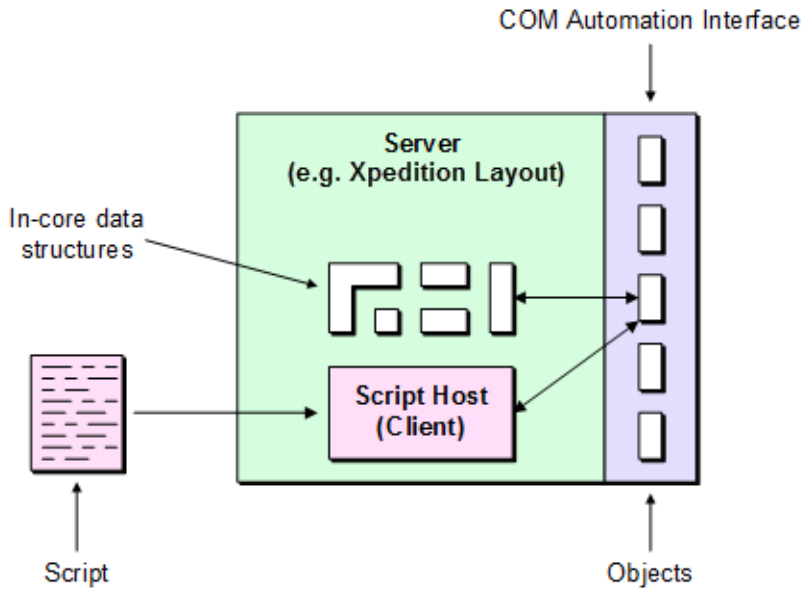
A *pulldown menu* appears when the end-user selects (left-clicks) a menu item from the menu bar. A popup menu appears when the end-user accesses a context-sensitive menu in a session window (right-clicks).

In-Process Client

The term *in-process client* refers to a script host that is running in the same process space as the server it is accessing. For example,

Xpedition Layout is a script host in its own right as is illustrated in Figure 4: In-process Client .

Figure 4. In-process Client

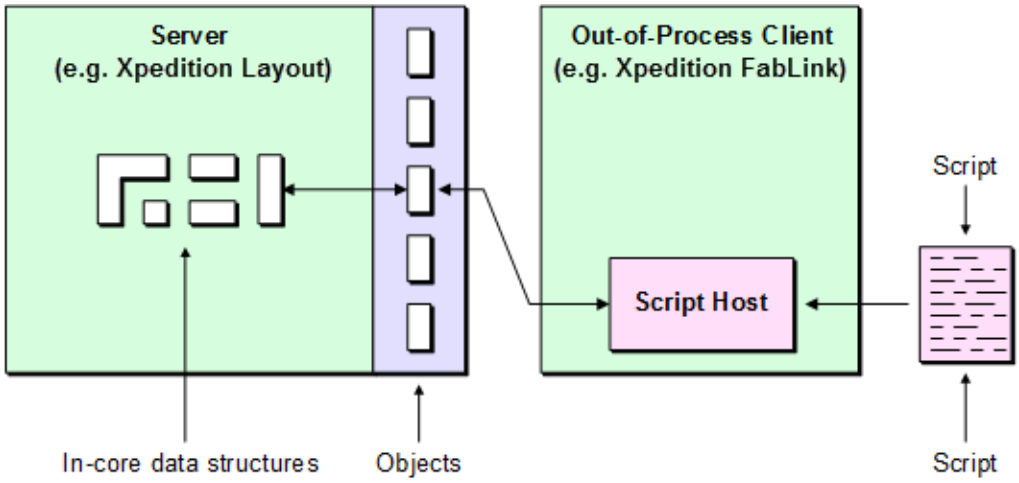


Out-of-Process Client

The term *out-of-process client* refers to a script host that is running in a different process space to the server it is accessing.

For example, consider a situation whereby Xpedition FabLink is acting as an out-of-process client that is accessing and manipulating data in Xpedition Layout (Figure 5: Out-of-process client).

Figure 5. Out-of-process client

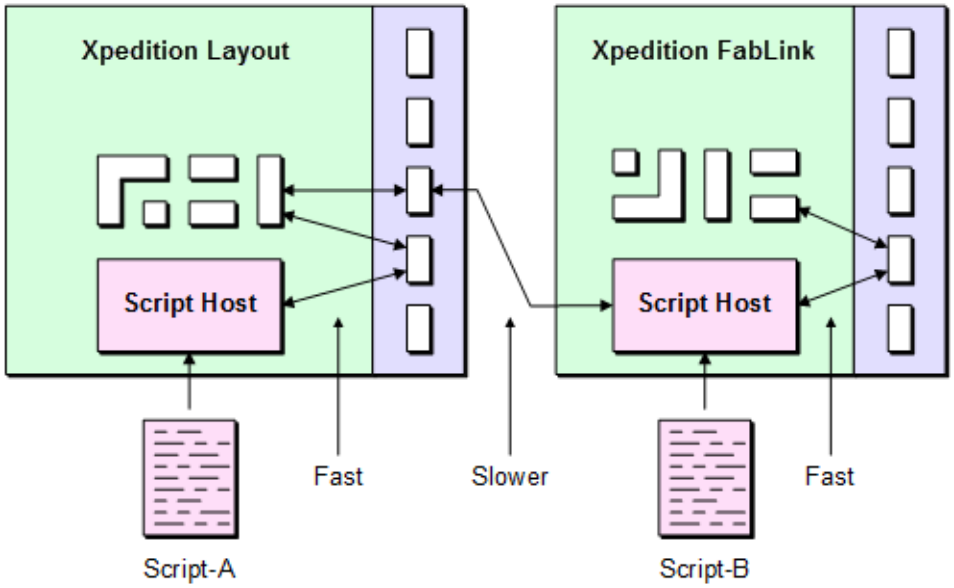


It is possible to create a single script that both Xpedition Layout and Xpedition FabLink run. That is, the out-of-process client (in Figure 5: Out-of-process client) can run the same script that the in-process client (in Figure 4: In-process Client) runs.

CAUTION:
Performance degrades significantly when a script communicates across process boundaries.

Consider the following scenario, a script host in Xpedition Layout is running *Script-A* and is acting as an in-process client to Xpedition Layout. At the same time, a script host in Xpedition FabLink is running *Script-B* and is acting as both an in-process client to Xpedition FabLink and as an out-of-process client to Xpedition Layout (Figure 6: Performance Degrades Across Process Boundaries).

Figure 6. Performance Degrades Across Process Boundaries



Because the Xpedition Layout script host is only accessing and manipulating data inside Xpedition Layout, there are no performance issues. Similarly, when the Xpedition FabLink script host accesses and manipulates Xpedition FabLink data, there are no performance issues. However, when the Xpedition FabLink script host acts as an out-of-process client to access and manipulate data in Xpedition Layout, then there is a negative impact on performance.

Chapter 3
VBScript Primer

This section provides an overview of Scripting Languages, and the VBScript Language.

Scripting Languages
VBScript Language

Scripting Languages

System programming languages such as C, C++, and Java™ are designed to allow programmers to build data structures, algorithms, and – ultimately – applications from the ground up. These languages are strongly typed, which means that all constants and variables used in a program must be associated with a specific data type (such as *Boolean*, *integer*, *real*, *character*, *string*, etc.) that are predefined as part of the language. Furthermore, certain operations may be allowable only with certain types of data.

By comparison, scripting languages are not intended for building applications from scratch. Instead, they assume that a collection of useful “components” already exist, where these components are typically created using a system programming language. You use scripting languages to connect, or glue, these components together, so they are sometimes referred to as *glue languages* or *system integration languages*.

Scripting languages are typically type-less; for example, a variable can hold an integer, then a real, and then a string, because this makes it easier to connect components together. It also speeds the process of script development. Furthermore, code and data are often interchangeable, which means one script can write another script on-the-fly and then execute it.

Scripting languages often perform a series of actions repeatedly; hence their name, which comes from the concept of a written script for a stage play, where the same words and actions are performed identically for each performance.

Possibly the first scripting language was Perl, which was presented to the world in 1987. Perl, which stands for *Practical Extraction and Report Language*, is an interpreted language that is optimized for scanning arbitrary text files extracting information from those text files, and printing reports based on that information (it is also useful for a wide variety of system management tasks).

Today, there are a number of widely used scripting languages, including JScript (short for Java Script), Perl, Python, Tcl (pronounced “tickle”), and VBScript or VBS (short for Visual Basic Script). The file extensions associated with these languages are as follows:

```
JScript          *.js
Perl             *.pls
Python           *.py
Tcl              *.tcl
VBScript         *.vbs
```

This tutorial focuses on VBScript because it is easier to learn. Languages like Perl and Tcl tend to be somewhat cryptic and very difficult to understand for non-programmers.

The script host featured in the Mentor Graphics AutoActive tools, such as Xpedition Layout and Xpedition FabLink, works with all of the listed scripting languages.

Note:
Tcl, Perl, and Python can be used to drive automation only on Windows platforms because COM technology is native to Windows.

When a script runs, the script host determines the language from the script's file extension. VBScript and JScript are native to the script host. In the case of the other languages, the script host checks your computer's registry to determine if you have the appropriate parser/executor module available on your system. If the required module is not on your system, you may obtain it from *www.activestate.com*.

Note:
JScript, Perl, Python, Tcl, and VBScript are typically interpreted. There are also compiled languages that you can use for automation. These languages include Visual Basic, C++, the .Net languages (C#, J#, and VB.Net), and Java (using a special Java/COM bridge). These languages can be used to drive automation only on Windows platforms because COM technology is native to Windows.

VBScript Language

See the following sections.

General-Purpose Statements
Arithmetic, Comparison, and Logical Operators
Control Statements
Functions and Subroutines
User Interaction Functions
Conversion and Formatting Functions
Error Handling

General-Purpose Statements

VBScript is an extremely rich and powerful language.

This chapter is intended to provide only a brief overview of some of the more common statements (refer to "VBScript References and Tools" on page 535).

Option Explicit
Dim—Dimension a Variable
Dim(x,y,z,…)—Dimension an Array of Variables
ReDim(x,y,z,…)—Re-dimension an Array of Variables
UBound—Return the Upper Bound
Set—Setting an Object to a Variable
Call—Calling a Method or Function

Option Explicit

By default, VBScript does not require *Dim* (dimension) statements, but this makes it easy for misspelled variable names to creep into a script. This sort of problem can be extremely time-consuming to track down and resolve. Thus, it is *strongly recommended* that the first statement in your script is *Option Explicit*; for example:

```
Option Explicit
     :
    etc.
```

Using this option means that you are forced to use *Dim* statements to declare your variables. In turn, this means that if you enter a misspelled variable name, the script host reports an error and guides you directly to the problem (refer to"Recommended Scripting Practices" on page 519).

Dim—Dimension a Variable

The *Dim* statement instantiates (declares) a variable. For example, consider the following snippet of code (observe that there are no end-of-statement delimiters):

```
Dim nameStr             ' Declare a variable called nameStr
nameStr = "Fred"        ' Assign a string to the variable
```

VBScript is an un-typed language. If a *string* is assigned to a variable, the variable becomes a string. If an *integer* is later assigned to the same variable, the variable becomes an integer. For example, consider the following snippet of code:

```
Dim nameStr             ' Declare a variable called nameStr
nameStr = "Fred"        ' Assign a string to the variable
  :
nameStr = 10            '  Assign an integer to the variable
```

You can assign two types of entities to variables: *primitive types* (Boolean, integer, real, string) and *object types* (object, collection). In addition, there are implicit object types, which include *Application* and

Scripting (refer to "Creating the Via Count Script" on page 62 for more details on these implicit object types.)

In order to keep track of things and improve the legibility of your code, it is recommended that when you declare a variable you post-fix its name with a short string specifying its primary type. The following are the strings recommended by this tutorial and used in the scripts:

```
BoolBoolean (e.g. 0 / 1 or True / False)' Primitive typeIntInteger (e.g.
 10)' Primitive typeRealReal number (e.g. 3.142)' Primitive typeStrString
 (e.g. "Hello World!")' Primitive typeObjObject' Object typeCollCollection
 (of objects)' Object type
```

For example, you can name a *Trace* object something like *traceObj*, or name a collection of *Trace* objects *traceColl*. Two special objects that appear at the top of the data hierarchy, and that are used in most scripts, are the *Application* and *Document* objects. Because they are used so often, it is common to include these object types in their corresponding variable names; for example, *pcbAppObj* and *pcbDocObj*.

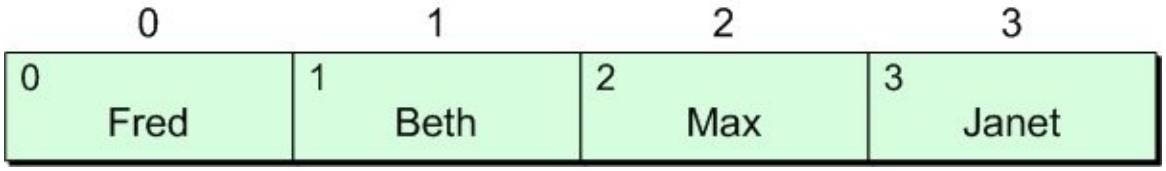
Dim(x,y,z,…)—Dimension an Array of Variables

The *Dim* statement can also be used to instantiate (declare) an array of variables. For example, consider the following snippet of code:

```
Dim nameStrArr(3)   ' Declare an array of variables
nameStrArr(0) = "Fred"
nameStrArr(1) = "Beth"
nameStrArr(2) = "Max"
nameStrArr(3) = "Janet"
```

This first line declares a one-dimensional array comprising four items (numbered from 0 to 3) and then assigns strings to each of these items. You can visualize this as illustrated in Figure 7: One-dimensional Array.

Figure 7. One-dimensional Array



Note:
The value in parentheses is the *upper-bound* of the array. The items in an array are numbered from 0 (zero). Therefore, the items in a four-item array are numbered 0, 1, 2, 3.

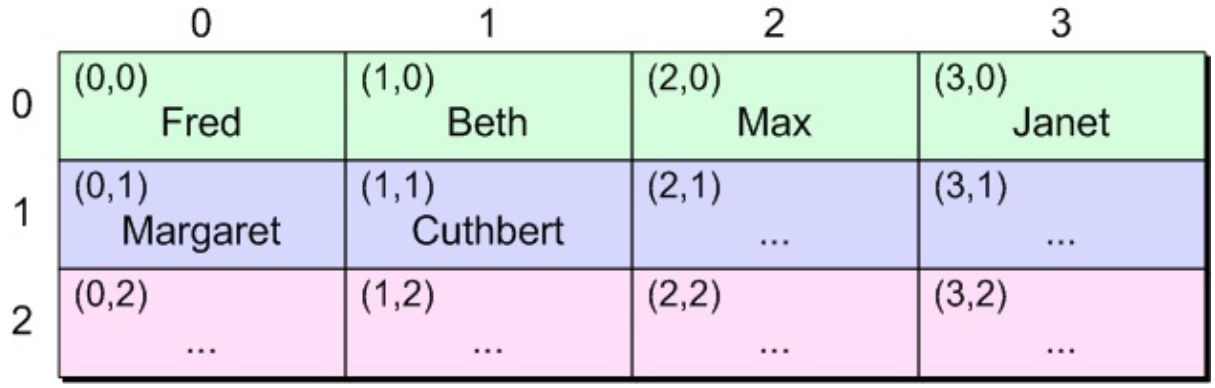
The *Dim* statement can also be used to instantiate multi-dimensional arrays. For example, consider the following snippet of code:

```
Dim nameStrArr(3,2)  ' Declare a 2D array of variables
nameStrArr(0,0) = "Fred"
nameStrArr(1,0) = "Beth"
nameStrArr(2,0) = "Max"
```

```
nameStrArr(3,0) = "Janet"
nameStrArr(0,1) = "Margaret"
nameStrArr(1,1) = "Cuthbert"
    :
   etc.
```

This first line declares a two-dimensional array comprising four “columns” and three “rows”. The remaining lines then assign strings to the array. You can visualize this as illustrated in Figure 8: Two-dimensional Array.

Figure 8. Two-dimensional Array



Similarly, *Dim StrArr(6,4,8)* declares a three-dimensional array comprising seven “columns”, five “rows”, and nine “layers” ... and so forth.

ReDim(x,y,z,…)—Re-dimension an Array of Variables

When parameters are associated with a *Dim* statement as shown in the previous topic, this instantiates (declares) a *static array* whose dimensions cannot be modified. For example, consider the following statement:

```
Dim nameStrArr(4)
```

This declares a one-dimensional array whose size is set as five items numbered from 0 to 4. Now consider the following statement:

```
Dim anotherNameStrArr()
```

In this case, parentheses indicate the instantiation of an array, but no parameters are specified with these parentheses. The result is a *dynamic array* whose size can change by means of the *ReDim* statement. For example, assume that you want the *anotherNameStrArr* to become a two-dimensional array containing a matrix of six by six items; this is achieved using the following statement:

```
ReDim anotherNameStrArr(5,5)
```

Once you apply a *ReDim* statement, you can assign values to the array. It is possible to apply additional *ReDim* statements to the same array; however, any data contained in the array is lost when a new *ReDim* statement is applied. For example, consider the following snippet of code:

```
Dim   myIntArr()      ' Declare the array
ReDim myIntArr(2,3)   ' Specify the array size
myIntArr(0,0) = 64    ' Assign values to the elements
myIntArr(1,0) = 36    '   in the array
    :
ReDim myIntArr(3,6,9) ' Change the size of the array
```

When the second *ReDim* statement is applied to the array, any data that has previously been loaded into the array is lost. This is because applying a *ReDim* statement causes the system to discard the original array and to create a new instantiation from scratch. The only exception to this rule is when the *ReDim* statement is combined with a *Preserve* statement. For example, consider the following snippet of code:

```
Dim   myIntArr()               ' Declare the array
ReDim myIntArr(2,3,5)          ' Specify the array size
myIntArr(0,0,0) = 64           ' Assign values to the elements
myIntArr(1,0,0) = 36           '   in the array
    :
ReDim Preserve myIntArr(2,3,9) ' Change the size of the array
```

The *Preserve* statement ensures that any data already contained in the array is maintained. However, there are restrictions; namely that it is possible only to *increase* the size of the final dimension; for example:

```
Dim   myIntArr()
ReDim myIntArr(2,3,6)
  :
ReDim Preserve myIntArr(2,3,9) ' Legal (last dimension increases)
:
ReDim Preserve myIntArr(2,3,7) ' Illegal (last dimension decreases)
```

Note:
The effects of the *Preserve* statement persist only during the ReDim statement to which it is applied. This means that even after you have applied a ReDim statement with a Preserve statement, you can subsequently apply yet another ReDim statement (with or without a qualifying Preserve statement) if you wish.

Using the *Preserve* statement can negatively effect the performance of your script if you apply it repeatedly. This is because the Preserve statement effectively causes a new array to be constructed from the ground up, it copies your original data into the new version of the array, and it then discards the original version of the array.

UBound—Return the Upper Bound

The *UBound* statement determines the upper bound of an array. For example, consider the following snippet of code:

```
Dim nameStrArr(2)
Dim sizeInt
 :
sizeInt = UBound(nameStrArr)
```

In this example, the variable *sizeInt* contains 2, which is the upper bound of the *nameStrArr* array. Note that you can also write the assignment to *sizeInt* as follows:

```
sizeInt = UBound(nameStrArr,1)
```

In this case, the second parameter associated with *UBound* specifies the dimension of interest. As *nameStrArr* is a one-dimensional array in this example, the only option is to use the number 1. By comparison, consider the following snippet of code:

```
Dim bigNameStrArr(10,20,30)
Dim sizeXInt
Dim sizeYInt
Dim sizeZInt
 :
sizeXInt = UBound(bigNameStrArr,1)
sizeYInt = UBound(bigNameStrArr,2)
sizeZInt = UBound(bigNameStrArr,3)
```

The variables *sizeXInt*, *sizeYInt*, and *sizeZInt* contain 10, 20, and 30, respectively.

Note:
The dimensions associated with an array are numbered from 1; this is different to the items forming each dimension, which are numbered from 0.

Set—Setting an Object to a Variable

The *Set* statement assigns an object reference (object or collection) to a variable as shown in the following code:

```
Dim traceObj
Set traceObj = traceColl.Item(1)
```

Note:
Collections start with item 1, while arrays are numbered from item 0.

The *Set* statement must also be used when a method or function returns an implicit object type such as an *Application* (refer to “Creating the Via Count Script” on page 62 for more details on these implicit object types). This is shown in the following code:

```
' Get the Application object
Dim pcbAppObj
Set pcbAppObj = Application
```

The *Set* statement cannot be used when assigning primitive types to variables as the following code shows:

```
Dim traceWidth
traceWidth = 10   ' We can't use the "Set" Statement here
```

Also note that an enumerate is considered to be a primitive type as it is simply a named integer.

Call—Calling a Method or Function

The *Call* statement passes program control to an explicitly named procedure or function. The *Call* is optional but must be specified whenever parameter lists are enclosed in parentheses. This is shown in the following code:

```
Call traceObj.MoveRelative(10,15)
```

Do not use the *Call* statement when calling a method that has parameters not enclosed in parentheses, for example:

```
traceObj.MoveRelative 10,15 ' No parentheses = No "Call" statement
```

However, it is recommended that you use parentheses and the *Call* statement.

When you call a function that returns a value, the *Call* statement may be used to “discard” that value if you do not wish to use it; that is, if you do not want to assign the returned value to a variable. In such a case, it is optional to enclose any parameters associated with that function, in parentheses. However, it is recommended that you *use* the parentheses.

A *subroutine* never returns a value. In the case of calling a subroutine, the *Call* statement is optional. Furthermore, irrespective as to whether you use the *Call* statement, it is optional to enclose any parameters associated with that subroutine, in parentheses. However, it is recommended that you *use* the parentheses.

Arithmetic, Comparison, and Logical Operators

VBScript features a variety of operators that implement arithmetic, comparison, logical, and string concatenation operations in expressions.

When an expression contains a number of operators, they are evaluated is a specific order. This is referred to as *operator precedence*. You can use parentheses to override the default operator precedence and to force portions of an expression to execute before others. Any operators located inside parentheses evaluate before operators outside those parentheses. In the case of nested parentheses, operators in the deepest level evaluate first, followed by the next deepest, and so forth. Within a pair of parenthesis, standard operator precedence is used.

For expressions that include operators from more than one category evaluate in the following order:

1. Arithmetic operators
2. Comparison operator
3. Logical operators

Arithmetic Operators
Comparison Operators
Logical Operators
String Operators

Arithmetic Operators

VBScript supports a suite of arithmetic operators as shown in the following table.

If an expression contains multiple arithmetic operators, they are evaluated in the following order of precedence:

Table 2. VBScript Arithmetic Operators

Symbol	Description
^	Exponentiation
-	Unary negation
*	Multiplication
/	Division
\	Integer division
Mod	Modulus arithmetic
+	Addition
-	Subtraction

```
' Arithmetic/Assignment Operator example

Dim xInt: xInt = 6
Dim yInt: yInt = 10
Dim ansInt

ansInt = xInt + yInt
```

Note:
In the code snippet, observe how colons separate multiple statements on the same line.

Comparison Operators

VBScript supports a suite of comparison operators as shown in the following table. Comparison operators all have equal precedence, so if an expression contains multiple comparison operators they are evaluated from left-to-right in the order in which they appear.

Table 3. VBScript Comparison Operators

Symbol	Description
=	Equality
<>	Inequality
<	Less-than
>	Greater-than
<=	Less-than or equal-to
>=	Greater-than or equal-to
Is	Object comparison

Note:
In VBScript, the equals (“=”) character is used both as an assignment operator and as the equality comparison operator. In the context of *If*, *Elseif*, and *While* control statements, '=' is interpreted as the comparison operator for equality; outside of these statements it is interpreted as an assignment operator.

```
' Comparison Operator example

Dim xInt: xInt = 6
Dim yInt: yInt = 10
```

```
If xInt < yInt Then
   ...
End If
```

Logical Operators

VBScript supports a suite of logical operators as shown in the following table. If an expression contains multiple logical operators, they are evaluated in the following order of precedence:

Table 4. VBScript Logical Operators

Symbol	Description
Not	Logical negation
And	Logical conjunction
Or	Logical disjunction
Xor	Logical exclusion
Eqv	Logical equivalence
Imp	Logical implication

```
' Logical Operator example

Dim val1Bool: val1Bool = True
Dim val2Bool: val2Bool = False

If val1Bool And val2Bool Then
   ...
End If
```

String Operators

VBScript supports the string concatenation operators shown in the following table.

Table 5. VBScript String Concatenation Operators

Symbol	Description
&	String concatenation
+	String concatenation

```
Dim countInt: countInt = 4

Dim displayStr
displayStr = "There were " & countInt & " items counted."
```

Note:
The difference between these two operators is that the '&' automatically performs any necessary conversions.

Control Statements

You control the flow of your scripts by means of conditional and looping statements that make decisions and repeat actions.

If … Then … End If
If … Then … Else … End If
For … To … Next
For Each … In … Next
While … Wend
Select Case … End Select

If … Then … End If

This control structure executes a statement or block of statements based on an expression that evaluates to a Boolean value of *True*.

In the following code sample, if *nameStr* is equal to “Fred”, then all statements between the *Then* and the *End If* keywords execute.

```
If nameStr = "Fred" Then
   ' You can add zero or more statements here
End If
```

You can place the *If … Then … End If* statement on a single line. If on a single line, the *End If* keyword is optional. The following shows an example of a single line statement:

```
If nameStr = "Fred" Then MsgBox("Hello Fred") End If
```

You can nest multiple *If … Then … End If* statements. The following shows an example of nested statements:

```
If firstNameStr = "Fred" Then
   If lastNameStr = "Blogs" Then
      MsgBox("Hello Fred Blogs")
   End If
End If
```

If … Then … Else … End If

This control structure executes a statement or block of statements based on a Boolean value (*True* or *False*) of an expression. If the expression evaluates to *True* then all statements between the *Then* and the *Else* keywords execute. If the expression evaluates to *False* then all statements between the *Else* and the *End If* keywords execute.

The following sample code shows an example of the *If ... Then ... Else ... End If* control structure:

```
If nameStr = "Fred" Then
   ' You can add zero or more statements here
Else
   ' You can add zero or more statements here
End If
```

For … To … Next

The *For … To … Next* control statement defines a loop that executes a group of statements a specified number of times. The counter variable determines the number of iterations and must have a numeric value. The counter is incremented (or decremented) automatically with each iteration of the loop. The *For* statement specifies the counter variable, the initial counter value and the maximum counter value.

The Step keyword of the For … To statement is optional. If this is omitted, the increment value defaults to +1. However, if the initial counter value is greater than the maximum counter value and you want to count down, then you must use a negative Step qualifier, for example, Step -1.

The following shows an example of the *For ... To ... Next* statement:

```
Dim counterInt, startInt, endInt, stepInt ' See note
startInt = 0 : endInt = 10 : stepInt = 2  ' See note

For counterInt = startInt To endInt Step stepInt
   ' You can add zero or more statements here
Next
```

Note:
In the first line of the code sample (the Dim statement), observe how commas separate multiple variable declarations on the same line.

In the second line of the code sample, observe how colons separate multiple statements on the same line.

The initial counter value, the maximum counter value and the increment value can be variables or a constant integers. For example, the following provides exactly the same function as the original code:

```
Dim counterInt
For counterInt = 0 To 10 Step 2
   ' You can add zero or more statements here
Next
```

The Exit For statement provides a way to prematurely exit a *For* loop. This can be used for example, when a conditional statement determines that the script finished whatever it was trying to do. The Exit For transfers control to the statement following the *Next* keyword. The following shows an example of this statement:

```
Dim counterInt
For counterInt = 0 To 10 Step 2
   ' Before: You can add zero or more statements here
If counterInt > 4 Then Exit For
   ' After: You can add zero or more statements here
Next
```

For Each … In … Next

The *For Each ... In ... Next* control statement loops through items of a collection of elements or an array. The statement starts with the first item in the collection and iterates to the last item. The following shows an example of this control statement:

```
Dim traceObj
For Each traceObj In traceColl
   ' You can add zero or more statements here
Next
```

You can use an End For statement to break out of the loop in exactly the same way as described for the For … To … Next loop.

While … Wend

The *While ... Wend* control statement repeats while a specified condition is True. The following shows an example of *While ... Wend*.

```
Dim tempInt
tempInt = 10
While tempInt > 0
   ' You can add zero or more statements here
   tempInt = tempInt - 1

Wend
```

Note:
There are also Do … Until statements that repeat until a condition exists and Do … While statements that repeat while a condition exists. However these statements are used less often and are not discussed here.

Select Case … End Select

The Select Case control statement is useful when there more than a few conditions to check. The general form of the Select Case statement is as follows:

```
Select Case <variable or expression>
   Case <value>
      ' You can add zero or more statements here
   Case <value>
```

```
      ' You can add zero or more statements here
   Case Else
      ' You can add zero or more statements here
End Select
```

In the example, <variable or expression> is either a single variable or an expression that resolves to a primitive type (integer, string, real, Boolean) or an enumerate (which resolves to an integer).The following illustrates the Select Case statement:

```
Select Case nameStr
   Case "Fred"
      ' You can add zero or more statements here
   Case "Bert"
      ' You can add zero or more statements here
   Case Else
      ' You can add zero or more statements here
End Select
```

The Case Else keyword, which is optional, indicates the statements to execute if there is no between the expression and the case items. It acts as a “catch-all”.

Functions and Subroutines

VBScript recognizes functions, which are declared with the *Function* ... *End Function* construct, and subroutines, which are declared with the *Sub* ... *End Sub* construct. Functions return a value; subroutines do not. Both function and subroutines can have parameters, which are In-Out parameters.

Functions and subroutines make your scripts easier to read, understand, manage, and reuse. Whenever you are performing a task that is repeated multiple times in your script, you should consider declaring this code as a function or a subroutine.

Function and End Function
Sub and End Sub

Function and End Function

Assume that you want to create a function named AddTwoInts that accepts two integers as parameters, adds these integers together, and displays the result. The following script achieves this:

```
Function AddTwoInts(num1Int, num2Int)
   Dim tempInt
   tempInt = num1Int + num2Int
   MsgBox( "The result is " & tempInt)
End Function
```

Note:
In some scripts, Private or Public keywords are inserted before the Function keyword. However, these keywords have no impact outside of a class declaration, and VBScript classes is an advanced topic that is outside the scope of this tutorial.

To call the function from elsewhere in the script use the following statement:

```
Call AddTwoInts(6,10)
```

In this example two integer constants are passed into the function, but these can be integer variables.

To return a value, assign the value to the function name itself:

```
Function AddTwoInts(num1Int, num2Int)
   Dim tempInt
   tempInt = num1Int + num2Int
   AddTwoInts = tempInt
End Function
```

Observe the last line before the End Function statement (in bold). The value of the variable tempInt is assigned to the function name, so the value is returned to the outside world. To use this function:

```
Dim totalInt
totalInt = AddTwoInts(6,10)
```

To exit a function early, use the Exit Function statement. For example, suppose you are only interested in adding two integers together as long as the result is less than or equal to 100:

```
Function AddTwoInts(num1Int, num2Int)
    Dim tempInt
    tempInt = num1Int + num2Int
    If tempInt > 100 Then Exit Function
    AddTwoInts = tempInt
End Function
```

If you exit a function before completion, and if the return value is not set, then the variable to which the return value is being assigned in the statement calling this function remains unchanged.

If a function returns a value, the script does not have to use it. For example:

```
Dim totalInt
totalInt = AddTwoInts(6,10) ' (a) Return value is used
Call AddTwoInts(7,11)       ' (b) Return value is discarded
```

The first time AddTwoInts is called (a), its return value is assigned to the integer variable totalInt; the second time AddTwoInts is called (b), its return value is ignored by the script.

A function (or a subroutine) does not require parameters. For this case, parentheses are optional when declaring the function. However, it is recommended to use parentheses for the sake of consistency. For example:

```
Function SillyFunction()
    MsgBox("I'm a silly function")
End Function
```

Similarly, it is a good programming practice to use the parentheses when calling a function from elsewhere in a script. This helps to emphasize that it is a function call, for example:

```
SillyFunction()
```

Sub and End Sub

Subroutines are very similar to functions. The main difference between subroutines and functions is that a subroutine cannot be used to return a value.

Sub replaces Function

End Sub replaces End Function

Exit Sub (if used) replaces Exit Function

User Interaction Functions

Two common VBScript functions perform user interaction for input and output operations:

- **InputBox()** — for input
- **MsgBox()** — for output

InputBox()
MsgBox()

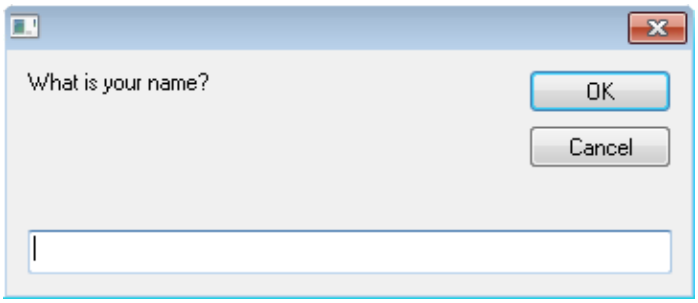
InputBox()

The *InputBox()* function displays a dialog, containing a string, which prompts the end-user about the data expected. The simplest way to use the InputBox() function is to call it with its first parameter set to the query string. For example, consider the following code:

```
Dim nameStr
nameStr = InputBox("What is your name?")
```

When the script performs the InputBox() statement, it displays the dialog and then pauses until the end-user clicks OK, as shown in Figure 9: Input Box Dialog—InputBox() .

Figure 9. Input Box Dialog—InputBox()



There are other parameters to the InputBox() function. These perform actions such as setting the title of the dialog, specifying the dialog location on the screen, and so forth.

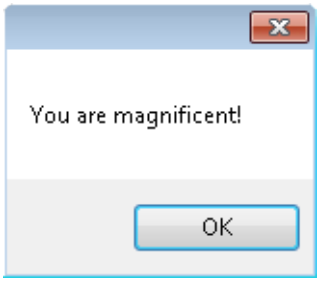
MsgBox()

The *MsgBox()* function displays a dialog containing a message for the end-user. The simplest way to use the MsgBox() function is to call it with its first parameter set to be the message string. For example, consider the following statement:

```
MsgBox("You are magnificent!")
```

This causes the script to display the string “You are magnificent!” in a dialog and then pauses until the end-user clicks OK (Figure 10: Message Box Dialog—MsgBox()).

Figure 10. Message Box Dialog—MsgBox()



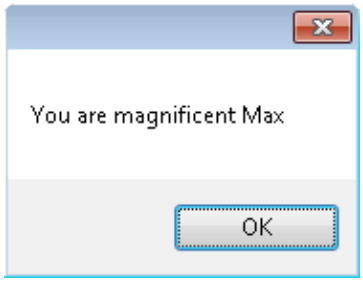
The following code snippet combines the InputBox() and MsgBox() functions:

```
Dim nameStr
nameStr = InputBox("What is your name?")
MsgBox("You are magnificent " & nameStr)
```

Note the ampersand ('&') character. This concatenates two strings together (actually, a string and a variable containing a string, in this example) and returns a single string that acts as the first parameter to the MsgBox() function.

Assume that the end-user enters the name “Max” into the InputBox() function. In this case, the MsgBox() displays “You are magnificent Max” in the dialog (Figure 11: “You are magnificent” Dialog).

Figure 11. “You are magnificent” Dialog



There are other parameters to the MsgBox() function. These allow you to perform actions such as setting the title of the dialog, specifying its location on the screen, and so forth.

An alternative to using the '&' operator to concatenate two strings is to use the '+' operator. The difference between these two operators is that the '&' automatically performs any necessary conversions; for example, consider the following snippet of code:

```
Dim tempInt
tempInt = 6
MsgBox("My lucky number is " & tempInt)
```

In this case, the '&' operator automatically converts the integer value 6 into its textual (string) equivalent before performing the concatenation.

Conversion and Formatting Functions

VBScript features a large number of conversion, string manipulation, and number formatting functions. A few simple examples of the more commonly used functions are presented in this section.

Conversion Functions
String Manipulation Functions
Special Characters
Format Number Function
Date and Time Functions

Conversion Functions

Not surprisingly, the conversion functions convert data from one type to another. This section presents examples of a few of these.

CInt()
CDbl()
CStr()

CInt()

Suppose you have a string “2468” that you want to convert into an integer; you do this using the CInt() function as demonstrated in the following snippet of code:

```
Dim testStr, testInt
testStr = "2468"
testInt = CInt(testStr)
```

CDbl()

The CDbl() function converts numeric or string values into real (double-precision) numbers. For example, suppose you have a string “3.142” that you want to convert into a real number; you do this using the CDbl() function as demonstrated in the following snippet of code:

```
Dim testStr, testReal
testStr = "3.142"
testReal = CDbl(testStr)
```

CStr()

Suppose you have an integer variable that you want to convert to a string. In this case, you use the CStr() function; for example:

```
Dim testSrt, testInt
testInt = "2468"
testSrt = CStr(testInt)
```

The CStr() function converts from any format into a string; for example, a real number like 3.142 are automatically converted into its string equivalent of “3.142”, a Boolean variable is converted into the words True or False, and so forth. In fact, the '&' string concatenation operator automatically calls the CStr() function if required.

String Manipulation Functions

VBScript provides a variety of functions that manipulate strings.

Len()
Left() and Right()
Replace()

Len()

The Len() function returns the length of a string:

```
Dim testStr, lenInt
testStr = "Hello World"
lenInt = Len(testStr)
```

In this example, the lenInt integer variable contains 11, which is the number of characters in the testStr string.

Left() and Right()

The Left() function returns a substring containing the specified number of leftmost characters of a string; for example:

```
Dim mainStr, subStr
mainStr = "Hello World"
subStr = Left(mainStr,5)
```

In this example, the substring subStr contains the leftmost five characters from mainStr; that is, subStr contains the word Hello.

The Right() function is the counterpart to the Left() function; the Right() function returns a substring containing the specified number of rightmost characters of a string; for example:

```
Dim mainStr, subStr
mainStr = "Hello World"
subStr = Right(mainStr,5)
```

In this example, the substring subStr contains the rightmost five characters from mainStr; that is, subStr contains the word World.

The first parameter to the Len(), Left(), and Right() functions is shown as being a string variable, but this can also be a string. Similarly, the second parameter to the Left() and Right() functions is shown as being an integer, but this can also be an integer variable. Consider the following snippet of code, which generates exactly the same result as the previous example:

```
Dim tempInt, subStr
tempInt = 5
subStr = Right("Hello World", tempInt)
```

Replace()

The Replace() function replaces one substring with another substring, where these substrings are not necessarily of the same length. For example, consider the following snippet of code:

```
Dim mainStr
mainStr = "You have a big plate"
mainStr = Replace(mainStr, "big", "small")
```

The first parameter is the source string, the second parameter is the substring to replace, and the third parameter is the replacement substring. Therefore, in this example, mainStr contains: You have a small plate.

The parameters to the *Replace()* function may be strings or string variables. The following snippet of code generates exactly the same result as the previous example:

```
Dim mainStr, origSubStr, newSubStr
origSubStr = "big"
newSubStr = "small"
mainStr = Replace("You have a big plate", origSubStr, newSubStr)
```

Special Characters

VBScript recognizes and supports a number of intrinsic constants that represent special characters. These are used for processing textual data or formatting output using with string manipulations.

The Chr() function converts an integer (or an integer variable) into a character. For example, Chr(9) returns a horizontal <Tab> character. For example, to replace a substring of four spaces in a string with a <Tab> character, do the following:

```
Dim mainStr
mainStr = "The answer is    64"
mainStr = Replace(mainStr, "    ", Chr(9))
```

The following lists some common character/string constants and their equivalent codes:

```
vbCrChr(13)Carriage returnvbLfChr(10)Line feedvbCrLfChr(13)Chr(10)Carriage
 return and line feedvbTabChr(9)Horizontal tab
```

There is also the very useful vbNewLine constant, which is platform-independent. That is, it either equals Chr(10) or Chr(13)Chr(10) depending on the operating system under which your script is running (Windows or UNIX/Linux).

Format Number Function

The FormatNumber() function formats a number. The general form of function is demonstrated in the following pseudo-code snippet:

```
Dim numberStr
numberStr =
 FormatNumber(number,decimalPlaces,includeLeadingZero,useParenthesis)
```

The first parameter, shown as *number*, is a numeric expression, which can be an integer, a real number, a numeric variable, or an expression based on numbers or variables that resolves to a number. Only the first parameter is required, the rest are optional.

The second parameter, *decimalPlaces*, specifies the number of digits the number should contain after the decimal point. For example:

```
Dim numberStr
numberStr = FormatNumber(3.142857,3)
```

The numberStr variable contains 3.143. The value is rounded during the process of discarding the unwanted digits.

The third parameter is a Boolean (*includeLeadingZero*); a value of True indicates to display a leading zero, while a value of False prevents the display of a leading zero. For example, consider the following snippet of code:

```
Dim yesStr, noStr
yesStr = FormatNumber(0.654,2,True)  ' yesStr = "0.65"
noStr  = FormatNumber(0.654,2,False) ' noStr =  ".65"
```

The fourth parameter is a Boolean (*useParenthesis*); a value of True indicates to display negative values in parentheses, while a value of False causes negative values to display using a minus sign. For example, consider the following snippet of code:

```
Dim yesStr, noStr
yesStr = FormatNumber(-3.142,2,,True)  ' yesStr = "(3.14)"
noStr  = FormatNumber(-3.142,2,,False) ' noStr = "-3.14"
```

Observe that any parameters that are not specified (as with the third parameter in the example) assumes their default values, which are defined as part of the host computer's regional settings.

Date and Time Functions

The Date() and Time() functions are useful when creating reports. The Date() function returns the current system date. For example:

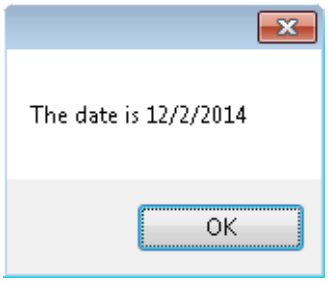
```
Dim dateStr
dateStr = Date()
MsgBox("The date is " & dateStr)
```

You can also use the function as part of the argument to the MsgBox(). For example:

```
MsgBox("The date is " & Date())
```

How the Date() is formatted depends on the regional settings associated with the host computer (month/day/year or day/month/year). This format can be modified using the FormatDateTime() function as discussed below. Figure 12: Display the Date Dialog displays the result of the *Date()* function for the previous code snippet.

Figure 12. Display the Date Dialog

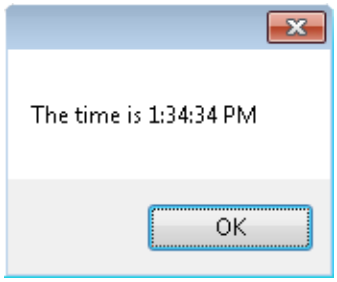


Similarly, the Time() function returns the current system time. For example:

```
MsgBox("The time is " & Time())
```

By default, the Time() function returns the date in the form hour:minute:second AM/PM. This format can be modified using the FormatDateTime() function as discussed below. Figure 13: Display the Time Dialog displays the result of the statement.

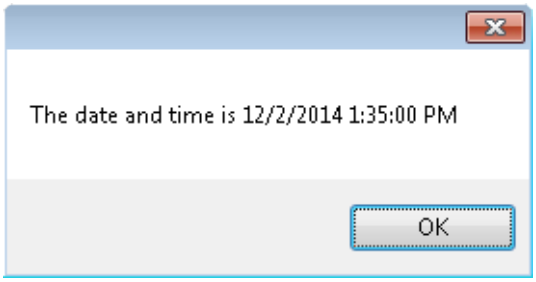
Figure 13. Display the Time Dialog



The Now() function returns the current date and time based on the system settings. The following two statements provide identical results (Figure 14: Display Date and Time Dialog):

```
MsgBox("The date and time is " & Date() & " " & Time())
MsgBox("The date and time is " & Now())
```

Figure 14. Display Date and Time Dialog



The FormatDateTime() function modifies the format of the date or the time. The first parameter to the FormatDateTime() function is typically either the Date() or Now() functions – for modifying the date format, or either the *Time()* or *Now()* – for modifying the time format. For example:

```
Dim tempStr
tempStr = FormatDateTime(Date(),<enumerate>) ' Format date
tempStr = FormatDateTime(Time(),<enumerate>) ' Format time
tempStr = FormatDateTime(Now(),<enumerate>)  ' Format date or time
```

The second parameter of the FormatDateTime() function is an enumerate (or an integer) that specifies the type of formatting to perform. The supported enumerate names, equivalent integer values, and corresponding results are as follows:

```
Enumerate NameValueUse WithExample Output
vbGeneralDate0Date or Now9/7/2006vbLongDate1Date or NowThursday, September
 07, 2006vbShortDate2Date or Now9/7/2006vbLongTime3Time or Now11:30:59
 AMvbShortTime4Time or Now11:30
```

Note:
The first parameter to the FormatDateTime() function can be a string or a string variable containing the date and/or time.

Error Handling

This topic relates to the way in which VBScript handles run-time (execution) errors.

The *Err* object is an object with global scope, meaning you do not need to create an instance of it within your script. The *Err* object contains methods and properties that provide information about a single runtime error. When an error occurs within your script the properties and methods of the *Err* object are assigned values that your script can access.

When running a script, if an error occurs, then by default the script aborts and the error displays either in a MsgBox (in the case of an in-process script) or on the command line (in the case of an out-of-process script).

In some cases you may not want the script to abort because you recognize that what you are trying to do may cause an error and you want to handle it within the script itself. The way to do this is to use the On Error Resume Next statement as follows:

```
' You may have some statements here
On Error Resume Next
' One or more statements that may cause an error go here
If Err.Number <> 0 Then
' One or more statements to handle the error
End If
```

The If… Then… End If statement tests against a value of 0, which is equivalent to a Boolean value of False. Any non-zero value is equivalent to a Boolean value of True. This means that the following code snippet is functionally equivalent to the previous example:

```
' You may have some statements here
On Error Resume Next
' One or more statements that may cause an error go here
If Err.Number Then
   ' One or more statements to handle the error
End If
```

The Number property of the Err object returns an error code. Different integers correspond to different error codes, and these integers are mapped to enumerates. For example, the enumerate for a DRC error in Xpedition Layout is epcbErrCodeDRCViolation. This means you can rewrite the test as follows:

```
' You may have some statements here
On Error Resume Next
' One or more statements that may cause an error go here
If Err.Number = epcbErrCodeDRCViolation Then
   ' One or more statements to handle the error
End If
```

The error code enumerates are defined in the Automation interface. (refer to "VBScript References and Tools" on page 535 for more information on using Object Browsers to locate this type of information). A list of these enumerates, their values, and their descriptions is located in the Automation documentation as EPcbErrCode.

As an alternative to using the Number property of the Err object, you might decide to use the Description property. For example, consider the following code snippet:

```
' You may have some statements here
On Error Resume Next
' One or more statements that may cause an error go here
If Err.Description = "DRC Violation" Then
   ' One or more statements to handle the error
End If
```

It is recommended to test the EPcbErrCode enumerate because it is guaranteed to remain the same from software release to release. It is unlikely that the Description string will change though there is no guarantee that it will not. Another use of Description property is when writing the error to a log file. For example, assume that there is a WriteErrorToLogFile() function that writes errors to a log file as in the following snippet of code:

```
' You may have some statements here
On Error Resume Next
' One or more statements that may cause an error go here
If Err.Number = epcbErrCodeDRCViolation Then
   Call WriteErrorToLogFile(Err.Description)
End If
```

After the script has handled an error, you can use the Clear method of the *Err* object to reset the properties of the Err object. For example:

```
' You may have some statements here
On Error Resume Next
' One or more statements that may cause an error go here
If Err.Number = epcbErrCodeDRCViolation Then
   ' One or more statements to handle the error
End If
Err.Clear
```

The technique of using the Err.Clear statement after handling an error might be viewed as “cleaning up our own mess”. This works if (a) you always do error-handling this way and (b) the script never calls an external script and/or library routine. (The concept of scripts calling other scripts and/or library routines is discussed later in this tutorial).

However, if the script has called an external script and/or library routine, then you cannot be sure that the external script has “cleaned up their mess.” To address this, use the Err.Clear statement just after the On Error Resume Next statement (*before* generating and handling the error) to ensure that the script does not respond to an error that occurred earlier in the script. For example:

```
' You may have some statements here
On Error Resume Next
Err.Clear
' One or more statements that may cause an error go here
If Err.Number = epcbErrCodeDRCViolation Then
   ' One or more statements to handle the error
End If
```

The technique of using the Err.Clear statement before handling an error might be viewed as “cleaning up someone else's mess AND cleaning up our own mess”. That is, this covers the case of the script calling an external script, and it also works for the script itself as long as it always handles errors in this manner.

A potential problem remains with this technique. Suppose that the script generates and responds to an error, and then it calls an external script and/or library routine. If this “third party code” attempts to perform error-handling, and if it performs the technique of applying the Err.Clear statement *after* it has handled an error, then it may detect a “false error” from the calling script (that is, an error that is already addressed but not cleared).

Therefore, the absolute safest practice is to use the Err.Clear statement both *before* handling an error (thereby cleaning up anyone else's mess) and also *after* handling an error (thereby cleaning up our own mess). For example:

```
' You may have some statements here
On Error Resume Next
Err.Clear
' One or more statements that may cause an error go here
If Err.Number = epcbErrCodeDRCViolation Then
   ' One or more statements to handle the error
End If
Err.Clear
```

In practice (and as seen throughout the remainder of this tutorial), it is generally assumed that all scripts use the same technique, which is to use the Err.Clear statement *before* generating and handling any errors.

Finally, if you want to return your script to the mode where an error causes the script to abort, you should use the On Error GoTo 0 statement. For example:

```
' You may have some statements here
Err.Clear
On Error Resume Next
' One or more statements that may cause an error go here
If Err.Number = epcbErrCodeDRCViolation Then
   ' One or more statements to handle the error
End If
On Error GoTo 0
' The rest of your script goes here
```

The scope of an On Error Resume Next statement is the routine in which it is called. When the calling routine terminates, the system automatically returns to the default error-handling mode. This is equivalent to using an On Error Goto 0 statement before the end of the routine. In addition, the system automatically returns to the default error-handling mode within any called routines. However, in this case the system returns to the On Error Resume Next mode when the called routine returns control to the calling routine.

The effects of On Error Resume Next apply only within the routine in which this statement is found. Any other routines (called or calling) use the default error mode unless otherwise specified within those routines.

Chapter 4
Running Your First Script

This section guides you through the process of running a simple “Hello World” script “Creating and Running the Hello World Script.”

You can create and run a script that counts the number of vias in a design. This script is the basis for scripts throughout this tutorial.

Creating and Running the Hello World Script

Creating and Running the Hello World Script

This script simply calls the VBScript *MsgBox*() function to display the string “Hello World” on the screen. You will be calling your script from within the Xpedition Layout application. Because this script does not access or modify any of the application's internal data structures, it does not require any special server validation code.

Creating the Hello World Script
Running the Hello World Script
Creating and Running the Via Count Script
Understanding the Data Hierarchy
Creating the Via Count Script
Running the Via Count Script
An Alternative Technique

Creating the Hello World Script

Xpedition Layout and other products in the flow support a built-in editor called the Integrated Development Environment (IDE)

Refer to “Creating Forms and Dialogs Using the IDE” on page 69. There is also a wide variety of third-party editors available for purchase or for free download. Each of these third-party editors offers a different set of capabilities and features.

In reality, scripts are ASCII files that you can create and modify using any text editor. If you are running under the Windows operating system, for example, invoke the Notepad editor and use it to enter the following statement:

```
MsgBox("Hello World")
```

This single statement is a very simple, one-line script. Choose **File > Save As** to save this file as HelloWorld.vbs in the *C:\Temp* folder on your system.

Running the Hello World Script

Run the script from within Xpedition Layout.

“Running Scripts from an Application” on page 161 details this technique.

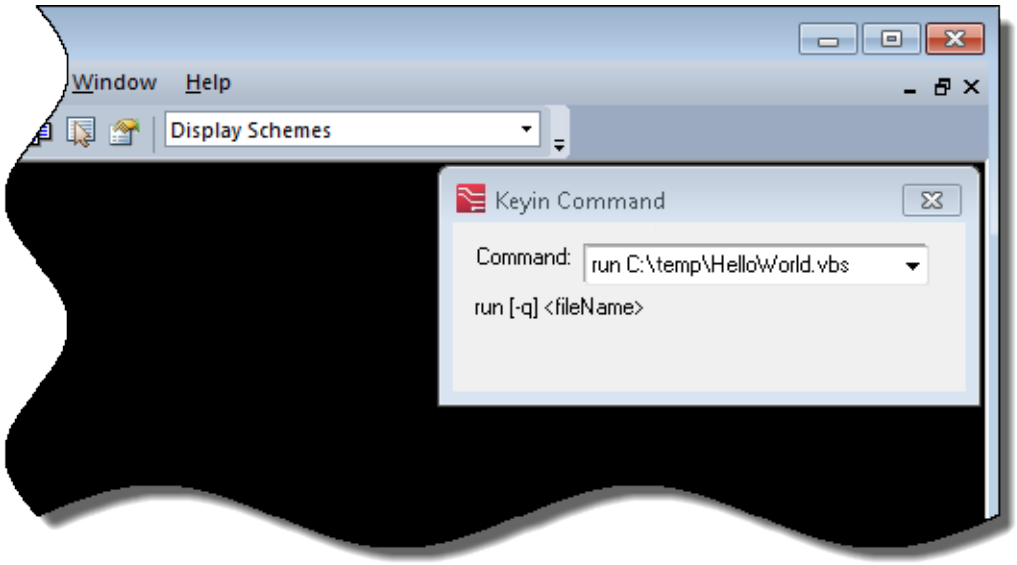
1. Launch Xpedition Layout.

   Some scripts require a design to be open in order to perform the script’s function. Other scripts do not require a design to be open. If a script does require a design to be open, you can either leave this task to the end-user, or you can write the script in such a way that it checks if a design is already open and, if not, to either open a design automatically or to prompt the end-user to do so.

2. Right-click the toolbar and ensure that the **Keyin Command** option is enabled.

3. Enter the string run *C:\temp\HelloWorld.vbs* into the **Keyin Command** field (Figure 15: Keyin Command Field).

Figure 15. Keyin Command Field



Note that, this string is case-insensitive on Windows platforms and case-sensitive on UNIX and Linux platforms.

If there are any spaces in the path name to the file, then the entire path must begin and end with double quotes (this does not include the space between the run keyword and the path name.)

4. Press Enter to run the script. The dialog displays “Hello World” (Figure 16: “Hello World” Dialog).

Figure 16. “Hello World” Dialog

5. Click **OK** to dismiss the dialog. (This also ends this script, because there is nothing else to do.)

Creating and Running the Via Count Script

This script counts and displays the number of vias in the layout design that is currently open and active in the Xpedition Layout application. As part of the process of creating this script, you use the Help system

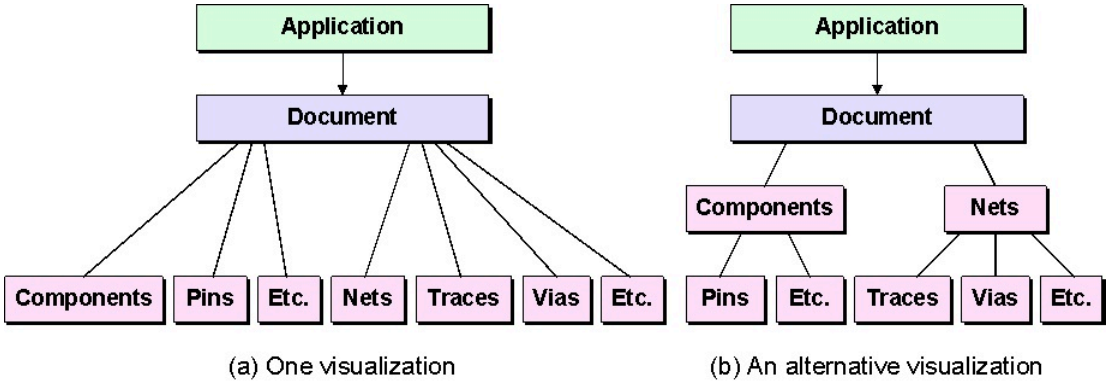
to work your way through the application's data structures. Because this script accesses the application's internal data structures, it does require some special server validation code.

Understanding the Data Hierarchy

The data structures inside Mentor Graphics applications are organized in a hierarchy. Each application may represent its data structures and hierarchy in a different way. For this discussion, consider Xpedition Layout data structures.

At the top of the hierarchy is the Application. Think of this as Xpedition Layout running without having a layout design open. Next in the hierarchy is a Document, or a layout design in Xpedition Layout. You can visualize the data residing in the *Document* in multiple ways; two examples are as illustrated in Figure 17: Visualizing the Data Hierarchy in Xpedition Layout.

Figure 17. Visualizing the Data Hierarchy in Xpedition Layout



In reality, all design data is "owned" by the Document. This means that you can access the Vias, for example, using Document.Vias as shown in Figure 17: Visualizing the Data Hierarchy in Xpedition Layout(a). Alternatively, you can visualize Vias as residing "under" Nets in the hierarchy, in which case you can access the Vias using Document.Nets.Vias as shown in Figure 17: Visualizing the Data Hierarchy in Xpedition Layout(b).

The set of Vias returned by the Document and the Net are different. This is because all of the Vias on the board return when accessing the Document; by comparison, only the Vias on a Net return when accessing a Net.

Creating the Via Count Script

Use the Notepad editor or the text/scripting editor of your choice to enter the following script. Then choose **File > Save As** to save this file as ViaCount.vbs in the *C:\temp* folder on your system.

```
Option Explicit

' Add any type libraries to be used.
Scripting.AddTypeLibrary("MGCPCB.ExpeditionPCBApplication")

' Get the Application object
Dim pcbAppObj
Set pcbAppObj = Application

' Get the active document
```

```
Dim pcbDocObj
Set pcbDocObj = pcbAppObj.ActiveDocument

' License the document
ValidateServer(pcbDocObj)

' Get the vias collection
Dim viaColl
Set viaColl = pcbDocObj.Vias

' Get the number of vias in collection
Dim countInt
countInt = viaColl.Count

MsgBox("There are " & countInt & " vias.")

''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
'Local functions

' Server validation function
Function ValidateServer(docObj)

    Dim keyInt
    Dim licenseTokenInt
    Dim licenseServerObj

    ' Ask Expedition’s document for the key
    keyInt = docObj.Validate(0)

    ' Get license server
    Set licenseServerObj =
                  CreateObject("MGCPCBAutomationLicensing.Application")

    ' Ask the license server for the license token
    licenseTokenInt = licenseServerObj.GetToken(keyInt)

    ' Release license server
    Set licenseServerObj = nothing

    ' Turn off error messages (validate may fail if the
    ' token is incorrect)
    On Error Resume Next
    Err.Clear

    ' Ask the document to validate the license token
    docObj.Validate(licenseTokenInt)
    If Err Then
        ValidateServer = 0
    Else
        ValidateServer = 1
    End If

End Function
```

Note:
Observe the statement that starts on Line 41 and continues on the following line (without a line number). This actually a single statement that occupies a single line in the script; it is displayed here using two lines only for purposes of formatting and presentation in this document. The same is true for the comment that starts on Line 49.

The next task is to step through the first half of this script line-by-line. The script starts with the Option Explicit statement, discussed in "VBScript Primer" on page 27 and "Option Explicit Statement" on page 522.

```
1  Option Explicit
```

Line 4 uses the Scripting object. This global object is always present and available to scripts running on Mentor Graphics applications. You do not have to define or declare the Scripting object, because it is already defined. There are a number of methods and properties associated with the Scripting object (refer to the Layout Automation Reference for more details).

```
3  ' Add any type libraries to be used.
4  Scripting.AddTypeLibrary("MGCPCB.ExpeditionPCBApplication")
```

The *AddTypeLibrary* method provides access to any enumerate values associated with a specified server. The Xpedition Layout server is specified using its Prog ID (this is MGCPCB.ExpeditionPCBApplication as discussed in Prog ID.

Line 7 declares a PCB Application object named pcbAppObj, and Line 8 assigns the application to this object.

```
6  ' Get the Application object
7  Dim pcbAppObj
8  Set pcbAppObj = Application
```

Similar to the Scripting object, Application is an implicit object that is always present and available to scripts running on Mentor Graphics applications. You do not have to define or declare the Application object, because it is already there.

It is simplest to run scripts within the application because the script attaches to that application. If the script runs outside the application and there are multiple instantiations of the application, then the script attaches to an arbitrary application instantiation).

This script assumes that the end-user opens a layout design prior to running the script. Assuming this, Line 11 declares a variable named pcbDocObj, and Line 12 sets the *Document* object the ActiveDocument property returns to this variable.

```
10  ' Get the active document
11  Dim pcbDocObj
12  Set pcbDocObj = pcbAppObj.ActiveDocument
```

On Line 15, the script calls the ValidateServer function passing the pcbDocObj object as a parameter. This user-declared function accesses the license server in order to acquire any necessary licenses required to run automation.

CAUTION:
The script must perform validation before the script attempts to use any methods and properties on the Document object.

```
14  ' License the document
15  ValidateServer(pcbDocObj)
```

Note:
VBScript allows you to call functions and subroutines before they are formally declared in the script.

Line 18 declares a variable named viaColl, and Line 19 assigns the collection of the vias associated with the Document object to viaColl.

```
17  ' Get the vias collection
18  Dim viaColl
19  Set viaColl = pcbDocObj.Vias
```

The via collection's Count method returns the number of vias in the collection. Line 22 declares an integer variable named countInt, and Line 23 uses the viaColl.Count statement to assign the number of vias in the design to *countInt*.

```
21  ' Get the number of vias in collection
22  Dim countInt
23  countInt = viaColl.Count
```

Finally, Line 25 calls the MsgBox() function to display a dialog with the total number of vias in the design.

```
25  MsgBox("There are " & countInt & " vias.")
```

Lines 31 and 61 in this script, declare the server validation function. It is not important to detail the server validation function. Simply copy it into scripts as required.

Running the Via Count Script

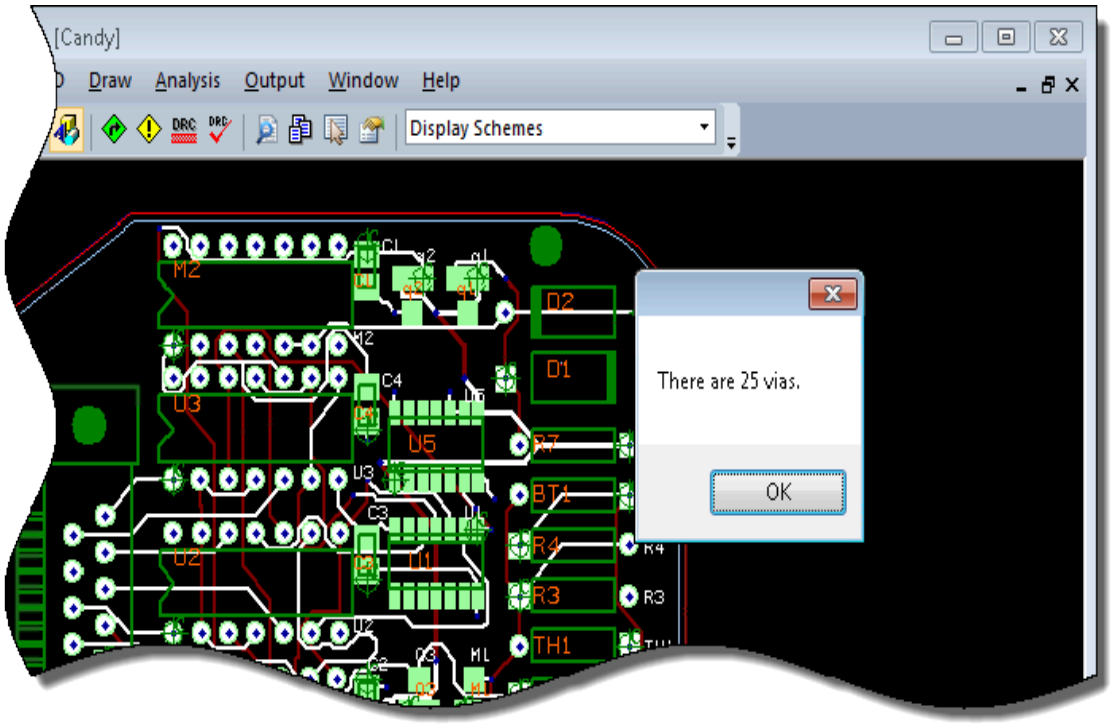
Run the script from within Xpedition Layout.

Note:
"Running Scripts from an Application" on page 161 details this technique.

1. Launch Xpedition Layout.
2. Open a layout design of your choice.
3. Right-click the toolbar and ensure **Keyin Command** is enabled.

4. Enter the string run *C:\temp\ViaCount.vbs* into the **Keyin Command** field, which is located toward the upper right-hand corner of the window (Figure 18: Keyin Command ViaCount.vbs).

Figure 18. Keyin Command ViaCount.vbs



5. Press the Enter key to run your script. A dialog displays and contains the number of vias in your design.

6. Click **OK** to dismiss the dialog. This also ends this script, because there is nothing else to do.

An Alternative Technique

The previous version of the Via Count script declares and accesses the Application and Document objects as follows:

```
' Get the Application object
Dim pcbAppObj
Set pcbAppObj = Application

' Get the active document
Dim pcbDocObj
Set pcbDocObj = pcbAppObj.ActiveDocument
```

An alternative technique is as shown in the following code snippet. This version uses the *Application* object directly:

```
6 ' Get the Application object
7 Dim pcbAppObj
8 Set pcbAppObj = Application
9
10 ' Get the active document
11 Dim pcbDocObj
12 Set pcbDocObj = Application.ActiveDocument
```

This alternative technique is presented only for completeness. Although it may seem more efficient, this technique is not recommended. The script, originally intended to run as an in-process client, is difficult to change to run as an out-of-process client. For example, consider the original version of the script, which assumes that it runs as an in-process client on Xpedition Layout:

```
6  ' Get the Application object
 7  Dim pcbAppObj
 8  Set pcbAppObj = Application
 9
10  ' Get the active document
11  Dim pcbDocObj
12  Set pcbDocObj = pcbAppObj.ActiveDocument
```

Now, assume that you modify the script to run as an out-of-process client. Out-of-process clients generally run from the command line, but they can also run in another application – Xpedition FabLink for example. The implicit Application object is not available to out-of-process scripts. In order to address this, you can easily modify the script as follows:

```
 6  ' Get the Application object
 7  Dim pcbAppObj
 8  Set pcbAppObj = Application
 9  Set pcbAppObj = GetObject(,"MGCPCB.ExpeditionPCBApplication")
10
11  ' Get the active document
12  Dim pcbDocObj
13  Set pcbDocObj = pcbAppObj.ActiveDocument
```

GetObject is a VBScript function. Use this function to attach to a running instantiation of Xpedition Layout by passing the *Prog ID* (this is discussed in “Prog ID” on page 21.) Xpedition Layout’s *Prog ID* is MGCPCB.ExpeditionPCBApplication.

If there are multiple instantiations of the same application open, then GetObject attaches to an arbitrary instantiation. If an instantiation of Xpedition Layout is not already running, use the CreateObject function to launch the application as shown below:

```
 6  ' Get the Application object
 7  Dim pcbAppObj
 8  Set pcbAppObj = Application
 9  Set pcbAppObj = CreateObject("MGCPCB.ExpeditionPCBApplication")
10
11  ' See note below with regard to opening a document
```

After the script launches Xpedition Layout, the script should either open a layout design or prompt the end-user to do so.

Chapter 5
Creating Forms and Dialogs Using the IDE

This section provides several exercises for creating forms and scripts using the Integrated Development Environment (IDE).

Forms (dialogs) are useful in instances where the end-user enters more data than the primitive InputBox() functionality supports. Forms are also useful for displaying information to the end-user in an organized manner, and having that information dynamically update. Scripts are good at pulling things together and automating tasks, but most scripting languages have limited graphical interface support. To address this, several Mentor Graphics products feature an Integrated Development Environment (IDE), allowing users to create forms. The IDE is the preferred mechanism for creating scripts and forms in the Xpedition Layout tool flow.

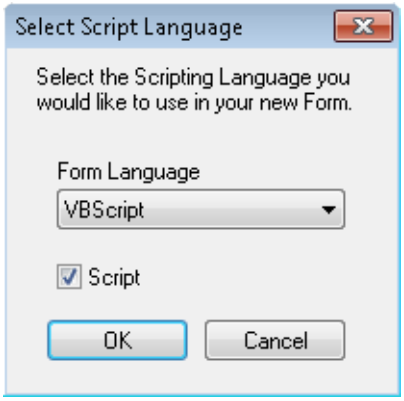
Creating a Script With the IDE
Handling Errors in the IDE
Creating a Dialog With the IDE
Launching the IDE in Form-editing Mode
Adding Buttons to the Form
Adding Code "Behind" the Cancel Button
Adding Code "Behind" the Count All Button
Adding Code "Behind" the Count Selected Button
Removing Duplicate Code
Assigning a Title
Going into Production

Creating a Script With the IDE

In this exercise, you copy the Via Count script into the IDE text editor and then run the script.

1. Launch Xpedition Layout and open a design containing a number of traces and vias. Choose **File > Scripting > New Script Form** to display the **Select Script Language** dialog (Figure 19: Select Script Language Dialog)

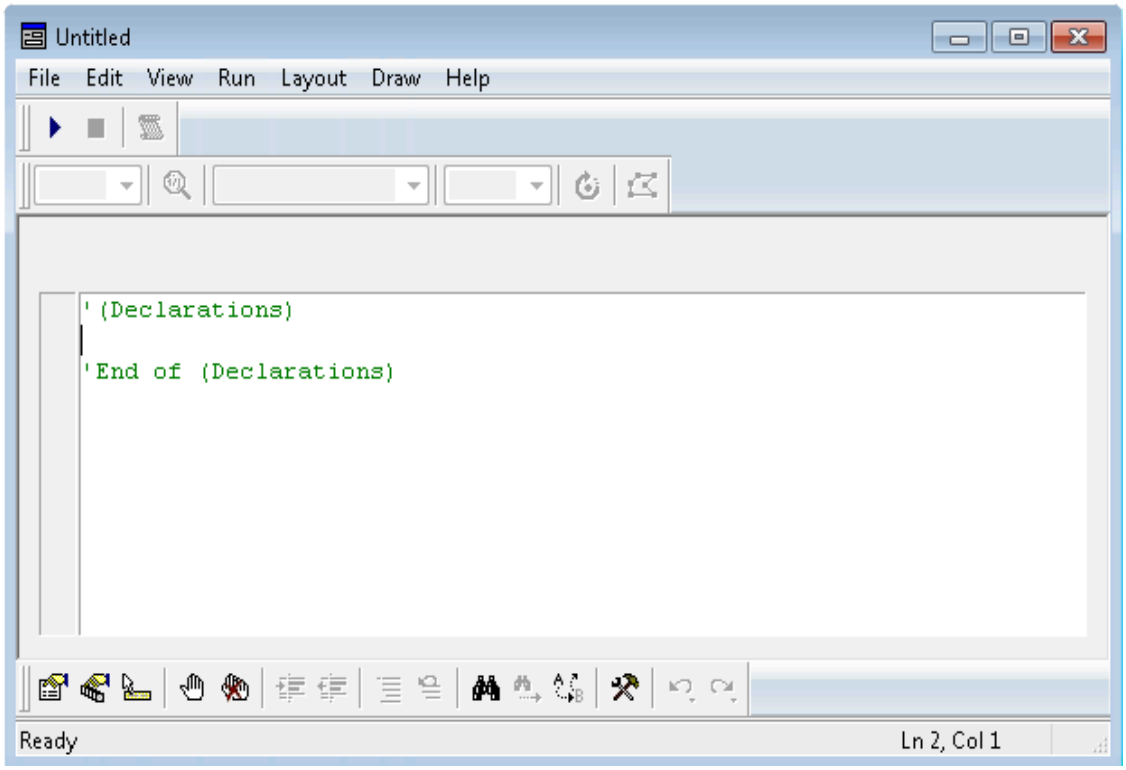
Figure 19. Select Script Language Dialog



2. Use the pull-down list to select VBScript in the Form Language field and ensure that the Script checkbox is enabled, (this indicates that you are creating a script only).

3. Click **OK** to accept these settings. IDE opens in its script-editing mode (Figure 20: IDE Script-editing Mode). Observe the (Declarations) and End of (Declarations) comments in the main editing window. You enter all of your script statements between these two comment lines.

Figure 20. IDE Script-editing Mode



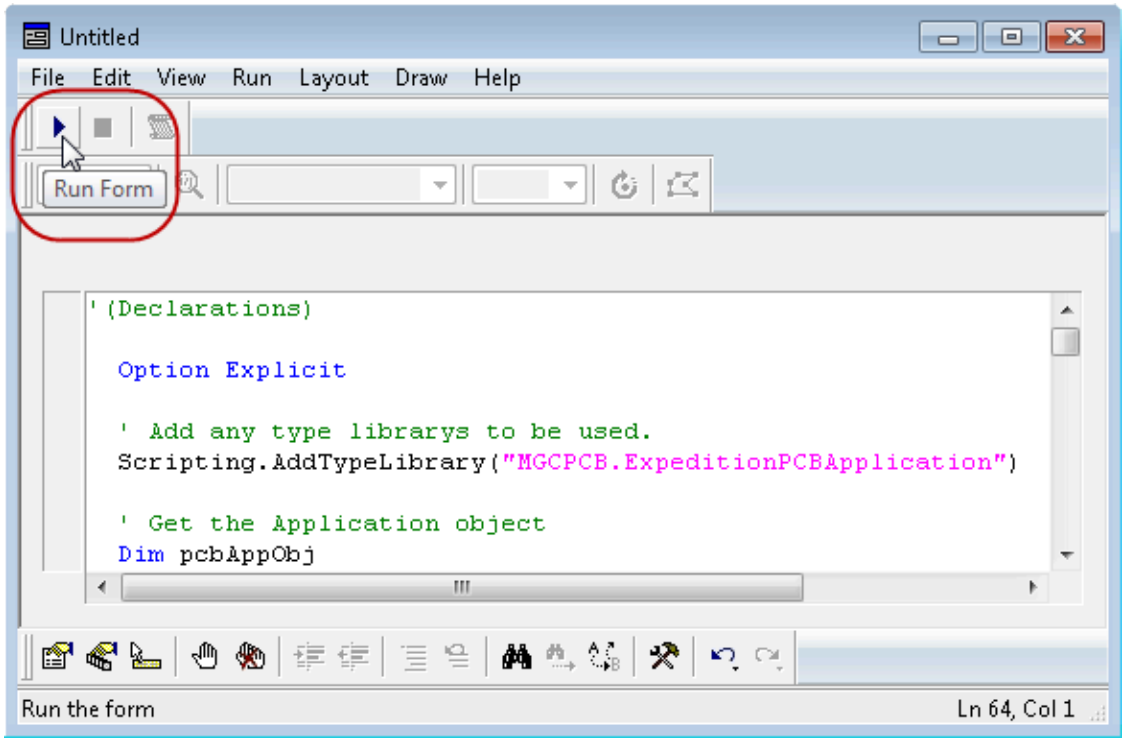
The IDE has syntax coloring and a limited IntelliSense capability (the ability to predict potential conclusions to the statement you are entering and offer you alternatives. Typically, you enter your script directly into the main editing window. For this example, however, use the script you created in the previous chapter.

4. Use the editor of your choice to open the ViaCount.vbs script. Select and copy the entire script.

5. In the IDE, choose **Edit > Paste** to paste the script between the (Declarations) and End of (Declarations) comments as illustrated in Figure 21: IDE Run Form.

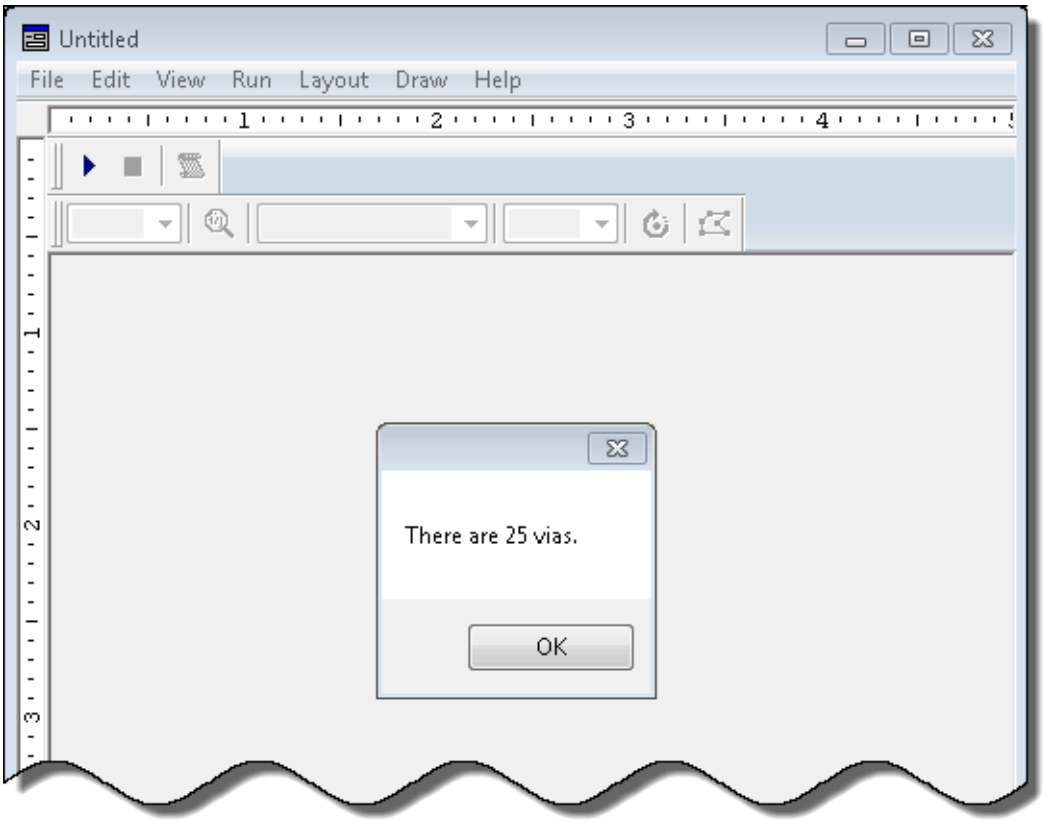
6. Choose **File > Save As** to save the IDE-based version of the script as ViaCount-IDE.vbs in the *C:\temp* folder.

Figure 21. IDE Run Form



7. Run the new script directly from the IDE by clicking the **Run Form** icon (Figure 21: IDE Run Form). The Run icon looks like the "Play" button on a media player and is located in the upper left-hand corner of the IDE. Alternatively, run the script by choosing **Run > Run** or the **F5** function key. The result is shown in Figure 22: Output of ViaCount.vbs . (The actual via count value may be different depending on the design you are using).

Figure 22. Output of ViaCount.vbs



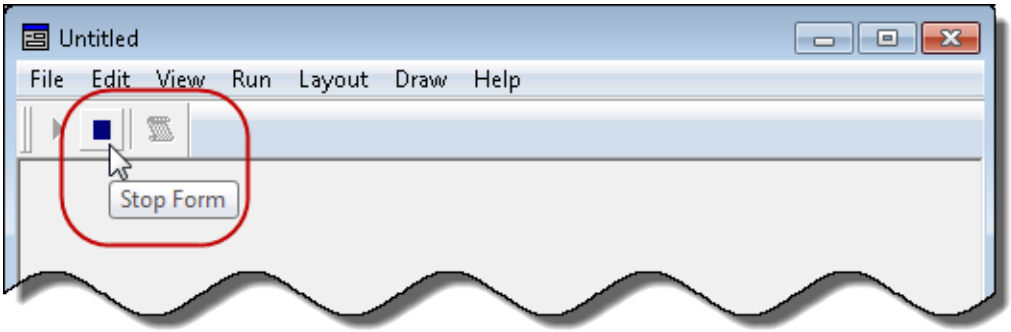
8. The script calls a MsgBox() function to display the total number of vias in the design. The IDE displays an empty form, named Untitled, in which the script runs. The dialog created by the MsgBox() function displays in front of the IDE form. (Figure 22: Output of ViaCount.vbs)

9. Click **OK** to dismiss the dialog.

10. Observe that the IDE form continues running. Click the **Stop** icon (Figure 23: Stopping the Script). The Stop icon looks like the "Stop" button on a media player and that is located in the

upper left-hand corner of the IDE form. This stops the form and returns the IDE to the script-editing mode.

Figure 23. Stopping the Script



Handling Errors in the IDE

An additional feature associated with the IDE is its error-handling capability. In order to see this in action, perform the following steps.

1. Introduce a deliberate error into your script; for example, change viasColl.Count to viasColl.ount on Line 25. (Note the line and column numbers in the lower right corner of the IDE.)

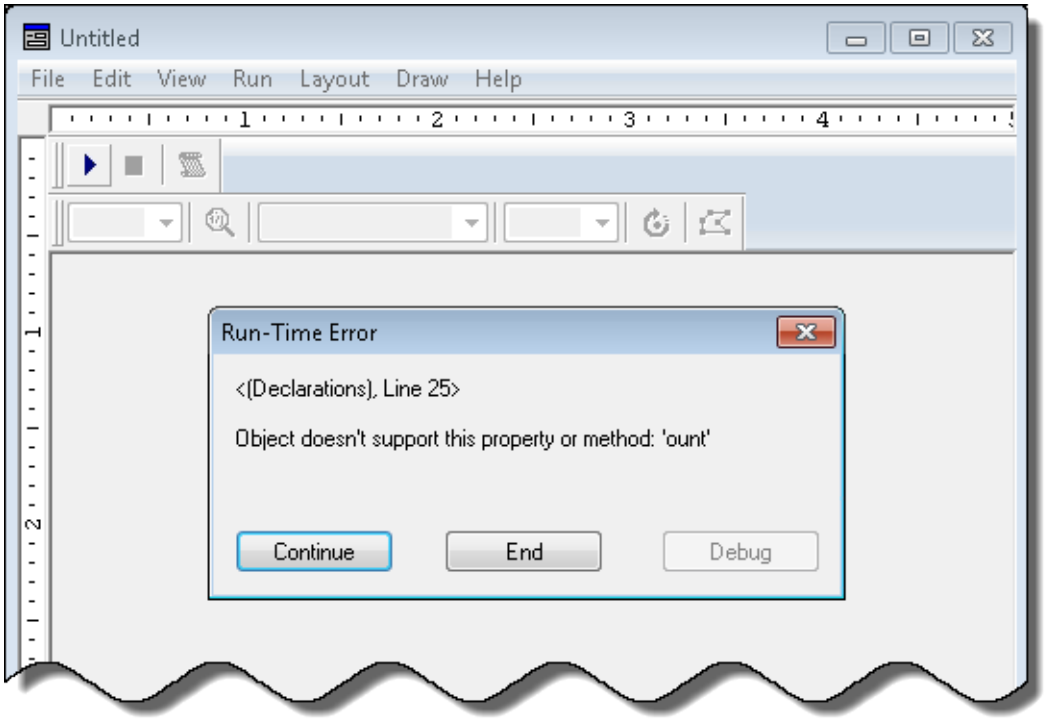
2. Run the new version of the script in the IDE by clicking the **Run Form** icon.

3. Observe the resulting **Run-Time Error** dialog (Figure 24: Run-Time Error Dialog). This indicates the error and presents two options: **Continue** or **End**.

a. Selecting **Continue** ignores this error and attempts to continue running the script.

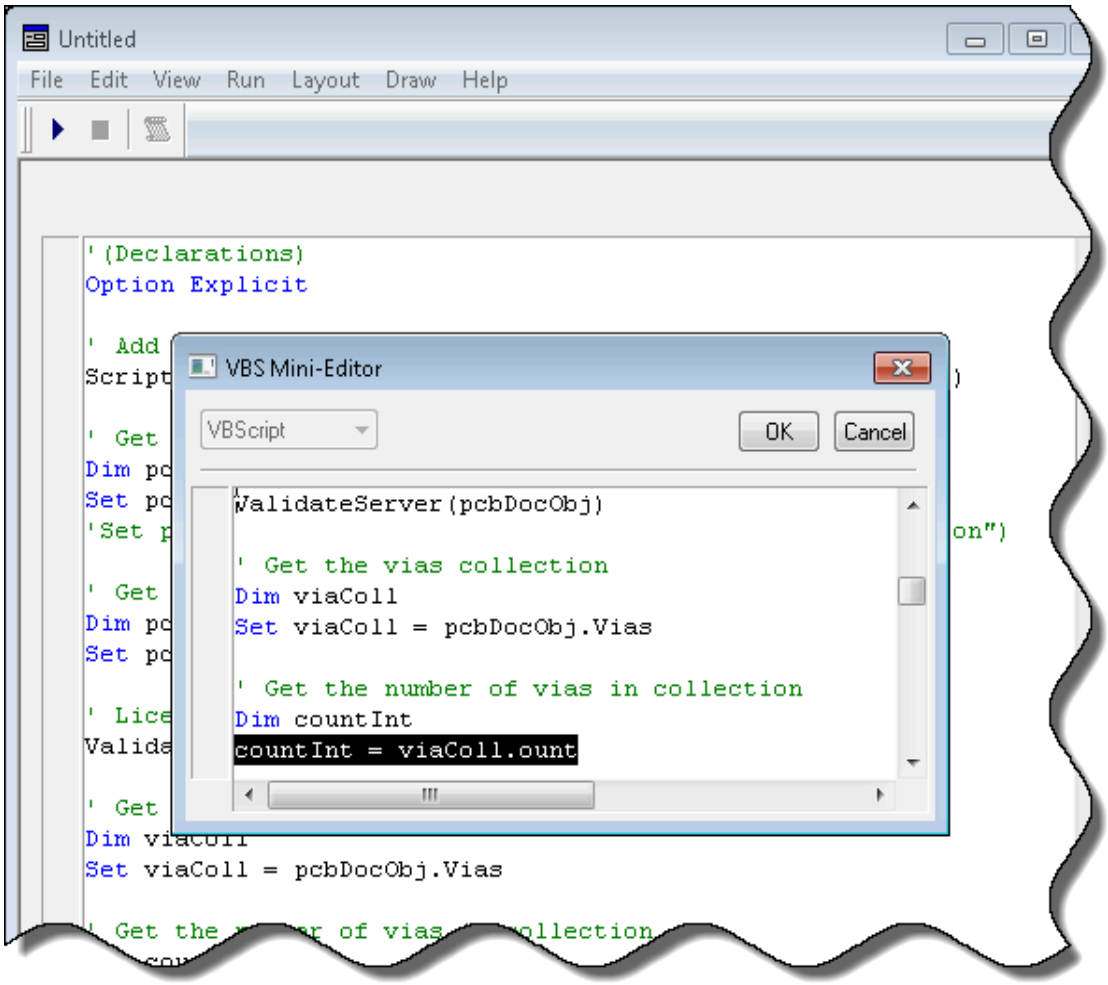
b. Selecting **End** launches the **VBS Mini-Editor** so that you can correct the error.

Figure 24. Run-Time Error Dialog



4. Click **End** to access the **VBS Mini-Editor**. This automatically highlights the statement containing the error as illustrated in Figure 25: VBS Mini-Editor .

Figure 25. VBS Mini-Editor



5. Correct the error (change viaColl.ount back to viaColl.Count) and click **OK**. This closes the VBS Mini-Editor and returns the IDE to script-editing.

Creating a Dialog With the IDE

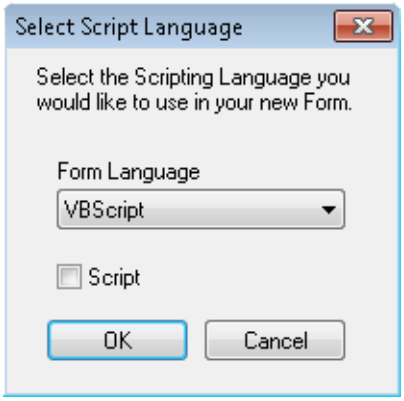
In this exercise, you use the IDE to create a simple form with three buttons named Count All, Count Selected, and Cancel. The Cancel button simply dismisses the form with no changes. The Count All button runs the Count Vias script. The *Count Selection* Button runs a modified version of the Count Vias script. Finally, you perform a few "housekeeping" tasks to make the form production ready.

Launching the IDE in Form-editing Mode

A setting controls whether the IDE launches in script-editing mode or form-editing mode.

1. Launch Xpedition Layout and open a design containing some number of traces and vias. Choose **File > New Script Form** to open the **Select Script Language** dialog (Figure 26: Select Script Language)

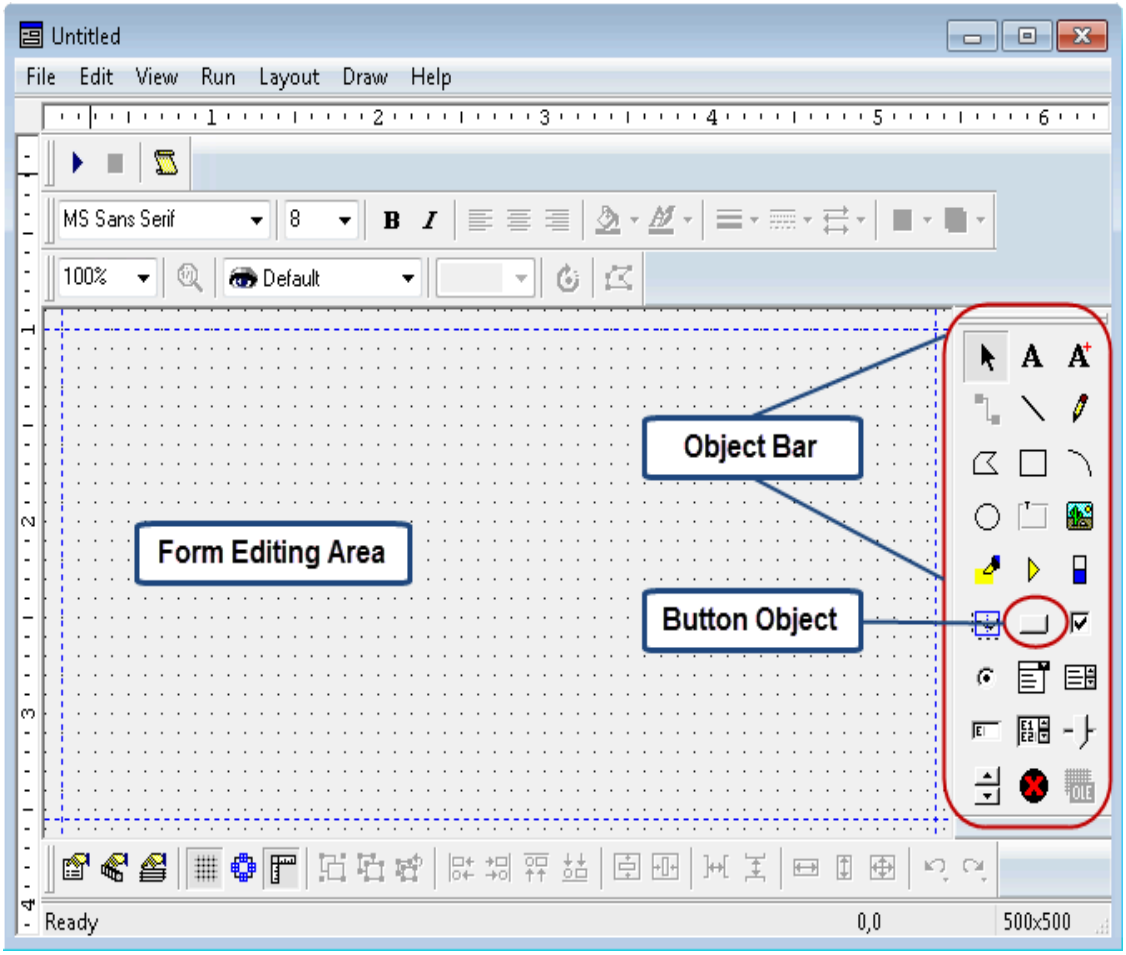
Figure 26. Select Script Language



2. Use the pull-down list to select VBScript in the **Form Language** field and ensure that the **Script** checkbox remains unchecked (this creates a form).

3. Click **OK** to accept these settings. The IDE opens in form-editing mode (Figure 27: IDE Form-editing Mode).

Figure 27. IDE Form-editing Mode

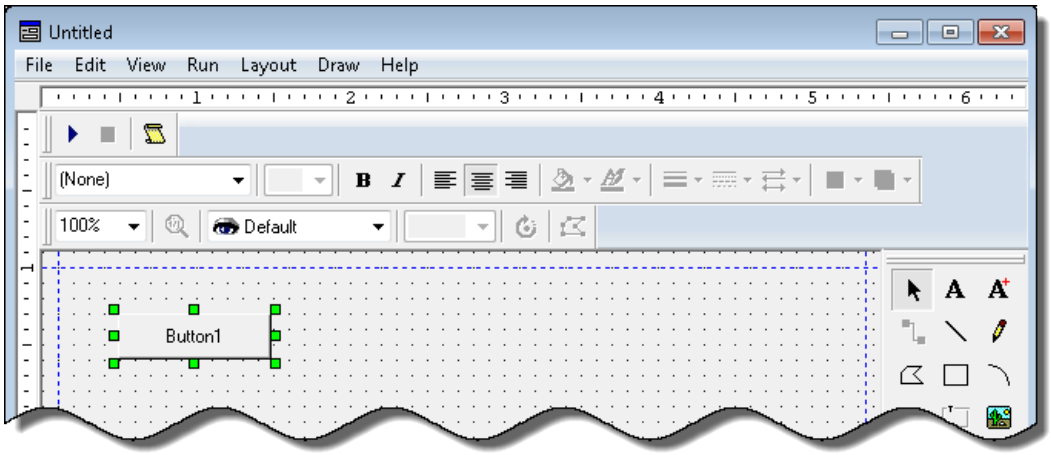


Adding Buttons to the Form

The **Object Bar** presents icons of the various objects you can add to forms. For this example, you add three buttons (Count All, Count Selected, and Cancel).

1. Click the **Button** icon and move the mouse cursor into the main form editing area and click again to place the button (Figure 28: Placing a Button).

Figure 28. Placing a Button

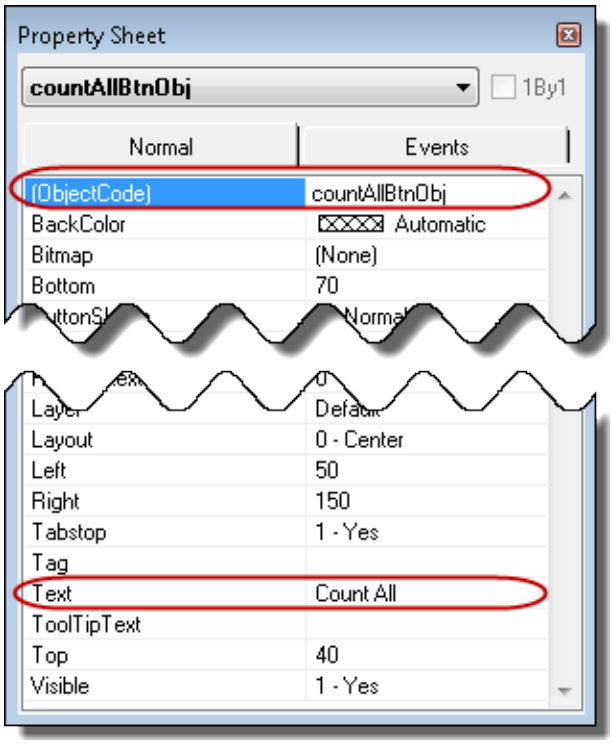


When in the form-editing mode, you can add controls such as buttons, radio buttons, combo boxes, check boxes, and list boxes to the form. You can also add static text, lines, and geometric shapes.

Observe the green "handles" on button (four in the corners and four on the sides). These handles resize the button. In addition, if you hover the mouse cursor over the button, it changes to a four-headed arrow; you can now move the button.

2. Right-click on the button and then choose **Object Properties** from the popup menu. This opens the **Property Sheet** dialog as shown in Figure 29: Property Sheet Dialog .

Figure 29. Property Sheet Dialog



3. Select the **Normal** tab. Observe the **(Object Code)** and **Text** items:

   - The **(Object Code)** property defines the variable the underlying script uses to access the control.

   - The **Text** property names (or annotates) the button the end-user sees on the form.

4. Set **(Object Code)** to countAllBtnObj (for "Count All Button Object"). This name automatically appears in the pull-down list field at the top of the **Property Sheet** dialog.

5. Set **Text** to Count All.

Tip
The following abbreviations are recommended for **(Object Code)** names:

- *BtnObj* = "Button object"
- *RadObj* = "Radio button object"
- *CmbObj* = "Combobox object"
- *ChkObj* = "Checkbox object"
- *LstObj* = "Listbox object"

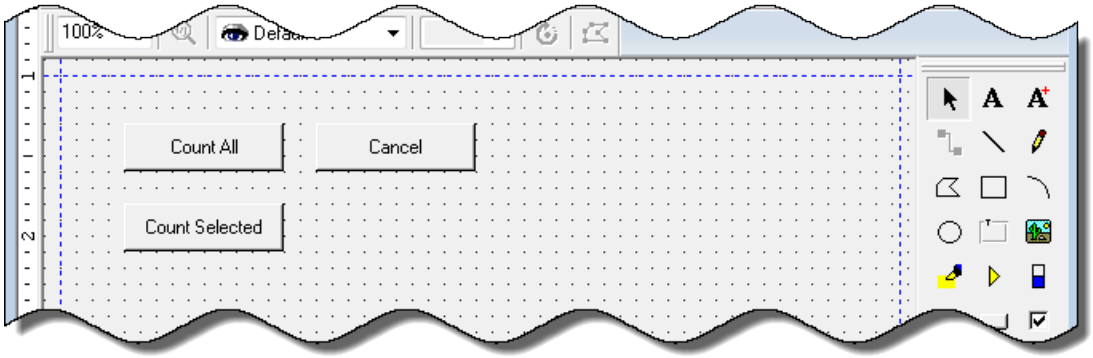
---

6. After changing the Property Sheet, close the dialog by pressing Esc or clicking the small red 'X' (Close) button on the right-hand side of the title bar. This may seem strange; however, you need not formally apply changes because they automatically apply as you enter them.

7. Repeat steps (1) through (6) to create two additional buttons with the following properties (the result is Figure 30: Count All, Count Selected, and Cancel Buttons):

Text	(Object Code)
Count Selected	countSelBtnObj
Cancel	cancelBtnObj

Figure 30. Count All, Count Selected, and Cancel Buttons

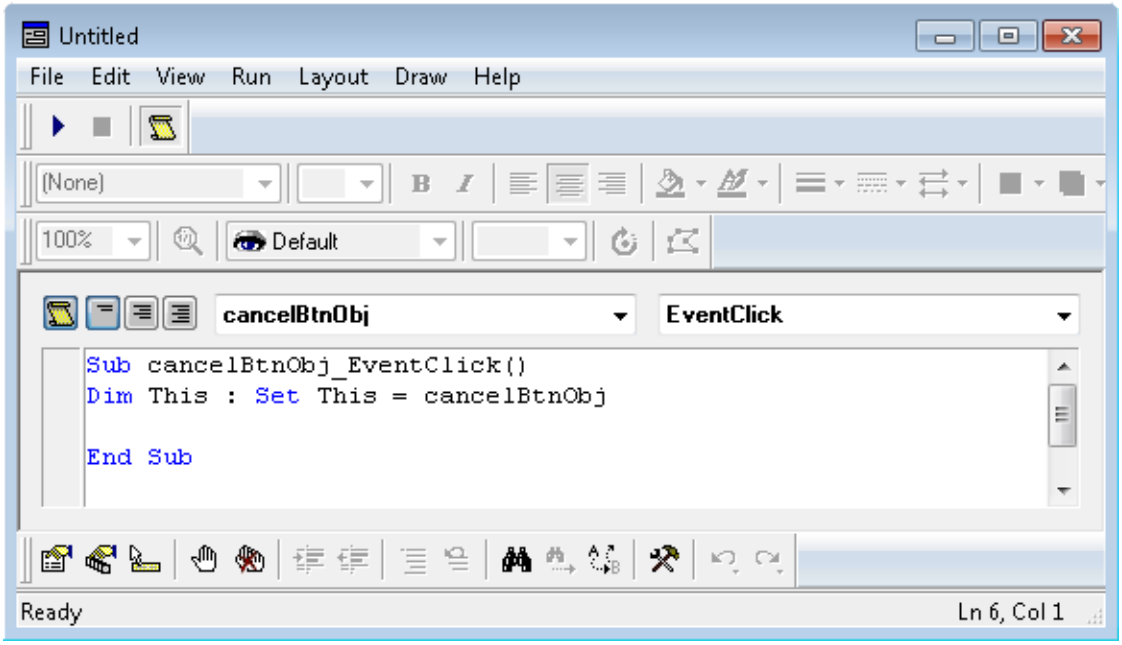


Adding Code "Behind" the Cancel Button

After designing the form, the next task is adding code (scripts) "behind" the buttons. There are multiple ways to do this. In general, the simplest technique is to double-click the button. This automatically opens the event handler’s routine for that button's click event.

1. Double-click the **Cancel** button; the text editor opens as illustrated in Figure 31: Event Handler for Cancel Button .

Figure 31. Event Handler for Cancel Button



Observe the drop-down box above the main edit window to the left-hand side. This lists the *objects* available to this form. By default, it lists the name of the selected button (cancelBtnObj).

Next, observe the drop-down box to the right-hand side. This lists the *event handlers* this object can implement. By default, the **EventClick** event handler is selected. The **EventClick** handles the end-user performing a single-click on the **Cancel** button. There is a variety of event handlers, including the following:

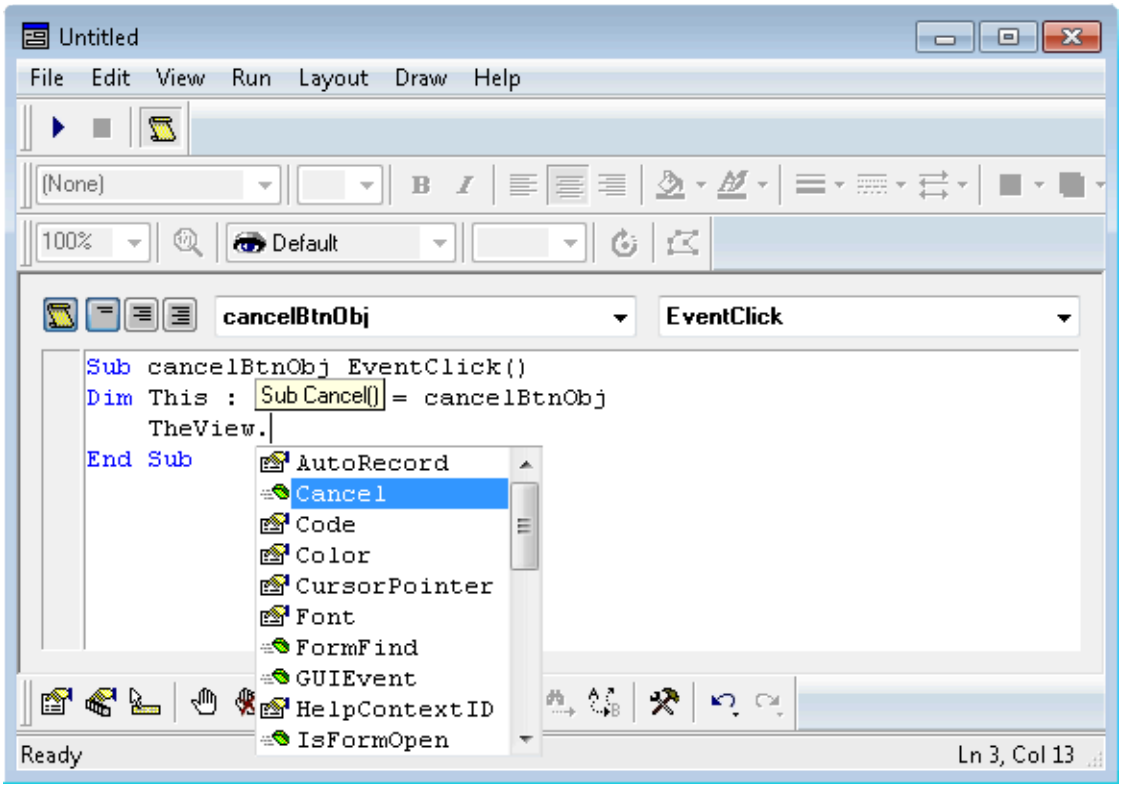
- **EventClick** — Event handler for a single click event
- **EventDblClick** — Event handler for a double-click event
- **EventInitialize** — Event handler for initializing the control (this runs when the form loads and create the control)

2. Place the cursor in the script window on the blank line between the Dim and End Sub statements. Enter the following text:

```
TheView.
```

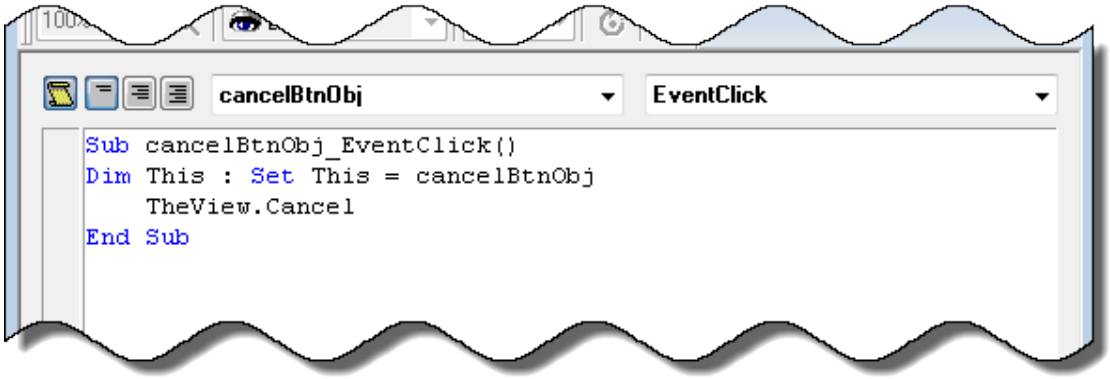
A popup window appears listing all of the properties and methods associated with the form (Figure 32: IDE Statement-completion). The IDE can complete a statement automatically.

Figure 32. IDE Statement-completion



3. Select the **Cancel** item (method) and press Enter to confirm the selection. This automatically completes the statement in the script window (Figure 33: Completed Statement).

Figure 33. Completed Statement

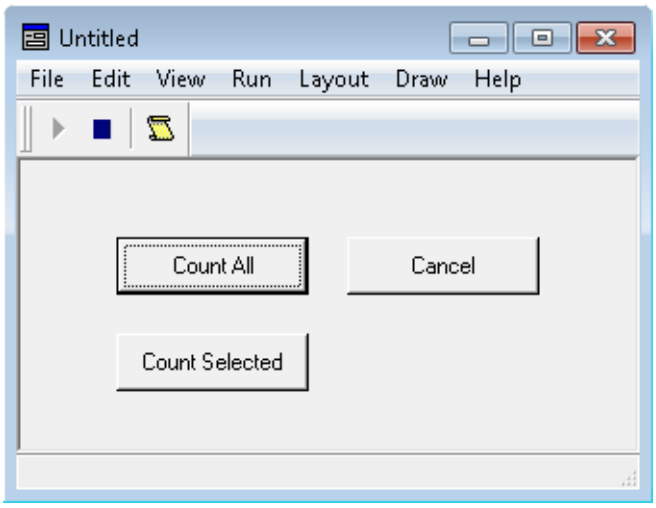


Note:
You can think of TheView as the **(Object Code)** item for the entire form.

4. Choose **File > Save As** and save the form as ViaCountForm.efm. The **.efm* extension denotes a form, the **.vbs* extension denotes a VBScript script.

5. Click **Run Form** to run the form. The result is shown in Figure 34: Running the Form .

Figure 34. Running the Form



The IDE has two editing modes (script-editing and form-editing), but it has only one run mode.

6. Click **Count All** and **Count Selected**. Nothing happens because you have not put the code "behind" these buttons. Click **Cancel** to dismiss the form.

Adding Code "Behind" the Count All Button

The last step in the previous section is to click **Cancel** to dismiss the form. This button performs as expected because the form finishes and closes. (This is why saving the file often, is a good idea).

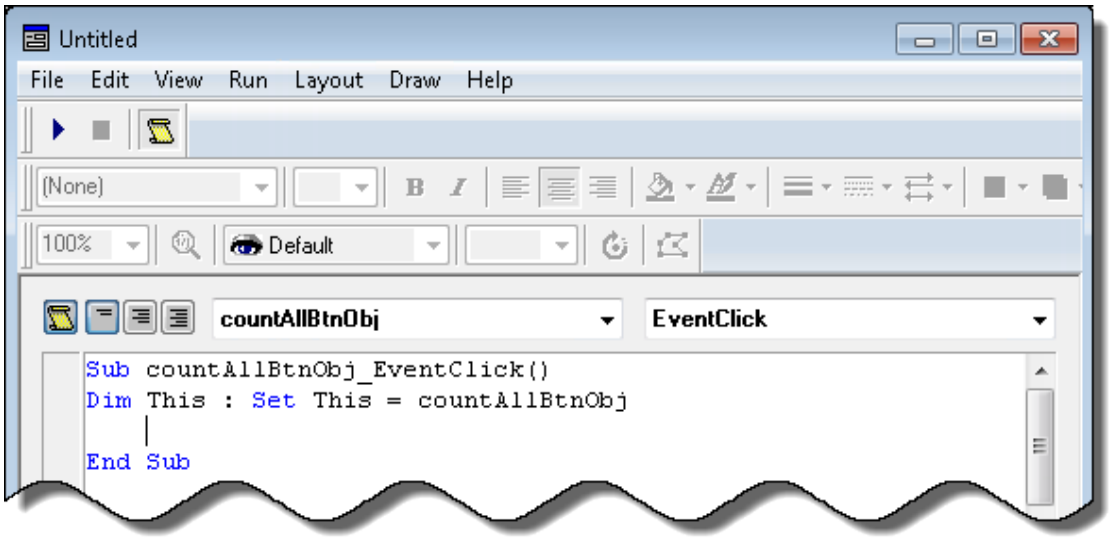
1. Choose **File > Open Script Form** to locate and open the ViaCountForm.efm file.

2. By default, the IDE opens in its run mode. Click the **Stop** icon to return the IDE to its edit mode.

   To open the IDE directly in edit mode, press and hold Ctrl while clicking **Open** on the **File > Open Script Form** dialog (Step 1).

3. Double-click the **Count All** button to open the event handler for **Count All** as illustrated in Figure 35: Event Handler for Count All Button .

Figure 35. Event Handler for Count All Button



4. Use the text or script editor of your choice to open up the ViaCount.vbs file. Select the code starting at the blank line following the Option Explicit statement and ending with the MsgBox() statement as shown below.

```
Option Explicit
```

```
' Add any type libraries to be used.
Scripting.AddTypeLibrary("MGCPCB.ExpeditionPCBApplication")
```

```
' Get the application object
Dim pcbAppObj
Set pcbAppObj = Application
```

```
' Get the active document
Dim pcbDocObj
Set pcbDocObj = pcbAppObj.ActiveDocument
```

```
' License the document
ValidateServer(pcbDocObj)
```

```
' Get the vias collection
Dim viaColl
Set viaColl = pcbDocObj.Vias
```

```
' Get the number of vias in collection
Dim countInt
countInt = viaColl.Count
```

```
MsgBox("There are " & countInt & " vias in the design." )
```

```
''''''
```

```
    ' Ask Expedition’s document for the key
    key = docObj.Validate(0)
```

```
    ' Get license server
    Set licenseServerObj =
    CreateObject("MGCPCBAutomationLicensing.Application")
```

```
    ' Ask the license server for the license token
    licenseTokenInt = licenseServerObj.GetToken(keyInt)
```

```
    ' Release license server
    Set licenseServerObj = nothing
```

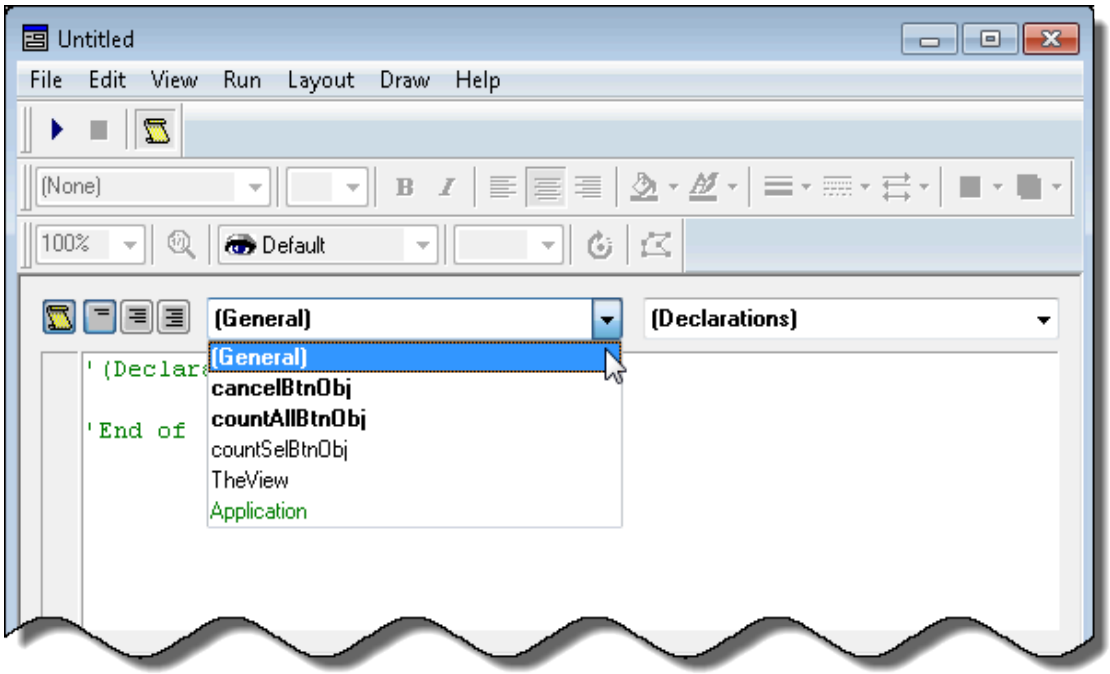
```
    ' Turn off error messages (validate may fail if the token is
    ' incorrect)
    On Error Resume Next
    Err.Clear
```

```
    ' Ask the document to validate the license token
    docObj.Validate(licenseTokenInt)
    If Err Then
        ValidateServer = 0
    Else
        ValidateServer = 1
    End If
```

```
End Function
```

7. In IDE, click the down-arrow icon for the object drop-down box as illustrated in Figure 37: Selecting the (General) Section .

Figure 37. Selecting the (General) Section



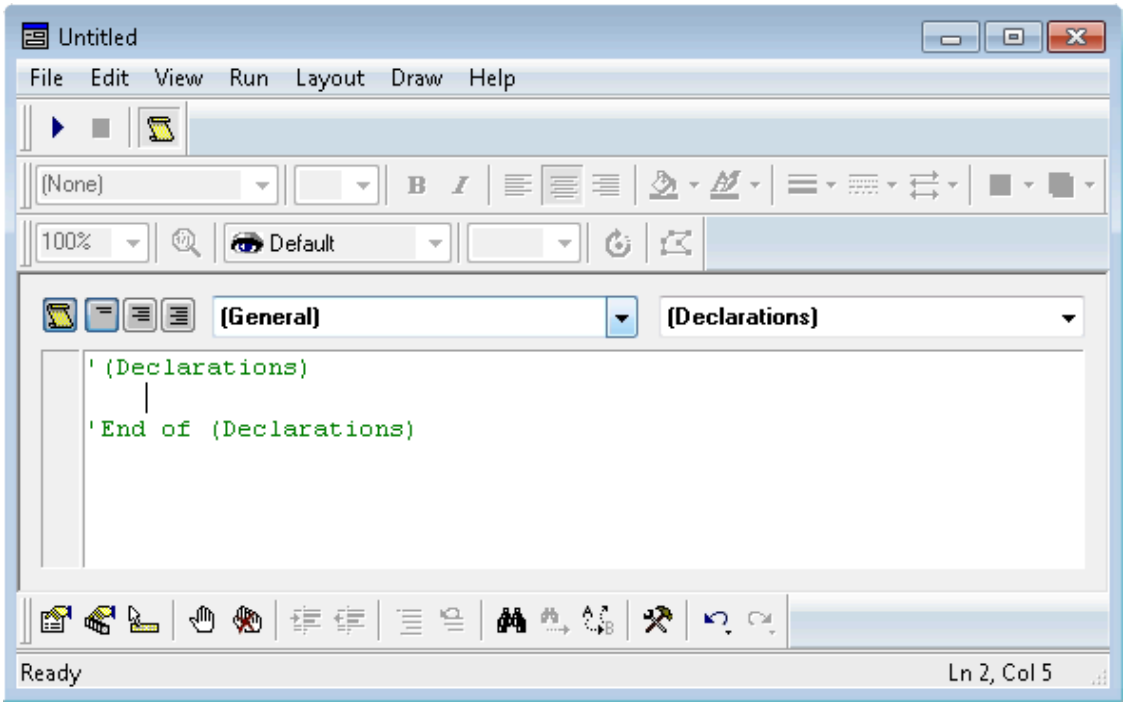
8. Select the **(General)** item. The (Declarations) and End of (Declarations) comments appear in the main editing window as illustrated in Figure 38: (General) Section

   Code that appears between the (Declarations) and End of (Declarations) comments in the **(General)** section is common to the entire script.

   Code in the **(General)** section (except for functions and subroutines) automatically runs when the form opens.

   Functions and subroutines in the **(General)** section run only when they are called. However, defining functions and subroutines here makes them available from anywhere in the form.

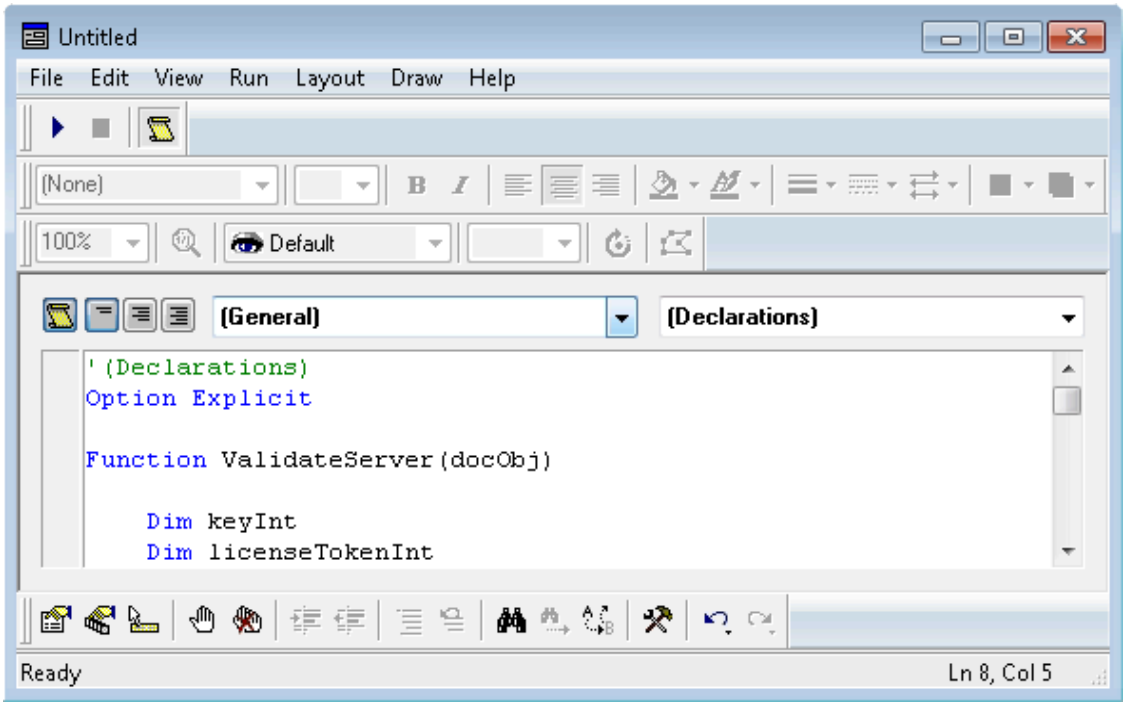
Figure 38. (General) Section



9. Place the cursor between the (Declarations) and End of (Declarations) comments, enter the Option Explicit statement and then paste the server validation code (Figure 39: Server Validation Code in the (General) Section).

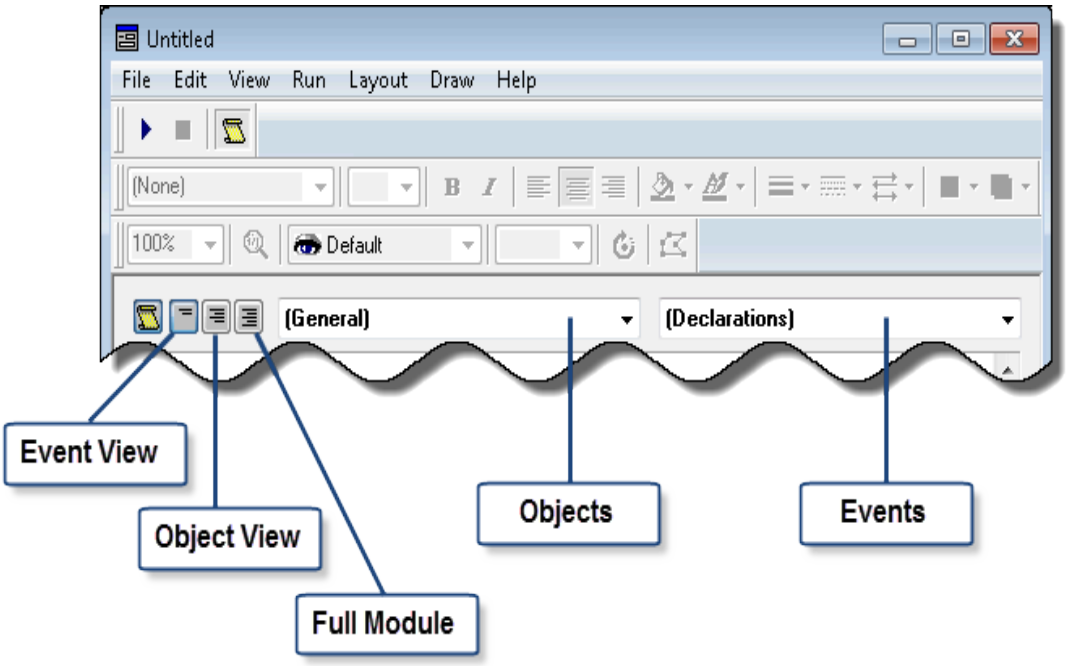
Note:
If you want, you can simply include the server validation code as part of the **Count All** button's single click event handler, but this means that you must include the same validation code in the **Count Selected** button's event handler.

Figure 39. Server Validation Code in the (General) Section



10. Observe the three icons above the editing area on the left-hand side (Figure 40: Event View, Object View, and Full Module Icons). The left-most of these is **Event View**. In this mode, the edit window displays only the code associated with the selected Object and the selected Event. You are working in this mode.

Figure 40. Event View, Object View, and Full Module Icons

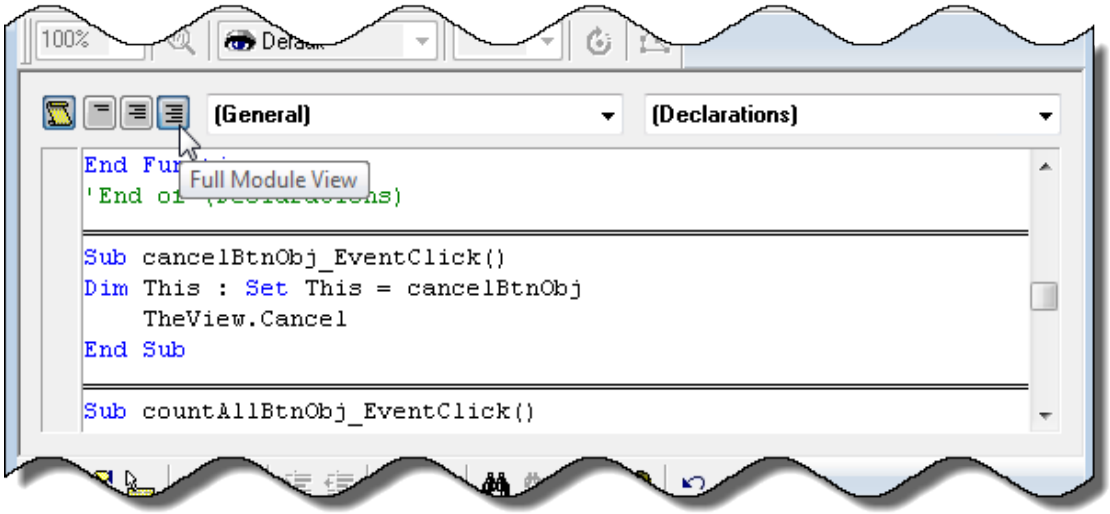


The middle icon is **Object View**. The edit window displays all event handlers for the selected object. For example, if you define code for single-click and double-click event handlers for an object, and if you select the object in the Object drop-down list, then clicking **Object View** displays both event handlers in the edit window.

The right-hand icon is **Full Module View**. The edit window displays all code for this form.

11. Click **Full Module View**; the results are as illustrated in Figure 41: Full Module View .

Figure 41. Full Module View



12. Scroll to the **(General)** section at top of the edit area. As you scroll down, you see the code for the **Cancel** and **Count All** objects. If you click your mouse cursor in the main edit window (or if you use the up/down arrow keys to move the cursor), the Object drop-down list automatically updates to display the object name for the code you are viewing.

Note:
In **Full Module** view, the edit window only displays code for defined event handlers. Therefore, you see only the single-click event handlers for the **Cancel** and **Count All** buttons, and not for the **Count Selected** button.

---

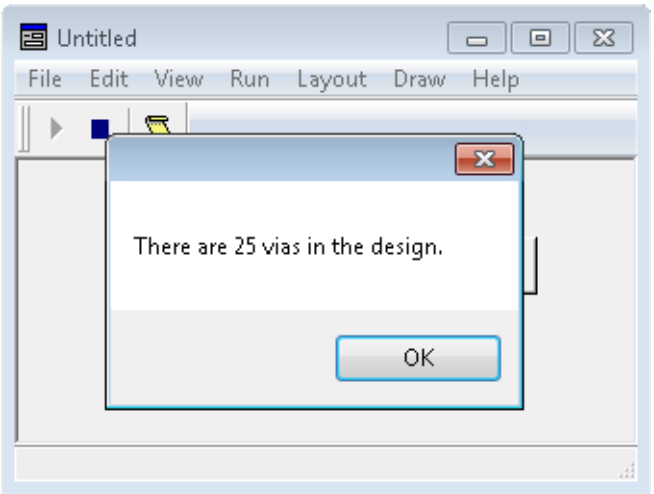
13. Click **Event View** to return the editor to the default mode.

14. Choose **File > Save** to save the current version of your ViaCountForm.efm file, and then click **Run Form**.

15. Click **Count Selected**. Nothing happens because you have not yet added the code "behind" this button.

16. Click **Count All**. The script associated with this button runs and the dialog displays (Figure 42: Count All Button Works).

Figure 42. Count All Button Works



17. Click **OK** to dismiss the dialog.

18. Click **Stop** to return to form-editing mode.

Adding Code "Behind" the Count Selected Button

The simplest technique for adding code "behind" the buttons is to double-click the button. This automatically opens the event handler's routine for that button's click event.

1. Double-click **Count Selected** to access the event handler for the **Count Selected** button. Observe that countSelBtnObj appears in the left-hand drop-down list.

2. Select the countAllBtnObj (this is the **Count All** button) from the Object drop-down.

3. Select and copy the code between the Dim and End Sub statements.

4. Return to the countSelBtnObj using the Object drop-down.

5. Place the cursor between the Dim and End Sub statements and then paste the code into this area.

6. Edit the code as shown.

   a. Add the enumerate epcbSelectSelected as the first parameter to the pcbDocObj.Vias method. This results in the viaColl object containing only the selected vias.

   b. Modify the message associated with the MsgBox() function to reflect the change.

```
' Add any type libraries to be used.
Scripting.AddTypeLibrary("MGCPCB.ExpeditionPCBApplication")

' Get the application object
Dim pcbAppObj
Set pcbAppObj = Application
'Set pcbAppObj = CreateObject("MGCPCB.ExpeditionPCBApplication")

' Get the active document
Dim pcbDocObj
Set pcbDocObj = pcbAppObj.ActiveDocument

' License the document
ValidateServer(pcbDocObj)

' Get the vias collection
Dim viaColl : Ser viaColl = pcbDocObj.Vias
Dim viaColl : Set viaColl = pcbDocObj.Vias(epcbSelectSelected)
Dim countInt : countInt = viaColl.Count

MsgBox("There are " & countInt & " vias in the design.")
MsgBox("There are " & countInt & " selected vias in the design.")
```

7. Choose **File > Save** to save the current version of the *ViaCountForm.efm* file, and then click **Run Form**.

8. Use your mouse to select a subset of the vias in the design. (Assure that you are in route mode).

9. Click **Count All**. The script associated with this button runs and a dialog displays the total number of vias.

10. Click **OK** to dismiss the dialog.

11. Click **Count Selected**. The script associated with this button runs and a dialog displays the number of selected vias.

12. Click **OK** to dismiss the dialog.

13. Use your mouse to select all of the vias in the design.

14. Click **Count Selected** again. Check that the number of selected vias is the same as the total number of vias reported by **Count All**.

15. Click **OK** to dismiss the dialog.

16. Click **Stop** to return to form-editing mode.

Removing Duplicate Code

In the previous task, you populate the code "behind" the **Count All** and **Count Selected** buttons in a simple manner. However, this results in duplicate code. This is not recommended, because it increases the effort to maintain the code. Generally, you should declare common code a single time in the **(General)** section. This task shows you how to remove the duplicate code.

Note:
Event handlers can access variables declared in the **(General)** section; similarly, event handlers can call functions and subroutines defined in the **(General)** section.

Tip
Type libraries in the **(General)** section, not in a function, as they apply to the entire form.

1. Click **Full Module** to display all form code.

2. Scroll through the code and compare the single-click event handlers for the **Count All** and **Count Selected** buttons. All code is common to both event handlers until after the validate server call. The **Count All** event handler as shown below:

```
' Add any type libraries to be used.
Scripting.AddTypeLibrary("MGCPCB.ExpeditionPCBApplication")
```

```
' Get the application object
Dim pcbApp
Set pcbApp = Application
```

```
' Get the active document
Dim pcbDoc
Set pcbDoc = pcbApp.ActiveDocument
```

```
' License the document
ValidateServer(pcbDoc)
```

```
' Code above this point is common to Count All and Count Selected
```

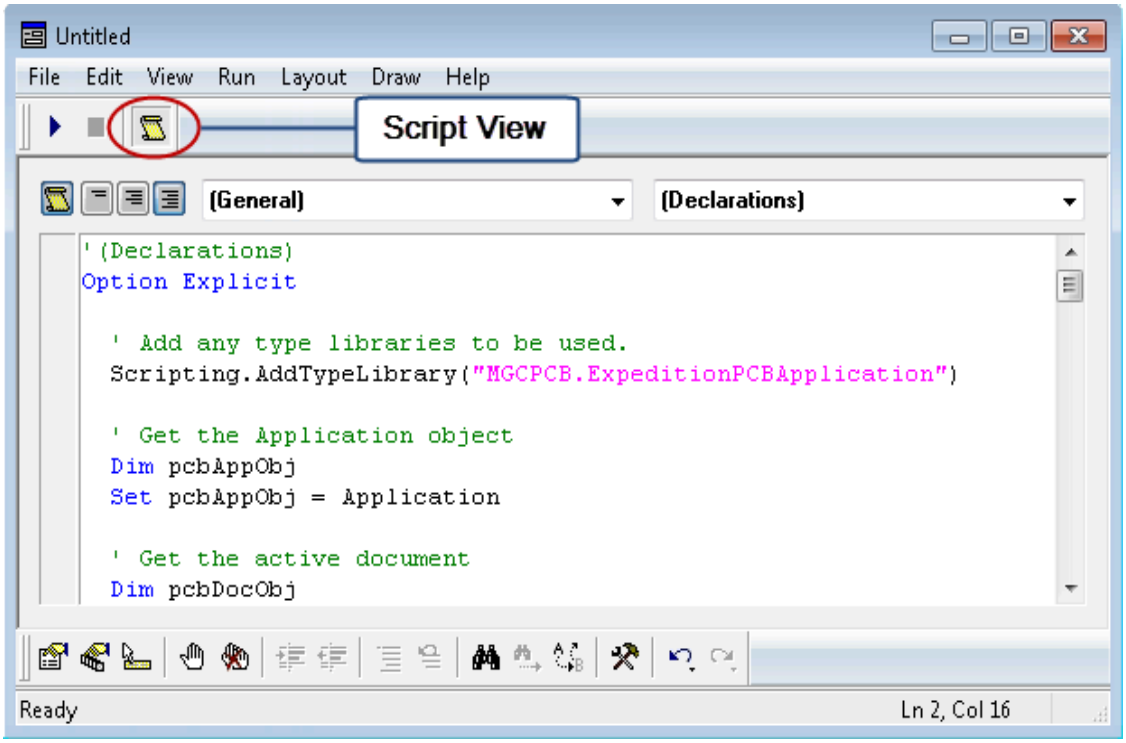
```
Dim viasColl : Set viasColl = pcbDoc.Vias
Dim viaCntInt :  viaCntInt = viasColl.Count
```

```
MsgBox("There are " & viaCntInt & " vias in the design.")
```

3. Select and cut (**Edit > Cut**) the common code from the **Count All** event handler.

4. Scroll to the **(General)** section, place your mouse cursor on the blank line below the Option Explicit statement, and paste (**Edit > Paste**) the common code as illustrated in Figure 43: Common Code in (General) Section .

Figure 43. Common Code in (General) Section



5. Scroll to the **Count Selected** code; select the same (common) code, and remove (**Edit > Cut** this code.

6. Save (**File > Save**) the *ViaCountForm.efm* form. Click **Run Form** and check that the **Count All** and **Count Selected** buttons still function as you expect. Do not click **Cancel**, because there are some additional edits to make.

The scripts assume that a PCB Layout design is already open. What happens if you attempt to run the script without having a design open? One solution is to add code that checks if a design is open, and if not, prompts the end-user to open a design.

Assigning a Title

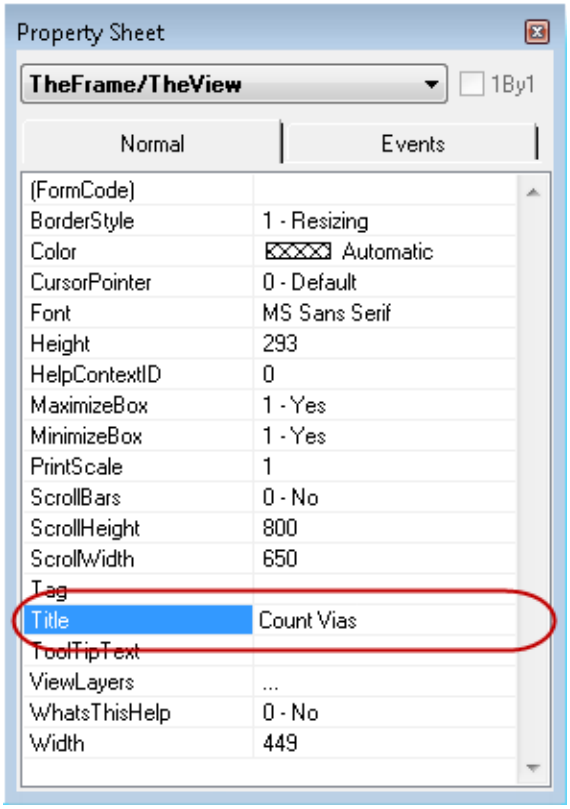
You can assign a title in the form's property sheet.

1. Click **Stop** to return the IDE to form-editing mode (use the **View Script** icon).

2. Right-click anywhere in the view, though not on a button.

3. Select **Form Properties** from the popup menu. The**Property Sheet** dialog appears. The dialog is associated with **TheFrame/TheView** (Figure 44: Setting Form Title)

Figure 44. Setting Form Title

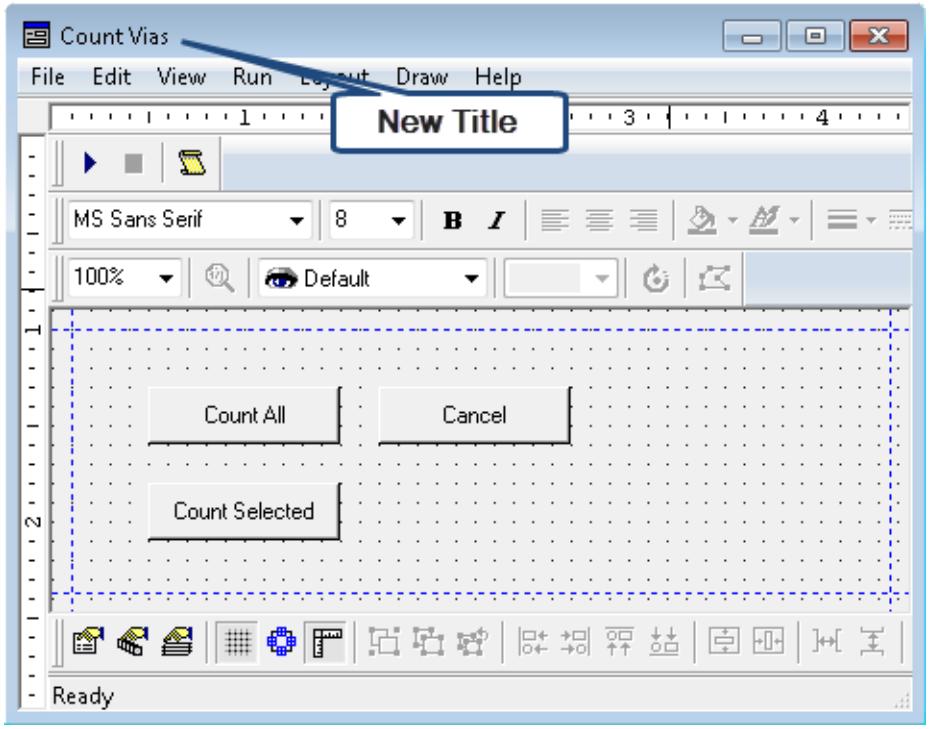


4. Change the **Title** property to “Count Vias”.

5. Press the Esc key or close the dialog. The changes automatically apply when you enter them.

6. Observe that the title bar automatically updates to reflect this new value as illustrated in Figure 45: Form's New Title . This title also displays when the form is running.

Figure 45. Form's New Title

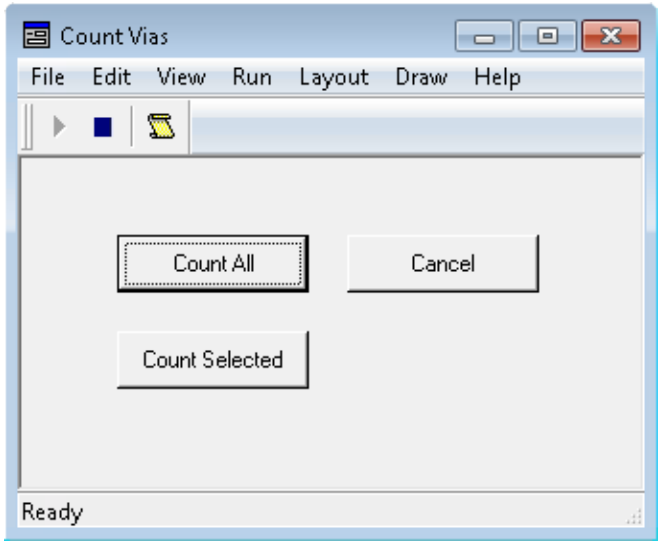


Going into Production

When the IDE is in run mode, the form displays the **Run Form**, **Stop Form** and **View Script** icons and the menus. When developing the form you need these functions. However, in production you do not want end-users to access and modify the form.

1. Click **Run Form** icon to enter IDE's run mode as illustrated in Figure 46: End-User Will See IDE Icons and Menus .

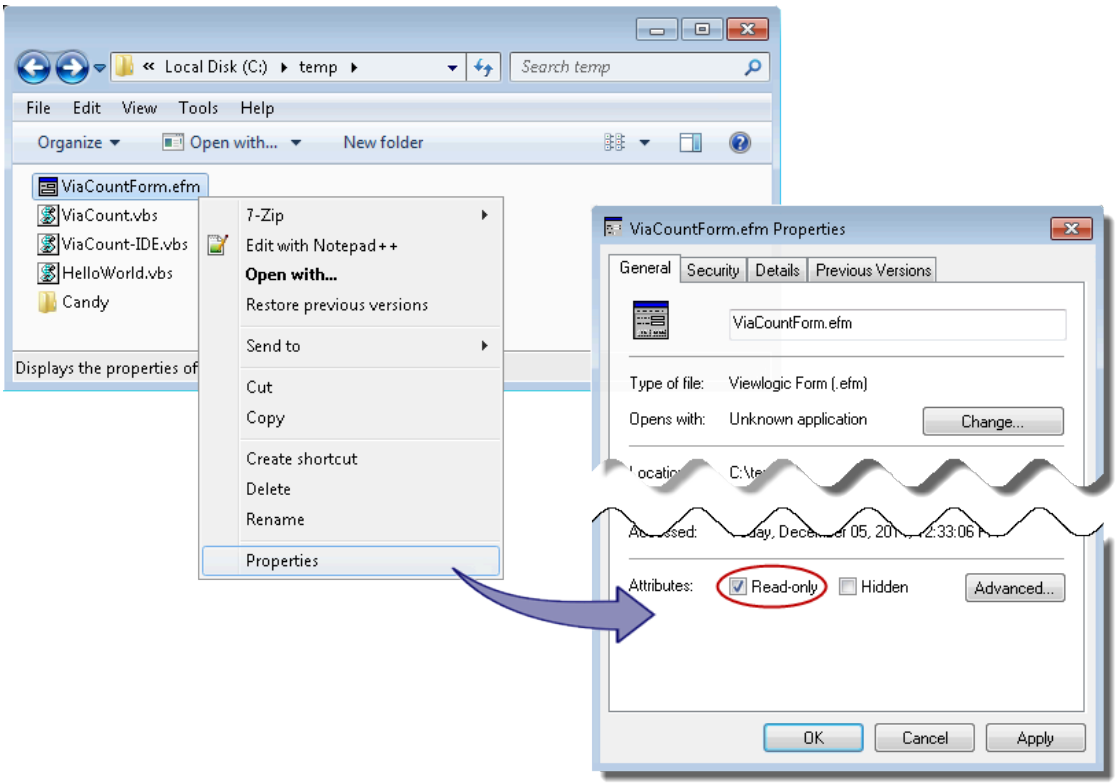
Figure 46. End-User Will See IDE Icons and Menus



2. Click **Cancel** to dismiss the form.

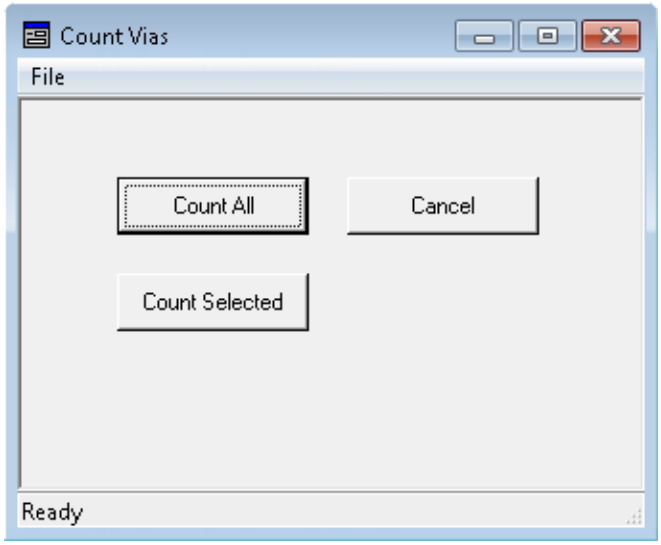
3. Use your computer's file explorer to locate the ViaCountForm.efm file (Figure 47: Setting the Form to Read-Only), right-click on the file, select the **Properties** item from the resulting popup menu, and set the file attribute to **Read-only**. Click **OK**

Figure 47. Setting the Form to Read-Only



4. In Xpedition Layout, choose **File > Open Script Form** to locate and open *ViaCountForm.efm*. The form now appears without the IDE icons and menus (Figure 48: Finished Form).

Figure 48. Finished Form



Chapter 6
Adding Accelerator Keys

This section describes how to associate a shortcut (accelerator) key with a script using the Key Bind Server.

The exercise is “Adding an Accelerator Key.”

The Key Bind Server is a separate server used by applications, such as Xpedition Layout and Xpedition FabLink, to define and run accelerator keys. For example, you can define an accelerator that runs a script when the end-user presses <Ctrl>+N (pressing the N key while pressing and holding the <Ctrl> key).

Adding an Accelerator Key

Adding an Accelerator Key

An accelerator (shortcut) key can be bound to the following objects:

- **A menu item** — Pressing the accelerator key launches a script that invokes and runs an existing menu’s command; this scenario is not discussed further in this chapter.

- **Any valid key-in command** — Pressing the accelerator key launches a script that runs any valid key-in command; that is, any command that you can enter in the key-in command field (this includes running other scripts).

The latter scenario is the focus of this exercise. For example, in “Running Your First Script” on page 59, you create a *ViaCount.vbs* script that is stored in a folder named *C:\temp*. You run this script by entering the following command into the **Keyin Command** field:

```
run C:\temp\ViaCount.vbs
```

If you intend to run this script often, it is easier to associate it with an accelerator key such as Ctrl+N.

Creating the Script

Use the text editor of your choice to enter the following script and save it as *ViaCountAcc.vbs* in your *C:\temp* folder (note that the "Acc" portion of this filename indicates a script associated with an accelerator key).

Line 1 declares Option Explicit (this statement is discussed in “VBScript Primer” on page 27 and “Option Explicit Statement” on page 522.

```
1  Option Explicit
```

Line 4 uses the AddTypeLibrary property of the Scripting object to access the type library associated with the Key Bind Server (the Scripting object is discussed in more detail in “Introducing Events and the Scripting Object” on page 179). Add this type library to your script anytime the script accesses the Key Bind Server.

```
3  ' Add the key bindings type library so we can use its enumerates
4  Scripting.AddTypeLibrary("MGCSDD.KeyBindings")
```

Lines 7 and 8 return the Application (this is discussed in “Running Your First Script” on page 59). This script does not return a Document and does not perform validation. This is because this script does not access or modify any data within the design.

```
6  ' Get the application object.
7  Dim pcbAppObj
8  Set pcbAppObj = Application
```

Lines 11 and 12 return the application’s Gui object. The Gui object is not stored in a variable (which you could do). Instead, it simply returns the Bindings object (also known as the BindingsTables object).

```
10  ' Get doc binding table (Key bindings used when design is open)
11  Dim docKeyBindTableObj
12  Set docKeyBindTableObj = pcbAppObj.Gui.Bindings("Document")
```

The string parameter to the *Bindings* property specifies the binding table to return. The options are as follows:

…Bindings("Application"	Only if no Document is open.
…Bindings("Document"	Only if a Document is open.
…Bindings("Accelerators"	Suitable for either case.

Finally, Lines 15 and 16 use the AddKeyBinding method of the Bindings object to add a new accelerator key to the document binding table. Observe the underscore ('_') character at the end of Line 15; this informs the script engine that this statement continues on the following line.

```
13
14  ' Add the new key binding.
15  Call docKeyBindTableObj.AddKeyBinding("Ctrl+N", _
16                "run C:\temp\ViaCount.vbs", BindCommand, BindAccelerator)
```

There are four parameters associated with the AddKeyBinding method:

1. **KeySequence —** This is the accelerator key (or key combination) to use. In this example, the script assigns the string "Ctrl+N".

2. **Command —** This is the command to run. It can be any valid menu command or key-in command. In this example, the script uses the same key-in command used in “Running Your First Script” on page 59 to run the *ViaCount.vbs* script.

3. **BindType —** The BindCommand enumerate indicates a key-in command rather than a menu command (refer to “Adding Menus and Menu Items” on page 107).

4. **KeyType —** The BindAccelerator enumerate is a required attribute that currently has no impact.

Running the Script

Run the script from within Xpedition Layout (this technique is detailed in "Running Scripts from an Application" on page 161).

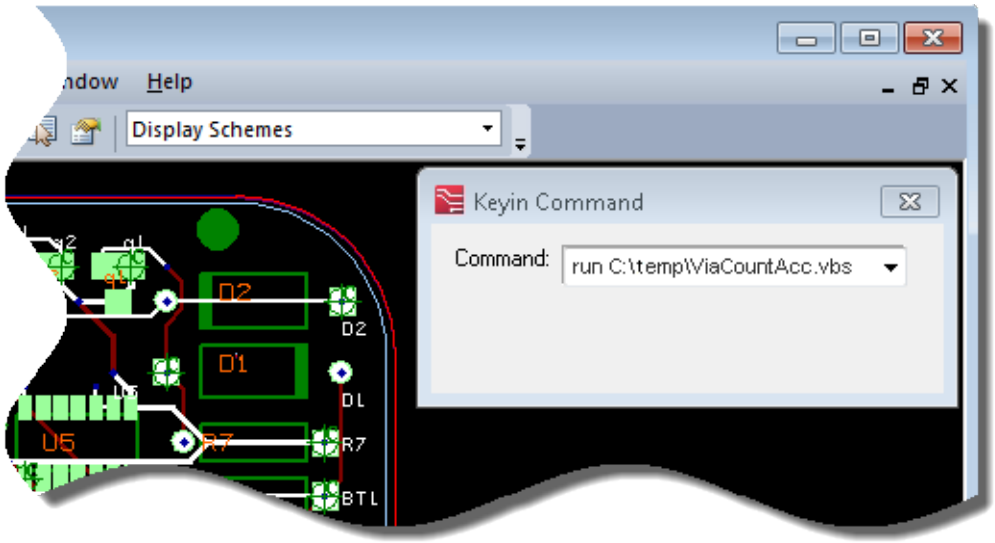
1. Launch Xpedition Layout.

2. You may optionally open a layout design of your choice. However, this is not required to proceed.

3. Right-click on the toolbar and ensure that the **Keyin Command** option is enabled.

4. Enter the string run *C:\temp\ViaCountAcc.vbs* into the **Keyin Command** field (Figure 49: Running the ViaCountAcc.vbs Script) and press Enter to run this script.

Figure 49. Running the ViaCountAcc.vbs Script



After running this script, the script exits and nothing appears to have happened. In reality, the script establishes a binding for the Ctrl+N key combination.

Test the accelerator key to ensure it runs the *ViaCount.vbs* script.

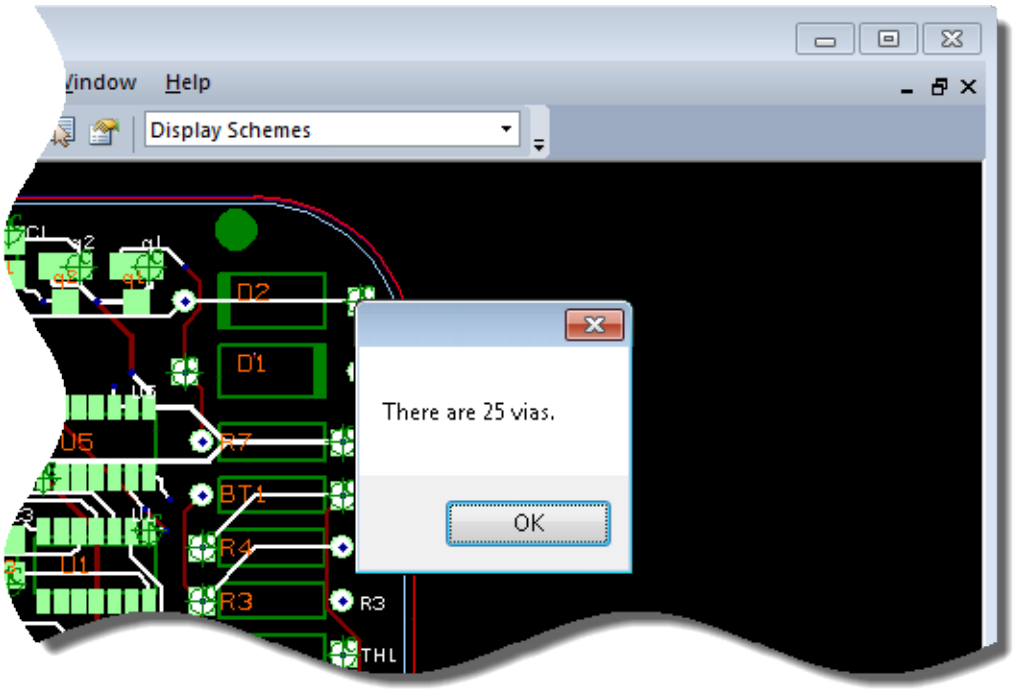
5. Open a layout design if you have not already done so.

6. Press the Ctrl+N key combination and observe that the accelerator key does indeed work as illustrated in Figure 50: Using <Ctrl>+N Accelerator to Count Vias .

   The Ctrl+N binding remains active for the duration of this session.

Note:
You can establish several key accelerator (shortcut) bindings in a single script. This type of script typically runs at startup (refer to "Running Startup Scripts" on page 171).

Figure 50. Using <Ctrl>+N Accelerator to Count Vias



Chapter 7
Adding Menus and Menu Items

This section describes how to add menus and menu items to an application's menu set using the Command Bar Server.

The Command Bar Server is a separate server that applications, such as Xpedition Layout and Xpedition FabLink, use to create or modify pulldown menus on the menu bar. For example, a script can add an item to an existing pulldown menu and then associate a second script with the added menu item.

A script can access the Command Bar Server to create and modify menus. This chapter presents examples of customizing menus.

Note:
Xpedition Layout and Xpedition Designer each use different command bar servers. Scripts must call the correct command bar server for the respective product. Use MGCSDD.CommandBarsEx for Xpedition Layout menus; use Viewlogic.CommandBarsEx for Xpedition Designer menus.

---

1. "Adding a Menu Item With Two Scripts" describes *ViaCountMenuDual.vbs*, which adds a new menu item to an existing pulldown menu and then binds this item to a small script that calls the ViaCount.vbs script.

2. "Adding a Menu Item With One Script" describes *ViaCountMenuSingle.vbs*, which modifies the script associated with a new item such that the script performs the via counting.

3. "Running the Script" describes *ViaCountMenuTop.vbs*, which creates a new pulldown menu.

Note:
The Command Bar Server also adds buttons on toolbars as discussed in "Adding Toolbar Buttons" on page 127.

Adding a Menu Item With Two Scripts
Adding a Menu Item With One Script
ViaCountMenuSingle.vbs
Adding a New Top-Level Menu
ViaCountMenuTop.vbs

Adding a Menu Item With Two Scripts

In this example, you create a script, *ViaCountMenuDual.vbs*, which adds a new item to the bottom of the **View** pulldown menu.

The script then ties this item to the *ViaCount.vbs* script created in "Running Your First Script" on page 59.

Creating the Script

Use the text editor of your choice to enter the following script and save it as *ViaCountMenuDual.vbs* in your *C:\temp* folder. Note that the “Menu” portion of this filename indicates the script is associated with a menu; the “Dual” portion of the filename only distinguishes this script from the next example.

Line 1 declares Option Explicit (this statement is discussed in “VBScript Primer” on page 27 and “Option Explicit Statement” on page 522).

```
1  Option Explicit
```

Line 4 uses the AddTypeLibrary property of the Scripting object to access the type library associated with the Command Bar Server (the Scripting object is discussed in more detail in “Introducing Events and the Scripting Object” on page 179). Add this type library to your script anytime a script accesses the Command Bar Server.

```
3  ' Add the command bar type library so we can use its enumerates
4  Scripting.AddTypeLibrary("MGCSDD.CommandBarsEx")
```

Lines 7 and 8 return the Application (this is discussed in “Running Your First Script” on page 59). This script does not return a Document and does not perform validation. This is because this script does not access or modify any data within the design.

```
6  ' Get the application object
7  Dim pcbApp
8  Set pcbApp = Application
```

Lines 11 and 12 return the application’s Gui object. This script example does not store the Gui object in a variable (which you could do). Instead, it simply returns the CommandBar object.

```
10  ' Get the document menu bar.
11  Dim docMenuBarObj
12  Set docMenuBarObj = pcbApp.Gui.CommandBars("Document Menu Bar")
```

The string parameter to the *CommandBars* property specifies a menu bar or a toolbar. The string “Document Menu Bar” indicates the menu bar and returns the appropriate CommandBar object.

To manipulate (and add items to) the menu bar object, the script must return the list of controls that contain the drop-down menus (each drop-down menu is a single control). To do this, Line 16 instantiates a collection named docMenuBarCtrlColl and Line 17 returns the list of menu bar controls associated with the menu bar.

```
14  ' Get the collection of controls for the menu
15  '(i.e. menu popup buttons, File, Edit, View, etc...)
16  Dim docMenuBarCtrlColl
17  Set docMenuBarCtrlColl = docMenuBarObj.Controls
```

Next, locate the **View** drop-down menu in the controls collection. Line 20 instantiates a variable named viewMenuObj. Line 21 uses the Item property on the collection of controls to access a specific control.

```
19  ' Find the View menu control
20  Dim viewMenuObj
21  Set viewMenuObj = docMenuBarCtrlColl.Item("&View")
```

The Item property accepts an index in the form of an integer or a string. This example uses the string “&View” as the parameter to locate the **View** drop-down menu. The '&' character is required before any character that appears underlined in the menu bar as illustrated in Figure 51: View Menu and Figure 52: Contents of the View Pulldown Menu . Underlined characters are shortcuts to access menus. For example, under the Windows operating system, you use “Alt+V” in Xpedition Layout to access the **View** menu.

Figure 51. View Menu

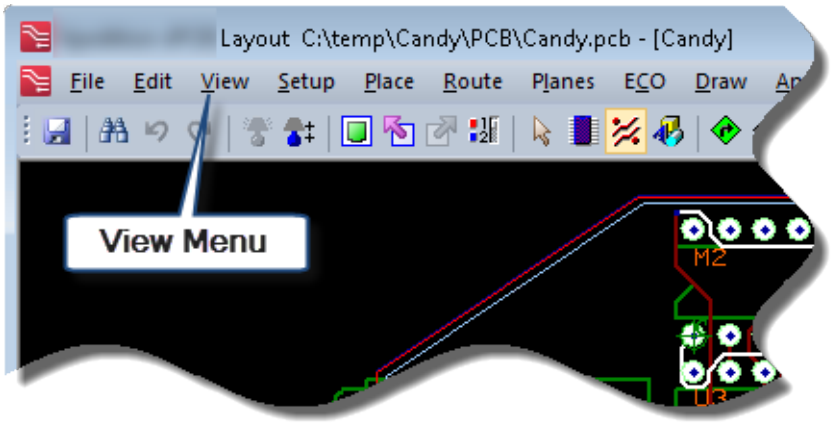
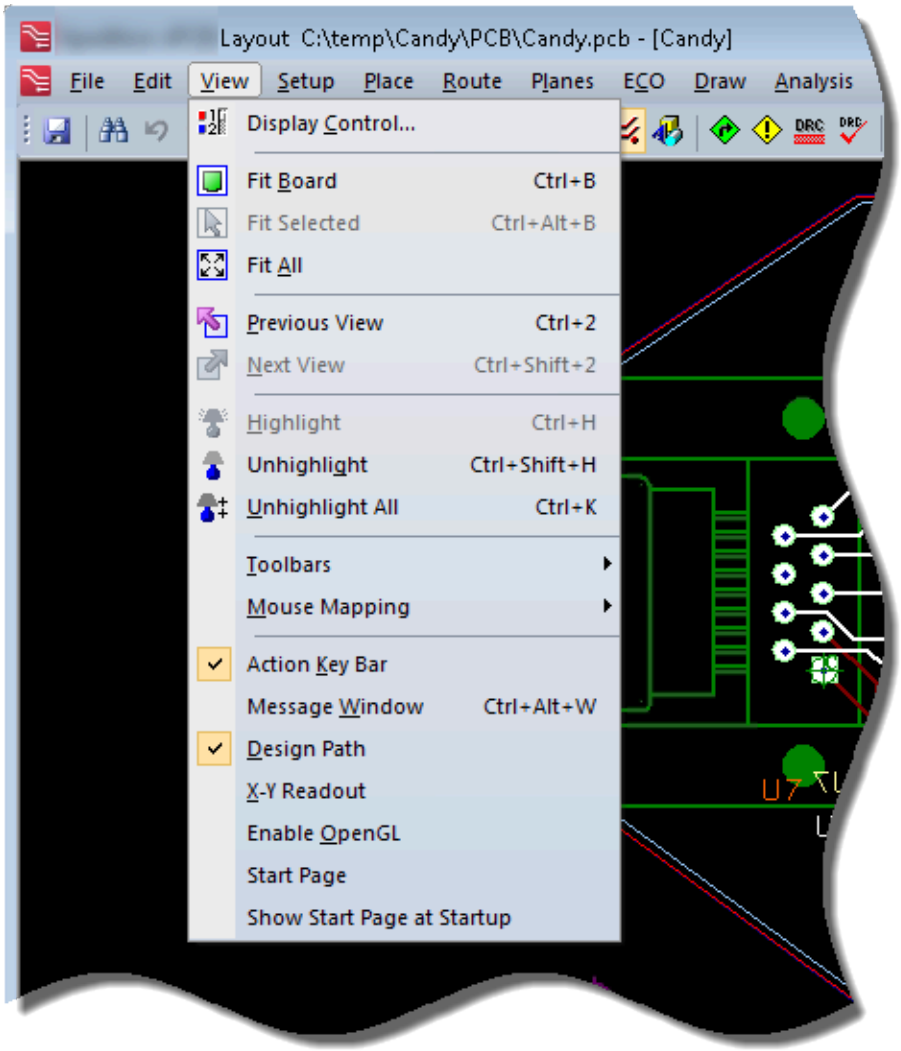


Figure 52. Contents of the View Pulldown Menu



The last item in the **View** menu is **Next View**. This example adds a new **Via Count** item after the **New View** item. First, return the collection of controls associated with the View control. To do this, Line 24 instantiates a collection named viewControlsColl and Line 25 returns the collection of controls associated with the **View** menu.

```
23  'Get the control collection for View
24  Dim viewControlsColl
25  Set viewControlsColl = viewMenuObj.Controls
```

Next, add the new item to the controls collection. Line 28 instantiates a variable named viaCntBtnObj that represents the new item (button). Line 29 uses the Add method to add this item to the list of View controls. There are four parameters associated with the Add method:

1. **Type** — This parameter specifies the control type. The types available are:

cmdControlButton	0	' A simple button
cmdControlPopup	1	' A menu
cmdControlButtonSeparator	2	' A button separator, horizontal line

2. **ID** — This parameter is an optional command ID (it is not generally used).

3. **Parameter** — This parameter is reserved for future use and is optional.

4. **Before** — This parameter specifies the position for the new control (a value of –1 indicates the end of the existing collection of items).

```
27  ' Create the new button by adding to the control collection
28  Dim viaCntBtnObj
29  Set viaCntBtnObj = viewControlsColl.Add(cmdControlButton,,,-1)
```

After the script adds the new item to the list of View controls, the script sets up the menu item. Line 32 uses the string “&Via Count” to define the caption that appears in the drop-down menu. The '&' character indicates that the following character ('V' in this example) appears underlined and acts as a shortcut key). Line 33 specifies the description text for the new item.

```
31  ' Configure the new button
32  viaCntBtnObj.Caption = "&Via Count"
33  viaCntBtnObj.DescriptionText = "Displays the number of vias."
34  viaCntBtnObj.Target = ScriptEngine
35  viaCntBtnObj.ExecuteMethod = "OnDisplayViaCount"
```

Lines 34 and 35 define the function to run when the end-user selects the new menu item. Line 34 specifies the “Target”, which you can visualize as the location where this function “lives.” This example uses ScriptEngine, which is an implicit object that equates to “The same engine that is running the current script.” Line 35 uses ExecuteMethod to specify that the script engine run the function named OnDisplayViaCount.

The name of this function is user-defined. This example simply prefix the name of the original function with OnDisplay, where “On” commonly signifies that a function is an event handler (in this case the event is the button-press) and “Display” indicates that the command displays something.

Observe the statement on Line 38. In previous chapters’ example scripts, the script exits after it runs. This would be a problem in this example, because if the script exits, then the script engine cannot call the button-press event handler. To solve this problem, Line 38 sets the DontExit property of the Scripting object to the Boolean value of True. This means that the script continues running until the current session ends.

```
37  ' Keep this script running so that the handler can be executed.
38  Scripting.DontExit = True
```

Note:
The DontExit property is valid only for internal scripts. You cannot use DontExit in scripts run with the mgcscript script engine (refer to "Running Scripts from the Command Line" on page 133). When using the *mgcscript* script engine, create an infinite loop to keep the script from exiting (refer to "Introducing Events and the Scripting Object" on page 179).

Finally, Lines 40 to 42 define the contents of the OnDisplayViaCount function. All handlers for menu entries have a single integer parameter named nID. The parameter must be present in order for the handler to work, however the function does nothing with the parameter. The body of this function contains the statement(s) to run when the end-user selects the item. In this example, a single statement calls the ProcessKeyin method of the Gui object. The ProcessKeyin method runs any valid key-in command. This example uses the same key-in command (the string parameter "run *C:\temp\ViaCount.vbs*") that is entered in "Running Your First Script" on page 59.

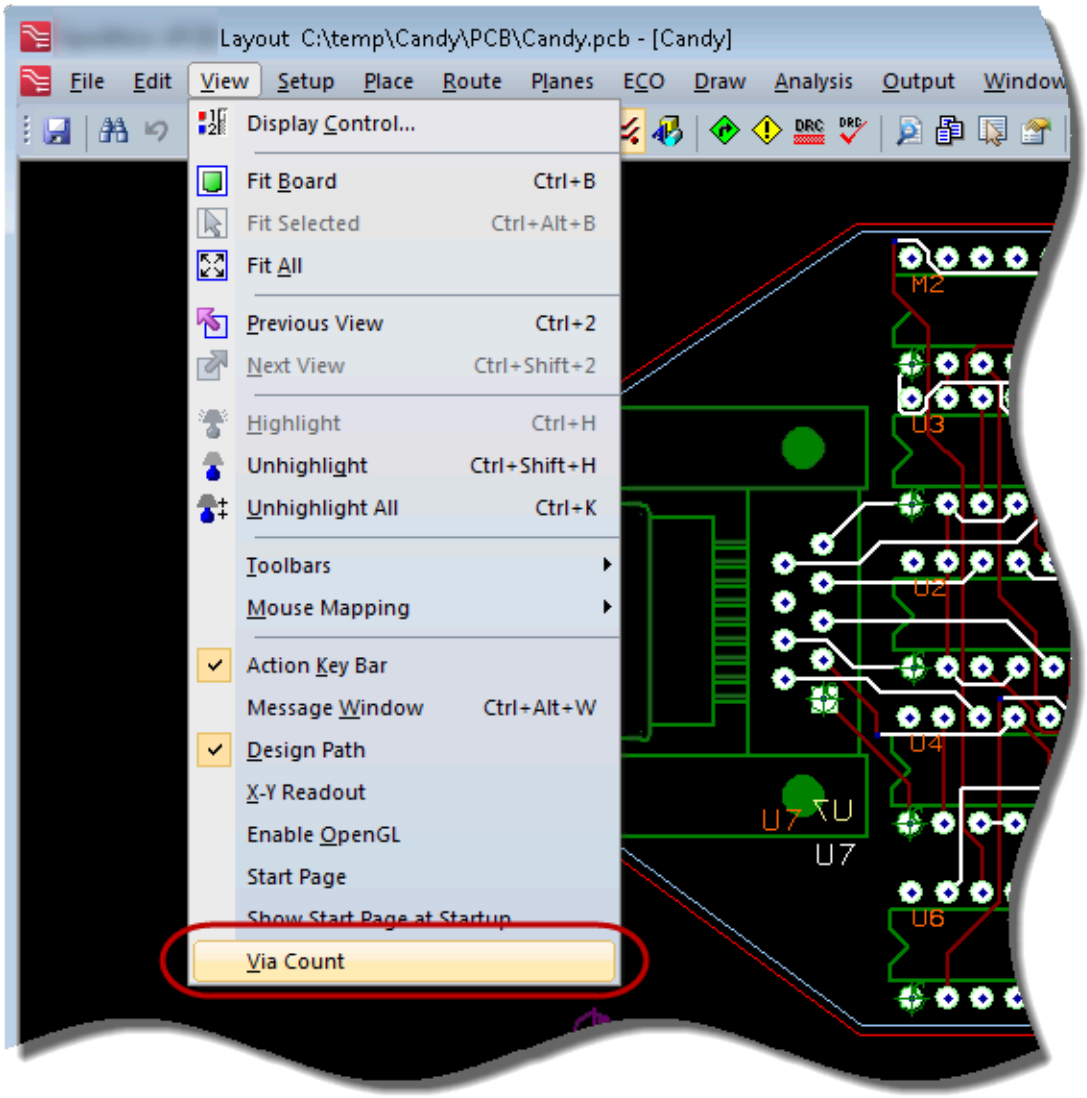
```
40  Function OnDisplayViaCount(nID)
41      Call pcbApp.Gui.ProcessKeyin("run C:\temp\ViaCount.vbs")
42  End Function
```

Running the Script

Run the script from within Xpedition Layout (this technique is detailed in "Running Scripts from an Application" on page 161).

1. Launch Xpedition Layout.
2. You may optionally open a layout design of your choice. However, this is not required to proceed.
3. Right-click on the toolbar and ensure that the **Keyin Command** option is enabled.
4. Enter the string "run *C:\temp\ViaCountMenuDual.vbs*" into the **Keyin Command** field and press Enter to run this script.
5. After you have run this script, the script exits and nothing appears to have changed. However, this script creates a new menu item as illustrated in Figure 53: New Via Count Item .

Figure 53. New Via Count Item

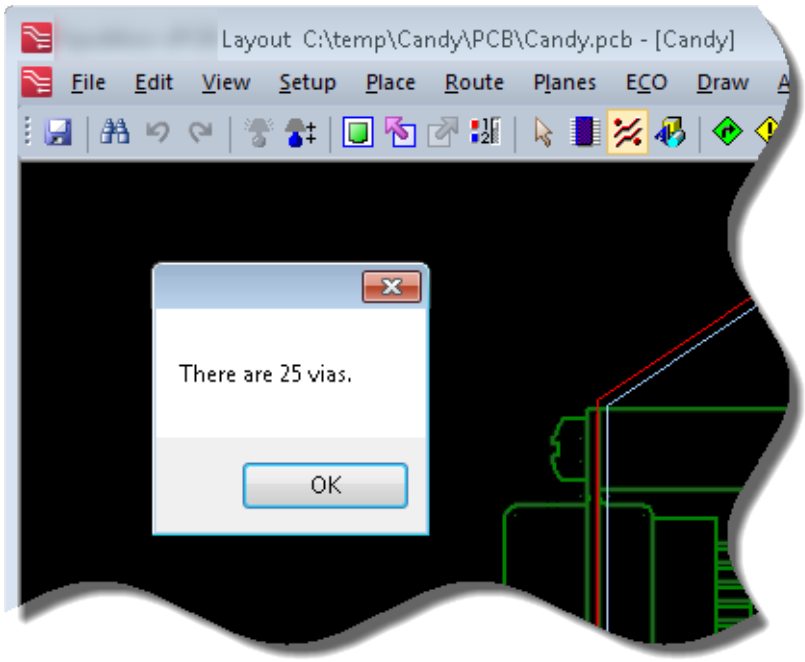


Test this new menu item to assure that it runs the *ViaCount.vbs* script.

6. Open a layout design if you have not already done so.

7. Choose **View > View Count** and observe that it launches the *ViaCount.vbs* script as illustrated in Figure 54: Using View > Via Count to Run ViaCount.vbs .

Figure 54. Using View > Via Count to Run ViaCount.vbs



Note:
The new menu item remains active for the duration of this application session. Scripts that modify or add menu items typically run automatically at startup (refer to "Running Startup Scripts" on page 171).

Tip
This example adds the new menu item to the bottom of the existing menu. In some cases, you may want to add the new item somewhere in the middle of an existing menu. Do this by using an integer to specify the location of the new item. However, this is not an ideal solution, because running another script can change the number of items in the menu. In order to address this issue, it is possible to create some "helper functions" (refer to "Basic Building-Block Examples" on page 187).

Adding a Menu Item With One Script

In this example, you combine the functionality of the previous example's scripts, *ViaCountMenuDual.vbs* and *ViaCount.vbs*. The resulting script, *ViaCountMenuSingle.vbs* adds a new item to the bottom of the **View** pulldown menu. The script then counts the number of vias in the design.

The previous example employs two scripts, which has its advantages in that it is very flexible and facilitates reuse. However, this also requires that you manage two scripts. The alternative is to create a single script.

Note:
The script created in the previous example is still running. In order to terminate the script and remove its associated menu item, close Xpedition Layout.

Creating the Script

Use the scripting editor of your choice to open the *ViaCountMenuDual.vbs* script created in the previous example and save it as *ViaCountMenuSingle.vbs* in your *C:\temp* folder.

Open the original *ViaCount.vbs* script. It should appear something like the following:

```
 0  ' ORIGINAL ViaCount.vbs SCRIPT
 1  Option Explicit
 2
 3  ' Add any type libraries to be used.
 4  Scripting.AddTypeLibrary("MGCPCB.ExpeditionPCBApplication")
 5
 6  ' Get the Application object
 7  Dim pcbAppObj
 8  Set pcbAppObj = Application
 9
10  ' Get the active document
11  Dim pcbDocObj
12  Set pcbDocObj = pcbAppObj.ActiveDocument
13
14  ' License the document
15  ValidateServer(pcbDocObj)
16
17  ' Get the vias collection
18  Dim viaColl
19  Set viaColl = pcbDocObj.Vias
20
21  ' Get the number of vias in collection
22  Dim countInt
23  countInt = viaColl.Count
24
25  MsgBox("There are " & countInt & " vias.")
26
27  ''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
28  'Local functions
29
30  ' Server validation function
31  Function ValidateServer(docObj)
32
33      Dim keyInt
34      Dim licenseTokenInt
35      Dim licenseServerObj
36
37      ' Ask Expedition’s document for the key
38      keyInt = docObj.Validate(0)
39
```

```
40      ' Get license server
41      Set licenseServerObj =
            CreateObject("MGCPCBAutomationLicensing.Application")
42
43      ' Ask the license server for the license token
44      licenseTokenInt = licenseServerObj.GetToken(keyInt)
45
46      ' Release license server
47      Set licenseServerObj = nothing
48
49      ' Turn off error messages (validate may fail if the
            token is incorrect
50      On Error Resume Next
51      Err.Clear
52
53      ' Ask the document to validate the license token
54      docObj.Validate(licenseTokenInt)
55      If Err Then
56          ValidateServer = 0
57      Else
58          ValidateServer = 1
59      End If
60
61  End Function
```

Open *ViaCountMenuSingle.vbs* and make the following changes:

1. Select Line 4 from the original *ViaCount.vbs* script (the line that adds the Xpedition Layout type library) and paste it in between Lines 3 and 4 in the *ViaCountMenuSingle.vbs* script (this is Line 4). *ViaCountMenuSingle.vbs* adds two type libraries. In the previous example, the *ViaCountMenuDual.vbs* script does not require the Xpedition Layout type library because the second script, *ViaCount.vbs*, adds the type library.

2. Select Lines 10 through 25 from the*ViaCount.vbs* script and use them to replace the contents of the OnDisplayViaCount function in the *ViaCountMenuSingle.vbs* script (these lines are Lines 42 through 56).

   Observe that the *ViaCountMenuSingle.vbs* script does not require Lines 7 and 8 from the *ViaCount.vbs* script (the statements that acquire the Application object) because these statements are in Lines 8 and 9 of the new script.

3. Finally, select Lines 27 through 61 from the *ViaCount.vbs* script (these are the statements that perform the server validation function) and paste them into the end of the *ViaCountMenuSingle.vbs* script. These lines appear as Lines 60 through 94.

The complete *ViaCountMenuSingle.vbs* script is shown in ViaCountMenuSingle.vbs.

ViaCountMenuSingle.vbs

This is the complete ViaCountMenuSingle.vbs script.

```
 0 ' NEW ViaCountMenuSingle.vbs SCRIPT
 1  Option Explicit
 2
 3  ' Add any type libraries to be used.
 4  Scripting.AddTypeLibrary("MGCPCB.ExpeditionPCBApplication")
 5  Scripting.AddTypeLibrary("MGCSDD.CommandBarsEx")
 6
 7  ' Get the application object.
 8  Dim pcbApp
 9  Set pcbApp = Application
10
11  ' Get the document menu bar.
12  Dim docMenuBarObj
13  Set docMenuBarObj = pcbApp.Gui.CommandBars("Document Menu Bar")
14
15  ' Get the collection of controls for the menu
16  '(i.e. menu popup buttons, File, Edit, View, etc...)
17  Dim docMenuBarCtrlColl
18  Set docMenuBarCtrlColl = docMenuBarObj.Controls
19
20  ' Find the View menu control
21  Dim viewMenuObj
22  Set viewMenuObj = docMenuBarCtrlColl.Item("&View")
23
24  'Get the control collection for View
25  Dim viewControlsColl
26  Set viewControlsColl = viewMenuObj.Controls
27
28  ' Create the new button by adding to the control collection
29  Dim viaCntBtnObj
30  Set viaCntBtnObj = viewControlsColl.Add(cmdControlButton,,,-1)
31
32  ' Configure the new button
33  viaCntBtnObj.Caption = "&Via Count"
34  viaCntBtnObj.DescriptionText = "Displays the number of vias."
35  viaCntBtnObj.Target = ScriptEngine
36  viaCntBtnObj.ExecuteMethod = "OnDisplayViaCount"
37
38  ' Keep this script running so that the handler can be executed.
39  Scripting.DontExit = True
40
41  Function OnDisplayViaCount(nID)
42    Dim pcbDoc
43    Set pcbDoc = pcbApp.ActiveDocument
44
45    ' License the document
46    ValidateServer(pcbDoc)
47
48    ' Get the vias collection
49    Dim viaColl
50    Set viaColl = pcbDoc.Vias
51
```

```
52    ' Get the number of vias in collection
53    Dim countInt
54    countInt = viaColl.Count
55
56    MsgBox("There are " & countInt & " vias.")
57  End Function
58
59
60  ''''''
```

1. Launch Xpedition Layout.

2. You may open a layout design of your choice; however, this is not required to proceed.

3. Right-click the toolbar and ensure that the **Keyin Command** option is enabled.

4. Enter the string run *C:\temp\ViaCountMenuSingle.vbs* into the **Keyin Command** field and press Enter to run this script.

5. After running this script, the script exits and nothing appears to have changed. However, this script creates a new menu item as illustrated in Figure 55: New Via Count Menu Item .

   Check this new menu item to ensure that it performs as expected.

6. Open a layout design if you have not already done so.

7. Choose **View > Via Count** and observe that the menu item counts the vias as illustrated in Figure 56: Using View > Via Count .

Figure 55. New Via Count Menu Item

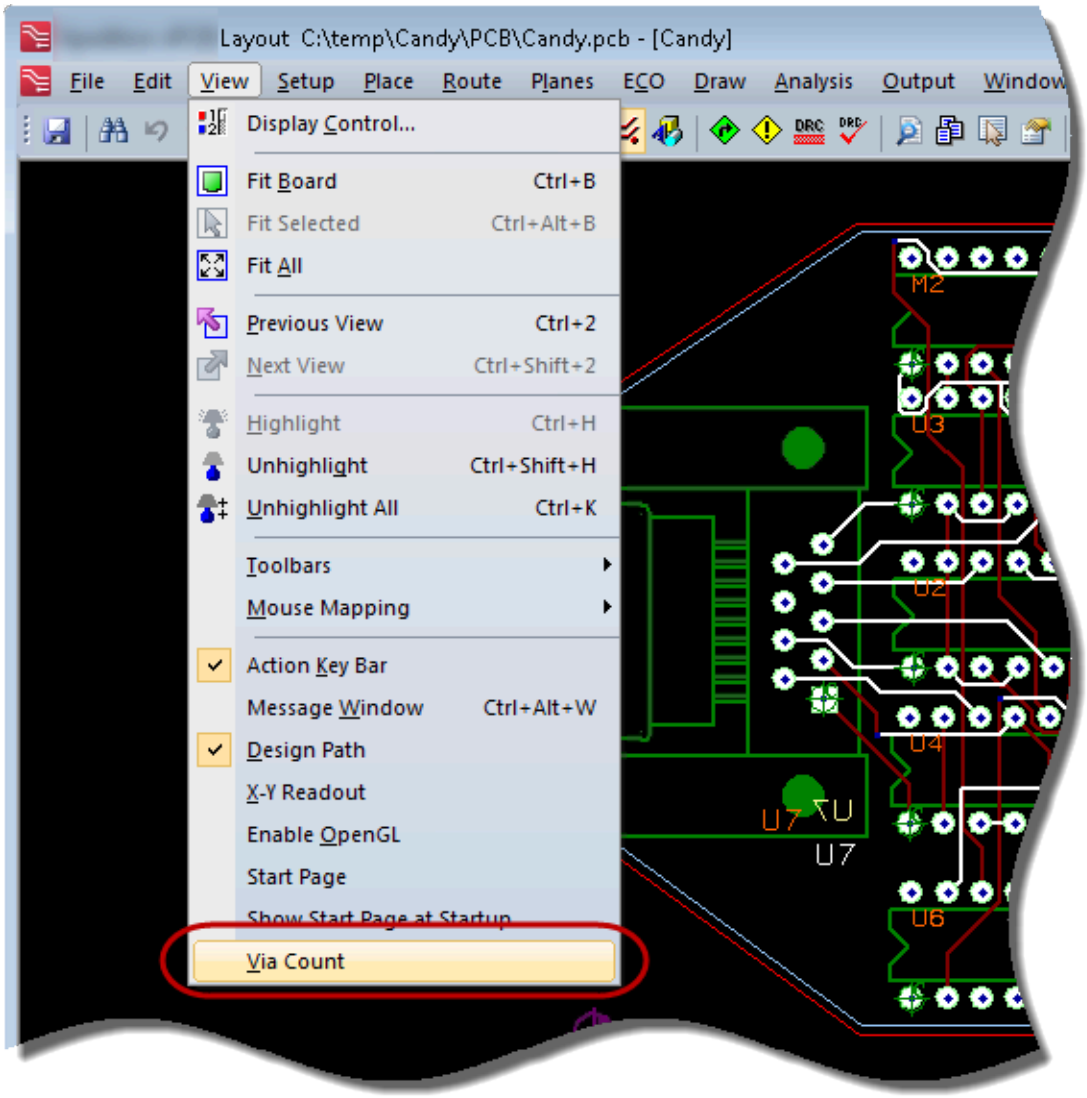
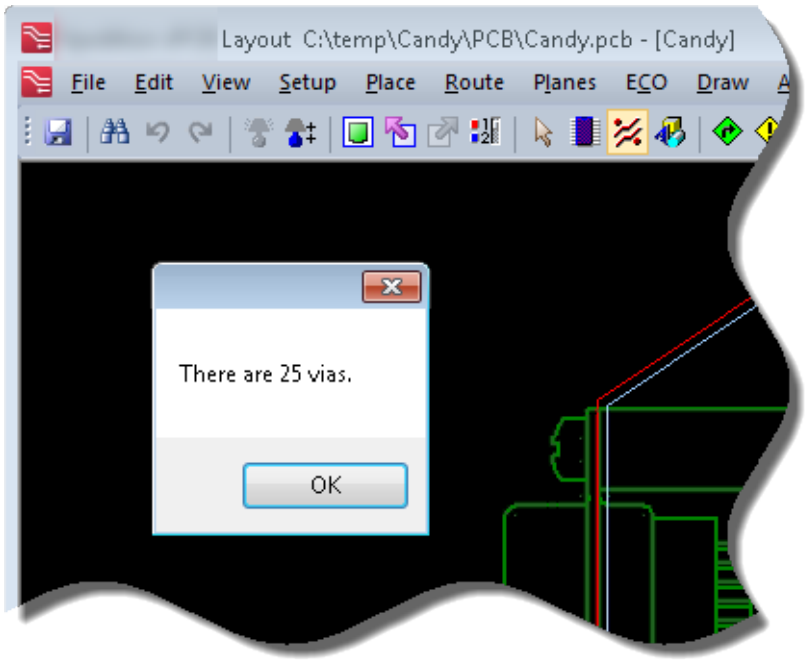


Figure 56. Using View > Via Count



Note:
The new menu item remains in the menu bar for the duration of this application session.

Adding a New Top-Level Menu

This example describes the process for creating a new top-level menu.

Note:
The script created in the previous example is still running. In order to terminate the script and remove its associated menu item, close Xpedition Layout.

Creating the Script

Use the scripting editor of your choice to open the *ViaCountMenuSingle.vbs* script and save it as *ViaCountMenuTop.vbs* in your *C:\temp* folder.

It is necessary to make only a few modifications to the script for this example. Open *ViaCountMenuTop.vbs* and make the following changes:

1. Modify the statements at Lines 20 through 22 as shown in ViaCountMenuTop.vbs. Rename the original viewMenuObj variable to countMenuObj. These modifications change “finding the existing **View** menu item” (using the *Item* property) to “adding a new main menu item” (using the *Add* method). The fourth parameter of the *Add* method, –1, at the end of Line 22, instructs the Command Bar Server to add this new menu to the end of the existing menu bar.

2. Add the comment in Line 24 and the statement on Line 25 as shown below. This adds the caption “Count” to the new menu item.

3. Change the variable name viewControlsColl to countControlsColl on Lines 28, 29, and 33.

4. Change the variable name viewMenuObj to countMenuObj on line 29.

ViaCountMenuTop.vbs shows the resulting script.

ViaCountMenuTop.vbs

This is the complete ViaCountMenuTop.vbs script.

```
 1  Option Explicit
 2
 3  ' Add any type libraries to be used.
 4  Scripting.AddTypeLibrary("MGCPCB.ExpeditionPCBApplication")
 5  Scripting.AddTypeLibrary("MGCSDD.CommandBarsEx")
 6
 7  ' Get the application object.
 8  Dim pcbApp
 9  Set pcbApp = Application
10
11  ' Find the document menu bar.
12  Dim docMenuBarObj
13  Set docMenuBarObj = pcbApp.Gui.CommandBars("Document Menu Bar")
14
15  ' Get the collection of controls for the menu
16  '(i.e. menu popup buttons, File, Edit, View, etc...)
17  Dim docMenuBarCtrlColl
18  Set docMenuBarCtrlColl = docMenuBarObj.Controls
19
20  ' Create the new button by adding to the control collection
21  Dim countMenuObj
22  Set countMenuObj = docMenuBarCtrlColl.Add(cmdControlPopup,,,-1)
23
24  ' Configure the menu control
25  countMenuObj.Caption = "Count"
26
27  'Get the control collection for the new Count menu
28  Dim countControlsColl
29  Set countControlsColl = countMenuObj.Controls
30
31  ' Create the new button by adding to the control collection
32  Dim viaCntBtnObj
33  Set viaCntBtnObj = countControlsColl.Add(cmdControlButton,,,-1)
34
35  ' Configure the new button
36  viaCntBtnObj.Caption = "&Via Count"
37  viaCntBtnObj.DescriptionText = "Displays the number of vias."
38  viaCntBtnObj.Target = ScriptEngine
```

```
viaCntBtnObj.ExecuteMethod = "OnDisplayViaCount"

' Keep this script running so that the handler can be executed.
Scripting.DontExit = True

Function OnDisplayViaCount(nID)
  ' Get the active document
  Dim pcbDoc
  Set pcbDoc = pcbApp.ActiveDocument

  ' License the document
  ValidateServer(pcbDoc)

  ' Get the vias collection
  Dim viaColl
  Set viaColl = pcbDoc.Vias

  ' Get the number of vias in collection
  Dim countInt
  countInt = viaColl.Count

  MsgBox("There are " & countInt & " vias.")
End Function


''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
'Local functions

' Server validation function
Function ValidateServer(docObj)

    Dim keyInt
    Dim licenseTokenInt
    Dim licenseServerObj

    ' Ask Expedition’s document for the key
    keyInt = docObj.Validate(0)

    ' Get license server
    Set licenseServerObj =
         CreateObject("MGCPCBAutomationLicensing.Application")

    ' Ask the license server for the license token
    licenseTokenInt = licenseServerObj.GetToken(keyInt)

    ' Release license server
    Set licenseServerObj = nothing

    ' Turn off error messages.
    On Error Resume Next
    Err.Clear

    ' Ask the document to validate the license token
```

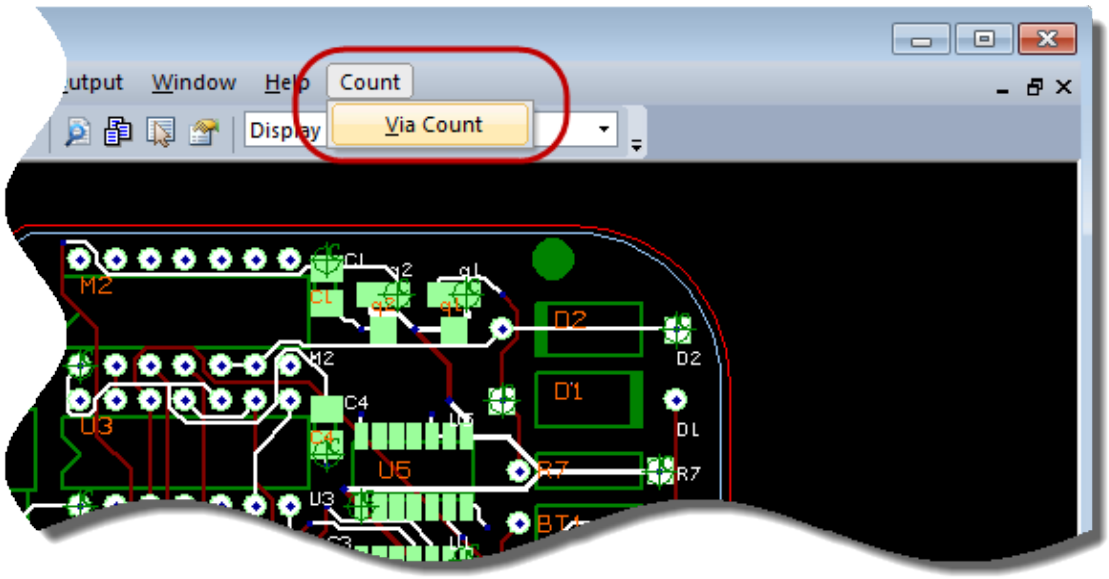
```
    docObj.Validate(licenseTokenInt)
    If Err Then
        ValidateServer = 0
    Else
        ValidateServer = 1
    End If

End Function
```

Running the Script

Run the script from within Xpedition Layout (this technique is detailed in "Running Scripts from an Application" on page 161).

1. Launch Xpedition Layout.
2. You may open a layout design of your choice; however, this is not required to proceed.
3. Right-click the toolbar and ensure that the **Keyin Command** option is enabled.
4. Enter the string run *C:\temp\ViaCountMenuTop.vbs* into the **Keyin Command** field and press **Enter** to run this script.
5. After running this script, the new menu appears as illustrated in Figure 57: New Count Menu .

Figure 57. New Count Menu



Check this new menu to ensure that it performs as expected.

6. Open a layout design if you have not already done so.

7. Choose **Count > Via Count** and ensure the new menu performs as expected.

Chapter 8
Adding Toolbar Buttons

This section describes how to add toolbar buttons (icons) to an application's toolbar using the Command Bar Server.

The Command Bar Server is a separate server that applications, such as Xpedition Layout and Xpedition FabLink, use to create toolbar buttons. For example, a script can add a button to an existing toolbar and then bind the button to another script.

A scripting or programming language accesses the Command Bar Server to create and modify toolbar buttons. This chapter presents an example that adds a new button to an existing toolbar and runs the ViaCount.vbs script.

Note:
Xpedition Layout and Xpedition Designer each use different command bar servers. Scripts must call the correct command bar server for the respective product. Use MGCSDD.CommandBarsEx for Xpedition Layout toolbars; use Viewlogic.CommandBarsEx for Xpedition Designer toolbars.

The Command Bar Server also modifies pulldown menus on the menu bar as discussed in "Adding Menus and Menu Items" on page 107.

Adding a New Button
Viewing the Utilities Toolbar

Adding a New Button

In this example, you create a script, *ViaCountToolBar.vbs*, which adds a new button to the end of the **Utilities** toolbar.

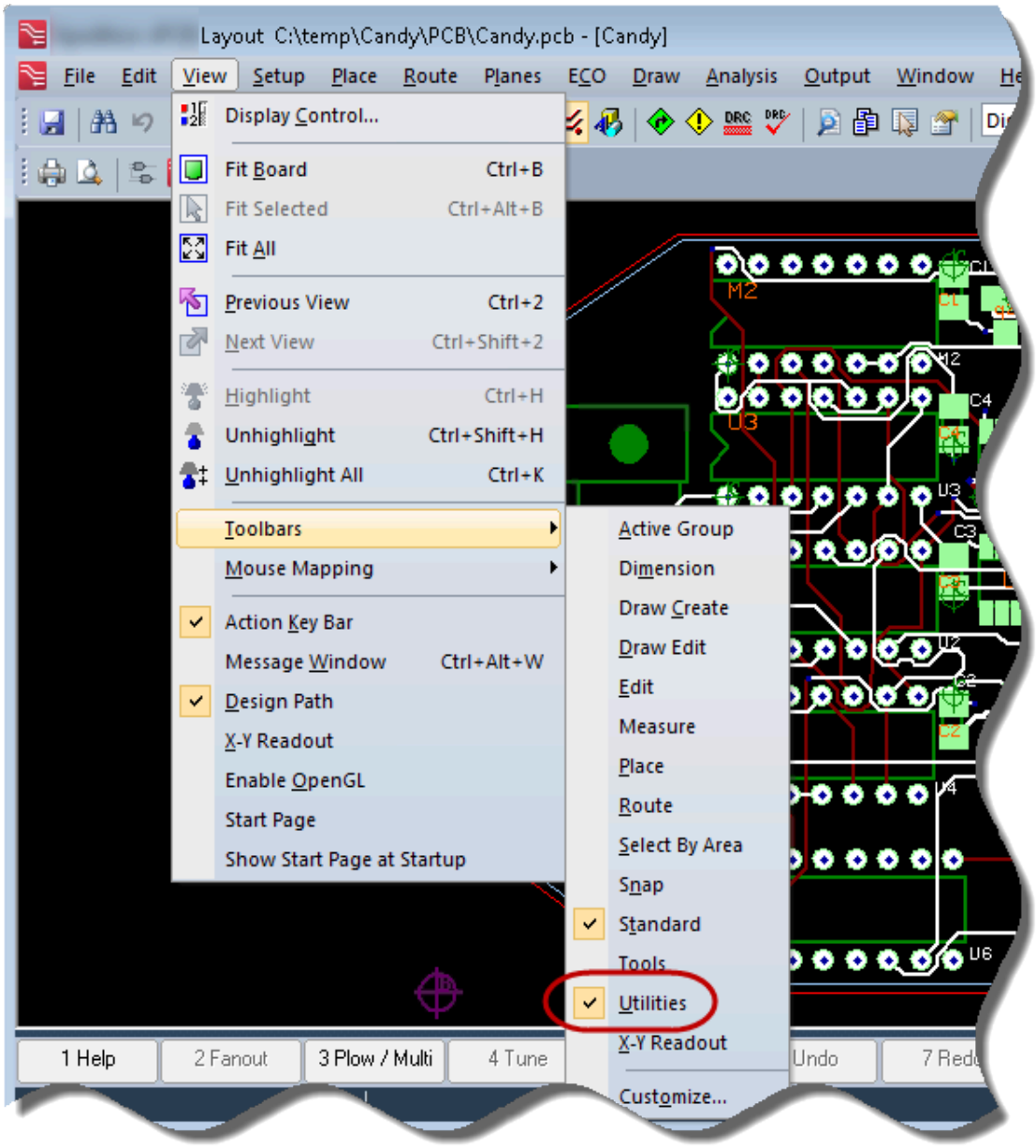
The script then ties this button to the *ViaCount.vbs* script created in "Running Your First Script" on page 59

Viewing the Utilities Toolbar

Before creating the script, view the **Utilities** toolbar in Xpedition Layout.

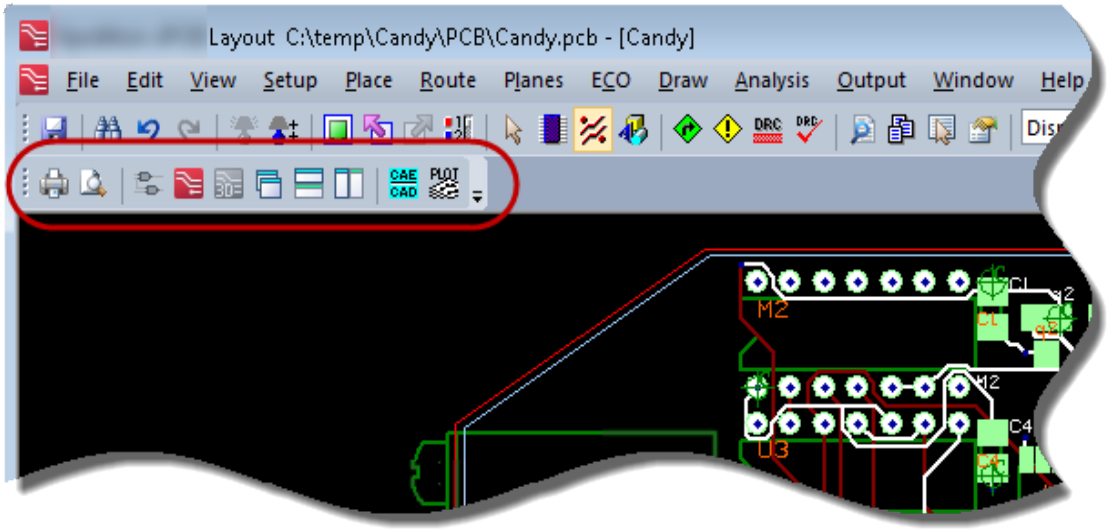
1. Launch Xpedition Layout.
2. You may optionally open a layout design of your choice. However, this is not required to proceed.
3. Choose**View > Toolbars** and ensure that the **Utilities** toolbar is visible as illustrated in Figure 58: View > Toolbars > Utilities .

Figure 58. View > Toolbars > Utilities



4. Observe that the last button in the **Utilities** toolbar is the **Plot** button as illustrated in Figure 59: Original Utilities Toolbar . The script adds a new **Via Count** button to the right of the **Plot** button.

Figure 59. Original Utilities Toolbar



Creating the Script

Use the scripting editor of your choice to enter the following script and save it as ViaCountToolBar.vbs in your C:\temp folder.

Line 1 declares Option Explicit (this statement is discussed in "VBScript Primer" on page 27 and "Option Explicit Statement" on page 522).

```
1  Option Explicit
```

Line 4 uses the AddTypeLibrary property of the Scripting object to access the type library associated with the Command Bar Server (the Scripting object is discussed in more detail in "Introducing Events and the Scripting Object" on page 179). Add this type library to your script anytime the script accesses the Command Bar Server.

```
3  ' Add the command bar type library so we can use its enumerates
4  Scripting.AddTypeLibrary("MGCSDD.CommandBarsEx")
```

Lines 7 and 8 return the Application (this is discussed in "Running Your First Script" on page 59). This script does not return a Document and does not perform validation. This is because this script does not access or modify any data within the design.

```
6  ' Get the application object (implicit app for internal scripts)
7  Dim pcbApp
8  Set pcbApp = Application
```

Lines 11 and 12 return the application's Gui object. This script example does not store the Gui object in a variable (which you could do). Instead, it simply returns the CommandBar object.

```
10  ' Find the document menu bar.
11  Dim utilitiesToolBar
12  Set utilitiesToolBar = pcbApp.Gui.CommandBars("Utilities")
```

The string parameter to the *CommandBars* property specifies a menu bar or a toolbar. The string “Utilities” indicates the **Utilities** toolbar and returns the appropriate CommandBar object.

To manipulate (and add items to) the toolbar, the script must return the list of controls that contain the buttons. To do this, Line 15 instantiates a collection named utilitiesControlsColl and Line 16 returns the list of button controls associated with the toolbar.

```
14  ' Get the collection of controls from utilities toolbar
15  Dim utilitiesControlsColl
16  Set utilitiesControlsColl = utilitiesToolBar.Controls
```

Next, add a new button to the collection of buttons. Line 19 instantiates a variable named viaCntBtnObj that represents the new button. Line 20 uses the Add method to add the button to the list of **Utilities** controls. There are four parameters associated with the Add method:

- **Type** — This parameter specifies the control type. In this example, cmdControlButton is a simple button.
- **ID** — This parameter is an optional command ID (it is not generally used).
- **Parameter** — This is reserved for future use and is optional.
- **Before** — This parameter specifies the position before which the new control is added (a value of –1 indicates the end of the existing collection of items).

```
18  ' Create the new button
19  Dim viaCntBtnObj
20  Set viaCntBtnObj =
       utilitiesControlsColl.Add(cmdControlButton,,,-1)
```

After the script adds the new button to the list of **Utilities** controls, the script sets up the button. Line 23 specifies the description text for the new button. Line 24 defines the "tool tip" text that displays when the end-user "hovers" the cursor over the button. Line 25 specifies the filename for the image (a bit map *.bmp file) of the button.

Note:
Use a graphics editor to create a suitable 16 x 16 pixel *.*bmp* button.

```
22  ' Configure the new button
23  viaCntBtnObj.DescriptionText = "Display # of vias in design."
24  viaCntBtnObj.TooltipText = "Display via count."
25  viaCntBtnObj.BitmapFile = "C:\temp\ViaCount.bmp"
26  viaCntBtnObj.Target = ScriptEngine
27  viaCntBtnObj.ExecuteMethod = "OnDisplayViaCount"
```

Line 26 and 27 define the function to run when the end-user selects the new button. Line 26 specifies the “Target”, which you can visualize as the location where this function “lives.” This example uses ScriptEngine, which is an implicit object that equates to—“The same engine that is running the current script.” Line 27 uses ExecuteMethod to specify that the script engine run the function named OnDisplayViaCount.

The name of this function is user-defined. This example simply prefixes the name of the original function with OnDisplay, where “On” commonly signifies that a function is an event handler (in this case the event is the button-press) and “Display” indicates that the command displays something.

Observe the statement on Line 30. In previous chapters’ example scripts, the script exits after it runs. This would be a problem in this example, because if the script exits, then the script engine cannot call the button-press event handler. To solve this problem, Line 30 sets the DontExit property of the Scripting object to the Boolean value of True. This means that the script continues running until the current session ends.

```
29  ' Keep this script running so that the handler can be executed.
30  Scripting.DontExit = True
```

Finally, Lines 32 to 34 define the contents of the OnDisplayViaCount function. All handlers for toolbar button entries have a single integer parameter named nID. The parameter must be present in order for the handler to work, however the function does nothing with the parameter. The body of this function contains the statement(s) to run when the end-user selects the item. In this example, only a single statement calls the ProcessKeyin method of the Gui object. The ProcessKeyin method runs any valid key-in command. This example uses the same key-in command (the string parameter “run C:\Temp \ViaCount.vbs”) that is entered in “Running Your First Script” on page 59.

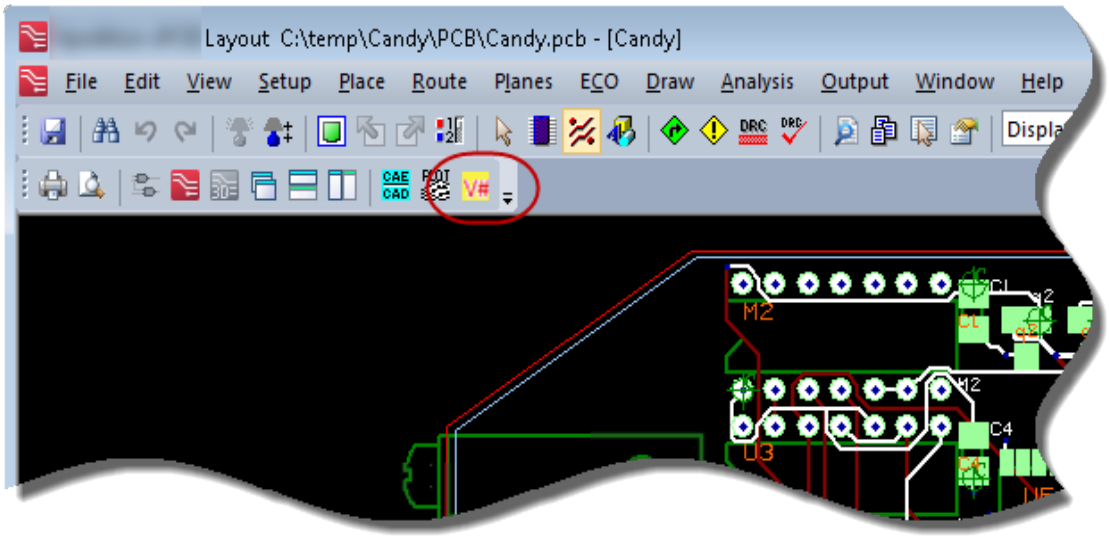
```
32  Function OnDisplayViaCount(nID)
33    Call pcbApp.Gui.ProcessKeyin("run C:\temp\ViaCount.vbs")
34  End Function
```

Running the Script

Run the script from within Xpedition Layout (this technique is detailed in “Running Scripts from an Application” on page 161).

1. Launch Xpedition Layout.
2. You may optionally open a layout design of your choice. However, this is not required to proceed.
3. Right-click on the toolbar and ensure that the **Keyin Command** option is enabled.
4. Enter the string run *C:\temp\ViaCountToolBar.vbs* into the **Keyin Command** field and press Enter to run this script.
5. After you have run this script, the new button appears on the **Utilities** toolbar as illustrated in Figure 60: New Via Count Button on the Utilities Toolbar .

Figure 60. New Via Count Button on the Utilities Toolbar



Test this new button to assure that it runs the ViaCount.vbs script.

6. Open a layout design if you have not already done so.

7. Click the new toolbar button and observe that it runs the ViaCount.vbs script.

Note:
The new toolbar item remains active for the duration of this session. Scripts that modify or add toolbar buttons typically run automatically at startup (refer to "Running Startup Scripts" on page 171).

Tip
This example adds the new button to the end (right-hand side) of the existing toolbar. In some cases, you may want to add the button somewhere in the middle of an existing toolbar. In order to facilitate this, it is possible to create some "helper functions" (refer to "Basic Building-Block Examples" on page 187).

Chapter 9
Running Scripts from the Command Line

This section provides several exercises for running scripts from the command line and include—passing arguments, attaching to existing applications, and creating invisible applications.

The Windows operating system has two built-in scripting engines:

- **Wscript.exe** — The most recent (and default) Windows scripting engine.
- **Cscript.exe** — The original (older) Windows scripting engine.

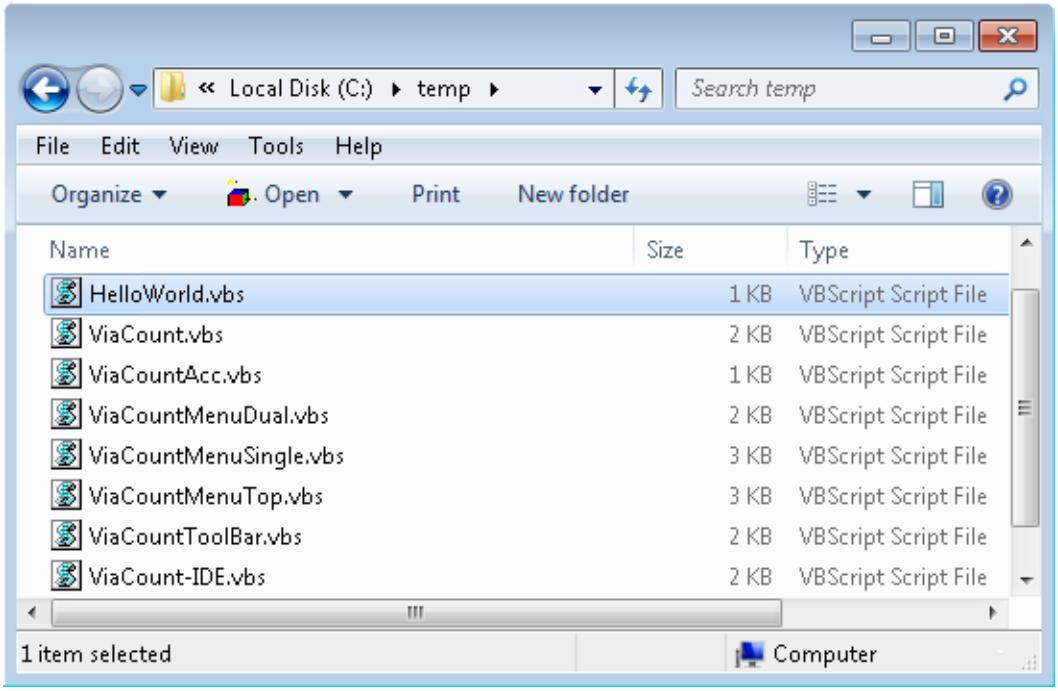
By default VBScript (**.vbs*) files associate with the *Wscript.exe* scripting engine. You can change this association to another scripting engine such as *Cscript.exe* or the Mentor Graphics engine, mgcscript.exe.

Note that there is no **.exe* file extension on UNIX and Linux systems. Also, only *mgcscript* is available on UNIX and Linux systems.

Each scripting engine has its own unique set of implicit objects. In order to illustrate what this means, perform the following simple procedure:

1. Open an Explorer window and use it to locate and view the contents of the *C:\temp* folder (directory), which is where you save scripts. This is illustrated in Figure 61: Contents of the C:\temp Folder .

Figure 61. Contents of the C:\temp Folder



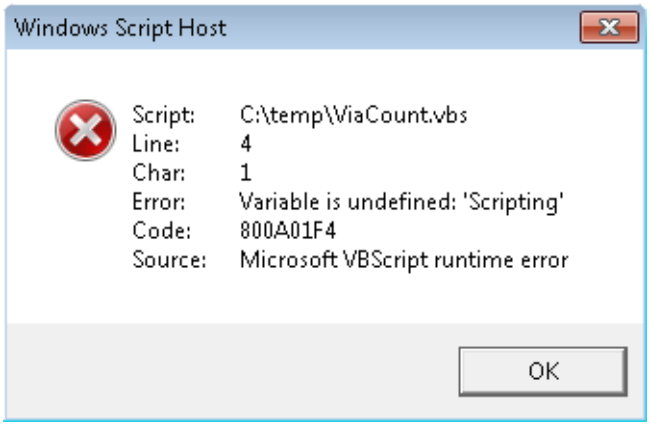
2. Double-click the *HelloWorld.vbs* script created in "VBScript Primer" on page 27. By default, this script runs on the *Wscript.exe* scripting engine. The result is the dialog shown in Figure 62: Hello World Script Runs in Wscript.exe .

Figure 62. Hello World Script Runs in Wscript.exe

3. Launch Xpedition Layout and open a layout design of your choice.

4. Double-click the *ViaCount.vbs* script created in Running Your First Script. This script fails to run as illustrated in Figure 63: Via Count Script Fails in Wscript.exe . It is attempting to access the Scripting object, which the *Wscript.exe* scripting engine does not support.

Figure 63. Via Count Script Fails in Wscript.exe



As an alternative, you can run scripts from the command line. To do this, you specify the scripting engine that runs the script as discussed in the following sections.

Running Command Line Scripts

Working With Applications and Documents

Running Command Line Scripts

This section contains exercises that illustrate how to run scripts from the command line. This includes scripts with and without arguments. This section assumes you are working in the Windows operating system.

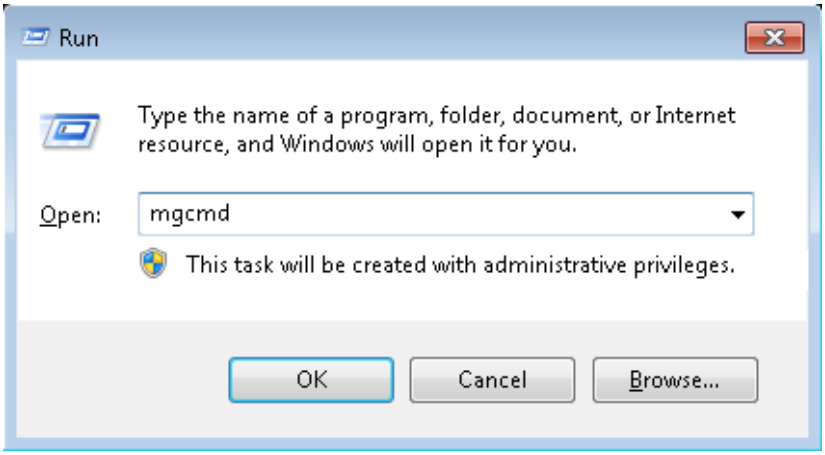
Running Command Line Scripts—With No Arguments
Displaying Command Line Arguments
Using Command Line Arguments
Adding Additional Error Handling

Running Command Line Scripts— With No Arguments

This example presents *mgcscript* running a script from the command line. The script has no arguments.

1. Do one of the following, based on your operating system.
   - **Windows 7 —** Click the Windows **Start** button, choose **Run**, then type "mgcmd" as illustrated in Figure 64: Using the Run Dialog .
   - **Windows 8 —** Press the **<Windows>-R** key combination, then type "mgcmd" as illustrated in Figure 64: Using the Run Dialog .

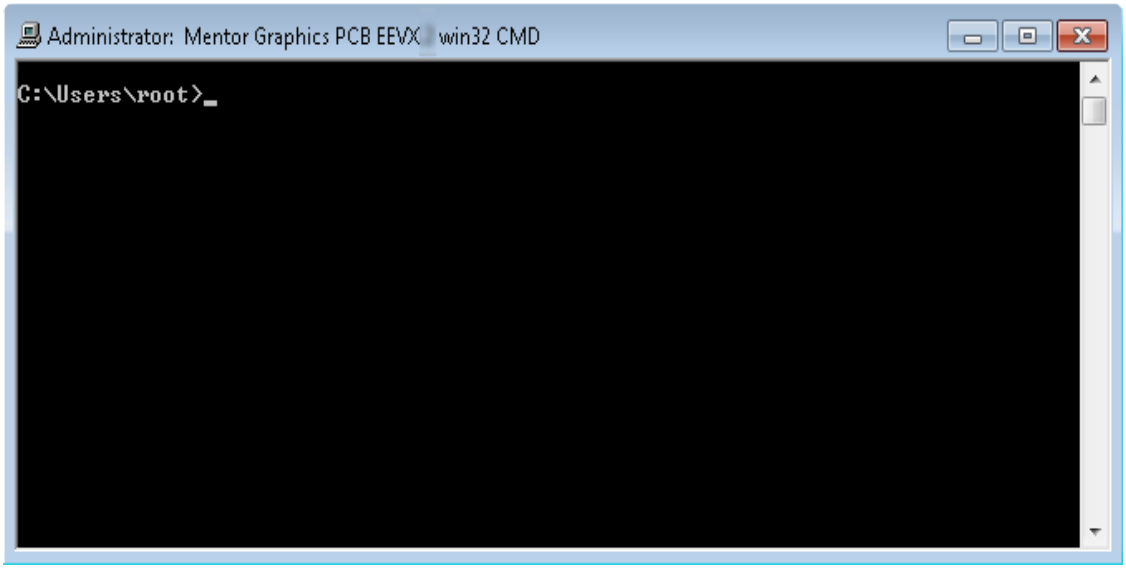
Figure 64. Using the Run Dialog



Note:
Multiple side-by-side releases of Mentor Graphics software are supported on Windows. When running a command line script, you must use the MGC PCB Command Window (“mgcmd”) rather than the Windows console.

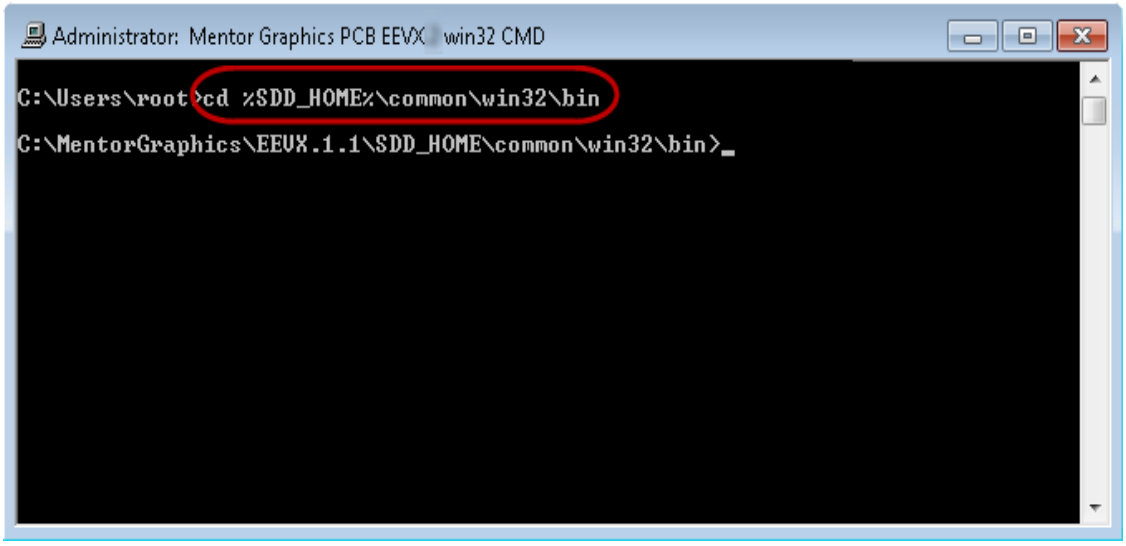
2. Click **OK** or press Enter to run this command. The command launches a console window (sometimes referred to as a terminal) as illustrated in Figure 65: Launching a Console (Terminal) Window .

Figure 65. Launching a Console (Terminal) Window



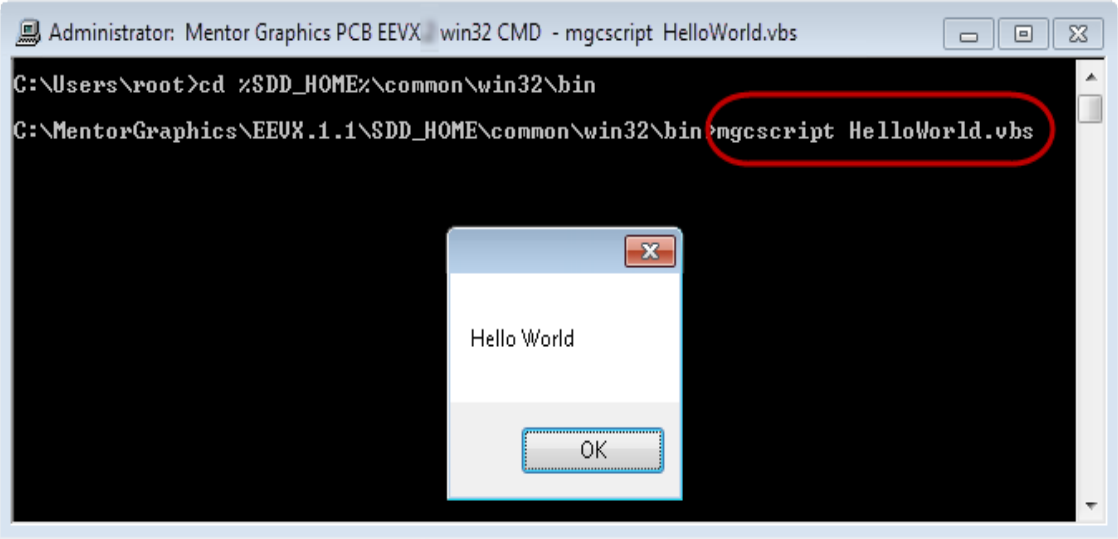
3. Enter the **cd** ("change directory") command to change the context to the *C:\temp* directory (more commonly referred to as a folder in Windows) as illustrated in Figure 66: Setting the Working Directory .

Figure 66. Setting the Working Directory



4. The Mentor Graphics scripting engine is named mgcscript.exe. Enter mgcscript HelloWorld.vbs and press Enter. This calls the mgcscript scripting engine and passes the HelloWorld.vbs script as an argument. The script runs as illustrated in Figure 67: Running the Hello World Script From the Command Line .

Figure 67. Running the Hello World Script From the Command Line



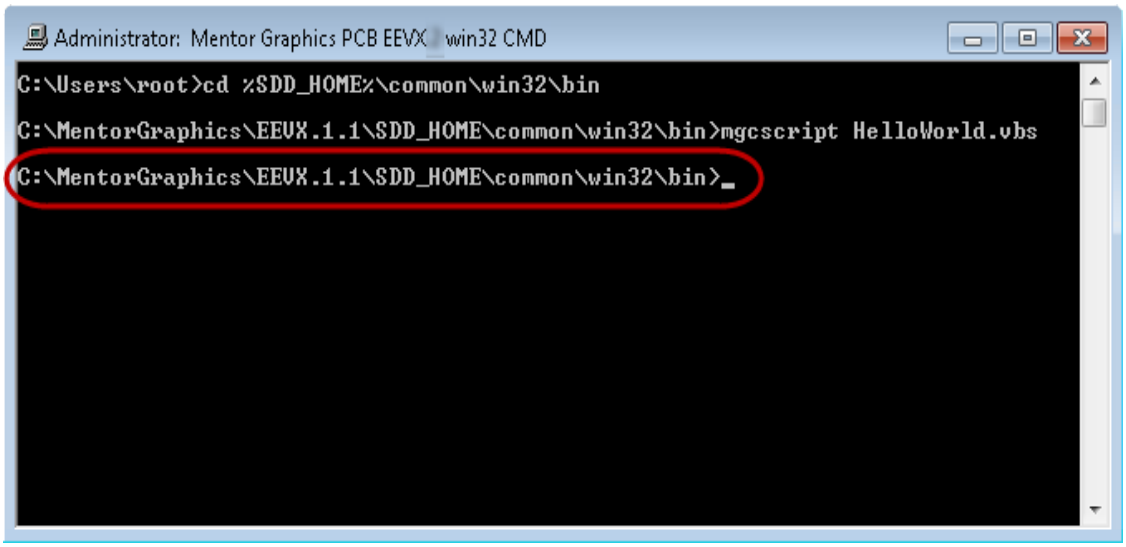
The reason for entering *mgcscript* rather than *mgcscript.exe* is that it is common practice to omit the *.exe* extension when calling an executable program. Windows assumes an *.exe* or *.bat* file.

Note:
The terms "argument" and "parameter" are interchangeable. However, for this tutorial "parameter" refers to values passed to a function and "argument" refers to values passed to a script from the command line.

5. In Figure 67: Running the Hello World Script From the Command Line , observe that the command line prompt has not displayed. This is because the script is still running. Click **OK** to

dismiss the dialog and end the script. The command line prompt displays as illustrated in Figure 68: Command Line Prompt Displays After the Script Finishes .

Figure 68. Command Line Prompt Displays After the Script Finishes



Displaying Command Line Arguments

This example presents a simple script that displays each command line argument in a dialog.

Creating the Script

Use the scripting editor of your choice to enter the following script and save it as DisplayArguments.vbs in the C:\temp folder:

```
 1  Option Explicit
 2
 3  Dim argumentColl
 4  Dim argumentObj
 5
 6  ' Get the collection of arguments
 7  Set argumentColl = ScriptHelper.Arguments
 8
 9  ' Iterate through arguments and display them
10  For Each argumentObj In argumentColl
11      MsgBox argumentObj
12  Next
```

Line 3 declares the variable argumentColl, which contains a collection of arguments. Line 4 declares the variable argumentObj, which holds a single argument object.

The ScriptHelper object is an explicit scripting object that is specific to the mgcscript.exe scripting engine. Arguments is a read-only property of the ScriptHelper object. Line 7 uses this property to assign the collection of command line arguments to the argumentColl variable.

Finally, Lines 10 through 12 use the For Each … In … Next control structure to display each argument in the collection. (Recall that For Each … In … Next is a special control structure that processes collections of objects. This structure is discussed in "For Each … In … Next" on page 42.)

Running the Script

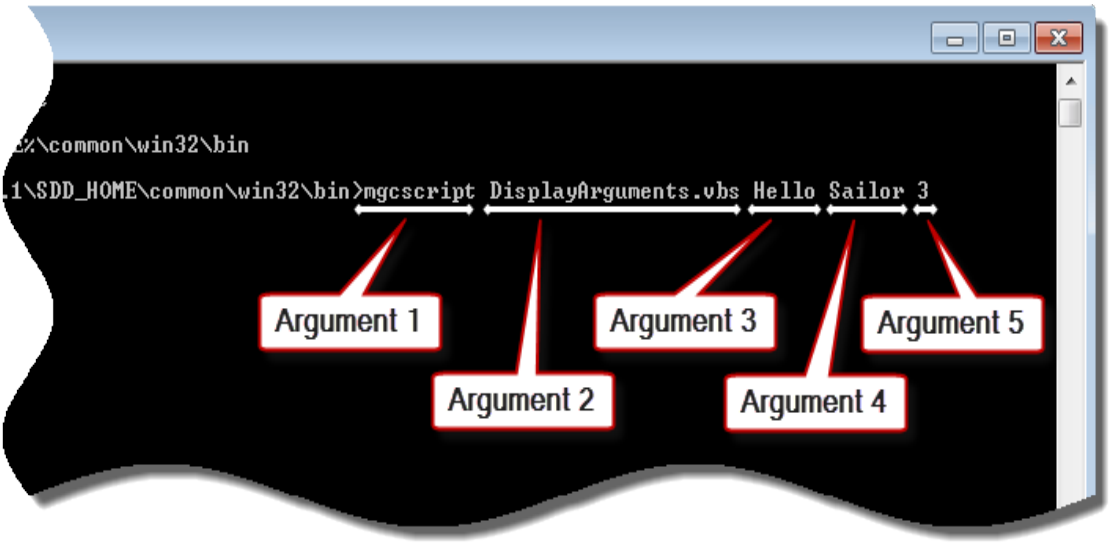
1. Enter mgcscript DisplayArguments.vbs Hello Sailor 3 into the command line as illustrated in Figure 69: Passing Arguments to a Script .

   Developers who perform initial tests on a system often create a simple Hello World program, whose sole purpose is to present the text "Hello World" to some display device. A regional equivalent to the Hello World program displays the message "Hello Sailor!" (this is supposed to have originated at SAIL, which stands for the Stanford Artificial Intelligence Lab).

   The call to the script engine, "mgcscript", and the name of the script, "DisplayArguments.vbs", are arguments on the command line. Therefore the entire command line, "mgcscript DisplayArguments.vbs Hello Sailor 3", has five arguments as illustrated in Figure 69: Passing Arguments to a Script .

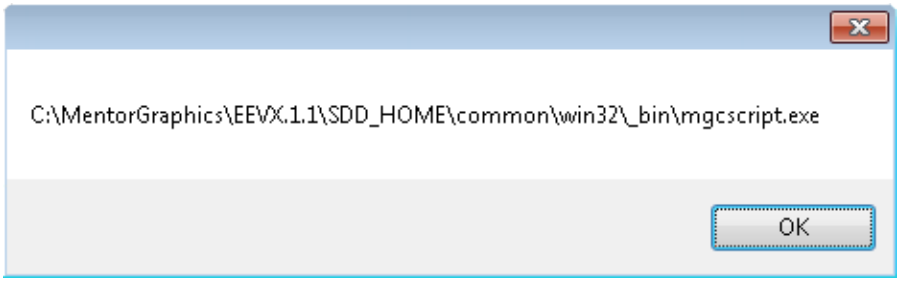
   The command line interprets each argument as a string. For example, the last argument '3' is a string, not an integer.

Figure 69. Passing Arguments to a Script



2. Press Enter to run this command and the DisplayArguments.vbs script. The first argument displays in a dialog as illustrated in Figure 70: Displaying the First Argument . Note that the full path name for this executable displays in the dialog.

Figure 70. Displaying the First Argument



3. Click **OK** to dismiss the dialog. The script now displays the second argument as illustrated in Figure 71: Displaying the Second Argument .

Figure 71. Displaying the Second Argument

4. Click **OK** to dismiss the dialog. The script now displays the third argument as illustrated in Figure 72: Displaying the Third Argument .

Figure 72. Displaying the Third Argument

5. Click **OK** to dismiss the dialog. The script now displays the fourth argument as illustrated in Figure 73: Displaying the Fourth Argument .

Figure 73. Displaying the Fourth Argument

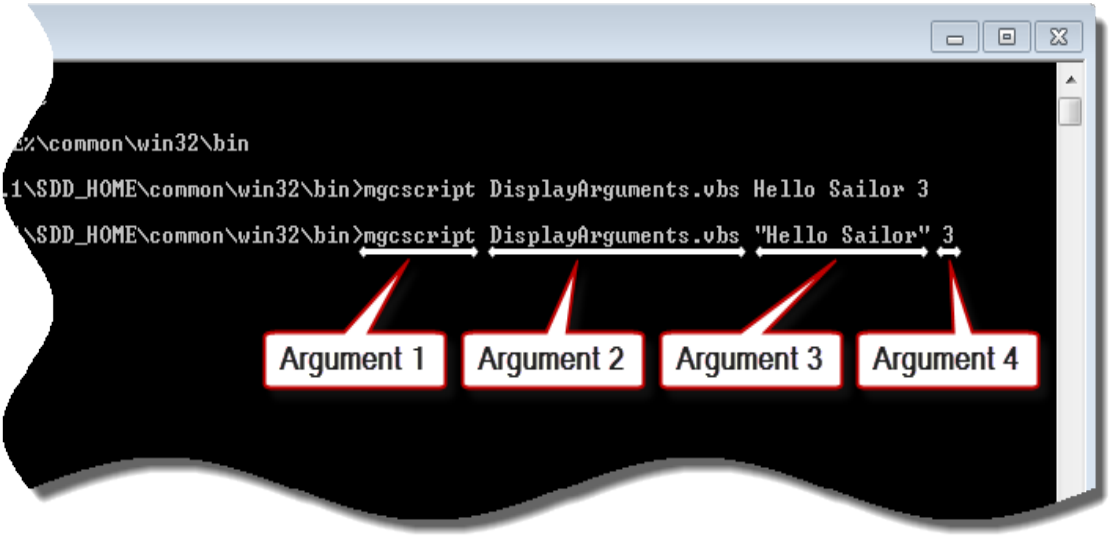
6. Click **OK** to dismiss the dialog. The script now displays the fifth argument as illustrated in Figure 74: Displaying the Fifth Argument .

Figure 74. Displaying the Fifth Argument

7. Click **OK** to dismiss the dialog. The script finishes.

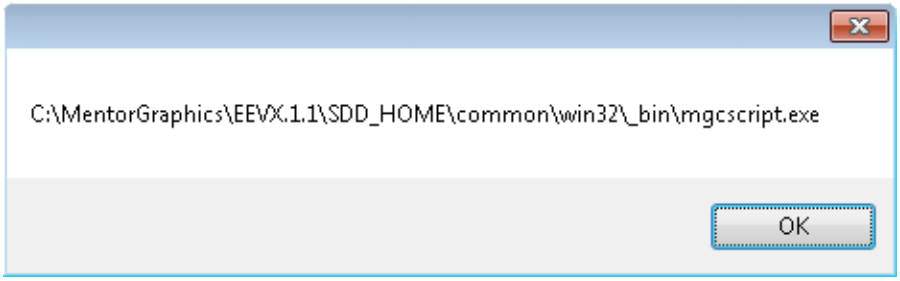
   In the previous steps, the system recognizes the words “Hello” and “Sailor” as two separate arguments. The words are delimited by a space. To treat the two words as a single argument, you must use double quotes as illustrated in Figure 75: Using Double Quotes to Create Single Arguments .

Figure 75. Using Double Quotes to Create Single Arguments



8. Press Enter to run this command and run the DisplayArguments.vbs script with a new set of arguments. The first argument displays in a dialog as illustrated in Figure 76: First Argument Displays.

Figure 76. First Argument Displays



9. Click **OK** to dismiss the dialog. The script now displays the second argument as illustrated in Figure 77: Second Argument Displays .

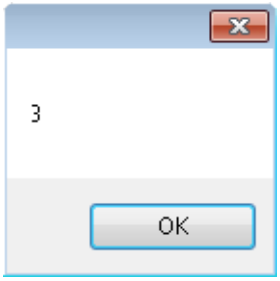
Figure 77. Second Argument Displays

10. Click **OK** to dismiss the dialog. The script now displays the third argument as illustrated in Figure 78: Third Argument Displays .

Figure 78. Third Argument Displays

11. Click **OK** to dismiss the dialog. The script now displays the fourth argument as illustrated in Figure 79: Fourth Argument Displays .

Figure 79. Fourth Argument Displays



12. Click **OK** to dismiss the dialog. The script finishes.

Using Command Line Arguments

This example presents a script that accepts two user-defined arguments; the first is a string, the second is an integer. The script displays the first argument, the number of times defined by the second argument.

Creating the Script

Use the scripting editor of your choice to enter the following script and save it as ParsingArguments.vbs in the C:\temp folder.

Line 1 declares Option Explicit. Line 3 declares the variable argumentColl, which contains the collection of arguments. Line 6 uses the Arguments property of the ScriptHelper object to assign the collection of command line arguments to the argumentColl variable.

```
 1  Option Explicit
 2
 3  Dim argumentColl
 4
 5  ' Get the collection of arguments
 6  Set argumentColl = ScriptHelper.Arguments
```

The script uses two user-defined arguments as follows:

```
mgcscript    ParsingArguments.vbs    "Hello Sailor"    3
(1)          (2)                     (3)              (4)
```

The two user-defined arguments are the strings “Hello Sailor “and “3” (used as an integer).

Line 9 uses the Count property on the collection or arguments to ensure that four arguments are present. If there are not four arguments, the statements on Lines 24 through 28 display an error message.

Line 11 instantiates a variable named textStr, and uses the Item property on the collection to assign the third argument, “Hello Sailor “, to this variable. Note that, the colon ":" character on Line 11 is a statement separator that allows multiple statements on a single line.

Line 12 instantiates a variable named rptInt (where rpt stands for "repeat") and uses the Item property on the collection to assign the fourth argument, 3, to this variable. VBScript automatically casts (converts) the string “3” to the integer 3 when Line 17 uses *rptInt*.

```
 8  ' Test for correct number of arguments
 9  If argumentColl.Count = 4 Then
10      ' Get 3rd and 4th arguments
11      Dim textStr: textStr = argumentColl.Item(3)
12      Dim rptInt:  rptInt  = argumentColl.Item(4)
```

Line 15 instantiates a variable named displayStr, which holds the display string. This string is initialized with a null value (an empty string). Generally, you do not need to initialize strings with null values; the reason for doing so is that displayStr concatenates with itself, requiring an initial value.

```
14      ' Loop to build up string to be displayed
15      Dim displayStr: displayStr = ""
```

Line 16 instantiates an iteration variable named i that controls the loop.

Line 17 starts a For … Next loop that iterates from 1 to rptInt. This is where VBScript automatically casts the string assigned to rptInt into an integer to control the loop. For each iteration, the displayStr string is concatenated with the third argument that is passed into the script, and with a "carriage-return line-feed" (newline) constant. The concatenated string is re-assigned to *displayStr*.

```
16      Dim i
17      For i = 1 To rptInt
18        displayStr = displayStr & textStr & vbCrLf
19      Next
```

Line 22 displays he resulting string in a MsgBox().

```
21      'Display string
22      MsgBox displayStr
```

Finally, Lines 26 and 27 define the error message to display if the end-user enters the incorrect number of arguments.

```
24  Else
25      ' Display an error
26      MsgBox "Usage: ParsingArguments.vbs " & _
27             "<string to display> <number to repeat>"
28  End If
```

Running the Script

Run the script in the Windows command line.

1. Enter the command as illustrated in Figure 80: Entering the ParsingArguments.vbs Script.

Figure 80. Entering the ParsingArguments.vbs Script

2. Press Enter to run the script. The output displays as illustrated in Figure 81: ParsingArgument.vbs Output.

Figure 81. ParsingArgument.vbs Output

3. Click **OK** to dismiss the dialog. The script finishes.

4. Test the script with too few and too many parameters.

Adding Additional Error Handling

In the script example Using Command Line Arguments, you include a simple error test to check that the correct number of arguments is present. To increase the quality of the script, the script should ensure that the last argument is an integer.

You do this by explicitly casting the argument to an integer before using it. For example, consider the original script:

```
Option Explicit

Dim argumentColl

' Get the collection of arguments
Set argumentColl = ScriptHelper.Arguments

' Test for correct number of arguments
If argumentColl.Count = 4 Then
    ' Get 3rd and 4th arguments
    Dim textStr: textStr = argumentColl.Item(3)
    Dim rptInt:  rptInt  = argumentColl.Item(4)

    ' Loop to build up string to be displayed
    Dim displayStr: displayStr = ""
    Dim i
    For i = 1 To rptInt
      displayStr = displayStr & textStr & vbCrLf
    Next

    'Display string
    MsgBox displayStr

Else
```

```
25      ' Display an error
26      MsgBox "Usage: ParsingArguments.vbs " & _
27             "<string to display> <number to repeat>"
28  End If
```

There are two lines in which you can cast the argument as an integer before using it; either on Line 12:

```
12      Dim rptInt:  rptInt  = argumentColl.Item(4)
12      Dim rptInt:  rptInt  = CInt(argumentColl.Item(4))
```

Or, on Line 17:

```
17      For i = 1 To rptInt
17      For i = 1 To CInt(rptInt)
```

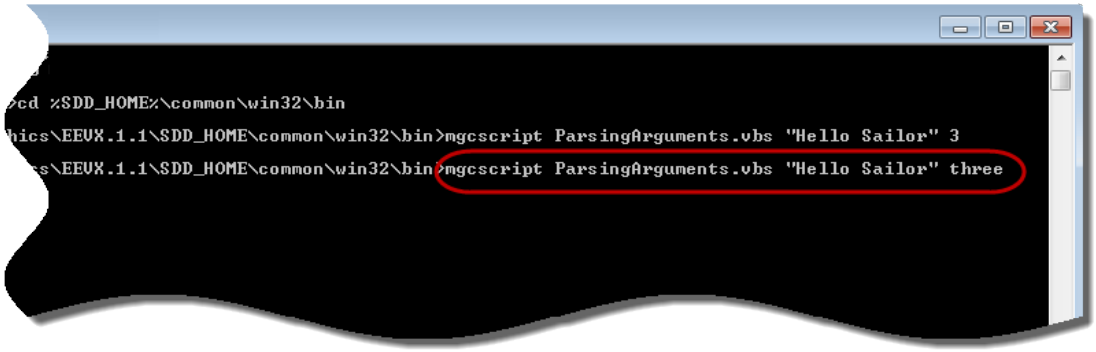
There are two reasons for performing this explicit casting:

1. **Debugging** — When debugging code it makes it clear what the script is doing; that is, the script is casting and using this argument as an integer.

2. **Messaging** — The error message the system displays contains better information.

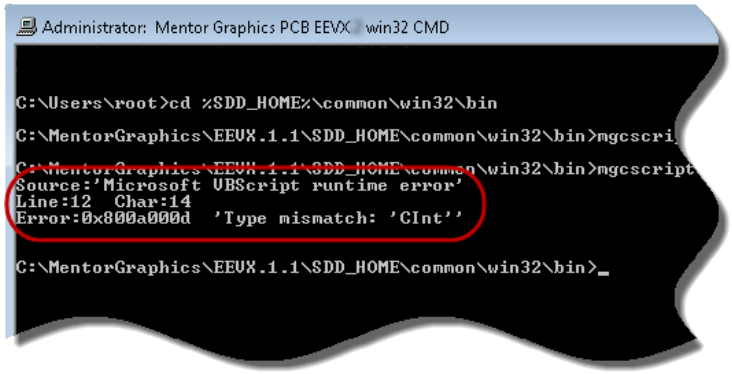
Assume the modification on Line 12 as shown previously. And assume the argument is “three” instead of “3” as illustrated in Figure 82: Changing the Last Argument to 'three'.

Figure 82. Changing the Last Argument to 'three'



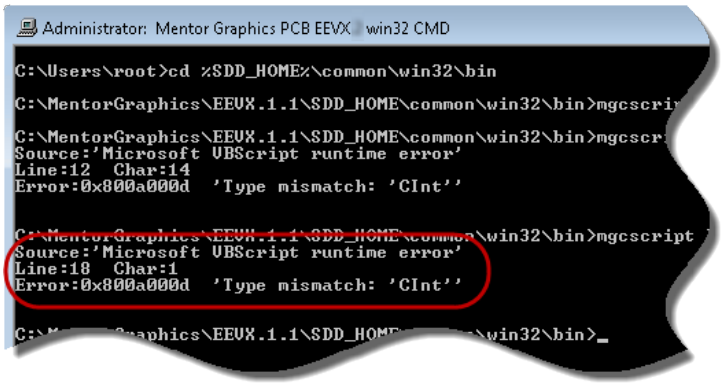
In this example, the system responds with the error message shown in Figure 83: 'CInt' Part of the Error Message.

Figure 83. 'CInt' Part of the Error Message



The error message indicates that the script expects an integer argument (or at least, a string that could be cast as an integer). If there is no explicit casting, then the error that occurs is illustrated in Figure 84: Less Helpful Error Message.

Figure 84. Less Helpful Error Message



This is slightly less helpful, because the error message indicates that the script is not expecting to process a string, but it does not indicate what the script is expecting.

The best way to handle this is to combine the explicit CInt cast technique on Line 12 with an On Error Resume Next statement (this statement is discussed in "VBScript Primer" on page 27). In the case of an error, this allows the script to use the Err object to determine the type of error and proceed accordingly (note that the Err object provides both the error number and description).

Working With Applications and Documents

Suppose you create a script that requires access to an application and a document. What happens if the application is not running? What happens if the application is running but a document is not opened?

There are three cases to consider as follows:

Application	Document
Open	Open
Open	Closed
Closed	Closed

Attaching to an Existing Application—With an Open Document
Attaching to an Existing Application—With No Open Document
Creating an Invisible Application
Creating a Visible Application

Attaching to an Existing Application—With an Open Document

(Application Open, Document Open)

This example attaches to an existing application that has an open document.

Creating the Script

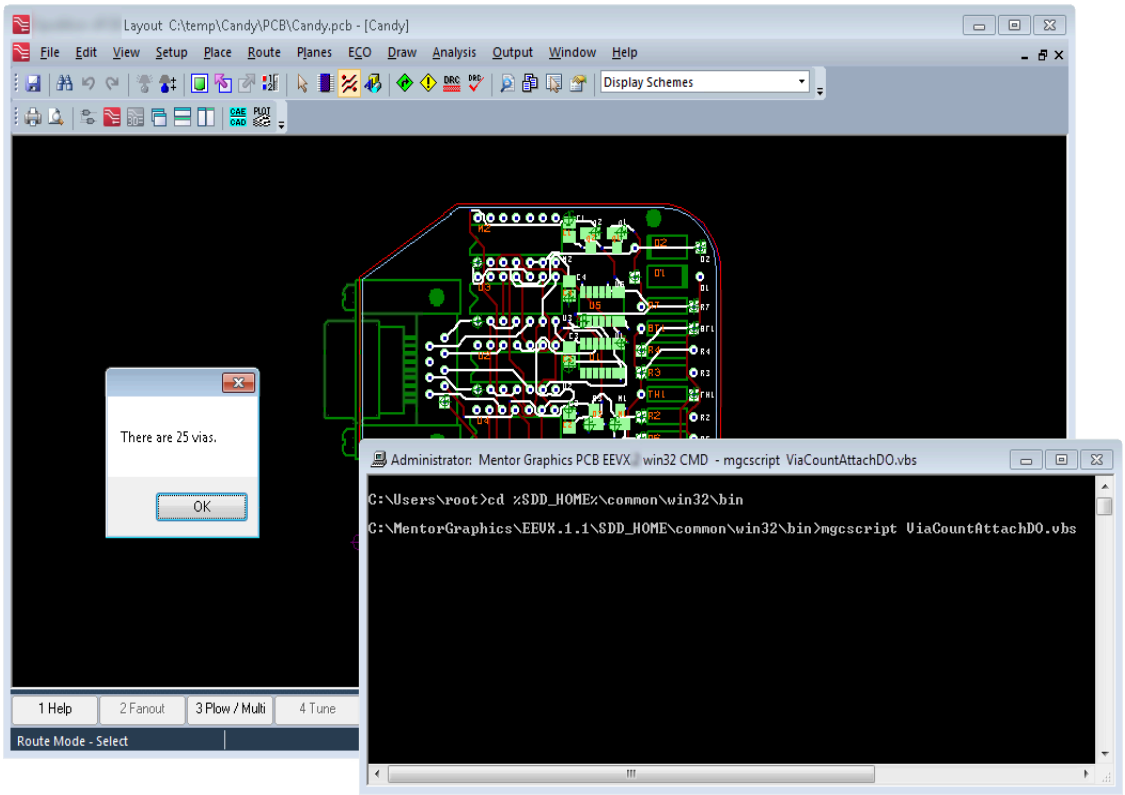
Use the scripting editor of your choice to open the *ViaCount.vbs* script (created in Running Your First Script) located in the *C:\temp* folder. Save the script as *ViaCountAttachDO.vbs*, where the "DO" portion of this filename stands for "Document Open". The following illustrates the initial script:

```

Option Explicit

' Add any type libraries to be used.
Scripting.AddTypeLibrary("MGCPCB.ExpeditionPCBApplication")

' Get the Application object
Dim pcbAppObj
Set pcbAppObj = Application

' Get the active document
Dim pcbDocObj
Set pcbDocObj = pcbAppObj.ActiveDocument

' License the document
ValidateServer(pcbDocObj)
```

```
17
18  ' Get the vias collection
19  Dim viaColl
20  Set viaColl = pcbDocObj.Vias
21
22  ' Get the number of vias in collection
23  Dim countInt
24  countInt = viaColl.Count
25
26  MsgBox("There are " & countInt & " vias.")
27
28  ''''''
```

4. Run the *ViaCountAttachDO.vbs* script from the command line and observe that the script attaches to the currently running instantiation of Xpedition Layout ("Running Scripts from an Application" on page 161).

Figure 85. Attaching to a Running Application with an Open Design



Attaching to an Existing Application—With No Open Document

(Application Open, Document Closed)

This example attaches to an existing application that has no document open.

Creating the Script

Use the scripting editor of your choice to open the *ViaCountAttachDO.vbs* file and save it as *ViaCountAttachDC.vbs* in the *C:\temp* folder. The "DC" portion of this filename stands for "Document Closed". The following shows the initial script:

```

Option Explicit

' Add any type libraries to be used.
Scripting.AddTypeLibrary("MGCPCB.ExpeditionPCBApplication")

```

```
 7  ' Get the Application object
 8  Dim pcbAppObj
 9  'Set pcbAppObj = Application
10  Set pcbAppObj = GetObject(,"MGCPCB.ExpeditionPCBApplication")
11
12  ' Get the active document
13  Dim pcbDocObj
14  Set pcbDocObj = pcbAppObj.ActiveDocument
15
16  ' License the document
17  ValidateServer(pcbDocObj)
18
19  ' Get the vias collection
20  Dim viaColl
21  Set viaColl = pcbDocObj.Vias
22
23  ' Get the number of vias in collection
24  Dim countInt
25  countInt = viaColl.Count
26
27  MsgBox("There are " & countInt & " vias.")
28
29  ''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
30  'Local functions
31
32  ' Server validation function
33  Private Function ValidateServer(docObj)
       :
      etc.
```

The script in this section uses the layout design named *Candy.pcb* located in the *C:\temp\Candy\PCB* folder (refer to "Running Scripts from an Application" on page 161 for details regarding typical locations of such documents).

Comment out the original Line 14 and insert a new Line 15 as shown. Line 15 explicitly opens the *Candy.pcb* layout document. Save and close the file.

```
14  'Set pcbDocObj = pcbAppObj.ActiveDocument
15  Set pcbDocObj = pcbAppObj.OpenDocument("C:\temp\Candy\PCB\Candy.pcb")
```

Running the Script

Run the script from within Console window.

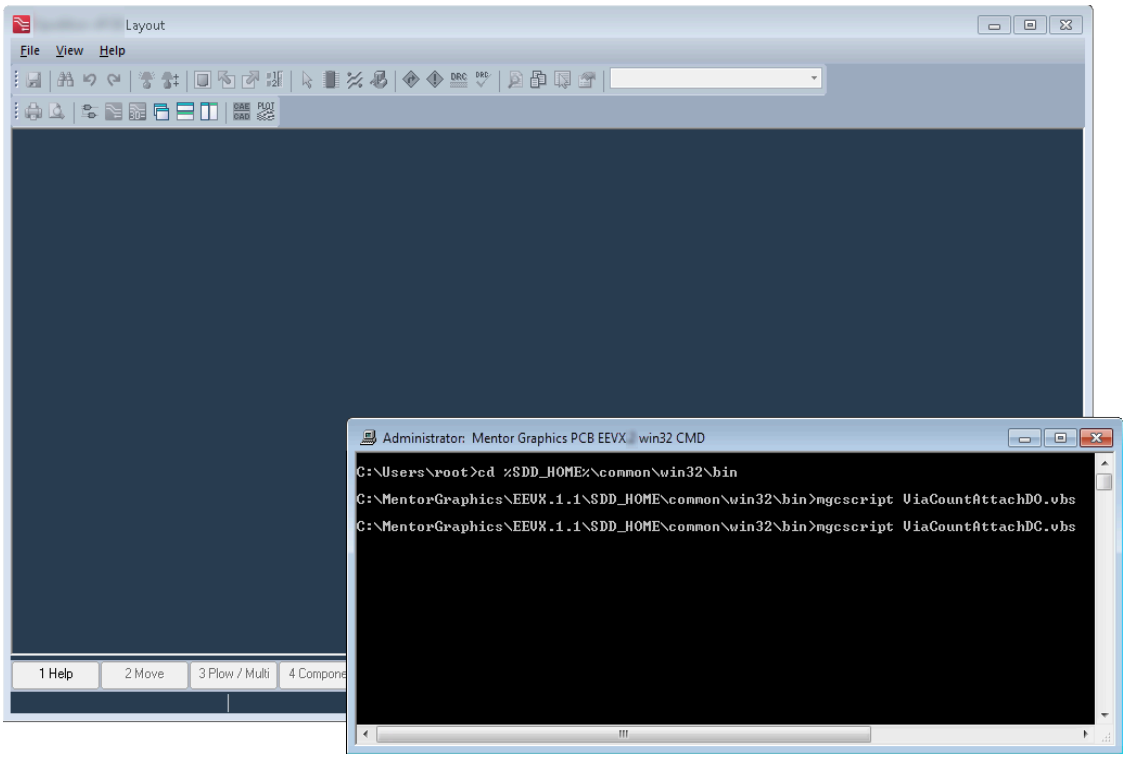
1. Close any instantiations of Xpedition Layout that are currently running, and then launch Xpedition Layout.

2. Do not open a layout design.

3. Open a Console window, set the working directory to *C:\temp.*

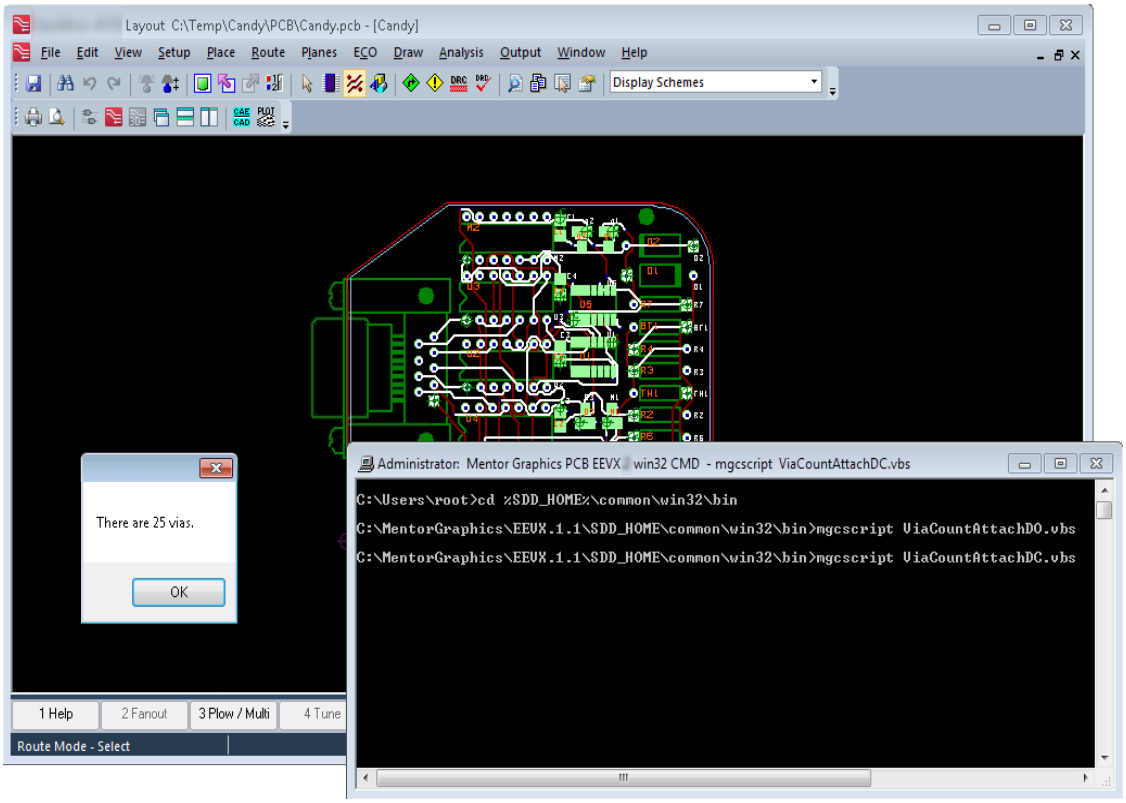
4. Enter, but do not run, the *ViaCountAttachDC.vbs* script as shown in Figure 86: Script Setup.

Figure 86. Script Setup



5. Press Enter to run the script. Observe that the script opens the *Candy.pcb* layout document and counts the vias as illustrated in Figure 87: Attaching to a Running Application and Opening a Design.

Figure 87. Attaching to a Running Application and Opening a Design



There is an easy way by which the script can test if a document is already open. Lines 14 and 15 in the script are currently:

```
12  ' Get the active document
13  Dim pcbDocObj
14  'Set pcbDocObj = pcbAppObj.ActiveDocument
15  Set pcbDocObj = pcbAppObj.OpenDocument("C:\temp\Candy\PCB\Candy.pcb")
```

Uncomment Line 14, which means that this statement accesses the active document if there is one. Modify Lines 15 through 17 as follows:

```
12  ' Get the active document
13  Dim pcbDocObj
14  Set pcbDocObj = pcbAppObj.ActiveDocument
15  If  pcbDocObj Is Nothing Then
16   Set pcbDocObj = pcbAppObj.OpenDocument("C:\temp\Candy\PCB\Candy.pcb")
17  End If
```

Note:
If you do perform these modifications, save the modified file with a different filename. The earlier version is reused in later exercises.

Line 15 now performs a test by comparing the Document object to the keyword Nothing. Nothing is a null object and is used for this type of comparison. (Nothing may also be used to pass a null object parameter into a function or subroutine.)

If the Document object does not exist, which means there is no currently active document, Line 16 explicitly opens the document.

As an alternative to explicitly “hard-coding” the document to open, the script can accept the name of the document as an argument. This is left as an exercise for the reader.

Creating an Invisible Application

This example creates an invisible application.

Creating the Script

Use the scripting editor of your choice to open the ViaCountAttachDC.vbs file you created in the previous exercise and save it as ViaCountCreateInvisible.vbs. The following illustrates the initial script:

```
 1
 2  Option Explicit
 3
 4  ' Add any type libraries to be used.
 5  Scripting.AddTypeLibrary("MGCPCB.ExpeditionPCBApplication")
 6
 7  ' Get the Application object
 8  Dim pcbAppObj
 9  'Set pcbAppObj = Application
10  Set pcbAppObj = GetObject(,"MGCPCB.ExpeditionPCBApplication")
11
12  ' Get the active document
13  Dim pcbDocObj
14  Set pcbDocObj = pcbAppObj.OpenDocument("C:\temp\Candy\PCB\Candy.pcb")
15
16  ' License the document
17  ValidateServer(pcbDocObj)
18
19  ' Get the vias collection
20  Dim viaColl
21  Set viaColl = pcbDocObj.Vias
22
23  ' Get the number of vias in collection
24  Dim countInt
25  countInt = viaColl.Count
26
27  MsgBox("There are " & countInt & " vias.")
28
29  ''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
```

```
30  'Local functions
31
32  ' Server validation function
33  Private Function ValidateServer(docObj)
       :
      etc.
```

Observe Line 10, the GetObject function attaches to an instantiation of the application that is already running, as the following illustrates:

```
10  Set pcbAppObj = GetObject(,"MGCPCB.ExpeditionPCBApplication")
```

Edit this line such that it now calls the CreateObject function to launch an new instantiation of the application as shown below:

```
10  Set pcbAppObj = CreateObject("MGCPCB.ExpeditionPCBApplication")
```

CAUTION:
The Prog ID is the *second* parameter to the *GetObject* function and the *first* parameter to the *CreateObject* function.

If there are one or more instantiations of the application running when the script calls the CreateObject function, an additional instantiation launches. To test if an instantiation of the application is already running before calling the CreateObject function, call the GetObject function in conjunction with the On Error Resume Next statement as discussed in “VBScript Primer” on page 27.

The change to Line 10 is the only modification to the script. However, there is something else to consider. By default, when a script calls the CreateObject function to launch an application, the application is not visible. This is what is preferred in many situations, for example during batch processing and analysis.

There is an issue to consider. When the script terminates, an invisible application continues to run in the background. Because an invisible application has no GUI, there is no easy way to close the application. One solution for closing the application is to use the operating system to "Kill" the application process (the actual mechanism behind this is dependent on the operating system). The preferred solution for closing the application is for the script to explicitly release any objects from the application and to then terminate the application. To release an object, the script must set the variable to which the object is assigned to Nothing.

Line 27 contains the last statement in the body of the script:

```
27  MsgBox("There are " & countInt & " vias.")
```

It is a good programming practice to close everything before displaying the result. To do this, insert the following lines before the MsgBox() statement:

```
27  ' Done with viaColl so release it.
28  Set viaColl = Nothing
29
30  ' Done with the document so close it.
31  pcbDocObj.Close
32  ' Done with pcbDocObj variable so release it.
33  Set pcbDocObj = Nothing
34
35  ' Done with the application so exit/quit.
```

```
36  pcbAppObj.Quit
37  ' Done with pcbAppObj variable so release it.
38  Set pcbAppObj = Nothing
39
40  MsgBox("There are " & countInt & " vias.")
```

Line 28 releases the viaColl variable by setting it to Nothing. Line 31 uses the Close method on the Document object to close the document (this is equivalent to **File > Close** in the GUI). After the document is closed, the script must still release the Document object by setting its variable (pcbDocObj) to Nothing, as seen on Line 33. Line 36 uses the Quit method on the Application object to close the application (this is equivalent to **File > Exit** in the GUI). Finally, after the application has exited, the script must release the Application object by setting its variable (pcbAppObj) to Nothing, as seen on Line 38.

After making the modifications, save and close the script.

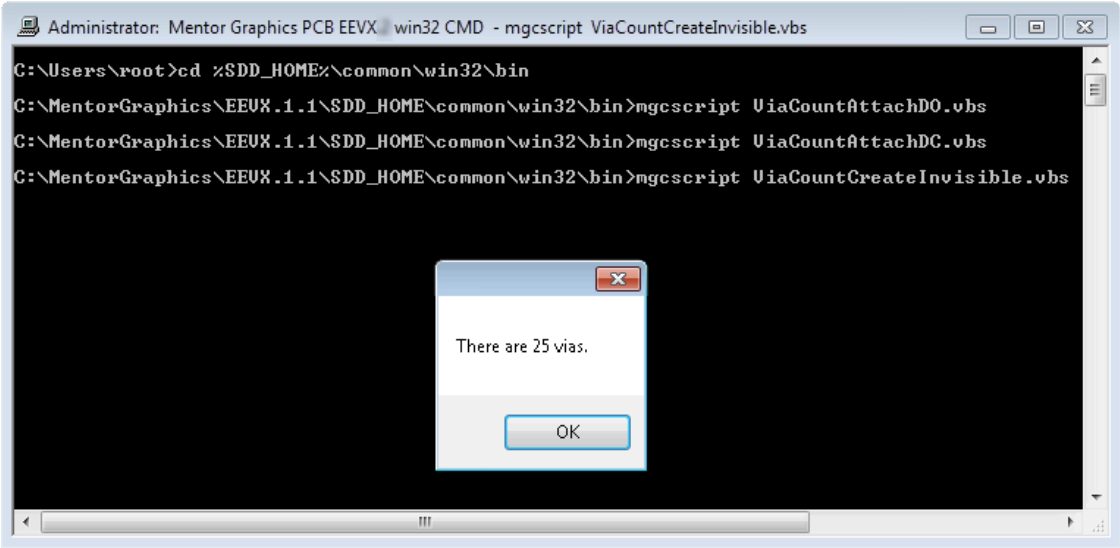
Running the Script

1. Ensure that no instantiations of Xpedition Layout are running.

2. Open a Console window and set the working directory to the *C:\temp* folder.

3. Run the *ViaCountCreateInvisible.vbs* script and observe the results as shown in Figure 88: Running a Script that Automatically Launches the Application.

Figure 88. Running a Script that Automatically Launches the Application



4. Observe that the Xpedition Layout GUI does not display, which is the expected behavior.

5. Click **OK** to dismiss the dialog. The script finishes.

Creating a Visible Application

This example creates a visible application.

Creating the Script

Use the scripting editor of your choice to open the *ViaCountCreateInvisible.vbs* file you created in the previous exercise and save it as *ViaCountCreateVisible.vbs*. The following shows the initial script (refer to the note following the script with regard to Lines 27 through 38):

```
 1
 2  Option Explicit
 3
 4  ' Add any type libraries to be used.
 5  Scripting.AddTypeLibrary("MGCPCB.ExpeditionPCBApplication")
 6
 7  ' Get the Application object
 8  Dim pcbAppObj
 9  'Set pcbAppObj = Application
10  Set pcbAppObj = CreateObject("MGCPCB.ExpeditionPCBApplication")
11
12  ' Get the active document
13  Dim pcbDocObj
14  Set pcbDocObj =
              pcbAppObj.OpenDocument("C:\temp\Candy\PCB\Candy.pcb")
15
16  ' License the document
17  ValidateServer(pcbDocObj)
18
19  ' Get the vias collection
20  Dim viaColl
21  Set viaColl = pcbDocObj.Vias
22
23  ' Get the number of vias in collection
24  Dim countInt
25  countInt = viaColl.Count
26
27  ' Done with viaColl so release it.
28  Set viaColl = Nothing
29
30  ' Done with the document so close it.
31  pcbDocObj.Close
32  ' Done with pcbDocObj variable so release it.
33  Set pcbDocObj = Nothing
34
35  ' Done with the application so exit/quit.
36  pcbAppObj.Quit
37  ' Done with pcbAppObj variable so release it.
38  Set pcbAppObj = Nothing
39
40  MsgBox("There are " & countInt & " vias.")
```

```
41
42  '''''''
```

Chapter 10
Running Scripts from an Application

This section shows you how to create and modify the WDIR environment variable, which facilitates multiple ways to run a script.

The common ways to run a script include:

- Using the key-in command field, (this is described in “Running Your First Script” on page 59).

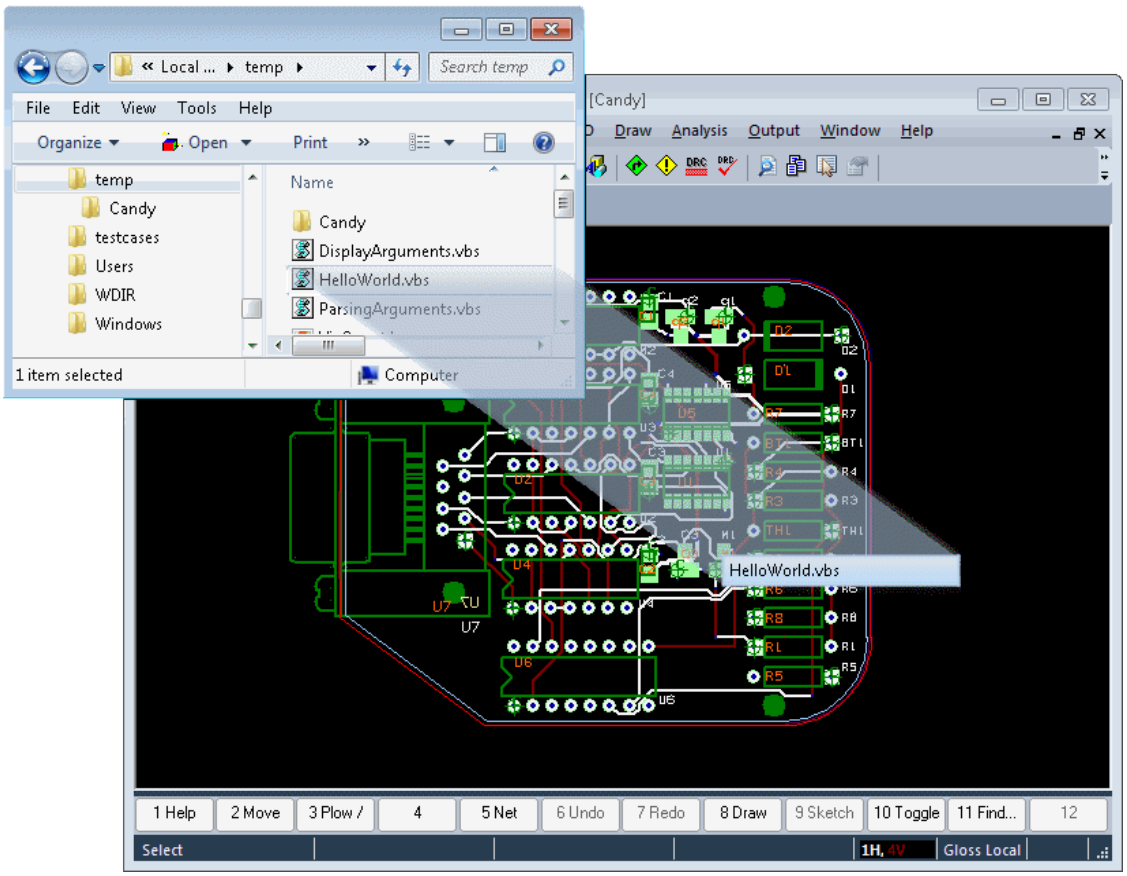
  Such a script can run without an open design (a layout design in the case of Xpedition Layout) and the script itself can open the design (this concept is discussed in “Running Scripts from the Command Line” on page 133).

- Tying a script to a menu item or to a toolbar button (these techniques were presented in “Adding Menus and Menu Items” on page 107 and “Adding Toolbar Buttons” on page 127, respectively).

- Automatically running a script when the application launches or when a document opens (these startup scripts are discussed in “Running Startup Scripts” on page 171).

- Using the mouse to drag-and-drop a script into an already open document as illustrated in the following figure.

Figure 89. Running a Script Using Drag-and-Drop



Using the WDIR Environment Variable

Creating and Modifying the WDIR Environment Variable

Using the WDIR Environment Variable

In previous examples, you save scripts in the *C:\temp* folder. This requires specifying the entire path, including the folder, when running the scripts.

When the full path is not included as part of a script's filename, the Mentor Graphics applications automatically check for the file in default folders in the following order:

1. **Product Area —** This is defined by the SDD_HOME environment variable and by the following path based on this variable: %SDD_HOME%\standard\automation\startup (refer to "Running Startup Scripts" on page 171).

2. **Design Area —** For Xpedition Layout, this is the folder containing the currently open **.pcb* layout design. If no document is open, then the application does not have this context and this option is excluded from the search.

3. **User-Defined Areas —** These are folders specified using the WDIR environment variable.

The WDIR Environment variable allows you to specify multiple folders to search. Separate each folder with semi-colons (;) when running on the Windows operating system and with colons (:) when running on UNIX and Linux.

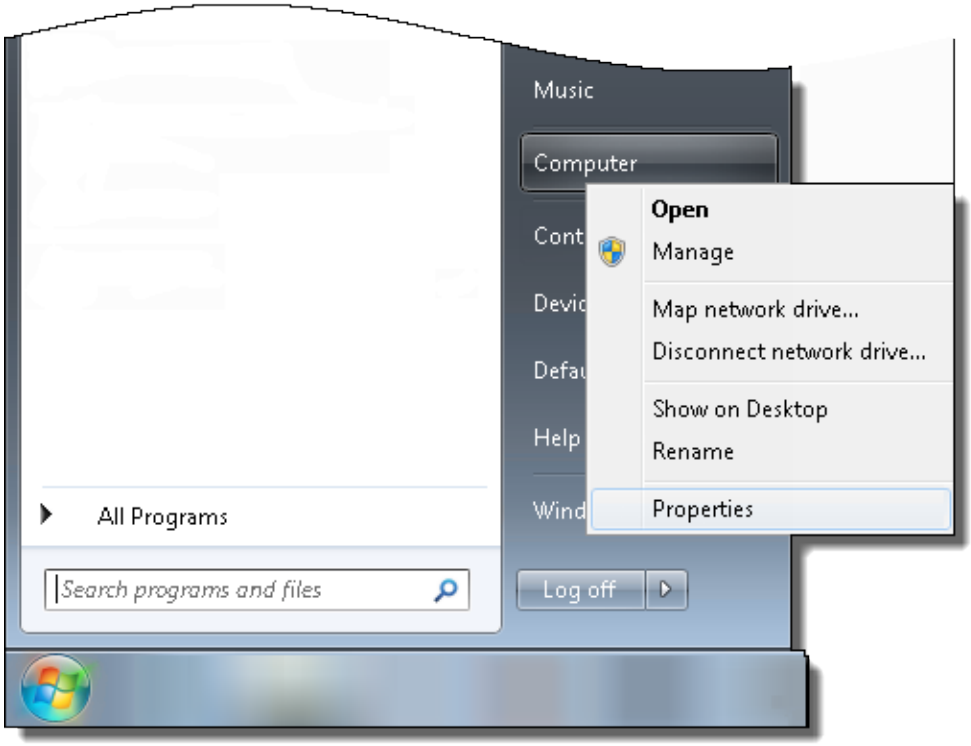
When the system finds a script with the correct name in any of the listed folders, the search ends and that script runs.

Creating and Modifying the WDIR Environment Variable

This exercise assumes you are running the Windows operating system.

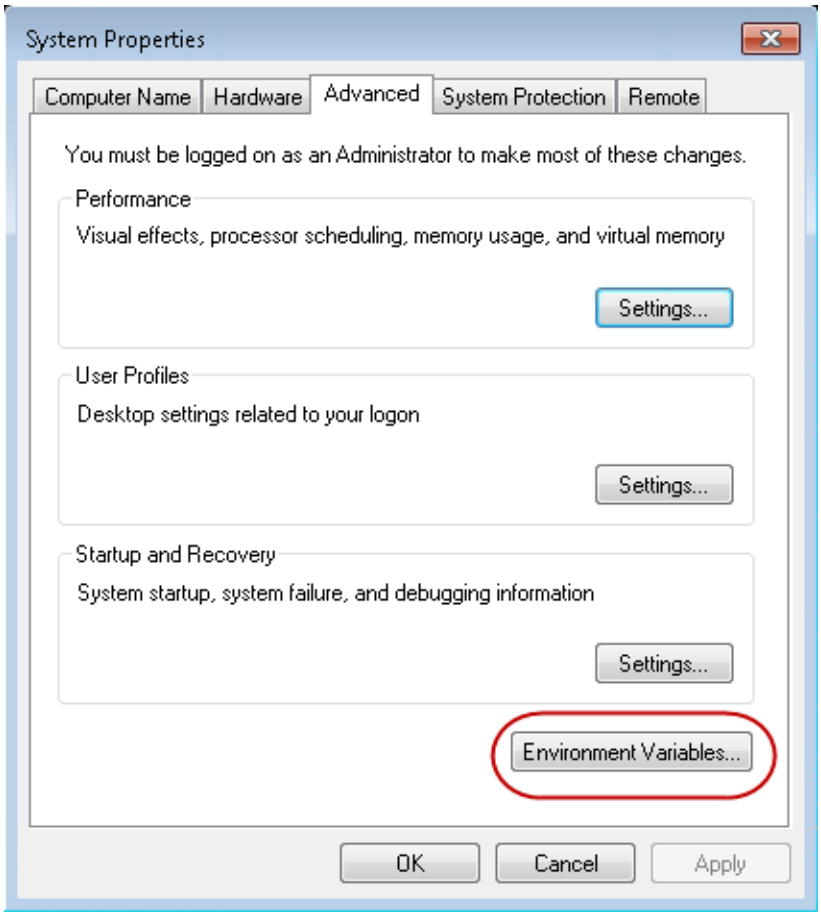
1. Right-click **My Computer**, and choose **Properties** from the resulting pop-up menu as illustrated in Figure 90: Accessing the System Properties Dialog .

Figure 90. Accessing the System Properties Dialog



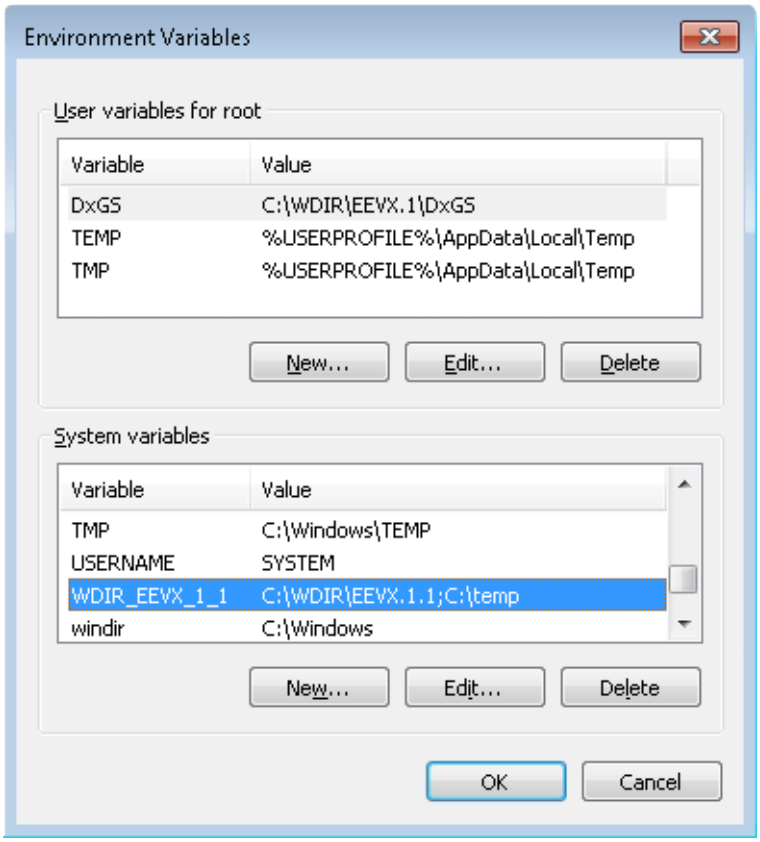
2. Observe the resulting **System Properties** dialog as illustrated in Figure 91: System Properties Dialog .

Figure 91. System Properties Dialog



3. Select the **Advanced tab** and click **Environment Variables** to access the **Environment Variables** dialog as illustrated in Figure 92: Environment Variables Dialog .

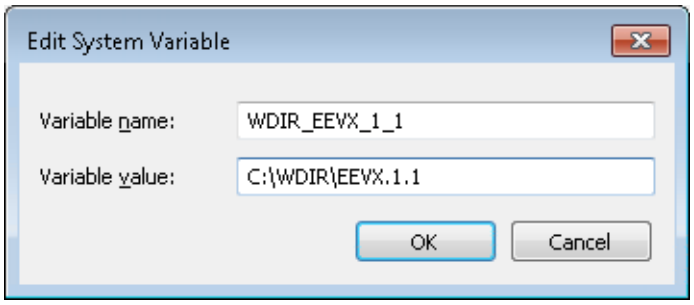
Figure 92. Environment Variables Dialog



Note:
Multiple side-by-side releases are supported on Windows. A uniquely named WDIR environment variable is created for each release. The name of the environment variable is based on the root name of the release, for example: WDIR_EEVX_2_7, WDIR_PADSVX_2_6, and so on.

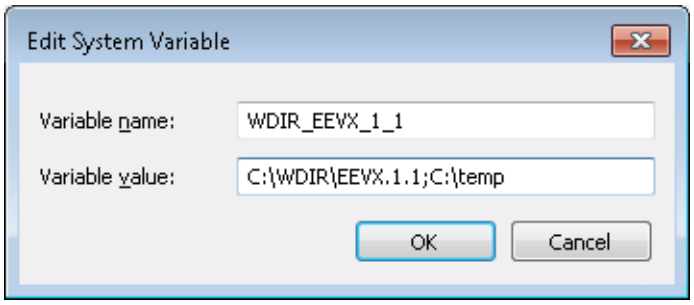
4. If an application's installation has already configured a WDIR variable, it is located in the **System Variables** list as illustrated in Figure 92: Environment Variables Dialog . To add a new folder, such as the *C:\temp* folder, to an existing WDIR variable, select the WDIR item and click **Edit**. The **Edit System Variable** dialog displays as illustrated in Figure 93: Edit System Variable Dialog . If no WDIR variable exists, proceed to Step 6.

Figure 93. Edit System Variable Dialog



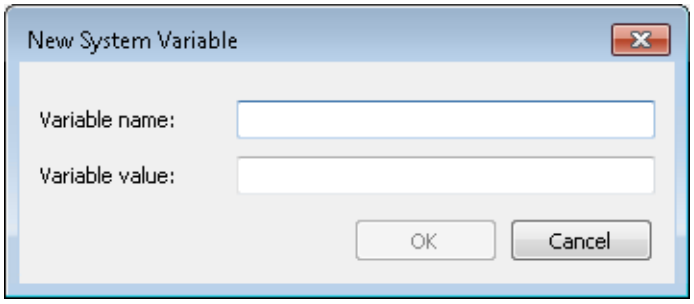
5. Add the new path(s) to the **Variable value** field as illustrated in Figure 94: Editing an Existing WDIR Variable . Proceed to Step 8.

Figure 94. Editing an Existing WDIR Variable



6. If there is no WDIR variable, then click **New** on the **Environment Variables** dialog. This returns the **New System Variable** dialog (Figure 95: New System Variable Dialog).

Figure 95. New System Variable Dialog



7. To add a new system variable, enter WDIR_<unique-identifier> in the **Variable name** field and enter the path(s) in the **Variable value** field as illustrated in Figure 96: Creating a New WDIR Variable .

Figure 96. Creating a New WDIR Variable

8. Click **OK** to exit the **Edit System Variable** dialog (or the **New System Variable** dialog); click **OK** to exit the **Environment Variables** dialog; and finally click **OK** to exit the **System Properties** dialog and accept the variable changes.

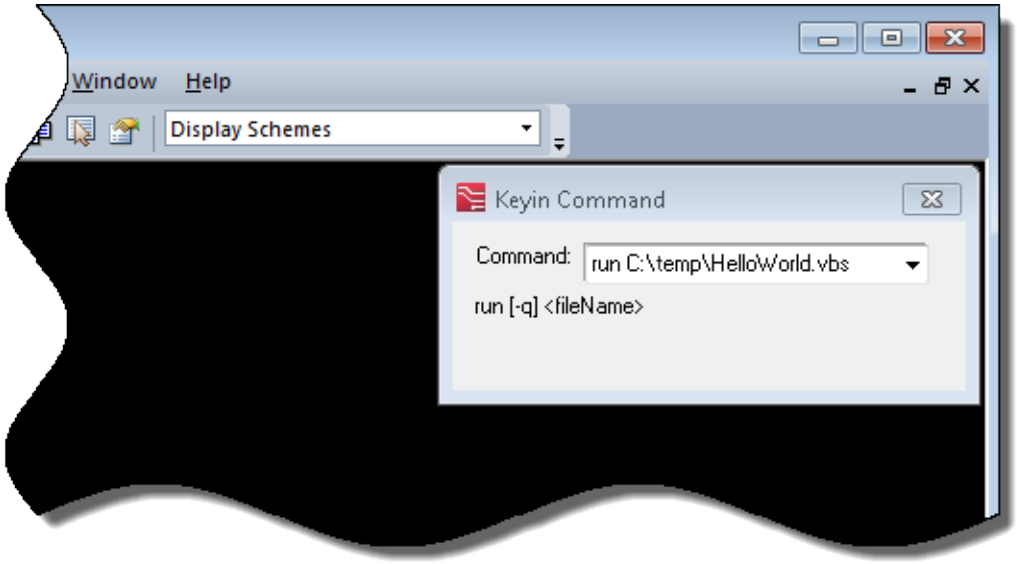
9. Close any currently open instantiations of Xpedition Layout.

10. Launch a new instantiation of Xpedition Layout.

11. Right-click on the toolbar and ensure that the **Keyin Command** option is enabled.

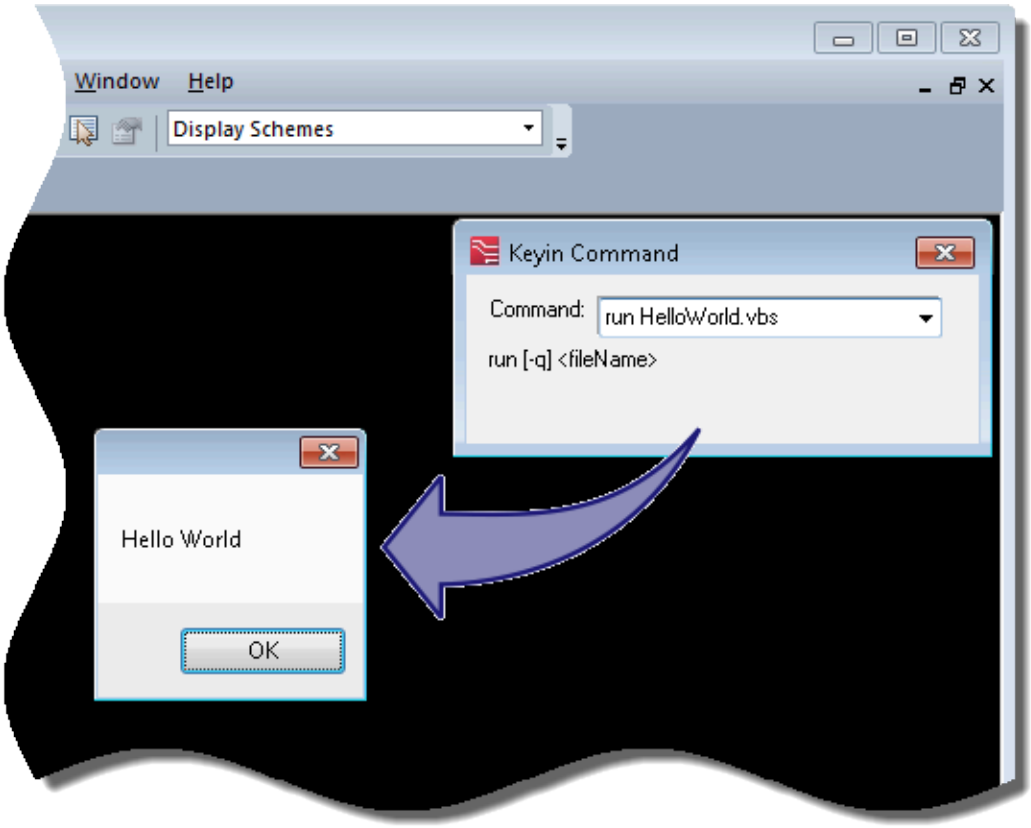
12. In "Running Your First Script" on page 59 you enter the string run into the **Keyin Command** field that runs the *HelloWorld.vbs* script (Figure 97: Specifying the Complete Path).

Figure 97. Specifying the Complete Path



13. Now that you have added the *C:\temp* folder to the WDIR environment variable, you can enter the string run *HelloWorld.vbs* into the **Keyin Command** field. The results is shown in Figure 98: Specifying the Relative Path

Figure 98. Specifying the Relative Path



In addition to absolute paths, you can specify relative paths with regard to any folder defined by the WDIR environment variable. For example:

- Assume that the WDIR environment variable includes *C:\temp*.
- A script named *MyTest.vbs* exists in the Test folder, which exists in the *C:\temp* folder. In this case, entering either of the following strings into the **Keyin Command** field runs the *MyTest.vbs* script:

```
run C:\temp\Test\MyTest.vbs     { Absolute }
run \Test\MyTest.vbs            { Relative }
```

You can also use environment variables as part of the key-in path; for example:

```
run %HOME%\MyTest.vbs
```

This assumes you are running in the Windows operating system; the equivalent when running in UNIX or Linux is:

```
run $HOME/MyTest.vbs
```

Chapter 11
Running Startup Scripts

This section contains exercises that show how to create application and document (design) startup scripts, and how to debug startup scripts.

Startup scripts are scripts that automatically run when an application first launches or when a document opens.

Startup scripts that run when an application launches are referred to as *application startup scripts*. Scripts that run when a design opens are referred to as *document startup scripts*. Usually, application scripts are what is required, however, this chapter describes both scenarios.

Using scripts.ini Files
Creating Application Startup Scripts
Exercise 1: Creating an Xpedition Layout Startup Script
Creating Document Startup Scripts
Debugging Startup Scripts

Using scripts.ini Files

A *scripts.ini* file specifies the scripts to run at startup. The file can contain multiple lists of scripts, where each list is associated with a particular application. For example, there can be one list of scripts associated with Xpedition Layout and another list of scripts associated with Xpedition FabLink.

There can also be multiple *scripts.ini* files. However, only one such file in each of the default folders. The system searches the default folders in the following order:

1. **Product Area —** This is defined by the SDD_HOME environment variable and by the following path based on this variable: *%SDD_HOME%\standard\automation\startup* (refer to Running Scripts from an Application).

   The startup folder is automatically created when you install Mentor Graphics products. This is also the preferred location for your scripts.ini file when the products are installed on a server and are accessible by multiple users, all of whom use the same startup scripts.

2. **Design Area —** For Xpedition Layout, this is the folder containing the currently open **.pcb* layout design. For Xpedition FabLink, this is the folder containing the **.pnl* design. There is a different design area for each design.

   The *scripts.ini* files stored in the design areas are scripts that run when a design opens. When the application is first launches, it does not process scripts.ini files in the design areas.

3. **User-Defined Areas —** These are folders specified using the WDIR environment variable. The process for creating this variable and modifying the folders associated with it is detailed in "Using the WDIR Environment Variable" on page 162.

This area is the most common and flexible option when for running personal startup scripts.

Note:
After an application locates and processes the first s*cripts.ini* file, the application continues to locate and process *scripts.ini* files in all of the remaining default folders. That is, the search does not stop once the first *scripts.ini* file is found.

Creating Application Startup Scripts

The format of the scripts.ini file for application startup scripts is as follows (where each script path includes the script's filename):

```
[<application name>]
Script#0=<script path>
Script#1=<script path>
Script#2=<script path>
   :
etc.
```

```
[<application name>]
Script#0=<script path>
Script#1=<script path>
Script#2=<script path>
   :
  etc.
```

A scripts.ini file can contain multiple lists of scripts, where each list is for a particular application. The application names to use in scripts.ini files are as follows:

Product	Key (name) to use in scripts.ini file
Xpedition Layout	Expedition PCB
PADS Pro Layout	Expedition PE
Board Station RE	Board Station RE
Board Station XE	Board Station XE
Xpedition Layout Viewer	Discovery PCB
Xpedition Layout Planner	Discovery PCB[1]
Xpedition FabLink	Fablink XE
SFX RE	SFX RE

1. The startup scripts for the Xpedition Layout Viewer and Xpedition Layout Planner must be specified under a single Discovery PCB application list in the scripts.ini file.

Product	Key (name) to use in scripts.ini file
Xpedition Designer	ViewDraw
EDM Library Tools	Library Manager
Cell Editor	Cell Editor

When an application launches, it searches for scripts.ini files. For each *scripts.ini* file, the application processes, in the listed order, the scripts associated with its own key. The order is important in certain cases; for example, when two scripts each add a new menu item to the "end" of the same pull-down menu. In this case, the relative positioning of the new items in the final menu depends on the order in which these scripts are listed and therefore processed.

Additionally, the scripts.ini parser (reader) automatically starts at Script#0 and proceeds until it detects a break in the sequence. Therefore, it is important that the first script to associate with Script#0, and that numbering remain sequential. A break commonly occurs if you delete an entry from the scripts.ini file, without renumbering the entries. Consider the following:

```
[<Application One>]
Script#1=<script path>            — Does not start with 0
Script#2=<script path>
Script#3=<script path>
   :
etc.
```

```
[<Application Two>]
Script#0=<script path>
Script#1=<script path>
Script#3=<script path>           — Skipped 2
    :
  etc.
```

In Application One, none of the scripts will run because this list does not start with Script#0. In Application Two, only Script#0 and Script#1 will run; because there is a break in the sequence after Script#1.

Exercise 1: Creating an Xpedition Layout Startup Script

Create an application startup script that automatically runs when Xpedition Layout first launches.

1. Create a folder named Startup under your *C:\temp* folder.

2. Use a text editor of your choice to open a new file and enter the following:

```
[Expedition PCB]
Script#0=C:\temp\ViaCountMenuSingle.vbs
```

2. This application has a slightly different format and search different product areas. Refer to the automation documentation for this product for details.

The *ViaCountMenuSingle.vbs* script is created in "Adding Menus and Menu Items" on page 107. The script adds a new **Via Count** menu item to the end of the **View** pull-down menu.

Observe that the full path for the script in entered in the file. If the *C:\temp* folder is included in the WDIR environment variable (as discussed in "Using the WDIR Environment Variable" on page 162), then you can enter the following:

```
[Expedition PCB]
Script#0=ViaCountMenuSingle.vbs
```

3. Save this file as *scripts.ini* in your new *C:\temp\Startup* folder.

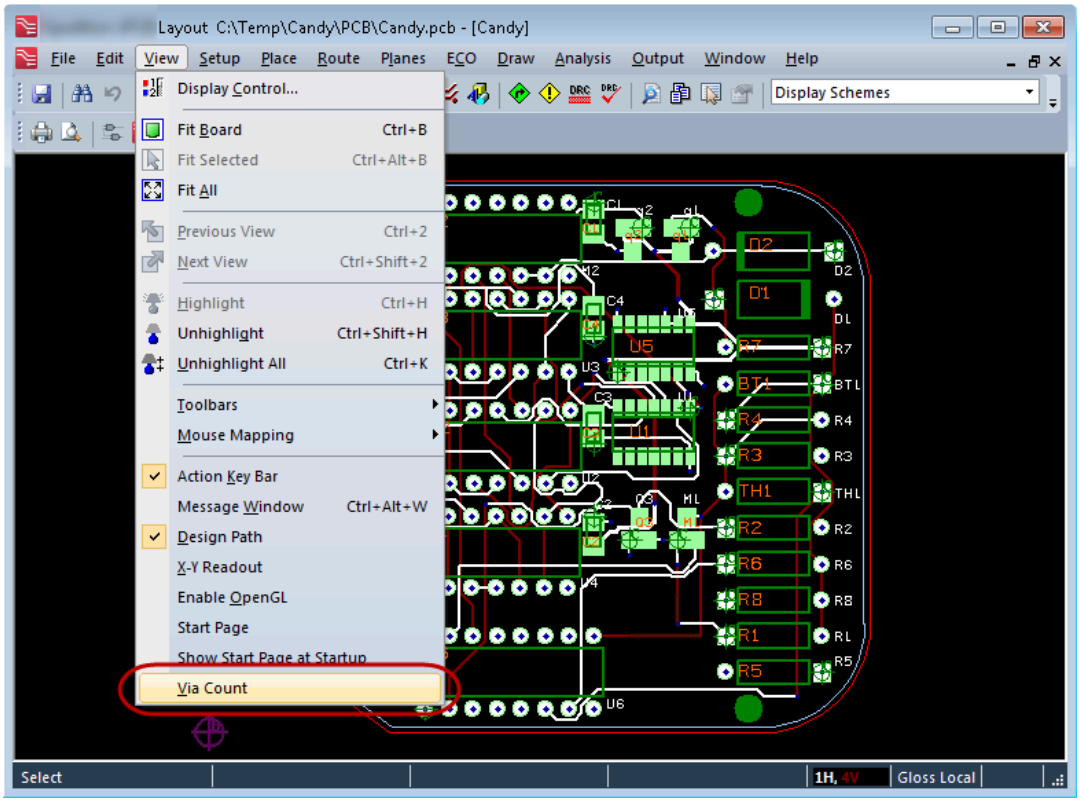
4. Edit the WDIR environment variable to add the new *C:\temp\Startup* folder to its list of folders (this is discussed in "Using the WDIR Environment Variable" on page 162).

5. Assure that no instantiations of Xpedition Layout are currently running, and then launch a new instantiation of Xpedition Layout.

6. Open a layout design of your choice.

7. Choose the **View** menu and observe that the script automatically added **Via Count** to the end of the menu as illustrated in Figure 99: Via Count Menu Item.

Figure 99. Via Count Menu Item



Creating Document Startup Scripts

Some startup scripts may run only when a design opens. In this case, you specify these scripts in a separate section of the scripts.ini file. These document startup scripts differ from the application startup scripts, by appending a " – Document" qualifier to the application name.

Exercise 2: Creating an Xpedition Layout Document Startup Script

Exercise 2: Creating an Xpedition Layout Document Startup Script

The following is a simple example of a document startup script. Assume that you want to run the original ViaCount.vbs script whenever a layout design opens in Xpedition Layout.

1. Use a text editor of your choice to open the C:\temp\Startup\scripts.ini file created in “Exercise 1” on page 173. This file is an application startup script and contains the following:

```
[Expedition PCB]
Script#0=C:\temp\ViaCountMenuSingle.vbs
```

2. Add a document startup script as follows:

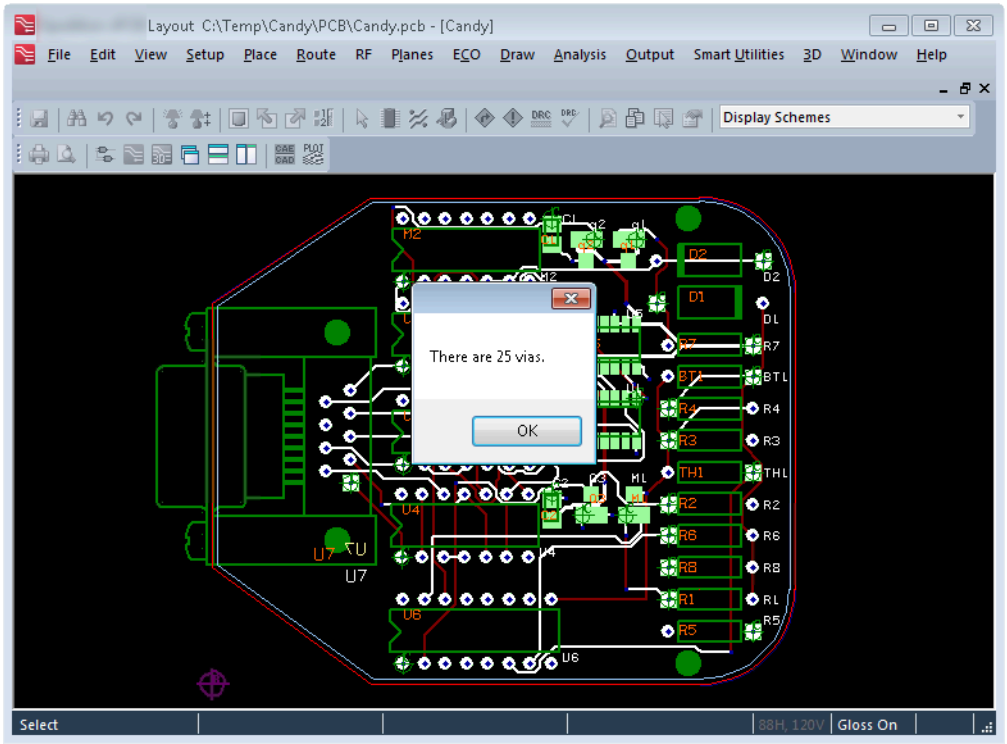
```
[Expedition PCB]
Script#0=C:\temp\ViaCountMenuSingle.vbs
[Expedition PCB - Document]
Script#0=C:\temp\ViaCount.vbs
```

3. Save and exit the file.

4. Assure no instantiations of Xpedition Layout are currently running, and then launch a new instantiation of Xpedition Layout.

5. Open a layout design of your choice and observe that the ViaCount.vbs script automatically runs when the design opens as illustrated in Figure 100: ViaCount.vbs Automatically Run When the Design Opens.

Figure 100. ViaCount.vbs Automatically Run When the Design Opens



6. Click **OK** to dismiss the dialog.

7. Choose **File > Close** to close the document and **File > Exit** to exit Xpedition Layout.

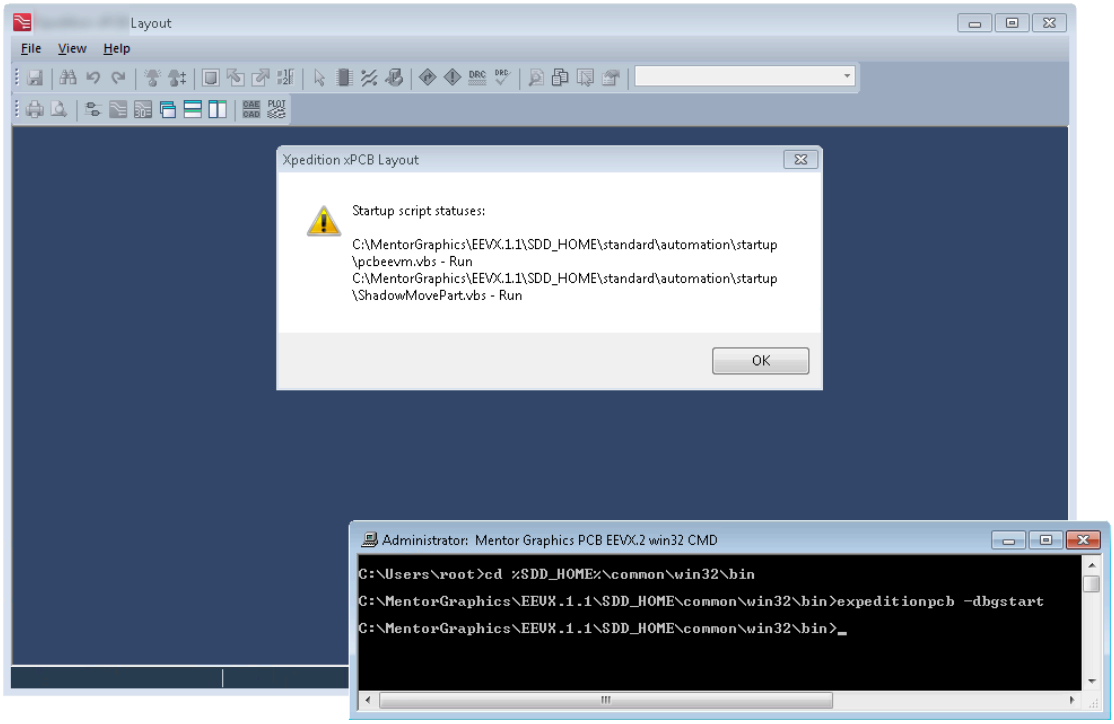
8. Rename the *scripts.ini* file in the *C:\temp\Startup* folder to scripts-(do-not-run).ini. If you do not do this, the *ViaCount.vbs* script automatically runs every time you open a layout design.

Debugging Startup Scripts

In some cases, you may want to debug your startup scripts by either (a) verifying a list of scripts run or (b) not running them.

To do this, launch Xpedition Layout from a Console window and specify the command line switches –**dbgstart** or –**nostart**, respectively (Figure 101: Debugging Startup Scripts).

Figure 101. Debugging Startup Scripts



Chapter 12
Introducing Events and the Scripting Object

This section introduces events and event handling functions. The exercises include handling events in scripts that run within the scope of the application and handling events in scripts that run out-of-process.

The *ViaCount.vbs* script (in all of its many variations) simply extracts information from the application and processes this information using methods and properties defined in the automation interface.

In some instances, the application should notify the scripts when specific occurrences take place; for example:

- When a document opens or closes (or prepares to close).
- When the application exits (or prepares to exit).
- When an object selection changes (the object is selected or unselected).
- When the design changes (moving or modifying objects).

Servers and clients perform notification using *events* and *event handlers*. Event handlers are functions in a script. The names of these event handler functions are generally predefined. The application calls these function when a specific event occurs.

The Scripting object has many functions, one of which is to facilitate script events.

This chapter introduces events and event handling functions. For more complex discussions and examples, refer to "Basic Building-Block Examples" on page 187.

Events and In-Process Scripts
Events and Out-of-Process Scripts Using mgcscript.exe

Events and In-Process Scripts

This exercise introduces handling events in scripts that run within the scope of the application.

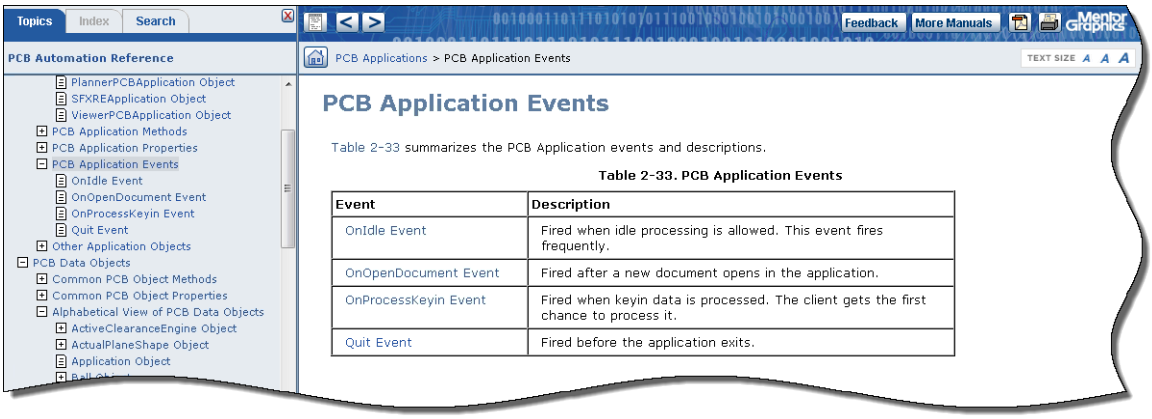
Creating the Script

Use the scripting editor of your choice to enter the following script and then save it in your *C:\temp* folder as *EventsInProcess.vbs*:

```

Option Explicit

' Add any type libraries to be used.
Scripting.AddTypeLibrary("MGCPCB.ExpeditionPCBApplication")

' Get the Application object
Dim pcbAppObj
Set pcbAppObj = GetObject(,"MGCPCB.ExpeditionPCBApplication")

```

```
11  ' Get the active document
12  Dim pcbDocObj
13  Set pcbDocObj = pcbAppObj.ActiveDocument
14
15  ' License the document
16  ValidateServer(pcbDocObj)
17
18  ' Attach events from the document object.
19  Call Scripting.AttachEvents(pcbDocObj, "pcbDocObj")
20
21  ' Set the Scripting.DontExit property to
22  ' true to keep the script running.
23  Scripting.DontExit = True
24
25  '''''''
```

Figure 102. Automation Help to Research Different Events



Consider Lines 18 and 19 as shown below:

```
18  ' Attach events from the document object.
19  Call Scripting.AttachEvents(pcbDocObj, "pcbDocObj")
```

The statement on Line 19 uses the AttachEvents method of the Scripting object to attach events of the document, *pcbDocObj,* to the script engine. The second parameter of the *AttachEvents* method is a string. This string is a common prefix for any of the event handler functions the script defines. The string itself is arbitrary such that you can name it anything. However, it is recommended that you set this string to be identical to the name of the first parameter (the variable that represents the object). This example uses "pcbDocObj" as the event handler prefix.

Consider Lines 21 through 23 as shown below:

```
21  ' Set the Scripting.DontExit property to
22  ' true to keep the script running.
23  Scripting.DontExit = True
```

Most scripts in this tutorial terminate after they run. This is a problem in this example, because if the script terminates then the script engine cannot call the event handler. Therefore, Line 23 sets the DontExit property of the Scripting object to the Boolean value of True. This means that this script continues running until the end of the current application session.

This example considers the event that occurs just before a document closes. This is known as the OnPreClose event.

Some events occur after an action occurs, in which case it is too late to do anything about that action. Other events, such as OnPreClose, occur just before an action occurs. *OnPre...* events allow the script to control whether the action occurs. Allowing and disallowing actions are controlled by the return values from event handlers. Events that cannot change behavior do not return a value. In this example, the script and end-user (using a message box) determine if the document closes.

Consider Lines 28 through 37 as shown below:

```
28  ' PreClose event handler
29  Function pcbDocObj_OnPreClose()
30      Dim ansEnum
31      ansEnum = MsgBox("Close this document?", vbYesNo)
32      If ansEnum = vbYes Then
```

```
33          pcbDocObj_OnPreClose = True
34      Else
35          pcbDocObj_OnPreClose = False
36      End If
37  End Function
```

The name of the event handler function on Line 29 is important and determines if this function fires. If either the name or the number of parameters differs in any way from the documentation, the application does not call the event handler function.

In this example, there are no parameters to pass into the function. The name of the function (pcbDocObj_OnPreClose) has several parts:

- The second parameter string specified in the *AttachEvents* method (this is "pcbDocObj" on Line 19).
- An underscore “_” character.
- The name of the event defined in the Automation Help documentation, which is *OnPreClose*.

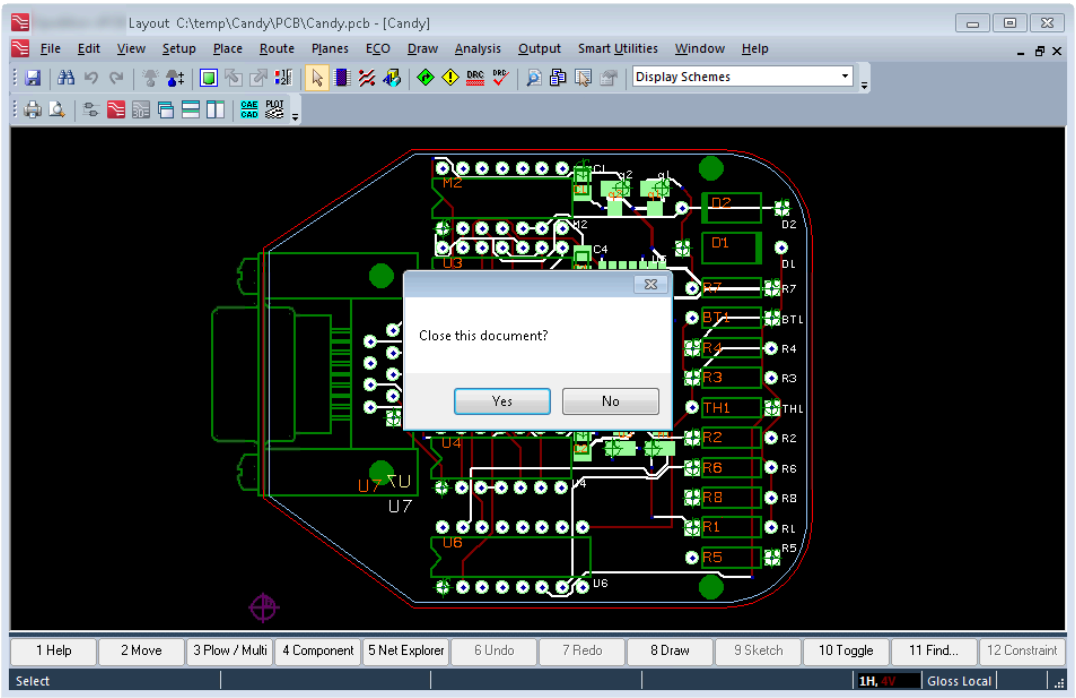
Line 30 instantiates a variable named ansEnum, which holds a Yes/No answer. Line 31 uses the MsgBox() function in a new mode. The second parameter to the *MsgBox()* function is set to the enumerated type vbYesNo. This is VBScript constant and results in a dialog containing two buttons. **Yes** —returns the enumerated type vbYes to the script. **No** —returns the enumerated type vbNo to the script.

The value returned from the MsgBox() function, vbYes or vbNo, is assigned to the ansEnum variable. Line 32 compares ansEnum to the enumerated type vbYes. If the end-user selects the **Yes** button, then the script allows the document to close returning a Boolean value of True. If the end-user selects **No**, then the script prevents the document from closing by returning a Boolean value of False.

Running the Script

1. Assure no instantiations of Xpedition Layout are currently running, and then launch a new instantiation of Xpedition Layout.
2. Open a layout design of your choice.
3. Drag-and-drop the EventsInProcess.vbs script from the C:\temp folder on the layout design (this technique of running a script is discussed in “Running Scripts from an Application” on page 161).
4. Choose **File > Close** to direct Xpedition Layout to close the layout design.
5. A dialog displays. Note the **Yes** and **No** buttons as illustrated in Figure 103: Event Invokes an In-process Event Handler Script.

Figure 103. Event Invokes an In-process Event Handler Script



6. Click **No** and observe that the document does not close.

7. Choose **File > Close** again to direct Xpedition Layout to close the layout design.

8. Click **Yes** and observe that the document closes.

Note:
It is common for script writers to refer to "sinking events". This is the process of receiving event notifications from a server and firing an event handler. When the event handler starts processing the event, it is sinking the event.

Events and Out-of-Process Scripts Using mgcscript.exe

This exercise introduces handling events in scripts that run out-of-process.

Consider Lines 21 through 23 from the previous script:

```
21  ' Set the Scripting.DontExit property to
22  ' true to keep the script running.
23  Scripting.DontExit = True
```

Line 23 uses the DontExit property of the Scripting object to prevent the script from exiting. This indicates that the script continue to run until the current application session finishes. This technique also works if the script runs as an out-of-process script from within another application such as Xpedition FabLink. However, this technique does not work if the script runs from the command line using a scripting engine such as mgcscript.exe, because the scripting engine does not have the context of the application closing. The scripting engine is not running inside the application, therefore, it does not understand when the application exits.

To work around this problem, use an infinite loop to prevent the script from exiting.

Creating the Script

Use the scripting editor of your choice to open the EventsInProcess.vbs script in the C:\temp folder and save it as EventsOutOfProcess.vbs. Replace the existing Lines 21 through 23 with the following (Lines 21 through 26), which implements an infinite loop:

```
21  ' Use an infinite loop to keep the script from exiting
22  Dim dontExitBool
23  dontExitBool = True
24  While dontExitBool
25    Scripting.Sleep(300)
26  Wend
```

Line 22 instantiates a dontExitBool variable. Line 23 assigns a Boolean value of True to the variable. Lines 24 through 26 us the *While ... Wend* control structure to create a loop as long as the dontExitBool variable is True. Line 25 causes the script to "go to sleep" for 300 milliseconds for each loop. This also prevents the script from consuming CPU cycles and frees the host computer to perform other tasks.

Modify the event handler as shown below:

```
31  ' PreClose event handler
32  Function pcbDocObj_OnPreClose()
33      Dim ansEnum
34      ansEnum = MsgBox("Close this document?", vbYesNo)
35      If ansEnum = vbYes Then
36          pcbDocObj_OnPreClose = True
37          dontExitBool = False
38      Else
39          pcbDocObj_OnPreClose = False
40      End If
41  End Function
```

This modification includes adding Line 37, which sets the dontExitBool variable to False. When the end-user indicates that the document is to close, the script sets the *dontExitBool* variable to False. This allows the infinite loop (Lines 24 through 26) to finish, which then allows the script to exit.

Running the Script

1. Assure no instantiations of Xpedition Layout are currently running and then launch a new instantiation of Xpedition Layout.

2. Open a layout design of your choice.

3. Open a Console window.

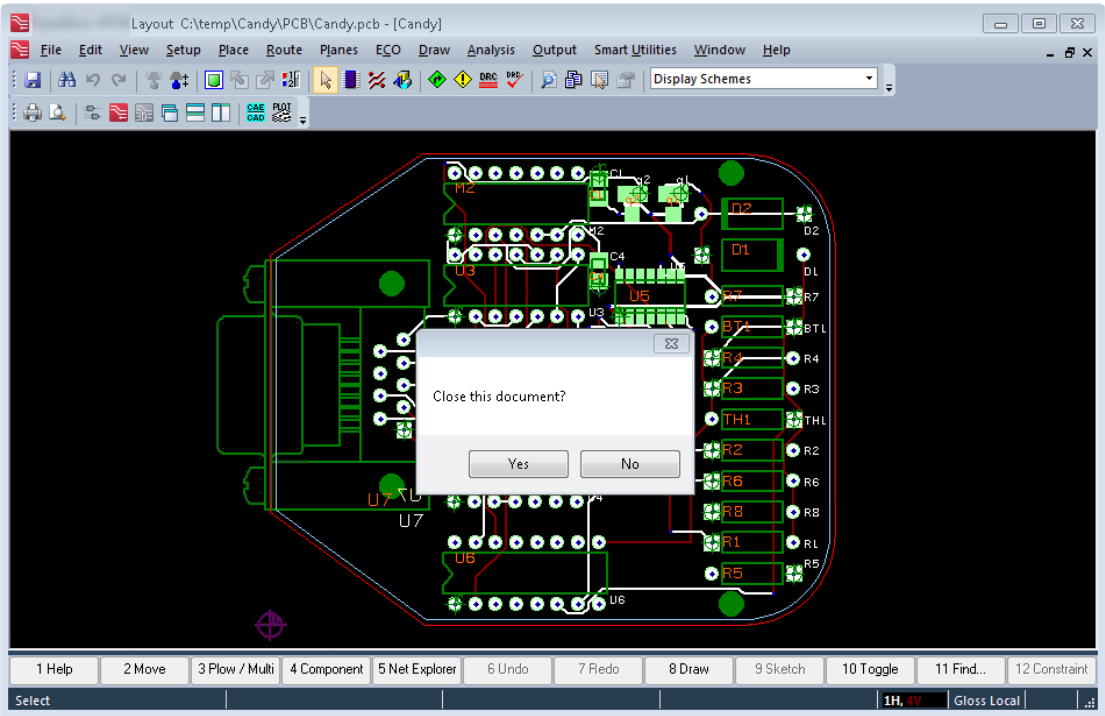
4. Change the working directory to the *C:\temp* folder.

5. Run the *EventsOutOfProcess.vbs* script.

6. Choose **File > Close** to instruct Xpedition Layout to close the layout design.

   A dialog displays as illustrated in Figure 104: Event Invokes an Out-of-process Event Handler Script.

Figure 104. Event Invokes an Out-of-process Event Handler Script



7. Click **No** and observe that the design does not close.

8. Again, choose **File > Close** to instruct Xpedition Layout to close the layout design.

9. Click **Yes** and observe that the design closes.

   At this point, the command prompt returns in the console window. When you click **Yes** the script exits, because the dontExitBool variable is False (Line 37).

Chapter 13
Basic Building-Block Examples

This section presents a number of basic “building-block” automation examples.

Each of these scripts (below) perform a fundamental task in isolation. You can combine these tasks in different ways to form more scripts.

Creating a Skeleton Script
Traversing Objects
Filtering Collections
Using the Display Control
Running Menu Commands
Creating Objects with Geometries
Creating Interactive Commands
Creating Input/Output (I/O) Files
Running Executables
Reading, Analyzing, and Updating a Design

Creating a Skeleton Script

Before proceeding, use the scripting editor of your choice to create the following "skeleton script". Save it as Skeleton.vbs in your C:\temp folder.

```

Option Explicit

' Add any type libraries to be used.
Scripting.AddTypeLibrary("MGCPCB.ExpeditionPCBApplication")

' Get the Application object
Dim pcbAppObj
Set pcbAppObj = Application

' Get the active document
Dim pcbDocObj
Set pcbDocObj = pcbAppObj.ActiveDocument

' License the document
ValidateServer(pcbDocObj)

' #### YOUR CODE WILL GO HERE

''''''
```

```
23  ' Server validation function
24  Private Function ValidateServer(docObj)
       :
      etc.
```

Lines 1 through 16 should be familiar. Following the Option Explicit statement, the script declares any type libraries it uses, returns an Application object, returns a Document object, and then calls the ValidateServer function to license the document.

Line 24 starts the same server validation function introduced in “Running Your First Script” on page 59.

Use this skeleton script as the starting point for all of the scripts in the remainder of this chapter. In each case, replace the comment on Line 18 with the new code for the building-block script.

Traversing Objects

In many cases, it is necessary to iterate through a collection of data objects in order to perform one of the following tasks:

- Locate one or more objects.
- Verify the existence of one or more objects.
- Report information on one or more objects.

The following topics present examples that demonstrate traversing through nets, components, and pins.

Exercise 1: Traversing Nets
Exercise 2: Traversing Components
Exercise 3: Traversing Pins
Task #1: Selecting all pins with the name “1”
Task #2: Selecting all of the pins on a user-specified net
Task #3: Selecting all of the pins on a specific component

Exercise 1: Traversing Nets

This script locates any open nets and presents the end-user with a list of these nets (or informs the end-user if no open nets exist).

Approach: To use a For Each … Next control loop to iterate through all of the nets in a layout design.

Note:
Unlike components, pins, and traces—each of which correspond to *physical* entities in the real world—nets are *logical* constructs that describe electrical connectivity. In the Xpedition Layout Automation Interface, nets are objects with associated methods and properties. They are treated the same as other objects, such as traces and pins.

Creating the Script

Use the scripting editor of your choice to open the *Skeleton.vbs* script in your *C:\temp* folder, save the script as *TraverseNets.vbs*, and then replace the comment on Line 18 with the following code:

```
18  ' Get a collection of nets.
19  Dim netColl         ' Collection of all nets
20  Dim netObj          ' Net object for iteration
21  Set netColl = pcbDocObj.Nets
22
23  ' Instantiate and initialize string to hold net names
24  Dim netListStr: netListStr = ""
25  ' Iterate through all nets
26  For Each netObj In netColl
27     ' Determine the number of opens.
```

```
28     If netObj.FromTos.Count > 0 Then
29        ' This net has an open.  Add its name to the list.
30        netListStr = netListStr & "    " & netObj.Name & vbCrLf
31     End If
32  Next
33
34  ' String to display result.
35  Dim displayStr
36  If netListStr = "" Then
37     ' If we didn't find any let the user know.
38     displayStr = "No opens found."
39  Else
40     ' If we found some tell the user which ones.
41     displayStr = "Open Net List: " & vbCrLf & netListStr
42  End If
43
44  ' Display the result.
45  MsgBox(displayStr)
```

Line 19 instantiates a variable netColl, which holds a collection of nets. Similarly, Line 20 instantiates a variable netObj, which holds a single Net object. Line 21 returns all of the nets from the design and assigns them to the netColl variable.

```
18  ' Get a collection of nets.
19  Dim netColl         ' Collection of all nets
20  Dim netObj          ' Net object for iteration
21  Set netColl = pcbDocObj.Nets
```

Line 24 instantiates and initializes a variable netListStr, which holds a list of open nets.

```
23  ' Instantiate and initialize string to hold net names
24  Dim netListStr: netListStr = ""
```

Lines 26 through 32 use a For Each … Next control loop to iterate through all of the nets in the collection. Whenever the script finds a net with an open (Line 28), the script adds the net name to the list in the netListStr variable.

```
25  ' Iterate through all nets
26  For Each netObj In netColl
27     ' Determine the number of opens.
28     If netObj.FromTos.Count > 0 Then
29        ' This net has an open.  Add its name to the list.
30        netListStr = netListStr & "    " & netObj.Name & vbCrLf
31     End If
32  Next
```

Note:
Observe the sub-string of spaces (" ") in the middle of Line 30. The extra space makes the output more visually appealing. Also on Line 30, observe the vbCrLf character constant (refer to "VBScript Primer" on page 27). This inserts a carriage return and line feed, which appears as a newline when the string displays or prints.

Lines 34 through 42 create the display output string. Line 35 instantiates a variable displayStr, which holds the final string to display.

```
34  ' String to display result.
35  Dim displayStr
36  If netListStr = "" Then
37     ' If we didn't find any let the user know.
38     displayStr = "No opens found."
39  Else
40     ' If we found some tell the user which ones.
41     displayStr = "Open Net List: " & vbCrLf & netListStr
42  End If
```

Line 36 checks if netListStr is empty, which means that no open nets exist in the design. In this case, displayStr is assigned "No opens found." If netListStr is not empty, displayStr is assigned a string containing three elements: the string "Open Net List:", a newline (vbCrLf), and the list of open nets (netListStr).

Finally, Line 45 uses the MsgBox() function to display the result.

```
44  ' Display the result.
45  MsgBox(displayStr)
```

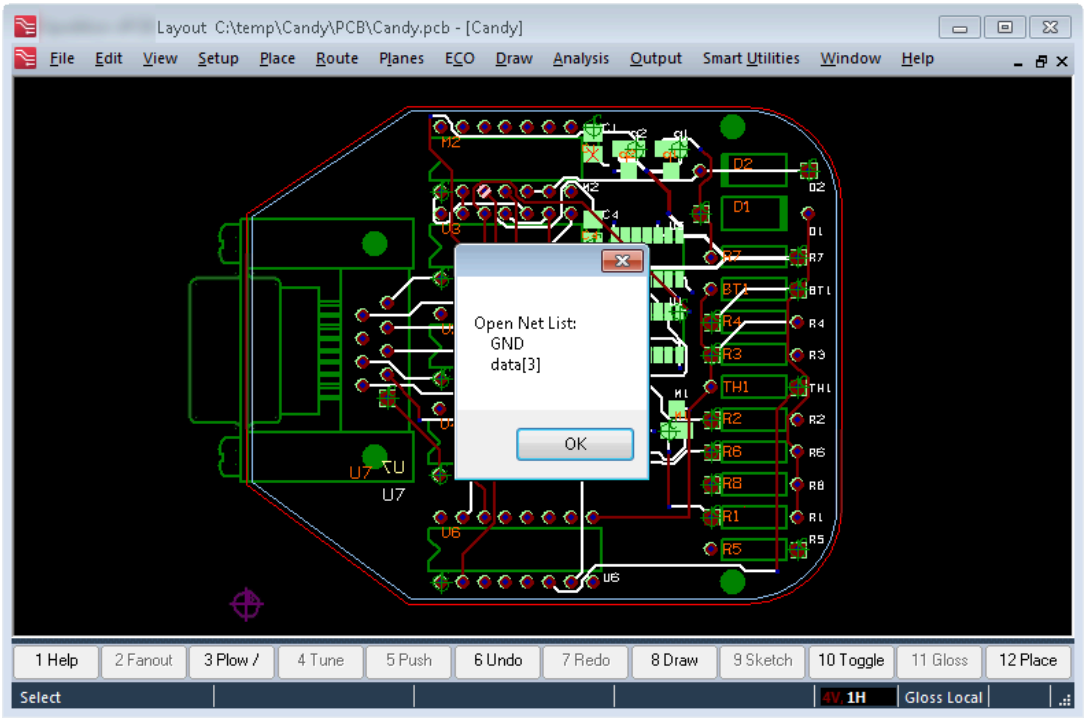
Running the Script

1. Close any instantiations of Xpedition Layout that are currently running, and then launch a new instantiation of this application.

2. Open the *Candy.pcb* design. This design does not contain any open nets, so modify the design to create two or more opens (do *not* save these changes).

3. Drag-and-drop the TraverseNets.vbs script into the layout design window. Observe the result as illustrated in Figure 105: Running the TraverseNets.vbs Script (the actual opens reported depend on changes you make to the design).

Figure 105. Running the TraverseNets.vbs Script



4. Click **OK** to dismiss the dialog and end the script.

5. Choose **File > Close** to close the design, do *not* save any changes you made as part of running this script.

Exercise 2: Traversing Components

This script unplaces all of the design's bottom-side components.

Approach: To use a For … Next control loop to iterate through all of the components in a layout design; filter the unplaced components and then unplace (delete) them.

Assumptions: Assumes that you are working with a design containing both top-side and bottom-side components.

Also of Interest: This script uses the status bar to inform the end-user of actions that are occurring.

Creating the Script

Use the scripting editor of your choice to open the *Skeleton.vbs* script in your *C:\temp* folder, save the script as *TraverseComponents.vbs*, and then replace the comment on Line 18 with the following code:

```
18  ' Let the user know what is going on
19  Call pcbAppObj.Gui.StatusBarText("Finding bottom side components...",

 epcbStatusField1)
```

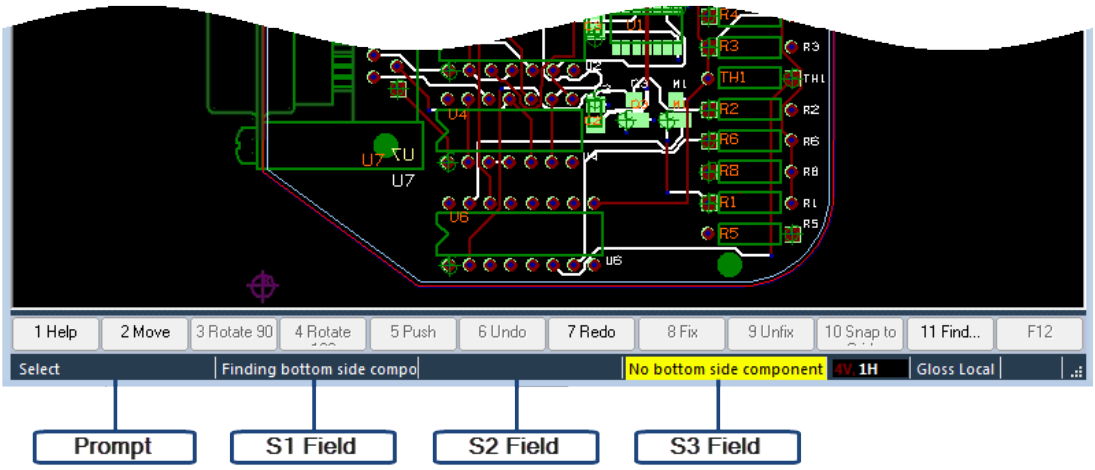
```
20
21  ' Get the component collection
22  Dim cmpsColl
23  Set cmpsColl = pcbDocObj.Components
24
25  ' Filter collection to remove any component not
26  ' found on bottom side of board.
27  Dim i
28  For i = cmpsColl.Count To 1 Step -1
29    ' Determine if this component is on the bottom.
30    If Not cmpsColl.Item(i).Side = epcbSideBottom Then
31      ' Component is not on bottom so remove it
32      cmpsColl.Remove(i)
33    End If
34  Next
35
36  ' Uplace (Delete) all components remaining in the collection
37  If cmpsColl.Count > 0 Then
38    ' Let the user know the components are being deleted.
39    Call pcbAppObj.Gui.StatusBarText("Deleting components...",

 epcbStatusField1)
40    ' Delete the bottom-side components
41       cmpsColl.Delete
42    ' Let the user know what changes were made.
43    Call pcbAppObj.Gui.StatusBarText(cmpsColl.Count &
                                    " components deleted.",
 epcbStatusField1)
44  Else
45    ' Let the user know that no components were deleted.
46    Call pcbAppObj.Gui.StatusBarText("No bottom side
                                    components found.",
 epcbStatusFieldError)
47  End If
```

On Line 19, the script uses the StatusBarText method of the Gui object to display a message on the status bar. Note that Line 19 is split into two lines for aesthetic reasons; in the actual script, this is a single line.

```
18  ' Let the user know what is going on
19  Call pcbAppObj.Gui.StatusBarText("Finding bottom side components...",

 epcbStatusField1)
```

The first parameter of the *StatusBarText* method is the string to display. The second parameter is an enumerate that specifies the field on the status bar (refer to Figure 106: Displaying Messages in the Status Bar). The S3 field is also the Error field (this field turns yellow when it is displaying an error).

Figure 106. Displaying Messages in the Status Bar



Line 22 instantiates a variable cmpsColl, which holds a collection of components. Line 21 returns all of the components in the design and assigns them to the cmpsColl variable.

```
21  ' Get the component collection
22  Dim cmpsColl
23  Set cmpsColl = pcbDocObj.Components
```

The next task is to filter the component collection by removing any components that are *not* on the bottom-side of the board. This may seem counter-intuitive as the goal is to unplace all of the bottom-side components. After the script removes all components that are not on the bottom-side, the collection contains only bottom-side components. The collection can then be unplaced using a single statement.

Another valid approach is to delete the bottom- side components as they are found.

On Lines 25 through 34, the script uses a For … Next control loop to iterate through all of the components in the collection. In particular, observe Line 28, which iterates from the last item in the collection to the first (rather than iterating from the first item to the last). The reason for doing this is that the script removes items based on their item number. Removing items changes the following item numbers, which could potentially change the iteration.

```
25  ' Filter collection to remove any component not
26  ' found on bottom side of board.
27  Dim i
28  For i = cmpsColl.Count To 1 Step -1
29    ' Determine if this component is on the bottom.
30    If Not cmpsColl.Item(i).Side = epcbSideBottom Then
31      ' Component is not on bottom so remove it
32      cmpsColl.Remove(i)
33    End If
34  Next
```

Line 30 checks if the current item is not on the bottom-side and if so, Line 32 uses the Remove method to remove this item from the collection. After the *For ... Next* loop completes, the collection contains only bottom-side components.

Note:
The Remove method does not delete the component from the board – the method simply removes the item from the collection.

Note:
The For … Next loop used in this Traversing Components example behaves differently compared with the For Each … Next loop used on Lines 26 through 32 in the previous "Traversing Nets" on page 189 example. When using the For Each … Next loop, the object is already assigned to a variable when the script performs the If … Then test inside the loop. By comparison, in this example the script uses an integer variable to control the For … Next loop. This is why Line 30 uses the item number to return the individual Component object and compare its Side property to the enumerate epcbSideBottom in order to determine whether or not to remove this component from the collection.

Lines 36 through 46 unplace (delete) all bottom-side components and inform the end-user of actions taking place.

```
36  ' Uplace (Delete) all components remaining in the collection
37  If cmpsColl.Count > 0 Then
38    ' Let the user know the components are being deleted.
39    Call pcbAppObj.Gui.StatusBarText("Deleting components...",

 epcbStatusField1)
40    ' Delete the bottom-side components
41       cmpsColl.Delete
42    ' Let the user know what changes were made.
43    Call pcbAppObj.Gui.StatusBarText(cmpsColl.Count &
                                   " components deleted.",
 epcbStatusField1)
44  Else
45    ' Let the user know that no components were deleted.
46    Call pcbAppObj.Gui.StatusBarText("No bottom side
                                   components found.",
 epcbStatusFieldError)
47  End If
```

Line 37 checks if there are any components remaining in the collection; if not, then there are no bottom-side components in this design, in which case the statement on Line 45 informs the end-user accordingly.

If there are bottom-side components, then Line 39 informs the end-user that the script is unplacing components; Line 41 uses the Delete method on the cmpsColl component collection to unplace all components in the collection; and Line 43 informs the end-user of the number of components deleted (unplaced).

Generally, a single method or property action in a script allows a single **Undo** action in the GUI. The exception to this is when performing actions on collections of objects. For the *Component* collection, the *Delete* method unplaces the components one at a time, so only the last component unplace operation can be undone. You can **Undo** the entire collection *Delete* method by enclosing the unplace operation in a transaction. To do this, replace Lines 40 and 41 with the following statements:

```
40        ' Use a transaction to group all changes onto a single undo
41     pcbDocObj.TransactionStart
42     ' Delete the bottom-side components
43     cmpsColl.Delete
44        ' End the transaction
45     pcbDocObj.TransactionEnd
```

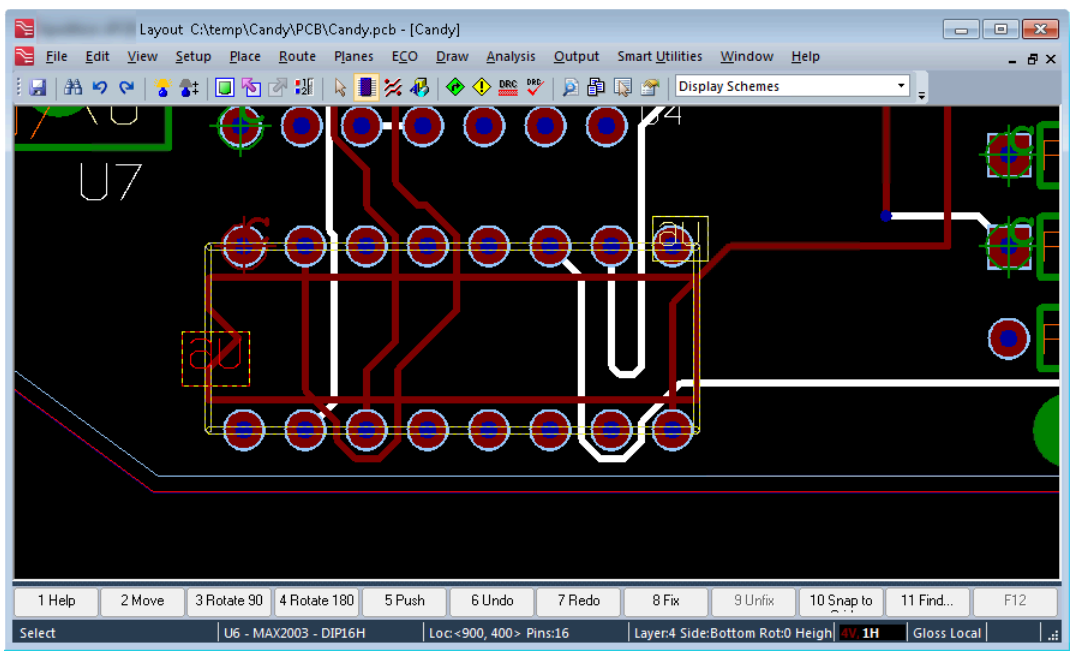
Line 41 declares the start of a transaction; Line 43 performs the unplace of the components; and Line 45 declares the end of the transaction. Transactions can be used to group multiple actions into a single **Undo** action in the GUI. In this case, the script uses a transaction to collect all of the component unplace actions into a single action.

Running the Script

1. Close any instantiations of Xpedition Layout that are currently running, and then launch a new instantiation of this application.

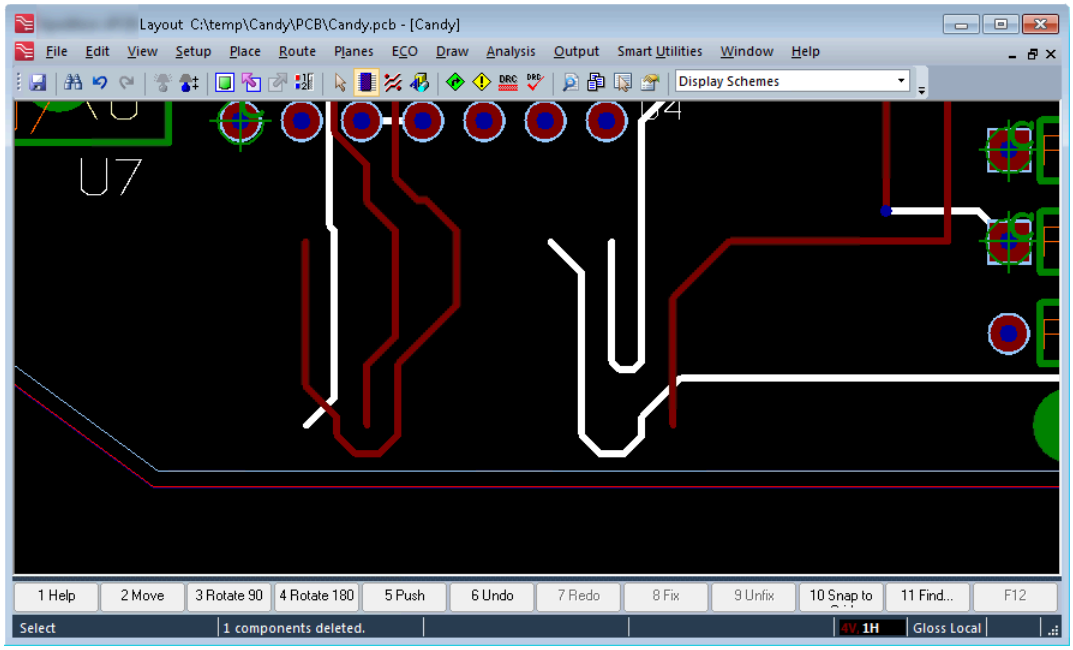
2. Open the Candy.pcb design. This design does not contain bottom-side components, so modify the design to push one of more components to the bottom-side (do not save these changes). Refer to Figure 107: Pushing Component to Bottom-side for an example.

Figure 107. Pushing Component to Bottom-side



3. Drag-and-drop the TraverseComponents.vbs script into the layout design window. Observe the result as illustrated in Figure 108: Bottom-side Component Deleted (the actual components deleted depend on changes you make to the design).

Figure 108. Bottom-side Component Deleted



4. Choose **File > Close** to close the design, do *not* save any changes you made as part of running this script.

Exercise 3: Traversing Pins

This script has three goals and three tasks. First, to selects all pins with the name '1'. Second, to select all of the pins on a specific net. Third, to select all of the pins on a specific component.

Approach: To use the pins associated with different types of objects (Document, Net, and Component).

Also of Interest: Using pairs of double-quotes in strings.

Creating the Script

Use the scripting editor of your choice to open the *Skeleton.vbs* script in your *C:\temp* folder, save the script as *TraversePins.vbs*, and then replace the comment on Line 18 with the following code:

```
18  ' Common varaibles
19  Dim pinColl
20  Dim pinObj
21
22  ''''''
```

```
  ' See if this is pin '1'
  If pinObj.Name = "1" Then
    pinObj.Selected = True
  End If
Next

' Let the user know what has been selected
MsgBox("All pins name ""1"" have been selected.")

' Clean up by unselecting
pcbDocObj.UnselectAll

''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
' Select all pins on a net

' Ask the user for a net to process
Dim netNameStr: netNameStr = InputBox("Enter a net name.")

' Get the net object
Dim netObj
Set netObj = pcbDocObj.FindNet(netNameStr)

' If we have a valid net object select all the pins on that net.
If Not netObj Is Nothing Then
  Set pinColl = netObj.Pins
  pinColl.Selected = True

  ' Let the user know what has been selected
  MsgBox("All pins on net " & netObj.Name & " have been selected.")
Else
  ' Tell the user their entry was invalid
  MsgBox(netNameStr & " was not found.")
End If

' Clean up by unselecting
pcbDocObj.UnselectAll

''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
' Select all pins on a component

' Ask the user for a component ref des to process
Dim cmpRefDesStr: cmpRefDesStr =
                                InputBox("Enter a component ref des.")

' Get component object
Dim cmpObj
Set cmpObj = pcbDocObj.FindComponent(cmpRefDesStr)

' If we have a valid component object select all pins on it.
If Not cmpObj Is Nothing Then
  Set pinColl = cmpObj.Pins
  pinColl.Selected = True

```

```
81    ' Let the user know what has been selected
82    MsgBox("All pins on net " & cmpRefDesStr & " have been selected.")
83  Else
84    ' Tell the user their entry was invalid
85    MsgBox(cmpRefDesStr & " was not found.")
86  End If
87
88  ' Clean up by unselecting
89  pcbDocObj.UnselectAll
```

On Lines 18 through 20, the script declares some common variables to use for all three tasks. Line 19 instantiates a variable pinColl, which holds a collection of pins. Line 20 instantiates a variable pinObj, which holds a single Pin object.

```
18  ' Common variables
19  Dim pinColl
20  Dim pinObj
```

Task #1: Selecting all pins with the name “1”

On Line 25, the script returns all of the pins from the design and assigns them to the pinColl variable.

```
25  Set pinColl = pcbDocObj.Pins
```

Lines 28 through 33 use a *For Each … Next* control loop to iterate through all of the pins in the collection. For each pin in the collection, Line 30 checks if the name of the *Pin* object is “1”. If the pin's name is “1”', then Line 31 sets the Selected property associated with this Pin object to True. This causes the pin to be selected in the GUI.

```
27  ' Select all pins with name '1'
28  For Each pinObj In pinColl
29    ' See if this is pin '1'
30    If pinObj.Name = "1" Then
31      pinObj.Selected = True
32    End If
33  Next
```

Line 36 uses a MsgBox() function to inform the end-user of the result.

```
35  ' Let the user know what has been selected
36  MsgBox("All pins name ""1"" have been selected.")
```

After the end-user dismisses the message box, Line 39 uses the UnselectAll method of the Document object to unselect everything in the design.

```
38  ' Clean up by unselecting
39  pcbDocObj.UnselectAll
```

Task #2: Selecting all of the pins on a user-specified net

On Line 45, the script instantiates a variable netNameStr, which holds the net name. Also on Line 45, the script uses an InputBox() function to ask the end-user to enter a net to process. The value this InputBox() function returns, is assigned to the netNameStr variable.

```
44  ' Ask the user for a net to process
45  Dim netNameStr: netNameStr = InputBox("Enter a net name.")
```

Line 48 instantiates a variable netObj, which holds a Net object. Line 49 uses the FindNet method to search for the net the end-user entered. If that net exists, it is assigned to the netObj variable; otherwise, the *FindNet* method automatically sets the netObj variable to the null object, Nothing.

```
47  ' Get the net object
48  Dim netObj
49  Set netObj = pcbDocObj.FindNet(netNameStr)
```

Line 52, checks if the net is found. If netObj is not equal to Nothing, then the net exists. In this case, Line 53 returns the collection of pins on the Net object and assigns them to the pinColl variable.

```
51  ' If we have a valid net object select all the pins on that net.
52  If Not netObj Is Nothing Then
53    Set pinColl = netObj.Pins
```

Line 54, sets the Selected property on the Pins collection (pinColl) to True. This causes the pins on the user-specified net to be selected in the GUI.

```
54    pinColl.Selected = True
```

Line 57 uses a MsgBox() function to inform the end-user of the result.

```
56    ' Let the user know what has been selected
57    MsgBox("All pins on net " & netObj.Name & " have been selected.")
```

If the test on Line 52 indicates that netObj equals to Nothing, then the net the end-user specifies does not exist. In this case, Line 60 informs the end-user accordingly.

```
58  Else
59    ' Tell the user their entry was invalid
60    MsgBox(netNameStr & " was not found.")
61  End If
```

After the end-user dismisses the message box, Line 64 uses the UnselectAll method to unselect everything in the design.

```
63  ' Clean up by unselecting
64  pcbDocObj.UnselectAll
```

Task #3: Selecting all of the pins on a specific component

On Line 70, the script instantiates a variable cmpRefDesStr, which contains the "component reference designator string". The reference designator is essentially the name of the component. Line 70 also uses an InputBox() function to ask the end-user to enter the reference designator to process. The value returned from this InputBox() function is assigned to the cmpRefDesStr variable.

```
69  ' Ask the user for a component ref des to process
70  Dim cmpRefDesStr: cmpRefDesStr =
                              InputBox("Enter a component ref des.")
```

Line 73 instantiates a variable cmpObj, which holds a Component object. Line 74 uses the FindComponent method to search for the component with a reference designator that matches what the end-user entered. If that component exists, it is assigned to the cmpObj variable; otherwise, the *FindComponent* method automatically sets the cmpObj variable to Nothing.

```
72  ' Get component object
73  Dim cmpObj
74  Set cmpObj = pcbDocObj.FindComponent(cmpRefDesStr)
```

Line 76 checks if the matching component is found. If cmpObj is *not* equal to Nothing, then the component exists. In this case, Line 78 returns the collection of pins for the Component object and assigns them to the pinColl variable.

```
76  ' If we have a valid component object select all pins on it.
77  If Not cmpObj Is Nothing Then
78    Set pinColl = cmpObj.Pins
```

Line 79 sets the Selected property of the Pins collection (pinColl) to True. This causes the pins on the user-specified component/reference designator to be selected in the GUI.

```
79    pinColl.Selected = True
```

Line 82 uses a MsgBox() function to inform the end-user of the result.

```
81    ' Let the user know what has been selected
82    MsgBox("All pins on net " & cmpRefDesStr & " have been selected.")
```

If the test on Line 77 indicates that cmpObj equals Nothing, then the component the end-user specified does not exist. In this case, on Line 85 informs the end-user accordingly.

```
83  Else
84    ' Tell the user their entry was invalid
85    MsgBox(cmpRefDesStr & " was not found.")
86  End If
```

After the end-user dismisses the message box, Line 89 uses the UnselectAll method to unselect everything in the design.

```
88  ' Clean up by unselecting
89  pcbDocObj.UnselectAll
```

The three tasks in this script use the Pins property to access a collection of pins. In each case, the property is associated with a different object. Line 25 specifies the Document object; Line 53 specifies a Net object; and Line 78 specifies a Component object. A different collection of pins is returned in each case.

```
25  Set pinColl = pcbDocObj.Pins
```

```
53  Set pinColl = netObj.Pins
```

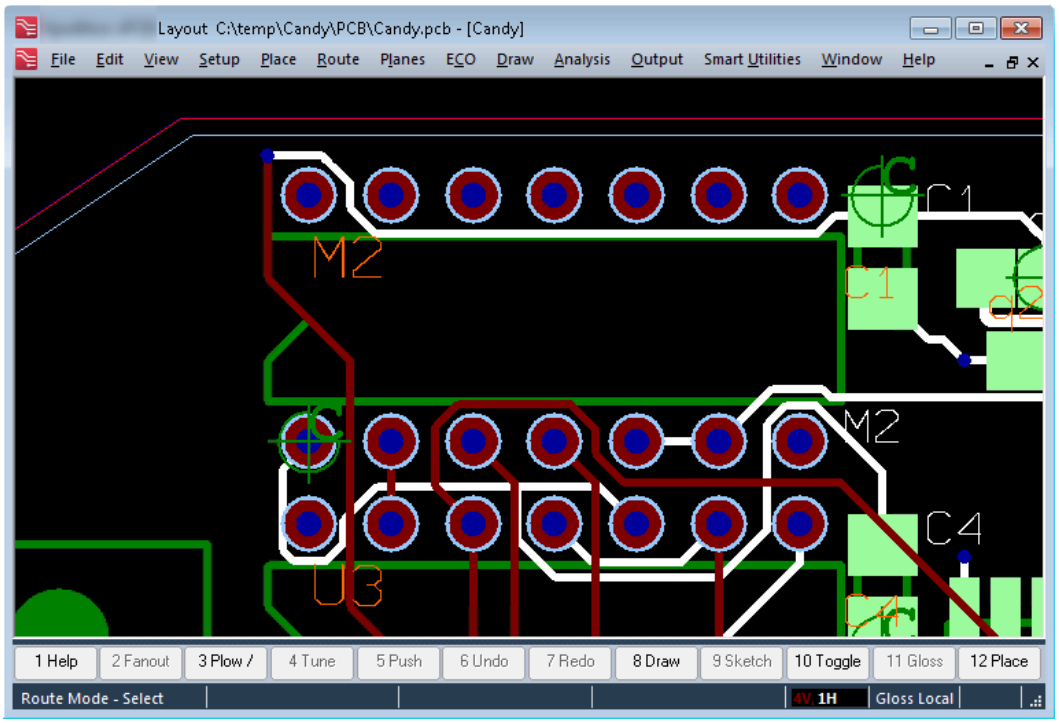
```
78  Set pinColl = cmpObj.Pins
```

Running the Script

1. Close any instantiations of Xpedition Layout that are currently running, and then launch a new instantiation of this application.

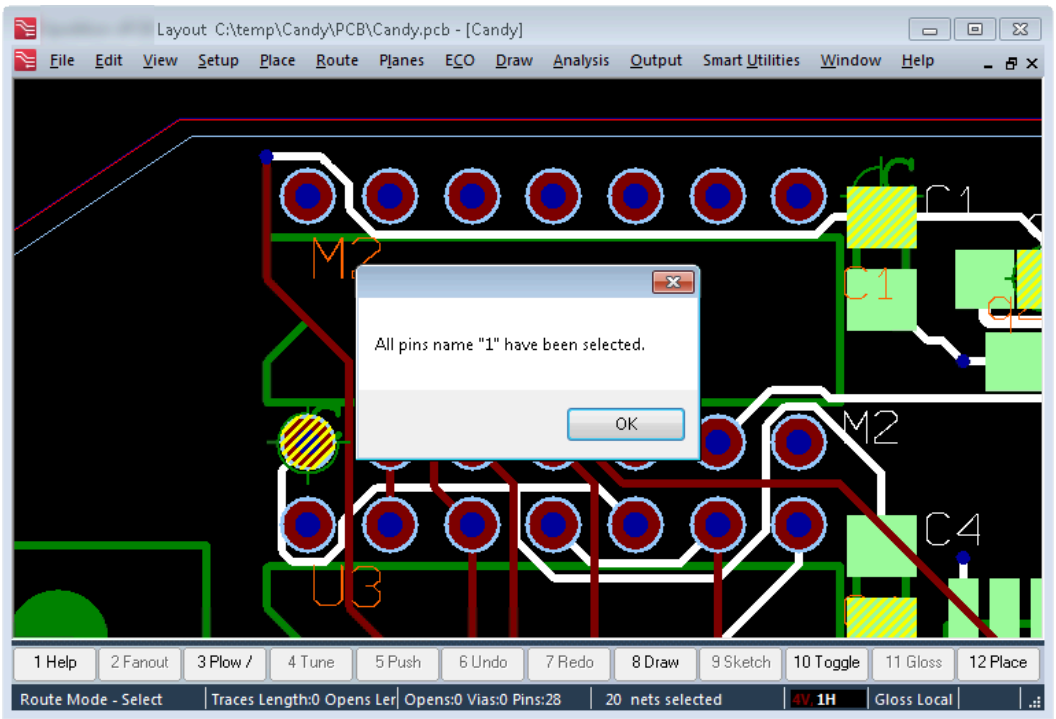
2. Open the Candy.pcb design and zoom in on component M2 as illustrated in Figure 109: Zoom On Component M2.

Figure 109. Zoom On Component M2



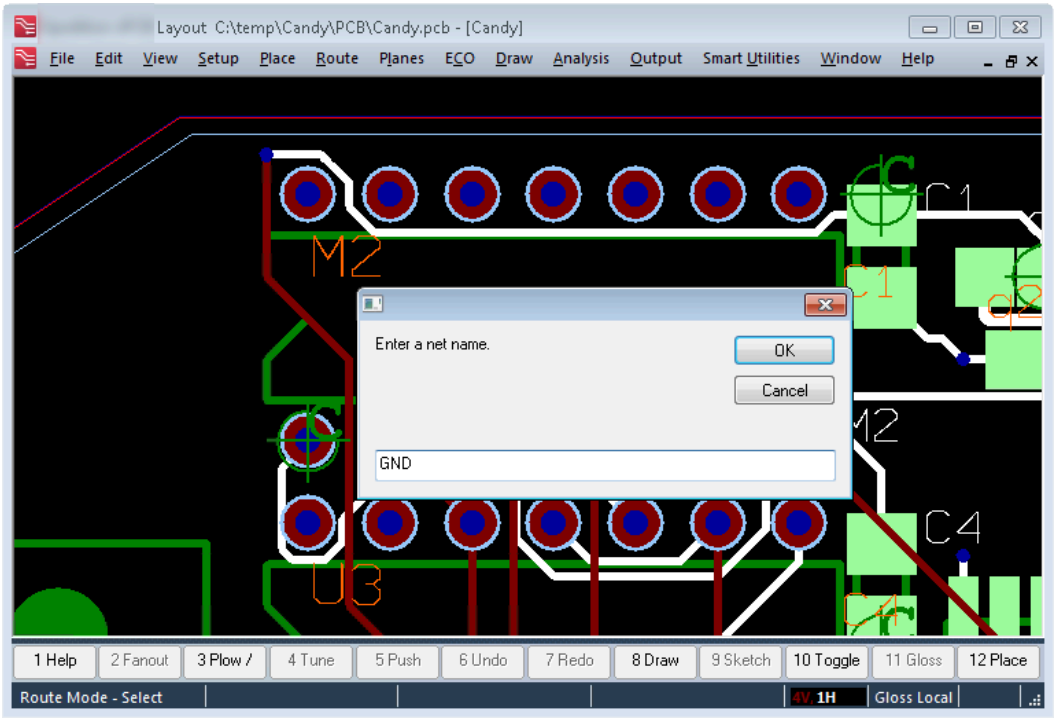
3. Drag-and-drop the TraversePins.vbs script into the layout design window. The message box for **Task #1** displays as in Figure 110: Result From Task #1 in the TraversePins.vbs Script. Observe that one pin in the lower-left-hand corner of component M2 is selected and one pin on capacitor C1 is selected.

Figure 110. Result From Task #1 in the TraversePins.vbs Script



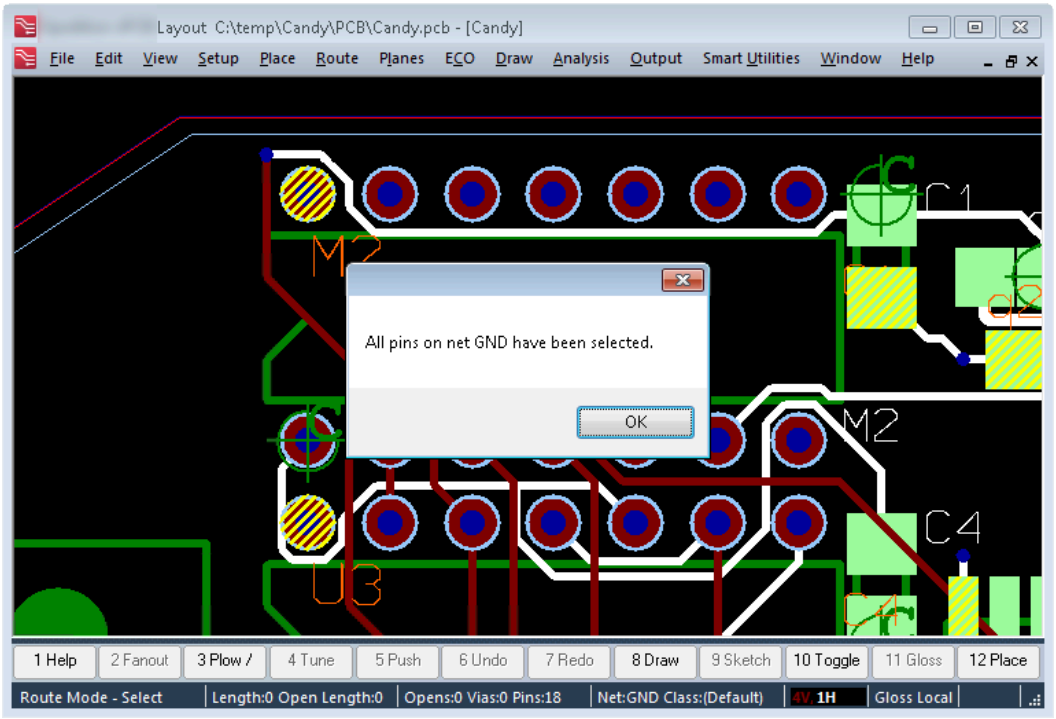
4. Click **OK** to dismiss the dialog. The input box for **Task #2** displays and requests the name of a net. Enter the name "GND" as illustrated in Figure 111: Query for Task #2.

Figure 111. Query for Task #2



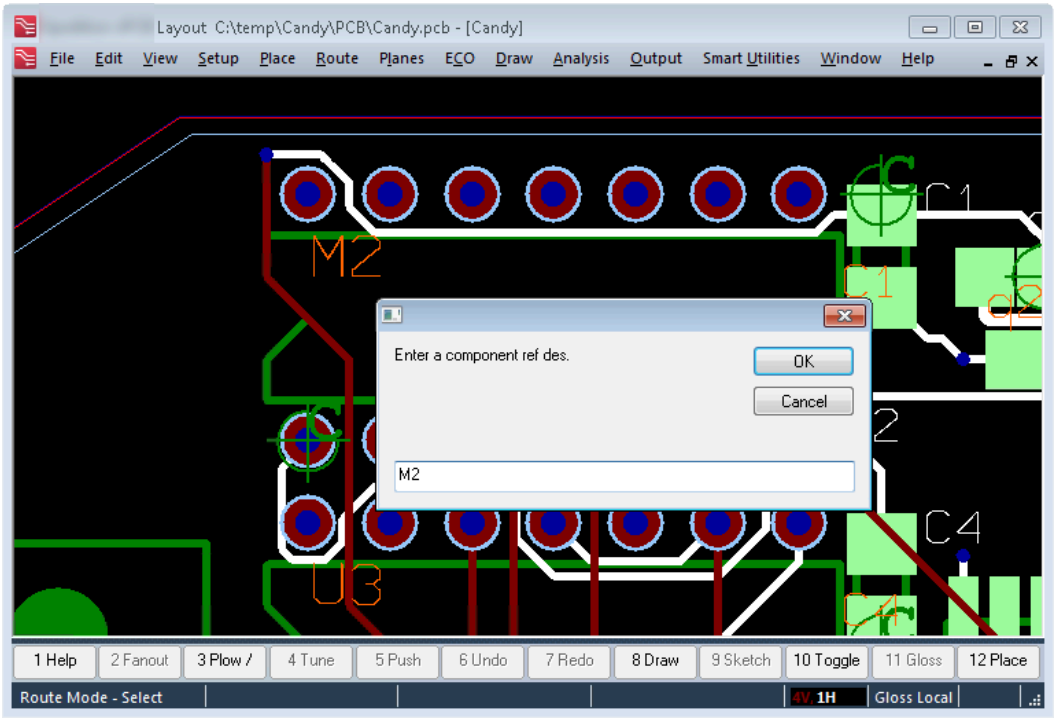
5. Click **OK** and observe the results from **Task #2**. Two new pins are selected and a new message box displays as illustrated in Figure 112: Result of Task #2.

Figure 112. Result of Task #2



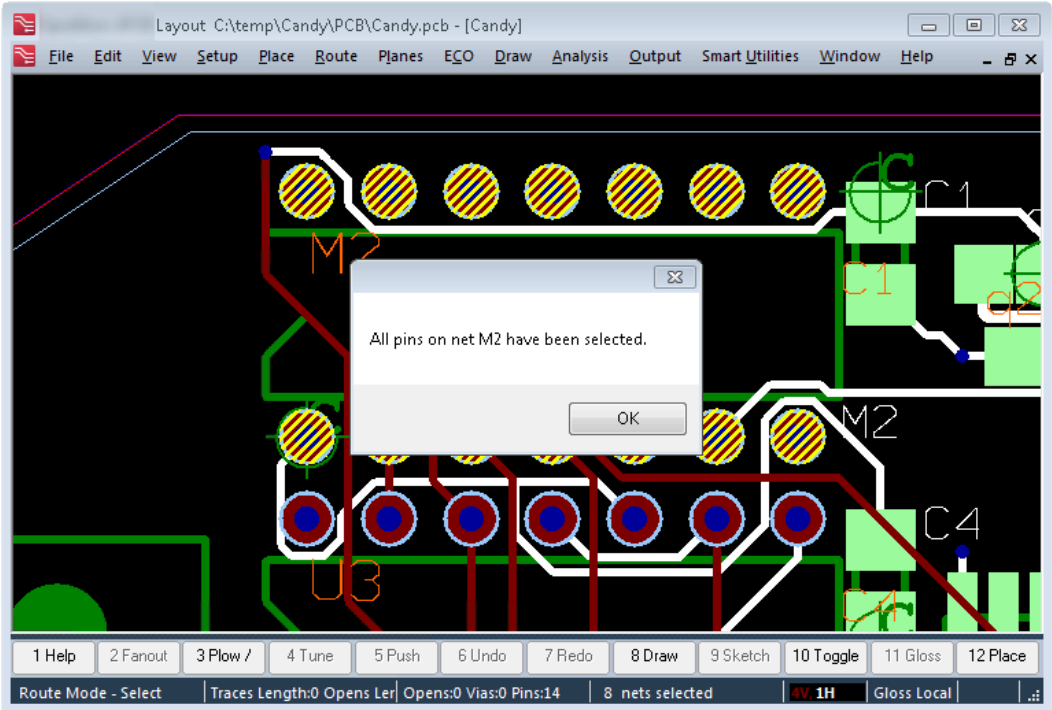
6. Click **OK** and then enter the reference designator "M2" into the **Task #3** Input box (Figure 113: Query for Task #3).

Figure 113. Query for Task #3



7. Click **OK** to accept the "M2" reference designator. Observe the results from **Task #3** (Figure 114: Result from Task #3).

Figure 114. Result from Task #3



8. Click **OK** to dismiss the message box and end the script.

Filtering Collections

The example scripts in the Traversing Objects section iterate through collections of objects and — in some cases — explicitly filter these collections. The Automation Interface has many methods that automatically filter collections. This section presents three examples that demonstrate collection filters.

Exercise 4: Finding Unplaced Components
Exercise 5: Deleting Traces
Exercise 6: Filtering Padstack Objects

Exercise 4: Finding Unplaced Components

To provide a list of any unplaced components in a design.

Approach: To use a filter on the Component property in order to access only unplaced components.

Creating the Script

Use the scripting editor of your choice to open the *Skeleton.vbs* script in your *C:\temp* folder, save the script as *UnplacedComponents.vbs*, and then replace the comment on Line 18 with the following code:

```
18  ' Get the component collection
19  Dim cmpsColl
20  Dim cmpObj
21  Set cmpsColl = pcbDocObj.Components(epcbSelectUnplaced)
22
23  ' Instantiate and initialize string to hold net names
24  Dim cmpListStr: cmpListStr = ""
25
26  ' List all unplaced components.
27  For Each cmpObj In cmpsColl
28    cmpListStr = cmpListStr & "     " & cmpObj.Name & vbCrLf
29  Next
30
31  ' String to display result.
32  Dim displayStr
33  If cmpListStr = "" Then
34    ' If we didn't find any let the user know.
35    displayStr = "No unplaced components found."
36  Else
37    ' If we found some tell the user which ones.
38    displayStr = "Unplaced Component List: " & vbCrLf & cmpListStr
39  End If
40
41  ' Display the result.
42  MsgBox(displayStr)
```

On Line 19, the script instantiates a variable cmpsColl, which holds a collection of components. Line 20 instantiates a variable cmpObj, which represents a single Component object. Line 21 uses the

Components property to return a collection of design components and assign them to the cmpsColl variable.

```
18  ' Get the component collection
19  Dim cmpsColl
20  Dim cmpObj
21  Set cmpsColl = pcbDocObj.Components(epcbSelectUnplaced)
```

The Components property is used in other scripts, but without a selection parameter. Without any parameters, *all* of the components in the design are returned in the collection. The Components property supports four parameters, each of which provide different filtering capabilities. In the case of Line 21, the enumerate epcbSelectUnplaced, which filters the collection such that only unplaced components are returned, is specified as the first parameter. Refer to Xpedition Layout Automation Help for more details on the *Document* object's *Components* property.

Line 24 instantiates and initializes a variable cmpListStr, which stores a list of names for unplaced components.

```
23  ' Instantiate and initialize string to hold net names
24  Dim cmpListStr: cmpListStr = ""
```

Lines 26 through 32 use a For Each … Next control loop to iterate through all of the components in the collection. For each component, Line 28 adds the component's name to the the cmpListStr variable.

```
26  ' List all unplaced components.
27  For Each cmpObj In cmpsColl
28    cmpListStr = cmpListStr & "     " & cmpObj.Name & vbCrLf
29  Next
```

On Line 28, the sub-string of spaces (" ") and the newline (*vbCrLf)* make the output more visually appealing when output. Refer to "VBScript Primer" on page 27 for information on *vbCrLf*.

Lines 31 through 39 build the string to display. Line 32 instantiates a variable displayStr, which holds the final string to output.

```
31  ' String to display result.
32  Dim displayStr
33  If cmpListStr = "" Then
34    ' If we didn't find any let the user know.
35    displayStr = "No unplaced components found."
36  Else
37    ' If we found some tell the user which ones.
38    displayStr = "Unplaced Component List: " & vbCrLf & cmpListStr
39  End If
```

Line 33 checks if cmpListStr is empty, which means that no unplaced components exist in the design. If no unplaced components are found then displayStr is assigned "No unplaced components found." If *cmpListStr* is not empty, there are unplaced components. In this case, the displayStr variable is assigned a string containing three elements: the string "Unplaced Components List:", a newline (vbCrLf), and the contents of the cmpListStr variable.

Finally, Line 42 uses the MsgBox() function to display the result.

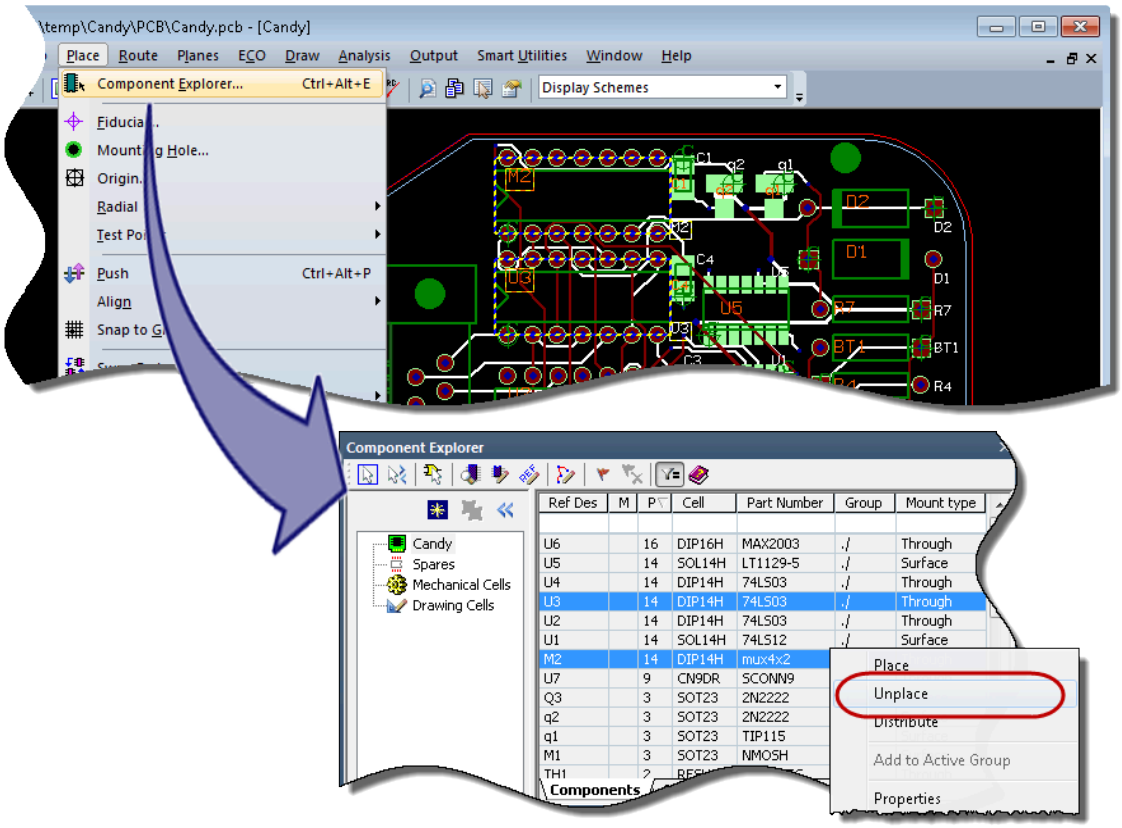
```
41  ' Display the result.
42  MsgBox(displayStr)
```

Running the Script

1. Close any instantiations of Xpedition Layout that are currently running, and then launch a new instantiation of this application.

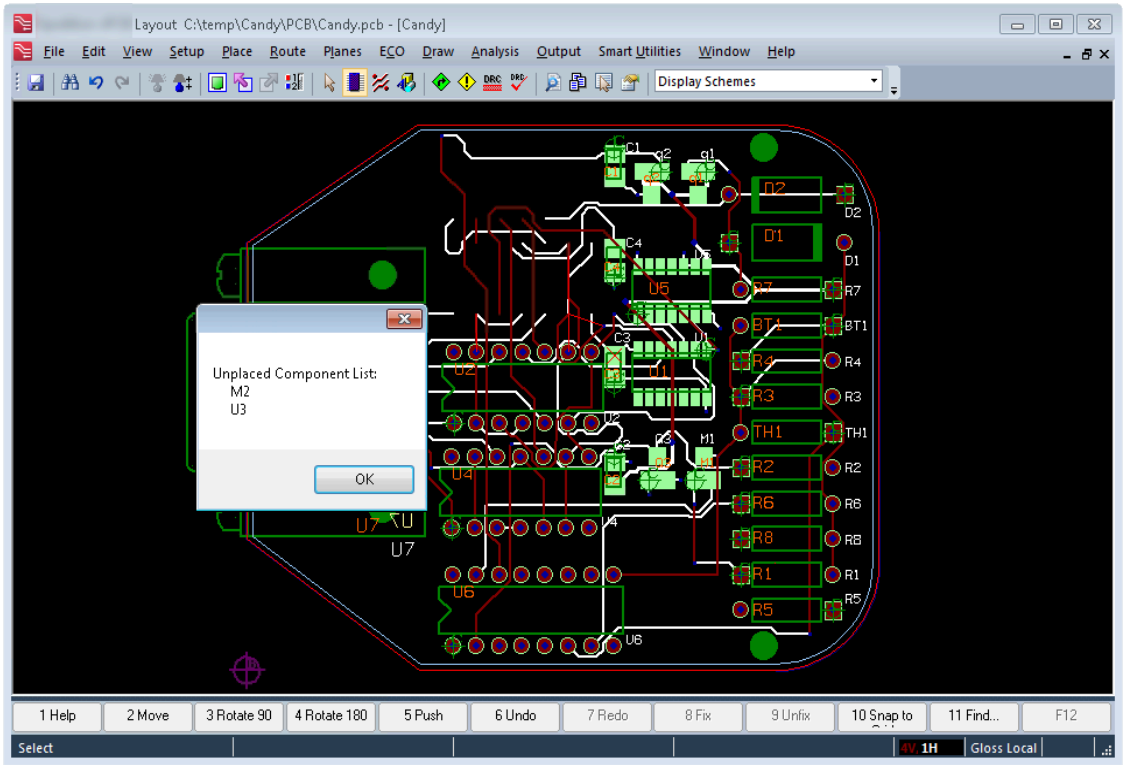
2. Open the Candy.pcb design. This design does not contain unplaced components, so modify the design to unplace one of more components as illustrated in Figure 115: Unplacing Two Components (do not save these changes).

Figure 115. Unplacing Two Components



3. Drag-and-drop the UnplacedComponents.vbs script into the layout design window. Observe the result as illustrated in Figure 116: Result of UnplacedComponents.vbs (the actual components deleted depend on changes you make to the design).

Figure 116. Result of UnplacedComponents.vbs



4. Choose **File > Close** to close the design, do *not* save any changes you made as part of running this script.

Exercise 5: Deleting Traces

This script deletes all unfixed and unlocked traces in a design.

Approach: To use a filter on the Traces property in order to locate and then delete any unfixed and unlocked traces.

Also of Interest: This script uses the status bar to inform the end-user as to what actions are occurring. It also provides an example of using the bit-mask properties associated with certain enumerates.

Creating the Script

Use the scripting editor of your choice to open the *Skeleton.vbs* script in your *C:\temp* folder, save the script as *DeleteTraces.vbs*, and then replace the comment on Line 18 with the following code:

```
18  ' Get a collection of unfixed and unlocked traces
19  Dim trcColl
20  Set trcColl =pcbDocObj.Traces(epcbSelectUnfixed Or epcbSelectUnlocked)
21
22  ' Store the number of traces in collection to report later
23  Dim cntInt
```

```
24  cntInt = trcColl.Count
25
26  ' Delete traces
27  trcColl.Delete
28
29  ' Let the user know what was done.
30  If cntInt = 0 Then
31    Call pcbAppObj.Gui.StatusBarText
            ("No unfixed or unlocked traces found.", epcbStatusFieldError)
32  Else
33    Call pcbAppObj.Gui.StatusBarText
                          (cntInt & " traces deleted.", epcbStatusField1)
34  End If
```

On Line 19, the script instantiates a variable trcColl, which holds a collection of traces. Line 21 uses the Traces property to return a collection of design traces and assigns them to the trcColl variable.

```
19  Dim trcColl
20  Set trcColl =pcbDocObj.Traces(epcbSelectUnfixed Or epcbSelectUnlocked)
```

Note:
The Traces property is used in other scripts, but without any parameters. This results in all of the design traces returning in the collection. The Traces property supports two parameters, both of which provide different filtering capabilities. Line 20, specifies the first parameter and uses the default value for the second.

Note:
Some enumerates are based on consecutive integers such as 0, 1, 2, 3, 4, 5, etc. Others are based on integers that correspond to individual bits such a 1, 2, 4, 8, 16, etc. The epcbSelectUnfixed and epcbSelectUnlocked enumerates on Line 20 are of this latter type. This allows you to logically OR enumerates together to form a single parameter. This means that the collection contains only unfixed *and* unlocked traces. It is important to note that although the parameters are OR'd together, this does not return a collection of traces that are unfixed *or* unlocked; instead, it returns traces that are both unfixed *and* unlocked. Both enumerates must be satisfied. (Refer to the Xpedition Layout Automation Help for more details on the enumerates and the parameters associated with the *Document* object's Traces property.)

Line 23 instantiates a variable cntInt, which contains the number of unfixed and unlocked traces in the collection. Line 24 uses the Count property of the traces collection to determine how many unfixed and unlocked traces there are in the design and assigns this value to the cntInt variable.

```
22  ' Store the number of traces in collection to report later
23  Dim cntInt
24  cntInt = trcColl.Count
```

Line 27 uses the Delete method on the trace collection to delete all of the unfixed and unlocked traces in the design.

```
26  ' Delete traces
27  trcColl.Delete
```

Finally, Lines 29 through 34 use the status bar to inform the end-user as to what occurred. A discussion on the status bar can be found in the Exercise 1: Traversing Nets script earlier in this chapter. Of interest here is that Line 31 uses the Error status field, which highlights in yellow when the message displays.

```
29  ' Let the user know what was done.
30  If cntInt = 0 Then
31    Call pcbAppObj.Gui.StatusBarText(
              "No unfixed or unlocked traces found.", epcbStatusFieldError)
32  Else
33    Call pcbAppObj.Gui.StatusBarText
                            (cntInt & " traces deleted.", epcbStatusField1)
34  End If
```

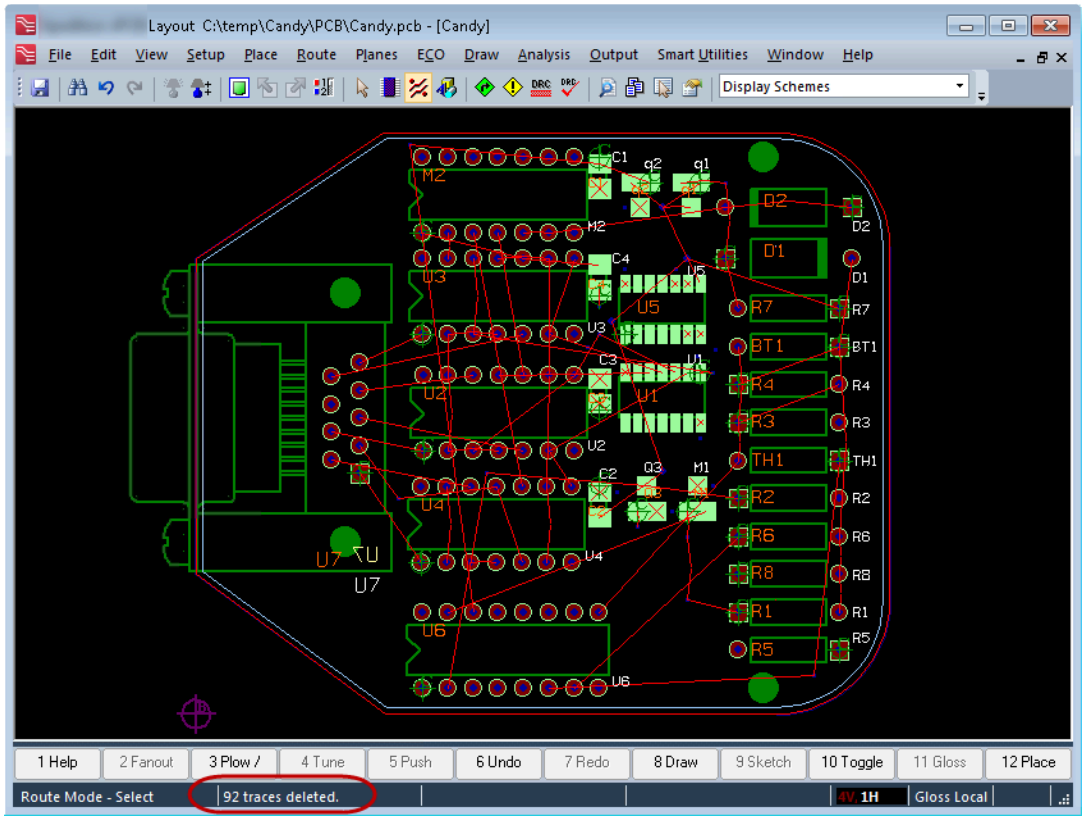
Running the Script

1. Close any instantiations of Xpedition Layout that are currently running, and then launch a new instantiation of this application.

2. Open the Candy.pcb design.

3. Drag-and-drop the DeleteTraces.vbs script into the layout design window. Observe the result as illustrated in Figure 117: Result of DeleteTraces.vbs (the actual number of deleted traces depends on the state of the design).

Figure 117. Result of DeleteTraces.vbs



4. Choose **File > Close** to close the design, do *not* save any changes you made as part of running this script.

Exercise 6: Filtering Padstack Objects

This script counts the number of vias or pins containing pads on layer 1 of the board.

Approach: To use a filter on the PadStackObject property associated with a Document object to return a collection of vias or pins.

Also of Interest: This script uses the PadstackObject object, which is a "general object" that represents pins, vias and other objects that have padstacks. It also provides an example of bit-mask properties associated with certain enumerates.

In previous examples, the scripts have focused on specific object types (Pins, Nets, Traces, Components, etc.). In some cases, however, you can access objects in a more general manner. For example, pins and vias have common characteristics, which is that they are both defined by a padstack.

The Xpedition Layout Automation Interface provides a PadstackObject object, which has methods and properties that are common to all padstack-based objects. This script uses the PadstackObject object.

Creating the Script

Use the scripting editor of your choice to open the *Skeleton.vbs* script in your *C:\temp* folder, save the script as *PadstackObjects.vbs*, and then replace the comment on Line 18 with the following code:

```
18  ' Get a collection of pins and vias.
19  Dim pstkObColl
20  Dim pstkObObj
21  Set pstkObColl = pcbDocObj.PadstackObjects(epcbPadstackObjectVia
                                              Or epcbPadstackObjectPin)
22
23  ' Iterate through all the padstack objects in the collection
24  Dim cntInt
25  For Each pstkObObj In pstkObColl
26    ' If the padstack object has a pad on layer 1 count it.
27    If pstkObObj.Pads(1).Count > 0 Then
28      cntInt = cntInt + 1
29    End If
30  Next
31
32  ' Tell the user what was found.
33  MsgBox("There are " & cntInt & " pins or vias with a pad on layer 1.")
```

On Line 19, the script instantiates a variable pstkObColl ("padstack object collection"), which holds a collection of objects. Similarly, Line 20 instantiates a variable pstkObObj ("padstack object object"), which represents a single object. Line 21 uses the PadstackObjects property of the Document object to return a collection of padstack objects and assigns them to the pstkObColl variable.

```
18  ' Get a collection of pins and vias.
19  Dim pstkObColl
20  Dim pstkObObj
21  Set pstkObColl = pcbDocObj.PadstackObjects(epcbPadstackObjectVia
                                              Or epcbPadstackObjectPin)
```

If no parameters are specified with the PadstackObjects property, then all of the padstack objects in the design are returned. In this case, Line 21 specifies a parameter that selects only the pads and vias.

Note:
Some enumerates are based on consecutive integers such as 0, 1, 2, 3, 4, 5, etc. Others are based on integers that correspond to individual bits such a 1, 2, 4, 8, 16, etc. The epcbPadstackObjectVia and epcbPadstackObjectVia enumerates shown on Line 21 are of this latter type. This allows you to logically OR enumerates together to form a single parameter. This means that the collection contains only vias *and* pins (Refer to the Xpedition Layout Automation Help for more details on the enumerates and the parameters associated with the *Document* object's PadstackObjects property.)

Line 23 instantiates a variable cntInt, which contains the number of pins and vias in the collection that have a pad on layer 1 of the board.

```
24  Dim cntInt
```

Lines 23 through 30 iterate through the collection counting and pins and vias that have a pad on layer 1 of the board. Line 27 uses the Pads property associated with the PadstackObject object (represented by the variable pstkObObj). The parameter (1) specifies to return only pads on layer 1.

```
23  ' Iterate through all the padstack objects in the collection
24  Dim cntInt
25  For Each pstkObObj In pstkObColl
26    ' If the padstack object has a pad on layer 1 count it.
27    If pstkObObj.Pads(1).Count > 0 Then
28      cntInt = cntInt + 1
29    End If
30  Next
```

Finally, Line 33 uses a MsgBox() to inform the end-user how many pins and vias in the design have a pad on layer 1.

```
32  ' Tell the user what was found.
33  MsgBox("There are " & cntInt & " pins or vias with a pad on layer 1.")
```

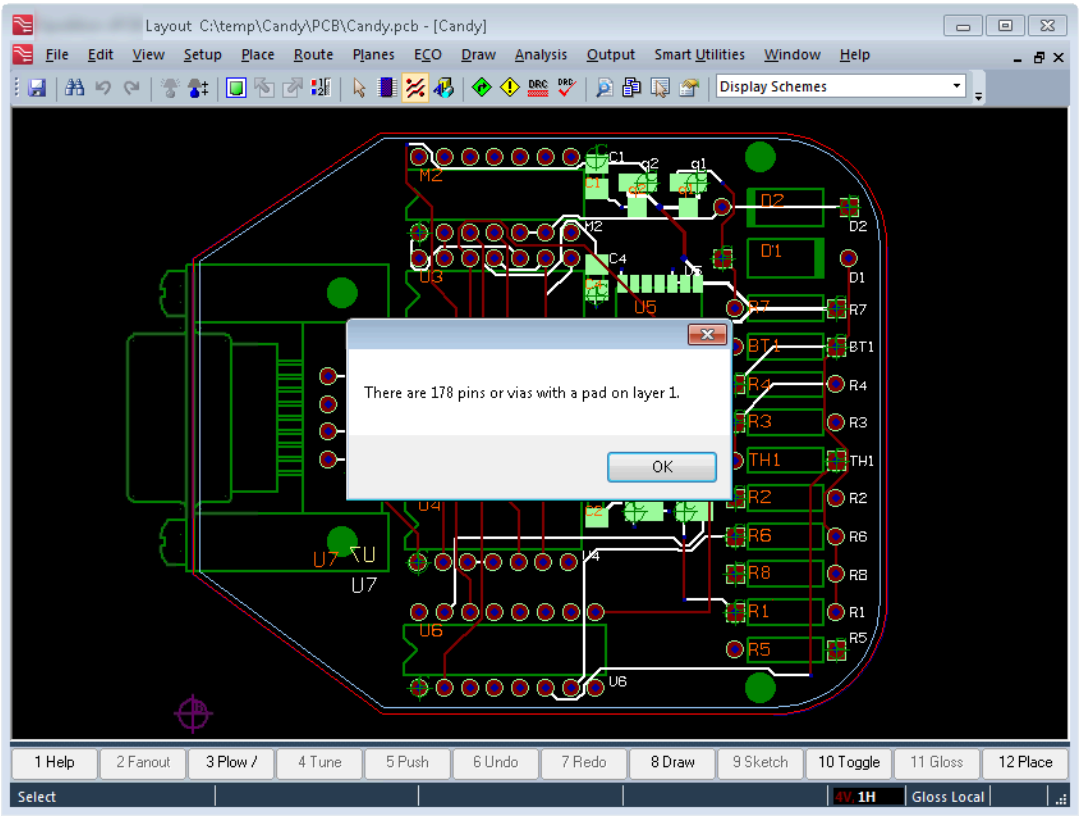
Running the Script

1. Close any instantiations of Xpedition Layout that are currently running, and then launch a new instantiation of this application.

2. Open the *Candy.pcb* design.

3. Drag-and-drop the PadstackObjects.vbs script into the layout design window. Observe the result as illustrated in Figure 118: Result of PadstackObjects.vbs(the actual number depends on the state of the design).

Figure 118. Result of PadstackObjects.vbs



4. Choose **File > Close** to close the design.

Using the Display Control

When using the GUI in Xpedition Layout, the Display Control dialog allows the end-user to select items to display and also the colors to use for the items.

Some aspects of display control are local, while others are global. For example, it is possible to open multiple views of the same design and to display different items (placement outlines, text annotations, etc.) in each view. These display controls may be considered "local" to each view. By comparison, if the end-user changes the color associated with an item type in one view, this change is automatically applied to all views. Thus, these controls are considered "global".

Similarly, in the Automation Interface, there are DisplayControl and GlobalDisplayControl objects, which control local and global display characteristics, respectively.

This section presents three examples that demonstrate the display control.

Exercise 7: Toggling the Display of Placement Outlines
Exercise 8: Changing the Layer Color
Exercise 9: Loading Scheme

Exercise 7: Toggling the Display of Placement Outlines

This script toggles the display of component outlines on and off.

Approach: To use the DisplayControl object to manipulate the display of placement outlines.

Creating the Script

Use the scripting editor of your choice to open the *Skeleton.vbs* script in your *C:\temp* folder, save the script as *TogglePlacementOutlines.vbs*, and then replace the comment on Line 18 with the following code:

```
18  ' Get the display control object.
19  Dim dispCtrlObj
20  Set dispCtrlObj = pcbDocObj.ActiveViewEx.DisplayControl
21
22  ' Get the state of the placement outline display
23  If dispCtrlObj.PartItems(epcbDCPlacementOutlines, epcbSideTop) =
                                                          False Then
24    ' It is off.  Turn it on.
25    dispCtrlObj.PartItems(epcbDCPlacementOutlines, epcbSideTop) = True
26  Else
27    ' It is on.  Turn it off.
28    dispCtrlObj.PartItems(epcbDCPlacementOutlines, epcbSideTop) = False
29  End If
```

On Line 19, the script instantiates a variable dispCtrlObj, which holds a DisplayControl object. Line 20 uses the ActiveViewEx property of the Document object to return the active View object for the design. The same statement uses the DisplayControl property of the View object to return the DisplayControl object and assigns this object to the dispCtrlObj variable.

```
18  ' Get the display control object.
19  Dim dispCtrlObj
20  Set dispCtrlObj = pcbDocObj.ActiveViewEx.DisplayControl
```

There are several different properties associated with the DisplayControl object. These include the GeneralItems, PartItems, UserLayer, and Miscellaneous properties. The PartItems property, which is the focus of this example script, handles the display of component text and graphics.

Lines 23 through 29 toggle the display of the top-side placement outlines on and off. The PartItems property supports two parameters. The first parameter specifies the item type, while the second specifies the board side. In this example, the parameters specify the placement outlines (enumerate epcbDCPlacementOutlines) and the top-side of the board (enumerate epcbSideTop).

```
22  ' Get the state of the placement outline display
23  If dispCtrlObj.PartItems(epcbDCPlacementOutlines, epcbSideTop) =
                                                                False Then
24    ' It is off.  Turn it on.
25    dispCtrlObj.PartItems(epcbDCPlacementOutlines, epcbSideTop) = True
26  Else
27    ' It is on.  Turn it off.
28    dispCtrlObj.PartItems(epcbDCPlacementOutlines, epcbSideTop) = False
29  End If
```

The PartItems property performs both "read" and "write" tasks. The If … Then statement on Line 23, “reads” the PartItems property. Depending on the current state, Lines 25 and 28 use the PartItems property to assign ("write") a new value.

Running the Script

Launch Xpedition Layout, open a design, and try running this script to make sure it behaves as expected.

Note:
This type of script would generally be associated with an accelerator (shortcut) key (for more information refer to “Adding an Accelerator Key” on page 103).

Exercise 8: Changing the Layer Color

This script changes the color of a user layer.

Approach: To use the GlobalDisplayControl object to manipulate colors.

Assumptions: The design contains a user layer named "Test".

Also of Interest: This demonstrates the Utility object.

Creating the Script

Use the scripting editor of your choice to open the *Skeleton.vbs* script in your *C:\temp* folder, save the script as *ChangeUserLayerColor.vbs*, and then replace the comment on Line 18 with the following code:

```
18  ' Get the display control object.
19  Dim dispCtrlObj
20  Set dispCtrlObj = pcbDocObj.ActiveViewEx.DisplayControl
21
22  ' Get the global display control object.
23  Dim dispCtrlGlobalObj
24  Set dispCtrlGlobalObj = dispCtrlObj.Global
25
26  ' Create a new color object
27  Dim clrObj
28  Set clrObj = pcbAppObj.Utility.NewColorPattern
29
30  ' Set the color.
31  clrObj.Red   = 200
32  clrObj.Green = 100
33  clrObj.Blue  = 200
34
35  ' Apply the color to the user layer
36  Set dispCtrlGlobalObj.UserLayerColor("Test") = clrObj
37
38  MsgBox("Color changed for user layer ""Test"".")
```

On Line 19, the script instantiates a variable dispCtrlObj, which holds a DisplayControl object. Line 20 uses the ActiveViewEx property of the Document object to return the active View object for the design. The same statement uses the DisplayControl property of the View object to return the DisplayControl object and assigns this object to the dispCtrlObj variable.

```
18  ' Get the display control object.
19  Dim dispCtrlObj
20  Set dispCtrlObj = pcbDocObj.ActiveViewEx.DisplayControl
```

Line 23 instantiates a variable dispCtrlGlobalObj, which holds a GlobalDisplayControl object. Line 24 uses the Global property of the DisplayControl object (stored in the dispCtrlObj variable) to access the GlobalDisplayControl object and assigns it to the dispCtrlGlobalObj variable.

```
22  ' Get the global display control object.
23  Dim dispCtrlGlobalObj
24  Set dispCtrlGlobalObj = displCtrlObj.Global
```

To associate a new color with a display item, it is necessary to create a new ColorPattern object. The *ColorPattern* object has RGB (reg, green, and blue) properties, each of which is defined by an integer (or an integer variable) in the range 0 to 255.

The Utility object creates new objects or empty collections as required. Line 27 instantiates a variable clrObj, which holds the new ColorPattern object. Line 28 uses the NewColorPattern property of the Utility object to create a new ColorPattern object and assigns it to the clrObj variable.

```
26  ' Create a new color object
27  Dim clrObj
28  Set clrObj = pcbAppObj.Utility.NewColorPattern
```

Lines 31 through 33 assign RGB values to the Red, Green, and Blue properties of the ColorPattern object stored in the clrObj variable.

```
30  ' Set the color.
31  clrObj.Red   = 200
32  clrObj.Green = 100
33  clrObj.Blue  = 200
```

These values correspond to a shade of mauve/purple as shown below:

Line 36 uses the UserLayerColor property of the GlobalDisplayControl object (in the dispCtrlGlobalObj variable) to assign the new ColorPattern object (in the clrObj variable) to the user layer "Test". Note that the Set keyword is used here because an *object* is assigned to the UserLayerColor property.

```
35  ' Apply the color to the user layer
36  Set dispCtrlGlobalObj.UserLayerColor("Test") = clrObj
```

Finally, Line 38 uses a MsgBox() to inform the end-user of the result.

```
38  MsgBox("Color changed for user layer ""Test"".")
```

Running the script

1. Close any instantiations of Xpedition Layout that are currently running, and then launch a new instantiation of this application.

2. Open the Candy.pcb design. (This design already contains a user layer named “Test”)

3. Choose **View > Display Control** to open the Display Control dialog.

4. Select the **General** tab, then ensure that the checkbox next to the user layer named "Test" is selected.

5. Drag-and-drop the ChangeUserLayerColor.vbs script into the design window and assure that the script works as expected.

Exercise 9: Loading Scheme

This script applies a predefined display scheme to a design view.

Approach: To use the DisplayControl object to load a predefined display scheme.

Assumptions: A local display scheme named PlacementOutlines is already defined for the design.

Creating the Script

Use the scripting editor of your choice to open the *Skeleton.vbs* script in your *C:\temp* folder, save the script as *LoadScheme.vbs*, and then replace the comment on Line 18 with the following code:

```
18  ' Get the display control object.
19  Dim dispCtrlObj
20  Set dispCtrlObj = pcbDocObj.ActiveViewEx.DisplayControl
21
22  ' Load a display scheme
23  dispCtrlObj.LoadScheme("Loc: PlacementOutlines")
```

On Line 19, the script instantiates a variable dispCtrlObj, which holds a DisplayControl object. Line 20 uses the ActiveViewEx property of the Document object to return the active View object of the design. The same statement uses the DisplayControl property of this View object to return the DisplayControl object and assigns this object to the dispCtrlObj variable.

```
18  ' Get the display control object.
19  Dim dispCtrlObj
20  Set dispCtrlObj = pcbDocObj.ActiveViewEx.DisplayControl
```

Line 23 uses the LoadScheme method of the DisplayControl object (the dispCtrlObj variable) and passes the method the new scheme name as a string.

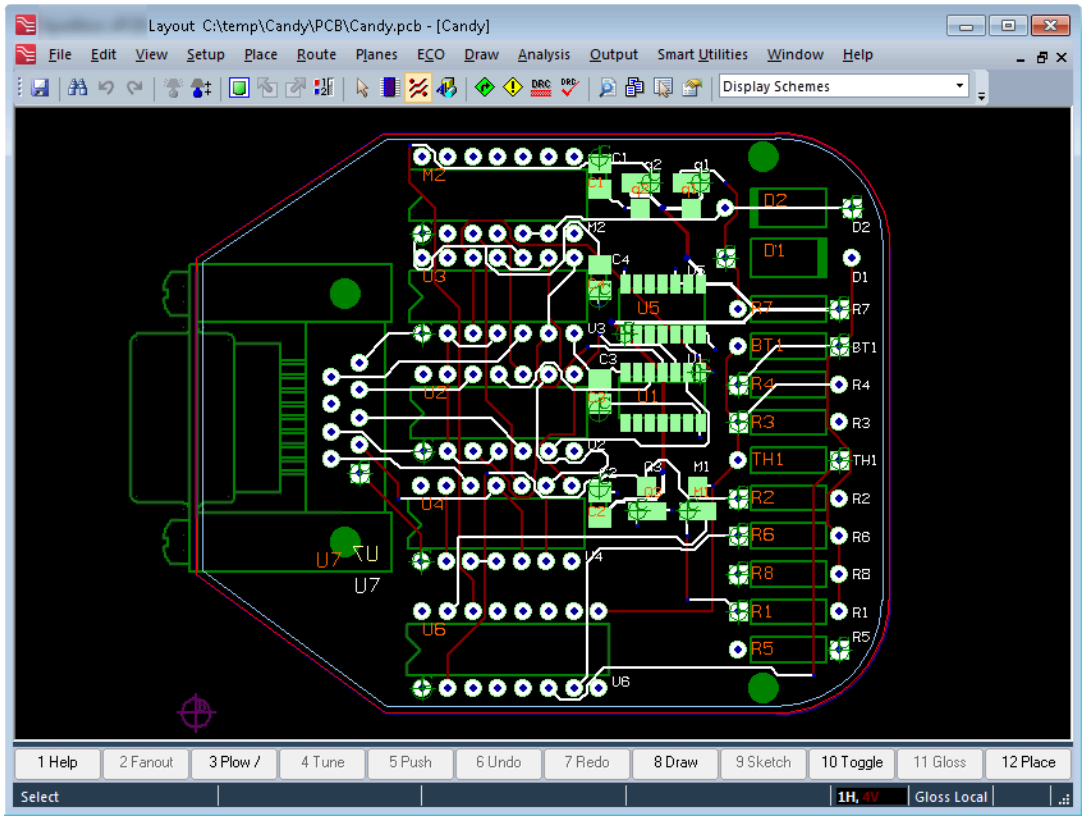
```
22  ' Load a display scheme
23  dispCtrlObj.LoadScheme("Loc: PlacementOutlines")
```

Running the Script

1. Close any instantiations of Xpedition Layout that are currently running, and then launch a new instantiation of this application.

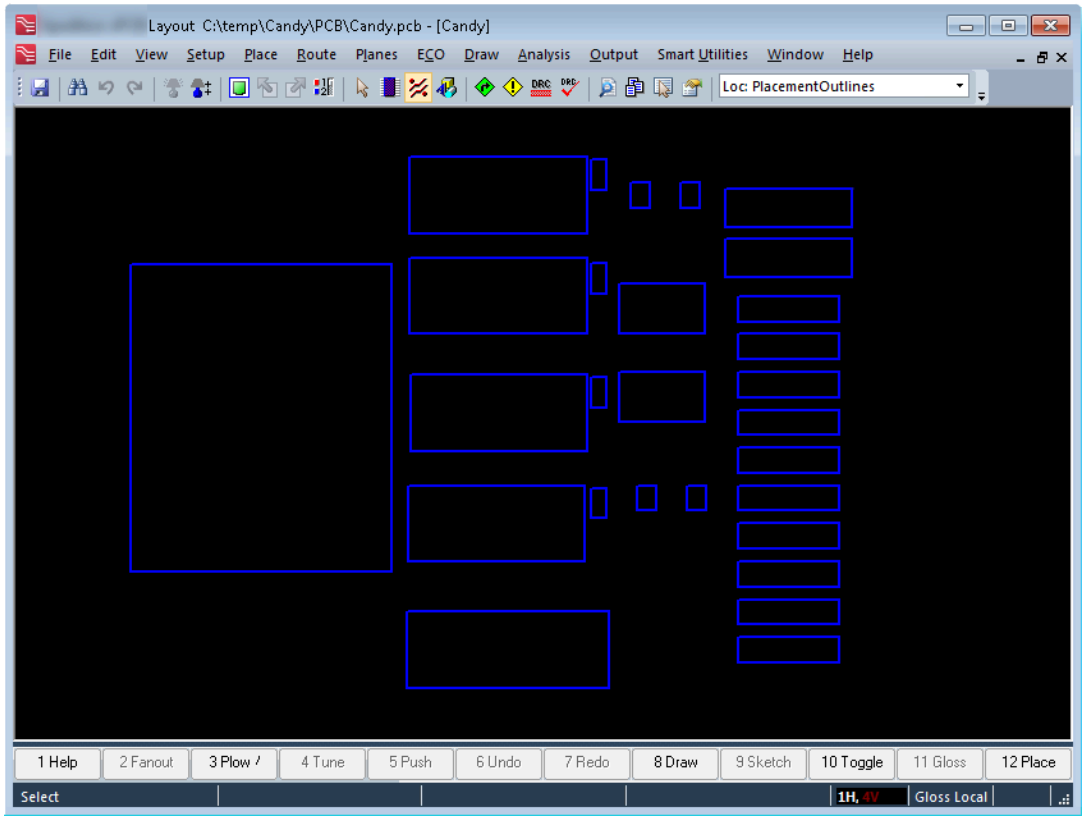
2. Open the Candy.pcb design. (Figure 119: Candy Design With Loc: Original Scheme)

Figure 119. Candy Design With Loc: Original Scheme



3. Drag-and-drop the *LoadScheme.vbs* script into the layout design window. Observe the result, which should be similar to Figure 120: Result of LoadScheme.vbs.

Figure 120. Result of LoadScheme.vbs



4. Change the display scheme back to **Loc: Original** using the Display Scheme drop-down on the toolbar.

5. Choose **File > Close** to close the design, do *not* save any changes you made as part of running this script.

Running Menu Commands

Generally speaking, one writes scripts that use the Automation interface to directly query designs for information and to manipulate objects. In some cases, however, it is valid to create scripts that run existing menu commands. This section demonstrates two different examples of this capability.

Exercise 10: Running the Fit Board Command
Exercise 11: Running the Route Command

Exercise 10: Running the Fit Board Command

This script runs the pre-existing **View > Fit Board** command.

Approach: To use the ProcessCommand method of the Gui property to run the command.

Creating the Script

Use the scripting editor of your choice to open the *Skeleton.vbs* script in your *C:\temp* folder, save the script as *FitBoard.vbs*, and then replace the comment on Line 18 with the following line of code:

```
18  pcbAppObj.Gui.ProcessCommand("View->Fit Board")
```

Line 18 uses the ProcessCommand method of the Gui property to run the **View > Fit Board** command. To do this, pass a string parameter of "View->Fit Board". Note the "->" characters and that there are no spaces before or after these characters. A space is specified in the "Fit Board" portion as this is how the command appears in the GUI.

This example uses the string "View->Fit Board" to explicitly define the full menu path. However, since there is only one **Fit Board** menu item in the interface, you could simply pass the string "Fit Board" as the parameter to Line 18 as follows:

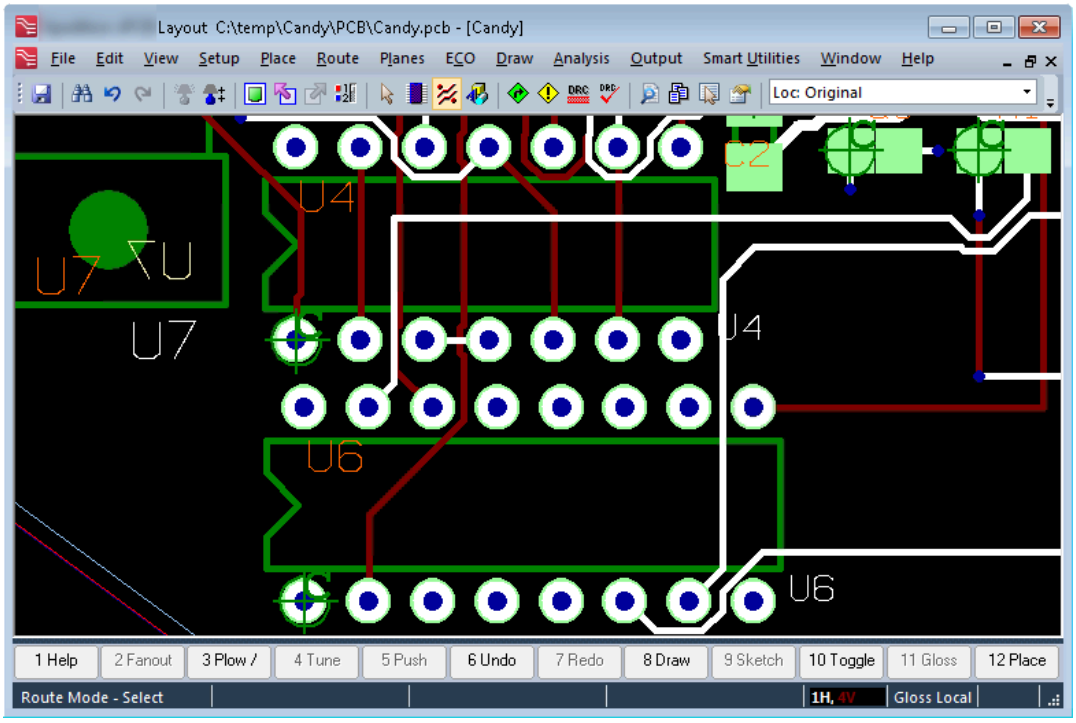
```
18  pcbAppObj.Gui.ProcessCommand("Fit Board")
```

Running the Script

1. Close any instantiations of Xpedition Layout that are currently running, and then launch a new instantiation of this application.

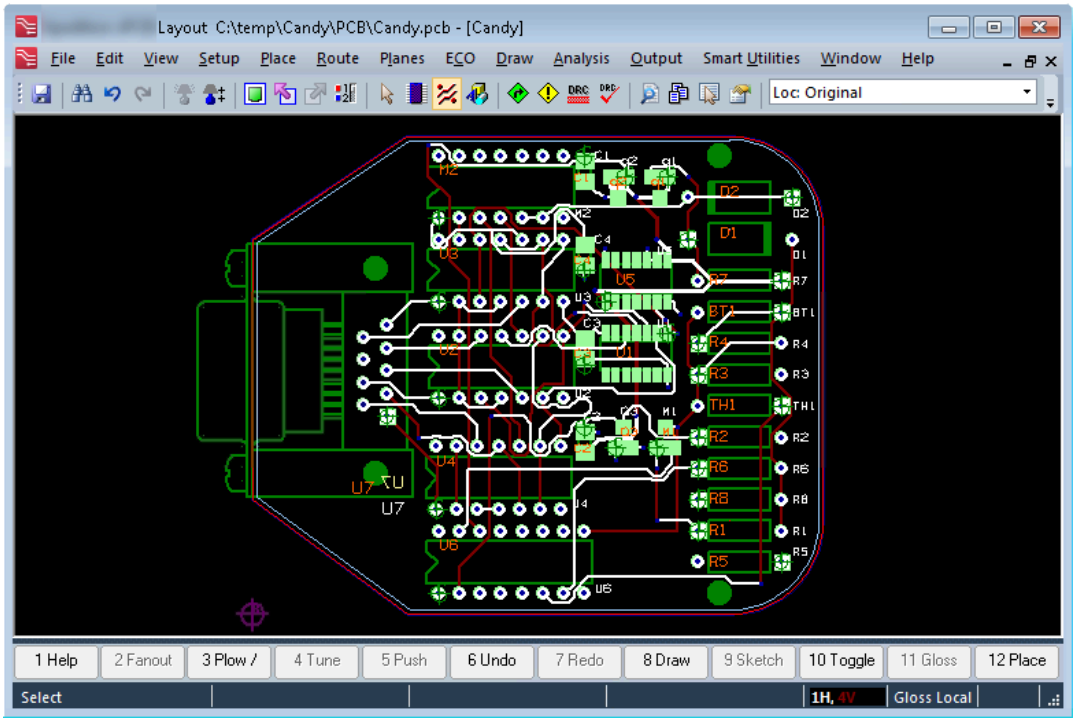
2. Open the Candy.pcb design. Zoom in on any part of the design as illustrated in Figure 121: Zoom in On the Design.

Figure 121. Zoom in On the Design



3. Drag-and-drop the FitBoard.vbs script into the layout design window. Observe the result as illustrated in Figure 122: Result of Running FitBoard.vbs.

Figure 122. Result of Running FitBoard.vbs



4. Choose **File > Close** to close the design.

Exercise 11: Running the Route Command

This script routes any open netlines (these open netlines are also referred to as "From-Tos").

Approach: To individually select each open netline and route it using the **Route > Interactive > Route** command.

Creating the Script

Use the scripting editor of your choice to open the *Skeleton.vbs* script in your *C:\temp* folder, save the script as *RouteNetlines.vbs*, and then replace the comment on Line 18 with the following lines of code:

```
18  ' Get the collection of netlines
19  Dim fromToColl
20  Dim fromToObj
21  Set fromToColl = pcbDocObj.FromTos
22
23  For Each fromToObj In fromToColl
24     ' Select the netline.
25     fromToObj.Selected = True
26     ' Run the Route command from the menu.
27     pcbAppObj.Gui.ProcessCommand("Route->Interactive->Route")
28     ' Unselect everything for the next iteration.
```

```
29     pcbDocObj.UnselectAll
30  Next
31
32  ' Tell the user what has happened.
33  If fromToColl.Count > 0 Then
34     ' Calculate how many we were able to route
35     Dim numRoutedInt: numRoutedInt = fromToColl.Count –
                                                pcbDocObj.FromTos.Count
36     Call pcbAppObj.Gui.StatusBarText(numRoutedInt & _
37                          " netlines have been routed.", epcbStatusField1)
38  Else
39     Call pcbAppObj.Gui.StatusBarText("No open netlines found.", _
40                                                  epcbStatusFieldError)
41  End If
```

On Line 19, the script instantiates a variable fromToColl, which holds a collection of FromTo objects. Line 20 instantiates a variable fromToObj, which represents a single FromTo object. Line 21 uses the FromTo property of the Document object to return a collection of FromTo objects and assigns them to the fromToColl variable.

```
18  ' Get the collection of netlines
19  Dim fromToColl
20  Dim fromToObj
21  Set fromToColl = pcbDocObj.FromTos
```

Lines 23 through 30 use a For Each … Next control loop to iterate through all of the FromTo objects in the collection. Line 25 uses the Selected property to select the current FromTo object (this highlights the object in the GUI). Line 17 runs the **Route > Interactive > Route** menu command by passing the string "Route->Interactive->Route" as a parameter to the ProcessCommand method. Line 29 unselects everything to clean up before the next iteration.

```
23  For Each fromToObj In fromToColl
24     ' Select the netline.
25     fromToObj.Selected = True
26     ' Run the Route command from the menu.
27     pcbAppObj.Gui.ProcessCommand("Route->Interactive->Route")
28     ' Unselect everything for the next iteration.
29     pcbDocObj.UnselectAll
30  Next
```

Finally, Lines 33 through 41 inform the end-user of the result by displaying messages in the status bar. Observe the statement spanning Lines 39 and 40; this displays a message in the status bar's Error field and has a yellow background.

```
32  ' Tell the user what has happened.
33  If fromToColl.Count > 0 Then
34     ' Calculate how many we were able to route
35     Dim numRoutedInt: numRoutedInt = fromToColl.Count –
                                                  pcbDocObj.FromTos.Count
36     Call pcbAppObj.Gui.StatusBarText(numRoutedInt & _
37                            " netlines have been routed.",
 epcbStatusField1)
38  Else
```

```
39      Call pcbAppObj.Gui.StatusBarText("No open netlines found.", _
40
 epcbStatusFieldError)
41   End If
```

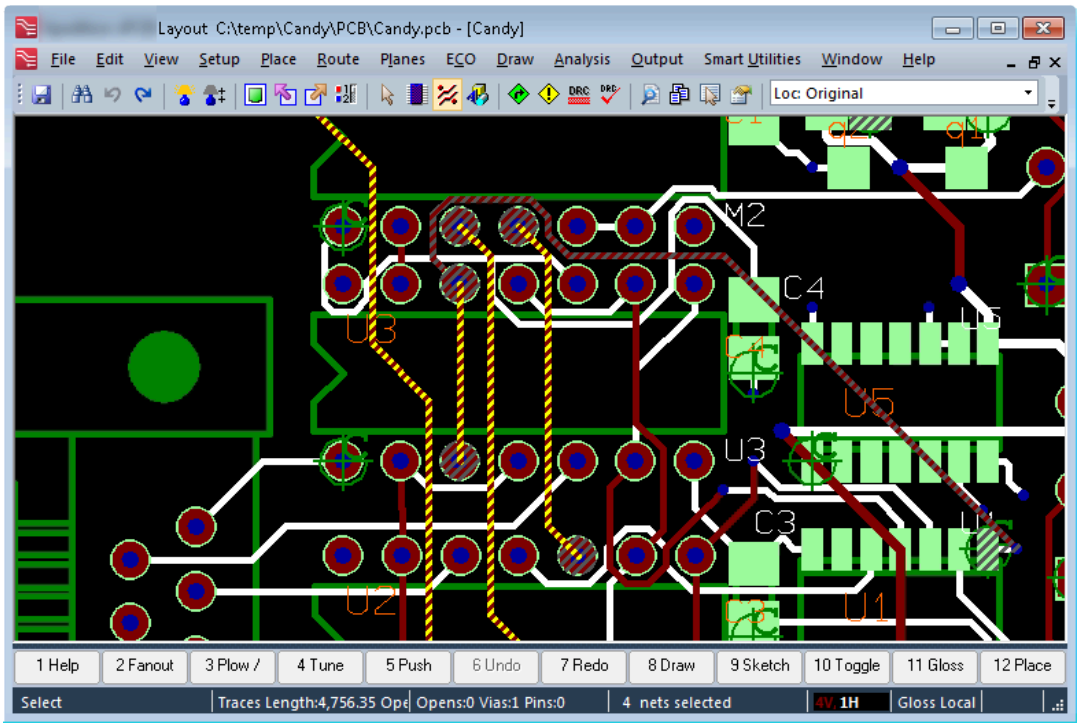
Note:
Line 33 uses the Count property on the collection of FromTo objects. Even though some of the FromTo objects are routed, the original FromTo collection still exists. You can still perform certain actions on the collection itself, such as using its Count property. However, you cannot perform any methods or properties on objects inside the collection. This is because those objects are no longer valid as the *FromTo* they represent, no longer exists.

Running the Script

1. Close any instantiations of Xpedition Layout that are currently running, and then launch a new instantiation of this application.

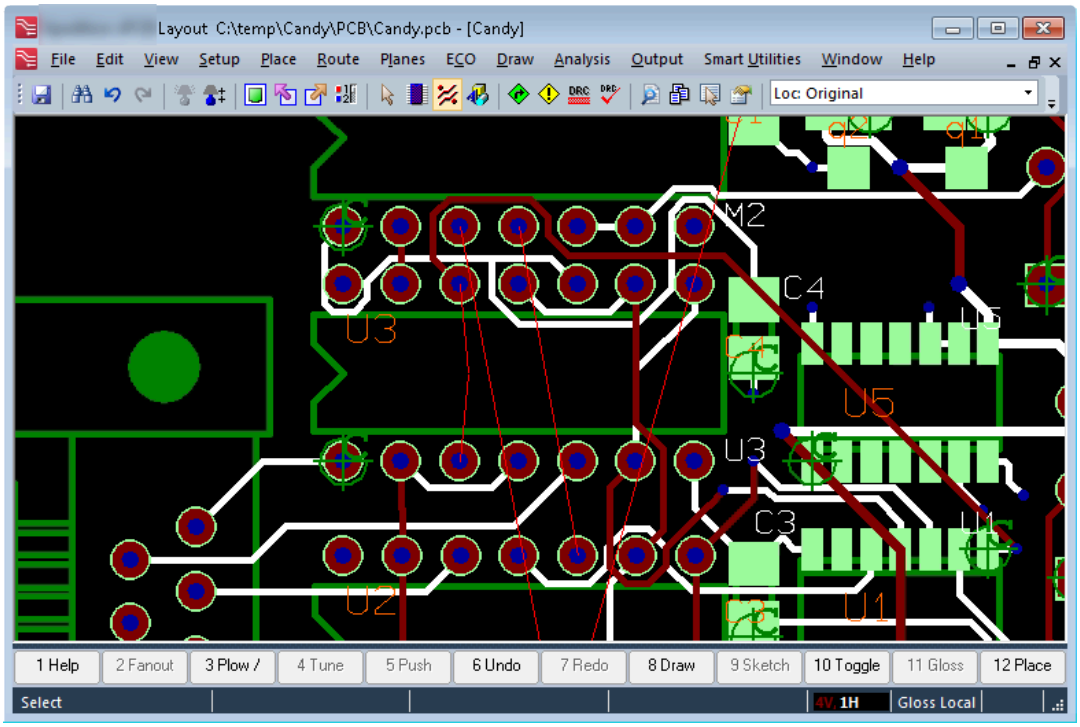
2. Open the *Candy.pcb* design, zoom in on component U3, and select the four traces crossing the middle of this component as illustrated in Figure 123: Zoom in on Component U3.

Figure 123. Zoom in on Component U3



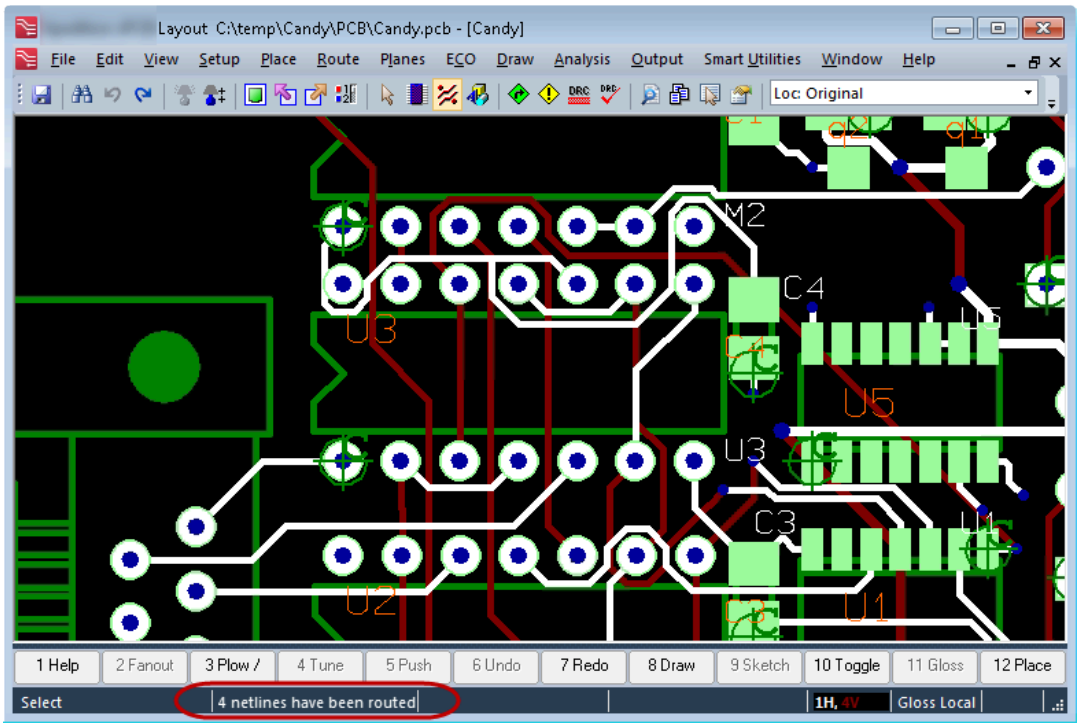
3. Delete these traces; this results in the design appearing as shown in Figure 124: Script and Delete the Four Traces.

Figure 124. Script and Delete the Four Traces



4. Drag-and-drop the RouteNetlines.vbs script on the design window. Observe that these four "From-Tos" route. A message displays in the status bar as illustrated in Figure 125: Script Routes the Four From-Tos.

Figure 125. Script Routes the Four From-Tos



Creating Objects with Geometries

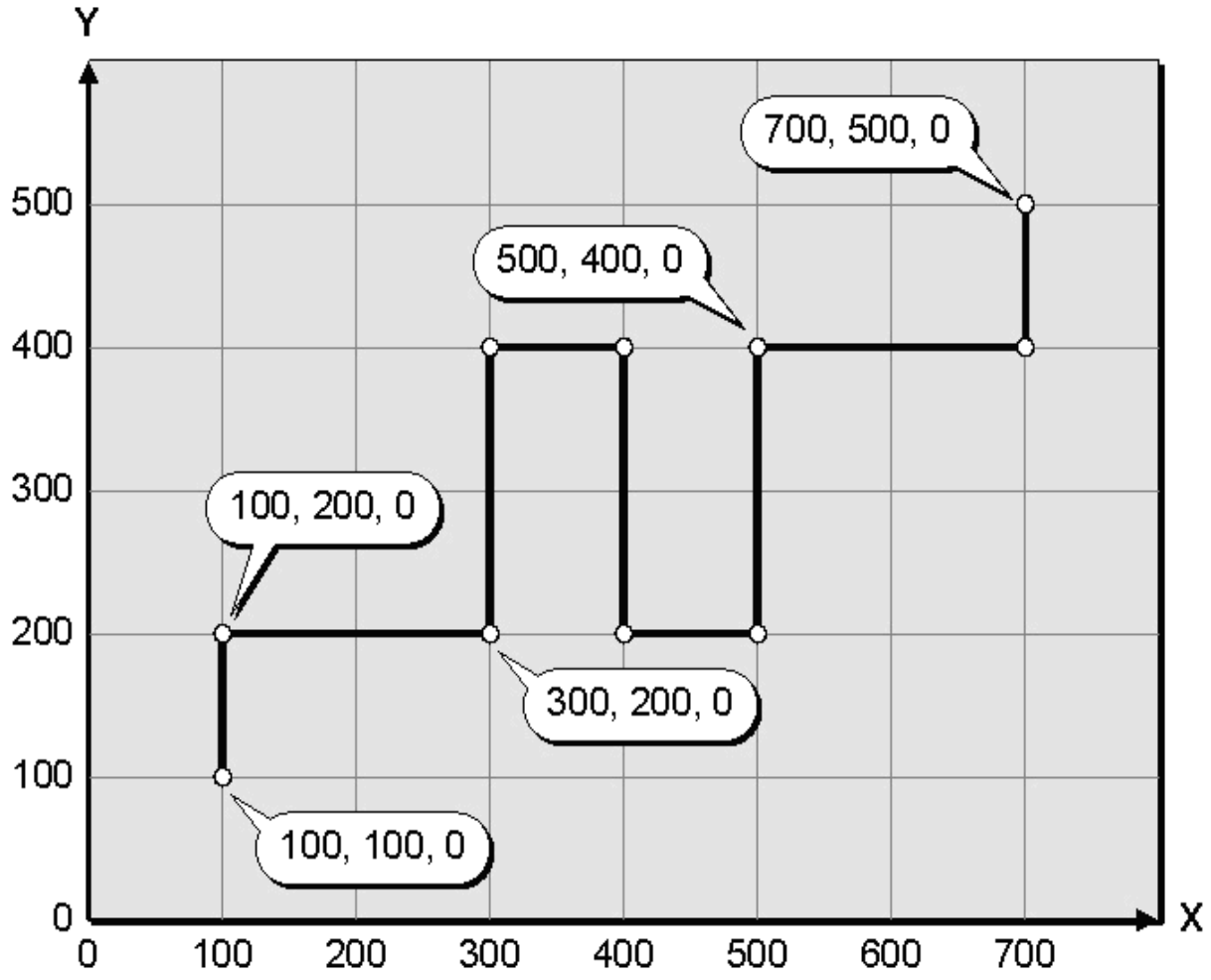
Some design objects have geometries associated with them; for example, traces, placement outlines, and the board outline. Geometries are specified as a list of vertices. Such a list is referred to as a "points array".

Each point in a points array is defined by three parameters. The first two parameters specify the X and Y coordinates of the point, while the third parameter defines the radius (R) in the case of circles or arc segments. Consider a geometry formed from a series of straight line segments as illustrated in Figure 126: Geometry Defined as Straight Line Segments .

The X/Y origin of the board (at coordinates 0,0) is in the lower left-hand corner. All X/Y values are measured from this point. The geometry starts with the point (X,Y,R = 100, 100, 0), this graphical element is described as a series of points as follows:

```
X,Y,R  =  100,  100,  0
X,Y,R  =  100,  200,  0
X,Y,R  =  300,  200,  0
X,Y,R  =  300,  400,  0
X,Y,R  =  400,  400,  0
X,Y,R  =  400,  200,  0
X,Y,R  =  500,  200,  0
X,Y,R  =  500,  400,  0
X,Y,R  =  700,  400,  0
X,Y,R  =  700,  500,  0
```

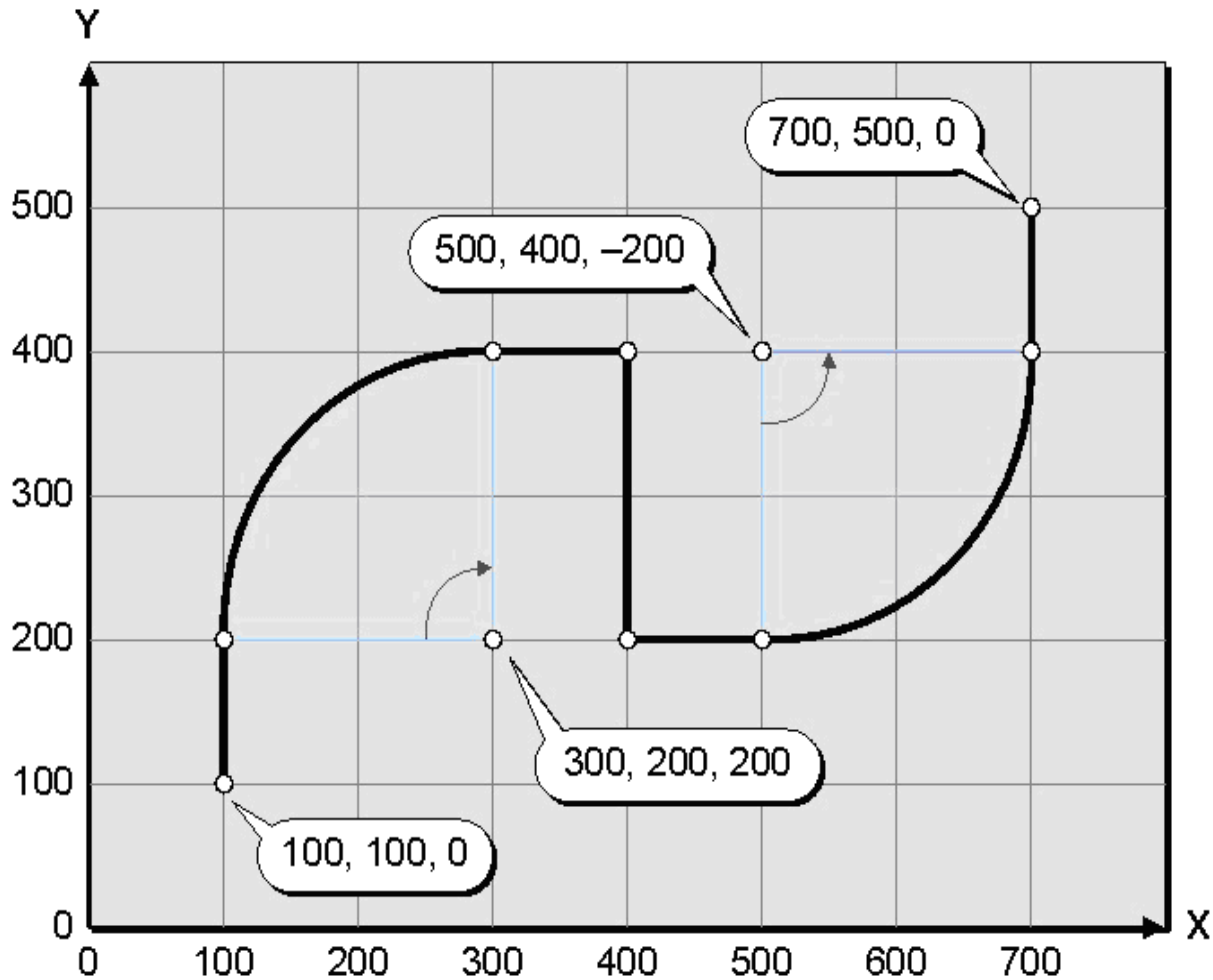
Figure 126. Geometry Defined as Straight Line Segments



In the case of geometries containing arcs, these elements are formed from three points: a "begin" point on the circumference (with R=0), the center of the arc (with R=radius), and an "end" point on the circumference (with R=0). For a circle, the "End" point has the same X/Y coordinates as the "Start" point.

Consider the geometry illustrated in Figure 127: Geometry Defined by Straight Lines and Arcs .

Figure 127. Geometry Defined by Straight Lines and Arcs



The geometry starts with the point, (X,Y,R = 100, 100, 0), this shape is described as a series of points as follows:

```
X,Y,R  =  100,  100,  0
X,Y,R  =  100,  200,  0
X,Y,R  =  300,  200,  200
X,Y,R  =  300,  400,  0
X,Y,R  =  400,  400,  0
X,Y,R  =  400,  200,  0
X,Y,R  =  500,  200,  0
X,Y,R  =  500,  400,  -200
X,Y,R  =  700,  400,  0
X,Y,R  =  700,  500,  0
```

The two geometries illustrated in Figure 126: Geometry Defined as Straight Line Segments and Figure 127: Geometry Defined by Straight Lines and Arcs contain exactly the same number of points and these points are defined by identical X/Y values. The only difference is that two of the points in Figure 127: Geometry Defined by Straight Lines and Arcs have R values. A positive R value results in a "clockwise path" for the arc or circle, while a negative R value results in an "counter-clockwise path".

Note:
Closed geometries, such as board outlines, have identical values in the first and last point of the points array.

Exercise 12: Adding Traces
Exercise 13: Adding User Layer Graphics

Exercise 12: Adding Traces

This script adds a trace between two pins.

Approach: To use the PutTrace method to add a trace to the board.

Assumptions: The design (Candy.pcb) has a missing trace.

Also of Interest: This script uses specific units of measure and uses transactions to control DRC.

Creating the Script

Use the scripting editor of your choice to open the *Skeleton.vbs* script in your *C:\temp* folder, save the script as *PutTraceNoTransac.vbs*, and then replace the comment on Line 18 with the following lines of code:

```
18  ' Set the current unit value so that this script
19  ' will still run correctly if the units are changed
20  ' in the design
21  pcbDocObj.CurrentUnit = epcbUnitMils
22
23  ' Create/Get all parameters needed to add a trace
24
25  ' Create a points array
26  Dim pntsArr(2, 1)
27  pntsArr(0,0) = 1200 : pntsArr(1,0) = 1800 : pntsArr(2,0) = 0
28  pntsArr(0,1) = 1400 : pntsArr(1,1) = 1800 : pntsArr(2,1) = 0
29
30  ' Get the number of points in the array
31  Dim numPntsInt : numPntsInt = UBound(pntsArr, 2) + 1
32
33  ' Get the Net object for the net the trace will be on.
34  Dim netObj: Set netObj = pcbDocObj.FindNet("change")
35
36  ' Provide a width
37  Dim widthReal : widthReal = 10
38
39  ' Don't add this trace to any component
40  Dim cmpObj : Set cmpObj = Nothing
41
42  ' Specify the layer to put the trace on
43  Dim layerInt : layerInt = 1
44
```

```
45  ' Put a new trace in the document
46  Call pcbDocObj.PutTrace(layerInt, netObj, widthReal, _
47                          numPntsInt, pntsArr, cmpObj, _
48                          epcbAnchorNone,epcbUnitCurrent)
```

The X/Y locations of vertices are represented and stored in "database units" (these are largely invisible to the end-user).

When a script acquires a *Document* object, it has an assigned CurrentUnit property value. This value reflects the current design units as presented in the GUI. The units type enumerates and their associated units/values are as follows:

Integer	Enumerate	Associated Units/Values
0	epcbUnitCurrent	Whatever is the current value
1	epcbUnitDatabase	Database units (beyond the scope of this document)
2	epcbUnitMils	Thousandths of an inch or mils
3	epcbUnitInch	Inches
4	epcbUnitMM	Millimeters
5	epcbUnitUM	Micrometers

If the value associated with the CurrentUnit property is changed in the script, the change does *not* change anything in the design. The property is local to the Document object. For example, if the CurrentUnit property is set to epcbUnitMM, any methods and properties accessed on your Document object or on any object returned from that Document object will return values in millimeters. Every method and property that returns or accepts coordinates or lengths has a Unit parameter. The *Unit* parameter allows you to specify the units for that method or property on an individual basis. The default value for the Unit parameter is epcbUnitCurrent, which is epcbUnitMM in this example.

In this script example, the vertices for the geometric shape are hard-coded, and these values are in mils. The design units in the *Candy.pcb* design are also in mils; however, the end-user may change this prior to running the script. So, Line 21 assigns the epcbUnitMils enumerate to the CurrentUnit property associated with the Document object.

```
18  ' Set the current unit value so that this script
19  ' will still run correctly if the units are changed
20  ' in the design
21  pcbDocObj.CurrentUnit = epcbUnitMils
```

Lines 26 through 28 define the points array. Line 26 instantiates an array that is three columns "wide" by two rows "deep". (Array dimensions start at 0 and is discussed in “VBScript Primer” on page 27.)

```
25  ' Create a points array
26  Dim pntsArr(2, 1)
27  pntsArr(0,0) = 1200 : pntsArr(1,0) = 1800 : pntsArr(2,0) = 0
28  pntsArr(0,1) = 1400 : pntsArr(1,1) = 1800 : pntsArr(2,1) = 0
```

The points array consists of two points that define a vertical line; the first point has X and Y values of 1200 and 1800 respectively; the second point has X and Y values of 1400 and 1800; and both points have R values of 0.

Lines 31 through 43 define the values of the parameters that the *putTrace* method requires. Line 31 instantiates a variable numPntsInt, which stores the number of vertices in the points array.

```
30  ' Get the number of points in the array
31  Dim numPntsInt : numPntsInt = UBound(pntsArr, 2) + 1
```

Line 31 uses the UBound function to determine the number of vertices; the value returned by the UBound function increments by 1 and assigned to the numPntsInt variable. The first parameter to the *UBound* function is the name of the points array, the second parameter is the dimension of interest. The points array instantiated on Line 26 uses the statement Dim pntsArr(2, 1). The second dimension has a value of 1, but the array dimensions starts at zero, so in order to determine the number of vertices, add a value of '1' to the value of this second dimension.

Next, return the Net object to which the Trace object is added. In this example, the trace is part of the “change” net. So, Line 34 instantiates a variable netObj and then uses the FindNet method to find the net named "change" and assigns this net to the netObj variable.

```
33  ' Get the Net object for the net the trace will be on.
34  Dim netObj: Set netObj = pcbDocObj.FindNet("change")
```

Line 36 instantiates a variable widthReal in which to hold the width of the trace, and assigns a value of 10 to this variable. (This is 10 mils, because the units associated with this script is mils).

```
36  ' Provide a width
37  Dim widthReal : widthReal = 10
```

The new trace is not associated with any particular component in this example. So, Line 40 instantiates a variable cmpObj in which to hold a Component object and assigns a value of Nothing to this variable.

```
39  ' Don't add this trace to any component
40  Dim cmpObj : Set cmpObj = Nothing
```

Finally, Line 43 instantiates a variable layerInt, which holds the layer on which the trace is created, and assigns a value of 1.

```
42  ' Specify the layer to put the trace on
43  Dim layerInt : layerInt = 1
```

Lines 46 through 48 contain a single statement that uses the PutTrace method to add a new trace to the design.

```
45  ' Put a new trace in the document
46  Call pcbDocObj.PutTrace(layerInt, netObj, widthReal, _
47                          numPntsInt, pntsArr, cmpObj, _
48                          epcbAnchorNone, epcbUnitCurrent)
```

The *PutTrace* method requires parameters as follows:

Parameter	Description
1	Target layer for the trace

Parameter	Description
2	Net object the trace is part of
3	Width of the trace
4	Number of vertices in the points array
5	The points array itself
6	The component object this trace is part of
7	An enumerate defining whether the trace is fixed or locked
8	An enumerate defining the units to use

Note that Lines 30 through 43 are defined only for clarity. Also note that the enumerates assigned to the last two parameters are the default values associated with these parameters. This means that you can comment out Lines 30 through 43 and replace the statement on Lines 46 through 48 with the following:

```
45  ' Put a new trace in the document
46  Call pcbDocObj.PutTrace(1, pcbDocObj.FindNet("change"), 10, _
47                          UBound(pntsArr, 2) + 1, pntsArr, _
48                          Nothing)
```

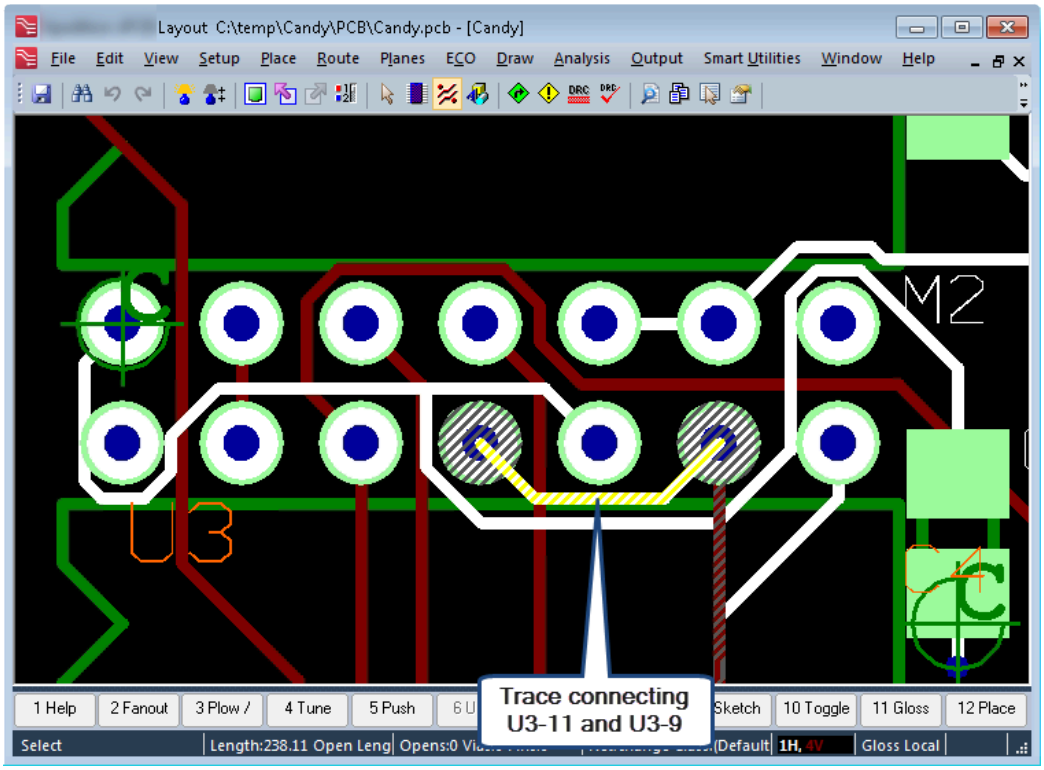
Running the Script

1. Close any instantiations of Xpedition Layout that are currently running, and then launch a new instantiation of this application.

2. Open the *Candy.pcb* design and zoom in on component U3.

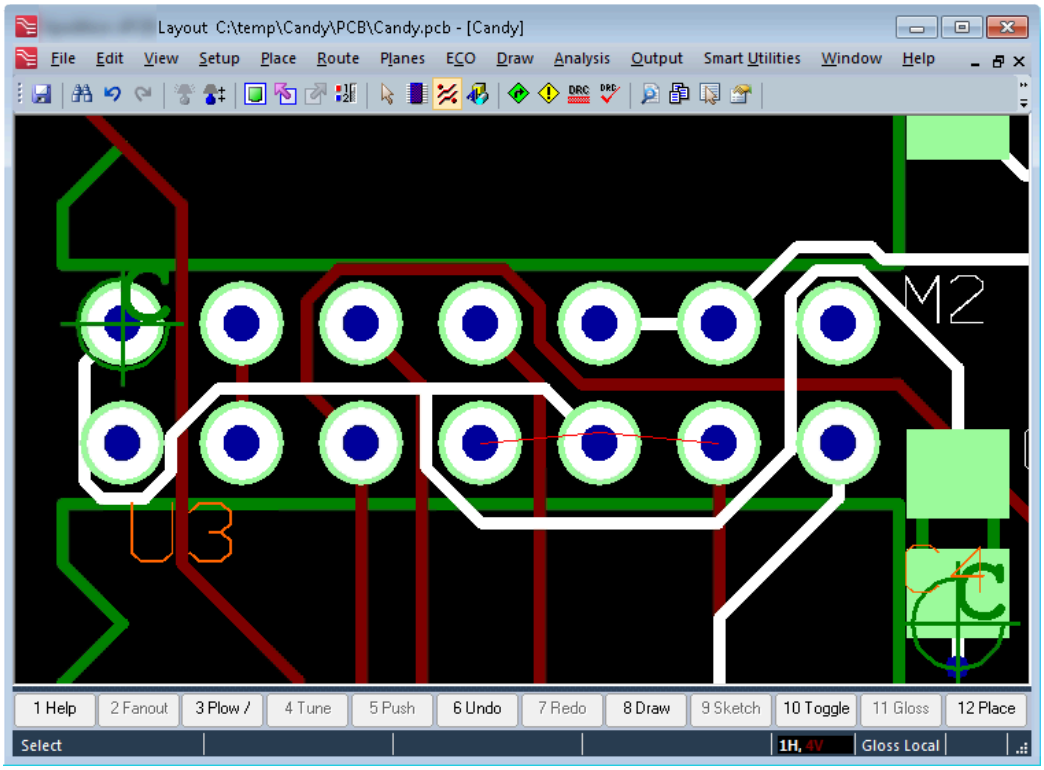
3. While in the Route mode, double-click the trace connecting pins U3-11 and U3-9 as illustrated in Figure 128: Select the Trace Connecting pins U3-11 and U3-9. Note that pin U3-11 has X/Y coordinates of 1200 and 1800, respectively; similarly, pin U3-9 has X/Y coordinates of 1400 and 1800, respectively. These units, while are in mils, match those in the points array.

Figure 128. Select the Trace Connecting pins U3-11 and U3-9



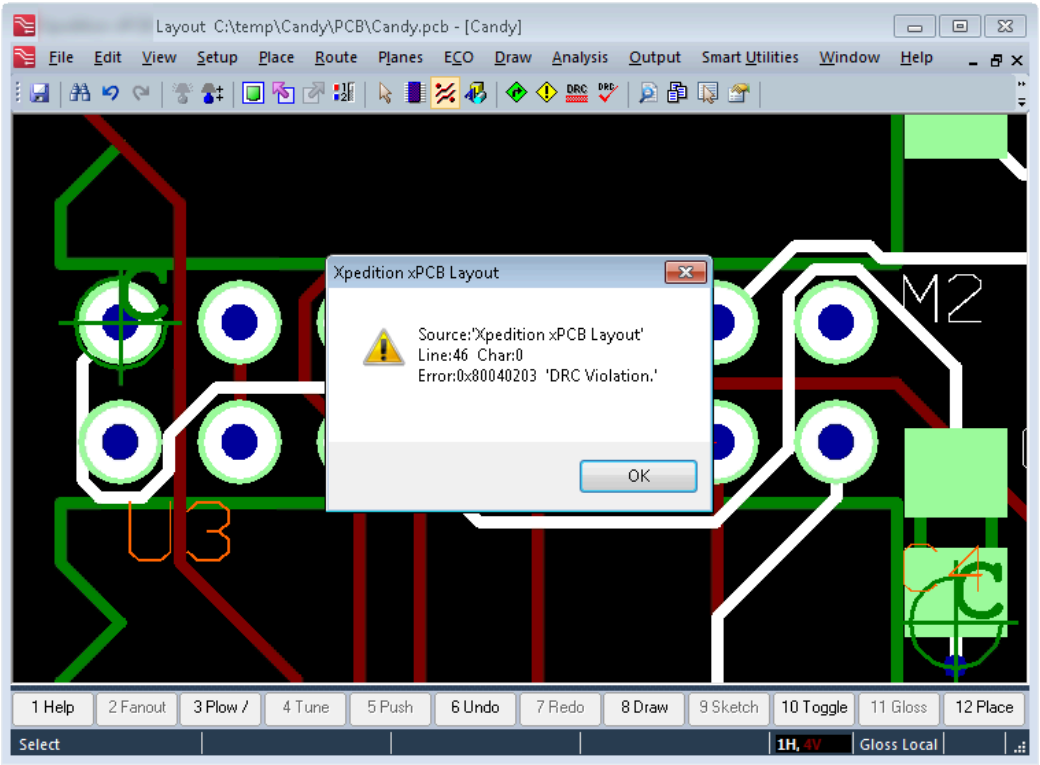
4. Delete this trace. The results appear as shown in Figure 129: Delete Trace Connecting Pins U3-11 and U3-9.

Figure 129. Delete Trace Connecting Pins U3-11 and U3-9



5. Drag-and-drop the *PutTraceNoTransac.vbs* script on the design window. Observe the resulting error message as illustrated in Figure 130: PutTraceNoTransac.vbs Script Results in an Error.

Figure 130. PutTraceNoTransac.vbs Script Results in an Error



The script does not create the new trace because there is a DRC fail. Figure 128: Select the Trace Connecting pins U3-11 and U3-9 shows the source of the problem. The script is attempting to route a trace directly between pins U3-11 and U3-9; this results in the trace passing over pin U3-10, which causes a DRC failure.

The way to address the problem is to use a transaction. Use the scripting editor of your choice to open the *PutTraceNoTransac.vbs* and save it as *PutTraceTransac.vbs.* Replace the original Lines 45 through 48 with the following lines of code:

```
45  ' Start a transaction to set the drc mode
46  If pcbDocObj.TransactionStart(epcbDrcModeResolve) Then
47    ' Put a new trace in the document
48    Call pcbDocObj.PutTrace(layerInt, netObj, widthReal, _
49                            numPntsInt, pntsArr, cmpObj, _
50                            epcbAnchorNone, epcbUnitCurrent)
51
52    ' End the transaction
53    pcbDocObj.TransactionEnd
54  Else
```

```
55    ' Let the user know the script failed.
56    Call pcbAppObj.Gui.StatusBarText("Unable to add trace", _
57                                          epcbStatusField1)
58  End If
```

Line 46 uses the TransactionStart method to start a transaction. The epcbDrcModeResolve enumerate passes as a parameter to specify the level of DRC this transaction requires. In this case, the system is resolving any DRC errors if possible.

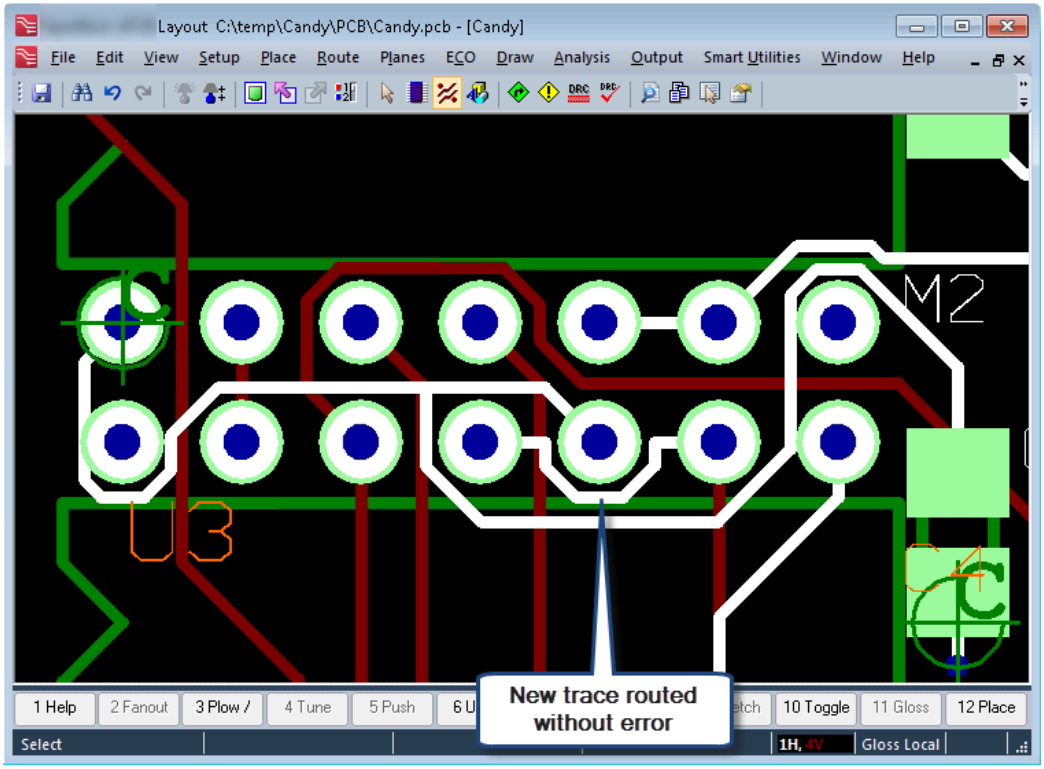
Lines 48 through 50 use the PutTrace method in exactly the same manner as for the original script. Line 53 ends the transaction.

The Else and associated statement on Lines 54 through 57 informs the end-user if the system fails to initiate the transaction for any reason.

Tip
The If … Then statement "wrapping" the transaction on Line 46 is not strictly necessary. You can initiate a transaction directly (this technique is demonstrated in the Exercise 2: Traversing Components script). Using the If … Then statement (this example) is good programming practice in case the database is currently being written by something else (the end-user, another script, etc.).

1. Drag-and-drop the *PutTraceTransac.vbs* script on the design window. Observe that the trace now routes as illustrated in Figure 131: Script Runs Without Error.

Figure 131. Script Runs Without Error



Note:
There are several DRC modes. It is recommended that you experiment with these modes in this script and observe the results.

Exercise 13: Adding User Layer Graphics

This script adds a graphical element to a user layer.

Approach: To use the PutUserLayerGfx method to add a graphical element to a user layer on the board.

Assumptions: The design (*Candy.pcb*) contains a user layer named Test.

Also of Interest: The graphical element this script creates includes two arcs.

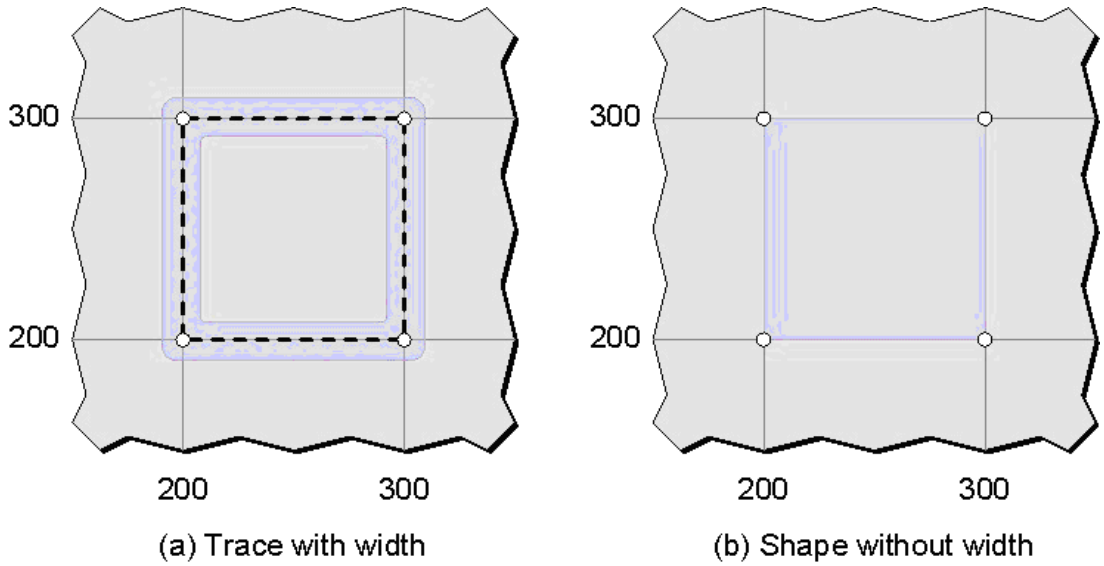
Before beginning it is worth noting that there is a difference between a path and a shape.

- Paths are graphical items (made up of a series of points) that have a width. Paths may or may not be closed. Traces have path geometries.

- Shapes are graphical items (made up of a series of points) that are closed and that have no width. Conductive areas have shape geometries.

Consider the case of a trace created in the form of a square 100 mils on a side with a width of 10 mils as illustrated in Figure 132: Comparison of Graphic Entities With and Without Width(a). (For this discussion, ignore any practical issues or potential DRC problems arise from creating such a trace.) If you zoom-in on this trace, its width scales the same factor as the zoom.

Figure 132. Comparison of Graphic Entities With and Without Width



By comparison, consider a shape that uses the same four points as illustrated in Figure 132: Comparison of Graphic Entities With and Without Width(b). In this case, the shape is filled with the user layer color. The shape's outline is always one pixel wide, independent of the zoom.

Different design objects may support either one or both of these geometric types. For example, a contour is a path in which the width is the diameter of the hole used to create that contour; the board outline is a path but the width is used only for display purposes (physically a board outline has zero width). Graphical objects on a user layer can be either a path or shape; they have no restrictions as there is no physical equivalent in the manufactured design.

Creating the Script

Use the scripting editor of your choice to open the *Skeleton.vbs* script in your *C:\temp* folder, save the script as *PutUserLayerGfx.vbs*, and then replace the comment on Line 18 with the following lines of code:

```
18  ' Set the current unit value so that this script
19  ' will still run correctly if the units are changed
20  ' in the design
21  pcbDocObj.CurrentUnit = epcbUnitMils
22
23  ' Create/Get all parameters needed to add a user layer graphics
24  Dim pntsArr(2, 9)
25  pntsArr(0,0) = 650 : pntsArr(1,0) = 2100 : pntsArr(2,0) = 0
26  pntsArr(0,1) = 500 : pntsArr(1,1) = 2000 : pntsArr(2,1) = 0
27  pntsArr(0,2) = 500 : pntsArr(1,2) = 1950 : pntsArr(2,2) = 0
28  pntsArr(0,3) = 550 : pntsArr(1,3) = 1950 : pntsArr(2,3) = -50
29  pntsArr(0,4) = 550 : pntsArr(1,4) = 1900 : pntsArr(2,4) = 0
30  pntsArr(0,5) = 650 : pntsArr(1,5) = 1900 : pntsArr(2,5) = 0
31  pntsArr(0,6) = 650 : pntsArr(1,6) = 1950 : pntsArr(2,6) = -50
```

```
32  pntsArr(0,7) = 700 : pntsArr(1,7) = 1950 : pntsArr(2,7) = 0
33  pntsArr(0,8) = 700 : pntsArr(1,8) = 2100 : pntsArr(2,8) = 0
34  pntsArr(0,9) = 650 : pntsArr(1,9) = 2100 : pntsArr(2,9) = 0
35
36  ' Get the number of points in the array
37  Dim numPntsInt : numPntsInt = UBound(pntsArr, 2) + 1
38
39  ' Get user layer object
40  Dim userLayerObj
41  Set userLayerObj = pcbDocObj.FindUserLayer("Test")
42
43  ' Provide a width
44  Dim widthReal : widthReal = 0
45
46  ' Set shape as filled
47  Dim filledBool : filledBool = True
48
49  ' Don't add this shape to a component
50  Dim cmpObj : Set cmpObj = Nothing
51
52  ' Add the user layer graphics
53  Dim usrLyrGfxObj
54  Set usrLyrGfxObj = pcbDocObj.PutUserLayerGfx(userLayerObj, _
55                     widthReal, numPntsInt, pntsArr, filledBool, _
56                     cmpObj, epcbUnitCurrent)
57
58  ' Select the newly added object
59  usrLyrGfxObj.Selected = True
60
61  ' Fit the selected object
62  Call pcbDocObj.ActiveViewEx.SetExtentsToSelection
```

In this example, the shape's vertices are hard-coded. These values are in mils. Line 21 assigns the epcbUnitMils enumerate to the CurrentUnit property.

```
18  ' Set the current unit value so that this script
19  ' will still run correctly if the units are changed
20  ' in the design
21  pcbDocObj.CurrentUnit = epcbUnitMils
```

Lines 24 through 34 instantiate the points array and specify the points array to use to create the shape.

```
23  ' Create/Get all parameters needed to add a user layer graphics
24  Dim pntsArr(2, 9)
25  pntsArr(0,0) = 650 : pntsArr(1,0) = 2100 : pntsArr(2,0) = 0
26  pntsArr(0,1) = 500 : pntsArr(1,1) = 2000 : pntsArr(2,1) = 0
27  pntsArr(0,2) = 500 : pntsArr(1,2) = 1950 : pntsArr(2,2) = 0
28  pntsArr(0,3) = 550 : pntsArr(1,3) = 1950 : pntsArr(2,3) = -50
29  pntsArr(0,4) = 550 : pntsArr(1,4) = 1900 : pntsArr(2,4) = 0
30  pntsArr(0,5) = 650 : pntsArr(1,5) = 1900 : pntsArr(2,5) = 0
```

```
31  pntsArr(0,6) = 650 : pntsArr(1,6) = 1950 : pntsArr(2,6) = -50
32  pntsArr(0,7) = 700 : pntsArr(1,7) = 1950 : pntsArr(2,7) = 0
33  pntsArr(0,8) = 700 : pntsArr(1,8) = 2100 : pntsArr(2,8) = 0
34  pntsArr(0,9) = 650 : pntsArr(1,9) = 2100 : pntsArr(2,9) = 0
```

Lines 37 through 50 specify the parameter values the *PutUserLayerGfx* method requires in order to add the shape to the board. Line 37 instantiates a variable numPntsInt, in which to store the number of vertices defined by the points array. This line also uses the UBound function to determine the number of vertices; the value the UBound function returns (plus 1) is assigned to the numPntsInt variable.

```
36  ' Get the number of points in the array
37  Dim numPntsInt : numPntsInt = UBound(pntsArr, 2) + 1
```

Line 40 instantiates a variable userLayerObj, which holds a UserLayer object. Line 41, uses the FindUserLayer method of the Document object to locate and return the user layer named "Test". Line 41 assigns the *userLayerObj* object to the userLayerObj variable.

```
39  ' Get user layer object
40  Dim userLayerObj
41  Set userLayerObj = pcbDocObj.FindUserLayer("Test")
```

Line 44 instantiates a variable widthReal in which to hold the width of the shape, and then assigns a value of 0 (zero) to this variable. This results in a shape with a 1-pixel outline.

```
43  ' Provide a width
44  Dim widthReal : widthReal = 0
```

Line 47 instantiates a variable filledBool and assigns True to this variable; this sets the shape as filled.

```
46  ' Set shape as filled
47  Dim filledBool : filledBool = True
```

For this example, the shape is not part of any component. Therefore, Line 50 instantiates a variable cmpObj in which to hold a Component object, and then assigns a value of Nothing to this variable.

```
49  ' Don't add this shape to a component
50  Dim cmpObj : Set cmpObj = Nothing
```

Line 53 instantiates a variable *usrLyrGfxObj*, which holds the new shape object. Lines 54 through 56 contain a single statement that uses the PutUserLayerGfx method to add the new shape to the design.

```
52  ' Add the user layer graphics
53  Dim usrLyrGfxObj
54  Set usrLyrGfxObj = pcbDocObj.PutUserLayerGfx(userLayerObj, _
55                     widthReal, numPntsInt, pntsArr, filledBool, _
56                     cmpObj, epcbUnitCurrent)
```

Line 59 uses the Selected property on the userLayerObj variable to select this object in the GUI. Line 62 uses the SetExtentsToSelection method of the *View* to zoom-in on the new shape.

```
' Select the newly added object
usrLyrGfxObj.Selected = True

' Fit the selected object
Call pcbDocObj.ActiveViewEx.SetExtentsToSelection
```

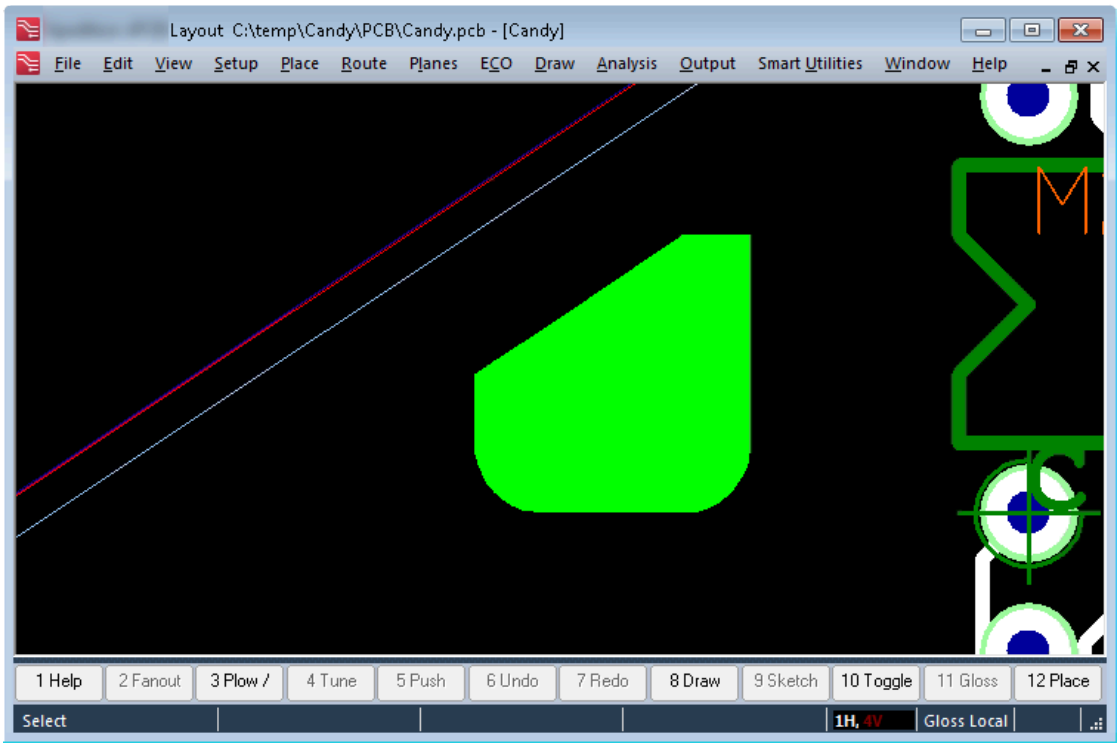
Running the Script

1. Close any instantiations of Xpedition Layout that are currently running, and then launch a new instantiation of this application.

2. Open the *Candy.pcb* design. Then choose **View > Display Control** to open the Display Control dialog. Select the **General** tab, and then ensure that the checkbox next to the user layer named "Test" is checked.

3. Drag-and-drop the *PutUserLayerGfx.vbs* script on the design window. Observe the resulting shape appear as illustrated in Figure 133: PutUserLayerGfx.vbs Script to Add a Graphical Element.

Figure 133. PutUserLayerGfx.vbs Script to Add a Graphical Element



Note:
This screenshot shows the shape as being unselected. If the shape is selected, its appearance changes depending on the current mode (Route, Draw, Place, etc.)

Creating Interactive Commands

There may be instances when you want to create scripts that interact with the end-user. For example, performing specific actions with mouse clicks or mouse moves. This topic presents an example of placing vias with mouse clicks.

Note:
This script uses the Command object. This object has a few methods and properties but has many events. In particular, you can use the Command object to receive any mouse or keyboard events.

Exercise 14: Placing Vias With the Mouse

Exercise 14: Placing Vias With the Mouse

This script places vias at locations where the end-user clicks the mouse.

Approach: To use the Command object to handle mouse click events.

Assumptions: That a "VIA20" padstack exists in the design; also that there is a net named "GND".

Creating the Script

Use the scripting editor of your choice to open the *Skeleton.vbs* script in your *C:\temp* folder, save the script as *PutVia.vbs*, and then replace the comment on Line 18 with the following lines of code:

```
18  ' Variable to hold the Command object
19  Dim cmdObj
20
21  ' Register a new command
22  Set cmdObj = pcbAppObj.Gui.RegisterCommand("Custom Place Via")
23
24  ' Attach events to the command object
25  Call Scripting.AttachEvents(cmdObj, "cmdObj")
26
27  ' Keep the script from exiting.
28  Scripting.DontExit = True
29
30 '''''''
```

```
41      Dim pstkObj
42      Set pstkObj = pcbDocObj.PutPadstack(1, _
43                                  pcbDocObj.LayerCount, "VIA20")
44
45      ' Get the net to put the via on
46      Dim netObj: Set netObj = pcbDocObj.FindNet("GND")
47
48      ' Don't attach it to a component
49      Dim cmpObj: Set cmpObj = Nothing
50
51      ' Turn off error handling so the script
52      ' can handle DRC violations
53      On Error Resume Next
54      Err.Clear
55      ' Add the via where the user specified.
56      Call pcbDocObj.PutVia(x, y, pstkObj, netObj, cmpObj)
57
58      ' See if there was an error
59      If Not Err.Number = 0 Then
60          ' Display the error on the status bar
61          Call pcbAppObj.Gui.StatusBarText(Err.Description, _
62                                       epcbStatusFieldError)
63      ' Clear the error
64          Err.Clear
65      Else
66          ' No error.  Let the user know what happened
67          Call pcbAppObj.Gui.StatusBarText("Placed via.", _
68                                          epcbStatusField3)
69      End If
70
71  End Function
```

On Line 19, the script instantiates a variable cmdObj, which holds the *Command* object.

```
18  ' Variable to hold the Command object
19  Dim cmdObj
```

Line 22 creates a *Command* object by using the RegisterCommand method of the Gui object, and assigns this to the cmdObj variable. The parameter passed to the RegisterCommand method is a string that appears in the Prompt field of the status bar while this command (script) is active.

```
21  ' Register a new command
22  Set cmdObj = pcbAppObj.Gui.RegisterCommand("Custom Place Via")
```

Line 25 uses the AttachEvents method of the Scripting object to attach an object's events to script functions. For more information refer to "Introducing Events and the Scripting Object" on page 179.)

The second parameter of the *AttachEvents* method is a string. This string is used as a common prefix for any of the event handler functions. The string itself is arbitrary, but it is a recommended programming practice to set this string to be identical to the name of the first parameter.

```
24  ' Attach events to the command object
25  Call Scripting.AttachEvents(cmdObj, "cmdObj")
```

With the exception of the scripts defined in the sections Adding Menus and Menu Items and Adding Toolbar Buttons, and Introducing Events and the Scripting Object, the majority of the scripts have finished once they have completed. This is a problem in this script, because if the script ends, the event handler cannot be called. So, Line 28 sets the DontExit property of the Scripting object to the Boolean value of True, which means that this script will continue running until the end of the current application session.

```
27  ' Keep the script from exiting.
28  Scripting.DontExit = True
```

A Command object is valid for only a specific duration. This duration ends in a number of ways; these include:

- Running another command.
- Pressing the Esc ("Escape") key.
- Terminating the object in the script itself.

Whenever any of these occur, the OnTerminate event is triggered. You should always have a handler for this event; in turn, this handler should, as a minimum, release the Command object. Additional cleanup may be required for other commands.

The event handler on Lines 33 through 36 sets the Command object stored in the cmdObj variable to Nothing in order to release the Command object.

```
33  Function cmdObj_OnTerminate()
34      ' Release the command
35      Set cmdObj = Nothing
36  End Function
```

The name of the event handler function (on Line 33) is key to making sure this function is fired. If either the name or the number of parameters differ in any way from the documentation, the system does not call the event handler function.

In the case of this example, there are no parameters to pass into the function. The name of the function (cmdObj_OnTerminate) has three parts as follows:

- The string specified as the second parameter to the *AttachEvents* method (this was the "cmdObj" string used on Line 25).
- An underscore '_ ' character.
- The name of the event as defined in the Layout Automation Reference documentation, which is *OnTerminate* in this example.

The mouse click event handler function is described on Lines 38 through 71. Line 38 declares the function itself. In this case, the name of the event is OnMouseClk, which is fired when the mouse button is released after being pressed (there are a number of other mouse click events, such as OnMouseDown, which fires when the mouse is pressed).

```
38  Function cmdObj_OnMouseClk(button, flags, x, y)
```

The OnMouseClk event requires four parameters. The first, named button in this example, specifies which mouse button was pressed. The second, named flags in this example, determines if one or more of the

Alt, Shift, or Ctrl keys are being pressed when the mouse click occurs. The event handler in this example does not use these first two parameters. Finally, the third and fourth parameters describe the X/Y location of the mouse cursor when the click event occurs.

A padstack is required for the via. Line 41 instantiates a variable to hold a Padstack object. Lines 42 and 43 use the PutPadstack method to add a Padstack object and assign it to the pstkObj variable. The first two parameters to the PutPadstack method specify the range of layers for the padstack (from 1 to the maximum number of layers in this example), while the third parameter is the name of the padstack ("VIA20" in this example).

```
40      ' Get all the paremeters needed to put a via
41      Dim pstkObj
42      Set pstkObj = pcbDocObj.PutPadstack(1,
43                                  pcbDocObj.LayerCount, "VIA20")
```

Note:
The PutPadstack method returns a padstack from the design if that padstack already exists. If that padstack does not exist in the design then the method acquires and returns the padstack from the local library.

A Net object is required for the via. This example connects the via to the GND net. So, Line 46 instantiates a variable netObj, uses the FindNet method to find the net named "GND" and assigns this net to the netObj variable.

```
45      ' Get the net to put the via on
46      Dim netObj: Set netObj = pcbDocObj.FindNet("GND")
```

For this example, the new via is not part of any component. So, Line 49 instantiates a variable cmpObj in which to hold a Component object, and then assigns a value of Nothing to this variable.

```
48      ' Don't attach it to a component
49      Dim cmpObj: Set cmpObj = Nothing
```

Line 53 turns off the automatic error handling so that the script can handle any DRC errors (refer to the Error Handling topic in “VBScript Primer” on page 27).

```
51      ' Turn off error handling so the script
52      ' can handle DRC violations
53      On Error Resume Next
54      Err.Clear
```

Line 56 uses the PutVia method to add the via to the design. The script uses the X/Y location generated by the mouse click event.

```
55      ' Add the via where the user specified.
56      Call pcbDocObj.PutVia(x, y, pstkObj, netObj, cmpObj)
```

Lines 59 through 71 check for any errors and also informs the end-user of results. Line 59 uses the Number property of the Err object to determine if there was an error. If this value is 0, then there was no error; otherwise, Line 61 informs the end-user about the error by using the Description property of the Err object and displaying this description in the Error field of the status bar.

```
58      ' See if there was an error
59      If Not Err.Number = 0 Then
60        ' Display the error on the status bar
61        Call pcbAppObj.Gui.StatusBarText(Err.Description, _
62                                         epcbStatusFieldError)
```

Line 64 uses the Clear method of the Err object to clear this error, otherwise system will detect it again on the next click of the mouse button.

```
63      ' Clear the error
64              Err.Clear
```

If there was no error, the script use the statement on Lines 67 and 68 to display a message on the status bar informing the end-user that a via was placed.

```
66        ' No error.  Let the user know what happened
67        Call pcbAppObj.Gui.StatusBarText("Placed via.", _
68                                         epcbStatusField3)
```

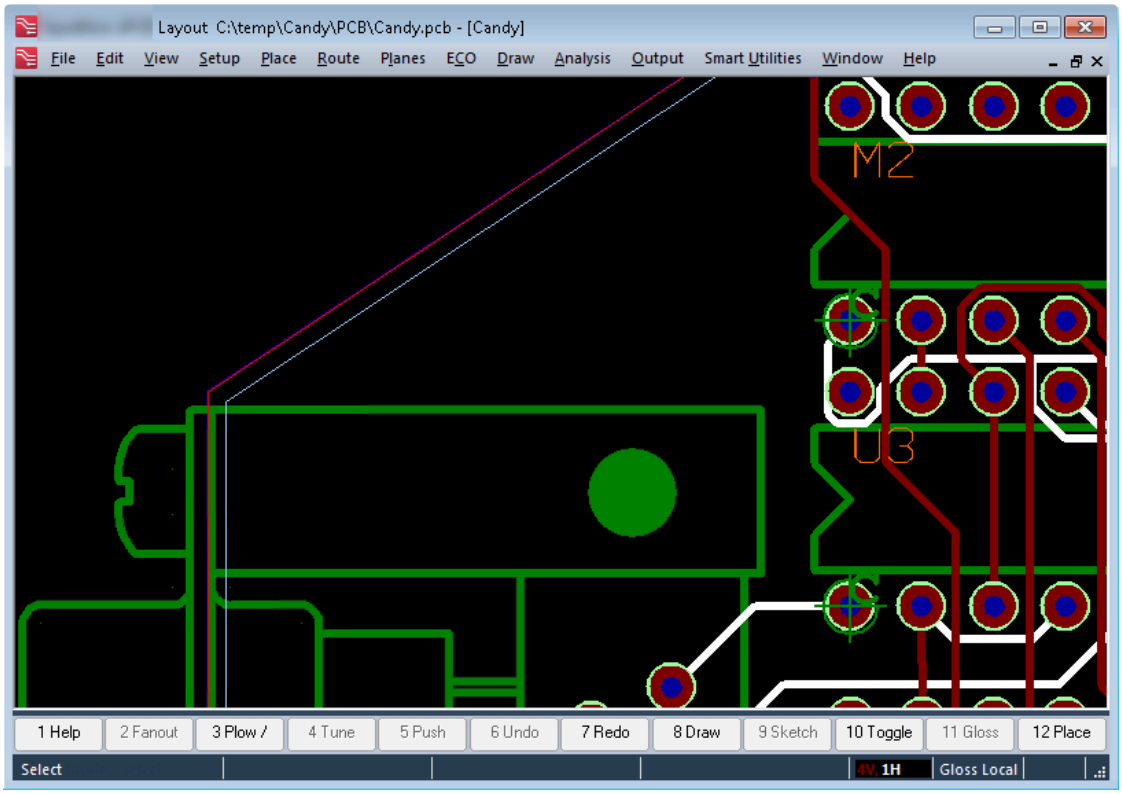
Running the Script

1. Close any instantiations of Xpedition Layout that are currently running, and then launch a new instantiation of this application.

2. Open the *Candy.pcb* design. This design contains a padstack named "VIA20".

3. Zoom-in on an empty area of the design. As shown in Figure 134: Zoom in on a Blank Area of the Design.

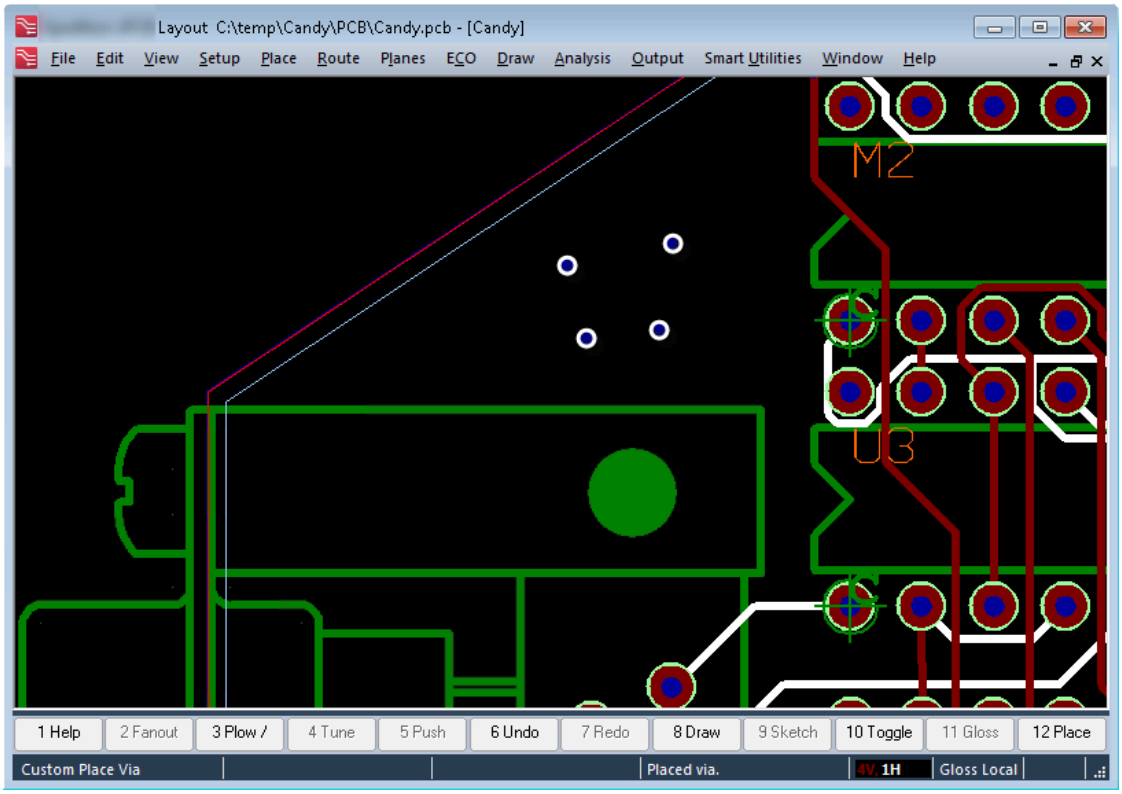
Figure 134. Zoom in on a Blank Area of the Design



4. Drag-and-drop the *PutVia.vbs* script on the design window. Observe that the "Custom Place Via" text appears in the Prompt field on the status bar as illustrated in Figure 135: Placing Vias (this was the string parameter specified on Line 22).

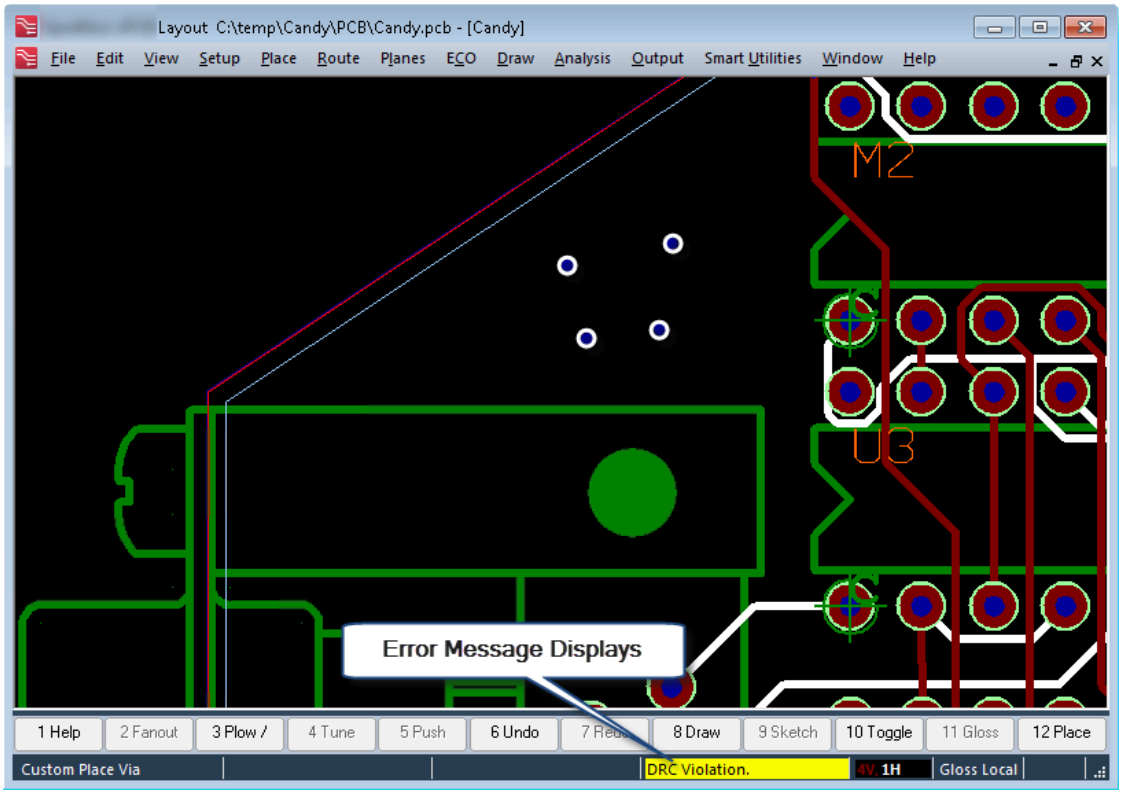
5. Click your mouse cursor a few times in the blank area of the design and observe that a new via appears with each click as illustrated in Figure 135: Placing Vias.

Figure 135. Placing Vias



6. Click on an existing pin (for example, M2-1) and observe that an error message appears in the Error field of the status bar as illustrated in Figure 136: An Error Occurs and Displays an Error String.

Figure 136. An Error Occurs and Displays an Error String



7. Finally, press <Esc> to end this script and observe that the "Custom Place Via" text disappears from the Prompt field on the status bar.

Creating Input/Output (I/O) Files

File Input/Output (I/O) does not have any direct relationship to Xpedition Layout Automation. However, it can be useful to be able to write information to, or read information from, files on the host system. This topic presents two examples:

1. writing a report to a file
2. reading and parsing data from a file

Note:
File I/O is not built into the VBScript language. It is provided by another COM server, which is invoked by the script. The same mechanism can be used by any COM-aware language such as JScript.

Exercise 15: Creating a Net Report
Exercise 16: Reading a Points Array From a File

Exercise 15: Creating a Net Report

This script creates and outputs a report file containing a list of nets, including the names of the nets and the numbers of opens and pins on each net.

Approach: To use the FileSystemObject object to create a new file and to then write data to this file.

Creating the Script

Use the scripting editor of your choice to open the *Skeleton.vbs* script in your *C:\temp* folder, save the script as *NetReport.vbs*, and then replace the comment on Line 18 with the following lines of code:

```
18  Dim overWriteInt: overWriteInt = 1
19
20  ' Create a FileSystemObject
21  Dim fileSysObj
22  Set fileSysObj = CreateObject("Scripting.FileSystemObject")
23  Dim txtStreamObj
24  Set txtStreamObj = fileSysObj.CreateTextFile(pcbDocObj.Path & _
25                        "LogFiles/MyNetReport.txt", overWriteInt)
26
27  ' Add a header
28  txtStreamObj.WriteLine()
29  txtStreamObj.WriteLine("My Net Report : " & Date)
30  txtStreamObj.WriteLine()
31
32  ' Get a collection of nets.
33  Dim netColl   ' Collection of all nets
34  Dim netObj   ' Net object for iteration
35  Set netColl = pcbDocObj.Nets
```

```
36
37  ' Iterate through all nets
38  For Each netObj In netColl
39    ' Report information on each net.
40    txtStreamObj.Write(netObj.Name)
41    txtStreamObj.WriteLine()
42    txtStreamObj.Write("    Opens: " & netObj.FromTos.Count)
43    txtStreamObj.WriteLine()
44    txtStreamObj.Write("    Pins: " & netObj.Pins.Count)
45    txtStreamObj.WriteLine()
46    txtStreamObj.WriteLine()
47  Next
48
49  ' Close the file
50  txtStreamObj.Close()
```

On Line 18, the script instantiates a variable overWriteInt and assigns it a value of 1. The variable indicates if a file should be overwritten. The value of 1 causes the system to overwrite any existing file.

```
18  Dim overWriteInt: overWriteInt = 1
```

Line 21 instantiates a variable fileSysObj, which holds the FileSystemObject object. Line 22 uses the CreateObject function and passes it the Prog ID of the FileSystemObject object as a parameter. The Prog ID is the string "Scripting.FileSystemObject" (for more information refer to “Prog ID” on page 21).

```
20  ' Create a FileSystemObject
21  Dim fileSysObj
22  Set fileSysObj = CreateObject("Scripting.FileSystemObject")
```

Note:
The FileSystemObject object is the COM server that provides a variety of File I/O capabilities, such as creating, copying, and deleting files and folders. The FileSystemObject object has many objects, properties, and methods associated with it. You can find out more information on this by using an Object Browser to view data on the Reference named FileSystemObject 1.0 Type Library (for more information on using Object Browsers refer to “Object Browsers” on page 536).

The script also creates a TextStream object. A TextStream object performs the tasks of reading from and writing to a file. One way to view this is that the TextStream object refers to the actual text in a file, as opposed to the file itself.

Line 23 instantiates a variable txtStreamObj, which holds the TextStream object. Lines 24 and 25 use the CreateTextFile method of the FileSystemObject object to create a new file and return the TextStream object associated with this file. The script then assigns this TextStream object to the txtStreamObj variable.

```
23  Dim txtStreamObj
24  Set txtStreamObj = fileSysObj.CreateTextFile(pcbDocObj.Path & _
25                        "LogFiles/MyNetReport.txt", overWriteInt)
```

The first parameter to the CreateTextFile method is the path to the file, which includes the name of the file. The script creates this path by combining two strings. The first string is returned from the Path property associated with the Document object. This is the path to the folder containing the current PCB

design. The second string specifies a combination of the LogFiles folder and the name of the report file to create, which is MyNetReport.txt.

The second parameter to the CreateTextFile method is the overWriteInt variable. This value instructs the CreateTextFile method to overwrite any existing file.

Lines 27 through 30 write a header to the report file. This header consists of a blank line, followed by a line containing the title of the report and the current date, followed by another blank line.

```
27  ' Add a header
28  txtStreamObj.WriteLine()
29  txtStreamObj.WriteLine("My Net Report : " & Date)
30  txtStreamObj.WriteLine()
```

Line 33 instantiates a variable netColl in which to hold a collection of Net objects. Line 34 instantiates a variable netObj to hold a single Net object. Line 35 returns a collection of Net objects and assigns them to the netColl variable.

```
32  ' Get a collection of nets.
33  Dim netColl   ' Collection of all nets
34  Dim netObj   ' Net object for iteration
35  Set netColl = pcbDocObj.Nets
```

Lines 38 through 47 use a For Each … Next control loop to iterate through each net in the collection. Data is output for each net.

```
37  ' Iterate through all nets
38  For Each netObj In netColl
39    ' Report information on each net.
40    txtStreamObj.Write(netObj.Name)
41    txtStreamObj.WriteLine()
42    txtStreamObj.Write("    Opens: " & netObj.FromTos.Count)
43    txtStreamObj.WriteLine()
44    txtStreamObj.Write("    Pins: " & netObj.Pins.Count)
45    txtStreamObj.WriteLine()
46    txtStreamObj.WriteLine()
47  Next
```

The Write statement outputs its associated data to the file. By comparison, the Writeline statement outputs the same data and then automatically appends a newline. Lines 42 and 43 can be combined into a single Writeline statement. Alternatively, you can delete the Writeline statement on Line 43 and append a VbCrLf constant to the end of the string. The same thing applies to the two statements on Lines 44 and 45.

Finally, Line 50 uses the Close method associated with the TextStream object to close the file.

```
49  ' Close the file
50  txtStreamObj.Close()
```

Running the Script

1. Close any instantiations of Xpedition Layout that are currently running, and then launch a new instantiation of this application.

2. Open the *Candy.pcb* design.

3. Drag-and-drop the *NetReport.vbs* script on the design window.

4. Use your browser to locate the following file:

   C:\temp\Candy\Pcb\LogFiles\MyNetReport.txt

5. Open this file in the text editor of your choice and observe its contents. It should appear similar to the following:

```
My Net Report : 12/18/2014

GND
    Opens: 0
    Pins: 18

VCC
    Opens: 0
    Pins: 19

Battery
    Opens: 0
    Pins: 6
           :
         etc.
```

Exercise 16: Reading a Points Array From a File

This script opens a text file, to read and parse its contents, and uses these contents to create a points array.

Approach: To use the FileSystemObject object to open an existing file and to then read data from this file.

Assumptions: The design contains a user layer named Test; also, a file named *PointsArrayString.txt* exists in your *C:\temp* folder prior running this script.

Creating the Script

Use the text editor of your choice to enter the following text and then save this file as *PointsArrayString.txt* in your *C:\temp* folder. This text describes the same X-Y-R values for the points array in the exercise, “Adding User Layer Graphics” on page 243.

```
(700,1950,0)
(650,1950,50)
(650,1900,0)
(550,1900,0)
(550,1950,50)
(500,1950,0)
(500,2000,0)
(650,2100,0)
(700,2100,0)
(700,1950,0)
```

Using the scripting editor, open the *Skeleton.vbs* script in your *C:\temp* folder, save the script as *ReadPointsArray.vbs*, and then replace the comment on Line 18 with the following lines of code:

```
18  ' Add any type library for file system object
19  Scripting.AddTypeLibrary("Scripting.FileSystemObject")
20
21  ' Create a FileSystemObject
22  Dim fileSysObj
23  Set fileSysObj = CreateObject("Scripting.FileSystemObject")
24
25  ' Create variable to hold name of input file
26  Dim pntsFileStr: pntsFileStr = "c:/temp/PointsArrayString.txt"
27
28  ' Determine if the file exists.
29  If Not fileSysObj.FileExists(pntsFileStr) Then
30     MsgBox("Can't find file " & pntsFileStr & ".")
31  Else
32     ' Open and read the file
33     Dim txtStreamObj
34     Set txtStreamObj = fileSysObj.OpenTextFile(pntsFileStr, _
35                                 ForReading)
36     ' Read the whole file.
37     Dim pntsArrStr: pntsArrStr = txtStreamObj.ReadAll
38     ' Close the file
39     txtStreamObj.Close()
40
41     ' Convert the string to a points array
42     Dim pntsArr: pntsArr = GetPointsArrayFromString(pntsArrStr)
43
44     ' Get the number of points in the array
45     Dim numPntsInt: numPntsInt = UBound(pntsArr, 2)
46
47     ' Get user layer object
48     Dim userLayerObj
49     Set userLayerObj = pcbDocObj.FindUserLayer("Test")
50
51     ' Provide a width
52     Dim widthReal: widthReal = 0
53
54     'shape as filled
55     Dim filledBool: filledBool = true
```

```
 56
 57     ' Don't add this shape to a component
 58     Dim cmpObj: Set cmpObj = Nothing
 59
 60     ' Add the user layer graphics
 61     Dim usrLyrGfxObj
 62     Set usrLyrGfxObj = pcbDocObj.PutUserLayerGfx(userLayerObj, _
 63                  widthReal, numPntsInt, pntsArr, filledBool, _
 64                       cmpObj, epcbUnitCurrent)
 65
 66     ' Select the newly added object
 67     usrLyrGfxObj.Selected = true
 68
 69     ' Fit the selected object
 70     Call pcbDocObj.ActiveViewEx.SetExtentsToSelection()
 71  End If
 72
 73  ''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
 74  'Local functions
 75
 76  ' Creates a points array from a string
 77  Function GetPointsArrayFromString(inputStr)
 78
 79      ' Split string into a multiple strings.  One for each line
 80      Dim linesArr: linesArr = Split(inputStr, vbCrLf)
 81
 82      ' Instantiate the array to return
 83      Dim pntsArr
 84    Redim pntsArr(2, UBound(linesArr))
 85
 86    ' Loop through each line reading the values
 87    Dim i
 88    For i = 0 To UBound(linesArr)
 89      ' Split on the comma to find each value
 90      Dim strArr: strArr = Split(linesArr(i), ",")
 91      Dim xReal: xReal = CDbl(Split(strArr(0), "(")(1))
 92      Dim yReal: yReal = CDbl(strArr(1))
 93      Dim rReal: rReal = CDbl(Split(strArr(2), ")")(0))
 94
 95         ' Put the values in the return array.
 96         pntsArr(0,i) = xReal
 97         pntsArr(1,i) = yReal
 98         pntsArr(2,i) = rReal
 99      Next
100
101      ' Return points array
102      GetPointsArrayFromString = pntsArr
103
104  End Function
```

Line 19 uses the AddTypeLibrary method of the Scripting object to add the library of enumerates associated with the FileSystemObject object.

```
18  ' Add any type library for file system object
19  Scripting.AddTypeLibrary("Scripting.FileSystemObject")
```

Note:
It is common practice to add all of the type libraries required by a script, in the same section at the beginning of the script. It is not done here because all of these building-block examples are based on the common *Skeleton.vbs* script.

Line 22 instantiates a variable fileSysObj, which holds the FileSystemObject object. Line 23 uses the CreateObject function and passes it the Prog ID of the FileSystemObject object as a parameter.

```
21  ' Create a FileSystemObject
22  Dim fileSysObj
23  Set fileSysObj = CreateObject("Scripting.FileSystemObject")
```

Line 26 instantiates a variable pntsFileStr and then assigns the name of the input file to this variable.

```
25  ' Create variable to hold name of input file
26  Dim pntsFileStr: pntsFileStr = "c:/temp/PointsArrayString.txt"
```

The “Creating a Net Report” on page 256 example hard-coded the name of the file as part of the function call. This script uses the same file name several times, so it makes more sense to assign the name of the file to a variable. This allows you to modify the name of the file by making a single change to the script.

Line 29 uses the FileExists method associated with the FileSystemObject object to determine if the input file exists. If the file does not exist, the statement on Line 30 reports an error.

```
29  If Not fileSysObj.FileExists(pntsFileStr) Then
30     MsgBox("Can't find file " & pntsFileStr & ".")
```

If the input file does exist, Line 33 instantiates a variable txtStreamObj, which holds the TextStream object. Line 34 uses the OpenTextFile method associated with the FileSystemObject object to create a new file and returns the TextStream object associated with this file. This TextStream object is assigned to the txtStreamObj variable.

```
31  Else
32     ' Open and read the file
33     Dim txtStreamObj
34     Set txtStreamObj = fileSysObj.OpenTextFile(pntsFileStr, _
35                                   ForReading)
```

The first parameter to the OpenTextFile method is the path to the input file (this includes the name of the file.) This example uses the value stored in the pntsFileStr variable assigned on Line 26.

The second parameter to the OpenTextFile method is the enumerate ForReading. This enumerate specifies to open this file for reading. Other related enumerates are ForAppending and ForWriting. The latter case is different to the previous script in which the CreateTextFile method creates a new file from scratch. The ForWriting enumerate informs the OpenTextFile method to open an existing file and then overwrite its contents.

Line 37 instantiates a variable pntsArrStr. The ReadAll method of the TextStream object reads the entire contents of the text file, and assigns the result to the pntsArrStr variable.

```
 36     ' Read the whole file.
 37     Dim pntsArrStr: pntsArrStr = txtStreamObj.ReadAll
```

The script no longer requires the input file to remain open, so Line 39 uses the Close method of the TextStream object to close the file.

```
 38     ' Close the file
 39     txtStreamObj.Close()
```

Line 42 instantiates a variable pntsArr, which holds the points array. The script calls the GetPointsArrayFromString function and passes the pntsArrStr variable as a parameter.

```
 41     ' Convert the string to a points array
 42     Dim pntsArr: pntsArr = GetPointsArrayFromString(pntsArrStr)
```

GetPointsArrayFromString is a user-defined function (described later) that returns a points array.

Lines 44 through 70 are taken directly from the “Adding User Layer Graphics” on page 243 exercise script. These lines generate the parameters required to add a graphic element using the points array in the design.

```
 44     ' Get the number of points in the array
 45     Dim numPntsInt: numPntsInt = UBound(pntsArr, 2)
 46
 47     ' Get user layer object
 48     Dim userLayerObj
 49     Set userLayerObj = pcbDocObj.FindUserLayer("Test")
 50
 51     ' Provide a width
 52     Dim widthReal: widthReal = 0
 53
 54     'shape as filled
 55     Dim filledBool: filledBool = true
 56
 57     ' Don't add this shape to a component
 58     Dim cmpObj: Set cmpObj = Nothing
 59
 60     ' Add the user layer graphics
 61     Dim usrLyrGfxObj
 62     Set usrLyrGfxObj = pcbDocObj.PutUserLayerGfx(userLayerObj, _
 63                    widthReal, numPntsInt, pntsArr, filledBool, _
 64                    cmpObj, epcbUnitCurrent)
 65
 66     ' Select the newly added object
 67     usrLyrGfxObj.Selected = true
 68
 69     ' Fit the selected object
 70     Call pcbDocObj.ActiveViewEx.SetExtentsToSelection()
```

Lines 77 through 104 define the GetPointsArrayFromString function. Line 77 declares the function name and the name of its parameter (inputStr).

```
77  Function GetPointsArrayFromString(inputStr)
```

Line 80 instantiates a variable linesArr in which to hold an array of strings. Each string is a line from the original text file. Line 80 also uses the VBScript Split function to split the string from the input file into an array of strings, and assigns this array of strings to the linesArr variable.

```
 79      ' Split string into a multiple strings.  One for each line
 80      Dim linesArr: linesArr = Split(inputStr, vbCrLf)
```

The original PointsArrayString.txt file appears like the following in a text editor:

```
(700,1950,0)
(650,1950,50)
(650,1900,0)
(550,1900,0)
(550,1950,50)
(500,1950,0)
(500,2000,0)
(650,2100,0)
(700,2100,0)
(700,1950,0)
```

In reality, the file on the system may be visualized as containing a single long string, in which the groups of parentheses-delimited values are separated by newlines, which are represented by vbCrLf constants as follows:

```
(700,1950,0)vbCrLf(650,1950,50)vbCrLf(650,1900,0)...etc.
```

The newlines are present in both the text editor and in the string read into the script; they exist as vbCrLf constants in the string.

The Split function always returns an array. The Split function on Line 80 passes two parameters: the first is a string variable containing the entire contents of the original text file; the second is the string or string constant to delimit the original string (in this case the vbCrLf string constant).

When the Split function detects a vbCrLf constant, it splits the string at that point. It also discards the character or string that controls the split, vbCrLf in this case. The result is an array of strings, which you can visualize as follows:

```
(700,1950,0)
(650,1950,50)
(650,1900,0)
(550,1900,0)
(550,1950,50)
(500,1950,0)
(500,2000,0)
(650,2100,0)
(700,2100,0)
(700,1950,0)
```

Line 83 instantiates a variable pntsArr, which holds the points array. Line 84 re-dimensions this variable with two dimensional parameters. The first parameter specifies that each row in the array contains three items (points) indexed from 0 through 2. The second parameter specifies the number of rows in the array;

this second parameter is defined using the UBound function to determine the number of items in the array of strings.

```
 82      ' Instantiate the array to return
83      Dim pntsArr
84    Redim pntsArr(2, UBound(linesArr))
```

Note:
The name (*pntsArr*) that instantiates the points array on Line 83 is the same name used in the body of the script on Line 42. However, the script re-instantiates it here with a new Dim statement (as opposed to using the existing variable name), which means that the new instantiation is *local* to this function and different from the original variable in the body of the script.

Line 87 instantiates a variable i, which iterates through the For loop declared on Line 88. This loop iterates through each item in the array of strings.

```
87    Dim i
88    For i = 0 To UBound(linesArr)
```

Consider the first iteration through the loop. In this case, the first item in the array of strings is:

```
(700,1950,0)
```

Line 90 declares a variable strArr and uses the Split function to split the contents of the current item in the string array using a comma as the delimiter.

```
89      ' Split on the comma to find each value
90      Dim strArr: strArr = Split(linesArr(i), ",")
```

This results in the strArr variable containing three items as follows:

```
(700
1950
0)
```

The first item (700 is stored in strArr(0), the second item 1950 is stored in strArr(1), and the third item 0) is stored in strArr(2).

Observe that when the Split function performs a split it discards the character or string controlling the split (the comma in this case).

Consider what happens when the Split function is called on Line 91.

```
91      Dim xReal: xReal = CDbl(Split(strArr(0), "(")(1))
```

A variable xReal is instantiated. This is a real number because the end-user can specify values as integers or reals. Also on this line, the Split function uses the "(" character to split the (700 stored in strArr(0).The result is an array containing two items as follows:

```
700
```

The first (empty) item has an index value of 0; the second item 700 has an index value of 1. The script uses the (1) located toward the right-hand side of the statement on Line 91 to access the second item, 700, in the resulting array. The script then casts the result as a real value and assigns this to the xReal variable.

Line 92 is relatively simple, it casts the contents of strArr(1) as a real and assigns the result to the yReal variable.

```
92      Dim yReal: yReal = CDbl(strArr(1))
```

Now consider the statements on Line 93:

```
93      Dim rReal: rReal = CDbl(Split(strArr(2), ")")(0))
```

Line 93 instantiates a variable rReal. The Split function uses the ")" character to split the 0) stored in strArr(2). The result is an array containing two items as follows:

```
0
```

The first item, 0, has an index value of 0; the second (empty) item has an index value of 1. This is why the script uses the (0) located toward the right-hand side of the statement on Line 93 to access the first item, 0, in the resulting array. The script then casts the result as a real value and assigns this to the rReal variable.

Lines 96 through 98 assign the X, Y, and R values (700, 1950, and 0) to the appropriate locations in the points array.

```
 96         pntsArr(0,i) = xReal
 97         pntsArr(1,i) = yReal
 98         pntsArr(2,i) = rReal
```

The script repeats the For … Next loop for each item in the array of strings. Finally, Line 102 returns the points array and Line 104 terminates this function.

```
101      ' Return points array
102      GetPointsArrayFromString = pntsArr
103
104  End Function
```

Running the Script

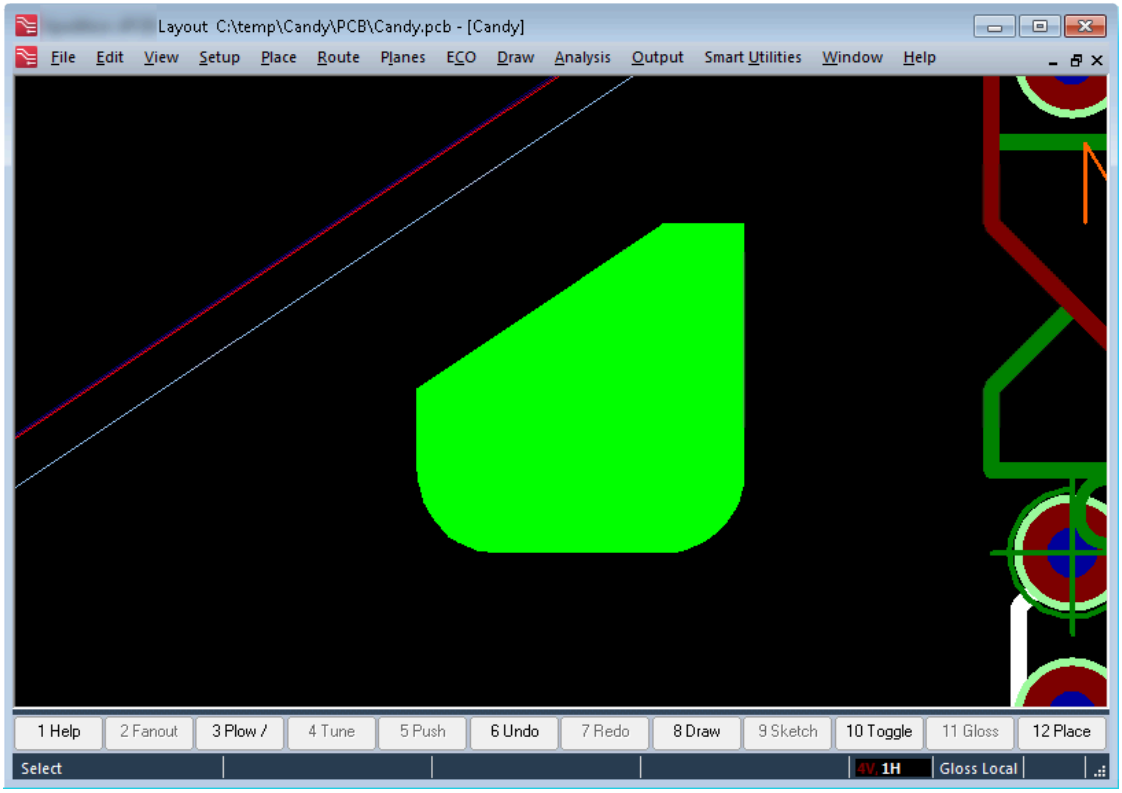
1. Close any instantiations of Xpedition Layout that are currently running, and then launch a new instantiation of this application.

2. Open the *Candy.pcb* design. The design contains a user layer named “Test”.

3. Choose **View > Display Control**, the Display Control dialog displays. Select the **General** tab and ensure that the checkbox next to the user layer named "Test" is checked.

4. Drag-and-drop the *ReadPointsArray.vbs* script on the design window.Observe the resulting shape appear as illustrated in Figure 137: ReadPointsArray.vbs Script Result.

Figure 137. ReadPointsArray.vbs Script Result



Running Executables

There are often cases where you may want to run (execute) separate applications from within a script; for example, to view a report in a text editor or to compress/decompress files.

Exercise 17: Viewing a Net Report With NotePad
Exercise 18: Compressing a File and E-mailing the Result

Exercise 17: Viewing a Net Report With NotePad

This script generates a net report as described in the “Creating a Net Report” exercise, and uses the Xpedition Layout Automation Interface to launch NotePad to view this report.

Approach: To use the *Exec* object from the *Viewlogic* COM server to run a system command that launches the NotePad editor.

Assumptions: That you are running under the Windows operating system; that the *Notepad.exe* application is available; and that you have already created the *NetReport.vbs* script as described in the “Creating a Net Report” on page 256 exercise.

Creating the Script

Use the scripting editor of your choice to open the *NetReport.vbs* script in your *C:\temp* folder, save the script as *ViewNetReport.vbs*. The section of this script of interest occurs between Lines 18 and 50:

```
18  Dim overWriteInt: overWriteInt = 1
19
20  ' Create a FileSystemObject
21  Dim fileSysObj
22  Set fileSysObj = CreateObject("Scripting.FileSystemObject")
23  Dim txtStreamObj
24  Set txtStreamObj = fileSysObj.CreateTextFile(pcbDocObj.Path & _
25                        "LogFiles/MyNetReport.txt", overWriteInt)
26
27  ' Add a header
28  txtStreamObj.WriteLine()
29  txtStreamObj.WriteLine("My Net Report : " & Date)
30  txtStreamObj.WriteLine()
31
32  ' Get a collection of nets.
33  Dim netColl   ' Collection of all nets
34  Dim netObj   ' Net object for iteration
35  Set netColl = pcbDocObj.Nets
36
37  ' Iterate through all nets
38  For Each netObj In netColl
39    ' Report information on each net.
40    txtStreamObj.Write(netObj.Name)
41    txtStreamObj.WriteLine()
42    txtStreamObj.Write("    Opens: " & netObj.FromTos.Count)
43    txtStreamObj.WriteLine()
```

```
44    txtStreamObj.Write("    Pins: " & netObj.Pins.Count)
45    txtStreamObj.WriteLine()
46    txtStreamObj.WriteLine()
47  Next
48
49  ' Close the file
50  txtStreamObj.Close
```

The last thing done in this script occurs on Line 50; this statement closes the *MyNetReport.txt* file after it is output. Add the following lines to this script:

```
52  ' Create a FileSystemObject
53  Dim execObj
54  Set execObj = CreateObject("Viewlogic.Exec")
55  ' Set the working directory
56  execObj.WorkingDirectory = pcbDocObj.Path + "LogFiles/"
57  ' Open notepad on the net report.
58  Call execObj.Run("notepad.exe MyNetReport.txt")
```

Line 53 instantiates a variable *execObj*, which holds the *Exec* object (where *Exec* is short for "Execute" and refers to an object that runs system commands). Line 54 calls the *CreateObject* function that is built into VBScript, and passes it the "Viewlogic.Exec" Prog ID.

```
53  Dim execObj
54  Set execObj = CreateObject("Viewlogic.Exec")
```

Note:
COM servers are machine-specific such that even two Windows machines can have a different set of COM servers. You should use *CreateObject* to access a COM server. The "Viewlogic.Exec" COM server is available with Mentor Graphics products.

Line 56 uses the *WorkingDirectory* property of the *Exec* object to set the working directory to the directory containing the *MyNetReport.txt* report file.

```
55  ' Set the working directory
56  execObj.WorkingDirectory = pcbDocObj.Path + "LogFiles/"
```

Line 58 uses the *Run* method of the *Exec* object to run the *notepad.exe* application and passes it the name of the net report file, *MyNetReport.txt*.

```
57  ' Open notepad on the net report.
58  Call execObj.Run("notepad.exe MyNetReport.txt")
```

The *Exec* object's *Run* method runs any command line string. So, you can substitute the *notepad.exe* application for the editor of your choice.

Running the Script

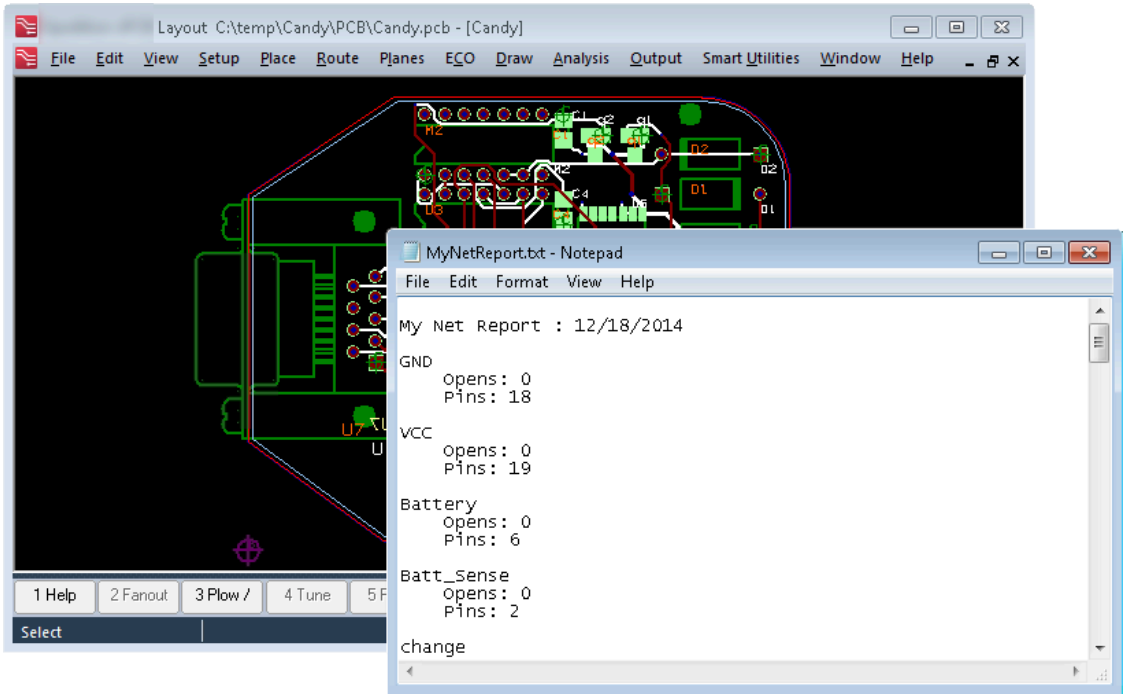
1. Close any instantiations of Xpedition Layout that are currently running, and then launch a new instantiation of this application.

2. Open the *Candy.pcb* design.

3. Drag-and-drop the *ViewNetReport.vbs* script on the design window.

4. Observe that after the script generates the net report, the script launches the Notepad editor and uses this editor to display your *MyNetReport.txt* file as illustrated in Figure 138: ViewNetReport.vbs Script Launches the NotePad Editor.

Figure 138. ViewNetReport.vbs Script Launches the NotePad Editor



Exercise 18: Compressing a File and E-mailing the Result

This script generates a net report as described in the “Creating a Net Report” exercise, and uses the Xpedition Layout Automation Interface to compress and E-mail this report.

Approach: To use the *Exec* object from the *Viewlogic* COM server to compress a file using WinZIP and to then use the *Outlook* COM server to generate an E-mail.

Assumptions: That you are running under the Windows operating system because the *Outlook* COM server is not available on UNIX/Linux systems. Also that you have already created the *NetReport.vbs* script as described in the “Creating a Net Report” on page 256 exercise.

Creating the Script

Use the scripting editor of your choice to open the *NetReport.vbs* script in your *C:\temp* folder, save the script as *ZipAndMailNetReport.vbs*. The section of this script of interest occurs between Lines 18 and 50:

```
18  Dim overWriteInt: overWriteInt = 1
19
20  ' Create a FileSystemObject
21  Dim fileSysObj
22  Set fileSysObj = CreateObject("Scripting.FileSystemObject")
23  Dim txtStreamObj
24  Set txtStreamObj = fileSysObj.CreateTextFile(pcbDocObj.Path & _
25                        "LogFiles/MyNetReport.txt", overWriteInt)
26
27  ' Add a header
28  txtStreamObj.WriteLine()
29  txtStreamObj.WriteLine("My Net Report : " & Date)
30  txtStreamObj.WriteLine()
31
32  ' Get a collection of nets.
33  Dim netColl   ' Collection of all nets
34  Dim netObj   ' Net object for iteration
35  Set netColl = pcbDocObj.Nets
36
37  ' Iterate through all nets
38  For Each netObj In netColl
39    ' Report information on each net.
40    txtStreamObj.Write(netObj.Name)
41    txtStreamObj.WriteLine()
42    txtStreamObj.Write("    Opens: " & netObj.FromTos.Count)
43    txtStreamObj.WriteLine()
44    txtStreamObj.Write("    Pins: " & netObj.Pins.Count)
45    txtStreamObj.WriteLine()
46    txtStreamObj.WriteLine()
47  Next
48
49  ' Close the file
50  txtStreamObj.Close
```

The last action in this script occurs on Line 50; this is where the script closes the *MyNetReport.txt* file after the file is output. Add the following lines to this script:

```
52 ' Create a FileSystemObject
53  Dim execObj
54  Set execObj = CreateObject("Viewlogic.Exec")
55
56  ' Set the working directory
57  execObj.WorkingDirectory = pcbDocObj.Path & "LogFiles/"
58
59  ' Tell the user what is going on
60  Call pcbAppObj.Gui.StatusBarText("Zipping net report...",
                                              epcbStatusField1)
```

```
61
62  ' Zip all the files in the Output directory
63  Dim visibleInt: visibleInt = 0
64  Dim waitBool: waitBool = True
65  Call execObj.Run("""C:/Program Files/WinZip/WinZip32.exe""
   -min -a -r NetReport.zip MyNetReport.txt", visibleInt, waitBool)
66
67  '  Tell the user what is going on
68  Call pcbAppObj.Gui.StatusBarText("Creating email...",
                                                 epcbStatusField1)
69
70  ' Create an outlook object
71  Dim outlookObj
72  Set outlookObj = CreateObject("Outlook.Application")
73
74  ' Create a new email
75  ' AddTypeLibrary doesn't work so hardcode the enumerate
76  Const olMailItem = 0
77  Dim mailObj
78  Set mailObj = outlookObj.CreateItem(olMailItem)
79
80  ' Setup the email
81  mailObj.Subject = "Net report for " & pcbDocObj.Name
82  mailObj.Attachments.Add(pcbDocObj.Path +
                                        "LogFiles/NetReport.zip")
83  mailObj.Display
84
85  ' Let the user know we have finished.
86  Call pcbAppObj.Gui.StatusBarText("Processing complete.",
                                                  epcbStatusField1)
```

Line 53 instantiates a variable *execObj* to hold the *Exec* object. Line 54 calls the *CreateObject* function and passes it the "Viewlogic.Exec" Prog ID.

```
53  Dim execObj
54  Set execObj = CreateObject("Viewlogic.Exec")
```

Line 56 uses the *WorkingDirectory* property of the *Exec* object to set the working directory to the directory/folder containing the *MyNetReport.txt* report file.

```
55  ' Set the working directory
56  execObj.WorkingDirectory = pcbDocObj.Path + "LogFiles/"
```

Line 60 displays a message on the status bar to inform the end-user of the current actions.

```
59  ' Tell the user what is going on
60  Call pcbAppObj.Gui.StatusBarText("Zipping net report...",
                                                 epcbStatusField1)
```

When you use the *Exec* object to run a command, it automatically invokes a console window in which to run the command. Line 63 instantiates a variable *visibleInt* and assigns a value of 0 to this variable. This value informs the system to make the console window invisible. The default, is that the window is visible. If you want to view the window, set *visibleInt* variable to 1.

```
62  ' Zip the file in the Output directory
63  Dim visibleInt: visibleInt = 0
```

Line 63 instantiates a variable *waitBool* and assigns a Boolean value of *True* to this variable. This value informs the system to wait for the *Run* method on Line 65 to finish performing its actions before proceeding with the remainder of the script.

```
64  Dim waitBool: waitBool = True
```

Line 65 uses the *Run* method of the *Exec* object to run the *WinZip32.exe* application and passes it the following arguments:

-min	Run WinZIP in the background (not visible)
-a	Add the specified file to the compressed file
-r	Include sub-folders (not relevant in this example)
MyNetReport.zip	The name of the compressed output file
MyNetReport.txt	The name of the input file to be compressed

```
65  Call execObj.Run("""C:/Program Files/WinZip/WinZip32.exe""
   -min -a -r NetReport.zip MyNetReport.txt", visibleInt, waitBool)
```

After the file is compressed, Line 68 displays a message on the status bar to inform the end-user of the current actions.

```
67  '  Tell the user what is going on
68  Call pcbAppObj.Gui.StatusBarText("Creating email...",
                                                    epcbStatusField1)
```

Line 71 instantiates a variable *outlookObj* to hold the *Outlook* object. Line 72 calls the *CreateObject* function, passes it the "Outlook.Application" Prog ID, and assigns the resulting *Outlook* object to the *outlookObj* variable.

```
70  ' Create an outlook object
71  Dim outlookObj
72  Set outlookObj = CreateObject("Outlook.Application")
```

Adding the Microsoft type library to the script does not provide access to the *Outlook* enumerates. To work around this problem, Line 76 declares a constant named *olMailItem* and assigns a value of 0. The *CreateItem* method of the Outlook *Application* object takes an enumerate as a parameter. A parameter value of 0 to the *CreateItem* method creates a *Mail* object (as opposed to a *Calendar* object or some other Outlook-related object).

Line 77 instantiates a variable *mailObj* to hold a *Mail* object. Line 78 uses the *CreateItem* method of the Outlook *Application* object to create a new *Mail* object. The script then assigns the *Mail* object to the *mailObj* variable.

```
74  ' Create a new email
75  ' AddTypeLibrary doesn't work so hardcode the enumerate
76  Const olMailItem = 0
77  Dim mailObj
78  Set mailObj = outlookObj.CreateItem(olMailItem)
```

Line 81 uses the *Subject* property of the *Mail* object to assign text to the **Subject** field. Line 82 uses the *Add* method of the *Attachments* collection of the *Mail* object to add an attachment to the E-mail (to do this pass a parameter that specifies the path to the newly created ZIP file).

```
80  ' Setup the email
81  mailObj.Subject = "Net report for " & pcbDocObj.Name
82  mailObj.Attachments.Add(pcbDocObj.Path + "LogFiles/NetReport.zip")
```

Line 83 uses the *Display* method of the *Mail* object to display the E-mail on the screen. This allows the end-user to fill in the **To** and **CC** fields and also add a message to the body of the E-mail if required (This could be done in the script if you know this information beforehand).

```
83  mailObj.Display
```

Finally, Line 86 displays a message on the status bar to inform the end-user that the script finished.

```
85  ' Let the user know we have finished.
86  Call pcbAppObj.Gui.StatusBarText("Processing complete.",
                                                 epcbStatusField1)
```

Running the Script

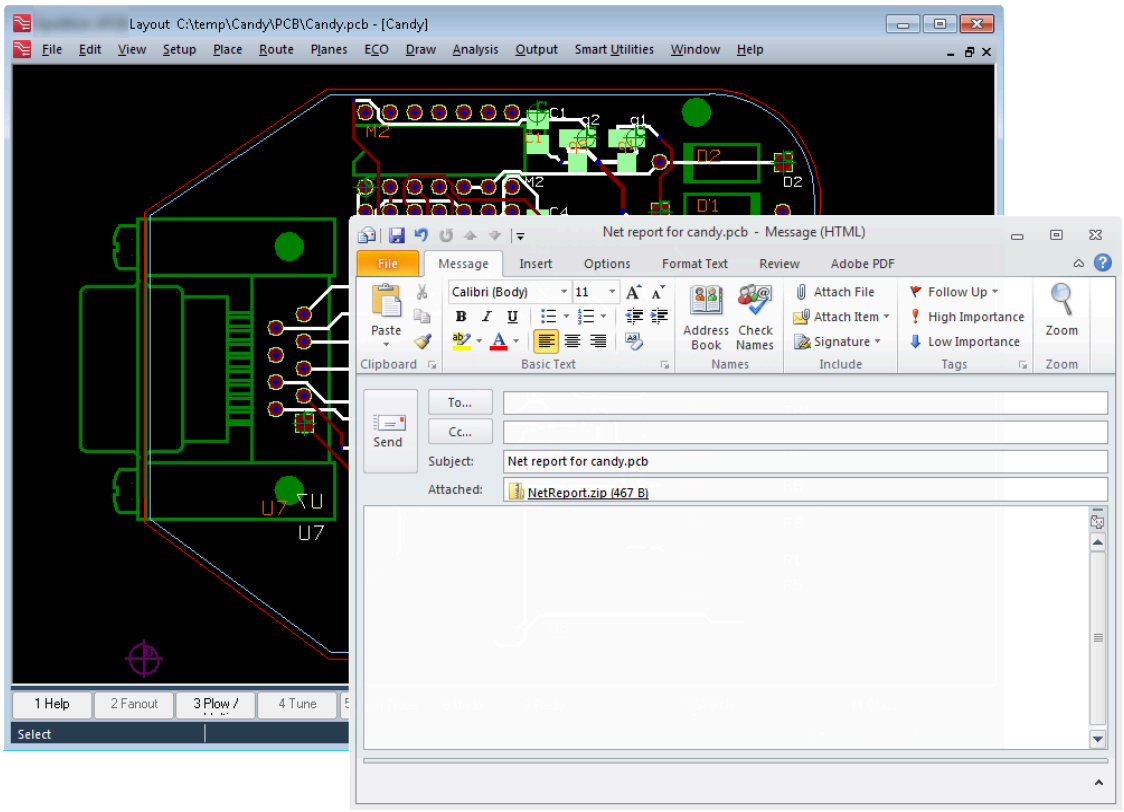
1. Close any instantiations of Xpedition Layout that are currently running, and then launch a new instantiation of this application.

2. Open the *Candy.pcb* design.

3. Drag-and-drop the *ZipAndMailNetReport.vbs* script on the design window.

4. Observe that after the script generates the net report, the script compresses the file and composes the E-mail as illustrated in Figure 139: ZipAndMailNetReport.vbs Script Creates an E-mail.

Figure 139. ZipAndMailNetReport.vbs Script Creates an E-mail



Reading, Analyzing, and Updating a Design

See the following sections.

Exercise 19: Generating Via Caps

Exercise 19: Generating Via Caps

This script creates graphics indicating via locations on a user layer named "Test".

Approach: To read the locations of all of the vias in a design and generate a graphic on a user layer at each of these locations.

Assumptions: A user layer named "Test" exists in the design.

Creating the Script

Use the scripting editor of your choice to open the *Skeleton.vbs* script in your *C:\temp* folder, save the script as *ViaCaps.vbs*, and then replace the comment on Line 18 with the following lines of code:

```
18 ' Set the unit to be used
19  pcbDocObj.CurrentUnit = epcbUnitMils
20
21  ' Start a transaction so that all changes
22  ' are on a single undo level
23  pcbDocObj.TransactionStart
24
25  ' Delete all the existing gfx on Test
26  Dim usrLayerGfxColl
27  Set usrLayerGfxColl = pcbDocObj.UserLayerGfxs(epcbSelectAll, "Test")
28
29  usrLayerGfxColl.Delete
30
31  ' Get the collection of Vias
32  Dim viaColl
33  Set viaColl = pcbDocObj.Vias
34
35  ' Collect parameters for placing graphics on user layer
36
37  ' Get the user layer
38  Dim userLayerObj
39  Set userLayerObj = pcbDocObj.FindUserLayer("Test")
40
41  ' Fill the graphics
42  Dim filledBool: filledBool = True
43
44  ' 0 width
45  Dim widthReal: widthReal = 0
46
47  ' Don't tie to a component
48  Dim cmpObj: Set cmpObj = Nothing
```

```
49
50  ' via cap diameter
51  Dim radiusDbl: radiusDbl = 25
52
53  ' Add circle for each via on user layer "Test"
54  Dim viaObj
55  For Each viaObj In viaColl
56      ' Create a circle points array at the via location
57      Dim pntsArr
58      pntsArr = pcbAppObj.Utility.CreateCircleXYR(viaObj.PositionX, _
59                                          viaObj.PositionY, radiusDbl)
60    Dim numPntsInt
61    numPntsInt = UBound(pntsArr, 2) + 1
62
63    ' Add the graphics to the user layer
64    Call pcbDocObj.PutUserLayerGfx(userLayerObj, widthReal, _
65                              numPntsInt, pntsArr, filledBool, _
66                                      cmpObj, epcbUnitCurrent)
67  Next
68
69  Call pcbAppObj.Gui.StatusBarText("Added " & viaColl.Count & _
70           " via caps on user layer ""Test""", epcbStatusField1)
71
72  pcbDocObj.TransactionEnd
```

Line 19 assigns the *epcbUnitMils* enumerate to the *CurrentUnit* property of the active *Document* object.

```
18  ' Set the unit to be used
19  pcbDocObj.CurrentUnit = epcbUnitMils
```

Line 23 starts a transaction that ensures any changes made by the script is collected into a single **Undo** command/level.

```
21  ' Start a transaction so that all changes
22  ' are on a single undo level
23  pcbDocObj.TransactionStart
```

Line 26 instantiates a variable *usrLayerGfxColl*, which holds a collection of graphic objects. Line 27 returns all of the graphics objects associated with the user layer named "Test" and assigns them to the collection. Line 29 deletes all of the graphics from the collection (this ensures that duplicate graphics are not created when you rerun the script).

```
25  ' Delete all the existing gfx on Test
26  Dim usrLayerGfxColl
27  Set usrLayerGfxColl = pcbDocObj.UserLayerGfxs(epcbSelectAll, "Test")
28
29  usrLayerGfxColl.Delete
```

Line 32 instantiates a variable *viaColl*, which holds a collection of vias. Line 33 returns all of the vias in the design and assigns them to the collection.

```
31  ' Get the collection of Vias
32  Dim viaColl
33  Set viaColl = pcbDocObj.Vias
```

Lines 35 through 51 process all of the parameters for placing graphics. Line 38 and 39 return the user layer "Test". Line 42 indicates these graphics are filled. Line 45 assigns the width for these graphics. Line 48 creates an empty *Component* object that indicates these graphics are not part of any particular component. Line 51 defines a radius for the graphics, which are circles.

```
35  ' Collect parameters for placing graphics on user layer
36
37  ' Get the user layer
38  Dim userLayerObj
39  Set userLayerObj = pcbDocObj.FindUserLayer("Test")
40
41  ' Fill the graphics
42  Dim filledBool: filledBool = True
43
44  ' 0 width
45  Dim widthReal: widthReal = 0
46
47  ' Don't tie to a component
48  Dim cmpObj: Set cmpObj = Nothing
49
50  ' via cap diameter
51  Dim radiusDbl: radiusDbl = 25
```

Line 54 instantiates a variable *viaObj*, which holds a single *Via* object. Lines 55 through 67 loop through each via in the collection; for each via, adds a circular graphic to the user layer "Test".

Line 57 instantiates a variable *pntsArr*, which holds a points array. Line 58 uses the *CreateCircleXYR* method of the *Utility* object to generate the points array for a circle. The script uses the *PositionX* and *PositionY* properties of the current *Via* object to determine the circle's X/Y location.

Lines 60 and 61 determine the number of points in the points array. Line 64 uses the *PutUserLayerGfx* method of the *Document* object to add the circle graphic to the user layer "Test".

```
53  ' Add circle for each via on user layer "Test"
54  Dim viaObj
55  For Each viaObj In viaColl
56      ' Create a circle points array at the via location
57      Dim pntsArr
58      pntsArr =
            pcbAppObj.Utility.CreateCircleXYR(viaObj.PositionX, _
59                        viaObj.PositionY, radiusDbl)
60    Dim numPntsInt
61    numPntsInt = UBound(pntsArr, 2) + 1
62
63    ' Add the graphics to the user layer
64    Call pcbDocObj.PutUserLayerGfx(userLayerObj, widthReal, _
65                              numPntsInt, pntsArr, filledBool, _
66                                      cmpObj, epcbUnitCurrent)
67  Next
```

Line 69 displays a message on the status bar to inform the end-user as to the number of via caps are added to user layer "Test".

```
69  Call pcbAppObj.Gui.StatusBarText("Added " & viaColl.Count & _
70          " via caps on user layer ""Test""", epcbStatusField1)
```

Finally, Line 72 ends the transaction started on Line 23.

```
72  pcbDocObj.TransactionEnd
```

Running the Script

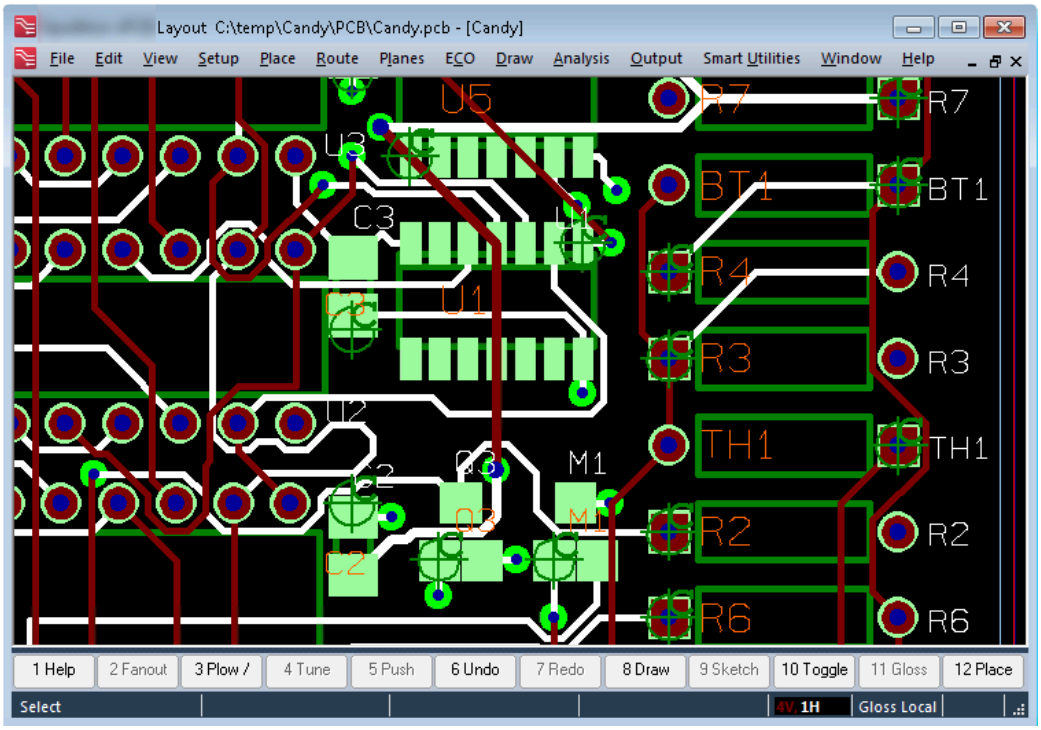
1. Close any instantiations of Xpedition Layout that are currently running, and then launch a new instantiation of this application.

2. Open the *Candy.pcb* design.

3. Choose **View > Display Control** to open the Display Control dialog. Select the **General** tab and ensure that the checkbox next to the user layer named "Test" is checked.

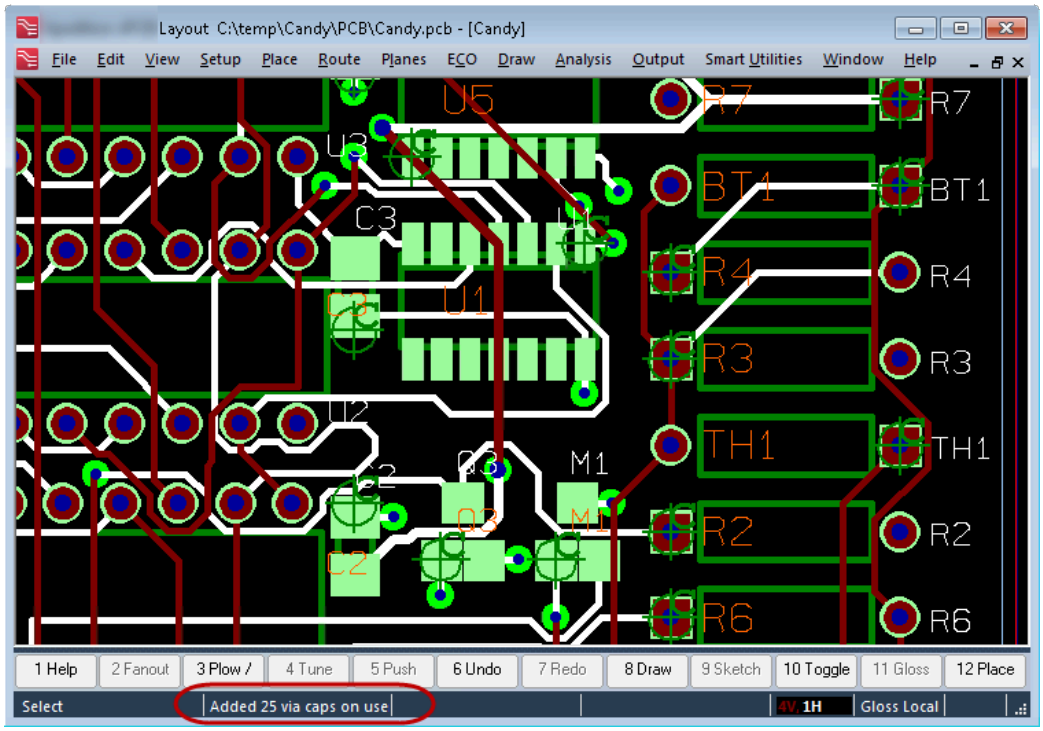
4. Zoom in on a section of the design containing vias as illustrated in Figure 140: Zoom in on Vias.

Figure 140. Zoom in on Vias



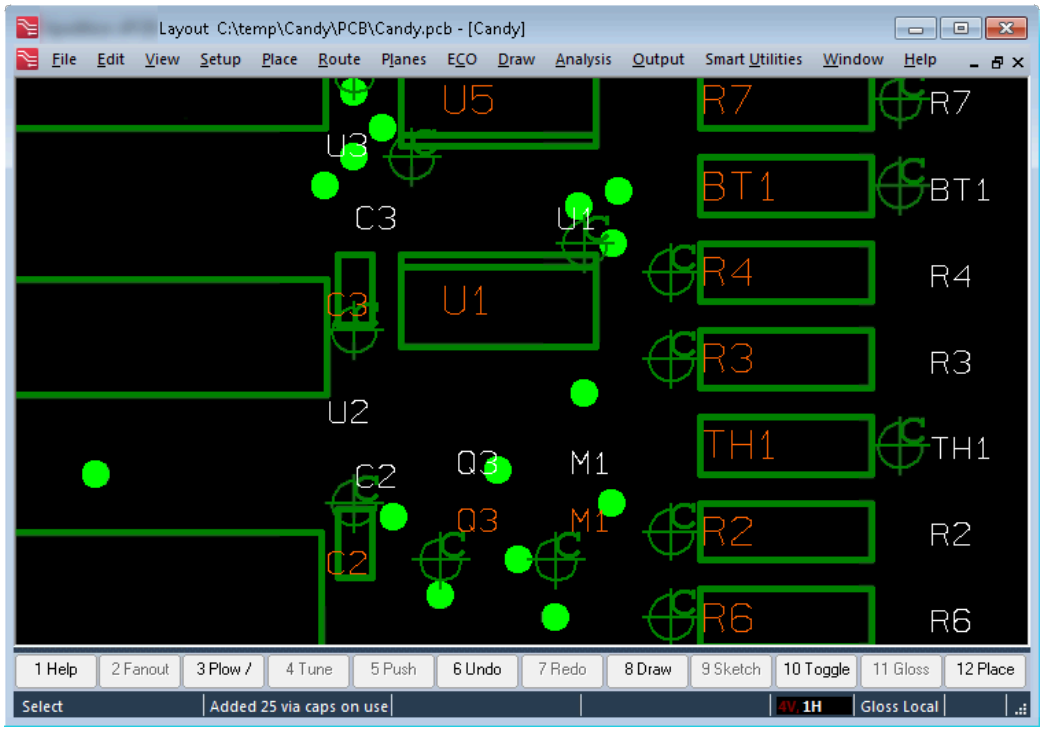
5. Drag-and-drop the *ViaCaps.vbs* script on the design window. Observe the via cap graphics appear as illustrated in Figure 141: ViaCap.vbs Script Adds Via Caps.

Figure 141. ViaCap.vbs Script Adds Via Caps



6. To view these new graphics more clearly, use the Display Control dialog to turn off vias, holes, traces, pads, and so forth as illustrated in Figure 142: Turn Off Graphics to View New Via Caps.

Figure 142. Turn Off Graphics to View New Via Caps



Chapter 14
Xpedition Layout Examples

This section presents several Xpedition Layout script examples.

Exercise 1: Displaying a Single Routing Layer
Exercise 2: Documenting the Layer Stackup
Exercise 3: Performing Assembly DRC
Exercise 4: Removing Unconnected Pads from Pins and Vias
Exercise 5: Integration with Excel
Exercise 6: Generating a Component CSV File
Exercise 7: Finalizing a Design
Exercise 8: Routing by Layer

Exercise 1: Displaying a Single Routing Layer

This script creates a collection of menu entries and accelerators that modify the **View > Display Control** settings. It enables the visibility of a single routing layer (and automatically disables the visibility of any other routing layers).

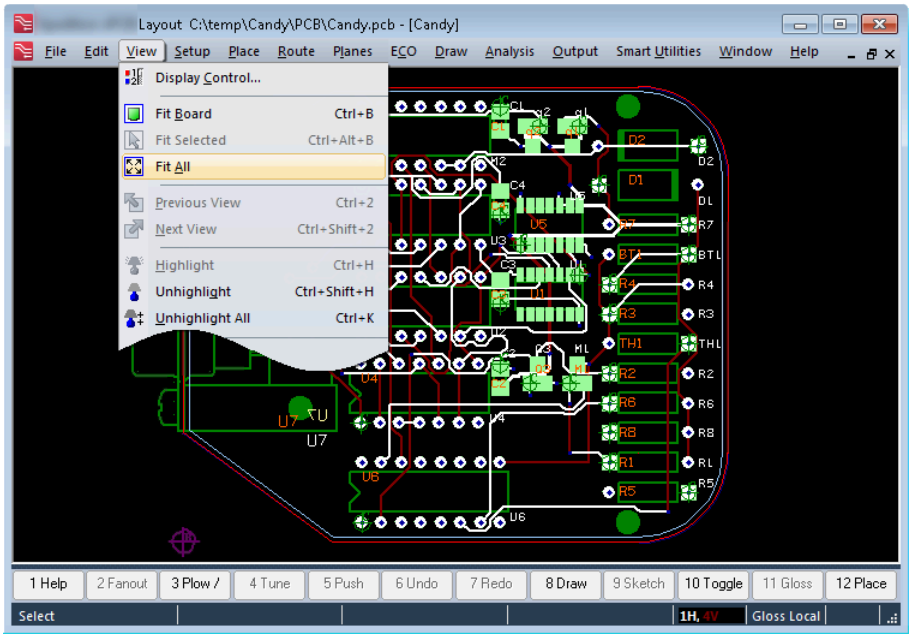
Points of Interest:	– Helper functions to add menu entries
	– Dynamic (run-time) code generation
Items Used:	– Command Bar Server
	– Key Bind Server
	– LayerObject Object
	– DisplayControl Object

Before Starting

Board designs may have different numbers of routing layers. This script searches a board design database to identify the various routing layers. It adds a series of items to the **View** pull-down menu—one for each routing layer (the name of each item reflects the routing layer with which it is associated). The script also establishes a set of accelerators—one for each routing layer. Selecting one of these menu items or accelerator keys displays the associated routing layer and disables all of the other routing layers.

To understand how the script works, launch Xpedition Layout and open a board design. Choose the **View** pull-down menu; you observe something similar to Figure 143: View Menu Before the Script Runs.

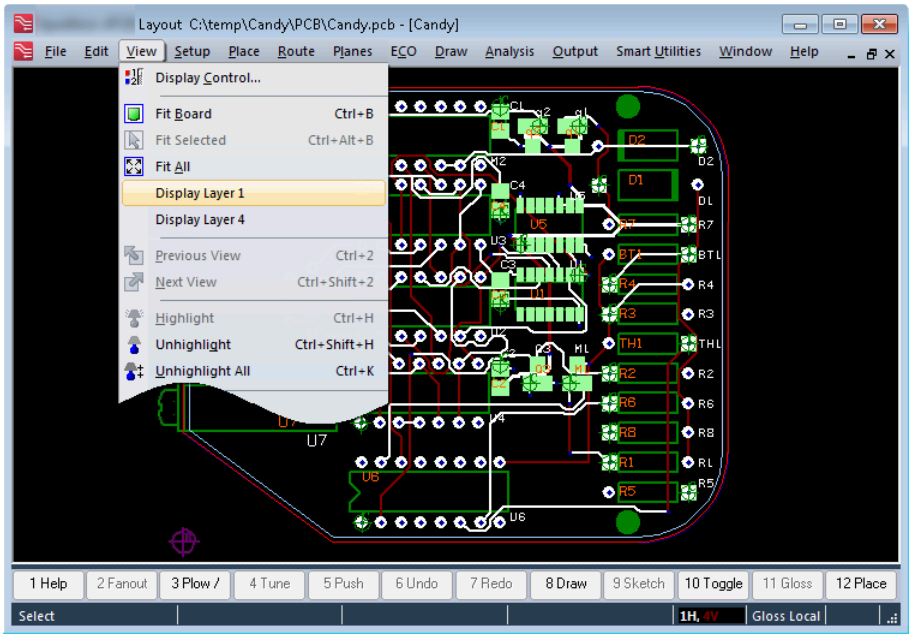
Figure 143. View Menu Before the Script Runs



Observe that the **Previous View** button/item currently follows the **Fit All** button/item.

Run the script *LayerDisplay.vbs* and choose the **View** pull-down menu once more. The script modifies the **View** pull-down menu to appear as shown in Figure 144: View Menu After the Script Runs.

Figure 144. View Menu After the Script Runs



In this example design, there are two routing layers, Layer 1 and Layer 4. If you choose the **View > Display Layer 1** item, the script displays any elements on Layer 1 and disables any elements on Layer 4.

Similarly, if you choose the **View > Display Layer 4** item, the script displays any elements on Layer 4 and disables any elements on Layer 1.

The script also creates two accelerator keys: Ctrl+1 and Ctrl+4 (this means pressing either the 1 or 4 keys while pressing and holding the Ctrl key). Using these accelerators has the same effect as using their corresponding menu items.

Script Sections

The key sections of this script are:

Constant Definitions	Defines the constants the script uses.
Initialization/Setup	Returns the *Application* and *Document* objects, licenses the document, adds the type libraries, and calls the subroutine that adds the menu entries and accelerators.
AddLayerMenus Subroutine	Adds the menu entries and accelerators.
Event Handler Generator Function	Dynamically generates event handlers.
Display Control Subroutine	Manipulates the **View > Display Control** settings.
Helper Functions and Subroutines	Defines menu manipulation "helper routines" that aid with dynamic code generation and adding menu entries.
	–AddMenuAfter() Function
	– GetMenuNumber() Function
Validate Server	Licenses the document and validates the server. Refer to "Running Your First Script" on page 59 for more details on this function.

Constant Definitions

This script section defines the constants the script uses.

Observe Lines 8 through 13. On Line 8, the script instantiates a variable and on Line 9, assigns a value to it. Similarly, on Line 10, the script instantiates a variable and on Lines 11 through 13 assigns a value to it.

As previously noted, this script features dynamic code generation, and the variables declared on Lines 8 and 10 facilitate this. In fact, the script uses these variables as constants, so why does the script use the *Dim* keyword and not the *Const* (for "Constant") keyword? The script uses Dim on Line 10 because the assignment on Lines 11 through 13 involves building a string, and cannot do this with a *Const*. By comparison, the assignment on Line 9 does not involve building a string, so the script can combine Lines 8 and 9 into a single *Const* declaration. The only reason to split it into a *Dim* on Line 8 and an assignment on Line 9 is for consistency with the *Dim* on Line 10 and the assignment on Lines 11 through 13.

```
1  ' Displays a single route layer while turning off display
2  ' of other route layers.  This is done by adding a menu
3  ' entry for each route layer and tying an accelerator
4  ' to that menu entry.                (Author: Toby Rimes)
5  Option Explicit
6
7  ' Strings used for code generation.
```

```
 8  Dim handlerTemplateNameStr
 9  handlerTemplateNameStr = "OnDisplayLayerXX"
10  Dim handlerTemplateStr
11  handlerTemplateStr = "Function OnDisplayLayerXX(nID)" & _
12                vbCrLf & "DisplayLayerOnly(XX)" & vbCrLf & _
13               "End Function"
14
```

Initialization/Setup

Lines 16 and 17 return the *Application* object; Lines 20 and 21 return the *Document* object; Line 24 licenses the document by calling the standard *ValidateServer()* function; and Lines 27 and 28 add the type libraries that this script requires.

```
15  ' Get the application object
16  Dim pcbAppObj
17  Set pcbAppObj = Application
18
19  ' Get the active document
20  Dim pcbDocObj
21  Set pcbDocObj = pcbAppObj.ActiveDocument
22
23  ' License the document
24  ValidateServer(pcbDocObj)
25
26  ' Add the type library so that we can use enums
27  Scripting.AddTypeLibrary("MGCPCB.ExpeditionPCBApplication")
28  Scripting.AddTypeLibrary("MGCSDD.KeyBindings")
29
```

On Line 31, the script calls the subroutine *AddLayerMenus()*. This subroutine adds the new menu buttons/items and creates the accelerator keys.

Even after the script adds the new menu buttons/items and programs the accelerator keys, the script should not exit. Exiting prevents the system from calling the new menu button/item and accelerator event handlers defined in the script. Thus, on Line 33, the script sets the *DontExit* property of the *Scripting* object to the Boolean value of *True*, causing the script to run continuously until the current session ends.

```
30  ' Add the menu entries and accelerators.
31  Call AddLayerMenus()
32
33  Scripting.DontExit = True
34
```

AddLayerMenus Subroutine

Lines 35 through 80 define the subroutine that adds the menu entries and accelerators.

On Lines 39 and 40, the script returns the *Document* object's menu bar.

```
35  ' Sub routine for setting up the menu entries and accelerators.
36  Sub AddLayerMenus()
37
38      ' Get the document menu bar.
```

```
39      Dim docMenuBarObj
40      Set docMenuBarObj = pcbAppObj.Gui.CommandBars("Document
                                                        Menu Bar")
41
```

Lines 43 and 44 return the **View** menu from the collection of controls associated with the *Document* object's menu bar.

```
42      ' Get the view menu
43      Dim viewMenuObj
44      Set viewMenuObj = docMenuBarObj.Controls.Item("&View")
45
```

To add accelerator key bindings, the script returns the *Bindings* object associated with the document.

```
46      ' Get the key bind object.
47      Dim docKeyBindTableObj
48      Set docKeyBindTableObj = pcbAppObj.Gui.Bindings("Document")
49
```

To add a menu button/item for each of the routing layers, the script uses the *LayerStack* property to return a collection of the *LayerObjects* that define the layer stackup. Observe the *False* parameter on Line 52, this indicates to exclude insulation layers from the collection. Therefore, the collection contains only routing (signal) and plane layers.

```
50      ' Get the layer stack
51      Dim layerObjColl
52      Set layerObjColl = pcbDocObj.LayerStack(False)
53
```

Next, the script iterates through the layers in the collection. Line 56 instantiates a variable to control the iteration loop and Line 57 defines the iteration order. Observe that the iteration is in *reverse* order so that the script adds new menu buttons/items from the bottom up; that is, each time the script adds a new button/item, it adds the new item directly under the **View > Fit All** button/item.

```
54      ' Proccess each layer in reverse order so that the
55      ' menus will be in layer order.
56      Dim i
57      For i = layerObjColl.Count To 1 Step -1
```

On Lines 58 and 59, the script returns the *LayerObject* object associated with the current iterator.

```
58          Dim layerObjObj
59          Set layerObjObj = layerObjColl.Item(i)
60
```

Lines 61 and 62 return the layer number associated with the current *LayerObject* object.

```
61          Dim layerInt
62          layerInt = layerObjObj.ConductorLayer
63
```

Line 65 checks that the layer is a routing (signal) layer as opposed to a plane layer.

```
64          ' We only want to process signal layers.
65          If layerObjObj.ConductorType =
                                    epcbConductorTypeSignal Then
```

When the layer is a routing layer (as per the test), Lines 67 and 68 declare a *CommandBarButton* object and then use the AddMenuAfter() Function to create an associated button/item object in the menu.

The first parameter of the AddMenuAfter() Function specifies the existing menu entry after which to insert the new menu entry; the second parameter is the *Menu* object to which to add the new button/item. Refer to Helper Functions and Subroutines for more details on the AddMenuAfter() Function.

```
66              ' Add the menu entry.
67              Dim displayLayerBtnObj
68              Set displayLayerBtnObj = AddMenuAfter("Fit All”,
                                                    viewMenuObj)
```

Line 69 defines the name to use for the new menu button/item. This name is built dynamically using the number of the current layer. Line 70 defines the description text to associate with this button/item.

```
69              displayLayerBtnObj.Caption = "Display Layer " & layerInt
70              displayLayerBtnObj.DescriptionText = "Sets
                              display to view only a single layer."
```

Line 71 defines the target where the menu handler resides. The script requires a “button-press” event handler function for each button – that is, for each layer. Because the script does not recognize the number of layers until run time, the script must create these functions dynamically at run time. Thus, on Line 72, the script calls the Event Handler Generator Function and passes it the layer number for the event handler to generate. The return value from this function is the name of the event handler, which the script assigns to the *ExecuteMethod* property.

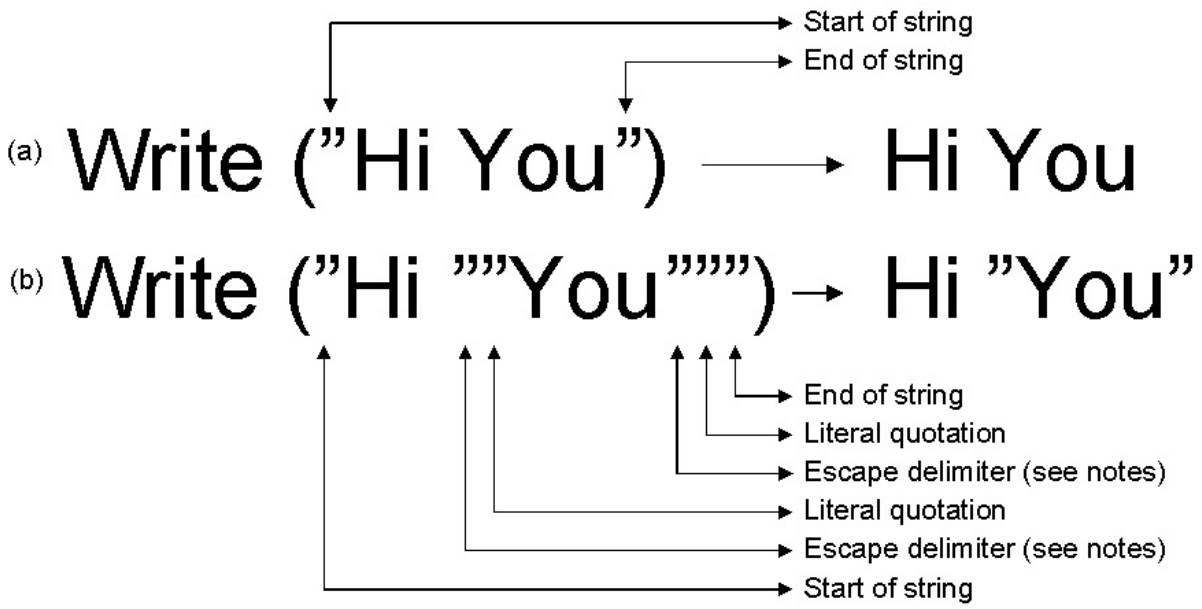
```
71              displayLayerBtnObj.Target = ScriptEngine
72              displayLayerBtnObj.ExecuteMethod =
                                  GenerateDisplayHandler(layerInt)
73
```

The script ties an accelerator key to the newly created menu item. Lines 75 through 77 show this by means of the keyin command **xm** ("execute menu"). These accelerator keys are Ctrl+<Layer Number>; that is, press the number of the layer while pressing and holding the Ctrl key.

```
74              ' Add an accelerator.
75              Call docKeyBindTableObj.AddKeyBinding("Ctrl+" & layerInt,_
76              "xm ""View->Display Layer "& layerInt & """" , _
77                                    BindCommand, BindAccelerator)
78          End If
79      Next
80  End Sub
81
```

Observe the single, dual, and quadruple double-quotation characters (", "", and """") on Line 76. This can be a bit complicated. To understand using multiple quotes refer to the example in Figure 145: Single, Double, and Quadruple Double-quotes.

Figure 145. Single, Double, and Quadruple Double-quotes



First, consider the example shown in Figure 145: Single, Double, and Quadruple Double-quotes(a). The first double-quote begins the string, next is the text (*Hi You*), and finally the second double-quote ends the string. The text the statement generates is *Hi You*.

Next, consider the more complex example shown in Figure 145: Single, Double, and Quadruple Double-quotes(b).

- The first double-quote begins the string.
- The system recognizes the second double-quote as an escape delimiter, because another double-quote follows. Thus, the third double-quote is a literal character to output.
- Similarly, the fourth double-quote is an escape delimiter informing the system that the fifth double-quote is a literal character to output.
- Finally, the sixth double-quote ends the string.
- The text the statement generates is *Hi "You"*.

As the script iterates in the code, the strings that Line 76 generates appears as:

```
xm "View->Display Layer 1"
xm "View->Display Layer 4"
```

For more information on using multiple double-quotes, refer to Figure 180: Triple and Quadruple Double-quotes on page 382 discussions.

Event Handler Generator Function

Lines 86 through 93 define a function that dynamically generates a unique event handler for each layer.

The function declaration on Line 86 accepts an integer parameter that defines a layer number. The *ExecuteGlobal()* function on Line 89 is a VBScript function that takes a string and runs it in the context of the running script. If this string defines a function, *ExecuteGlobal()* makes the function available, it does not actually perform the function.

Observe the parameters passed into the *ExecuteGlobal()* function on Line 89. In order to generate this parameter, the script uses the "constant" declared on Lines 10 through 13 as a template. The script substitutes a substring in the template with the layer number to generate the unique handler. A similar trick exists on Line 92 for generating the function name returned with this handler.

```
82  ' Function for generating the command handlers at runtime.
83  ' This is required becuase the layer stack is only known
84  ' at runtime.
85  ' layerInt - Integer
86  Function GenerateDisplayHandler(layerInt)
87      ' Generate handler function for this layer. Calling
88      ' ExecuteGlobal makes this func. available to the script.
89      ExecuteGlobal(Replace(handlerTemplateStr, "XX", "" & layerInt))
90
91      ' Return the name of the generated handler function.
92      GenerateDisplayHandler = Replace(handlerTemplateNameStr,
                                         "XX", "" & layerInt)
93  End Function
94
```

Note:
Dynamically generated code is difficult to debug, therefore it should be used only for simple functions. In the example, the script creates a basic function that calls a more complex helper function. The more complex (but non-dynamically created) helper function performs the display manipulations.

In the *Candy.pcb* design, the routing layers are 1 and 4. Thus, if you create the handlers by hand, they are as follows (the line numbers will be different). The previous function generates these handlers dynamically (except for comments):

```
1  ' Layer 1 handler
2  Function OnDisplayLayer1(nID)
3      DisplayLayerOnly(1)
4  End Function
5
6  ' Layer 4 handler
7  Function OnDisplayLayer4(nID)
8      DisplayLayerOnly(4)
9  End Function
```

Display Control Subroutine

Lines 95 through 132 define a subroutine that sets a specified layer to the active layer, displays the active layer, and disables the display of all other routing (signal) layers.

Observe that on Line 100, the script locks the server to improve performance because the script performs display manipulations. Server locking prevents the display from updating until Line 130 calls the *UnlockServer* method.

```
 95  ' Sub routine to set this layer to that active
 96  ' layer and turn off display of all other layer.
 97  ' layerInt - Integer
 98  Sub DisplayLayerOnly(layerInt)
 99      ' Lock the server for better performance
100      pcbAppObj.LockServer
101
```

On Lines 104 and 105, the script determines the number of layers in the design, and on Line 107 it verifies that this is a valid number.

```
102      ' Determine how many layers there are
103      Dim layerCntInt
104      layerCntInt = pcbDocObj.LayerCount
105
106      ' Make sure the layer is valid for this job
107      If layerInt > 0 And layerInt <= layerCntInt Then
```

Lines 109 and 110 return the *DisplayControl* object.

```
108          ' Get the display control object
109          Dim dispCntrlObj
110          Set dispCntrlObj = pcbDocObj.ActiveViewEx.DisplayControl
111
```

Line 114 activates the layer of interest and Line 117 displays this layer.

```
112          ' Set layer of interest to the active layer to allow
113          ' us to turn off other layers.
114          pcbAppObj.Gui.ActiveRouteLayer = layerInt
115
116          ' Turn on the layer
117          dispCntrlObj.ConductorLayer(layerInt, epcbDCTrace) = True
118
```

On Lines 120 through 126, the script loops, turning every layer, except the active layer, off. It is not possible to disable the display of the active layer, so the script does not require the test on Line 123, but it includes this test to show what the script trying to achieve.

```
119          ' Turn off all the other layers
120          Dim i
121          For i = 1 To layerCntInt
122              ' If layer is not one we want on then turn off
123              If Not i = layerInt Then
124                  dispCntrlObj.ConductorLayer(i, epcbDCTrace) = False
125              End If
126          Next
127      End If
128
```

Finally, on Line 130, the script unlocks the server and on Line 132 ends the subroutine.

```
129      ' Unlock the server
130      pcbAppObj.UnlockServer
131
132  End Sub
133
```

Helper Functions and Subroutines

This script section declares several generic "helper" routines. These routines can be used in other scripts and are not focused on this script exclusively.

AddMenuAfter() Function

Lines 142 through 149 declare the *AddMenuAfter()* function, which adds a new menu button/item after an existing menu button/item.

The return value from the *AddMenuAfter()* function – as defined on Line 147 – specifies the location where the new menu button/item is added into the menu. Observe that, on Line 144, this function calls the GetMenuNumber() Function.

```
134  ' Menu manipulation functions.
135
136  ' Creates a new menu entry, menuToAdd, on menuBar after the
137  ' afterMenuEntry entry.  If the entry is not found the menu is
138  ' added at the end.
139  ' menuToAddStr - String
140  ' afterMenuEntryStr - String
141  ' menuBarObj - CommandBarSvr menu object
142  Function AddMenuAfter(afterMenuEntryStr, menuBarObj)
143      Dim entryNumInt
144      entryNumInt = GetMenuNumber(afterMenuEntryStr,
                                                 menuBarObj) + 1
145      If entryNumInt > menuBarObj.Controls.Count Then
146         entryNumInt = -1
147      End If
148      Set AddMenuAfter =
            menuBarObj.Controls.Add(cmdControlButton,,,entryNumInt)
149  End Function
```

GetMenuNumber() Function

Lines 155 through 172 declare the *GetMenuNumber()* function, which returns the menu index number associated with a given menu button/item string.

In this case, the *GetMenuNumber()* function may be considered to be a "helper's helper". It performs its task by iterating through all of the buttons/items and comparing their captions with the one it is searching for. The value returned by this function – as defined on Line 166 – is the required menu index number.

```
150
151  ' Function that returns a menu index for a given
152  ' menu entry string.
153  ' menuToFindStr - String
154  ' menuBarObj - CommandBarSvr menu object
```

```
Function GetMenuNumber(menuToFindStr, menuBarObj)
    Dim ctrlColl : Set ctrlColl = menuBarObj.Controls
    Dim ctrlObj

    Dim menuCntInt : menuCntInt = 1
    GetMenuNumber = -1

    For Each ctrlObj In ctrlColl
       Dim captStr: captStr = ctrlObj.Caption
       captStr = Replace(captStr, "&", "")
       If captStr = menuToFindStr Then
           GetMenuNumber = menuCntInt
           Exit For
       End If
       menuCntInt = menuCntInt + 1
    Next

End Function
```

Creating and Testing the Script

1. Use the scripting editor of your choice to enter the script described (including the license server function) and save it as *LayerDisplay.vbs*.

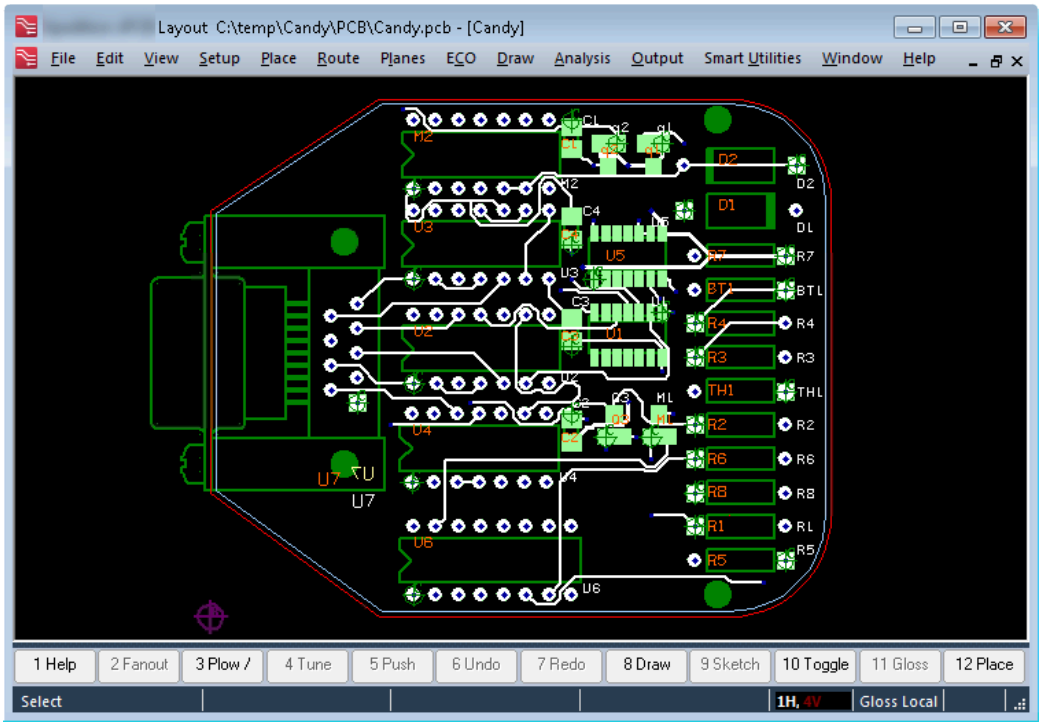
2. Close any instantiations of Xpedition Layout that are currently running, and then launch a new instantiation of this application.

3. Open the *Candy.pcb* design.

4. Run the *LayerDisplay.vbs* script.

5. Choose the new **View > Display Layer 1** menu item (or use the new Ctrl+1 accelerator) and observe the results as illustrated in Figure 146: View > Display Layer 1.

Figure 146. View > Display Layer 1

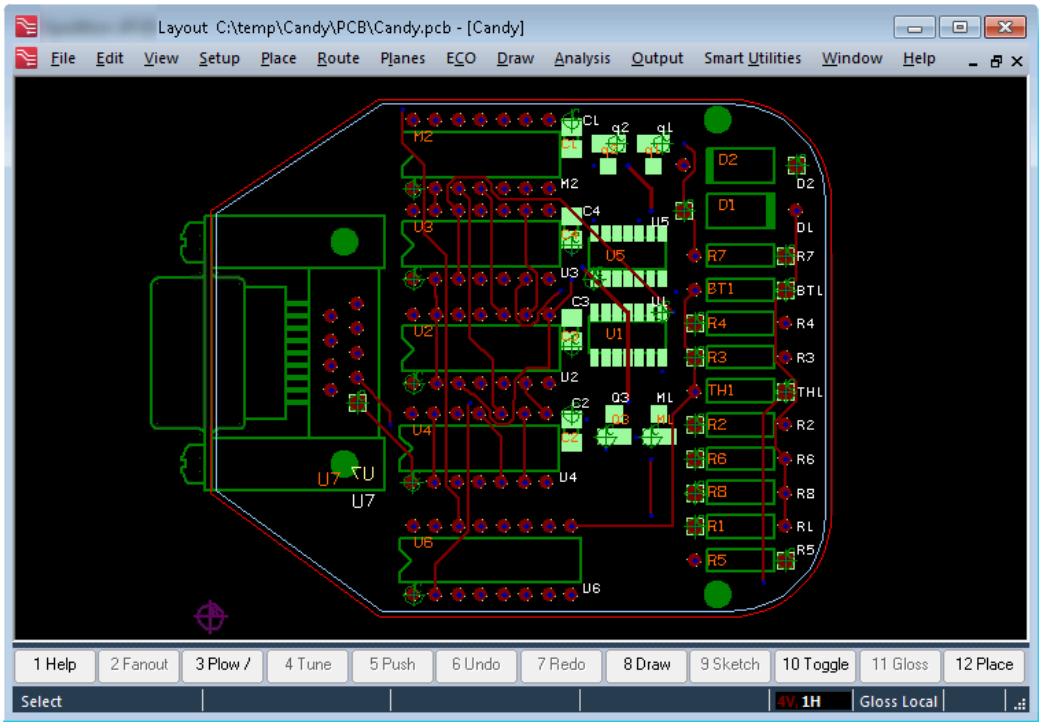


6. Observe that only white signals from routing layer 1 are displayed, while any red signals from routing layer 4 are disabled.

7. Choose the new **View > Display Layer 4** menu item (or use the new Ctrl+4 accelerator) and observe the results as illustrated in Figure 147: View > Display Layer 4.

8. Observe that only red signals from routing layer 4 are displayed, while any white signals from routing layer 1 are disabled.

Figure 147. View > Display Layer 4



9. Close this instantiation of Xpedition Layout.

10. Launch a new instantiation of this application.

11. Open one of your own designs that contain more than two routing layers.

12. Run the *LayerDisplay.vbs* script.

13. Check that the script creates new menu buttons/items, creates accelerator keys and that these perform as you expect.

Enhancing the Script

There are a number of ways to enhance this script. Several suggestions follow (practice your scripting skills by implementing these suggestions):

- Set this script to run as a startup script (refer to "Running Startup Scripts" on page 171).

- Modify the script to turn display of pads on or off for each layer.

- Modify the script such that whenever a new document opens the script automatically removes any menu entries (and accelerators) associated with the previous design and then adds new menu entries (and accelerators) for the open design.

Exercise 2: Documenting the Layer Stackup

This script documents the layer stackup by creating text and graphics on a user layer that represents the layer stackup.

Approach:	To use motion graphics to present the end-user with a view of the layer stackup that can be moved and then placed on a user layer.
Points of Interest:	– Implementing a command using motion graphics
Items Used:	– Command Object
	– Motion Graphics Methods

Before Starting

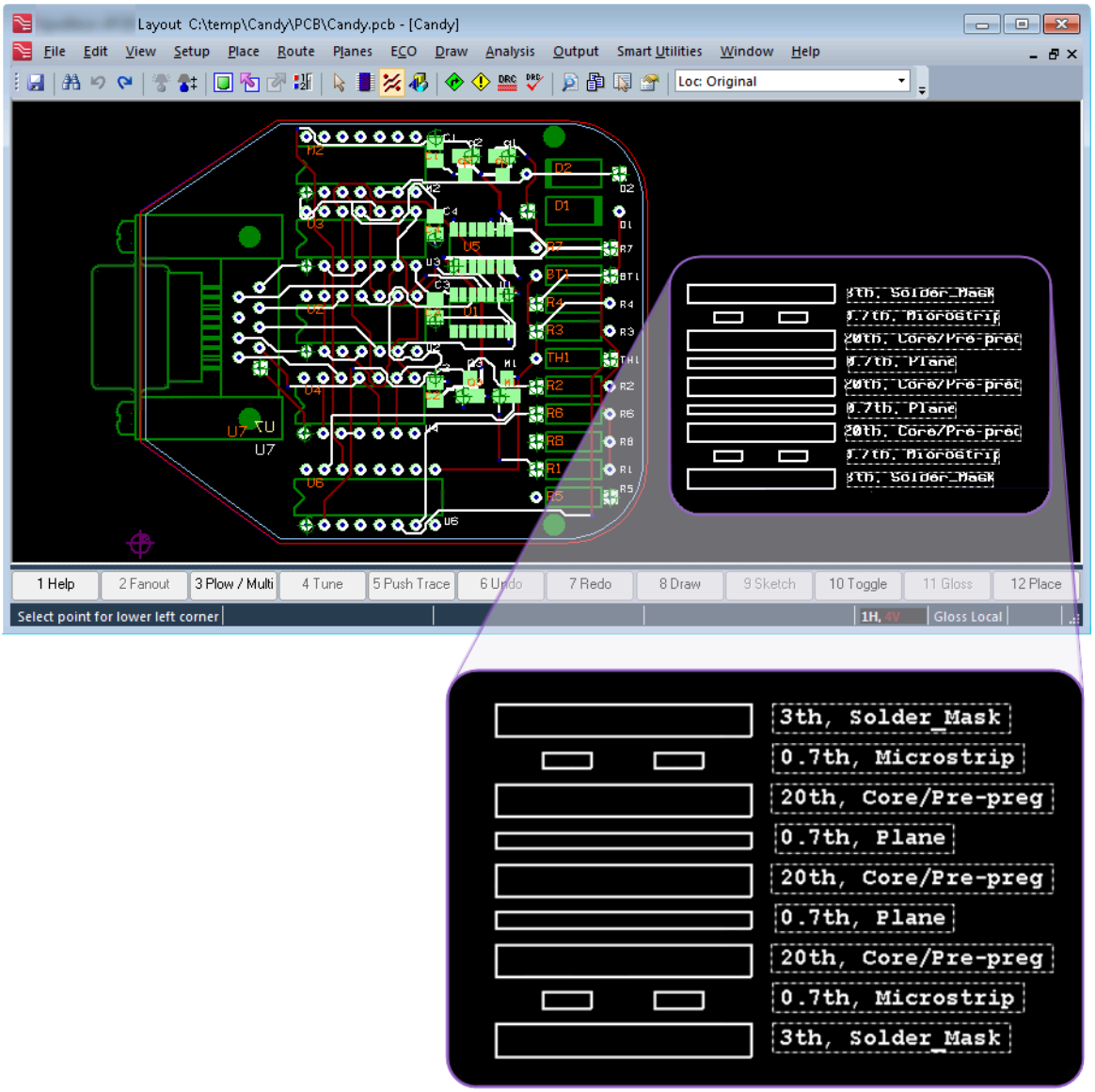
The term "motion graphics" refers to special graphical representations of text and objects that can be "attached" to the mouse cursor. When the mouse moves the motion graphics follow the cursor. You can use motion graphics for a variety of purposes, including documenting the layer stackup as this example script illustrates.

When you are working with a layout design (created by yourself or by someone else), you can use **Setup > Setup Parameters** to examine the layer stackup. However, it can be useful to display this information graphically in the design itself.

To understand how the script works, launch Xpedition Layout, open the *Candy.pcb* layout design, and run the script *LayerStackup.vbs*. When running the script, motion graphics appear on the screen as illustrated in Figure 148: Motion Graphics Generated by the Script.

Observe that the graphical items are shown as white outlines with no fill; also that the text items are shown with dotted lines around them. If you move the mouse, these motion graphics follow the cursor (hence their "motion" name).

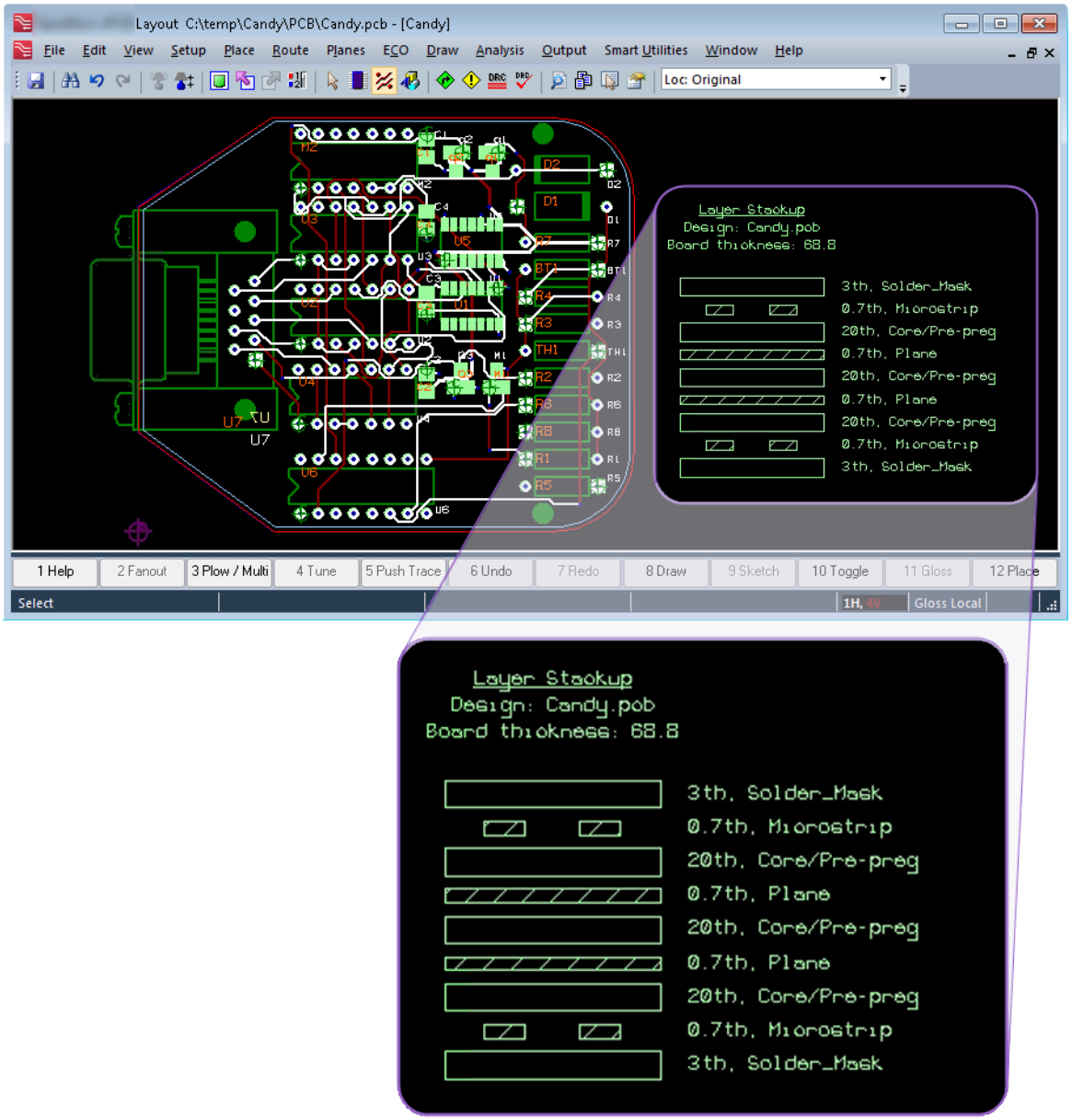
Figure 148. Motion Graphics Generated by the Script



When you place the motion graphics in the required location, clicking the mouse button causes the script to create actual graphics on a user layer as illustrated in Figure 149: "Real" Graphics Placed on a User Layer.

Observe that some graphical items now have a fill pattern; also that text items no longer have dotted lines around them; also that header information is added.

Figure 149. "Real" Graphics Placed on a User Layer



Script Sections

The key sections of this script follow:

Constant Definitions	Defines the constants the script uses.
Initialization/Setup	Returns the *Application* and *Document* objects, licenses the document, and adds the type libraries. Also performs any additional setup associated with this script.

Event Handlers	Defines the event handlers, which detects and processes mouse clicks, for the *Command* object.
Generate Motion Graphics Function	Generates the motion graphics on the screen.
Generate User-Layer Graphics Function	Generates the actual graphics and places them on the user layer.
Utility Functions and Subroutines	Defines special "workhorse" routines that are focused on this script.
	– GenerateHeaderOnUserLayer() Function
	– GenerateStackupGraphics() Function
	– DrawDielectricLayer() Function
	– DrawSignalLayer() Function
	– DrawPlaneLayer() Function
Helper Functions and Subroutines	Defines generic "helper" routines that other scripts can use.
	– ConfigureUserLayer() Function
	– GetLayerDescription() Function
	– DrawText() Function
	– DrawRectangle() Function
Miscellaneous Functions and Subroutines	Defines additional routines that do not fit in the "utility" or "helper" classes
	– ClearMotionGraphics() Function
Validate Server	Licenses the document and validates the server. Refer to "Running Your First Script" on page 59 for more details on this function.

There are many details with regard to generating and manipulating graphics. Thus, it makes things easier to consider first, a high-level (pseudo-code) representation of the actions the script performs:

```
DrawStackupMotionGraphics()
    ClearMotionGraphics()
    GenerateStackupGraphics()
        For all layers
            DrawDielectricLayer()
                DrawRectangle()
                DrawText()
            DrawSignalLayer()
                DrawRectangle()
                DrawRectangle()
                DrawText()
```

```
            DrawPlaneLayer()
                DrawRectangle()
                DrawText()
        End For
```

```
cmdObj_OnMouseClk()
    ClearMotionGraphics()
    ConfigureUserLayer()
    GenerateStackupGraphicsOnUserLayer()
        GenerateStackupGraphics()
            For all layers
                DrawDielectricLayer()
                    DrawRectangle()
                    DrawText()
                DrawSignalLayer()
                    DrawRectangle()
                    DrawRectangle()
                    DrawText()
                DrawPlaneLayer()
                    DrawRectangle()
                    DrawText()
            End For
    GenerateHeaderOnUserLayer()
```

The script has two main sections. The first section draws the stackup motion graphics. After clearing any existing motion graphics, the script generates the graphic and text items associated with each dielectric, signal, and plane layers of the board. The script creates these items as motion graphics, which means they automatically follow the mouse cursor until the end-user clicks a mouse button.

The second portion of the script addresses what to do when the end-user clicks a mouse button. The script clears any existing motion graphics, configures a user layer, and regenerates the graphic and text items associated with each dielectric, signal, and plane layers of the board. This graphics and text generation process is identical to the first portion of the script, except that the script creates these items on the user layer. The script finishes by creating some header information on the user layer.

Constant Definitions

In the first part of the script (Lines 8 through 24), the script declares the constants it uses throughout the remainder of the script. These constants define the size (width and height) of the various graphical rectangles that are drawn, the offsets of these graphical elements with respect to each other, and the offsets of the text elements with respect to the graphical elements. Having all of these elements declared at the beginning of the script makes it easy to modify the elements.

```
  1  ' This script implements a command for placing a layer stackup
  2  ' representation on a user layer.  Motion graphics are used to
  3  ' show the layer stackup documentation during placement.
  4  ' (Authors: Michael Lowder, John Dube, Toby Rimes)
  5  Option Explicit
  6
  7  ' The unitsEnum of all the constants below are in mils
  8  Const DIELECTRICBOXX = 800   'box width for dielectric layers
  9  Const DIELECTRICBOXY = 100   'box height for dielectric layers
 10  Const TRACEBOXX = 150        'box width for trace layers
 11  Const TRACEBOXY = 50         'box height for trace layers
```

```
12  Const TRACEBOX1OFFSETX = 150 'start position of 1st trace box
13  Const TRACEBOX2OFFSETX = 500 'start position of 2nd trace box
14  Const PLANEBOXX = 800        'box width for plane layers
15  Const PLANEBOXY = 50         'box height for plane layers
16  Const TEXTOFFSETX = 900      'X offset, text for each layer
17  Const TEXTOFFSETYDIE = 50    'Y offset, text for dielectric
18  Const TEXTOFFSETYCOND = 25   'Y offset, text for conductive
19  Const TEXTHEIGHT = 50        'height of text
20  Const USERLAYERNAME = "Layer Stackup"
21  Const STEPCONDY = 150        'spacing above conductive layer
22  Const STEPDIEY = 100         'spacing above dielectric layer
23  Const STEPTITLEY = 100       'spacing in title area
24  Const TITLETEXTOFFESTX = 400 'start position of title text
```

Initialization/Setup

On Line 27, the script adds the type library to use:

```
26  ' Add any type libraries to be used.
27  Scripting.AddTypeLibrary("MGCPCB.ExpeditionPCBApplication")
28
```

On Lines 30 through 34, the script declares the global variables. In addition to the usual *Application* and *Document* objects, it also declares a variable to hold a *Command* object.

```
29  ' Global variables
30  Dim cmdObj                   'Command object
31  Dim pcbAppObj                'Application object
32  Dim pcbDocObj                'Document object
```

On Line 33, the script declares a "tag" and initializes it with a value of -1. When the script creates any motion graphics it associates this tag with them; this allows you to access them and delete them later.

```
Dim tagMotionInt : tagMotionInt = -1 ' Tag for identifying
                                       motion graphics.
```

On Line 34, the script defines the specific units to associate with this script (apart from anything else, these units apply to the constants the script declared on Lines 8 through 24).

```
34  Dim unitsEnum : unitsEnum = epcbUnitMils ' Units to be used
                                               throughout
```

Lines 36 through 43 perform the usual steps of acquiring an *Application* object, acquiring a *Document* object, and licensing the document (validating the server).

```
36  ' Get the application object.
37  Set pcbAppObj = Application
38
39  ' Get the active document
```

```
40  Set pcbDocObj = pcbAppObj.ActiveDocument
41
42  ' License the document
43  ValidateServer(pcbDocObj)
```

Line 46 specifies the units to associate with the document.

```
45  ' Set the unitsEnum to match the hard coded values.
46  pcbDocObj.CurrentUnit = unitsEnum
```

On Line 49, the script uses the *RegisterCommand* method to create a *Command* object, which becomes the active command at this time. Essentially the script is saying: *"This is now the active command, so any events should come to this command."* In addition, the string parameter "Place Layer Stackup" is the text that appears in the *Prompt* field on the status bar. This allows the script to receive notification of a right-mouse click so that it knows when and where to place the user layer graphics. This also prevents the system from receiving mouse events while the command is active; that is, as long as the **Place Layer Stackup** command is the active command, the main program will not process mouse events.

```
48  ' register a cmd so we can get a mouse point
49  Set cmdObj = pcbAppObj.Gui.RegisterCommand("Place Layer
                                                  Stackup")
```

Line 52 defines the units to associate with the *Command* object. This means that when the script returns a mouse-click event, the coordinates associated with the mouse cursor report in terms of these units. The reason to do this is that the *Command* object does not inherit anything from the *Document* object.

```
51  ' Command unitsEnum are separate from the doc so set those too.
52  cmdObj.Unit = unitsEnum
```

Line 55 formally attaches any events to the *Command* object.

```
54  ' attach the events to the command
55  Call Scripting.AttachEvents(cmdObj, "cmdObj")
```

On Line 58, the script calls the function that creates the stackup motion graphics and passes it the current position of the mouse cursor. This is because the initial position of the motion graphics defines the current location of the mouse cursor. At this point, the motion graphics appears on the screen, after which the graphics automatically follow the position of the mouse cursor (until the end-user clicks a mouse button).

```
57  ' Create the motion graphics
58  Call GenerateStackupMotionGraphics(pcbDocObj.ActiveViewEx.
               MousePositionX, pcbDocObj.ActiveViewEx.MousePositionY)
```

Line 62 defines the display of a message in the status bar to prompt the end-user to click the mouse button to place the layer stackup graphics on a user layer.

```
61  ' set the prompt field in the status bar
62  Call pcbAppObj.Gui.StatusBarText("Select point for
         lower left corner of stackup...", epcbStatusFieldPrompt)
```

Line 66 sets the *DontExit* property of the *Scripting* object to the Boolean value of *True*. This means this script continues running until the end of the current application session, thereby allowing the script to detect and handle events.

```
65  ' Hang around to listen to events
66  Scripting.DontExit = True
```

Event Handlers

Lines 71 through 87 describe the mouse-click event handler. The way this works is detailed in “Creating Interactive Commands” on page 248.

On Line 79, the script calls a function to delete any existing motion graphics. Line 81 calls a function to recreate the graphics on the user layer. Line 83 sets the return value of this function to *True* to indicate to the calling script that this function was used. Now, only the first mouse-click is of interest, so Line 85 terminates the top-level command (which is the command) and returns control to whatever was active prior to running this script.

```
68  '''''''
```

On Line 115, the script calls a function to generate the stackup graphics; the first parameter to this function defines the user layer on which to add any graphics. To instruct the function to create the graphics as motion graphics, pass a value of *Nothing*.

On Line 119, the script uses the *MotionGfxDrawAnyUndrawn* method to force the graphics to draw without the having to wait for the end-user to move the mouse cursor.

```
101  ' Motion Graphics Generation Function
102
103  ' Main function for creating the stackup motion grahics
104  ' xReal - Real
105  ' yReal - Real
106  Function GenerateStackupMotionGraphics(xReal, yReal)
107      ' Remove any existing motion gfx.
108      Call ClearMotionGraphics()
109
110      ' Get a new tag for the motion gfx.
111      tagMotionInt = pcbDocObj.ActiveViewEx.MotionGfxGetTag()
112
113      ' Generate the motion gfx.
114      Dim totalThicknessReal
115      Call GenerateStackupGraphics(Nothing, xReal, yReal,
                                              totalThicknessReal)
116
117      ' Make sure everything is drawn now so we don't
118      ' have to wait for the first mouse move.
119      Call pcbDocObj.ActiveViewEx.MotionGfxDrawAnyUndrawn()
120  End Function
```

Generate User-Layer Graphics Function

Lines 125 through 152 represent the main function used to generate the stackup graphics on the user layer. This is very similar to the previous function, but there are some differences to consider in more detail.

On Lines 131 and 132, the script instantiates and configures a *UserLayer* object.

```
122  '''''''''''''''''''''''''''''''''''''
123  ' User Layer Generation Function
124
125  ' Main function for drawing graphics on the user layer
126  ' Input is the bottom left position of the graphics
127  ' xReal - Real
128  ' yReal - Real
129  Function GenerateStackupGraphicsOnUserLayer(xReal, yReal)
130      ' Ensure the user layer exists
131      Dim usrLyrObj
132      Set usrLyrObj = ConfigureUserLayer(USERLAYERNAME)
```

Line 135 starts a transaction, which allows the undo of any changes made by this function with a single **Undo** command.

```
134      ' start the transaction
135      pcbDocObj.TransactionStart
```

Line 137 defines a variable to record the total thickness of the board and initializes this value to 0.0. The script uses this value as part of the header information. (Observe that the script declared a similar parameter on Line 114 in the motion graphics function, but did not initialize it there as part of the motion graphics.)

```
137      Dim totalThicknessReal : totalThicknessReal = 0.0
```

On Lines 141 and 142, the script calls a function to generate the stackup graphics on the user layer specified by the first parameter. It passes in the X and Y values associated with the current position of the mouse cursor. It also passes in the variable to hold the total thickness of the board.

```
139      ' Generate the graphics and get the new y position
140      ' for the top of the graphics.
141      Dim curYReal
142      curYReal = GenerateStackupGraphics(usrLyrObj, xReal, yReal,
                                            totalThicknessReal)
```

The value this function returns is the Y-axis value associated with the highest (topmost) graphic. Also, the value of *totalThicknessReal* updates to reflect the total thickness of the board. This is because the *totalThicknessReal* parameter, is an In-Out parameter (in fact, *all* of the parameters are In-Out parameters).

Line 145 increases the Y-axis value to add in a constant value for the header the script creates. When combined with the X-axis value (which is not changed), the new Y-axis value defines the location of the bottom left-hand corner of the header.

```
144      ' step the Y position
145      curYReal = curYReal + STEPTITLEY
```

On Line 148, the script calls a function to generate the header and add it to the user layer.

```
147      ' Create header at updated yReal position
148      Call GenerateHeaderOnUserLayer(usrLyrObj, xReal, curYReal,
                                            totalThicknessReal)
```

Line 151 ends the transaction initiated on Line 135.

```
150      ' end the transaction
151      pcbDocObj.TransactionEnd
```

Line 154 clears the prompt field in the status bar.

```
153      ' clear the prompt field in the status bar
154      Call pcbAppObj.Gui.StatusBarText("", epcbStatusFieldPrompt)
155  End Function
```

Utility Functions and Subroutines

This script section declares utility routines. These routines are similar to the generic Helper Functions and Subroutines, except that they are specific to this script and may not be used elsewhere without changes.

GenerateHeaderOnUserLayer() Function

Lines 161 through 203 declare the function that generates the header information on the user layer. The parameters to this function are the user layer on which to place the graphics, the X and Y values of the bottom left-hand corner of the header, and the total thickness of the board.

On Line 167, the script declares a local variable to hold a copy of the Y value (the script modifies the value of this copy).

```
161  ' Function for drawing header information for stackup graphics.
162  ' usrLyrObj - UserLayer object
163  ' xReal - Real
164  ' yReal - Real
165  ' totalThicknessReal - Real
166  Function GenerateHeaderOnUserLayer(usrLyrObj, xReal, yReal,
                                                totalThicknessReal)
167      Dim curYReal: curYReal = yReal
```

On Lines 169 and 170, the script creates a string to describe the thickness of the board.

```
168      ' now we need to add the header text
169      Dim textStr
170      textStr = "Board thickness: " & totalThicknessReal
```

Lines 173 through 177 take the string just created and put it on the user layer.

```
172      ' put the text string on the user layer
173      Call pcbDocObj.PutUserLayerText(textStr, _
174          xReal + TITLETEXTOFFESTX, curYReal, usrLyrObj, _
175          TEXTHEIGHT, 0, 5, "VeriBest Gerber 0", _
176          , epcbJustifyHCenter, epcbJustifyBottom, _
177          Nothing, unitsEnum)
```

Line 180 modifies the Y-axis value.

```
179      ' step the Y position
180      curYReal = curYReal + STEPTITLEY
```

Line 183 creates a string to describe document name.

```
182      ' now we need to add the header text
183      textStr = "Design: " & pcbDocObj.Name
```

Lines 185 through 190 take the string just created and puts it on the user layer.

```
185      ' put the text string on the user layer
186      Call pcbDocObj.PutUserLayerText(textStr, _
187          xReal + TITLETEXTOFFESTX, curYReal, usrLyrObj, _
188          TEXTHEIGHT, 0, 5, "VeriBest Gerber 0",, _
189          epcbJustifyHCenter, epcbJustifyBottom, _
190          Nothing, unitsEnum)
```

Line 193 modifies the Y-axis value.

```
192      ' step the Y position
193      curYReal = curYReal + STEPTITLEY
```

Line 196 creates a string to describe a title for all of the information presented to the end-user.

```
195      ' now we need to add the header text
196      textStr = "Layer Stackup"
```

Lines 198 through 203 take the string just created and put it on the user layer.

```
198      ' put the text string on the user layer
199      Call pcbDocObj.PutUserLayerText(textStr, _
200          xReal + TITLETEXTOFFESTX, curYReal, usrLyrObj,_
201          TEXTHEIGHT, 0, 5, "VeriBest Gerber 0", _
202          epcbTextAttrIsUnderlined, epcbJustifyHCenter,
               epcbJustifyBottom, Nothing, unitsEnum)
203  End Function
```

GenerateStackupGraphics() Function

Lines 205 through 258 declare the function that constructs the stackup graphics. This function creates both the motion graphics and the actual graphics to place on the user layer.

The first parameter to this function defines the user layer on which to place the graphics. If this parameter has a value of *Nothing*, then the script creates motion graphics. The *xReal* and *yReal* parameters define the location of the bottom left-hand corner of the graphics. The last parameter holds the total thickness of the board (header information uses this value).

On Line 214, the script declares a local variable to hold a copy of the Y-value (the script modifies this copy).

```
205  ' Helper function for generating stackup graphics.
206  ' If useLayerObj is Nothing motion graphics are
207  ' generated. If it is a user layer graphics are generated
208  ' on the user layer.
209  ' usrLyrObj - UserLayer object
210  ' xReal - Real
211  ' yReal - Real
212  ' totalThicknessReal - Real
213  Function GenerateStackupGraphics(usrLyrObj, xReal, yReal,
                                                totalThicknessReal)
214      Dim curYReal: curYReal = yReal
```

Line 216 declares one variable to hold a *LayerObject* object and another to hold a *LayerObjects* collection. Line 218 returns the *LayerStack*, which is a collection of LayerObject objects (the *True* parameter specifies to include insulating/dielectric layers).

```
216      Dim lyrObjColl, lyrObjObj
217      ' get the layer objects collection
218      Set lyrObjColl = pcbDocObj.LayerStack(True)
```

Line 222 declares an integer variable. Line 223 uses this variable to iterate through the collection of layers in reverse order. The reason for iterating in reverse order is that the script is building the stackup graphics from "bottom" to "top". (Ideally, you use a "*For Each … In … Next*" loop rather than a "*For … To … Next*" loop, but you cannot use the former to iterate through a collection in reverse order.)

```
220      ' since the script is starting at the bottom and working the way
221      ' up, loop through the collection backward in a for loop
222      Dim i
223      For i = lyrObjColl.Count to 1 Step -1
```

Line 226 obtains the next layer object from the collection.

```
225          ' get the layer object from the collection
226          Set lyrObjObj = lyrObjColl.Item(i)
```

Lines 229 and 230 update the variable used to keep track of the total thickness of the board.

```
228          ' add the layer thickness to the total
229          totalThicknessReal = totalThicknessReal + _
230              lyrObjObj.LayerProperties.Thickness
```

At this point, note that a board may comprise a variety of different layers: *dielectric*, *conductor* (there are two types – *signal* and *plane*), *user*, and *fabrication* (there are many types).

On Line 233, the script checks if this is a dielectric (insulating) layer. If so, Line 235 calls a function to draw graphic and text elements to represent the layer, then Line 238 modifies the local Y-axis value to reflect the addition of these elements in the stackup. Observe that, thus far, the script has not determined whether to create the graphics as actual graphics or as motion graphics; it determines this at a lower level.

```
232          ' if a dielectric layer, then draw its graphic and text
233          If lyrObjObj.Type = epcbLayerTypeInsulation Then
234              ' Draw the layer
235              Call DrawDielectricLayer(lyrObjObj, usrLyrObj,
                                          xReal, curYReal, unitsEnum)
236
237              ' step the Y position
238              curYReal = curYReal + STEPCONDY
```

If the layer was not a dielectric layer, then Line 241 checks if it is a conducting layer and – if it is – then Line 242 checks if it is a signal layer. If this is a signal layer, Line 244 calls a function to draw graphic and text elements to represent the layer. Alternatively, if the latter is a plane layer (as determined by the test on Line 246) then Line 248 calls a function to draw graphic and text elements to represent *this* layer. Line 253 modifies the local Y-axis value to reflect the addition of these elements in the stackup.

```
240         ' if a conductive layer then draw is graphic and text
241         ElseIf lyrObjObj.Type = epcbLayerTypeConductor Then
242             If lyrObjObj.ConductorType =
                                    epcbConductorTypeSignal Then
243                 ' Draw the signal layer.
244                 Call DrawSignalLayer(lyrObjObj, usrLyrObj,
                                        xReal, curYReal, unitsEnum)
245
246             ElseIf lyrObjObj.ConductorType =
                                     epcbConductorTypePlane Then
247                 ' Draw the plane layer.
248                 Call DrawPlaneLayer(lyrObjObj, usrLyrObj,
                                        xReal, curYReal, unitsEnum)
249
250             End If
251
252             ' step the Y position
253             curYReal = curYReal + STEPDIEY
254         End If
255     Next
```

In reality, the test on Line 246 is redundant and is included in this script only for clarity. Finally, on Line 257, the script returns the final Y-axis value reflecting the contributions of all of the layers in the stackup.

```
257     GenerateStackupGraphics = curYReal
258 End Function
```

DrawDielectricLayer() Function

Lines 260 through 273 declare the function that controls the drawing of the graphic and text elements associated with a dielectric layer. The first parameter defines the layer in the stackup (use this to obtain a description of the layer). The second parameter defines the user layer on which these graphics are to be presented (if this parameter has a value of *Nothing*, the result creates motion graphics). The third and fourth parameters define the location of the bottom left-hand corner for these elements. The fifth parameter defines the units to use.

On Line 269, the script calls the DrawRectangle() Function to draw a rectangle associated with this layer. Observe the constants used to define the rectangle's width and height (the script declared these constants at the beginning of the script). Also observe the *False* parameter, which informs the helper function not to fill this rectangle.

Line 271 calls the DrawText() Function to draw the text associated with this layer. Observe that the script uses the GetLayerDescription() Function to retrieve the layer description from the *LayerObject* object, and then passes this description as a parameter to the DrawText() Function.

```
260 ' Function for drawing dielectric layers.  Works for user layer
261 ' grpahics and motion graphics.
262 ' layerObjObj - LayerObject object
263 ' userLayerObj - UserLayer object
264 ' originxReal - Real
265 ' originYReal - Real
266 ' unitsEnum - Enumerate (EPcbUnit)
267 Function DrawDielectricLayer(layerObjObj, userLayerObj, _
```

```
268                             originXReal, originYReal, unitsEnum)
269   Call DrawRectangle(userLayerObj, originXReal, originYReal, _
270       DIELECTRICBOXX, DIELECTRICBOXY, False, unitsEnum)
271   Call DrawText(userLayerObj, GetLayerDescription(layerObjObj),_
272                         originXReal + TEXTOFFSETX,
                          originYReal + TEXTOFFSETYDIE, unitsEnum)
273  End Function
```

DrawSignalLayer() Function

Lines 275 through 290 declare the function that controls the drawing of the graphic and text elements associated with a signal layer. This is similar to the DrawDielectricLayer() Function. The main differences are (a) it draws two small rectangles side-by-side (refer to Figure 149: "Real" Graphics Placed on a User Layer for an example) and (b) these rectangles have a fill pattern as defined by the *True* parameters on Lines 285 and 287.

```
275  ' Function for drawing signal layers.  Works for user layer
276  ' grpahics and motion graphics.
277  ' layerObjObj - LayerObject object
278  ' userLayerObj - UserLayer object
279  ' originxReal - Real
280  ' originYReal - Real
281  ' unitsEnum - Enumerate (EPcbUnit)
282  Function DrawSignalLayer(layerObjObj, userLayerObj, _
283                           originXReal, originYReal, unitsEnum)
284    Call DrawRectangle(userLayerObj, originXReal + _
285            TRACEBOX1OFFSETX, originYReal, TRACEBOXX,
                                      TRACEBOXY, True, unitsEnum)
286    Call DrawRectangle(userLayerObj, originXReal + _
287            TRACEBOX2OFFSETX, originYReal, TRACEBOXX,
                                      TRACEBOXY, True, unitsEnum)
288    Call DrawText(userLayerObj, _
289       GetLayerDescription(layerObjObj), originXReal +
          TEXTOFFSETX, originYReal + TEXTOFFSETYCOND, unitsEnum)
290  End Function
```

DrawPlaneLayer() Function

Lines 292 through 305 declare the function that controls the drawing of the graphic and text elements associated with a plane layer. This is similar to the DrawDielectricLayer() Function. The main difference is that the rectangle has a fill pattern as defined by the *True* parameter on Lines 302. Observe that the script has not determined, thus far, whether to create the graphics as actual graphics or as motion graphics. This is determined at a lower level.

```
292  ' Function for drawing plane layers.  Works for user layer
293  ' graphics and motion graphics.
294  ' layerObjObj - LayerObject object
295  ' userLayerObj - UserLayer object
296  ' originxReal - Real
297  ' originYReal - Real
298  ' unitsEnum - Enumerate (EPcbUnit)
299  Function DrawPlaneLayer(layerObjObj, userLayerObj, _
300                          originXReal, originYReal, unitsEnum)
```

```
301    Call DrawRectangle(userLayerObj, originXReal, originYReal, _
302        PLANEBOXX, PLANEBOXY, True, unitsEnum)
303    Call DrawText(userLayerObj, _
304         GetLayerDescription(layerObjObj), originXReal +
              TEXTOFFSETX, originYReal + TEXTOFFSETYCOND, unitsEnum)
305  End Function
```

Helper Functions and Subroutines

This script section declares several generic "helper" routines. These routines can be used in other scripts and are not focused on this script exclusively.

ConfigureUserLayer() Function

Lines 311 through 345 declare a function that either returns an existing user layer or – if the layer does not exist – the function creates the layer and returns it. Also, the function makes sure that this user layer displays.

Observe the parameter passed into the function on Line 314. This is a string that defines the name of the user layer (this string is defined by the constant the script declared on Line 20 and which was subsequently passed as a parameter on Line 132).

```
311  ' Returns user layer object by this name and turns on display
312  ' of user layer.  If user layer does not exist it is created.
313  ' userLayerNameStr - String
314  Function ConfigureUserLayer(userLayerNameStr)
```

Line 316 searches for a user layer by this name.

```
315      ' Check if the user layer already exists.
316      Dim usrLyrObj
317      Set usrLyrObj = pcbDocObj.FindUserLayer(userLayerNameStr)
```

If no user layer exists (as determined by the test on Line 319), then the script creates it on Line 321. Alternatively, if this user layer does exist, then Lines 324 and 325 remove any graphics and text that may already be present on this layer.

```
319      If usrLyrObj Is Nothing Then
320          ' It doesn't exist. Create it.
321           Set usrLyrObj =
           pcbDocObj.SetupParameter.PutUserLayer(userLayerNameStr)
322      Else
323          ' It does exist. Remove any gfx and text on the layer
324          pcbDocObj.UserLayerGfxs(, userLayerNameStr).Delete
325          pcbDocObj.UserLayerTexts(, userLayerNameStr).Delete
326      End If
```

Line 329, ensures that the user layer is on in *DisplayControl*.

```
328      ' ensure the user layer is turned on
329      pcbDocObj.ActiveViewEx.DisplayControl.
                                UserLayer(userLayerNameStr) = True
```

Line 333 accesses the global display control object. Line 334 accesses the color pattern object for this user layer.

```
331      ' get the color pattern object for this user layer
332      Dim colorPatternObj, globalDisplayControl
333      Set globalDisplayControl =
                        pcbDocObj.ActiveViewEx.DisplayControl.Global
334      Set colorPatternObj =
               globalDisplayControl.UserLayerColor(userLayerNameStr)
```

Line 337 sets the color pattern object index to 13, which is the diagonal stripe fill for the signal and plane layers (refer to Figure 149: "Real" Graphics Placed on a User Layer for an example). Line 341 assigns the new color pattern object to the user layer.

```
336      ' set it to the pattern
337      colorPatternObj.Index = 13
338
339      ' now we have to set the UserLayerColor object to the
340      ' color pattern object we just modified
341      Set globalDisplayControl.UserLayerColor(userLayerNameStr) =
                                                     colorPatternObj
```

Finally, the script returns the modified *UserLayer* object and exits the function.

```
343      ' Return the user layer object.
344      Set ConfigureUserLayer = usrLyrObj
345  End Function
```

GetLayerDescription() Function

Lines 347 through 352 declare a function that accesses the layer's description. The input parameter to this function is a *LayerObject* object. On Lines 350 and 351, the script builds and returns a string that contains both the thickness of the layer and its description.

```
347  ' Formats a string for the layer description
348  ' layerObjObj - LayerObject object
349  Function GetLayerDescription(layerObjObj)
350      GetLayerDescription =
             layerObjObj.LayerProperties.Thickness & "th, " & _
351          layerObjObj.LayerProperties.Description
352  End Function
```

DrawText() Function

Lines 354 through 375 declare a function that creates text either as motion graphics or as actual graphics on a specified user layer. The first parameter to the function specifies the user layer; if this parameter has a value of *Nothing*, the function displays the text as motion graphics. The second parameter is the text string to display. The third and fourth parameters specify the X/Y coordinates of the lower left corner of the text. The fifth parameter specifies the units to use.

```
354  ' Creates text on either a user layer or in motion graphics.
355  ' userLayerObj - UserLayer object
356  ' originxReal - Real
357  ' originYReal - Real
358  ' unitsEnum - Enumerate (EPcbUnit)
359  Function DrawText(userLayerObj, textStr, originXReal,
                                          originYReal, unitsEnum)
```

On Line 361, the script tests if the user layer has a value of *Nothing*. If this is the case, then Lines 363 through 367 use the *MotionGfxPutText* method to generate this text as motion graphics. By deferring the motion graphics check to this function, the higher level functions are simplified allowing them to have the same code for creating both motion and user layer graphics.

```
360      ' If layer passed in is Nothing, then draw motion graphics
361     If userLayerObj Is Nothing Then
362       Call pcbDocObj.ActiveViewEx.MotionGfxPutText(textStr, _
363            originXReal, originYReal, epcbMotionGfxPointMoveXY, _
364            TEXTHEIGHT, 0, 5, "VeriBest Gerber 0", _
365            epcbTextAttrIsBold, epcbJustifyLeft, _
366            epcbJustifyVCenter, tagMotionInt, Nothing, _
367            True, unitsEnum, epcbAngleUnitDegrees)
```

For the *MotionGfxPutText* method, the first parameter is the text string to display. The second and third parameters specify the X/Y coordinates of the lower left corner of the text. The fourth parameter is an enumerate that defines how the motion graphics moves (a value of *epcbMotionGfxPointMoveXY* causes the graphics to follow the cursor in both the X and Y planes; by comparison, values of *epcbMotionGfxPointMoveX* or *epcbMotionGfxPointMoveY* cause the graphics to follow the cursor only in the X or Y planes, respectively).

The fifth, sixth, seventh, and eighth parameters define the height, rotation, pen width, and font of the text, respectively. The ninth parameter specifies whether the text is to be bold, italic, underlined, or a combination of any of these attributes. The tenth parameter specifies the horizontal justification (left, center, or right) while the eleventh parameter defines the vertical justification.

The twelfth parameter is the "tag" identifier the script declared on Line 33 and assigned on Line 111 (use the identifier to delete this motion graphic later). The thirteenth parameter defines the color object of the text (pass a value of *Nothing*, which defaults to white). The fourteenth parameter is a Boolean that can be set to *True* or *False* to specify whether to display a boundary box. The fifteenth parameter specifies the linear units, while the sixteenth parameter defines the angular units (degrees versus radians) – this latter value is of interest only if the text is being rotated, which it is not in this example.

If the text is not motion graphics, but instead actual graphics, Lines 368 through 375 place the text on the specified user layer. The parameter descriptions for the *PutUserLayerText* method are similar to those of the *MotionGfxPutText* method.

```
368     Else
369       Call pcbDocObj.PutUserLayerText(textStr, _
370            originXReal, originYReal, userLayerObj, _
371            TEXTHEIGHT, 0, 5, "VeriBest Gerber 0",, _
372            epcbJustifyLeft, epcbJustifyVCenter, _
373            Nothing, unitsEnum)
374     End If
375  End Function
```

DrawRectangle() Function

Lines 377 through 398 declare a function that creates a rectangle either as motion graphics or as actual graphics on a specified user layer. The first parameter to the function specifies the user layer; if this parameter has a value of *Nothing*, the function displays the rectangle as motion graphics. The second and third parameters specify the X/Y coordinates of the lower left corner of the rectangle. The fourth and fifth parameters specify the width and height of the rectangle, respectively. The sixth parameter is a Boolean that specifies if the rectangle is to be filled or not (this is only used if the rectangle is drawn on a user layer – rectangles in motion graphics do not support fills). The seventh parameter specifies the units to use.

```
377  ' Creates a rectangle on a user layer or in motion graphics.
378  ' userLayerObj - UserLayer object
379  ' originxReal - Real
380  ' originYReal - Real
381  ' unitsEnum - Enumerate (EPcbUnit)
382  Function DrawRectangle(userLayerObj, originXReal, originYReal,
                                  width, height, bFill, unitsEnum)
```

On Line 384, the script tests if the user layer has a value of *Nothing*. If this is the case, then on Lines 385 through 387, the script uses the *MotionGfxPutRect* method to generate this rectangle as a motion graphic. By deferring the motion graphics check to this function, the higher level functions are simplified allowing them to have the same code for creating both motion and user layer graphics.

```
383      ' If user layer passed in is Nothing, then motion graphics
384      If userLayerObj Is Nothing Then
385          Call pcbDocObj.ActiveViewEx.MotionGfxPutRect(originXReal,
                 originYReal, epcbMotionGfxPointMoveXY, _
386              originXReal + width, originYReal + height,
                 epcbMotionGfxPointMoveXY, _
387              tagMotionInt, 1, Nothing, unitsEnum)
```

For motion graphics, the first and second parameters specify the X/Y coordinates of the lower left-hand corner of the rectangle. The third parameter is an enumerate that defines how the motion graphics associated with the lower left-hand corner moves (a value of *epcbMotionGfxPointMoveXY* causes it to follow the cursor in both the X and Y planes; by comparison, values of *epcbMotionGfxPointMoveX* or *epcbMotionGfxPointMoveY* causes it to follow the cursor only in the X or Y planes, respectively).

The fourth and fifth parameters define the X/Y coordinates of the upper right-hand corner of the rectangle relative to its lower left-hand corner. The sixth parameter is an enumerate that defines how the motion graphics associated with the upper right-hand corner moves.

The seventh parameter is the "tag" identifier the script declared on Line 33 and assigned on Line 111 (this is the identifier used to delete this motion graphic later). The eighth parameter defines the width of the line. The ninth parameter defines the color of the line (pass a value of *Nothing*, which defaults to white). The tenth parameter specifies the linear units.

For actual graphics, Lines 388 through 398 place the text on the specified user layer. Lines 391 and 392 build a points array, and then the *PutUserLayerGfx* method on Lines 395 and 396 draws the rectangle on the user layer. (For more details on this, refer to “Exercise 13: Adding User Layer Graphics” on page 243.)

```
388      Else
389         Dim pntsArr
390         ' get the rectangular points array for this layer
391         pntsArr = pcbAppObj.Utility.CreateRectXYR(originXReal, _
```

```
392         originYReal, originXReal + width, originYReal + height)
393
394         ' put the rectangle on the user layer
395         Call pcbDocObj.PutUserLayerGfx(userLayerObj, 0, _
396              UBound(pntsArr, 2) + 1, pntsArr, bFill, Nothing,
                                                        unitsEnum)
397      End If
398  End Function
```

Miscellaneous Functions and Subroutines

This script section declares any miscellaneous routines (they cannot easily be classified as “Utility” or “Helper” routines).

ClearMotionGraphics() Function

Lines 401 through 408 declare a function that removes any motion graphics. The *MotionGfxDeleteByTag* method uses the “tag” identifier the script declared on Line 33 and assigned on Line 111 (observe that Line 407 resets this tag after it deletes the motion graphics.

```
401  ' Miscelaneous Functions
402
403  ' Deletes all motion graphics.
404  Function ClearMotionGraphics()
405      ' Remove motion graphics and reset the tag.
406      pcbDocObj.ActiveViewEx.MotionGfxDeleteByTag(tagMotionInt)
407      tagMotionInt = -1
408  End Function
```

Creating and Testing the Script

1. Use the scripting editor of your choice to enter the script described (including the license server function) and save it as *LayerStackup.vbs*.

2. Close any instantiations of Xpedition Layout that are currently running, and then launch a new instantiation of this application.

3. Open the *Candy.pcb* design.

4. Run the *LayerStackup.vbs* script and observe the motion graphics representing the layer stackup appear on the screen.

5. Move the mouse cursor and observe that the motion graphics follow it around the screen.

6. Locate the motion graphics in a clear area of the design and then click the left mouse button. Observe that the script replaces motion graphics with "real" graphics (including a header) on the user layer.

Enhancing the Script

There are a number of ways to enhance this script. Several suggestions follow (practice your scripting skills by implementing these suggestions):

- Set this script to run as a startup script (refer to “Running Startup Scripts” on page 171).
- Vary the constant definitions to try to achieve more aesthetically pleasing graphics and text representations.
- Vary the font and line colors and fill patterns to try to achieve more aesthetically pleasing graphics and text representations.

Exercise 3: Performing Assembly DRC

Scope: Xpedition Enterprise

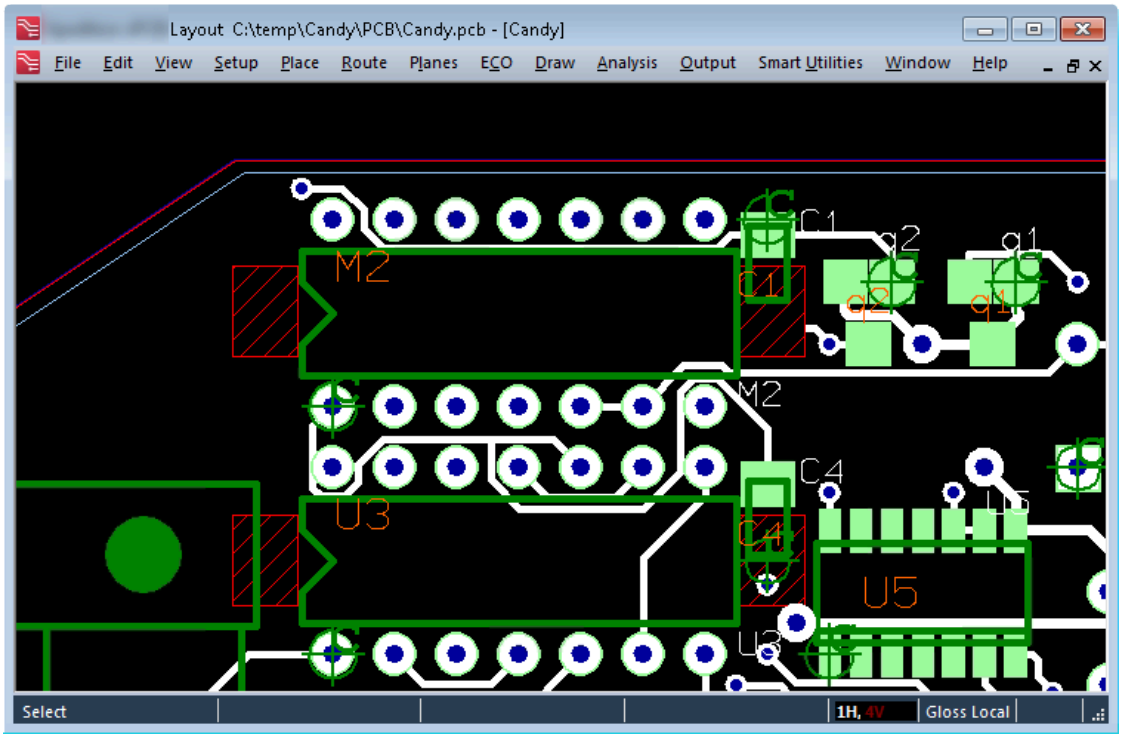
This script performs online assembly DRC checking whenever a component is placed to ensure that there is not a violation.

Approach:	To use component placement events to initiate the checks; also to use "gripper graphics" to determine gripper-to-component violations.
Points of Interest:	– Using name-value property pairs
	– Finding components in a given area
	– Checking shape-to-shape overlaps
	– Using the Output window
Items Used:	– Mask Engine Server
	– PickComponents Method
	– OnPreComponentPlace Event
	– Output Window Addin

Before Starting

The main script described in this section performs online DRC checking whenever a component is placed to ensure that there is not a violation. In order to facilitate this, a "Gripper Graphics" user layer is already exists as part of the *Candy.pcb* design, so the first thing to do is turn on the display of this layer. Observe, for example, the red gripper graphics shown on either side of components **M2** and **U3** as illustrated in Figure 150: Candy.pcb Displaying "Gripper Graphics".

Figure 150. Candy.pcb Displaying "Gripper Graphics"



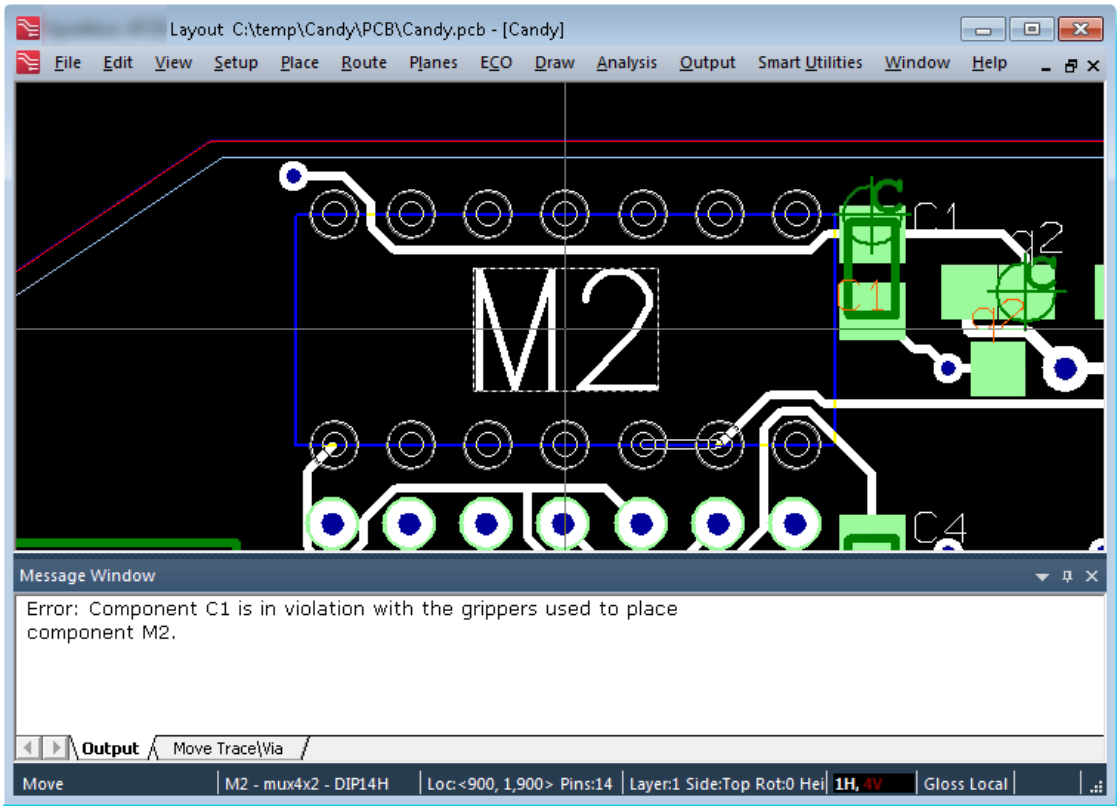
In the real world, you use some mechanism to establish the component placement order; the actual mechanism depends on each company's internal process. For example, you might define the placement order in a side-file and then use a special script to access the contents of that side-file to assign the placement order of components on the board.

For simplicity, two scripts are created. *AssignPlacementOrder.vbs* assigns a numeric (integer) placement order based on an alphanumeric sort of component names. *AssemblyDRC.vbs* performs the DRC.

To understand how these scripts work, open the *Candy.pcb* layout design, and enable the display of the "Gripper Graphics" user layer as discussed.

Run the *AssignPlacementOrder.vbs* script to assign the placement order and then run the *AssemblyDRC.vbs* script. Now, select component **M2** and attempt to place it in its current/original position. Observe that the **Output** window appears displaying a message that informs about a DRC violation as illustrated in Figure 151: Script Detects a DRC Violation.

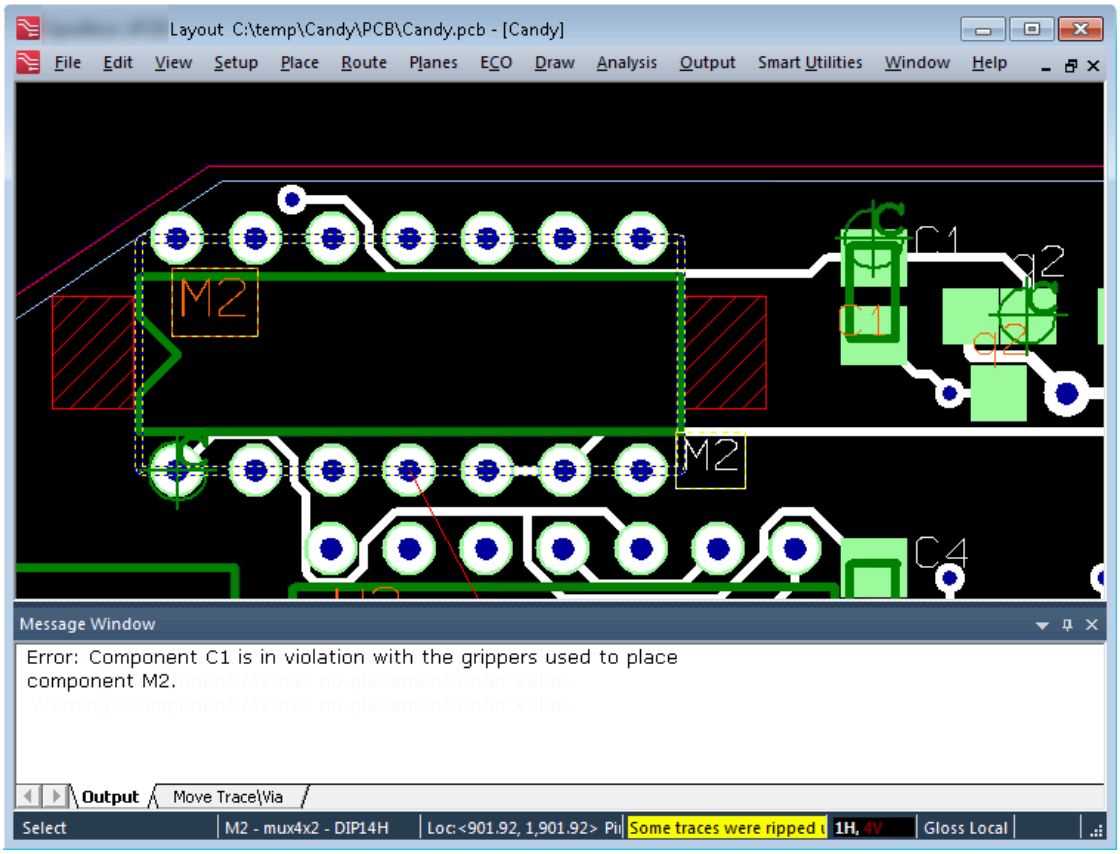
Figure 151. Script Detects a DRC Violation



When the script detects a DRC violation, it will not place the selected component, but will instead keep this component as motion graphics attached to the cursor (refer to the discussions in Exercise 2: Documenting the Layer Stackup for more details on the concept of motion graphics).

Next, drag component **M2** away from capacitor **C1** and attempt to place it in a clear area. This time the placement is successful as illustrated in Figure 152: Component is Successfully Placed. Observe that the original DRC error message remains on display in the **Output** window. The reason for this is that the script did not clear this message from the window (refer to the Enhancing the Script discussions).

Figure 152. Component is Successfully Placed



The Assign Placement Order Script

This is the simple script used to assign a placement order to the components on the board (you must run this script before the main script). The key sections of this script follow:

Constant Definitions	Define the constants the script uses.
Initialization/Setup	Returns the *Application* and *Document* objects, licenses the document, and adds the type libraries. Also performs any additional setup associated with this script.
AssignPlacementOrder () Function	Assigns a numeric (integer) placement order based on an alphanumeric sort of component names.
Validate Server	Licenses the document and validates the server. Refer to "Running Your First Script" on page 59 for more details on this function.

Constant Definitions

The only constant declaration this script uses occurs on Line 7.

```
 1  ' Create a placement order for all components and assign to each
 2  ' each component. In an actual process this script would read
 3  ' the ordering from another location.       (Author: Toby Rimes)
 4  Option Explicit
 5
 6  ' Constants
 7  Const PLACEMENT_ORDER = "PLACEMENT_ORDER"
```

Initialization/Setup

On Line 10, the script adds the type library this script requires. Lines 13 and 17 return the *Application* object; Lines 14 and 20 return the *Document* object. Line 23 licenses the document by calling the standard *ValidateServer()* function; Line 26 calls the AssignPlacementOrder () Function.

```
 9  ' add any type libraries to be used.
10  Scripting.AddTypeLibrary("MGCPCB.ExpeditionPCBApplication")
11
12  ' Global Variables
13  Dim pcbAppObj
14  Dim pcbDocObj
15
16  ' Get the application object
17  Set pcbAppObj = Application
18
19  ' Get the active document
20  Set pcbDocObj = pcbAppObj.ActiveDocument
21
22  ' License the document
23  ValidateServer(pcbDocObj)
24
25  ' Call main function
26  Call AssignPlacementOrder()
```

AssignPlacementOrder () Function

On Lines 32 through 47, the script declares the *AssignPlacementOrder()* function. Lines 34 and 35 obtain a collection of all of the components on the board. Line 38 uses the *Sort* method to perform an alphanumerical sort on the collection. Lines 43 to 45 assigns a placement order to each component.

```
28  '''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
29  ' Main Functions
30
31  ' Assings a numbering to the components.
32  Function AssignPlacementOrder()
33      ' Get the collection of components
34      Dim cmpColl
35      Set cmpColl = pcbDocObj.Components
36
37      ' Sort them so that it is a known order
38      Call cmpColl.Sort()
39
40      ' Number the components.
```

```
41      Dim cmpObj
42      Dim i : i = 1
43      For Each cmpObj In cmpColl
44          Call cmpObj.PutProperty(PLACEMENT_ORDER, i & "")
45          i = i + 1
46      Next
47  End Function
```

Observe the *PutProperty* method on the *Component* object in Line 44. This allows assignment of a "Property Pair" to the component, where a property pair comprises a name and a value. In this example, the name is "PLACEMENT_ORDER" as assigned to the PLACEMENT_ORDER constant on Line 7; the value is a string generated by concatenating the iterating integer with a null string (this concatenation is performed so as to convert the integer to its string equivalent).

The Assembly DRC Script

The key sections of this script follow:

Constant Definitions	Defines the constants the script uses.
Initialization/Setup	Returns the *Application* and *Document* objects, licenses the document, and adds the type libraries. Also performs any additional setup associated with this script.
Event Handler	Defines a pre-component-place event handler that triggers just before a component is actually placed.
IsAssemblyViolation() Function	Defines a function that checks if there is an assembly violation.
Utility Functions and Subroutines	Defines special "workhorse" routines that are focused on this script.
	– IsCmp2CmpAssemblyViolation() Function
	– GetNearbyComponents() Function
	– GetPlacementNumber() Function
Helper Functions and Subroutines	Defines generic "helper" routines that other scripts can use.
	– Output() Function
	– LoadAddin() Function
	– FindAddin() Function
	– RemoveTrailingNewLine() Function
Validate Server	Licenses the document and validates the server. Refer to “Running Your First Script” on page 59 for more details on this function.

A high-level (pseudo-code) representation of the script follows. The script checks each component that is "near" to the component being placed. How the script checks the two components depends on which was placed first.

```
OnPreComponentPlace()
    IsAssemblyViolation()
        GetPlacementOrder(cmp)
        GetNearbyComponents()
        For Each Near Component
            GetPlacementNumber(nearCmp)
            If This one first Then
                IsCmp2CmpAssemblyViolation(nearCmp, cmp)
            Else
                IsCmp2CmpAssemblyViolation(cmp, nearCmp)
            End If
        End For
    Output violation
```

Constant Definitions

This section of the script declares any constants to use.

Line 10 declares the name of the user layer containing the gripper graphics (the script is hard-coding this name/layer).

Line 12 declares the name of the property for placement order. Recall that a property can be attached to a component. The *AssignPlacementOrder()* script assigns a property to every component in the design. The name of this property is "PLACEMENT_ORDER", and the value associated with each property is another string representing the component's placement order.

Line 14 defines a hard-coded distance value. The script uses this value to locate components in close proximity to the component being placed.

```
 1  ' This script implements a simple assembly check to ensure
 2  ' grippers don't make contact with other components during
 3  ' assembly. It utilizes the OnPreComponentPlace event to get
 4  ' notification of the placement and trigger the analysis.
 5  ' Violations written to the output window. (Author: Toby Rimes)
 6  Option Explicit
 7
 8  ' Constants
 9  ' User layer representing gripper area
10  Const GRIPPERGFX_LAYER = "gripper_outline"
11  ' Name of component property pair holding the placement order
12  Const PLACEMENT_ORDER = "PLACEMENT_ORDER"
13  ' Pick distance for finding nearby components (th)
14  Const PICK_OFFSET = 50
15
16  ' Add-in enumerates for add-ins. These cannot
17  ' be added via AddTypeLibrary so are redefined here.
```

```
18  Const PlacementLocationLeft   = 0
19  Const PlacementLocationTop    = 1
20  Const PlacementLocationRight  = 2
21  Const PlacementLocationBottom = 3
```

The script uses the **Output** window addin to output messages. The **Output** window type library cannot be included using the *AddTypeLibrary* method, so Lines 18 through 21 declare the enumerates the script requires as constants.

The **Output** window can dock at the Top, Bottom, Left, or Right of the main window. The script docks it at the bottom, so only requires the constant declaration on Line 21. The other declarations are here for completeness.

Initialization/Setup

Lines 24 and 25 add the type libraries this script requires. Lines 28 and 29 declare variables in which to hold the *Document* and *Application* objects, respectively. Line 30 defines the units to use throughout the script (these units apply to the pick offset constant on Line 14). Line 33 returns the *Application* object; Line 36 returns the *Document* object; Line 24 licenses the document by calling the standard *ValidateServer()* function; and Line 42 assures that the document units match the script units by setting the document units to the script units.

```
23  ' add any type librarys to be used.
24  Scripting.AddTypeLibrary("MGCPCB.ExpeditionPCBApplication")
25  Scripting.AddTypeLibrary("MGCPCBEngines.MaskEngine")
26
27  ' Global Variables
28  Dim pcbDocObj
29  Dim pcbAppObj
30  Dim unitsEnum : unitsEnum = epcbUnitMils ' Units to be used
31
32  ' Get the application object
33  Set pcbAppObj = Application
34
35  ' Get the active document
36  Set pcbDocObj = pcbAppObj.ActiveDocument
37
38  ' License the document
39  ValidateServer(pcbDocObj)
40
41  ' Set the unitsEnum to match the hard coded values.
42  pcbDocObj.CurrentUnit = unitsEnum
43
44  ' Attach the events to the Document
45  Call Scripting.AttachEvents(pcbDocObj, "pcbDocObj")
46
47  ' Hang around to listen to events
48  Scripting.DontExit = True
```

Line 45 attaches events to the *Document* object. Line 48 sets the *DontExit* property of the *Scripting* object to *True*. This means the script continues running until the end of the current application session, thereby allowing the script to detect and handle events.

Event Handler

On Lines 53 through 77, the script declares the pre-component-placement event handler. This handler, which is part of the Xpedition Layout automation interface, fires (triggers) when the end-user attempts to place a component.

The first parameter to this function is the component object being placed. The second and third parameters define the X/Y coordinates of the component's origin. The automation interface automatically populates the values assigned to these parameters when this event triggers.

```
53  ' Pre component placement handler (defined in MGCPCB interface)
54  ' Fired before placement of a component.
55  ' cmpObj - Component object
56  ' originXReal - Real
57  ' originYReal - Real
58  ' layerInt - Integer
59  ' rotationDouble - Real
60  ' mirrorBool - Mirror
61  Function pcbDocObj_OnPreComponentPlace(cmpObj, originXReal, _
62                 originYReal, layerInt, rotationReal, mirrorBool)
```

On Line 64, the script declares a variable to hold a string describing any violation. This variable passes as an In-Out parameter to the IsAssemblyViolation() Function, which Line 65 calls (*all* parameters in VBScript are In-Out parameters).

If there is an assembly violation, then Line 67 sets the return value from this function to *False*; this prevents the component from being placed and keeps it as motion graphics. If there is no assembly violation, then Line 70 sets the return value from this function to *True*.

```
63      ' Check for an assembly violation
64      Dim violationDescStr
65      If IsAssemblyViolation(cmpObj, violationDescStr) Then
66          ' Violation exists. Don't allow placement
67          pcbDocObj_OnPreComponentPlace = False
68      Else
69          ' No violation. Allow placement
70          pcbDocObj_OnPreComponentPlace = True
71      End If
```

On Line 74, the script tests whether the violation string is empty. If the string contains something, then there is a violation and the script passes the violation string to the **Output** window. Note that there may be a violation string *without* actually having an assembly violation. For example, the violation string may contain errors that a component does not have a valid placement order number.

```
73      ' Display violation description in the output window.
74      If Not violationDescStr = "" Then
75          Output(violationDescStr)
76      End If
77  End Function
```

IsAssemblyViolation() Function

Lines 88 through 144 declare the main function that determines if there is an assembly violation. The first parameter to this function is the name of the component being placed; the second is a string that contains descriptions of any violations this function detects.

```
 82  ' Returns true if the component is in any assembly violation
 83  ' Returns false if there are no assembly violations.
 84  '
 85  ' descStr will contain a description of the violations.
 86  ' cmpObj - Component object
 87  ' descStr - String (in/out parameter)
 88  Function IsAssemblyViolation(cmpObj, descStr)
```

On Line 91, the script declares a local variable to hold a string containing all violations this function detects and initializes it to be an empty string.

```
 90      ' String to hold the violation description
 91      Dim violationDescStr : violationDescStr = ""
```

Line 94 sets this function's return value to *False*.

```
 93      ' Initialize the return value
 94      IsAssemblyViolation = False
```

Line 97 declares and assigns a variable that holds the placement order number associated with the component being placed. It calls the GetPlacementNumber() Function to determine the placement order number.

```
 96      ' Get the placement number of cmpObj
 97      Dim cmpPlaceInt : cmpPlaceInt = GetPlacementNumber(cmpObj)
```

Line 100 checks to ensure that the placement order number is valid. If not (which can occur if this component is not assigned a number), then Lines 101 and 102 add an appropriate warning to the local violation message string. The script ends this warning with a vbCrLf (carriage return and line feed).

```
 99      ' Ensure the placement number is valid
100      If cmpPlaceInt = -1 Then
101          violationDescStr = violationDescStr & "Warning:
             Component " & cmpObj.Name & _
102          " has no placement order value." & vbCrLf
```

If the placement order number is valid, then Line 105 declares a component collection and Line 106 uses the GetNearbyComponents() Function to retrieve a collection of nearby components.

```
103      Else
104          ' Get the nerby components
105          Dim nearCmpColl
106          Set nearCmpColl = GetNearbyComponents(PICK_OFFSET, cmpObj)
```

Line 109 declares a variable that holds an individual near component object; Line 110 iterates through the collection of nearby components.

```
108          ' Iterate to find components in violation
109          Dim nearCmpObj
110          For Each nearCmpObj In nearCmpColl
```

Line 112 declares and assigns a variable that holds the placement order number associated with the nearby component. The script calls the GetPlacementNumber() Function to determine this placement order number.

```
111              ' Get the placement ordering of nearCmpObj
112              Dim nearCmpPlaceInt : nearCmpPlaceInt =
                                       GetPlacementNumber(nearCmpObj)
```

Line 115 checks to ensure that the placement order number is valid. If not valid, then Lines 116 and 117 add an appropriate warning to the local violation message string. The script ends this warning with a vbCrLf.

```
114              ' Ensure the placement number is valid
115              If nearCmpPlaceInt = -1 Then
116                 violationDescStr = violationDescStr & "Warning:
                    Component " & nearCmpObj.Name & _
117                 " has no placement order value." & vbCrLf
```

On Line 120, the script checks if the placement order number associated with the nearby component is *lower* than the component the script is currently evaluating. This means that the nearby component is placed *earlier*. If this is the case, then Line 122 calls the IsCmp2CmpAssemblyViolation() Function and passes it the nearby component object as the first parameter and the current component object as the second parameter.

If the placement order number associated with the nearby component is *higher* than the component the script is currently evaluating, the nearby component is placed *later*. In this case, Line 131calls the IsCmp2CmpAssemblyViolation() Function and passes it the current component object as the first parameter and the nearby component object as the second parameter.

In both cases, if the IsCmp2CmpAssemblyViolation() Function returns a value of *True* (indicating a violation), the script adds an appropriate warning to the local violation message string. The script ends these warnings with a vbCrLf (carriage return and line feed). Observe that in both cases the *IsAssemblyViolation()* return value is set to *True*, which informs the calling script that the function detects an assembly violation.

```
119              ' Determine which is placed first
120              ElseIf nearCmpPlaceInt < cmpPlaceInt Then
121                  ' Guessed correctly.  Check for a violation
122                  If IsCmp2CmpAssemblyViolation(nearCmpObj,cmpObj) Then
123                      ' There is a violation
124                      violationDescStr = violationDescStr & _
125                      "Error: Component " & nearCmpObj.Name & _
126                      "is in violation with the grippers " &
                         "used to place component " &
                         cmpObj.Name & vbCrLf
127                      IsAssemblyViolation = True
128                  End If
129              Else
130                  ' Check with the different order.
131                  If IsCmp2CmpAssemblyViolation(cmpObj,
```

```
                        nearCmpObj) Then
132                      ' There is a violation
133                      violationDescStr = violationDescStr & _
134                      "Error: Component " & cmpObj.Name & _
135                      " is in violation with the grippers used" &
                         " to place component " &
                         nearCmpObj.Name & vbCrLf
136                      IsAssemblyViolation = True
137                  End If
138              End If
139          Next
140      End If
```

Finally, the local violation string, *violationDescStr*, passes as a parameter to the RemoveTrailingNewLine() Function. This function removes any trailing carriage return/line feed characters. The script then assigns the function result to the main violation string parameter, *descStr*.

```
141
142      ' Set description string for the calling function to use
143      descStr = RemoveTrailingNewLine(violationDescStr)
144  End Function
```

Utility Functions and Subroutines

This script section declares utility routines. These routines are similar to the generic Helper Functions and Subroutines, except that they are specific to this script and may not be used elsewhere without changes.

IsCmp2CmpAssemblyViolation() Function

Lines 154 through 200 declare the function that checks for component-to-component assembly violations, *IsCmp2CmpAssemplyViolation ()*. It accepts two *Component* object parameters, where the first component must be placed before the second.

```
149  ' Checks for a component to component assembly violation.
150  ' The order matters.  The first parameter must be the component
151  ' placed prior to the component in the second parameter
152  ' placeCmpObj Component object
153  ' unplacedCmpObj Component object
154  Function IsCmp2CmpAssemblyViolation(placedCmpObj, unplacedCmpObj)
```

The Mask Engine is an application that allows you to specify any number of geometric shapes and to then perform operations on the shapes such as AND, OR, XOR, and SUBTRACT. In this example, the script uses the Mask Engine to determine specific gripper-to-component violations.

This function uses Mask Engine as follows:

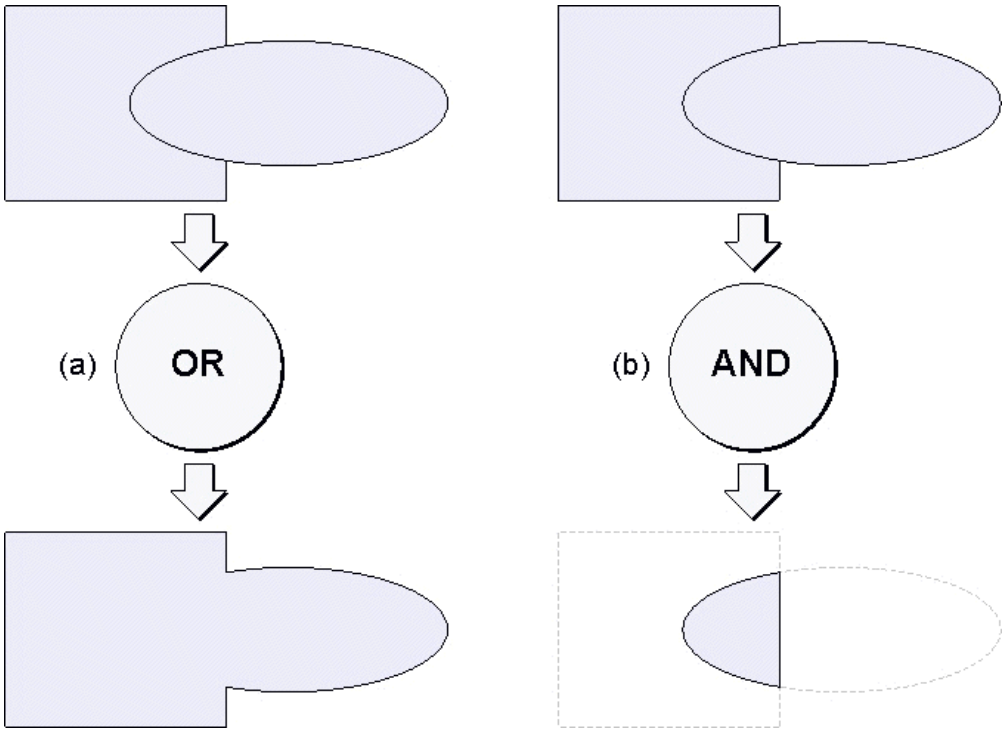
1. The function passes a number of points arrays corresponding to different geometric shapes to the Mask Engine.

2. The function specifies the operation to perform on these shapes.

3. The Mask Engine generates a new points array containing the geometric result of this operation.

As an example, consider what happens if you use the Mask Engine to perform an OR operation on two overlapping shapes as illustrated in Figure 153: Mask Engine Performs OR and AND Operations(a). Similarly, consider the result of performing an AND operation on the same two shapes as illustrated in Figure 153: Mask Engine Performs OR and AND Operations(b).

Figure 153. Mask Engine Performs OR and AND Operations



On Lines 156 and 157, the script creates the Mask Engine object, which is a separate server.

```
155      ' Create the mask engine server
156      Dim maskEngineObj
157      Set maskEngineObj = CreateObject("MGCPCBEngines.MaskEngine")
```

Line 160 sets the units for the Mask Engine.

```
159      ' Set the current unit
160      maskEngineObj.CurrentUnit = unitsEnum
```

Lines 164 and 165 acquire the points array associated with the placed component.

```
162      ' Create mask to represent the placement
163      ' outline of placed component
164      Dim pntsArr
165      pntsArr =
                 placedCmpObj.PlacementOutlines.Item(1).Geometry.PointsArray
```

Lines 168 through 170 create a mask that corresponds to the points array for the placed component. Line 168 instantiates a variable to hold a *Mask* object; Line 169 creates a new/empty *Mask* object; and Line 170 defines the mask by adding a shape (in the form of a points array) to it.

```
167      ' Create mask
168      Dim maskPlacedObj
169      Set maskPlacedObj = maskEngineObj.Masks.Add()
170      Call maskPlacedObj.Shapes.AddByPointsArray(UBound(pntsArr,
                                                     2) + 1, pntsArr)
```

Lines 173 through 179 create a mask to represent the gripper. Lines 173 and 174 create a new/empty *Mask* object.

```
172      ' Create a mask to represent the gripper
173      Dim maskUnplacedObj
174      Set maskUnplacedObj = maskEngineObj.Masks.Add()
```

Lines 177 and 178 acquire a collection of gripper graphics from the unplaced component.

```
176      ' Get the gripper gfx objects
177      Dim gripperGfxColl
178      Set gripperGfxColl = _
179                  unplacedCmpObj.UserLayerGfxs(epcbSelectAll,
                                                  GRIPPERGFX_LAYER)
```

Lines 183 through 186 iterate through the gripper graphics and for each graphic adds a points array to the unplaced component *Mask* object.

```
181      ' Add a shape for each gripper gfx belonging to the
           unplaced component
182      Dim gripperGfxObj
183      For Each gripperGfxObj In gripperGfxColl
184          pntsArr = gripperGfxObj.Geometry.PointsArray
185          Call maskUnplacedObj.Shapes.AddByPointsArray
                                 (UBound(pntsArr, 2) + 1, pntsArr)
186      Next
```

Line 190 “AND”s the two *Mask* objects together by calling the appropriate Boolean operation on the placed *Mask* object and passing the unplaced *Mask* object.

```
188      ' AND the masks together to determine if there
189      ' is any overlap
190      Call maskPlacedObj.BooleanOp(emeBooleanOpAND,maskUnplacedObj)
```

On Line 193, the script checks the result, which is in the placed *Mask* object. If there are 0 (zero) shapes. there are no overlaps, so Line 195 sets the return value of this function to *False* (indicating no violation). If there is one or more shapes, there are overlaps, so Line 198 sets the return value of the function to *True* (indicating a violation).

```
192      ' Check the result, which is in maskPlacedObj.
193      If maskPlacedObj.Shapes.Count = 0 Then
194          ' There are no shapes so there are no overlaps
195          IsCmp2CmpAssemblyViolation = False
196      Else
```

```
197         ' There is one or more shapes so there is overlap
198         IsCmp2CmpAssemblyViolation = True
199     End If
200 End Function
```

GetNearbyComponents() Function

Lines 205 through 228 declare a function that checks if there are any components in close proximity to a specified component. This function accepts two parameters: the offset from the *Component* object's placement outline (this defines the area to search) and the *Component* object itself.

```
202 ' Returns a collection of components within a specified
                    Distance (distanceReal) of a component (cmpObj)
203 ' distanceReal - Real
204 ' cmpObj - Component object
205 Function GetNearbyComponents(distanceReal, cmpObj)
```

Lines 207 and 208 determine the extrema of the component's placement outline. This returns the values: MinX, MinY, MaxX, and MaxY, where MinX/MinY define the lower-left point of the placement outline and MaxX/MaxY define the upper-right point.

```
206     ' Get placement outline extrema.  Assume only 1 outline.
207     Dim extremaObj
208     Set extremaObj = cmpObj.PlacementOutlines.Item(1).Extrema
```

Lines 212 through 215 use the *PickComponents* method to locate and return all components in the specified area. The boundary of this area is defined with the first four parameters to the method. Each of these parameters is extended by specified offset amount (the offset is passed into this function as a parameter). The fifth parameter specifies to include all components; the sixth parameter specifies to include all cell types; the seventh parameter specifies the layer to include (a value of 0 indicates "all layers"). The eighth parameter is set to *True*, indicating to include components that currently display.

```
210     ' Use pick component to get all component in this area
211     Dim cmpColl
212     Set cmpColl = pcbDocObj.PickComponents(extremaObj.MinX –
                                                  distanceReal, _
213         extremaObj.MinY - distanceReal, extremaObj.MaxX +
                                                  distanceReal, _
214         extremaObj.MaxY + distanceReal, epcbCompAll, _
215                                       epcbCelltypeAll, 0, True)
```

In addition to any nearby components, the *PickComponent* method also returns the original component. The original component should not be included in this function. Lines 219 to 224 iterate through the collection checking for and then removing the original component.

```
217     ' Remove the original component
218     Dim i
219     For i = 1 To cmpColl.Count
220         If cmpColl.Item(i).Name = cmpObj.Name Then
221             cmpColl.Remove(i)
222             Exit For
223         End If
224     Next
```

Finally, the script returns the component collection and exits the function.

```
226      ' Return the component collection
227      Set GetNearbyComponents = cmpColl
228  End Function
```

GetPlacementNumber() Function

Lines 232 through 247 declare a simple function that returns the placement number associated with a passed *Component* object.

```
230  ' Returns the placement number of this component as an integer
231  ' cmpObj - Component object
232  Function GetPlacementNumber(cmpObj)
```

Line 235 uses the *FindProperty* method to acquire the *Property* object for the *PLACEMENT_ORDER* property.

```
233      ' Get the property
234      Dim propObj
235      Set propObj = cmpObj.FindProperty(PLACEMENT_ORDER)
```

Line 239 checks to make sure that the property exists, in which case Line 240 returns the property value. If the property does not exist Line 242 sets the property value to –1 (to indicate a non-valid value).

```
237      ' Get the property value
238      Dim placeNumStr
239      If Not propObj Is Nothing Then
240          placeNumStr = propObj.Value
241      Else
242          placeNumStr = -1
243      End If
```

Line 246 converts the value (a string) to an integer and returns it.

```
245      ' Convert the value to an int and return
246      GetPlacementNumber = CInt(placeNumStr)
247  End Function
```

Helper Functions and Subroutines

This script section declares several generic "helper" routines. These routines can be used in other scripts and are not focused on this script exclusively.

Output() Function

Lines 255 through 267 declare a simple function that accepts a text string as a parameter and displays it in the **Output** window.

```
252  ' Appends the test string to the text in the output window.
253  ' Creates the output window if needed.
254  ' textStr - String
255  Function Output(textStr)
```

On Line 258, the script calls the LoadAddin() Function, which loads any required addins. In this example, it loads the **Output** window and places it at the bottom of the Xpedition Layout window. If the specified addin is already loaded the addin is returned. The first parameter to this function is the addin *ProgID* (the **Output** window for this script); the second parameter is an arbitrary addin name; the third parameter specifies an addin script (ignored in this example); and the fourth parameter specifies the addin placement location.

```
256      ' Get or load the outputwindow add-in
257      Dim outputAddinObj
258      Set outputAddinObj =
                     LoadAddin("MGCSDDOutputLogControl.Addin", _
259                  "Output Window", "",PlacementLocationBottom)
```

Line 263 acquires the addin control object.

```
261      ' Get the output control from the add-in
262      Dim outputCtrlObj
263      Set outputCtrlObj = outputAddinObj.Control
```

Line 266 calls the *AddTab* method, which adds a tab named “Output” to the **Output** window. The tab is now active, so the *AppendText* method appends the text string to display to the tab.

```
260
264
265      ' Get or add the Output tab and append the text
266      outputCtrlObj.AddTab("Output").AppendText(textStr & vbCrLf)
267  End Function
```

LoadAddin() Function

Lines 276 through 295 declare a function that loads any required addins. The first parameter to this function is the addin *ProgID*; the second parameter is an arbitrary addin name; the third parameter specifies an addin script; and the fourth parameter is the addin placement location.

```
269  ' Load an add-in and return it.  If the add-in is already
270  ' loaded it is returned.
271  ' progIdStr - String (prog id of add-in to load)
272  ' localNameStr - String (name assigned to add-in for use
                                        in later identification)
273  ' scriptStr - String (script to associate with add-in.
274  '             Unused in this script)
275  ' placementEnum - String (docking location of add-in)
276  Function LoadAddin(progIdStr, localNameStr, scriptStr, placementEnum)
```

Line 279 calls the FindAddin() Function, which returns the addin if it is already loaded or returns *Nothing* if the addin is not loaded.

```
277      ' Check if addin is already loaded.
278      Dim addinObj
279      Set addinObj = FindAddin(localNameStr)
```

Line 282 checks if the addin is already loaded. If not, then Lines 284 and 285 call the *Addins* collection *Add* method to load the addin. The first parameter to this method is the addin *ProgID*; the second parameter is an arbitrary addin name; the third parameter specifies an addin script; and the fourth parameter specifies the addin placement location.

```
281      ' Load it if not already loaded.
282      If addinObj Is Nothing Then
283          ' Use the add method on the Addins collection
                                                to load the add-in
284          Set addinObj = pcbAppObj.Addins.Add(progIdStr, _
285              localNameStr, scriptStr, placementEnum, "")
286      End If
```

Finally, Lines 289 through 295 assures the addin is visible, sets the addin as this function's return value, and exits the function.

```
280
287
288      ' Ensure the addin is visible
289      If Not addinObj Is Nothing Then
290          addinObj.Visible = True
291      End If
292
293      ' Return the addin
294      Set LoadAddin = addinObj
295  End Function
```

FindAddin() Function

Lines 300 through 313 declare a simple function that checks if an addin is already loaded. If the addin is loaded, the function returns that addin; if not loaded, the function returns *Nothing*. The function accepts a single string parameter that identifies the addin.

```
297  ' Returns the add-in if already loaded
298  ' Returns nothing otherwise
299  ' localNameStr - String (Name for identifying add-in)
300  Function FindAddin(localNameStr)
```

Line 302 initializes the return value of this function to *Nothing*.

```
301      ' Initialize the return value
302      Set FindAddin = Nothing
```

Lines 306 through 312 iterate through all addins checking the addin's *Name* against the passed string. If the addin is found, the script sets the return value of this function to the addin and exits the loop.

```

    ' Iterate through the add-ins.
    Dim addinObj
    For Each addinObj In pcbAppObj.Addins
        ' If we find the add-in return it.
        If addinObj.Name = localNameStr Then
            Set FindAddin = addinObj
            Exit For
        End If
    Next
End Function
```

RemoveTrailingNewLine() Function

This helper function accepts a text string and searches for a trailing newline (carriage return and line feed characters). If a newline is found the function removes the newline and returns the resulting string. If a newline is not found the function simply returns the string.

```
' Removes the trailing vbCrLf if it exists
' textStr - String
Function RemoveTrailingNewLine(textStr)
    If Right(textStr, 2) = vbCrLf Then
        textStr = Left(textStr, Len(textStr) - 2)
    End If
    RemoveTrailingNewLine = textStr
End Function
```

Creating and Testing the Script

1. Use the scripting editor of your choice to enter the scripts described (including the license server function) and save them out as *AssignPlacementOrder.vbs* and *AssemblyDRC.vbs*.

2. Close all instantiations of Xpedition Layout that are currently running, and then launch a new instantiation of this application.

3. Open the *Candy.pcb* design and turn on the display of the "Gripper Graphics" user layer.

4. Run the *AssignPlacementOrder.vbs* script.

5. Run the *AssemblyDRC.vbs* script.

6. Select component **M2** and attempt to place it in its current/original position. Observe that the **Output** window appears displaying a message informing you about a DRC violation as illustrated in Figure 151: Script Detects a DRC Violation.

7. Drag component **M2** away from capacitor **C1** and attempt to place it in a clear area. Ensure that the placement is successful as illustrated in Figure 152: Component is Successfully Placed.

Enhancing the Script

There are a number to enhance this script. Several suggestions follow (practice your scripting skills by implementing these suggestions):

- The *AssemblyDRC.vbs* script only checks for a DRC violation when you attempt to place a component. One useful addition to the script is to automatically check all of the existing placed components for violations when the script is first run.

- The *AssemblyDRC.vbs* script does not distinguish between component placement outlines and the sides of the board, so it may report non-existent errors. The script can be enhanced to handle this condition.

- In the Before Starting discussions at the beginning of this section (refer to Figure 152: Component is Successfully Placed), it noted that once the script detects an error, the message remains in the **Output** window even after the offending component is successfully placed. Another useful addition to the *AssemblyDRC.vbs* script is to remove any offending error message once the cause of that message is addressed and resolved. Use the Help system and an Object Browser to obtain the information you require in order to achieve this task – refer to "Object Browsers" on page 536 for more information on using Object Browsers.

Exercise 4: Removing Unconnected Pads from Pins and Vias

This script determines the connectivity of individual pins and vias and removes any pads that are not connected to any pins and vias from the padstack.

Approach:	To use a form that allows the end-user to select which pins and vias they want to process; to then determine the connectivity associated with each of the selected pins and vias and to remove unconnected pads accordingly.
Points of Interest:	– Using the IDE to create a form
	– Modifying padstacks
	– Working with connectivity
Items Used:	– Connected Object Property
	– Dictionary Object
	– Padstack Object
	– Objects Collection

Before Starting

The functionality provided by this script is very similar to the functionality provided by the Padstack Processor build into Xpedition Layout (you can access this processor by choosing **Edit > Modify > Padstack Processor**.) However, when it comes to deleting pads, the Padstack Processor has certain

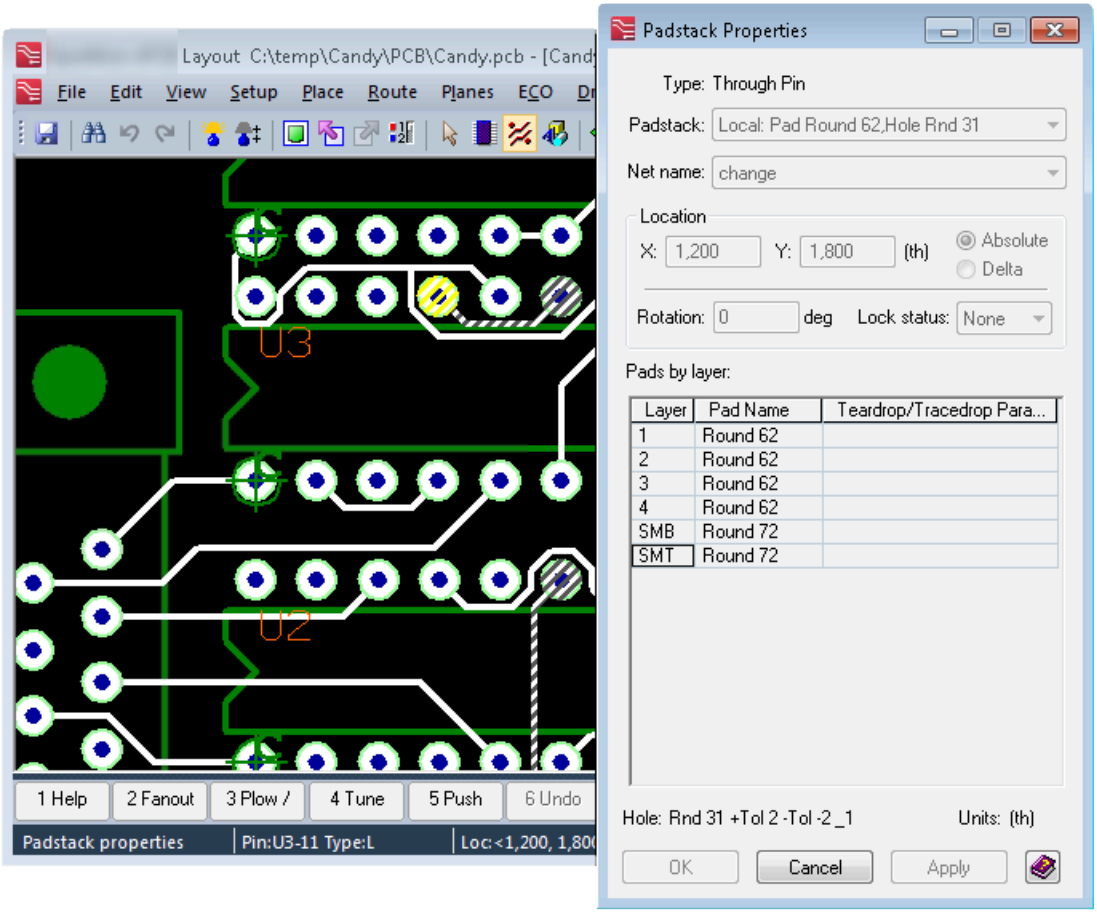
limitations, such as not being able to delete top and bottom pads. This script has no such limitations, and can be used as a starting point from which you can further customize its capabilities to meet your unique requirements.

Before starting, there are a few points regarding padstacks with which you should be familiar. A *Padstack* reference defines the base configuration of a padstack. *Padstack* objects – which may be associated with pins and vias – are instantiations of *Padstack* references.

When the script creates a new board design, it starts with a set of original *Padstack* references that are pulled from a library. Note that you cannot make changes to these originals, even if they are not being used. However, you can use the automation interface to create clones of these originals and change the clones. Also note that you cannot make changes to a clone once it is used; instead create another clone of the original (or a clone of a clone) and modify this new clone.

To understand how the script works, launch Xpedition Layout, open the *Candy.pcb* design, and select pin 11 on component **U3**. Open the **Padstack Properties** dialog as illustrated in Figure 154: U3.11 Padstack Before the Script Runs. Observe that the default padstack has pads named **Round 62** on routing layers 1, 2, 3, and 4; also pads named **Round 72** on the top and bottom solder mask layers.

Figure 154. U3.11 Padstack Before the Script Runs



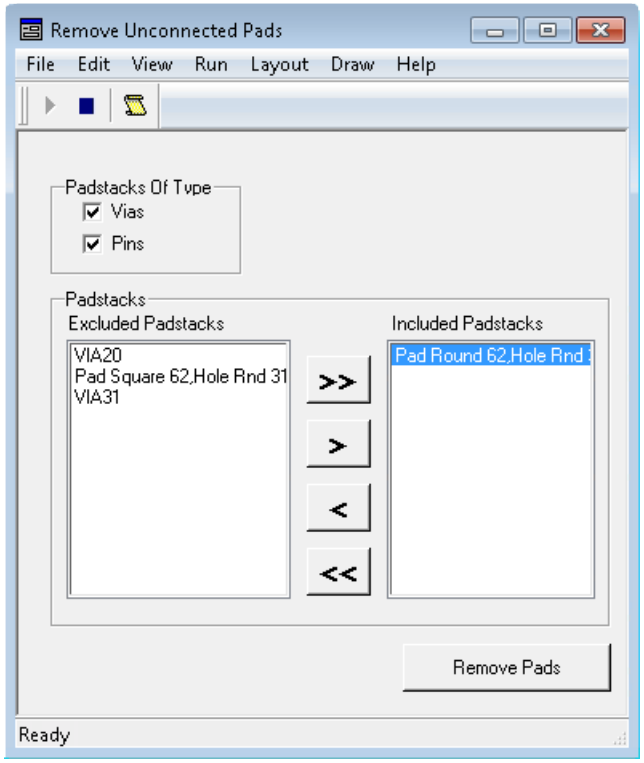
Drag-and-drop the *RemoveUnconnectedPads.efm* script onto the design. The Remove Unconnected Pads form opens as illustrated in Figure 155: Remove Unconnected Pads Form/Dialog.

Observe the two list options **Excluded Padstacks** (on the left) and **Included Padstacks** (on the right). Initially, these lists are empty. Click the **Vias** checkbox and two via types (*VIA20* and *VIA31*) appear in

the **Excluded Padstacks** list. Similarly, click the **Pads** checkbox and two pad types (*Pad Round 62, Hole Rnd 31* and *Pad Square 62, Hole Rnd 31*) appear in the **Excluded Padstacks** list.

If you select the *Pad Round 62, Hole Rnd 31* item in the **Excluded Padstacks** list and then click the button with the single right-pointing arrow “>”, the item moves from the **Excluded Padstacks** list to the **Included Padstacks** list.

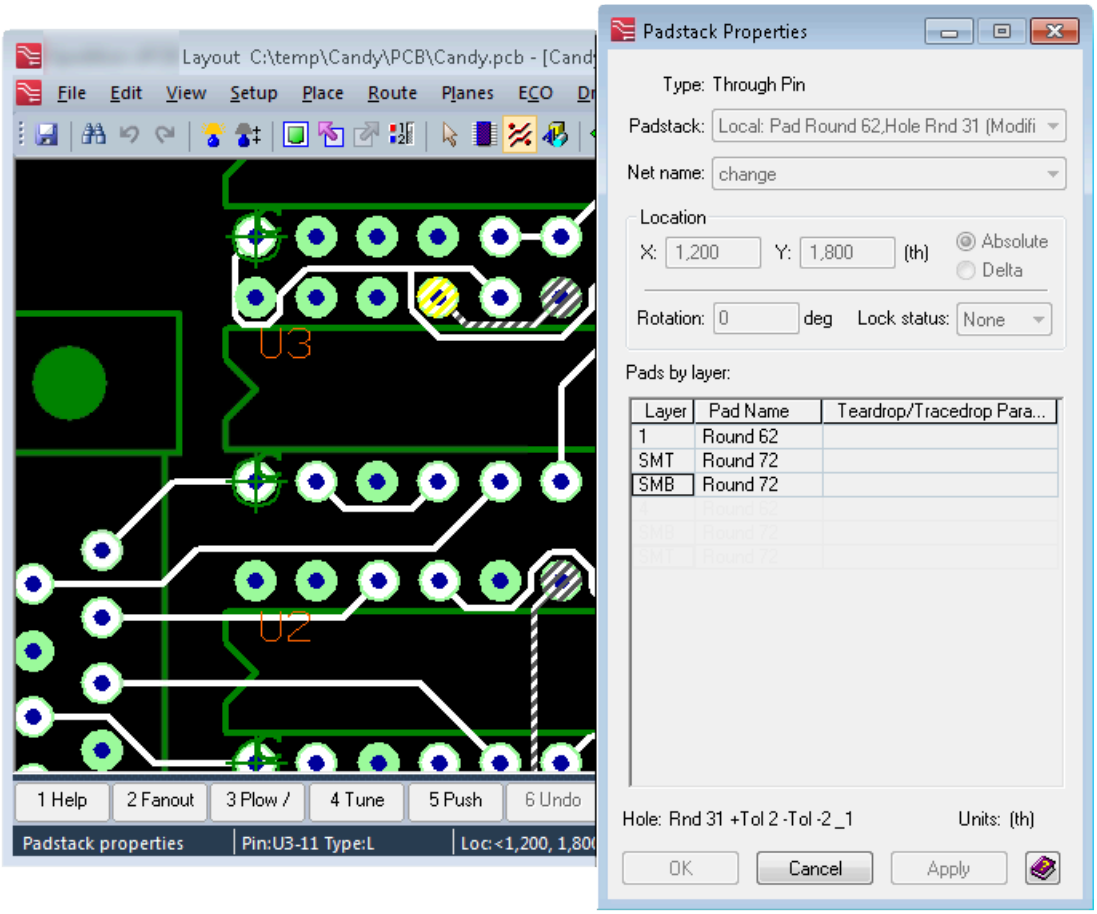
Figure 155. Remove Unconnected Pads Form/Dialog



Click the **Remove Pads** button. Select pin 11 on component **U3**, and re-access the **Padstack Properties** dialog, the result is as illustrated in Figure 156: U3.11 Padstack After the Script Runs.

Observe that a new padstack is created (cloned from the original padstack) and that – in addition to the **Round 72** pads on the top and bottom solder mask layers – this new version has only one **Round 62** pad on routing layer 1 (the pads on routing layers 2, 3, and 4 are removed). Also observe that any unconnected pads on other pins on layer 1 (**U3.12**, **U3.13**, and **U3.14**, for example) are removed leaving only the green solder mask pads.

Figure 156. U3.11 Padstack After the Script Runs



Form Design

The *RemoveUnconnectedPads.efm* script creates the form, and then it populates the various elements on the form with their section of the script. The key scripting sections "behind" the form are as follows (these are shown in the order in which they are presented):

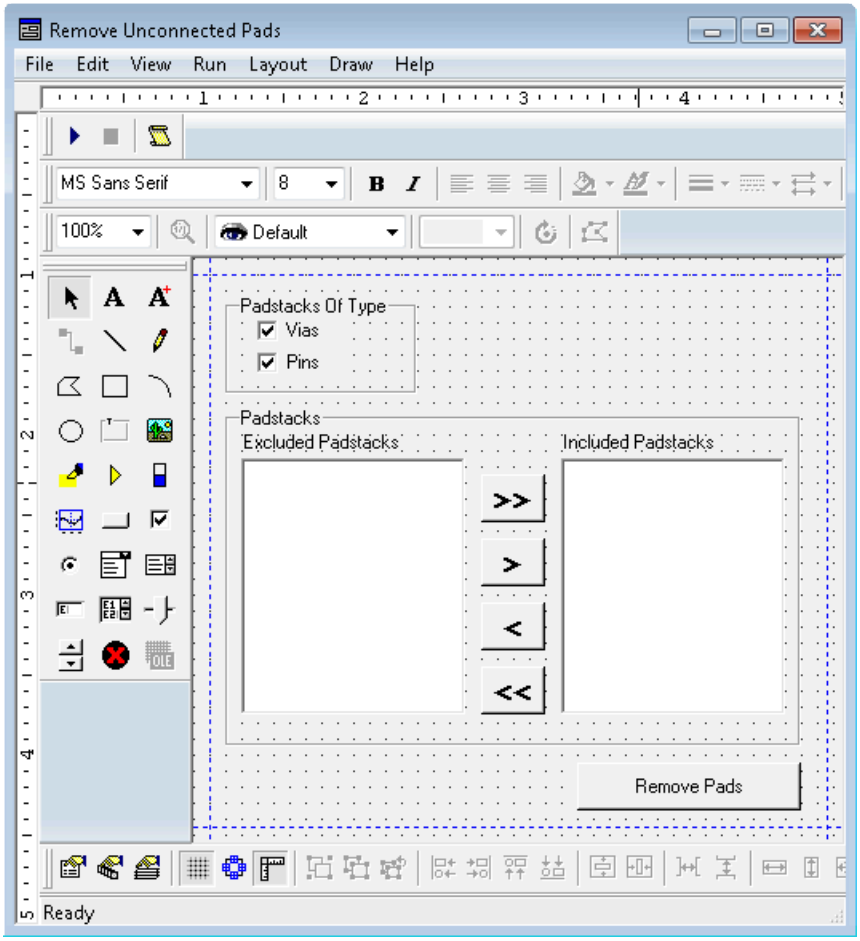
- Initialization and Setup
- Exclude List Initialization
- Include List Initialization
- Event Change Handler for Pins Checkbox
- Event Change Handler for Vias Checkbox
- Include All Button
- Exclude All Button

- Include Selected Button
- Exclude Selected Button
- Remove Unconnected Pads Button
- Helper Functions

Adding Elements to the Form

Launch the IDE in form editor mode as discussed in "Creating Forms and Dialogs Using the IDE" on page 69 and then create the form as illustrated in Figure 157: Creating the Form/Dialog. Choose **File > Save As** to save the form in whatever folder you are using to store your scripts; save the form with the name *RemoveUnconnectedPads.efm* (as described in Creating Forms and Dialogs Using the IDE, the *.efm* extension denotes a form, whereas the *.vbs* extension denotes a VBScript). As you add the code behind the various elements on the form, it is recommended to periodically save the work.

Figure 157. Creating the Form/Dialog



Observe two *Frames* on this form – one is labeled **Padstacks of Type** and the other is labeled **Padstacks**. Inside the **Padstacks of Type** frame are two *Checkboxes* labeled **Vias** and **Pins**.

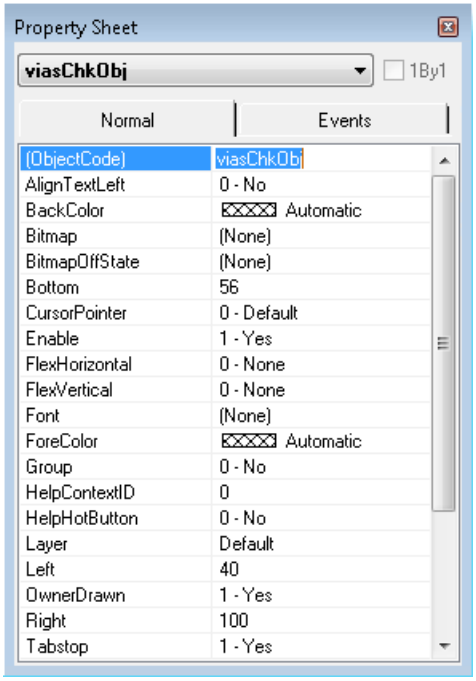
Inside the **Padstacks** frame are two *List* boxes; the left- and right-hand boxes are annotated with free text "**Excluded Padstacks**" and "**Included Padstacks**", respectively.

Also inside the **Padstacks** frame are four buttons with text annotations of "**>>**", "**>**", "**<**", and "**<<**". Finally, the button at the bottom of the form with the "**Remove Pads**" annotation.

Naming Elements on the Form

It is important that you name all of the objects on the form with which the script associates scripting code. Right-click the **Vias** checkbox control and select the **Object Properties** item from the pop-up menu. Change the **(Object Code)** item from its default value to the name *viasChkObj* as illustrated in Figure 158: Checkbox Property Sheet Dialog.

Figure 158. Checkbox Property Sheet Dialog



Observe that this name automatically appears in the field at the top of the form. This is the name the script uses to access this control (the control is accessed in the same way as other objects in the interface). Once you have made this change, click the red 'X' button located in the upper right-hand corner of the form to close the Property Sheet.

Repeat this process for the other items on the form as follows:

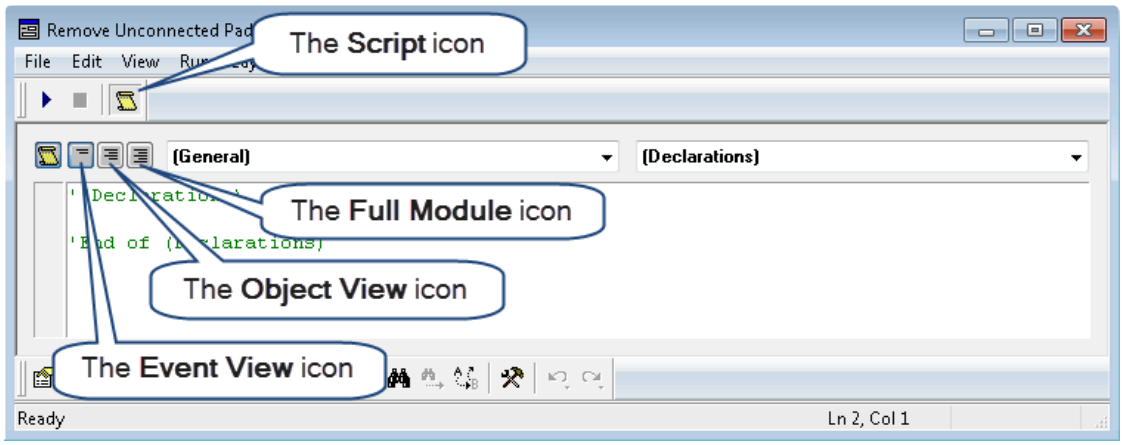
- Change the name of the **Pins** checkbox control to *pinsChkObj*.
- Change the name of the **Excluded Padstacks** list box to *excludeListObj*.
- Change the name of the **Included Padstacks** list box to *includeListObj*.
- Change the name of the **Include All** button (the **">>"** annotation) to *includeAllBtnObj*.
- Change the name of the **Include Selected** button (the **">"** annotation) to *includeBtnObj*.

- Change the name of the **Exclude Selected** button (the **"<"** annotation) to *excludeBtnObj*.
- Change the name of the **Exclude All** button (the **"<<"** annotation) to *excludeAllBtnObj*.
- Change the name of the **Remove Pads** button to *removePadsBtnObj*.

Preparing to Add Code "Behind" the Elements on the Form

After designing the form, you add the code (sub-scripts) "behind" the various elements on the form. There are a number of mechanisms to achieve this. For this example, click the **Script** icon and then make sure that the **Event View** icon is active as illustrated in Figure 159: Adding Code "Behind" the Form Elements.

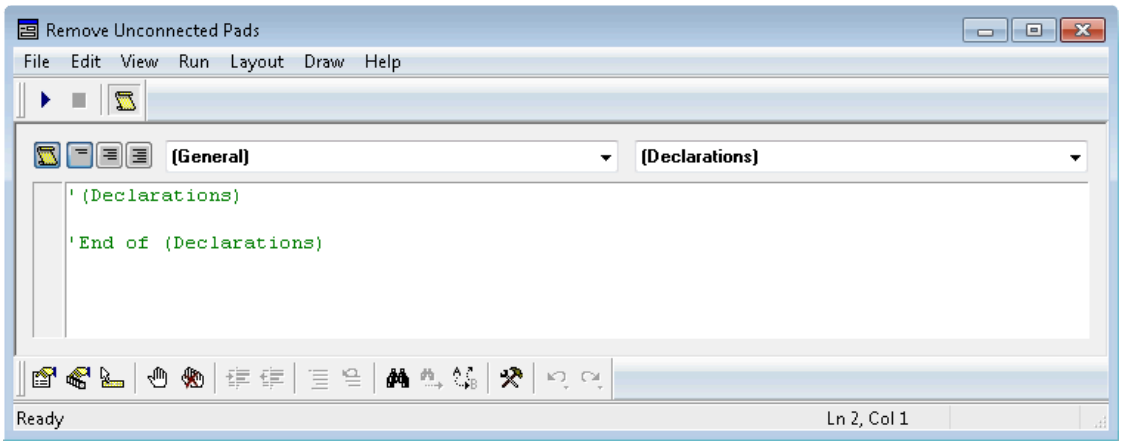
Figure 159. Adding Code "Behind" the Form Elements



Initialization and Setup

Ensure that **(General)** is selected in the left-hand (Object) list at the top of the form. This causes the right-hand list to be automatically populated with the **(Declarations)** item as illustrated in Figure 160: Adding the Initialization and Setup Code.

Figure 160. Adding the Initialization and Setup Code



Place the cursor between the *'(Declarations)* and *'End of (Declarations)* comments and then enter the code shown below. Lines 1 through 18 perform the usual tasks of acquiring the *Application* and *Document* objects and then licensing/validating the document. Lines 20 through 52 represent the standard *ValidateServer()* function.

This section is not discussing the *Initialization and Setup* code shown below, it is standard and is described in detail in previous sections.

CAUTION:
Do *not* enter the line numbers shown in any of the code associated with this section. They are included here only as an aid for the discussions.

```
 1
 2  Option Explicit
 3
 4  ' Add any type librarys to be used.
 5  Scripting.AddTypeLibrary("MGCPCB.ExpeditionPCBApplication")
 6
 7  ' Global variables
 8  Dim pcbAppObj                 'Application object
 9  Dim pcbDocObj                 'Document object
10
11  ' Get the application object.
12  Set pcbAppObj = Application
13
14  ' Get the active document
15  Set pcbDocObj = pcbAppObj.ActiveDocument
16
17  ' License the document
18  ValidateServer(pcbDocObj)
19
20  '''''''
```

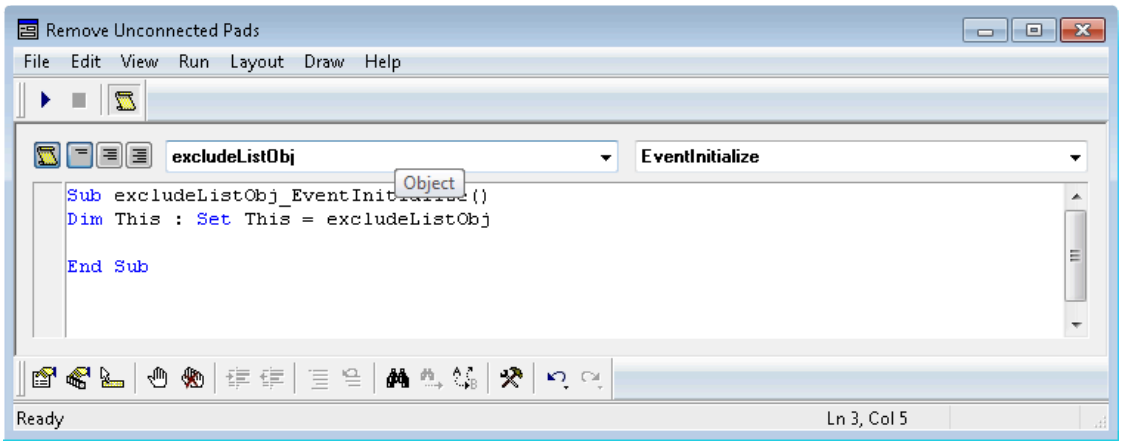
```
40      ' Turn off error messages.  Validate may fail if the token
          is incorrect
41      On Error Resume Next
42      Err.Clear
43
44      ' Ask the document to validate the license token
45      doc.Validate(licenseToken)
46      If Err Then
47          ValidateServer = 0
48      Else
49          ValidateServer = 1
50      End If
51
52  End Function
```

Exclude List Initialization

This function is called automatically when the dialog/form is first run; then it is called explicitly when the states of either the **Pins** or **Vias** checkboxes are modified.

Ensure that **excludeListObj** is selected in the left-hand list at the top of the form; also that **EventInitialize** is selected in the right-hand list at the top of the form (the **EventInitialize** for each control is automatically called when the form is first run; furthermore, it can be called from other locations in the form as required.). The result is illustrated in Figure 161: Adding the Exclude List Initialization Code.

Figure 161. Adding the Exclude List Initialization Code



Place your cursor on the blank line before the *End Sub* statement and enter the following code from Lines 3 to 31:

```
1  Sub excludeListObj_EventInitialize()
2  Dim This : Set This = excludeListObj
3      ' Clear the list
4      Call This.ResetContent()
5
6      ' Use a dictionary to determine if we have
7      ' already added this padstack
8      ' There may be duplicate padstack names because
9      ' a padstack that is processed with the
```

```
10      ' padstack processor or automation is considered
11      ' a different padstack with the same name
12      Dim dictObj
13      Set dictObj = Createobject("Scripting.Dictionary")
14
15      ' Get the collection of padstacks
16      Dim pstkColl
17      Set pstkColl = pcbDocObj.Padstacks
18
19      Dim pstkObj
20      For Each pstkObj In pstkColl
21          ' Only add it if it is a type the user wants
22          If ((pstkObj.Type = epcbPadstackObjectPin And
                                  pinsChkObj.Value = 1 And _
23               Not pstkObj.IsSMD) Or _
24              (pstkObj.Type = epcbPadstackObjectVia And
                                 viasChkObj.Value = 1)) And _
25             Not dictObj.Exists(pstkObj.Name) Then
26                 ' Add it to the list
27              Call This.AddString(pstkObj.Name)
28              ' Add it to the dictionary so we know not to
                    add it twice
29              Call dictObj.Add(pstkObj.Name, True)
30          End If
31      Next
32  End Sub
```

Let's walk through this code line-by-line.

Line 4 clears any existing entries in the list.

```
4      Call This.ResetContent()
```

The form presents a single list entry for each excluded *Padstack* reference and all of its clones. In order to facilitate this, on Lines 12 and 13, the script creates a *Dictionary* object. The *Dictionary* is used to quickly check if any particular name is already added.

```
12      Dim dictObj
13      Set dictObj = Createobject("Scripting.Dictionary")
```

Lines 16 and 17 acquire a collection of all the *Padstack* objects in the design (this includes the original references and all of the clones).

```
16      Dim pstkColl
17      Set pstkColl = pcbDocObj.Padstacks
```

Line 19 instantiates a variable to hold a *Padstack* object. Lines 20 through 31 iterates through each *Padstack* object in the collection. For each *Padstack* object the script checks:

1. If it is a *Pin Padstack* AND the **Pins** checkbox is active AND it is not a surface-mount device (SMD) or

2. If it is a *Via Padstack* AND the **Vias** checkbox is active.

When the form is first run, neither the **Pins** nor **Vias** checkboxes are selected/active, so nothing is added to the list and it is therefore empty.)

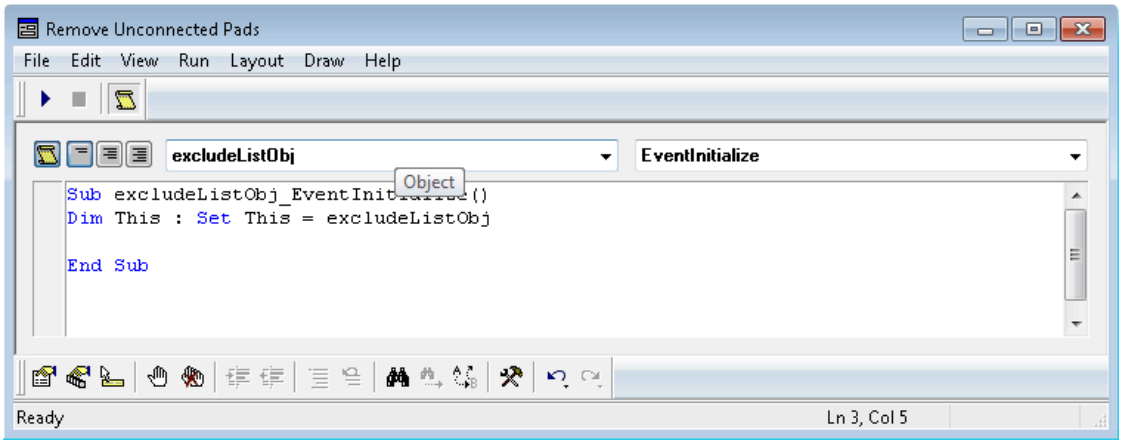
If either of these cases is true, Line 25 checks if this entry already exists in the dictionary. If it does not already exist, Line 27 adds it to the list and Line 29 adds it to the *Dictionary* object.

```
19      Dim pstkObj
20      For Each pstkObj In pstkColl
21          ' Only add it if it is a type the user wants
22          If ((pstkObj.Type = epcbPadstackObjectPin And
                                  pinsChkObj.Value = 1 And _
23               Not pstkObj.IsSMD) Or _
24              (pstkObj.Type = epcbPadstackObjectVia And
                                 viasChkObj.Value = 1)) And _
25             Not dictObj.Exists(pstkObj.Name) Then
26              ' Add it to the list
27              Call This.AddString(pstkObj.Name)
28              ' Add it to the dictionary so we know not to add it twice
29              Call dictObj.Add(pstkObj.Name, True)
30          End If
31      Next
32  End Sub
```

Include List Initialization

Ensure that **includeListObj** is selected in the left-hand list at the top of the form; also that **EventInitialize** is selected in the right-hand list at the top of the form. The result is illustrated in Figure 162: Adding the Include List Initialization Code.

Figure 162. Adding the Include List Initialization Code



Place your cursor on the blank line before the *End Sub* statement and enter Lines 3 and 4 from the following code:

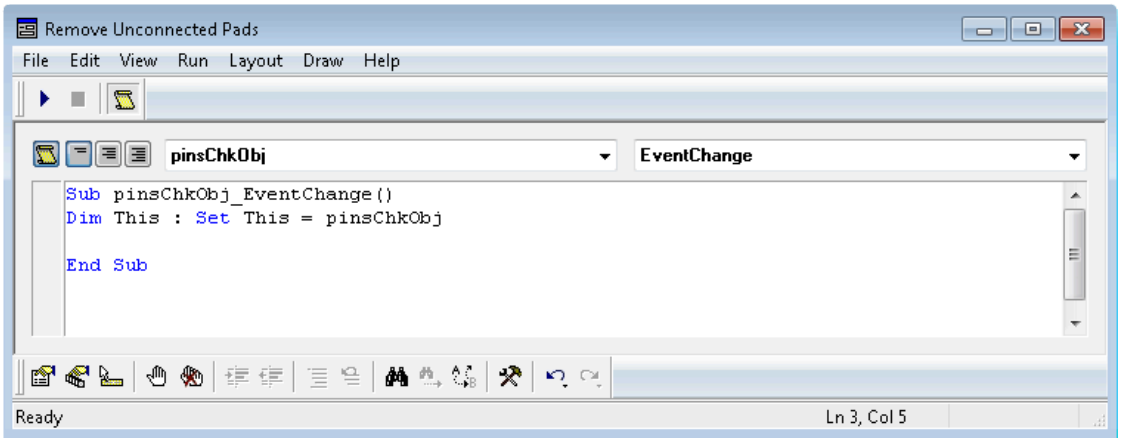
```
1  Sub includeListObj_EventInitialize()
2  Dim This : Set This = includeListObj
3      ' Clear the list
4      Call This.ResetContent()
5  End Sub
```

Line 4 clears any existing entries in the list.

Event Change Handler for Pins Checkbox

Ensure that **pinsChkObj** is selected in the left-hand list at the top of the form; also that **EventChange** is selected in the right-hand list at the top of the form (**EventChange** is fired automatically any time the control changes; in the case of a checkbox, it is fired whenever that box is checked or unchecked). The result is illustrated in Figure 163: Adding the Pins Checkbox Event Change Handler Code.

Figure 163. Adding the Pins Checkbox Event Change Handler Code



Place your cursor on the blank line before the *End Sub* statement and enter Lines 3 through 5 from the following code:

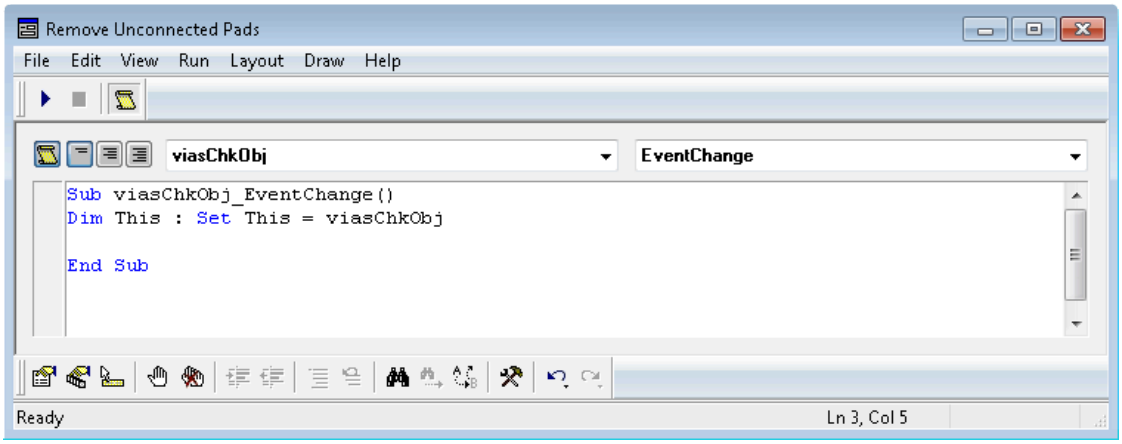
```
1  Sub pinsChkObj_EventChange()
2  Dim This : Set This = pinsChkObj
3      ' Reinitialize the lists
4      Call excludeListObj_EventInitialize()
5      Call includeListObj_EventInitialize()
6  End Sub
```

Whenever the state of the **Pins** checkbox changes (regardless if it is selected or deselected), Lines 4 and 5 call the functions that re-initialize the **Exclude List** and **Include List**, respectively.

Event Change Handler for Vias Checkbox

Ensure that **viasChkObj** is selected in the left-hand list at the top of the form; also that **EventChange** is selected in the right-hand list at the top of the form. The result is illustrated in Figure 164: Adding the Vias Checkbox Event Change Handler Code.

Figure 164. Adding the Vias Checkbox Event Change Handler Code



Place your cursor on the blank line before the *End Sub* statement and enter Lines 3 through 5 from the following code:

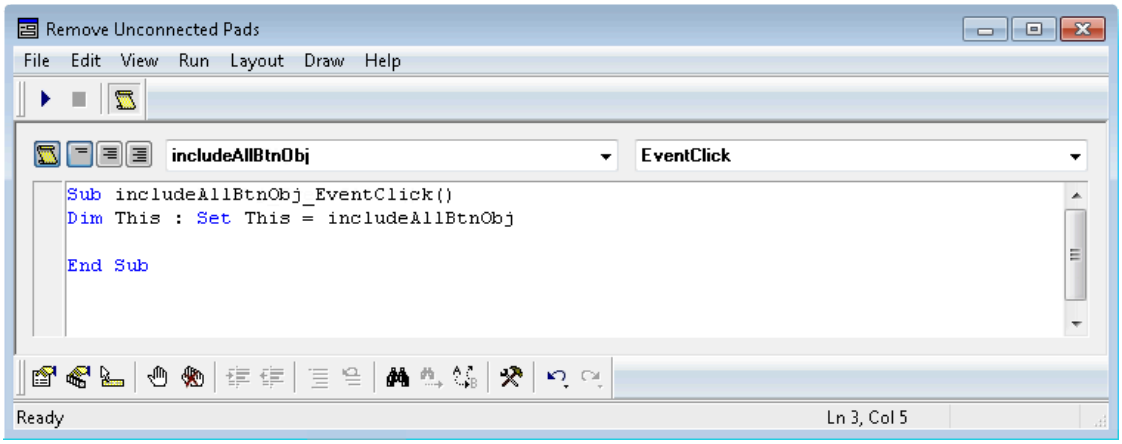
```
1  Sub viasChkObj_EventChange()
2  Dim This : Set This = viasChkObj
3      ' Reinitialize the lists
4      Call excludeListObj_EventInitialize()
5      Call includeListObj_EventInitialize()
6  End Sub
```

Whenever the state of the **Vias** checkbox changes (regardless if it is selected or de-selected), Lines 4 and 5 call the functions that re-initialize the **Exclude List** and **Include List**, respectively.

Include All Button

Ensure that **includeAllBtnObj** is selected in the left-hand list at the top of the form; also that **EventClick** is selected in the right-hand list at the top of the form. The result is illustrated in Figure 165: Adding the Include All Button Code.

Figure 165. Adding the Include All Button Code



Place your cursor on the blank line before the *End Sub* statement and enter Lines 3 through 11 from the following code:

```
 1  Sub includeAllBtnObj_EventClick()
 2  Dim This : Set This = includeAllBtnObj
 3      Dim entryInt
 4      ' Add all pastacks to include list
 5      For entryInt = excludeListObj.GetCount() To 0 Step -1
 6          If Not excludeListObj.GetText(entryInt) = "" then
 7              Call
         includeListObj.AddString(excludeListObj.GetText(entryInt))
 8              ' Remove the entry from the exclude list
 9              call excludeListObj.DeleteString(entryInt)
10          End If
11      Next
12  End Sub
```

On Lines 5 through 11, the script iterates through all items that are in the **Exclude List**; for each of these items it adds the item to the **Include List** and then deletes the item from the **Exclude List**.

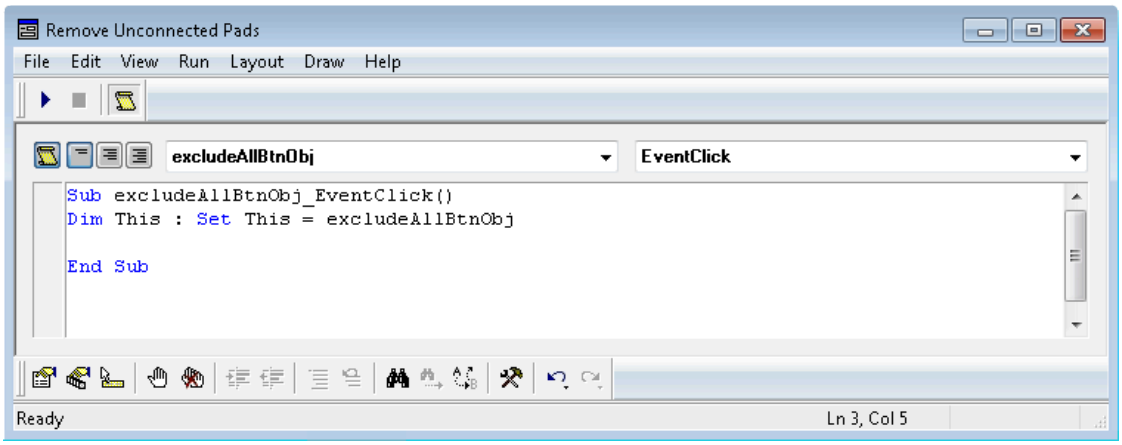
Tip
This button and the other **Include/Exclude** buttons are implemented in a way that they work for any list contents.

Tip
It is possible to copy a control from one form to another. When you do this, all of the code associated with that control is also copied.

Exclude All Button

Ensure that **excludeAllBtnObj** is selected in the left-hand list at the top of the form; also that **EventClick** is selected in the right-hand list at the top of the form. The result is illustrated in Figure 166: Adding the Exclude All Button Code.

Figure 166. Adding the Exclude All Button Code



Place your cursor on the blank line before the *End Sub* statement and enter Lines 3 through 12 from the following code:

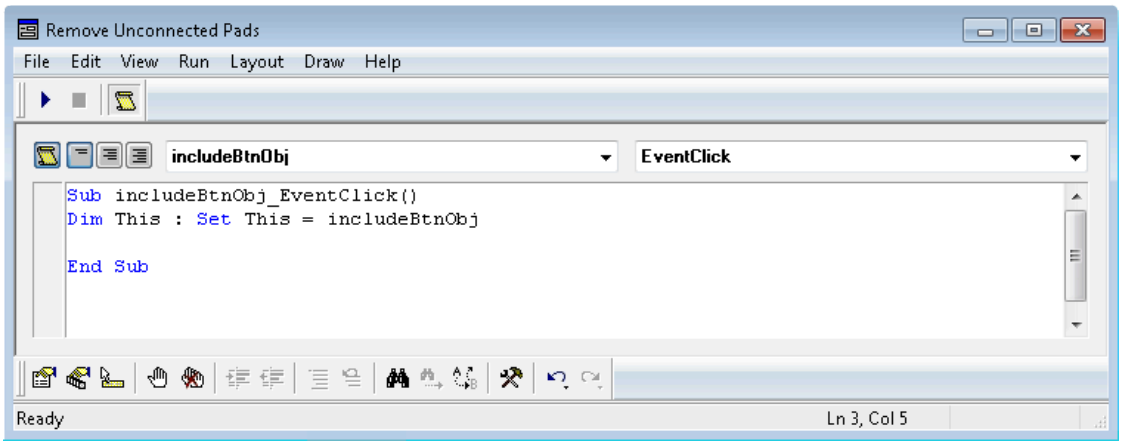
```
 1  Sub excludeAllBtnObj_EventClick()
 2  Dim This : Set This = excludeAllBtnObj
 3      Dim entryInt
 4      ' Add all pastacks to include list
 5      For entryInt = includeListObj.GetCount() To 0 Step -1
 6          If Not includeListObj.GetText(entryInt) = "" Then
 7              ' Add the entry to the exclude list
 8              Call
         excludeListObj.AddString(includeListObj.GetText(entryInt))
 9              ' Remove the entry from the include list
10              Call includeListObj.DeleteString(entryInt)
11          End If
12      Next
13  End Sub
```

On Lines 5 through 11, the script iterates through all items in the **Include List**; for each of these items it adds the item to the **Exclude List** and then deletes the item from the **Include List**.

Include Selected Button

Ensure that **includeBtnObj** is selected in the left-hand list at the top of the form; also that **EventClick** is selected in the right-hand list at the top of the form. The result is illustrated in Figure 167: Adding the Include Selected Button Code.

Figure 167. Adding the Include Selected Button Code



Place your cursor on the blank line before the *End Sub* statement and enter Lines 3 through 11 from the following code:

```
 1  Sub includeBtnObj_EventClick()
 2  Dim This : Set This = includeBtnObj
 3      ' Get the selected entry
 4      Dim entryInt
 5      entryInt = excludeListObj.GetCurSel()
 6      If Not excludeListObj.GetText(entryInt) = ""  Then
 7          ' Add this entry to the include list
```

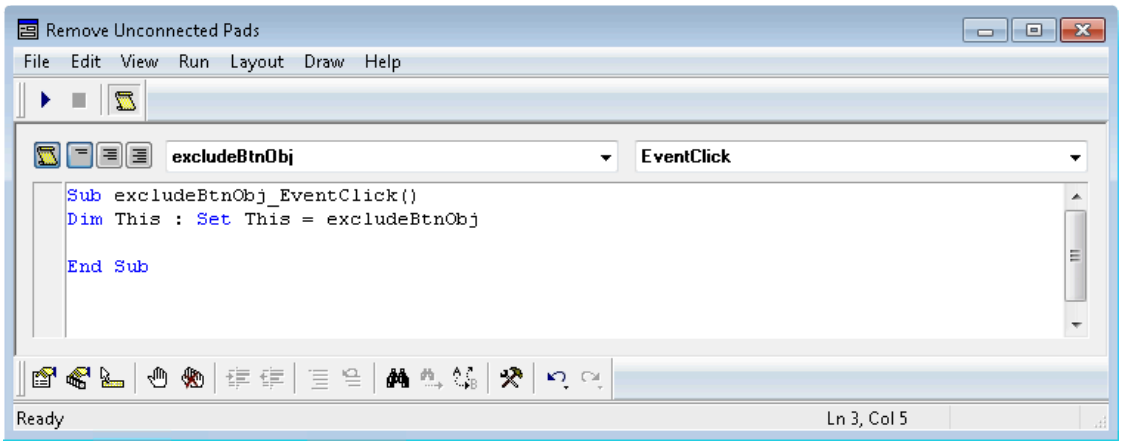
```
 8          Call
                 includeListObj.AddString(excludeListObj.GetText(entryInt))
 9          ' Remove it from the exclude list
10          Call excludeListObj.DeleteString(entryInt)
11      End If
12  End Sub
```

This is similar to the code for the **Include All** button. The main difference is that there is no loop. Instead, on Line 5, the script acquires the currently selected item in the **Exclude List**; on Line 8 it adds the item to the **Include List**; and on Line 10 it deletes the item from the **Exclude List**.

Exclude Selected Button

Ensure that **excludeBtnObj** is selected in the left-hand list at the top of the form; also that **EventClick** is selected in the right-hand list at the top of the form. The result is illustrated in Figure 168: Adding the Exclude Selected Button Code.

Figure 168. Adding the Exclude Selected Button Code



Place your cursor on the blank line before the *End Sub* statement and enter Lines 3 through 11 from the following code:

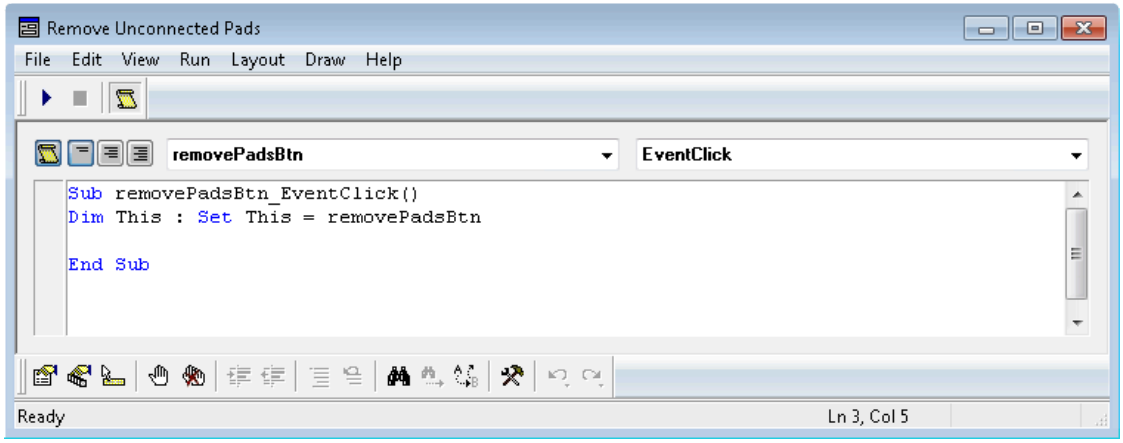
```
 1  Sub excludeBtnObj_EventClick()
 2  Dim This : Set This = excludeBtnObj
 3      ' Get the selected entry
 4      Dim entryInt
 5      entryInt = includeListObj.GetCurSel()
 6      If Not includeListObj.GetText(entryInt) = "" Then
 7          ' Add this entry to the include list
 8          Call
                 excludeListObj.AddString(includeListObj.GetText(entryInt))
 9          ' Remove it from the exclude list
10          Call includeListObj.DeleteString(entryInt)
11      End If
12  End Sub
```

This is similar to the code for the **Exclude All** button. The main difference is that there is no loop. Instead, on Line 5, the script acquires the currently selected item in the **Include List**; on Line 8 it adds the item to the **Exclude List**; and on Line 10 it deletes the item from the **Include List**.

Remove Unconnected Pads Button

Ensure that **removePadsBtn** is selected in the left-hand list at the top of the form; also that **EventClick** is selected in the right-hand list at the top of the form. The result is illustrated in Figure 169: Adding the Remove Pads Button Code.

Figure 169. Adding the Remove Pads Button Code



Place your cursor on the blank line before the *End Sub* statement and enter Lines 3 through 49 from the following code:

```
 1  Sub removePadsBtn_EventClick()
 2  Dim This : Set This = removePadsBtn
 3      ' Build a collection of pins and vias to be processed
 4
 5      ' Create an empty generic collection
 6      Dim pinViaColl
 7      Set pinViaColl = pcbAppObj.Utility.NewObjects()
 8
 9      ' Get the collection of all pins
10      Dim pinColl, pinObj
11      Set pinColl = pcbDocObj.Pins
12
13      ' Get the collection of all vias
14      Dim viaColl, viaObj
15      Set viaColl = pcbDocObj.Vias
16
17      ' Add objects from the include list to this collection
18      Dim entryInt
19      For entryInt = 0 To includeListObj.GetCount()
20          Dim pstkNameStr
21          pstkNameStr = includeListObj.GetText(entryInt)
22          If Not pstkNameStr = "" Then
23              Dim i
24              For i = pinColl.Count To 1 Step -1
25                  Set pinObj = pinColl.Item(i)
26                  If pinObj.OriginalPadstack.Name = pstkNameStr Then
27                      ' Add it to the collection to be processed
```

```
28                      pinViaColl.Add(pinObj)
29                      ' And remove it so we don't process it next time
30                      pinColl.Remove(i)
31                  End If
32              Next
33              For i = viaColl.Count To 1 Step -1
34                  Set viaObj = viaColl.Item(i)
35                  If viaObj.OriginalPadstack.Name =
                                                pstkNameStr Then
36                      ' Add it to the collection to be processed
37                      pinViaColl.Add(viaObj)
38                      ' And semove it so we don't process it next time
39                      viaColl.Remove(i)
40                  End If
41              Next
42          End If
43      Next
44
45      pcbDocObj.TransactionStart
46      ' Remove the unconnected pads.
47      Call RemoveUnconnectPads(pinViaColl)
48      pcbDocObj.TransactionEnd
49
50  End Sub
```

The **Remove Unconnected Pads** button functionality builds a simple collection that includes both pins and vias that use the *PadstackObject* objects in the **Include List**. With this collection, the script removes any unconnected pads from the pins and vias in the collection.

On Lines 6 and 7, the script uses the *NewObjects* property of the *Utility* object to create a new/empty object collection that can contain objects of different types.

```
 5      ' Create an empty generic collection
 6      Dim pinViaColl
 7      Set pinViaColl = pcbAppObj.Utility.NewObjects()
```

Lines 10 and 11 acquire a collection of all of the *Pin* objects in the design.

```
 9      ' Get the collection of all pins
10      Dim pinColl, pinObj
11      Set pinColl = pcbDocObj.Pins
```

Lines 14 and 15 acquire a collection of all of the *Via* objects in the design.

```
14      Dim viaColl, viaObj
15      Set viaColl = pcbDocObj.Vias
```

On Lines 18 through 43, the script iterates through each item (*Padstack* name) in the **Include List**. Line 21 returns the name for the current entry and Line 22 checks to assure that this name is not empty.

```
17      ' Add objects from the include list to this collection
18      Dim entryInt
19      For entryInt = 0 To includeListObj.GetCount()
20          Dim pstkNameStr
21          pstkNameStr = includeListObj.GetText(entryInt)
22          If Not pstkNameStr = "" Then
```

If the Padstack name is not empty, then Lines 24 through 32 iterate through the *Pin* collection in reverse order. Line 26 checks if this *Pin* object has the same *Padstack* name as the current *Padstack*. If it does match, then Line 28 adds the *Pin* object to the heterogeneous *Pin/Via* collection and Line 30 removes the *Pin* object from the *Pin* collection. (The script performs this removal only as a performance enhancement to save time checking this *Pin* the next time).

```
23              Dim i
24              For i = pinColl.Count To 1 Step -1
25                  Set pinObj = pinColl.Item(i)
26                  If pinObj.OriginalPadstack.Name = pstkNameStr Then
27                      ' Add it to the collection to be processed
28                      pinViaColl.Add(pinObj)
29                      ' And remove it so we don't process it next time
30                      pinColl.Remove(i)
31                  End If
32              Next
```

Similarly, on Lines 33 through 41, the script iterates through the *Vias* collection in reverse order. Line 35 checks if this *Via* object has the same *Padstack* name as the current *Padstack*. If the names match, then Line 37 adds the *Via* object to the heterogeneous *Pin/Via* collection and Line 39 removes the *Via* object from the *Vias* collection. (The script performs the removal only as a performance enhancement to save time checking this *Via* the next time).

```
33              For i = viaColl.Count To 1 Step -1
34                  Set viaObj = viaColl.Item(i)
35                  If viaObj.OriginalPadstack.Name = pstkNameStr Then
36                      ' Add it to the collection to be processed
37                      pinViaColl.Add(viaObj)
38                      ' And semove it so we don't process it next time
39                      viaColl.Remove(i)
40                  End If
41              Next
42          End If
43      Next
```

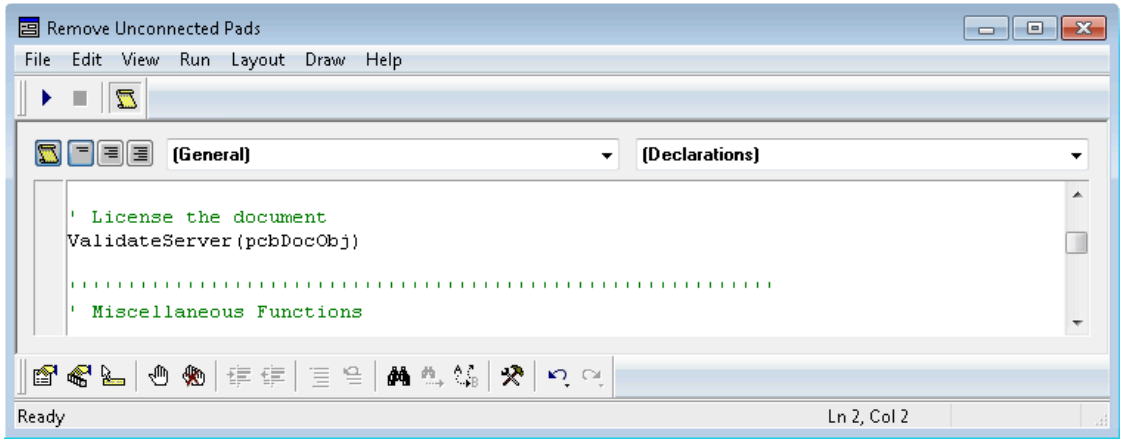
Line 45 starts a transaction, which allows undo of the changes made by this routine with a single **Undo**. Line 47 calls the RemoveUnconnectPads() Subroutine and passes it the collection just built. Line 48 ends the transaction.

```
45      pcbDocObj.TransactionStart
46      ' Remove the unconnected pads.
47      Call RemoveUnconnectPads(pinViaColl)
48      pcbDocObj.TransactionEnd
49
50  End Sub
```

Helper Functions

Ensure that **(General)** is selected in the left-hand (Object) list at the top of the form. This causes the right-hand list to be automatically populated with the **(Declarations)** item as illustrated in Figure 170: Adding the RemoveUnconnectPads and GetConnectedLayers Code.

Figure 170. Adding the RemoveUnconnectPads and GetConnectedLayers Code



Scroll until you reach the comment preceding the code associated with the *ValidateServer()* function, place your cursor just before this function, and insert the following code associated with two helper routines: RemoveUnconnectPads() Subroutine and GetConnectedLayers() Function:

```
 1
 2  ''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
 3  ' Utility functions
 4
 5  ' Removes unconnected pads from padstack objects in collection
 6  ' pstkObjColl - PadstackObject Collection
 7  Sub RemoveUnconnectPads(pstkObjColl)
 8      Dim pstkObjObj
 9      For Each pstkObjObj In pstkObjColl
10          ' Initilize an array of bools where entry i
11          ' corresponds to layer i.  Let false mean
12          ' there is no connection for that layer
13          Dim layerArr
14          layerArr = GetConnectedLayers(pstkObjObj)
15
16          ' Padstacks must be cloned to be modified.
17          Dim newPstkObj
18          Set newPstkObj = pstkObjObj.CurrentPadstack.Clone()
19
20          ' Remove all the pads
21          Dim lyrObject
22          Set lyrObject = pcbAppObj.Utility.NewLayerObject()
23          Dim modifiedBool : modifiedBool = False
24          Dim i
25          For i = 1 To UBound(layerArr)
26              ' If this layer isn't connected and there
```

```
            ' is a pad on the layer delete it
            If (Not layerArr(i)) And
                        (Not newPstkObj.Pads(i).Count = 0 )Then
                lyrObject.ConductorLayer = i
                newPstkObj.DeletePad(lyrObject)
                modifiedBool = True
            End If
        Next

        ' Apply cloned padstack back to the padstack object
        If modifiedBool Then
            pstkObjObj.CurrentPadstack = newPstkObj
        End If
    Next
End Sub

' Returns an array for all layers where entry i
' contains true if layer i is connected and false
' if not connected.
' pstkObjObj - PadstackObject Object
Function GetConnectedLayers(pstkObjObj)
    ' Initilize an array of bools where entry i
    ' corresponds to layer i.  Let false means
    ' there is not connection for that layer
    Dim layerArr()
    ReDim layerArr(pcbDocObj.LayerCount)
    Dim i
    For i = 1 to UBound(layerArr)
        layerArr(i) = False
    Next

    ' Determine which layers are missing connections
    Dim connectedObjColl
    Set connectedObjColl = pstkObjObj.ConnectedObjects

    Dim connectedObj
    For Each connectedObj In connectedObjColl
        ' This must be a metal object and all
        ' metal objects have a Layer property
        ' so we can use the Layer property
        ' without determining the object type
        layerArr(connectedObj.Layer) = True
    Next

    ' Return the array
    GetConnectedLayers = layerArr
End Function
```

RemoveUnconnectPads() Subroutine

This routine accepts as input a collection of *Pins* and *Vias*.

```
1
2  ''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
3  ' Utility functions
4
5  ' Removes unconnected pads from padstack objects in collection
6  ' pstkObjColl - PadstackObject Collection
7  Sub RemoveUnconnectPads(pstkObjColl)
```

On Lines 9 through 39, the script uses a *For … Next* loop to iterate through the collection. Line 14 calls the GetConnectedLayers() Function, which returns an array of the layers that specifies whether or not each layer has a connection.

```
10          ' Initilize an array of bools where entry i
11          ' corresponds to layer i.  Let false means
12          ' there is not connection for that layer
13          Dim layerArr
14          layerArr = GetConnectedLayers(pstkObjObj)
```

The only way to modify a *PadstackObject* object is to first clone it; this is done on Lines 17 and 18.

```
16          ' Padstacks must be cloned to be modified.
17          Dim newPstkObj
18          Set newPstkObj = pstkObjObj.CurrentPadstack.Clone()
```

Lines 21 and 22 create a new *Layer* object (this is required by the *DeletePad* method). Line 23 creates a Boolean flag to keep track of whether or not a *PadstackObject* object is modified. The script initializes this flag to *False*.

```
20          ' Remove all the pads
21          Dim lyrObject
22          Set lyrObject = pcbAppObj.Utility.NewLayerObject()
23          Dim modifiedBool : modifiedBool = False
```

Lines 25 through 33 iterate through the connected layers array. Line 28 checks if the item in the unconnected layer array is *False* (indicating no connection) and if there is no connection it checks if there is a *Pad* on this layer. If there is a *Pad* on this layer, Line 29 sets the *Layer* object to the current layer; Line 30 deletes the *Pad*; Line 31 sets the modified flag to *True*.

```
24          Dim i
25          For i = 1 To UBound(layerArr)
26              ' If this layer isn't connected and there
27              ' is a pad on the layer delete it
28              If (Not layerArr(i)) And
                            (Not newPstkObj.Pads(i).Count = 0 )Then
29                  lyrObject.ConductorLayer = i
30                  newPstkObj.DeletePad(lyrObject)
31                  modifiedBool = True
32              End If
33          Next
```

Line 36 checks if a *PadstackObject* modification occurred and if so assigns the cloned *PadstackObject* back to its original *PadstackObject* object (*Pin* or *Via*).

```
35          ' Apply cloned padstack back to the padstack object
36          If modifiedBool Then
                pstkObjObj.CurrentPadstack = newPstkObj
38          End If
39      Next
40  End Sub
```

GetConnectedLayers() Function

This function accepts as input a single *PadstackObject* object (*Pin* or *Via*), determines its connections, and returns an array of layers with each layer marked as connected or unconnected.

```
42  ' Returns an array for all layers where entry i
43  ' contains true if layer i is connected and false
44  ' if not connected.
45  ' pstkObjObj - PadstackObject Object
46  Function GetConnectedLayers(pstkObjObj)
```

On Line 50, the script instantiates a *Layer* array and on Line 51 initializes the array to contain the number of layers on the board. Array numbering starts from 0, you can create an array from 0 to (*LayerCount* – 1), but it is easier to create an array containing from 0 to *LayerCount* and then only use array items 1 to *LayerCount* (this is easier because the board's layers are numbered from 1 upwards).

```
47      ' Initilize an array of bools where entry i
48      ' corresponds to layer i.  Let false means
49      ' there is not connection for that layer
50      Dim layerArr()
51      ReDim layerArr(pcbDocObj.LayerCount)
```

Lines 53 through 55 iterate through the array, setting the Boolean value associated with each layer to *False*.

```
52      Dim i
53      For i = 1 to UBound(layerArr)
54          layerArr(i) = False
55      Next
```

Lines 58 and 59 acquires the collection of *Connected* objects for this *PadstackObject* object.

```
57      ' Determine which layers are missing connections
58      Dim connectedObjColl
59      Set connectedObjColl = pstkObjObj.ConnectedObjects
```

Lines 62 through 68 iterate through the *Connected objects* collection; whenever it finds a connected object it sets the corresponding Boolean flag in the array to be *True* (that is, the *Layer* property of each *Connected* object in the collection indexes into the layer array to set the appropriate Boolean value to *True*).

```
61      Dim connectedObj
62      For Each connectedObj In connectedObjColl
63          ' This must be a metal object and all
64          ' metal objects have a Layer property
```

```
65          ' so we can use the Layer property
66          ' without determining the object type
67          layerArr(connectedObj.Layer) = True
68      Next
```

Finally, the script sets the return value to the array of Boolean values and exits the function.

```
70      ' Return the array
71      GetConnectedLayers = layerArr
72  End Function
```

Creating and Testing the Script

1. Use the IDE to create the form and populate it.
2. Close any instantiations of Xpedition Layout that are currently running, and then launch a new instantiation of this application.
3. Open the *Candy.pcb* design and save it as *CandyPadTest.pcb*.
4. Run the *RemoveUnconnectedPads.efm* form/script and experiment with it to get a good feel for how it works, and then consider some of the ways in which this form/script can be augmented as discussed in the following topic.

Enhancing the Script

There are a number of ways to enhance this script. Several suggestions follow (practice your scripting skills by implementing these suggestions):

- Add a **Cancel** button to the form so that the end-user can dismiss the form without performing any actions.
- Add functionality so that when the end-user clicks the **Remove Pads** button, the script generates a report and/or presents the feedback as to what design modifications the script performed.
- Add a check box allowing/disabling the removal of pads on the top and bottom routing layers.
- Add functionality to select/highlight the pins/vias that the script modifies.
- Add functionality to be able to click a selected/highlighted pin/via and present the end-user a *MsgBox()* with "Was-Is" type information for the pin/via.

Exercise 5: Integration with Excel

This script creates a new Excel spreadsheet and then automatically loads the spreadsheet with component information that the script extracts from the layout design. The script also provides a cross-probing capability from the layout design to the Excel spreadsheet.

Approach:	To use Excel's automation interface to create a new spreadsheet and to populate it with component information.

	Also to manipulate selection and cell coloring using Excel's automation interface.
Points of Interest:	– Driving two automation servers (Xpedition Layout and Excel) from the same script.
Items Used:	– Excel Server
	– OnSelectionChange Xpedition Layout Event
	– VBScript Err Object
Special Notes:	This script works only on the Windows operating system and Excel must be installed. Refer to "Exercise 6: Generating a Component CSV File" on page 374 for details on a cross-platform equivalent to this script.

Before Starting

It is often advantageous to be able to take data from a layout design and to present this data in different ways; for example, in an Excel spreadsheet. It is even more beneficial if interaction occurs between the two applications.

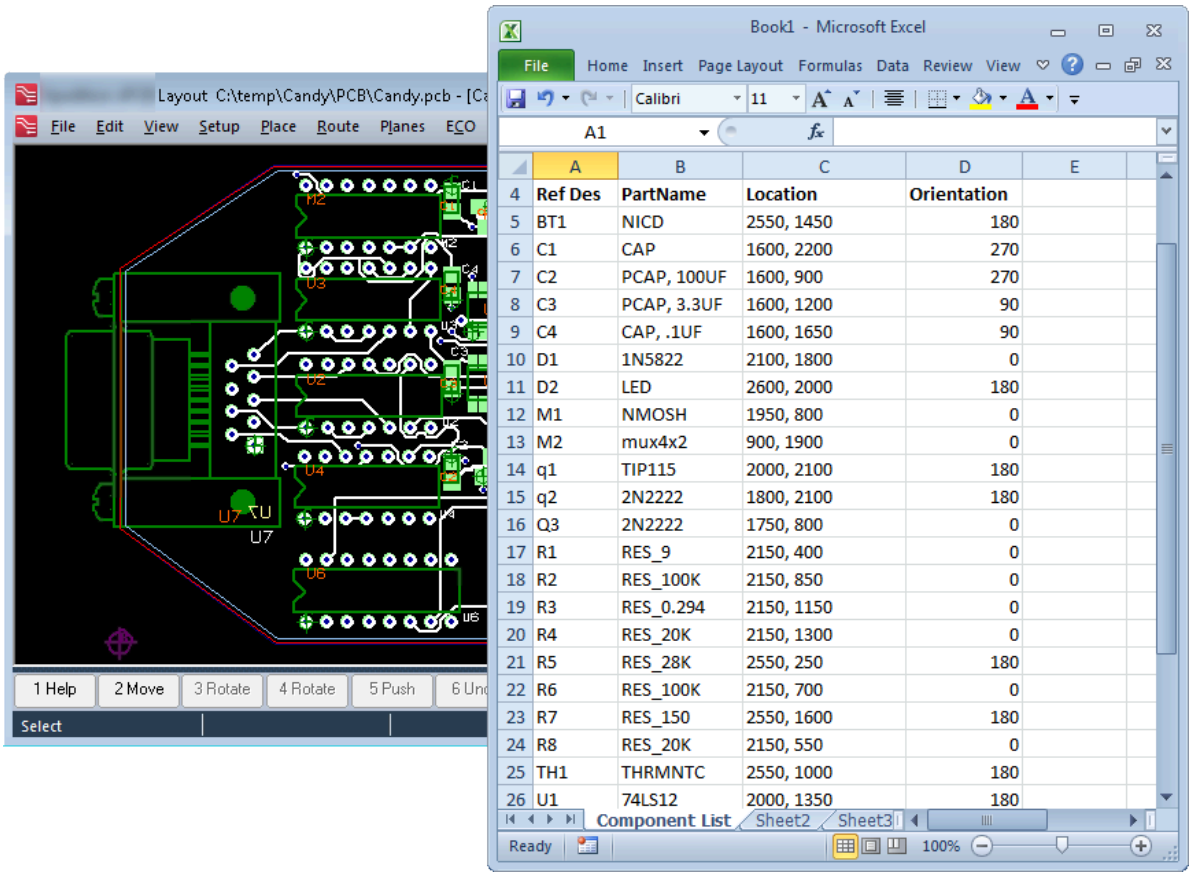
This script generates an Excel spreadsheet, populates it with component information, and then waits for selections to occur in the layout design. When a selection occurs in the layout design, the script automatically locates, selects, and highlights the corresponding items in the Excel spreadsheet.

To understand how the script works, launch Xpedition Layout, open the *Candy.pcb* layout design, and run the script *ExcelCompList.vbs*. The result is similar to Figure 171: Automatically Generates an Excel Spreadsheet.

Observe that the script generates column header information. Then, for each component, the script populates a row of the spreadsheet with that component's reference designator ("Ref Des"), Part Name, X/Y Location, and Orientation (the components are sorted alpha-numerically using their reference designators.

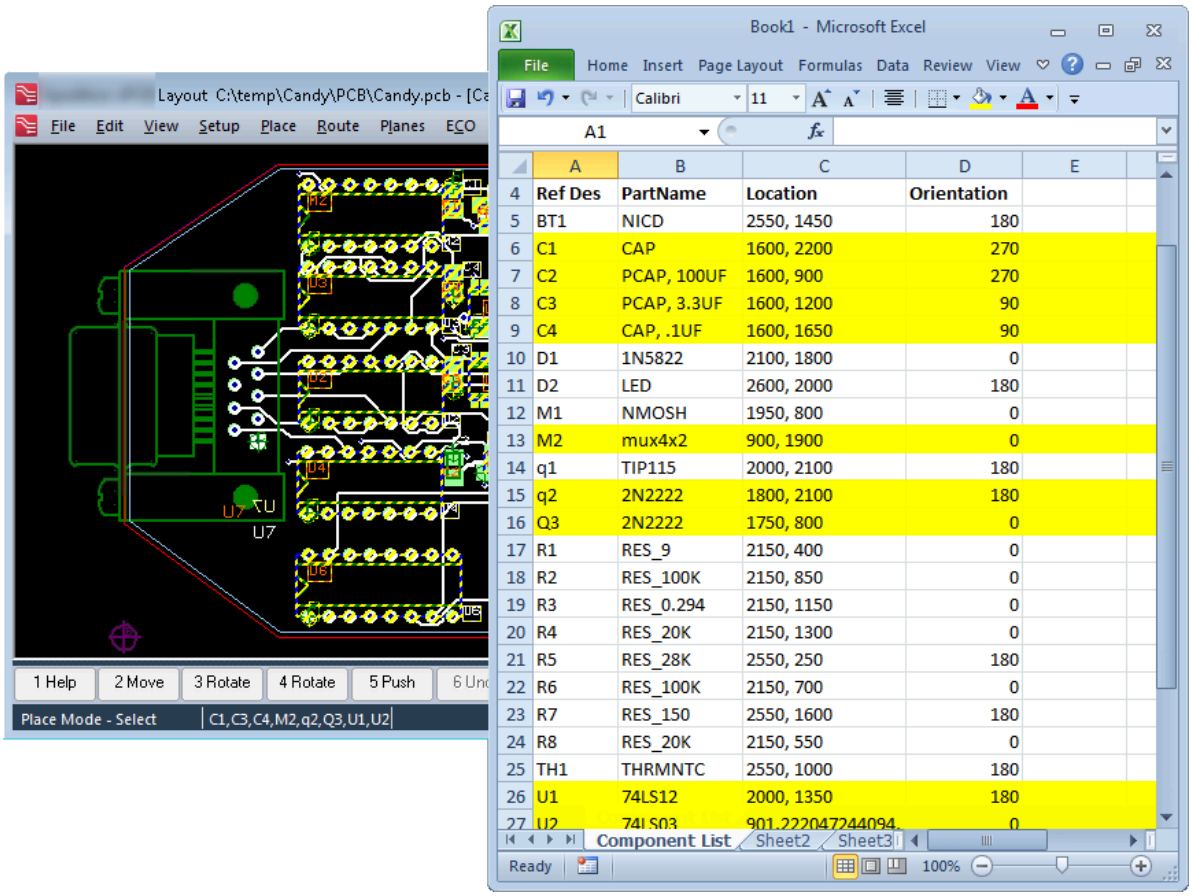
The script also supports cross-probing between Xpedition Layout and Excel. When you select one or more components in the layout design, the corresponding rows in the Excel spreadsheet are located, selected, and highlighted (yellow) as illustrated in Figure 172: Cross-probing Between Xpedition Layout and Excel.

Figure 171. Automatically Generates an Excel Spreadsheet



	A	B	C	D	E
4	Ref Des	PartName	Location	Orientation	
5	BT1	NICD	2550, 1450	180	
6	C1	CAP	1600, 2200	270	
7	C2	PCAP, 100UF	1600, 900	270	
8	C3	PCAP, 3.3UF	1600, 1200	90	
9	C4	CAP, .1UF	1600, 1650	90	
10	D1	1N5822	2100, 1800	0	
11	D2	LED	2600, 2000	180	
12	M1	NMOSH	1950, 800	0	
13	M2	mux4x2	900, 1900	0	
14	q1	TIP115	2000, 2100	180	
15	q2	2N2222	1800, 2100	180	
16	Q3	2N2222	1750, 800	0	
17	R1	RES_9	2150, 400	0	
18	R2	RES_100K	2150, 850	0	
19	R3	RES_0.294	2150, 1150	0	
20	R4	RES_20K	2150, 1300	0	
21	R5	RES_28K	2550, 250	180	
22	R6	RES_100K	2150, 700	0	
23	R7	RES_150	2550, 1600	180	
24	R8	RES_20K	2150, 550	0	
25	TH1	THRMNTC	2550, 1000	180	
26	U1	74LS12	2000, 1350	180	

Figure 172. Cross-probing Between Xpedition Layout and Excel



Script Sections

The key sections of this script follow:

Constant Definitions	Defines the constants the script uses.
Initialization/Setup	Returns the *Application* and *Document* objects, licenses the document, and adds the type libraries. Also performs any additional setup associated with this script.
Event Handler	Triggers when a new selection occurs in the Xpedition Layout layout design, and reflects this selection in the Excel spreadsheet.
Load Excel Subroutine	Defines the main subroutine that populates the Excel spreadsheet with the component information extracted from Xpedition Layout.
Utility Functions and Subroutines	Defines special "workhorse" routines that are focused on this script.
	– SelectComponentsInExcel() Subroutine

	– DefineHeaders() Subroutine
	– AddComponents() Subroutine
	– AddComponent() Subroutine
Helper Functions and Subroutines	Defines generic "helper" routines that other scripts can use.
	– ExcelIsRunning() Function
	– SelectExcelRange() Subroutine
	– RemoveExcelFill() Subroutine
Validate Server	Licenses the document and validates the server. Refer to "Running Your First Script" on page 59 for more details on this function.

Constant Definitions

Lines 10 through 23 define the constants this script uses. Line 10 specifies the sheet name for the Excel spreadsheet.

```
 1  ' This script creates excel and loads it with component info.
 2  ' The script will also supports cross-probing by "listening"
 3  ' for selections in the PCB tool and making those selections
 4  ' in the Excel spreadsheet.
 5  ' Cross-probing can be done from Excel to the PCB tool, but
 6  ' that requires using Excel's VBA.       (Author: Toby Rimes)
 7  Option Explicit
 8
 9  ' Constants
10  Const SHEETNAME_COMP_LIST = "Component List"
```

Lines 12 through 15, assign column labels in the spreadsheet to constant names in the script. In addition to making the script more readable, this makes it easy to rearrange the column order in the future.

```
12  Const REFDES_COL = "A"
13  Const PARTNAME_COL = "B"
14  Const LOCATION_COL = "C"
15  Const ORIENTATION_COL = "D"
```

Similarly, in Lines 17 and 18, the script declares constants that specify that the header information appears on row 4 of the spreadsheet and that the first component information appears on row 5. (The script author's personal preference is to leave some rows blank for future additions such as titles, descriptions, and so forth.)

```
17  Const HEADER_ROW = 4
18  Const FIRST_COMPONENT_ROW = 5
```

It is not possible to add the Excel type library using the *AddTypeLibrary* method. To get around this, on Lines 22 and 23, the script declares the enumerates as constants.

```
20  ' Import type library doesn't work for Excel.
21  ' Define used Excel constants here.
22  Const xlWhole = 1
23  Const xlPart = 2
```

The script only uses *xlwhole*; the script declares the other enumerate for clarity. You can use Excel Help or an Object Browser to discover more information about Excel automation (refer to “Object Browsers” on page 536).

Initialization/Setup

Lines 25 through 55 perform initialization and setup. Lines 25 through 40 add type libraries, return the *Application* and *Document* objects, and license the document (validate the server). The only difference to previous scripts occurs on Line 31, where the script declares a variable to hold an Excel application object.

```
25  ' Add any type libraries to be used.
26  Scripting.AddTypeLibrary("MGCPCB.ExpeditionPCBApplication")
27
28  ' Global variables
29  Dim pcbAppObj                'Application object
30  Dim pcbDocObj                'Document object
31  Dim excelAppObj              'Excel application
32
33  ' Get the application object.
34  Set pcbAppObj = Application
35
36  ' Get the active document
37  Set pcbDocObj = pcbAppObj.ActiveDocument
38
39  ' License the document
40  ValidateServer(pcbDocObj)
```

On Line 43, the script creates an Excel application. On Line 46, the script calls the main LoadExcel() subroutine, which populates the spreadsheet.

```
42  ' Create the Excel application
43  Set excelAppObj = CreateObject("Excel.Application")
44
45  ' Load Excel
46  Call LoadExcel()
```

Once the Excel spreadsheet loads, Line 49 assures it is visible to the end-user.

Tip
Generally, it is better to make applications like this spreadsheet visible after they have been loaded, organized, resized, and so forth, because observing these actions can be "messy" (the exception to this is when you are debugging your script, when you may decide to make the application visible before you start loading it).

```
48  ' Make the excel application visible.
49  excelAppObj.Visible = True
```

Line 52 attaches events to the Xpedition Layout layout design, which receive notification of any selection changes. Line 55 sets the *DontExit* property of the *Scripting* object to *True*. This means this script continues running until the end of the current application session and allows the script to detect and handle events.

```
50
51  ' Attach events to the doc object to get selection changes.
52  Call Scripting.AttachEvents(pcbDocObj, "pcbDocObj")
53
54  ' Hang around to listen to events
55  Scripting.DontExit = True
```

Event Handler

Lines 62 through 68 declare an event handler subroutine that triggers when a new selection occurs in the Xpedition Layout layout design. Line 62 declares the subroutine.

Note:
The *typeEnum* parameter specifies the selection type. Xpedition Layout assigns values to this parameter. The parameter is intended for future use, but must be present in order for the script to work.

```
58  ' Event Handler
59
60  ' Document event fired when there is a selection change
61  ' typeEnum - Unused Enumerate
62  Sub pcbDocObj_OnSelectionChange(typeEnum)
```

Line 63 calls the ExcelIsRunning() Function that assures Excel is still running (the end-user could have exited the Excel application, in which case the script fails). Lines 64 and 65 obtain the collection of objects that is currently selected in the layout design. Line 66 calls the SelectComponentsInExcel() Subroutine, which locates, selects, and highlights the components defined by the collection.

```
63      If ExcelIsRunning(excelAppObj) Then
64          Dim cmpsColl
65          Set cmpsColl = pcbDocObj.Components(epcbSelectSelected)
66          Call SelectComponentsInExcel(cmpsColl)
67      End If
68  End Sub
```

Load Excel Subroutine

Lines 74 through 95 declare the LoadExcel () subroutine, which loads the Excel spreadsheet with component information.

On Line 43, the script creates the Excel application (you can think of this as invoking Excel, but COM refers to this as creating an *Application* object). On Lines 76 and 77, the script uses Excel's *Add* method on its *Workbooks* collection to add a new *Workbook* object (the script creates the new Excel document).

```
73  ' Loads excel with components and header information
74  Sub LoadExcel()
75      ' Create a workbook
76      Dim workbookObj
77      Set workbookObj = excelAppObj.Workbooks.Add
```

By default, a new *Workbook* object contains three *WorkSheet* objects. Lines 80 and 81 acquire the first *Worksheet* object.

```
79      ' Get the first sheet
80      Dim cmpListSheetObj
81      Set cmpListSheetObj = workbookObj.Worksheets.Item(1)
```

Line 84 renames the worksheet by setting its *Name* property to the string constant the script declares on Line 10.

```
83      ' Rename the worksheet.
84      cmpListSheetObj.Name = SHEETNAME_COMP_LIST
```

Line 87 calls the DefineHeaders() Subroutine passing the current *Worksheet* object as a parameter.

```
86      ' Set the header information
87      Call DefineHeaders(cmpListSheetObj)
```

Lines 90 and 91 return a collection of all of the components in the Xpedition Layout layout design.

```
89      ' Get the components
90      Dim cmpColl
91      Set cmpColl = pcbDocObj.Components
```

Line 94 uses the built-in *Sort* method to sort the collection.

Note:
The *Sort* method performs an alpha-numeric sort on the component reference designators. You can create a *Sort()* routine if required, or add the components to the spreadsheet unsorted then use Excel's sort functions.

```
93      ' Sort the component collection
94      Call cmpColl.Sort()
```

Line 97 calls the AddComponents() Subroutine, which adds component information for the *Components* collection to the Excel spreadsheet.

```
96      ' Add the collection
97      Call AddComponents(cmpListSheetObj,
                                        FIRST_COMPONENT_ROW, cmpColl)
98  End Sub
```

Utility Functions and Subroutines

This script section declares utility routines. These routines are similar to the generic Helper Functions and Subroutines, except that they are specific to this script and may not be used elsewhere without changes.

SelectComponentsInExcel() Subroutine

Lines 105 through 131 declare the routine that locates, selects, and highlights component rows in the Excel spreadsheet. The parameter passed into this routine is a collection containing the components to select.

```
103  ' Select the components in the spreadsheet.
104  ' cmpsColl - Component Collection
105  Sub SelectComponentsInExcel(cmpsColl)
```

On Lines 107 and 108, the script returns the *Worksheet* object that contains the main component list from Excel (this is the first sheet because the script added the components earlier).

```
106      ' Get the sheet
107      Dim cmpListSheetObj
108      Set cmpListSheetObj = excelAppObj.Sheets.Item(1)
```

In Excel, a *Range* object represents any combination of individual or multiple cells. Lines 111 and 112 instantiate a *Range* object and initialize it with *Nothing*.

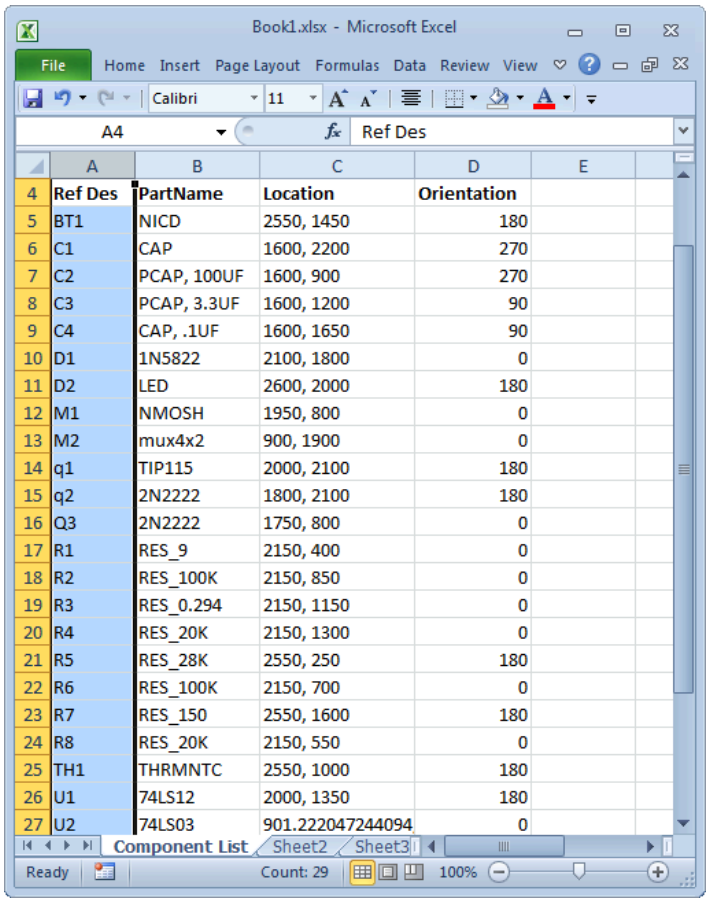
```
110      ' Instantiate a range object to hold multiple cells
111      Dim multiRangeObj
112      Set multiRangeObj = Nothing
```

Lines 117 through 130 iterate through the collection of components to select. Lines 119 and 120 acquire the entire reference designator *Column* as a *Range* object. You do not actually observe this on the screen, but the effect is the same as if you select this column by hand as illustrated in Figure 173: Selecting the Reference Designator Column (Column A).

Using the reference designator range, the script uses the *Find* method and passes the component object name (its reference designator) as the first parameter. Observe the *xlWhole* constant that the script declares on Line 22. This specifies a whole match in the component name, not a partial match. The group of three commas in the parameter list to the *Find* method indicates default values for these parameters.

```
114      ' Loop through all components to build a Range
115      ' of rows for the components in cmpsColl
116      Dim cmpObj
117      For Each cmpObj In cmpsColl
118          ' Find the ref des in the Ref Des column.
               Match the whole name.
119          Dim foundCellObj
120          Set foundCellObj =
           cmpListSheetObj.Columns(REFDES_COL).Find(cmpObj.Name,,,xlWhole)
```

Figure 173. Selecting the Reference Designator Column (Column A)



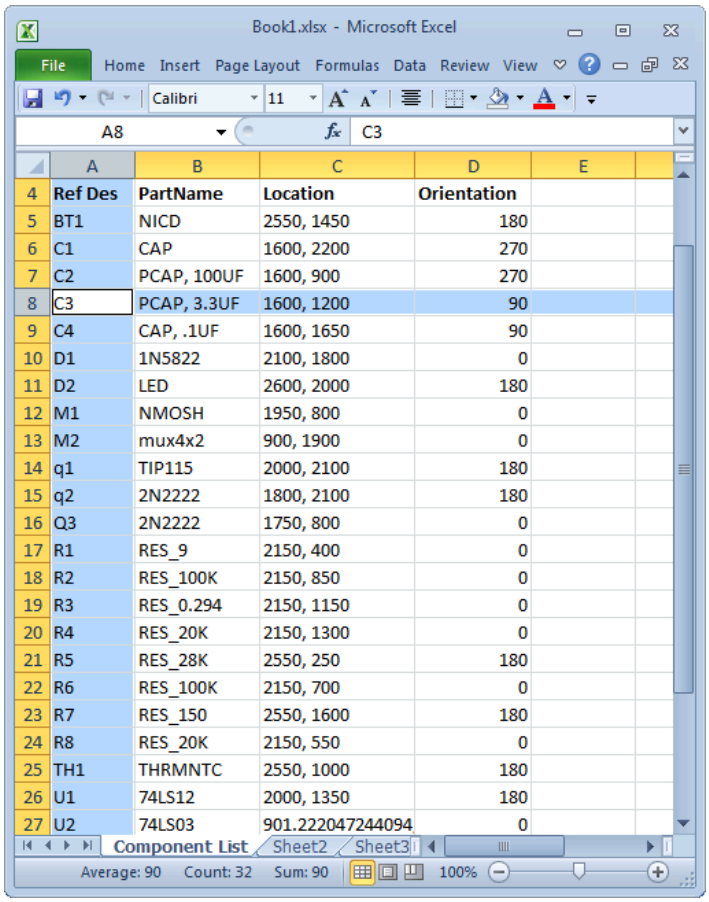
	A	B	C	D	E
4	Ref Des	PartName	Location	Orientation	
5	BT1	NICD	2550, 1450	180	
6	C1	CAP	1600, 2200	270	
7	C2	PCAP, 100UF	1600, 900	270	
8	C3	PCAP, 3.3UF	1600, 1200	90	
9	C4	CAP, .1UF	1600, 1650	90	
10	D1	1N5822	2100, 1800	0	
11	D2	LED	2600, 2000	180	
12	M1	NMOSH	1950, 800	0	
13	M2	mux4x2	900, 1900	0	
14	q1	TIP115	2000, 2100	180	
15	q2	2N2222	1800, 2100	180	
16	Q3	2N2222	1750, 800	0	
17	R1	RES_9	2150, 400	0	
18	R2	RES_100K	2150, 850	0	
19	R3	RES_0.294	2150, 1150	0	
20	R4	RES_20K	2150, 1300	0	
21	R5	RES_28K	2550, 250	180	
22	R6	RES_100K	2150, 700	0	
23	R7	RES_150	2550, 1600	180	
24	R8	RES_20K	2150, 550	0	
25	TH1	THRMNTC	2550, 1000	180	
26	U1	74LS12	2000, 1350	180	
27	U2	74LS03	901.222047244094	0	

On Line 122, the script checks for cells containing the current component's reference designator. You do not actually observe this on the screen, but the effect is the same as illustrated in Figure 174: Refining the Selection With a Specific Row (Row 8).

If the script finds a cell containing the current component's reference designator, then Line 123 checks if the *Range* object is empty (*Nothing*). If the *Range* object is empty, Line 127 loads the range with all of the cells forming the row associated with this component. If the *Range* object already contains something, Lines 124 and 125 run the *Union* method on the existing *Range* object and the new row, which creates a new *Range* object containing all of the required cells up to this point.

```
121         ' If we found something add it to the multi range
122         If Not foundCellObj Is Nothing Then
123             If Not multiRangeObj Is Nothing Then
124                 Set multiRangeObj =
                        excelAppObj.Union(multiRangeObj, _
125                     cmpListSheetObj.Rows(foundCellObj.Row))
126             Else
127                 Set multiRangeObj =
                        cmpListSheetObj.Rows(foundCellObj.Row)
128             End If
129         End If
130     Next
```

Figure 174. Refining the Selection With a Specific Row (Row 8)



	A	B	C	D	E
4	Ref Des	PartName	Location	Orientation	
5	BT1	NICD	2550, 1450	180	
6	C1	CAP	1600, 2200	270	
7	C2	PCAP, 100UF	1600, 900	270	
8	C3	PCAP, 3.3UF	1600, 1200	90	
9	C4	CAP, .1UF	1600, 1650	90	
10	D1	1N5822	2100, 1800	0	
11	D2	LED	2600, 2000	180	
12	M1	NMOSH	1950, 800	0	
13	M2	mux4x2	900, 1900	0	
14	q1	TIP115	2000, 2100	180	
15	q2	2N2222	1800, 2100	180	
16	Q3	2N2222	1750, 800	0	
17	R1	RES_9	2150, 400	0	
18	R2	RES_100K	2150, 850	0	
19	R3	RES_0.294	2150, 1150	0	
20	R4	RES_20K	2150, 1300	0	
21	R5	RES_28K	2550, 250	180	
22	R6	RES_100K	2150, 700	0	
23	R7	RES_150	2550, 1600	180	
24	R8	RES_20K	2150, 550	0	
25	TH1	THRMNTC	2550, 1000	180	
26	U1	74LS12	2000, 1350	180	
27	U2	74LS03	901.222047244094	0	

Finally, after iterating through the components in the collection, Line 133 calls the SelectExcelRange() Subroutine, which selects and highlights all cells in the *Range* object.

```
131
132       ' Select all the rows in the range.
133       Call SelectExcelRange(cmpListSheetObj, multiRangeObj)
134   End Sub
```

DefineHeaders() Subroutine

Lines 138 through 145 declare the routine that adds header text to the *Worksheet* object. The *Worksheet* object passes as a parameter into the routine. Lines 139 through 142 uses the *Range* property associated with the selected *Worksheet* object to add the new header information.

The parameter to the *Range* property is a string containing the column and row information. For example, to select the cell located at the intersection of column A and row 4, pass the *Range* property a string of "A4". Observe the statement on Line 139. In this case, the script concatenates two string constants, REFDES_COL and HEADER_ROW. Line 12 assigns the string "A" to the constant REFDES_COL; Line 17 assigns the string "4" to the constant HEADER_ROW. Thus, Line 139 assigns the string value "Ref Des" to the cell at column A row 4.

```
136  ' Creates header information.
137  ' sheetObj - Excel Worksheet Object
138  Sub DefineHeaders(sheetObj)
139      sheetObj.Range(REFDES_COL & HEADER_ROW).Value = "Ref Des"
140      sheetObj.Range(PARTNAME_COL & HEADER_ROW).Value = "PartName"
141      sheetObj.Range(LOCATION_COL & HEADER_ROW).Value = "Location"
142      sheetObj.Range(ORIENTATION_COL & HEADER_ROW).Value =
                                                          "Orientation"
```

Finally, Line 144 sets the *Bold* property of the *Font* property associated with the header row to *True*. This creates bold header text in the Excel spreadsheets.

```
144      sheetObj.Rows(HEADER_ROW).Font.Bold = True
145  End Sub
```

AddComponents() Subroutine

Lines 151 through 162 declare the routine that coordinates the adding of component information into the Excel spreadsheet. The parameters to this routine are the *Worksheet* object, the row on which to place the first component, and the collection of components to add.

```
147  ' Add the collection of components
148  ' sheetObj - Excel Worksheet Object
149  ' startRowInt - Integer
150  ' cmpColl - Component Collection
151  Sub AddComponents(sheetObj, startRowInt, cmpColl)
```

On Lines 152 and 153, the script instantiates and initializes an integer to keep track of the current row.

```
152      Dim rowInt
153      rowInt = startRowInt
```

Line 154 declares a component object. Lines 155 through 158 iterate through the collection of components. For each component in the collection the script calls the AddComponent() Subroutine, which adds the component details. Parameters to this routine are the *Worksheet* object, the row on which to add this component, and the *Component* object to add.

```
154      Dim cmpObj
155      For Each cmpObj In cmpColl
156          Call AddComponent(sheetObj, rowInt, cmpObj)
157          rowInt = rowInt + 1
158      Next
```

Finally, Line 161 uses the *AutoFit* method to automatically fit all of the columns.

```
160      ' Adjust the cells size to fit the text
161      Call sheetObj.Columns.AutoFit()
162  End Sub
```

AddComponent() Subroutine

Lines 168 through 177 declare the routine that adds the component information into the *Worksheet* object. The parameters passed into this routine are the *Worksheet* object, the row on which to add this component, and the *Component* object to add.

```
164  ' Add a single component to row rowInt
165  ' sheetObj - Excel Worsheet Object
166  ' rowInt - Integer
167  ' cmpObj - Component Object
168  Sub AddComponent(sheetObj, rowInt, cmpObj)
```

Lines 169 through 176 assign component information to appropriate cells in the spreadsheet. Observe that Line 171 checks if the component is placed. If the component is placed, Lines 172 and 173 load the appropriate cells in the spreadsheet with that component's X/Y location and orientation. If the component is not placed, Line 175 loads the "Location" cell with a string of "Unplaced".

```
169      sheetObj.Range(REFDES_COL & rowInt).Value = cmpObj.Name
170      sheetObj.Range(PARTNAME_COL & rowInt).Value = cmpObj.PartName
171      If cmpObj.Placed Then
172          sheetObj.Range(LOCATION_COL & rowInt).Value =
                                  cmpObj.PositionX & ", " & cmpObj.PositionY
173          sheetObj.Range(ORIENTATION_COL & rowInt).Value =
                                                        cmpObj.Orientation
174      Else
175          sheetObj.Range(LOCATION_COL & rowInt).Value = "Unplaced"
176      End If
177  End Sub
```

Helper Functions and Subroutines

This script section declares several generic "helper" routines. These routines can be used in other scripts and are not focused on this script exclusively.

ExcelIsRunning() Function

There is a chance that Excel can be closed by the end-user prior to their attempting to make a selection. If there is no test for this, the script can terminate unexpectedly. Generally, this type of check is addressed by “listening” for the *Close* event from the Excel application. However, Excel's interface does not allow events to be caught by scripts (this is discussed in more detail later in this tutorial).

In order to get around this, the script traps any errors that occur. If there are no errors, Excel is running; otherwise, Excel is closed.

Lines 184 through 208 declare a helper function that checks if Excel is still running. The parameter passed into this function is the Excel *Application* object.

```
182  ' Returns true if Excel is still running false otherwise
183  ' appObj - Excel Application Object
184  Function ExcelIsRunning(appObj)
```

Line 186 assumes that Excel is running and initializes the return value from this function to *True*.

```
185      ' Initialize return value
186      ExcelIsRunning = True
```

Line 189 checks if the variable holding the Excel *Application* object is set to *Nothing* (refer to Lines 204 through 207 to discover why this might be the case). If *Nothing*, Line 190 sets the return value to *False* and Line 191 exits this function.

```
188      ' If the variable is nothing return false
189      If appObj Is Nothing Then
190          ExcelIsRunning = False
191          Exit Function
192      End If
```

Line 196 uses the *On Error Resume Next* statement to prevent errors from terminating the script (refer to the *Error Handling* topic in "VBScript Primer" on page 27). Line 197 clears the *Err* object so as not to display any previous errors that may have occurred.

```
194      ' Trying to call a method on Excel. If there is an
195      ' exception we will assume Excel is shut down.
196      On Error Resume Next
197      Err.Clear
```

Line 200 declares a dummy object and Line 201calls the Excel interface. If an error occurs with the call, Excel is closed. (The *Sheets* property is chosen in this script, but almost any call is sufficient).

```
199      ' Make a call that would cause an exception
200      Dim sheetsObj
201      Set sheetsObj = appObj.Sheets
```

Line 204 checks if an error occurs. If so, it sets the Excel *Application* object to *Nothing*, sets the return value to *False* and exits the function. Setting the Excel *Application* object to *Nothing* allows the script to perform the test on Line 189. For example, if the end-user tries to select something else, this routine is called again.

```
202
203      ' Check the error value.
204      If Err Then
205          Set appObj = Nothing
206          ExcelIsRunning = False
207      End If
208  End Function
```

Assuming Excel is still active and there is no error, then the script returns to the mode where any other errors cause it to terminate. In this case, however, there is no requirement for the *On Error GoTo 0* statement (this was detailed in the *Error Handling* topic in "VBScript Primer" on page 27). This is because the scope of the *On Error Resume Next* statement is limited to the function or subroutine in which it is called.

SelectExcelRange() Subroutine

Lines 213 through 224 declare the routine that selects one or more cells in an Excel spreadsheet. The cells selected are defined by an Excel *Range* object. The parameters to this routine are the *Worksheet* object and the *Range* object.

```
210  ' Selects a range of objects and colors the range yellow
211  ' sheetObj - Excel Worksheet object
212  ' rangeObj - Excel Range Object
213  Sub SelectExcelRange(sheetObj, rangeObj)
```

Line 215 calls the RemoveExcelFill() Subroutine and passes the *Worksheet* object. This function sets all of the cells on the specified worksheet to have a background color of white.

```
214      ' Remove the yellow
215      Call RemoveExcelFill(sheetObj)
```

Line 216 checks if the *Range* object is valid. If valid, Line 218 sets the *ColorIndex* property of the *Interior* property of the *Range* object to a value of 6, which is yellow (refer to the notes at the end of this routine). Then, Line 219 uses the *Select* method to select all of the cells defined by the *Range* object.

If the *Range* object is not valid, Line 222 causes an unselection of the currently selected cells by forcing a selection on cell "A1".

```
216      If Not rangeObj Is Nothing Then
217          ' Set the interior to yellow and select
218          rangeObj.Interior.ColorIndex = 6
219          Call rangeObj.Select()
220      Else
221          ' Cause an unselection by selecting first cell
222          Call sheetObj.Range("A1").Select()
223      End If
224  End Sub
```

Tip
The script sets the *ColorIndex* property to 6, which is yellow. The color values in Excel are undoubtedly documented somewhere on the planet, but the authors of this tutorial could not find this documentation anywhere. The way to work around this is to use the Excel Graphical User Interface (GUI) to open a Worksheet and to set a cell to the required color (yellow). Then create a small script to access the cell and to display the value of its *ColorIntex* property in a *MsgBox()*. It is helpful to perform this task for a range of colors and to then create your own set of enumerates for use in future scripts. (Tricks like this can also be used with other automation interfaces.)

RemoveExcelFill() Subroutine

Lines 228 through 231 declare the routine that turns any yellow cells to a white background. The value –4142 (negative 4142) is white.

```
226  ' Sets the fill to white for all the cells on a sheet
227  ' sheetobj - Excel Sheet Object
228  Sub RemoveExcelFill(sheetObj)
229      ' Set all to white
230      sheetObj.Cells.Interior.ColorIndex = -4142
231  End Sub
```

Creating and Testing the Script

1. Use the scripting editor of your choice to enter the script described (including the license server function) and save the script as *ExcelCompList.vbs*.

2. Close any instantiations of Xpedition Layout that are currently running, and then launch a new instantiation of this application.

3. Open the *Candy.pcb* design.

4. Run the *ExcelCompList.vbs* script and observe the populated Excel spreadsheet appear on your screen.

5. Go into **Place** mode, select or multi-select components, and observe the corresponding items being located, selected, and highlighted in the Excel spreadsheet.

6. Close the Excel spreadsheet and make a new selection. Observe that the script does not fail or crash because of the test included in the event handler on Line 63.

Enhancing the Script

There are a number of ways to enhance this script. Several suggestions follow (practice your scripting skills by implementing these suggestions):

- Add functionality that displays a "Progress Meter" in Xpedition Layout's status bar to reflect the status of the Excel load.

  Depending on the size of the design, when you first run this script it can take a few seconds for the script to create and load the Excel spreadsheet. Currently, the end-user has no indication as to what is going on during this time.

- Add functionality that provides the end-user with the ability to choose the highlight color from a palette presented as a pop-up form/dialog launched from an accelerator key.

- Add functionality that allows the end-user to associate different highlight colors with different components (resistors = red, capacitors = blue, chips = green, and so forth).

- Experiment with making the Excel spreadsheet header text a slightly bigger font and/or changing its color.

- Modify the ExcelIsRunning() Function to inform the end-user that there is a problem when Excel is closed and the end-user attempts to make a selection in the layout design.

- Modify the script to perform cross-probing from Excel back to Xpedition Layout. (The script only supports cross-probing from Xpedition Layout to Excel.)

  You implement cross-probing in Excel as a macro (a large portion of the code shown here works in an Excel macro). Using the Excel Macro Editor is beyond the scope of this tutorial, but a significant amount of documentation on this topic is available both online and in the Excel Macro Editor tool.

Exercise 6: Generating a Component CSV File

This script reads the component information from an Xpedition Layout design and writes it to a text file in Comma Separated Value (CSV) format. Depending on the operating system, the script then opens the appropriate Microsoft or OpenOffice viewer and uses it to display the CSV file.

Approach:	To pull component information out of a layout design and to use the File System Object Server to create a text file and write the component information to this file.
Points of Interest:	– Writing cross-platform scripts
	– Writing to a text file
Items Used:	– File System Object Server
	– ViewLogic.Exec Server
	– Scripting.IsUnix Method

Before Starting

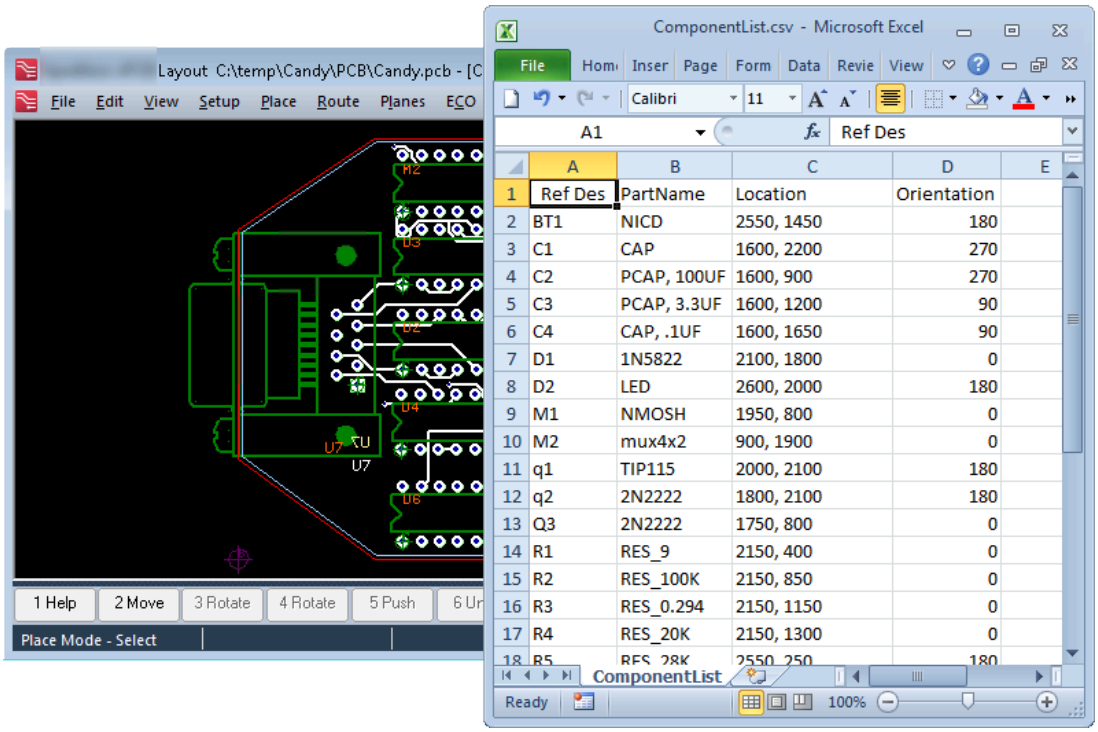
In Exercise 5: Integration with Excel, the script passes data directly from the layout design into a running Excel document. That script runs only on the Windows operating system. This section presents a similar script, *ComponentCVS.vbs*, which works across multiple platforms (Windows, Unix, Linux, etc.)

The script first outputs information to a text file in Comma Separated Value (CSV) format. The CSV file can be used by any application that supports CSV files on any platform.

Once the script generates the CSV file, it checks to determine the current operating system, launches an appropriate spreadsheet viewer for that platform, and uses this viewer to display the contents of the CSV file.

To understand how the script works, launch Xpedition Layout, open the *Candy.pcb* layout design, and run the script *ComponentCVS.vbs.* The result is similar to Figure 175: View the Contents of the CSV File.

Figure 175. View the Contents of the CSV File



	PartName	Location	Orientation
BT1	NICD	2550, 1450	180
C1	CAP	1600, 2200	270
C2	PCAP, 100UF	1600, 900	270
C3	PCAP, 3.3UF	1600, 1200	90
C4	CAP, .1UF	1600, 1650	90
D1	1N5822	2100, 1800	0
D2	LED	2600, 2000	180
M1	NMOSH	1950, 800	0
M2	mux4x2	900, 1900	0
q1	TIP115	2000, 2100	180
q2	2N2222	1800, 2100	180
Q3	2N2222	1750, 800	0
R1	RES_9	2150, 400	0
R2	RES_100K	2150, 850	0
R3	RES_0.294	2150, 1150	0
R4	RES_20K	2150, 1300	0
R5	RES_28K	2550, 250	180

Script Sections

The key sections of this script follow:

Constant Definitions	Defines the constants the script uses.
Initialization/Setup	Returns the *Application* and *Document* objects, licenses the document, and adds the type libraries. Also performs any additional setup associated with this script.
Create CSV File Subroutine	Creates a text file and loads it with component information in CSV format.
Utility Functions and Subroutines	Defines special "workhorse" routines that are focused on this script.
	– WriteHeaders() Subroutine
	– WriteComponents() Subroutine
	– WriteComponent() Subroutine
Helper Functions and Subroutines	Defines generic "helper" routines that other scripts can use.
	– ViewCSV() Function

Validate Server	Licenses the document and validates the server. Refer to "Running Your First Script" on page 59 for more details on this function.

Constant Definitions

The only constant this script uses, occurs on Line 8, which is the hard-coded name of the CSV file. This file is created in the Log Files folder of the layout design.

```
1  ' This script creates a CSV and loads it with component
2  ' info. It also opens the appropriate Microsoft or OpenOffice
3  ' viewer to display the CSV file.         (Author: Toby Rimes)
4
5  Option Explicit
6
7  ' Constatnts
8  Const FILE_NAME = "ComponentList.csv"
```

Initialization/Setup

Lines 11 through 34 perform the initialization and setup tasks. Lines 11 through 24 add the type libraries the script requires, return the *Application* and *Document* objects, and license the document by calling the *ValidateServer()* function.

```
10  ' Add any type libraries used.
11  Scripting.AddTypeLibrary("MGCPCB.ExpeditionPCBApplication")
12
13  ' Global variables
14  Dim pcbAppObj                 'Application object
15  Dim pcbDocObj                 'Document object
16
17  ' Get the application object.
18  Set pcbAppObj = Application
19
20  ' Get the active document
21  Set pcbDocObj = pcbAppObj.ActiveDocument
22
23  ' License the document
24  ValidateServer(pcbDocObj)
```

On Lines 27 and 28, the script returns the path of the design document and appends the path with the *LogFiles* folder name and the *ComponentList.csv* file name.

```
26  ' Create the csv file name
27  Dim fileNameStr
28  fileNameStr = pcbDocObj.Path & "LogFiles\" & FILE_NAME
```

Line 31 calls the main *CreateCSV()* subroutine, which creates and populates the CSV file. The filename passes to this function. Then, Line 34 calls the ViewCSV() Function, which displays the contents of the CSV file using an appropriate spreadsheet viewer.

```
30  ' Create the csv file.
31  Call CreateCSV(fileNameStr)
32
33  ' Show the csv file in a viewer
34  Call ViewCSV(fileNameStr)
```

Create CSV File Subroutine

Lines 41 through 66 declare the *CreateCSV ()* function, which creates and populates a text file in CSV format. Lines 43 and 44 create a *FileSystemObject* object (refer to "Creating Input/Output (I/O) Files" on page 256 for more details on reading and writing files).

```
39  ' Create a CSV file and load it with information
40  ' fileNameStr - String
41  Sub CreateCSV(fileNameStr)
42      ' Create the file system object
43      Dim fileSysObj
44      Set fileSysObj = CreateObject("Scripting.FileSystemObject")
```

Lines 47 through 49 create a new text file named *ComponentList.csv* by using the *CreateTextFile* method. As part of this process it obtains a *TextStream* object to output the text file.

```
46      ' Create a file and get its text stream
47      Dim overWriteInt: overWriteInt = 1
48      Dim txtStreamObj
49      Set txtStreamObj = fileSysObj.CreateTextFile(fileNameStr,
                                                     overWriteInt)
```

Line 52 calls the WriteHeaders() Subroutine, which writes the column headers to the output file.

```
51      ' Set the header information
52      Call WriteHeaders(txtStreamObj)
```

Lines 55 and 56 return a collection containing all of the components in the layout design. Line 59 uses the built-in *Sort* method to sort the collection. This method performs an alpha-numeric sort on the component reference designators. You can create a *Sort()* routine if required.

```
54      ' Get the components
55      Dim placedCmpColl
56      Set placedCmpColl = pcbDocObj.Components
57
58      ' Sort the component collection
59      Call placedCmpColl.Sort()
```

Line 62 calls the WriteComponents() Subroutine, which writes the component information into the text file. Lines 65 and 66 close the text file and exit this subroutine.

```
61      ' Write the collection
62      Call WriteComponents(txtStreamObj, placedCmpColl)
63
64      ' Close the file.
65      Call txtStreamObj.Close()
66  End Sub
```

Utility Functions and Subroutines

This script section declares utility routines. These routines are similar to the generic Helper Functions and Subroutines, except that they are specific to this script and may not be used elsewhere without changes.

WriteHeaders() Subroutine

This simple routine accepts a *TextStream* object as a parameter and writes a string to the text file. The string consists of four comma-delimited header titles, each of which appears at the top of a column of component information. The *Writeline* statement automatically appends a newline character (or characters, depending on the current operating system) to the end of the string.

```
71  ' Creates header information.
72  ' txtStreamObj - FSO TextStream Object
73  Sub WriteHeaders(txtStreamObj)
74      Call txtStreamObj.WriteLine("Ref Des,
                                        PartName,Location,Orientation")
75  End Sub
```

WriteComponents() Subroutine

This routine accepts a *TextStream* object and a component collection as parameters. The routine cycles through all components in the collection and for each component, calls the WriteComponent() Subroutine.

```
77  ' Add the collection of components
78  ' txtStreamObj - FSO TextStream Object
79  ' cmpColl - Component Collection
80  Sub WriteComponents(txtStreamObj, cmpColl)
81      Dim cmpObj
82      For Each cmpObj In cmpColl
83          Call WriteComponent(txtStreamObj, cmpObj)
84      Next
85  End Sub
```

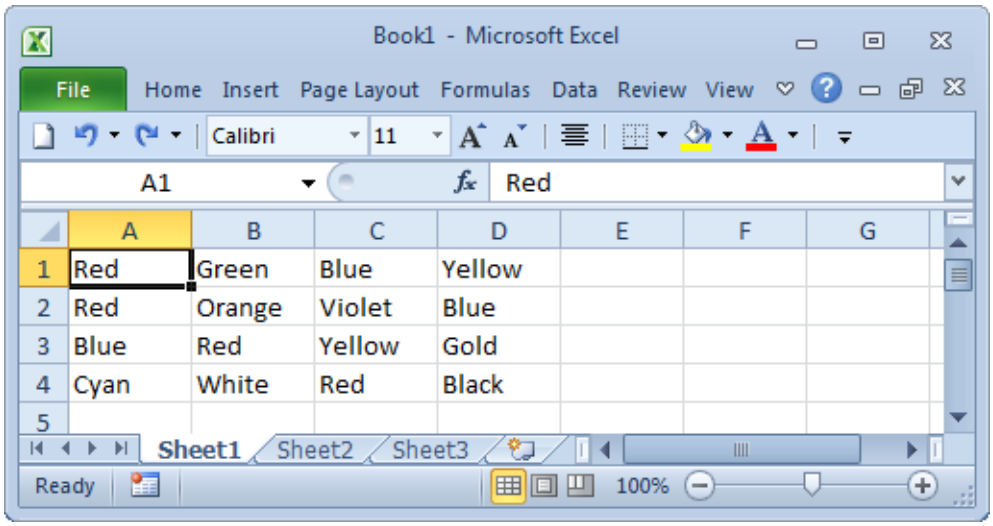
WriteComponent() Subroutine

Before describing this routine, you should understand CSV file delimiters. For this discussion, assume that you want to generate a CSV file that populates a spreadsheet containing four columns, and that each of these columns contains the name of a single color. For example, the CSV file looks like the following:

```
Red, Green, Blue, Yellow
Red, Orange, Violet, Blue
Blue, Red, Yellow, Gold
Cyan, White, Red, Black
```

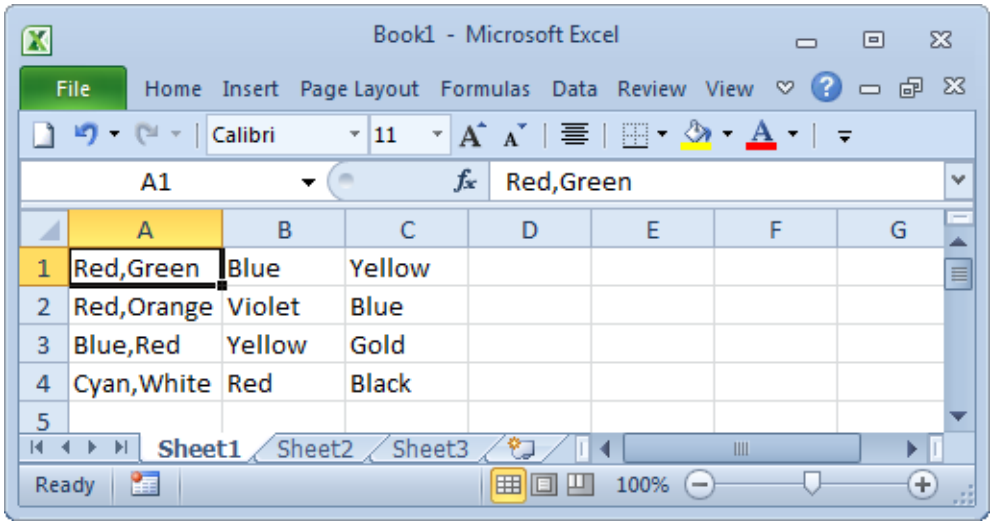
In CSV files the comma following each color, separates or “delimits” each column entry; and the invisible newline character following each line of text, delimits each row. If this CSV file is loaded into a spreadsheet, it appears similar to Figure 176: Spreadsheet Cells Containing Individual Values .

Figure 176. Spreadsheet Cells Containing Individual Values



Suppose that the spreadsheet should appear as illustrated in Figure 177: Spreadsheet Cells Containing Comma Separated Values , with two comma-separated colors, in the first column.

Figure 177. Spreadsheet Cells Containing Comma Separated Values

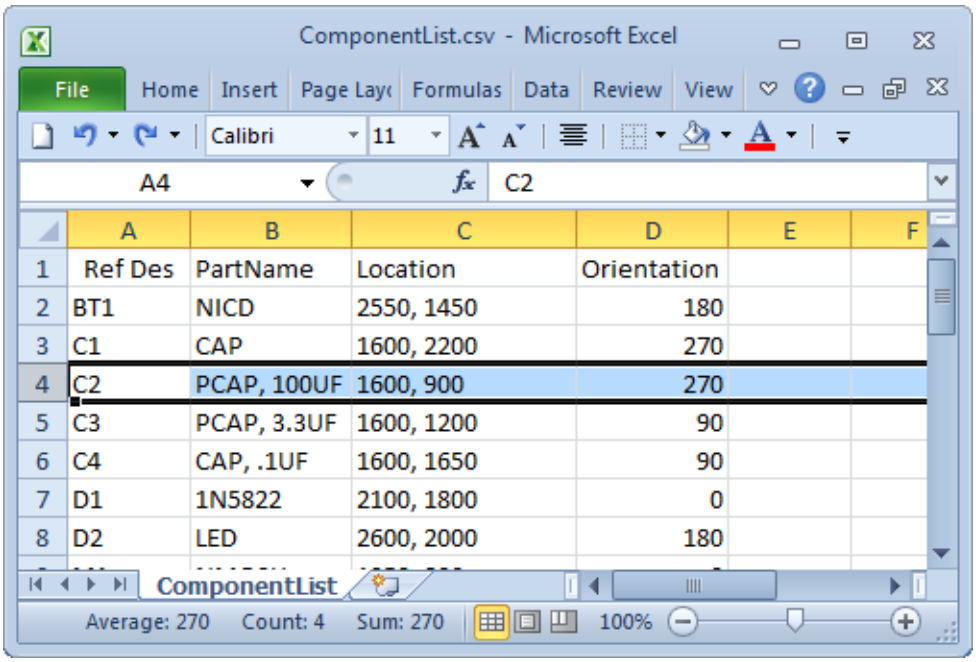


This is done by creating a CSV file that encloses the comma-separated values with double-quotes (“) as:

```
"Red, Green", Blue, Yellow
"Red, Orange", Violet, Blue
"Blue, Red", Yellow, Gold
"Cyan, White", Red, Black
```

In the *Candy.pcb* design, the items forming the first three columns of the CSV file (representing the reference designator, part name and component location), can each contain comma-separated values. For example, consider the **C2** component in the spreadsheet associated with the design shown in Figure 178: Cells in Component Spreadsheet Containing Multiple Values .

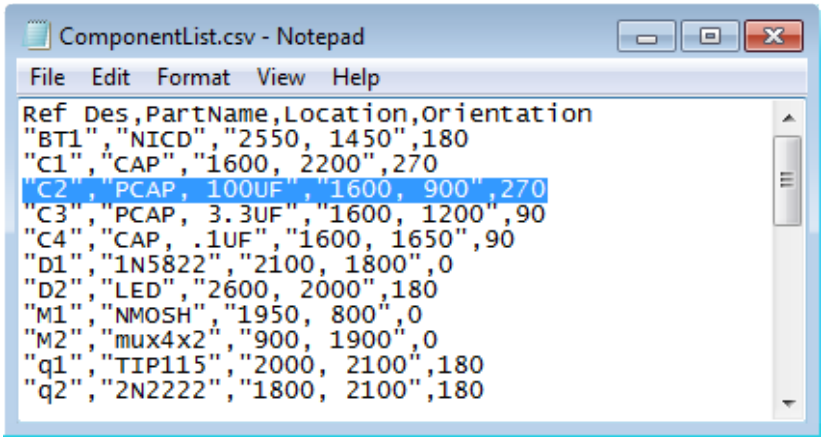
Figure 178. Cells in Component Spreadsheet Containing Multiple Values



	A	B	C	D	E	F
1	Ref Des	PartName	Location	Orientation		
2	BT1	NICD	2550, 1450	180		
3	C1	CAP	1600, 2200	270		
4	C2	PCAP, 100UF	1600, 900	270		
5	C3	PCAP, 3.3UF	1600, 1200	90		
6	C4	CAP, .1UF	1600, 1650	90		
7	D1	1N5822	2100, 1800	0		
8	D2	LED	2600, 2000	180		

The **Ref Des**, **PartName**, and **Location** columns can each contain comma-separated values (only the **PartName**, and **Location** columns in this example). This requires that the script enclose items in these three columns using double-quotes(“). The CSV file that populates the spreadsheet is illustrated in Figure 179: CSV File Format .

Figure 179. CSV File Format



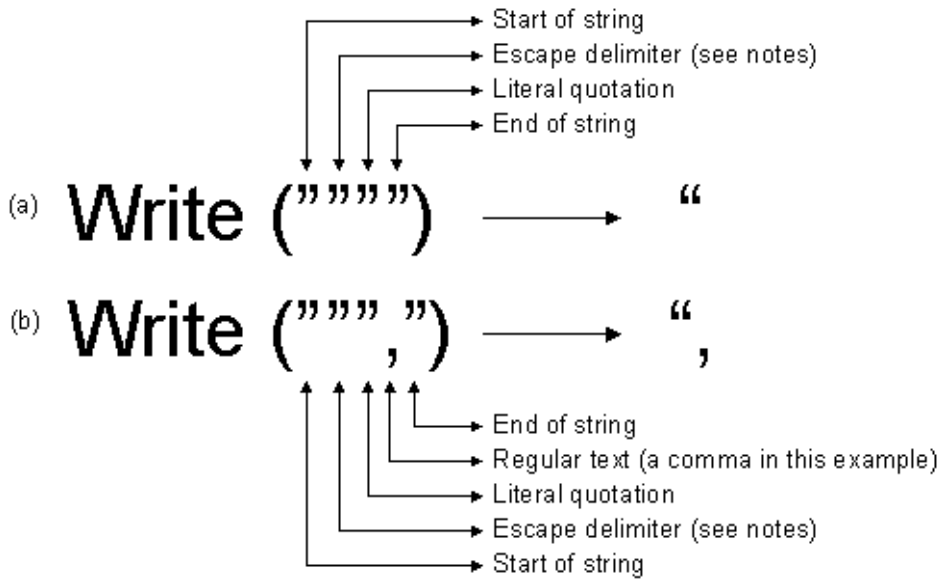
Note:
The functionality in this WriteComponent() Subroutine could be embedded in the WriteComponents() Subroutine. However, when defined in the WriteComponent() Subroutine, it is easier to read and understand.

The WriteComponent() Subroutine accepts a *TextStream* object and a *Component* object as parameters. Line 91 outputs the component's name (its reference designator); Line 92 outputs its part name. Line 93 checks if the component is placed. If the component is placed, then Lines 94 and 95 output its X/Y location and Line 96 outputs its orientation. Alternatively, if the component is not placed, then Line 98 outputs the string "Unplaced". Finally, Line 100 uses a *WriteLine* statement with an empty string to append a newline character (or characters, depending on the operating system) to the end of this row in the CSV file.

```
 87  ' Add a single component
 88  ' txtStreamObj - FSO TextStream Object
 89  ' cmpObj - Component Object
 90  Sub WriteComponent(txtStreamObj, cmpObj)
 91      Call txtStreamObj.Write("""" & cmpObj.Name & """,")
 92      Call txtStreamObj.Write("""" & cmpObj.PartName & """,")
 93      If cmpObj.Placed Then
 94          Call txtStreamObj.Write("""" & cmpObj.PositionX _
 95               & ", " & cmpObj.PositionY & """,")
 96          Call txtStreamObj.Write(cmpObj.Orientation)
 97      Else
 98          Call txtStreamObj.Write("Unplaced")
 99      End If
100      Call txtStreamObj.WriteLine("")
101  End Sub
```

Observe the groups of three and four double-quotes (""" and """") that occur on Lines 91, 92, 94, and 95. Figure 180: Triple and Quadruple Double-quotes (a) illustrates the four double-quotes usage.

Figure 180. Triple and Quadruple Double-quotes



The first quote starts the string; the second quote acts as an escape delimiter, which informs the system that the following character is not intended to close the string, but instead should be treated as a literal character to output; the third quote is the quotation character to output; and the fourth quote closes the string.

Similarly, in Figure 180: Triple and Quadruple Double-quotes (b) the first quote starts of the string; the second quote acts as an escape delimiter; the third quote is a quotation character to output. Then, include any regular text (this example includes a comma), and finally the fourth quote closes the string.

Helper Functions and Subroutines

This script section declares several generic "helper" routines. These routines can be used in other scripts and are not focused on this script exclusively.

ViewCSV() Function

Lines 108 through 120 declare the function that takes the content of the CSV file and displays it using whichever spreadsheet application is appropriate for the current operating system.

Lines 110 and 111 create a *ViewLogic.Exec* object, which runs system commands (refer to "Running Executables" on page 268).

```
106  ' Displaye the CSV file in the appropriate viewer
107  ' fileNameStr - String
108  Function ViewCSV(fileNameStr)
109      ' Create an Exec object
110      Dim execObj
111      Set execObj = CreateObject("Viewlogic.Exec")
```

On Line 113, the script checks what platform the script is running on. The *Scripting.IsUnix* method returns *True* if the script is running on Unix or Linux. If this is the case, then Line 115 uses the *Run* method to launch the OpenOffice Calc spreadsheet application (this executable is called *oocalc*) and passes it the name of the CSV file. Alternatively, if the *IsUnix* method returns False, the script assumes that it is

running under Windows. In this case, Line 118 uses the *Run* method to launch the Excel application (this executable is called *Excel.exe*) and passes it the name of the CSV file.

```
113       If Scripting.IsUnix() Then
114           ' Open the csv file in OpenOffice Calc
115           Call execObj.Run("oocalc " & fileNameStr)
116       Else
117           ' Open the csv file in notepad
118            Call execObj.Run("Excel.exe " & fileNameStr)
119       End If
120  End Function
```

On Windows, it is optional whether or not to specify the *.exe* extension (there is no *.exe* extension on UNIX or Linux).

Note:
This script requires that the appropriate spreadsheet application executable is installed on the platform and can be found using the *PATH* variable.

Creating and Testing the Script

1. Use the scripting editor of your choice to enter the script described (including the license server function) and save the script as *ComponentCVS.vbs*.
2. Close any instantiations of Xpedition Layout that are currently running, and then launch a new instantiation of this application.
3. Open the *Candy.pcb* design.
4. Run the *ComponentCVS.vbs* script and observe the spreadsheet appear as illustrated in Figure 175: View the Contents of the CSV File.
5. Use a text editor of your choice to display the contents of the *ComponentList.csv* file generated by the script and confirm that it looks like the file shown in Figure 179: CSV File Format .

Enhancing the Script

There are a number of ways to enhance this script. Several suggestions follow (practice your scripting skills by implementing these suggestions):

- Modify the data and format to create a bill of materials (BOM) output in CSV format.
- Add error checking to determine if the required application was not found and notify the end-user accordingly.

Exercise 7: Finalizing a Design

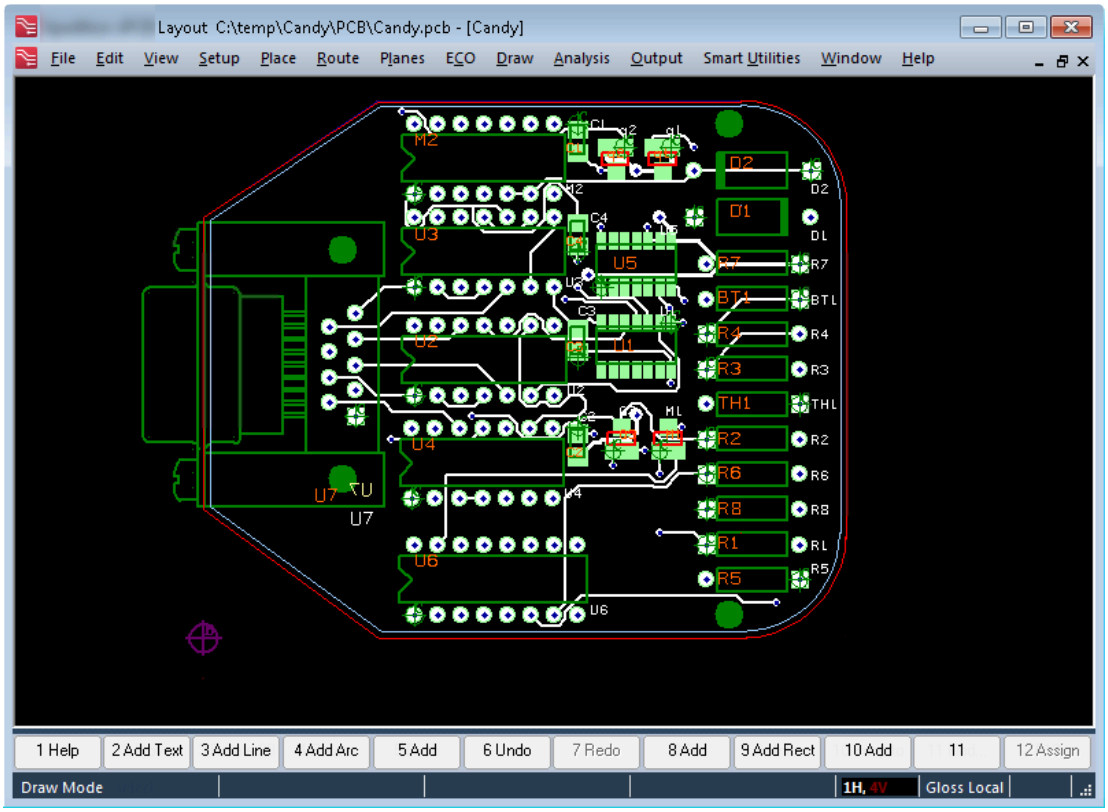
This script finalizes a layout design by automatically adding board dimensions and a company logo to the design.

Approach:	To use the automation interface to add dimension graphics that reflect the horizontal and vertical extents of the board. Also to place a pre-defined drawing cell containing a company logo in a location relative to the horizontal and vertical extents of the board.
Points of Interest:	– Adding Dimensions
	– Placing Drawing Cells
	– Use of Advanced VBScript Structures (Class objects)
	– Working with Points Arrays
Items Used:	– PutPointToPointDimension Method
	– PutComponent Method
	– Extrema Object

Before Starting

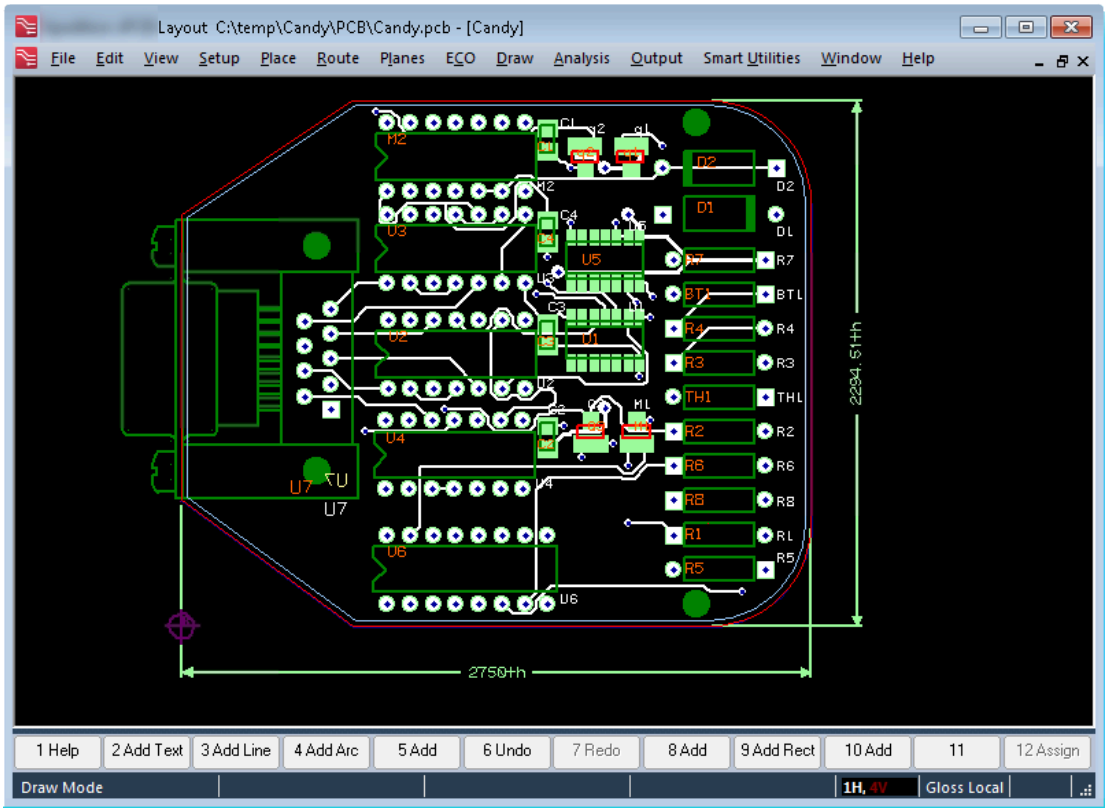
After a board design is complete and prior to sending it to manufacturing – there may be some common documentation items to add. This script demonstrates how to add the board's horizontal and vertical dimensions and a company logo. Consider the familiar *Candy.pcb* design as illustrated in Figure 181: Candy.pcb Design.

Figure 181. Candy.pcb Design



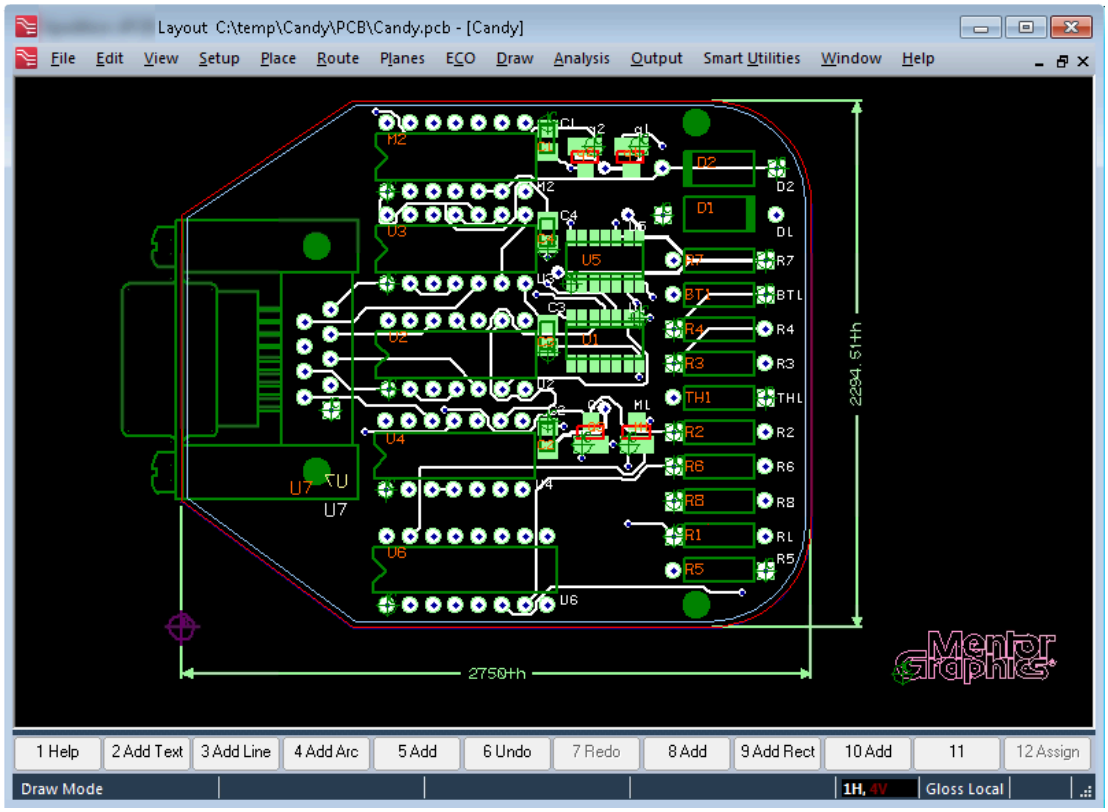
Choose **Draw > Dimension > Between Two Elements** to add the horizontal and vertical board dimensions by hand as Figure 182: Adding Vertical and Horizontal Board Dimensions illustrates.

Figure 182. Adding Vertical and Horizontal Board Dimensions



Next, choose **Place > Place Parts and Cells** to select and place a drawing cell containing the company logo as Figure 183: Adding Company Logo illustrates. The *Candy.pdb* design includes a cell named *Logo*.

Figure 183. Adding Company Logo



The *FinalizeDesign.vbs* script performs the previous to operations for the end-user. The main reasons for creating and using such a script are:

1. To assure that all required finishing tasks are run on every design.

2. To assure that such tasks are performed consistently on all designs.

Script Sections

The key sections of the *FinalizeDesign.vbs* script follow:

Constant Definitions	Defines the constants the script uses.
Initialization/Setup	Returns the *Application* and *Document* objects, licenses the document, and adds the type libraries. Also performs any additional setup associated with this script.
Main Functions and Subroutines	Defines the main functions and subroutines the script uses.
Add Board Dimensions Subroutine	– AddBoardDimensions()
Add Company Logo Subroutine	– AddCompanyLogo()

Utility Functions and Subroutines	Defines special "workhorse" routines that are focused on this script.
	– AddVertDimExtrema() Subroutine
	– AddHorizDimExtrema() Subroutine
	– FindExtremaPoint() Function
	– BetterMatch() Function
	– BetterValueMatch() Function
Helper Functions and Structures	Defines generic "helper" routines that other scripts can use.
	– CreateUserLayer() Function
	– Class Point Structure
Validate Server	Licenses the document and validates the server. Refer to "Running Your First Script" on page 59 for more details on this function.

Constant Definitions

The constant definitions on Lines 11 through 14 define the placement locations.

```
 1  ' This script performs actions that might be taken to finish
 2  ' out a design.  It places board dimensions and adds a company
 3  ' logo to the design.
 4  '                                          (Author: Toby Rimes)
 5
 6  Option Explicit
 7
 8  ' Constants
 9
10  ' Location constants
11  Const LOC_TOP = 1
12  Const LOC_BOTTOM = 2
13  Const LOC_RIGHT = 3
14  Const LOC_LEFT = 4
```

The script uses the definitions on Lines 17 and 18 for indexing into the points arrays; these constants make the script more readable.

```
16  ' PointsArray index constants
17  Const X_NDX = 0
18  Const Y_NDX = 1
```

Line 21 defines the name of the user layer upon which to add the dimensional graphics and text.

```
20  ' Dimension user layer
21  Const DIMENSION_LAYER_NAME = "Documentation"
```

Lines 24 through 26 specify the amounts by which to offset the dimensions and logo from the board outline. You can split the single dimension offset on Line 24 into separate vertical and horizontal offsets.

```
23  ' Dimenstion and logo offset value
24  Const DIMENSION_OFFSET = 200
25  Const LOGO_VERT_OFFSET = 200
26  Const LOGO_HORIZ_OFFSET = 400
```

Lines 29 and 30 hard-code the name of the logo cell and the name of the user layer on which to display this cell. These names are not required to be identical.

```
28  ' Logo cell and userlayer name
29  Const LOGO_CELL_NAME = "Logo"
30  Const LOGO_LAYER_NAME = "Logo"
```

Initialization/Setup

Lines 33 through 47 perform the standard initialization tasks of adding type libraries, defining the units to use throughout the script, acquiring the *Application* and *Document* objects, and licensing the document (validating the server).

```
32  ' Add any type libraries used.
33  Scripting.AddTypeLibrary("MGCPCB.ExpeditionPCBApplication")
34
35  ' Global variables
36  Dim pcbAppObj                'Application object
37  Dim pcbDocObj                'Document object
38  Dim unitsEnum : unitsEnum = epcbUnitMils ' Units used
39
40  ' Get the application object.
41  Set pcbAppObj = Application
42
43  ' Get the active document
44  Set pcbDocObj = pcbAppObj.ActiveDocument
45
46  ' License the document
47  ValidateServer(pcbDocObj)
```

Line 50 sets the document units equal to the script units.

```
49  ' Set the unitsEnum to match the hard coded values.
50  pcbDocObj.CurrentUnit = unitsEnum
```

Line 54 starts a transaction, which means that any design changes from this point to the end of the transaction can be reversed by means of a single UNDO command.

```
52  ' Start a transaction to put all changes on single
53  ' undo level.
54  pcbDocObj.TransactionStart
```

Line 57 calls the main *AddBoardDimensions()* subroutine and passes it two parameters: the first specifies where to add the horizontal dimensional information (at the bottom of the board in this example); the second specifies where to add the vertical dimensional information (to the right of the board in this example).

```
56  ' Add dimensions
57  Call AddBoardDimensions(LOC_BOTTOM, LOC_RIGHT)
```

Similarly, Line 60 calls the main *AddCompanyLogo()* subroutine and passes it two parameters that specify where the company logo displays (offset from the bottom-right corner of the board outline in this example).

```
59  ' Add company logo
60  Call AddCompanyLogo(LOC_BOTTOM, LOC_RIGHT)
```

Finally, Line 63 ends the transaction that Line 54 initiates.

```
62  ' End the transaction
63  pcbDocObj.TransactionEnd
```

Add Board Dimensions Subroutine

Lines 71 through 79 declare the main *AddBoardDimensions ()* subroutine that adds the dimensional information to the layout design. The first parameter to this routine specifies the board edge on which to display the horizontal dimension (the permitted options are the top or the bottom). The second parameter specifies the board side on which to display the vertical dimension (the permitted options are the left or the right).

Lines 73 and 74 return the *BoardOutline* object. Lines 77 and 78 call the AddVertDimExtrema() Subroutine and AddHorizDimExtrema() Subroutine to add the vertical and horizontal dimension graphics and text.

```
68  ' Adds dimesion graphics to mark the extrema of the board.
69  ' horizDimLocConst - Constant (LOC_TOP, LOC_BOTTOM)
70  ' vertDimLocConst - Constant (LOC_RIGHT, LOC_LEFT)
71  Sub AddBoardDimensions(horizDimLocConst, vertDimLocConst)
72      ' Add the dimensions to the board outline
73      Dim boardOutlineObj
74      Set boardOutlineObj = pcbDocObj.BoardOutline
75
76      ' Add a vertical and horizontal dimension
77      Call AddVertDimExtrema(boardOutlineObj, vertDimLocConst)
78      Call AddHorizDimExtrema(boardOutlineObj, horizDimLocConst)
79  End Sub
```

Add Company Logo Subroutine

Lines 84 through 113 declare the main *AddCompanyLogo ()* subroutine that adds the logo to the layout design. The options are to place the logo to the upper-left, upper-right, lower-left, or lower-right of the board outline.

The first parameter to this routine specifies the board edge on which to display the horizontal dimension (the permitted options are the top or the bottom). The second parameter specifies the board side on which to display the vertical dimension (the permitted options are the left or the right).

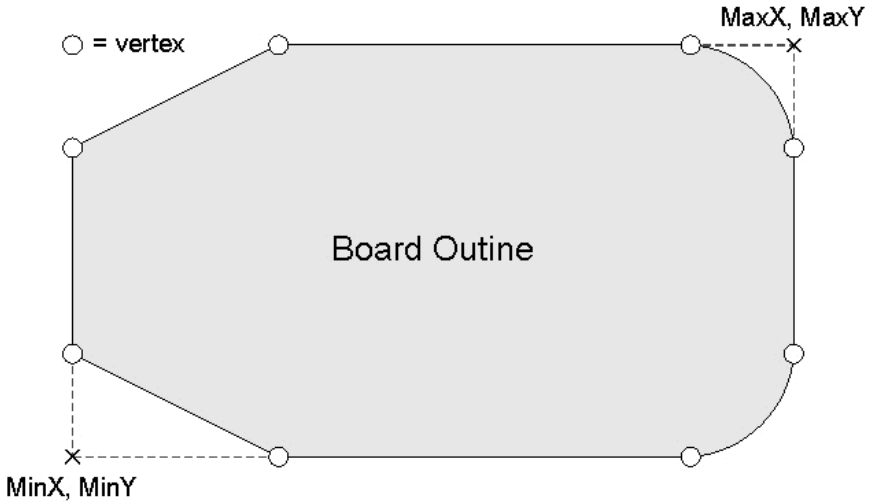
```
84  Sub AddCompanyLogo(vertLocConst, horizLocConst)
```

Lines 88 and 89 acquire the *Extrema* object for the *BoardOutline* object.

```
86      ' Get the extrema of the board to guide the
87      ' placement of the dimension
88      Dim extremaObj
89      Set extremaObj = pcbDocObj.BoardOutline.Extrema
```

This object has associated *MinX*, *MaxX*, *MinY*, and *MaxY* properties that define the extreme edges of the board. Figure 184: Extrema Object's MinX, MaxX, MinY, and MaxY Properties illustrates the Extrema for the *Candy.pcb* board.

Figure 184. Extrema Object's MinX, MaxX, MinY, and MaxY Properties



Lines 92 through 103 calculate the horizontal and vertical position for the logo. The calculation of the horizontal value depends on whether the logo is placed to the left or right of the board outline; similarly, the calculation of the vertical value depends on whether the logo is placed to the top or bottom of the board. The resulting *xPosReal* and *yPosReal* values define the placement of the cell's origin that contains the logo graphics (the cell's origin is set to the center of the logo in the example).

```
91      ' Calculate the logo location
92      Dim xPosReal, yPosReal
93      If horizLocConst = LOC_RIGHT Then
94          xPosReal = extremaObj.MaxX + LOGO_HORIZ_OFFSET
95      Else
96          xPosReal = extremaObj.MinX - LOGO_HORIZ_OFFSET
```

```
 97      End If
 98
 99      If vertLocConst = LOC_TOP Then
100          yPosReal = extremaObj.MaxY + LOGO_VERT_OFFSET
101      Else
102          yPosReal = extremaObj.MinY - LOGO_VERT_OFFSET
103      End If
```

Drawing cells are treated the same way as components, so Line 106 specifies that the logo's rotation should be zero and Line 109 uses the *PutComponent* method to place the drawing cell containing the logo. The parameters to the *PutComponent* method are the name of the cell, whether or not to pull it from a central library (*False* in this example, the *True* option is not currently supported), the X and Y coordinates, and the rotation. All additional parameter values are default.

```
105      ' Give it a zero orientation
106      Dim rotationReal: rotationReal = 0
107
108      ' Place the drawing cell
109      Call pcbDocObj.PutComponent(LOGO_CELL_NAME, False,
                                     xPosReal, yPosReal, rotationReal)
```

Finally, in order to make sure that the logo is visible, Line 112 turns on the display of the appropriate user layer.

```
110
111     ' Ensure the logo user layer is visible.
112     pcbDocObj.ActiveViewEx.DisplayControl.UserLayer(LOGO_LAYER_NAME) =
                                                                      True
113  End Sub
```

Utility Functions and Subroutines

This script section declares some utility routines. These routines are similar to the generic Helper Functions and Structures, except that they are specific to this script and may not be used elsewhere without changes.

AddVertDimExtrema() Subroutine

Lines 122 through 150 declare the routine that adds the vertical dimensional information. The first parameter to this routine is any graphics object (the script passes in the board outline); the second parameter defines whether the vertical dimension graphics and text are positioned to the left or right of the board.

```
118  ' Adds dimension marking the vertical extrama of gfxObj and
119  ' places the dimension on the locConst side of the object.
120  ' gfxObj - ConductorLayerGfx, UserLayerGfx, or
                                        FabricationLayerGfx Object
121  ' locConst - Constant (LOC_RIGHT, LOC_LEFT)
122  Sub AddVertDimExtrema(gfxObj, locConst)
```

Next, Line 126 locates the vertex at the top of the board and Line 127 locates the vertex at the bottom of the board. (Although the script already know the extents of the board from the *Extrema* object – as detailed in the discussions pertaining to Lines 88 and 89 – dimensions can be added only between vertices.)

To do this, the script uses the FindExtremaPoint() Function. The first parameter to this function is the graphics object (the board outline in this case); the second parameter specifies the *primary* search condition – whether the script is searching for the vertex at the top of the graphic object or the bottom of the graphic object; the third parameter specifies the *secondary* search condition – whether the script is searching in the left- or right-hand sides of the graphic object.

```
123      ' Find the max and min points in the
124      ' gfxObj points array
125      Dim maxPointCls, minPointCls
126      Set maxPointCls = FindExtremaPoint(gfxObj, LOC_TOP, locConst)
127      Set minPointCls = FindExtremaPoint(gfxObj, LOC_BOTTOM, locConst)
```

Why two search conditions? Consider the board outline illustrated in Figure 184: Extrema Object's MinX, MaxX, MinY, and MaxY Properties. In this case there are two vertices on the top of the board and two on the bottom. If the primary search criterion locates the vertex on the top of the board, and if there are multiple vertices on the top of the board, the secondary search criterion acts as a "tie-breaker".

Note that the FindExtremaPoint() Function is also used by the AddHorizDimExtrema() Subroutine. This means that if the primary search parameter passed into this routine is top or bottom, then the secondary search parameter must be left or right.

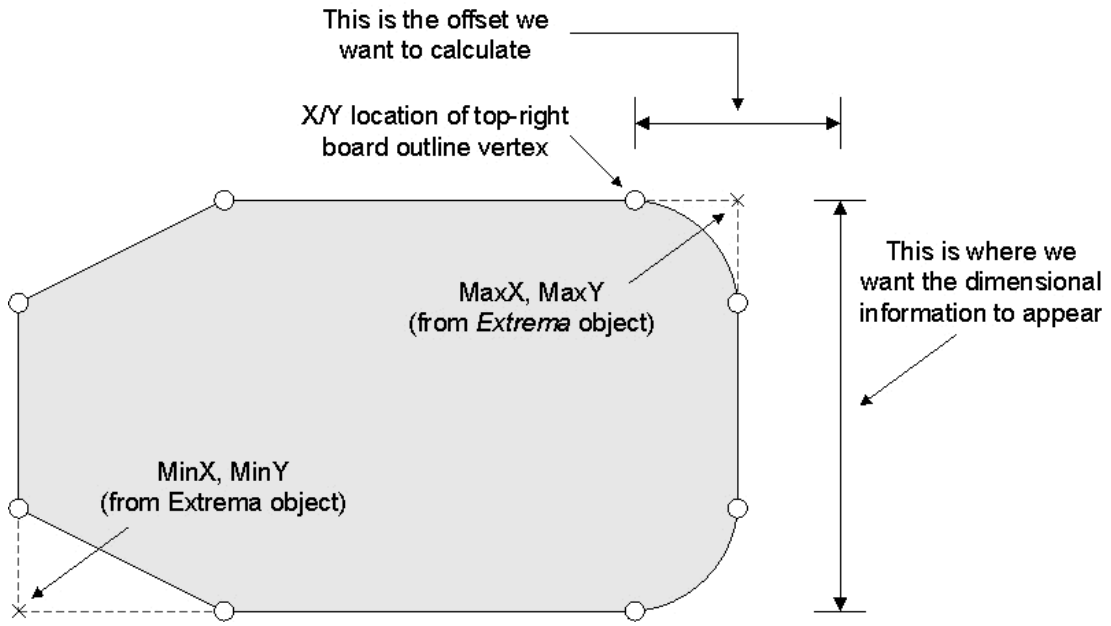
Line 131 acquires the *Extrema* object of the graphics object.

```
129      ' Get the extrema of the board to guide the
130      ' placement of the dimension
131      Dim extremaObj: Set extremaObj = gfxObj.Extrema
```

Lines 134 through 139 use the *Extrema* objects's *MinX* and *MaxX* properties and the horizontal value of the board outline vertex, to determine the offset for the vertical dimensional information. This calculation depends on if the dimensional information is to the left or the right of the board outline. If the right-hand side, then the result is illustrated in Figure 185: Calculating the Horizontal Offset for the Vertical Dimension.

```
128
132
133      ' Calculate the desired offset
134      Dim offsetReal
135      If locConst = LOC_RIGHT Then
136          offsetReal = extremaObj.MaxX - maxPointCls.X +
                                                DIMENSION_OFFSET
137      Else
138          offsetReal = extremaObj.MinX - maxPointCls.X –
                                                DIMENSION_OFFSET
139      End If
```

Figure 185. Calculating the Horizontal Offset for the Vertical Dimension



On Lines 142 and 143, the script creates a new *Layer* object and assigns the dimension user layer to it. Then, Line 144 calls the CreateUserLayer() Function; if the required user layer does not yet exist this function creates it, otherwise (if the layer does exist) this function simply returns it.

```
141      ' Create the layer object to put the dimension on
142      Dim layerObj
143      Set layerObj = pcbAppObj.Utility.NewLayerObject
144      layerObj.UserLayer = CreateUserLayer(DIMENSION_LAYER_NAME)
```

Finally, Lines 147 through 150, use the *PutPointToPointDimension* method to add the vertical dimension to the layout design. This method places a point to point dimension between two objects (in this example using the same object – the board outline). The parameters to this function are the "From" object, the X and Y locations on the "From" object, the "To" object, the X and Y locations on the "To" object, an offset (this can be positive or negative), an enumerate specifying whether to apply a horizontal or vertical dimension, the layer object upon which the dimension is placed, and the scheme to use (this defines the line style, arrow style, format of the font, etc. – pass *Nothing* to indicate the default scheme).

```
145
146      ' Add the dimension
147      Call pcbDocObj.PutPointToPointDimension(gfxObj, _
148           maxPointCls.X, maxPointCls.Y, gfxObj, _
149           minPointCls.X, minPointCls.Y, offsetReal, _
150           epcbOrientationVertical, layerObj, Nothing)
151  End Sub
```

AddHorizDimExtrema() Subroutine

Lines 157 through 186 declare the routine that adds the horizontal dimensional information to the layout design object. This routine works similarly to the AddVertDimExtrema() Subroutine in the previous section. The differences are left as an exercise for the reader.

```
153  ' Adds dimension marking the horizontal extrama of gfxObj and
154  ' places the dimension on the locConst side of the object.
155  ' gfxObj - ConductorLayerGfx, UserLayerGfx, or
                                      FabricationLayerGfx Object
156  ' locConst - Constant (LOC_TOP, LOC_BOTTOM)
157  Sub AddHorizDimExtrema(gfxObj, locConst)
158      ' Find the max and min points in the
159      ' gfxObj points array
160      Dim maxPointCls, minPointCls
161      Set maxPointCls = FindExtremaPoint(gfxObj, LOC_RIGHT, locConst)
162      Set minPointCls = FindExtremaPoint(gfxObj, LOC_LEFT, locConst)
163
164      ' Get the extrema of the board to guide the
165      ' placement of the dimension
166      Dim extremaObj: Set extremaObj = gfxObj.Extrema
167
168      ' Calculate the desired offset
169      Dim offsetReal
170      If locConst = LOC_TOP Then
171          offsetReal = (extremaObj.MaxY - maxPointCls.Y +
                                            DIMENSION_OFFSET) * -1
172      Else
173          offsetReal = (extremaObj.MinY - maxPointCls.Y –
                                            DIMENSION_OFFSET) * -1
174      End If
175
176      ' Create the layer object to put the dimension on
177      Dim layerObj
178      Set layerObj = pcbAppObj.Utility.NewLayerObject
179      layerObj.UserLayer = CreateUserLayer(DIMENSION_LAYER_NAME)
180
181      ' Add the dimension
182      Call pcbDocObj.PutPointToPointDimension(gfxObj, _
183           maxPointCls.X, maxPointCls.Y, gfxObj, _
184           minPointCls.X, minPointCls.Y, offsetReal, _
185           epcbOrientationHorizontal, layerObj, Nothing)
186  End Sub
```

FindExtremaPoint() Function

Lines 194 through 220 declare the function that finds the extreme vertex in a graphical object's points array. The three parameters to this function are the graphical object (the board outline in the example), the primary search criterion, and the secondary search criterion.

```
188  ' Returns extrema point of a vertex in gfxObj's points array.
189  ' The extrema direction is defined by primareLocConst and
190  ' secondaryLocConst
191  ' gfxObj - ConductorLayerGfx, UserLayerGfx, or
                FabricationLayerGfx Object
192  ' primaryLocConst - Constant (LOC_TOP, LOC_BOTTOM,
                                  LOC_LEFT, LOC_RIGHT)
```

```
193  ' secondaryLocConst - Constant (LOC_TOP, LOC_BOTTOM,
                                     LOC_LEFT, LOC_RIGHT)
194  Function FindExtremaPoint(gfxObj, primaryLocConst,
                                              secondaryLocConst)
```

The reason for having both primary and secondary search criteria is that there may be multiple vertices on each edge of the board. For example, if the primary search criterion is looking for a vertex on the top of the board and several vertices have the same maximum Y value, then the secondary search criterion looks for the right- or left-most vertex. This means that if the primary search criterion is top or bottom, then the secondary search criterion must be left or right; if the primary search criterion is left or right, then the secondary search criterion must be top or bottom.

Line 196 acquires the points array associated with the graphics object (the board outline in this case).

```
195      ' Get the points array for the geometry.
196      Dim pntsArr: pntsArr = gfxObj.Geometry.PointsArray
```

All of the previous scripts used pre-defined objects associated with various automation interfaces. In turn, these objects have pre-defined properties and methods. VBScript allows creation of custom-built objects, each of which can contain their own properties and methods. This is done by means of a *Class* structure.

Later in this script, a simple *Class* structure called *Point* is defined, where "Point" is an arbitrary name (refer to the discussions associated with Lines 318 through 321). The *Point* class has two properties: *X* and *Y*.

Line 200, defines two variables: *bestPointCls* (which holds the current best point) and *testPointCls* (which holds the next point to compare with the current best point).

```
198      ' Create a point class and initialize it
199      ' to the first point in the array.
200      Dim bestPointCls, testPointCls
```

Line 201 uses a *New* statement to create a new *Point* object and assigns the *Point* object to the *bestPointCls* variable. Lines 202 and 203 initialize the *X* and *Y* properties of the variable to the first point in the board outline points array (*X_NDX* and *Y_NDX* are the constants the script declared on Lines 17 and 18, respectively).

```
201      Set bestPointCls = New Point
202      bestPointCls.X = pntsArr(X_NDX, 0)
203      bestPointCls.Y = pntsArr(Y_NDX, 0)
```

Lines 205 through 216 iterate through all of the points in the points array starting with the second point (this assumes the first point is currently the best match).

```
204      Dim i
205      For i = 1 To UBound(pntsArr, 2)
```

Line 207 uses a *New* statement to create a new *Point* object and assigns the *Point* object to the *testPointCls* variable. Lines 208 and 209 initialize the *X* and *Y* properties of the variable to the second point in the board outline points array.

```
206          ' Create a point structure for this index
207          Set testPointCls = New Point
208          testPointCls.X = pntsArr(X_NDX, i)
209          testPointCls.Y = pntsArr(Y_NDX, i)
```

On Lines 212 and 213, the script calls the BetterMatch() Function to determine if the test point is a better match than the current best point. This determination is based on the primary and secondary search criteria passed into the main function. If this function returns *True*, indicating that the test point is a better match than the current best point, then Line 214 copies the test *Point* object into the current best *Point* object.

```
211          ' Check if this is a better match
212          If BetterMatch(bestPointCls, testPointCls, _
213              primaryLocConst, secondaryLocConst) Then
214              Set bestPointCls = testPointCls
215          End If
216      Next
```

Finally, Line 218 returns the best *Point* object.

```
210
217
218      Set FindExtremaPoint = bestPointCls
219
220  End Function
```

BetterMatch() Function

Lines 229 through 272 declare the function that determines whether the test point (the second parameter passed into the function) is a better match than the current best point (the first parameter). The function bases this determination on the primary and secondary search criteria, which are the third and fourth parameters respectively.

```
222  ' Returns true if testPointCls is a better match than
223  ' bestPointCls according to the primaryLocConst and
224  ' secondaryLocConst criteria. Returns false otherwise.
225  ' bestPointCls - Point Class
226  ' testPointCls - Point Class
227  ' primaryLocConst - Constant (LOC_TOP, LOC_BOTTOM,
                                          LOC_LEFT, LOC_RIGHT)
228  ' secondaryLocConst - Constant (LOC_TOP, LOC_BOTTOM,
                                          LOC_LEFT, LOC_RIGHT)
229  Function BetterMatch(bestPointCls, testPointCls, _
230                       primaryLocConst, secondaryLocConst)
```

On Line 232, the script initializes this function's return value to *False*, indicating the test point is not a better match.

```
231      ' Initialize return value
232      BetterMatch = False
```

Lines 234 through 271 specifies a *CASE* statement based on the primary search criterion. For each variant of the primary search criterion (top, bottom, left, right), the script checks if the test point is "better" than the current best point. In each case the definition of "better" depends on the primary search criterion.

If the test point is "better", the script sets this function's return value to *True*. Otherwise, it performs a second check to determine if the test point is "equal" to the current best point in terms of the primary

search criterion. If this is the case, the script calls the BetterValueMatch() Function to check the secondary criterion.

```
233
234     Select Case primaryLocConst
235         Case LOC_TOP
236             ' Do the primary check
237             If testPointCls.Y > bestPointCls.Y Then
238                 BetterMatch = True
239             ElseIf testPointCls.Y = bestPointCls.Y Then
240                 ' Do secondary check if primary is equal
241                 BetterMatch = BetterValueMatch(bestPointCls.X, _
242                     testPointCls.X, secondaryLocConst)
243             End If
244         Case LOC_BOTTOM
245             ' Do the primary check
246             If testPointCls.Y < bestPointCls.Y Then
247                 BetterMatch = True
248             ElseIf testPointCls.Y = bestPointCls.Y Then
249                 ' Do secondary check if primary is equal
250                 BetterMatch = BetterValueMatch(bestPointCls.X, _
251                     testPointCls.X, secondaryLocConst)
252             End If
253         Case LOC_RIGHT
254             ' Do the primary check
255             If testPointCls.X > bestPointCls.X Then
256                 BetterMatch = True
257             ElseIf testPointCls.X = bestPointCls.X Then
258                 ' Do secondary check if primary is equal
259                 BetterMatch = BetterValueMatch(bestPointCls.Y, _
260                     testPointCls.Y, secondaryLocConst)
261             End If
262         Case LOC_LEFT
263             ' Do the primary check
264             If testPointCls.X < bestPointCls.X Then
265                 BetterMatch = True
266             ElseIf testPointCls.X = bestPointCls.X Then
267                 ' Do secondary check if primary is equal
268                 BetterMatch = BetterValueMatch(bestPointCls.Y, _
269                     testPointCls.Y, secondaryLocConst)
270             End If
271     End Select
272 End Function
```

BetterValueMatch() Function

Lines 279 through 292 declare the function that checks the second criterion (acts as a "tie-breaker") for the previous function. In this case, it accepts as parameters the current best point, the test point, and the secondary search criterion.

```
274  ' Returns true if bestValReal is a better match than
275  ' testValReal according to the locConst criteria.  Returns
       false otherwise.
276  ' bestValReal - Real
277  ' testValReal - Real
278  ' locConst - Constant (LOC_TOP, LOC_BOTTOM, LOC_LEFT,
                                                        LOC_RIGHT)
279  Function BetterValueMatch(bestValReal, testValReal, locConst)
```

Line 281 initializes this function's return value to *False*, indicating the test point will not be a better match.

```
280      ' Initialize return value to false
281      BetterValueMatch = False
```

Lines 283 through 285 check values that are either the top edge or the right side of the board. If the test value is *greater* than the current best candidate value, then the test is better so returns *True*. Alternatively, Lines 287 through 289 check values that are either the bottom edge or the left side of the board. If the test value is *less* than the current best candidate value, then the test value is better so returns *True*.

```
282      ' Top or right must be greater
283      If (locConst = LOC_TOP Or locConst = LOC_RIGHT) And _
284         testValReal > bestValReal Then
285         BetterValueMatch = True
286      ' Bottom or left must be less
287      ElseIf (locConst = LOC_BOTTOM Or locConst = LOC_LEFT) And _
288         testValReal < bestValReal Then
289         BetterValueMatch = True
290      End If
291  End Function
```

Helper Functions and Structures

This script section declares several generic "helper" routines. These routines can be used in other scripts and are not focused on this script exclusively.

CreateUserLayer() Function

Lines 299 through 313 declare the function that creates and returns a new user layer or returns an existing user layer. The function accepts a single parameter, which specifies the name of the user layer.

```
296  ' Creates a user layer named userLayerNameStr if it does
297  ' not already exists otherwise returns the user layer object
298  ' userLayerNameStr - String
299  Function CreateUserLayer(userLayerNameStr)
```

On Line 301, the script declares a variable named *usrLyrObj*. Line 302 uses the *FindUserLayer* method to search for the *UserLayer* object and assigns it to the variable. If the *UserLayer* object does not exist, the variable is assigned a value of *Nothing*.

```
300      ' Check if the user layer already exists.
301      Dim usrLyrObj
302      Set usrLyrObj = pcbDocObj.FindUserLayer(userLayerNameStr)
```

Line 304 checks if the variable contains *Nothing*. If this is the case, then the *UserLayer* object does not exist, so Line 306 creates it.

```
304      If usrLyrObj Is Nothing Then
305          ' It doesn't exist. Create it.
306          Set usrLyrObj =
                 pcbDocObj.SetupParameter.PutUserLayer(userLayerNameStr)
307      End If
```

To assure the user layer is visible, Line 310 turns on the display of the user layer. Finally, Line 312 sets the return value from this function to the *UserLayer* object.

```
303
308
309      ' ensure the user layer is turned on
310      pcbDocObj.ActiveViewEx.DisplayControl.
                                 UserLayer(userLayerNameStr) = True
311
312      Set CreateUserLayer = usrLyrObj
313  End Function
```

Class Point Structure

VBScript allows you to create custom-built objects, each of which can contain their own properties and methods. This is achieved by means of a *Class* structure.

On Line 318, the script defines a simple *Class* structure called *Point* (where "Point" is an arbitrary name). Lines 219 and 320 define two properties, *X* and *Y*, which are associated with the *Point* class.

```
315  ' Class for maintaining point information
316  ' X - Real
317  ' Y - Real
318  Class Point
319      Dim X
320      Dim Y
321  End Class
```

Note:
Anything defined that falls between the *Class* and *End Class* statements, whether it is variables, subroutines or functions, is a *member* of that class. You access and modify *members* using the *Name.Property* or *Name.Method* syntax.

Note:
Any script can be written without using *Class* structures, but if used properly *Classes* can make your scripts more organized, readable, and reusable.

Creating and Testing the Script

1. Use the scripting editor of your choice to enter the script described (including the license server function) and save it as *FinalizeDesign.vbs*.

2. Close any instantiations of Xpedition Layout that are currently running, and then launch a new instantiation of this application.

3. Open the *Candy.pcb* design.

4. Run the *FinalizeDesign.vbs* script and observe that the dimensions and logo appear in your design.

Enhancing the Script

There are a number of ways to enhance this script. Several suggestions follow (practice your scripting skills by implementing these suggestions):

- Modify the script to add some other documentation cells.

- Modify the script to add the date of the finalization and/or the name of the designer.

Exercise 8: Routing by Layer

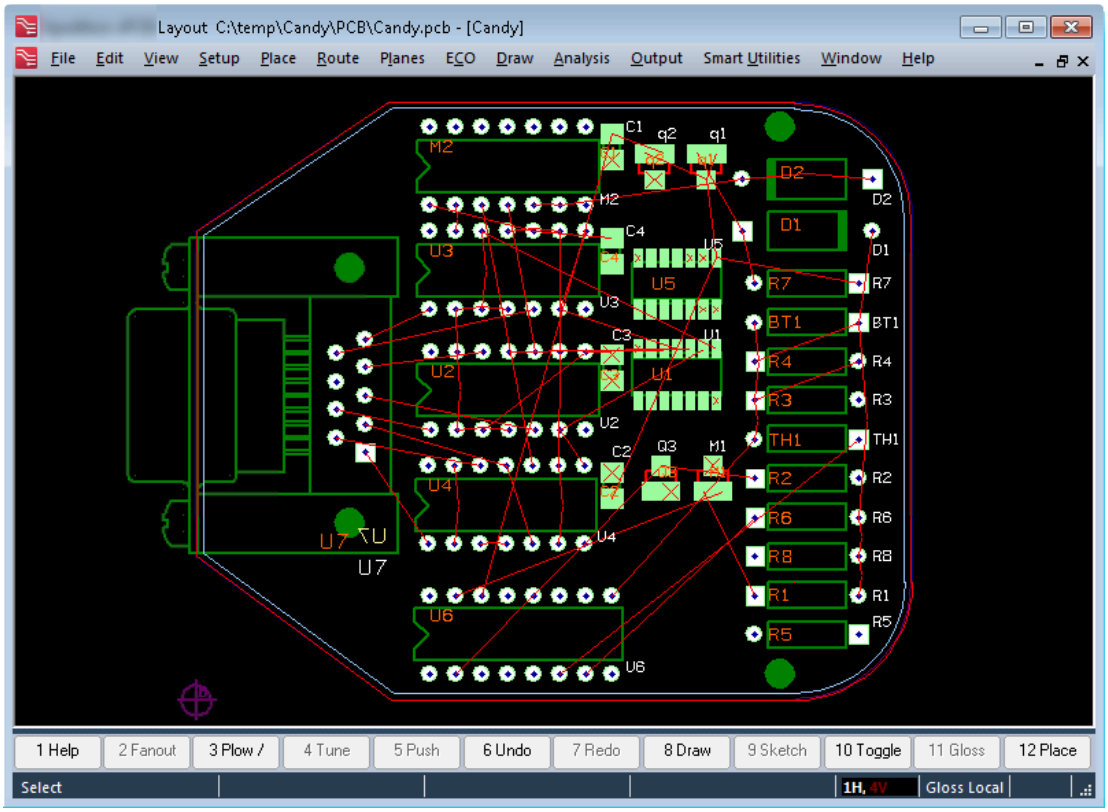
This script guides the auto-router by specifying that certain nets route only on certain layers.

Approach:	To setup a route pass and to associate nets that meet certain criteria with this route pass. Also to restrict the route pass to a specific set of layers.
Points of Interest:	– Running and controlling the auto-router
	– Using Layer objects
	– Performing geometric calculations
Items Used:	– RoutePass Object
	– LayerObject Object
	– Dictionary Object

Before Starting

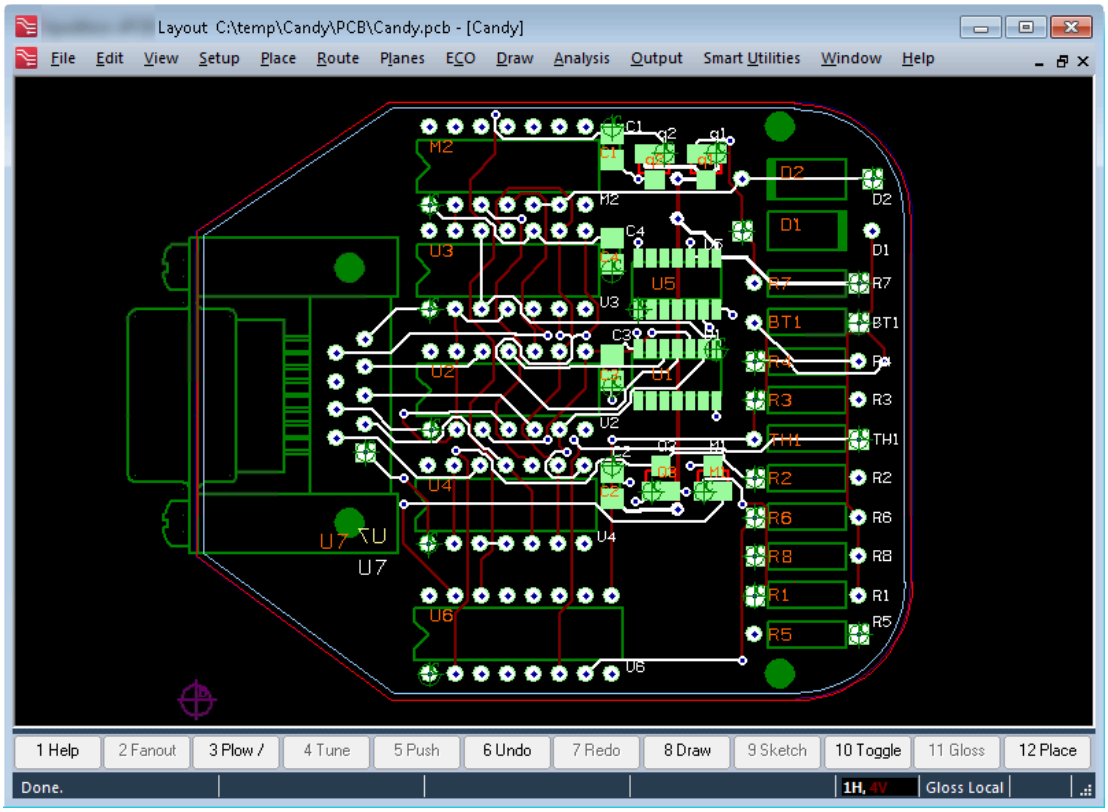
To understand how this script works, launch Xpedition Layout, open the *Candy.pcb* layout design, and choose **Route > Edit Routes > Delete all Traces and Vias**. The result is shown in Figure 186: Edit > Delete All Traces and Vias.

Figure 186. Edit > Delete All Traces and Vias



Run the *RouteByLayer.vbs* script. The results illustrated in Figure 187: After RouteByLayer.vbs Script Runs.

Figure 187. After RouteByLayer.vbs Script Runs



There are certain cases where end-user observation of a board's characteristics result in better routing of certain nets. Thus, depending on the type of board (as determined by the end-user), you can use the automation interface to make calculations, to perform evaluations, and to control the auto-router accordingly.

The script *RouteByLayer.vbs* routes the entire board. It evaluates the From-To points of each netline and determines if a net has a strong horizontal or vertical bias. Based on these evaluations, the script sets-up the auto-router to route horizontally-biased nets on signal layers on the top-half of the board; to route vertically-biased nets on the signal layers on the bottom-half of the board.

In Figure 187: After RouteByLayer.vbs Script Runs, observe that the signal layers on the top-half of the board contain the horizontally-biased nets (white); and that the signal layers on the bottom-half of the board contain the vertically-based nets (red).

Script Sections

The key sections of the *RouteByLayer.vbs* script follow:

Constant Definitions	Defines the constants the script uses.
Initialization/Setup	Returns the *Application* and *Document* objects, licenses the document, and adds the type libraries. Also performs any additional setup associated with this script.

Main Functions and Subroutines	Defines the main functions and subroutines the script uses.
Run Fanout Route Pass Subroutine	– RunFanoutPass()
Run Angle-Bias-Based Route Pass Subroutine	– RunAngleBiasRoutePass()
Utility Functions and Subroutines	Defines special "workhorse" routines that are focused on this script.
	– RunRoutePass() Subroutine
	– FindNets() Function
Helper Functions and Subroutines	Defines generic "helper" routines that other scripts can use.
	– GetSignalLayer() Function
	– GetSignalLayerCount() Function
	– GetAngleOfLine() Function
	– RadiansToDegrees() Function
Validate Server	Licenses the document and validates the server. Refer to "Running Your First Script" on page 59 for more details on this function.

Constant Definitions

The only constant this script defines is PI.

```
 1  ' This script demonstrates how to use the RoutePass object.
 2  ' It attempts to guide the auto-router by forcing netlines
 3  ' with a vertical bias to be routed on the bottom signal
 4  ' layers first.
 5  '                                        (Author: Toby Rimes)
 6  Option Explicit
 7
 8  ' Constants
 9  Const PI = 3.14159265358979323846264338327 95
```

Initialization/Setup

Lines 12 through 25 perform the standard tasks of adding any required type libraries, acquiring the *Application* and *Document* objects, and licensing the document by calling the standard *ValidateServer()* function.

```
11  ' Add any type libraries to be used.
12  Scripting.AddTypeLibrary("MGCPCB.ExpeditionPCBApplication")
13
14  ' Global variables
```

```
15  Dim pcbAppObj                 'Application object
16  Dim pcbDocObj                 'Document object
17
18  ' Get the application object.
19  Set pcbAppObj = Application
20
21  ' Get the active document
22  Set pcbDocObj = pcbAppObj.ActiveDocument
23
24  ' License the document
25  ValidateServer(pcbDocObj)
```

On Line 28, the script calls the main *RunFanoutPass()* subroutine.

Line 31 calls the *RunAngleBiasRoutePass()* subroutine. This routine performs two passes; it first searches for vertically-biased nets and routes these nets on the bottom signal layers, it then routes the horizontally-biased nets on the top signal layers.

Finally, Line 34 displays a message on the status bar to inform the end-user that the script is complete.

```
26
27  ' Run a fanout pass
28  Call RunFanoutPass()
29
30  ' Run the angle bias route passes.
31  Call RunAngleBiasRoutePass()
32
33  ' Let the user know the script is done.
34  Call pcbAppObj.Gui.StatusBarText("Done.")
```

Run Fanout Route Pass Subroutine

Lines 43 through 56 declare the *RunFanoutPass ()* subroutine, which performs the initial fanout route pass. Lines 45 and 46 use the *NewRoutePass* method to acquire a *RoutePass* object and assigns the object to the variable *fanoutPassObj*.

```
42  ' Run a fanout route pass
43  Sub RunFanoutPass()
44      ' Get a new RoutePass object
45      Dim fanoutPassObj
46      Set fanoutPassObj = pcbDocObj.NewRoutePass
```

Line 49 sets the *PassType* method. The first parameter is an enumerate from the type library that specifies to perform a *fanout* pass. The second and third parameters specify the start and end efforts, respectively (these efforts are both set to 1 in this example). The fourth and fifth parameters are both set to *False*, thereby indicating that the script is not using a via grid or a route grid, respectively.

```
48      ' Set as fanout pass
49      Call fanoutPassObj.PassType(epcbARFanoutPass, 1, 1, False, False)
```

Line 51 uses the *Items* method to set the design items to route. The first parameter is an enumerate that specifies all nets, which requires that the second parameter, a collection, is set to *Nothing* (using other enumerates for the first parameter requires different values on the second).

```
50      ' Route all nets
51      Call fanoutPassObj.Items(epcbARAllNetsItem, Nothing)
```

Line 53 uses the *StatusBarText* method on the *Gui* object to display a message on the status bar to inform the end-user as to what is happening.

```
52      ' Let the user know what is happening
53      Call pcbAppObj.Gui.StatusBarText("Running fanout pass")
```

Finally, Line 55 uses the *Go* method to start this route pass.

```
54      ' Start the route pass.
55      Call fanoutPassObj.Go()
56  End Sub
```

Run Angle-Bias-Based Route Pass Subroutine

Lines 61 through 88 declare the subroutine that performs two angle-bias-based route passes. First, the script determines the board's signal layers. Lines 63 and 64 acquire the board’s layer stackup (the *False* parameter on Line 64 indicates no insulation layers, only the conducting plane and signal layers).

```
58  ' Run angle bias route pass. Routes the vertical
59  ' netlines on the bottom half of the board and routes
60  ' remaining netlines on the top half of the board.
61  Sub RunAngleBiasRoutePass()
62      ' Get the layer stack
63      Dim layerStackColl
64      Set layerStackColl = pcbDocObj.LayerStack(False)
```

Lines 67 and 68 create an empty *LayerObject* object collection.

```
66      ' Get an empty LayerObject collection
67      Dim signalLayerColl
68      Set signalLayerColl = pcbAppObj.Utility.NewLayerObjects
```

Line 72 declares a variable to control a loop, and then on Lines 73 through 75, the script iterates through all of the signal layers in reverse order (the reason for iterating in reverse order is discussed later in this section).

Observe the GetSignalLayerCount() Function, which returns the number of signal layers. On Line 73, the script uses this function to define the scope of the iteration from the bottom signal layer (the one with the highest number) to the midpoint layer. Meanwhile, on Line 74, the script returns the i th signal layer and adds it to the collection.

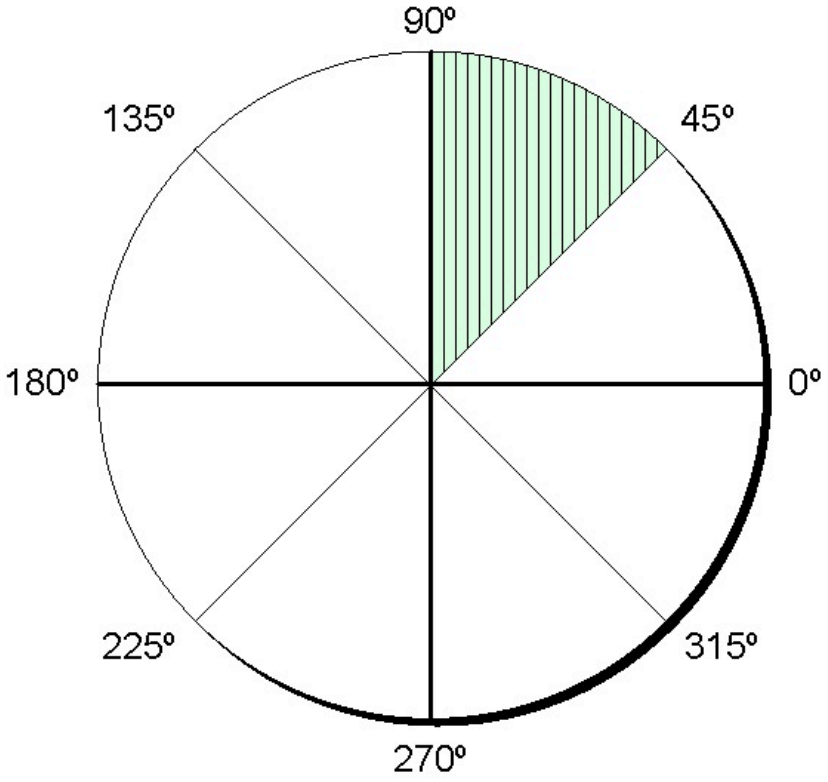
```
70      ' Add the signal layers from the bottom
71      ' half of the board.
72      Dim i
73      For i = GetSignalLayerCount(layerStackColl) To
                   Int(GetSignalLayerCount(layerStackColl) / 2) + 1 Step
-1
74          signalLayerColl.Add(GetSignalLayer(layerStackColl, i))
75      Next
```

Line 78 calls the RunRoutePass() Subroutine, which causes the auto-router to route vertically-biased nets or horizontally-biased nets. The required bias is defined by the first and second parameters, which specify the *From* and *To* angles, respectively. The third parameter is the signal layer collection, which currently contains only the signal layers on the bottom-half of the board.

```
77      ' Route vertical netlines on bottom half of board.
78      Call RunRoutePass(45, 90, signalLayerColl)
```

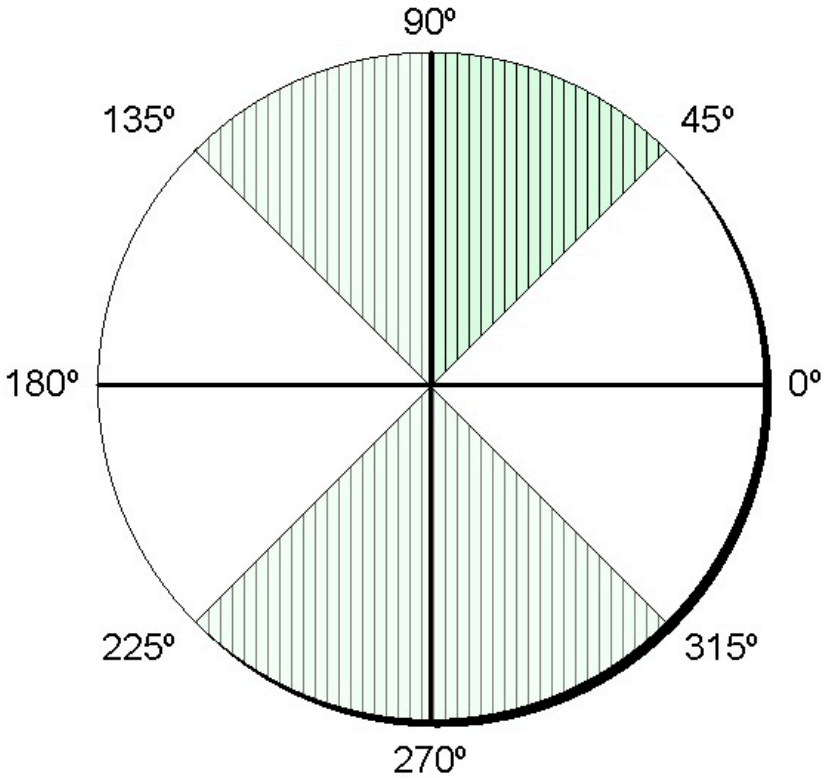
Now, observe 45 and 90 as parameters to specify *From* and *To* angles of 45º and 90º. The script uses these values to specify vertically-biased nets; however, the values cover only a small range of the available possibilities as illustrated in Figure 188: Nets Defined by From/To Angles of 45º and 90º.

Figure 188. Nets Defined by From/To Angles of 45º and 90º



The RunRoutePass() Subroutine calls the FindNets() Function, which calls the GetAngleOfLine() Function. The GetAngleOfLine() Function effectively "reflects" corresponding areas in the other three quadrants back into the first area as illustrated in Figure 189: Nets Affected by the From/To Angles of 45º and 90º.

Figure 189. Nets Affected by the From/To Angles of 45º and 90º



You could *explicitly* route vertically-biased nets on one set of signal layers and then *explicitly* route horizontally-biased nets on another (unique) set of signal layers. In practice, however, once you route the vertical nets on one set of layers, you can auto-route the remaining nets using any of the signal layers. Thus, Lines 81 through 83 *add* the signal layers from the top-half of the board *into* the existing collection.

```
80      ' Add the remaining signal layers.
81      For i = Int(GetSignalLayerCount(layerStackColl) / 2) To 1 Step -1
82          signalLayerColl.Add(GetSignalLayer(layerStackColl, i))
83      Next
```

Finally, on Line 87, the script calls the RunRoutePass() Subroutine again, but this time specifies *From* and *To* angles of 0º and 90º, which defines an entire quadrant. The other three quadrants are reflected back into the first, so, the script effectively specifies all of the nets in the design. This is OK, as the first pass routed all of the vertically-biased nets, so only the horizontally-biased nets remain to route.

```
85      ' Route the horizontal and remaining vertical
86      ' netlines on the bottom and top half of the board.
87      Call RunRoutePass(0, 90, signalLayerColl)
88  End Sub
```

Utility Functions and Subroutines

This script section declares utility routines. These routines are similar to the generic Helper Functions and Subroutines, except that they are specific to this script and may not be used elsewhere without changes.

RunRoutePass() Subroutine

Lines 99 through 126 declare the routine that runs a route pass using all of the netlines with an angle (vertical or horizontal bias) that falls within a certain range. This routine accepts three parameters: the *From* angle, the *To* angle, and a layer collection on which to route the selected nets.

```
 93  ' Runs a Route route pass using all the netlines with
 94  ' an angle between fromAngleReal and toAngleReal.  Only
 95  ' allows routing on layers defined in layerColl
 96  ' fromAngleReal - Real
 97  ' toAngleReal - Real
 98  ' layerColl - LayerObjects Collection
 99  Sub RunRoutePass(fromAngleReal, toAngleReal, layerColl)
```

Lines 101 and 102 b acquire a new *RoutePass* object.

```
100      ' Get a new RoutePass object
101      Dim routePassObj
102      Set routePassObj = pcbDocObj.NewRoutePass
```

Line 106 declares a variable to hold a collection of nets. Line 107 calls the FindNets() Function, which accepts the *From* and *To* angle values as parameters.

```
104      ' Get the collection of nets with newlines at the
105      ' desired angle.
106      Dim netColl
107      Set netColl = FindNets(fromAngleReal, toAngleReal)
```

Line 110 sets the *PassType* method. The first parameter is an enumerate from the type library that specifies to perform a *route* pass. The second and third parameters specify the start and end efforts, respectively (these efforts are both set to 1 in this example). The fourth and fifth parameters are both set to *False*, thereby indicating that the script is not using a via grid or a route grid, respectively.

```
109      ' Set the pass type
110      Call routePassObj.PassType(epcbARRoutePass, 1, 1,  False, False)
```

Line 112 uses the *Items* method to set the design items to route. The first parameter is an enumerate that specifies a specific collection of nets, the second parameter is the collection of nets itself (compare this with the previous use of this method on Line 51, refer to Run Fanout Route Pass Subroutine).

```
111      ' Add the net collection
112      Call routePassObj.Items(epcbARNetsItem, netColl)
```

By default, a route pass is set to route on all layers. Thus, Line 114 removes all of the layers, then Lines 117 through 121 iterate through the collection of signal layers adding each using the *LayerSelect* property.

```
113      ' It is initialized to all layers so remove the layers.
114      Call routePassObj.LayerSelect(epcbARLyrsRemoveLayer, 0)
115      ' Add only the layers specified
116      Dim layerObj
117      For Each layerObj In layerColl
118          ' Bug in the automation interface. LayerSelect
```

```
119          ' uses (base 0 layers).
120          Call routePassObj.LayerSelect(epcbARLyrsAddLayer,
                                                    layerObj.ConductorLayer -
 1)
121      Next
```

Observe that the *LayerSelect* property used on Line 120 accepts two parameters. The first is an enumerate, which informs the system to *add* a layer. The second parameter is an integer that specifies the layer number. For reasons that are beyond the scope of this tutorial, the *RoutePass* object uses 0-based (zero-based) layer numbering. This means that layer 1 in the real world is considered to be layer 0 by the *RoutePass* object; layer 2 in the real world is considered to be layer 1 by the *RoutePass* object; and so forth.

The *LayerSelect* method associated with the *RoutePass* object understands a parameter of 0 to mean "all layers", which means that it is not possible for the script to select layer 1 in the real world. This does not impact the script as the script iterates through the signal layers in reverse order, first on Lines 73 through 75 and later on Lines 81 through 83. Also, recall that once the vertical nets are routed on a set of layers, the script auto-routes the remaining nets using any of the signal layers.

In order to understand how this works, assume that the script is working with a board containing six conducting layers as follows:

1	Signal	{Top half of board}
2	Signal	{Top half of board}
3	Plane	
4	Plane	
5	Signal	{Bottom half of board}
6	Signal	{Bottom half of board}

Lines 73 through 75 add signal layers 6 and 5 to the collection, and then Line 78 calls the RunRoutePass() Subroutine the *first* time. The first time, the loop on Lines 117 through 121 adds layer 6 (as layer 5 from the perspective of the *RoutePass* object and its *LayerSelect* method on Line 120) and then adds layer 5 (as layer 4).

Lines 81 through 83 add signal layers 2 and 1 to the existing collection, and then Line 87 calls the RunRoutePass() Subroutine the *second* time. The second time, the loop on Lines 117 through 121 adds layer 6 (as layer 5), then layer 5 (as layer 4), then layer 2 (as layer 1), and finally layer 1 (as layer 0). Layer 0 in this final iteration is understood by the *LayerSelect* method to mean "all layers", but it is OK because that is what is required in this case. (Note that it is OK to attempt to add the same layer multiple times. If the layer is already in the collection then adding it again has no effect.)

On Line 123, the script uses the *StatusBarText* method on the *Gui* object to display a message on the status bar to inform the end-user as to what is happening.

```
122      ' Let the user know what is happening
123      Call pcbAppObj.Gui.StatusBarText("Running route pass for angles "
                                          & fromAngleReal & " - " &
 toAngleReal)
```

Finally, on Line 125, the script uses the *Go* method to start the route pass.

```
124      ' Start the route pass.
125      Call routePassObj.Go()
126  End Sub
```

FindNets() Function

Lines 132 through 164 declare the function that returns the nets with netlines between the specified *From* and *To* angles.

```
128  ' Returns a collection of nets that have netlines between
129  ' the angles fromAngleReal and toAngleReal
130  ' fromAngleReal - Real
131  ' toAngleReal - Real
132  Function FindNets(fromAngleReal, toAngleReal)
```

Lines 133 and 134 create a *Dictionary* object (a temporary mapping that ensures the script does not add the same net more than once).

```
133      Dim netDictObj
134      Set netDictObj = Createobject("Scripting.Dictionary")
```

Lines 137 and 138 return a collection of *FromTos* that contain all of the netlines on the board.

```
136      ' Get the collection of all netlines
137      Dim fromToColl
138      Set fromToColl = pcbDocObj.FromTos
```

Lines 141 through 151 iterate through the collection of *FromTos*. Line 143 calls the GetAngleOfLine() Function to determine the angle associated with the current *FromTo*. Line 145 checks if this angle matches the specified criteria and if so, Line 147checks if this net is already in the *Dictionary* object. If the net is not in the *Dictionary* object, Line 148 adds the net to the *Dictionary* object.

```
140      Dim fromToObj, angleReal
141      For Each fromToObj In fromToColl
142          ' Get the angle of the netline
143          angleReal = GetAngleOfLine(fromToObj.FromX, fromToObj.FromY,
                                        fromToObj.ToX, fromToObj.ToY)
144          ' Determine if the angle is in range
145          If (angleReal >= fromAngleReal) And
                (angleReal <= toAngleReal) Then
146                  ' Only add it if it is not present.
147              If Not netDictObj.Exists(fromToObj.Net.Name) Then
148                  Call netDictObj.Add(fromToObj.Net.Name, fromToObj.Net)
149              End If
150          End If
151      Next
```

Lines 154 and 155 create an empty net collection.

```
153      ' Create a net collection from the nets in the dictionary.
154      Dim netColl
155      Set netColl = pcbAppObj.Utility.NewNets
```

Lines 158 through 160 iterate through the contents of the *Dictionary* object and adds all of the nets in this object to the net collection.

```
157      Dim keyStr
158      For Each keyStr In netDictObj.Keys
159          netColl.Add(netDictObj.Item(keyStr))
160      Next
```

Note:
The *Dictionary* objects use a key-value mapping structure. So, to iterate through the items in the dictionary, the script actually iterates through the collection of *Keys*. For each *Key*, the script uses the *Item* property to return the value (the *Net* object in this case) associated with the *Key*.

Finally, on Line 163, the script returns the collection of nets.

```
162      ' Return the collection of nets
163      Set FindNets = netColl
164  End Function
```

Helper Functions and Subroutines

This script section declares several generic "helper" routines. These routines can be used in other scripts and are not focused on this script exclusively.

GetSignalLayer() Function

Lines 174 through 190 declare the function that takes a collection of layers (this collection can include insulating, plane, and signal layers) and returns the nth signal layer. For example, assume the following layer collection:

1	Signal	{1st signal layer}
2	Signal	{2nd signal layer}
3	Plane	
4	Plane	
5	Signal	{3rd signal layer}
6	Signal	{4th signal layer}

The *GetSignalLayer()* function accepts two parameters: the first is a collection of layers, the second is the number of the signal layer (the nth signal layer) to return; for example:

Value of second parameter	Equates to	Actual layer number (layer returned)
1	First signal layer	1
2	Second signal layer	2
3	Third signal layer	5
4	Fourth signal layer	6

```
170  ' Given a layer collection returns the nthLayerInt signal
171  ' layer object
172  ' layerColl - LayerObjects Collection
173  ' nthLayerInt - Integer
174  Function GetSignalLayer(layerColl, nthLayerInt)
```

On Line 175, the script initializes the return value of this function to *Nothing*.

```
175      Set GetSignalLayer = Nothing
```

Line 176 instantiates a variable that holds a *Layer* object. Line 177 instantiates an integer variable and initializes it to 0 (this variable keep track of number of many signal layers found).

```
176      Dim layerObj
177      Dim numFoundInt : numFoundInt = 0
```

Lines 181 through 189 iterate through the layer collection. Line 182 checks if the layer type is a conductor and if so, checks if the conductor type is a signal layer. If the layer is a signal layer, Line 183 increments the count of signal layers found and Line 184 compares the count to the count passed in as a parameter. If this is the layer to return, Line 185 sets the function's return value to this layer and Line 186 exits the function.

```
178
179      ' Iterate through the collection until the nth
180      ' signal layer is found.
181      For Each layerObj In layerColl
182          If layerObj.Type = epcbLayerTypeConductor And
                      layerObj.ConductorType = epcbConductorTypeSignal
 Then
183              numFoundInt = numFoundInt + 1
184              If numFoundInt = nthLayerInt Then
185                  Set GetSignalLayer = layerObj
186                  Exit Function
187              End If
188          End If
189      Next
190  End Function
```

GetSignalLayerCount() Function

Lines 194 through 207 declare the function that counts the number of signal layers in the layer collection passed in as a parameter.

```
192  ' Returns the number of signal layers in layerColl
193  ' layerColl - LayerObjects Collection
194  Function GetSignalLayerCount(layerColl)
```

Line 195 instantiates an integer variable and initialize it to 0 (this variable keeps track of the number of signal layers found).

```
195      Dim numInt: numInt = 0
```

Lines 199 through 203 iterate through the layer collection. Line 200 checks if the layer type is a conductor and if so, checks if the conductor type is a signal layer. If the layer is a signal layer, then Line 201 increments the number of signal layers found.

```
197      ' Iterate through the collection counting signal layers.
198      Dim layerObj
199      For Each layerObj In layerColl
200          If layerObj.Type = epcbLayerTypeConductor And
                      layerObj.ConductorType = epcbConductorTypeSignal
 Then
201             numInt = numInt + 1
202          End If
203      Next
```

Line 206 sets the function's return value to the number of signal layers found and exits the function.

```
196
204
205      ' Return the number of layers
206      GetSignalLayerCount = numInt
207  End Function
```

GetAngleOfLine() Function

Lines 214 through 229 declare the function that determines the angle of a line in degrees (from 0º to 90º). This function accepts four parameters that define the X/Y coordinates of the start and end of the line.

```
209  ' Returns the angle of the line in degrees from 0 to 90.
210  ' x1Real - Real
211  ' y1Real - Real
212  ' x2Real - Real
213  ' y2Real - Real
214  Function GetAngleOfLine(x1Real, y1Real, x2Real, y2Real)
```

Line 215 declares a variable that holds the angle of the line.

```
215      Dim angleReal
```

The equation the script uses on Line 224 does not handle the extreme 0º and 90º cases:

- Line 218 checks if the Y coordinates of the line's two end-points are identical; if identical, this is a horizontal line, so Line 219 sets the angle to 0 radians (or 0º).
- Line 220 checks if the X coordinates of the line's two end-points are identical; if identical, this is a vertical line, so Line 221 sets the angle to PI/2 radians (or 90º).

If the line is neither horizontal nor vertical, then the script uses the equation on Line 224 to calculate its actual angle in radians. Observe the VBScript *Atn* (arctan) and *Abs* (absolute value) functions. It is the *Abs* function that causes the upper-left, lower-left, and lower-right quadrants to be "reflected" into the upper-right quadrant as per the discussions associated with Figure 188: Nets Defined by From/To Angles of 45º and 90º and Figure 189: Nets Affected by the From/To Angles of 45º and 90º.

```
217      ' Handle the 0 and 90 case.
218      If y1Real = y2Real Then
219          angleReal = 0
220      ElseIf x1Real = x2Real Then
221          angleReal = PI/2
222      Else
223          ' Calculate all other cases
224          angleReal = Atn(Abs( (y2Real - y1Real) / (x2Real - x1Real) ))
225      End If
```

Finally, on Line 228, the script calls the RadiansToDegrees() Function to convert the angle of the line from radians to degrees. The function returns degrees to the calling routine.

```
216
226
227      ' Convert to degrees.
228      GetAngleOfLine = RadiansToDegrees(angleReal)
229  End Function
```

RadiansToDegrees() Function

Lines 233 through 235 declare a simple function that converts an angle specified in radians into a corresponding value in degrees.

```
231  ' Converts radians to degrees
232  ' radiansReal - Real
233  Function RadiansToDegrees(radiansReal)
234      RadiansToDegrees = radiansReal * 180 / PI
235  End Function
```

Creating and Testing the Script

1. Use the scripting editor of your choice to enter the script (including the license server function) and save it as *RouteByLayer.vbs*.
2. Close any instantiations of Xpedition Layout that are currently running, and then launch a new instantiation of this application.

3. Open the *Candy.pcb* design.

4. Choose **Route > Edit Routes > Delete all Traces and Vias** to remove any existing traces and vias.

5. Run the *RouteByLayer.vbs* script. Observe that the top signal layers contain the horizontally-biased nets (white); also that the bottom signal layers contain the vertically-based nets (red).

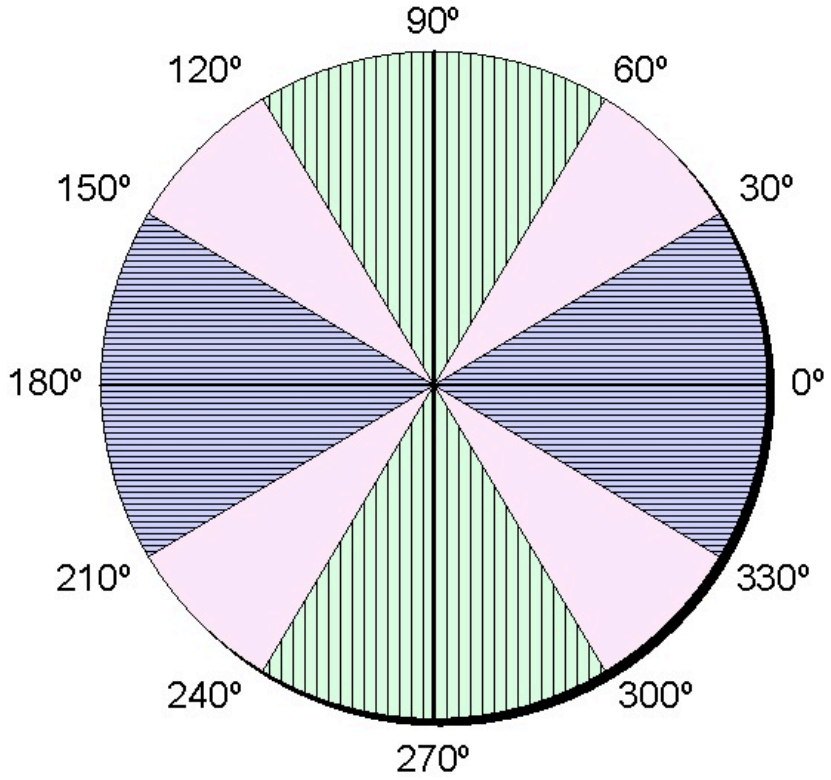
Enhancing the Script

There are a number of ways to enhance this script. Several suggestions follow (practice your scripting skills by implementing these suggestions):

- Use a board containing a substantial amount of surface-mount devices and create a new version of the script that directs the auto-router to route the board in the following manner:
  - Top-to-Top routes on the top of the board.
  - Bottom-to-Bottom routes on the bottom of the board.
  - Z-axis (Top-to-Bottom) routes on the inner layers of the board.
- Use a board with six or more signal layers and route vertically-biased tracks (green) on some layers, horizontally-biased tracks (blue) on other layers, and the tracks that are neither

horizontally-biased or vertically-biased (mauve) on yet other layers as illustrated in Figure 190: Categorizing and Routing Nets as Horizontally-biased, Vertically-biased, and "In-between".

Figure 190. Categorizing and Routing Nets as Horizontally-biased, Vertically-biased, and "In-between"



Chapter 15
Xpedition FabLink Examples

This section introduces Xpedition FabLink and presents several exercises focused on automating panel design and producing manufacturing outputs and documentation.

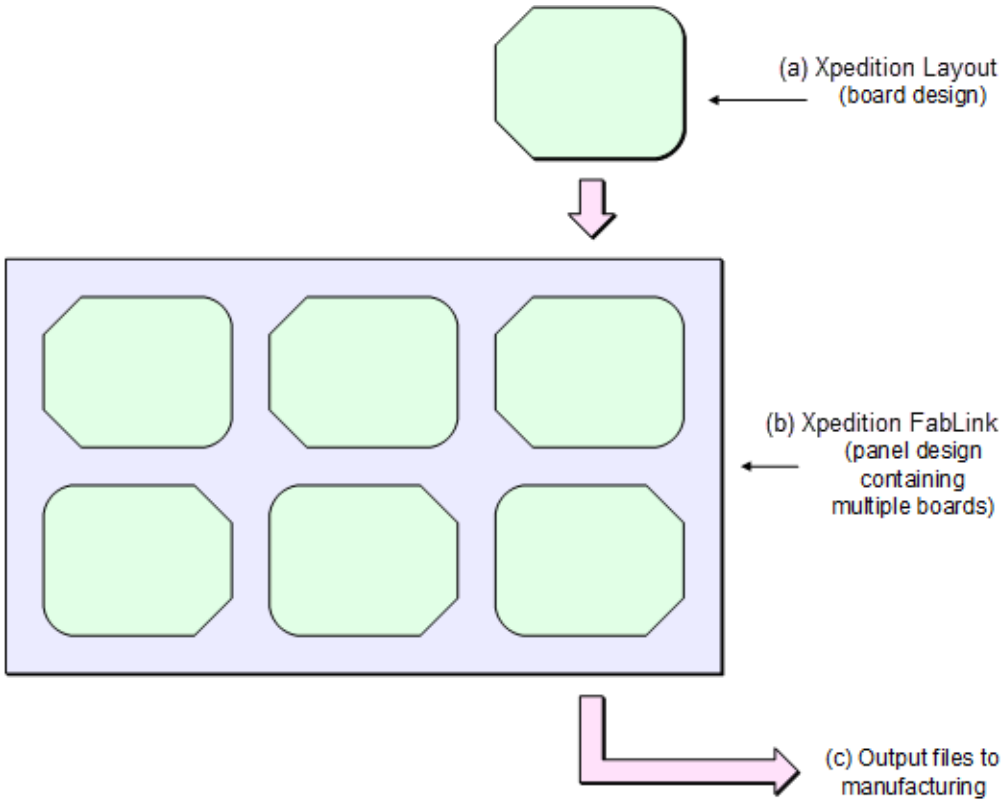
Introducing Xpedition FabLink
Exercise 1: Generating a BOM
Exercise 2: Generating a Panel Side View
Exercise 3: Creating a Via Mask
Exercise 4: Generating Manufacturing Output Files
Exercise 5: Automated Board Placement on a Panel

Introducing Xpedition FabLink

Xpedition FabLink is a manufacturing data creation, generation, and verification environment. Its integrated Design for Fabrication (DFF) analysis provides advanced fabrication and manufacturing Design Rule Checking (DRC), thereby reducing clean-up cycles to and from the fabricators and improving the quality, manufacturability, and yield of PCB designs.

Xpedition FabLink may be regarded as being "the next step" from Xpedition Layout because it helps verify your data and prepare files for manufacture. One way to think about this, is that Xpedition FabLink provides a "Bridge" between Xpedition Layout and the board manufacturing environment. Another way to contrast the two applications is that Xpedition Layout creates a single board design, while Xpedition FabLink can also create a panel containing multiple boards with the sheet size of fabrication materials used in the PCB manufacturing process as illustrated in Figure 191: High-level View of Xpedition FabLink Versus Xpedition Layout .

Figure 191. High-level View of Xpedition FabLink Versus Xpedition Layout



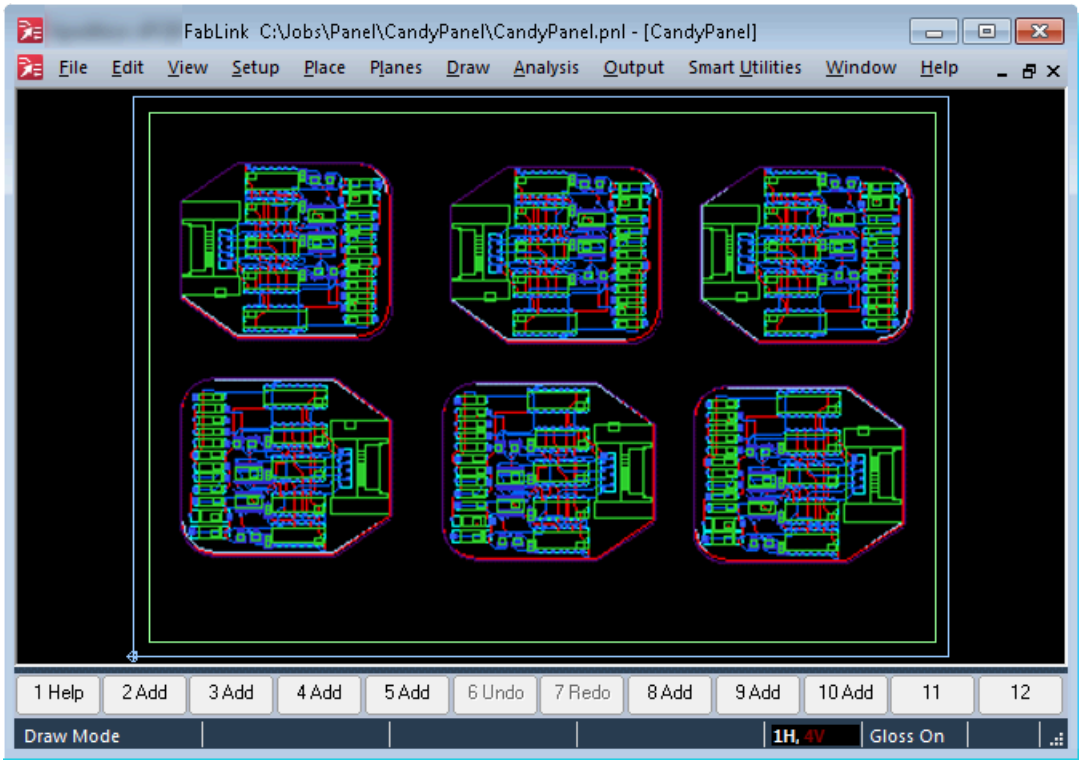
Xpedition FabLink From an Automation Perspective

The automation interface that Xpedition FabLink uses to access data, is the same interface that Xpedition Layout uses. However, there is certain functionality in Xpedition FabLink that is not applicable to (and therefore not available in) Xpedition Layout, and vice versa.

Xpedition FabLink has additional data structures, for example, the *PanelOutline* object (a *PanelOutline* object is to a panel, as a *BoardOutline* object is to a board). The end result is that different portions of the interface are automatically enabled or disabled depending on the application.

Figure 192: Xpedition FabLink Panel Containing Six Boards shows a panel design named *CandyPanel* containing six instances of the *Candy.pcb* board design.

Figure 192. Xpedition FabLink Panel Containing Six Boards



Xpedition FabLink's Graphical User Interface (GUI) visually depicts the items on each board (components, traces, pins, vias, etc.). However, the only items that are accessible via Xpedition FabLink (using its GUI *or* its automation interface) are the *Board* objects on the panel and/or any other data or objects that were added to the panel using Xpedition FabLink.

Xpedition FabLink is not intended for designing boards, therefore it is not possible to access items that are part of a board design from within Xpedition FabLink.

Exercise 1: Generating a BOM

This script generates a Build-of-Materials (BOM) for a panel by accessing components within each of the boards on the panel, consolidating the information, and outputting the desired data in the desired format. It accesses board information from Xpedition FabLink, component information from Xpedition Layout, and part information from the Parts Database (Parts DB).

Points of Interest:	– Using a Xpedition FabLink script to access data in Xpedition Layout.
	– Using a Xpedition FabLink script to access data in the Parts DB.
	– Using dictionary-based tree structures to store and manipulate component and part data.
Items Used:	– Xpedition Layout Server

	– Parts Editor Server
	– Dictionary Object

Before Starting

This script uses the Xpedition Layout, Xpedition FabLink, and Parts Editor automation engines. The script is intended to run from the command line; this means that there is no requirement to invoke Xpedition Layout, Xpedition FabLink, or the Parts Editor applications prior to running the script, because the script automatically launches these applications as required.

Note:
If Xpedition Layout, Xpedition FabLink, or the Parts Editor applications are running when the script is run, the script invokes new instantiations of these applications as required and then closes these new instantiations when the script no longer needs the applications.

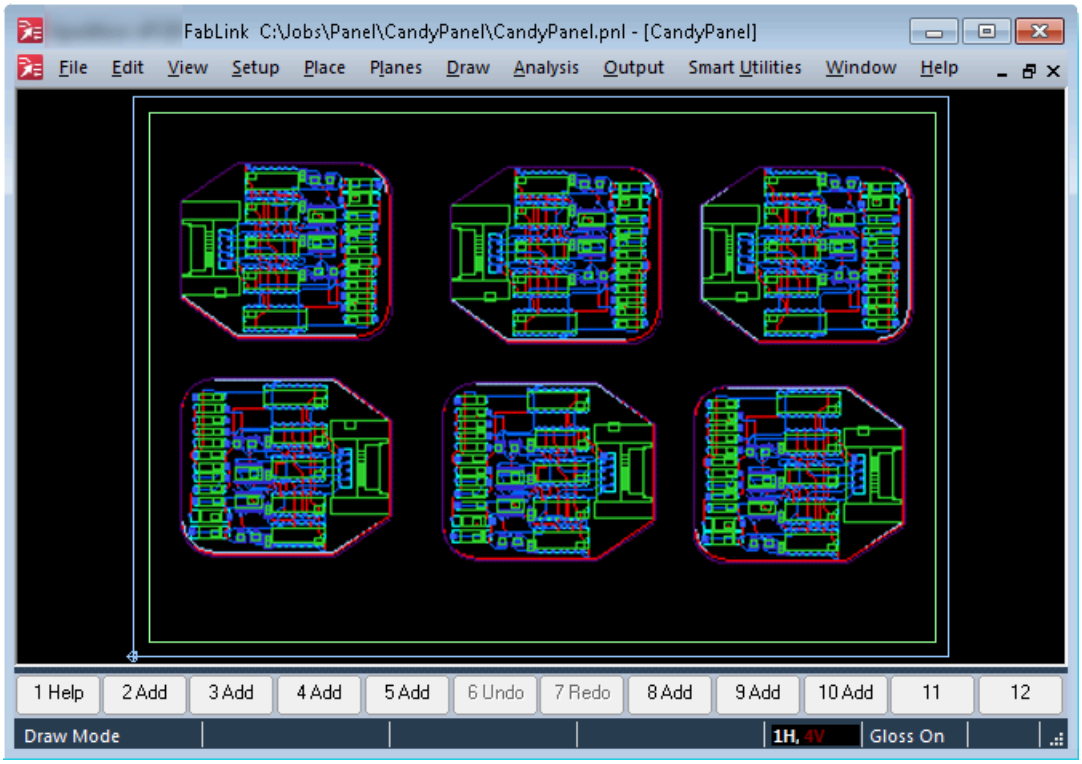
The script described in this exercise uses the Parts Editor automation server. There is also Padstack Editor and Cell Editor automation servers that scripts can use in the same manner as the Parts Editor. The *Prog IDs* for these interfaces are:

```
MGCPCBLibraries.PartsEditorDlg
MGCPCBLibraries.PadstackEditorDlg
CellEditorAddin.CellEditorDlg.CellEditorDlg
```

Some panels are homogenous (all of the boards on the panel are of the same type); other panels can be heterogeneous (the boards on the panel are two or more different types). For this exercise, you work with a homogeneous panel named *CandyPanel.pnl* that contains six identical copies of the *Candy.pcb* board design as illustrated in Figure 193: Xpedition FabLink CandyPanel.pnl.

If you know that all of the panels this script processes are homogeneous, the script requires access to components on one of the boards and then multiplies the number of each type of component by the number of boards on the panel. However, the script is written such that it gathers enough information to support both homogeneous and heterogeneous panels. This results in a script that can be easily extended to support any required output format.

Figure 193. Xpedition FabLink CandyPanel.pnl



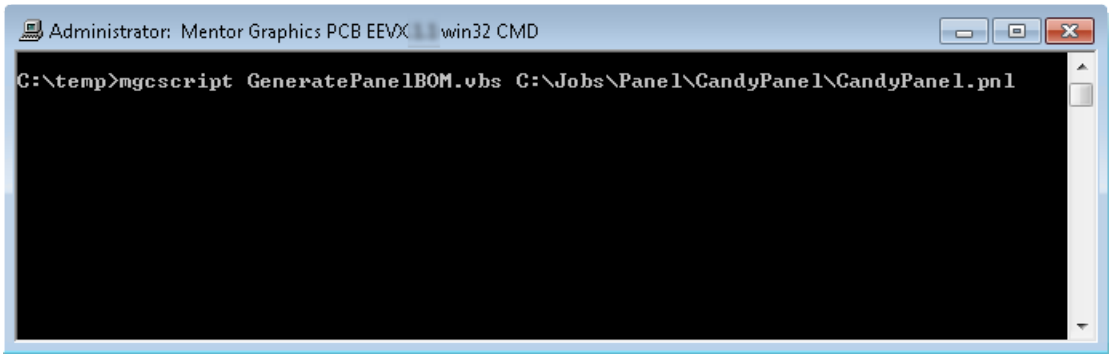
To understand how the script operates perform the following procedure:

1. Launch a console (terminal) window and set its context to *C:\temp* (see "Running Scripts from the Command Line" on page 133 for more details on using console windows)

2. Use the *mgcscript* engine to run the *GeneratePanelBOM.vbs* script on the *CandyPanel.pnl* design. (Refer to Figure 194: Running GeneratePanelBOM.vbs Script).

   The argument to this script is the name of the panel file you want to process (*c:\jobs\Panel\CandyPanel\CandyPanel.pnl*)

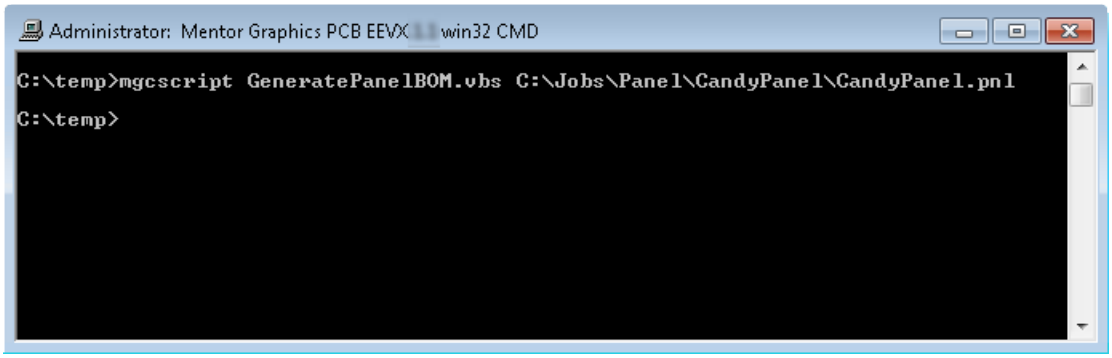
Figure 194. Running GeneratePanelBOM.vbs Script



3. Press the Enter key

   For a few seconds the script gathers the required information and generates the output file. The console window prompt reappears, indicating that the script has completed as illustrated in Figure 195: GeneratePanelBOM.vbs Script Completed.

Figure 195. GeneratePanelBOM.vbs Script Completed



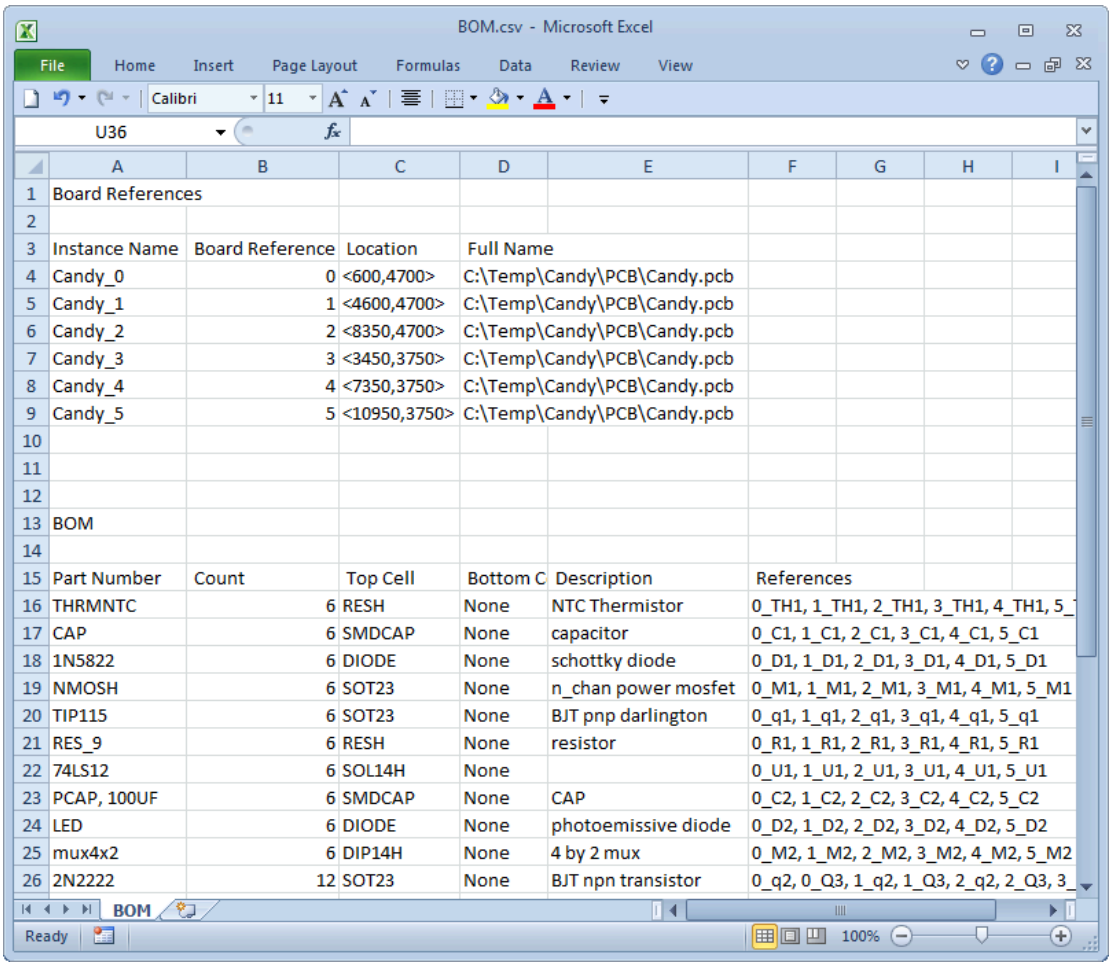
   C:\jobs\Panel\CandyPanel\Output\BOM.csv

4. Launch Excel

5. Choose **File > Open** to display the Open dialog.

6. Set the **Files of type** field to **Text Files (*.prn; *.txt; *.csv)**, and then open the BOM.csv file, the result is as illustrated in Figure 196: Using Excel to View the BOM.csv File.

Figure 196. Using Excel to View the BOM.csv File



	A	B	C	D	E	F
1	Board References					
2						
3	Instance Name	Board Reference	Location	Full Name		
4	Candy_0	0	<600,4700>	C:\Temp\Candy\PCB\Candy.pcb		
5	Candy_1	1	<4600,4700>	C:\Temp\Candy\PCB\Candy.pcb		
6	Candy_2	2	<8350,4700>	C:\Temp\Candy\PCB\Candy.pcb		
7	Candy_3	3	<3450,3750>	C:\Temp\Candy\PCB\Candy.pcb		
8	Candy_4	4	<7350,3750>	C:\Temp\Candy\PCB\Candy.pcb		
9	Candy_5	5	<10950,3750>	C:\Temp\Candy\PCB\Candy.pcb		
10						
11						
12						
13	BOM					
14						
15	Part Number	Count	Top Cell	Bottom C	Description	References
16	THRMNTC	6	RESH	None	NTC Thermistor	0_TH1, 1_TH1, 2_TH1, 3_TH1, 4_TH1, 5_
17	CAP	6	SMDCAP	None	capacitor	0_C1, 1_C1, 2_C1, 3_C1, 4_C1, 5_C1
18	1N5822	6	DIODE	None	schottky diode	0_D1, 1_D1, 2_D1, 3_D1, 4_D1, 5_D1
19	NMOSH	6	SOT23	None	n_chan power mosfet	0_M1, 1_M1, 2_M1, 3_M1, 4_M1, 5_M1
20	TIP115	6	SOT23	None	BJT pnp darlington	0_q1, 1_q1, 2_q1, 3_q1, 4_q1, 5_q1
21	RES_9	6	RESH	None	resistor	0_R1, 1_R1, 2_R1, 3_R1, 4_R1, 5_R1
22	74LS12	6	SOL14H	None		0_U1, 1_U1, 2_U1, 3_U1, 4_U1, 5_U1
23	PCAP, 100UF	6	SMDCAP	None	CAP	0_C2, 1_C2, 2_C2, 3_C2, 4_C2, 5_C2
24	LED	6	DIODE	None	photoemissive diode	0_D2, 1_D2, 2_D2, 3_D2, 4_D2, 5_D2
25	mux4x2	6	DIP14H	None	4 by 2 mux	0_M2, 1_M2, 2_M2, 3_M2, 4_M2, 5_M2
26	2N2222	12	SOT23	None	BJT npn transistor	0_q2, 0_Q3, 1_q2, 1_Q3, 2_q2, 2_Q3, 3_

This file contains two main sections:

- **Board References section** — (at the top) lists the *Instance Names* associated with each board along with their corresponding *Board References* (these are integer values generated by the script), *Locations* (origins), and *Full Name* (the full path name to the board).

- **BOM section** — (at the bottom) lists the *Part Number* for each type of component, the total *Count* for that component, and other data. In the *References* column the script prefixes each component's reference designator with a *Board Reference* number (from the first section of the file) corresponding to the board on which that component resides.

Script Sections

The key sections of this script are:

Constant Definitions	Defines the constants the script uses.
Initialization/Setup	Parses the command line and checks for the required number of arguments (one in this case). Creates the Xpedition FabLink application and then opens the panel design document specified by the argument.
Main Functions and Subroutines	The main functions and subroutines defined in the script.
	– BuildBOMDataTree() Function
	– GenerateBOMFile() Subroutine
Utility Functions and Subroutines	Special "workhorse" routines that are focused on this script.
	– GetCellReferenceName() Function
	– GetPCBDocObj() Subroutine
	– GetPartsEditor() Subroutine
	– ShutdownApps() Subroutine
Helper Functions and Subroutines	Generic "helper" routines that other scripts can use.
	– InitTree() Function
Validate Server	Licenses the document and validates the server. Refer to "Running Your First Script" on page 59 for more details on this function.

This script requires information from three different sources (COM servers), which can be complicated. To simplify things, the script is divided into two main routines:

- BuildBOMDataTree() Function —retrieves and stores the data in a form that facilitates working with the data
- GenerateBOMFile() Subroutine —formats the output

This scheme is advantageous in that it allows easy modification of the output format. In addition, it allows creation of several output routines – each of which generates a different format while only having to read and store the original data a single time (reading the data is the time-consuming part of the process).

The script requires a logical, hierarchical, and easily accessible way to store all of the data. In order to achieve this, the script uses a tree structure, which is created and populated by the BuildBOMDataTree() Function and then used by the GenerateBOMFile() Subroutine.

The tree structure is implemented using multiple *Dictionary* objects. Anything can be stored in the value field of a *Dictionary* object, including other *Dictionary* objects. In the case of this script, the script creates a tree that has the following overall form:

Figure 197. BOM Tree Data Structure

```
<root> - {Top "Level 0" Tree [Dictionary]}

  BoardRefs - { "Level 1" Tree [Dictionary]}
    <board_instance_name> { "Level 2" Tree [Dictionary]}
    XLoc - {Real}
    YLoc - {Real}
    FullName - {String}
        <board_instance_name> { "Level 2" Tree [Dictionary]}
    XLoc - {Real}
    YLoc - {Real}
    FullName - {String}
              :
            etc.

    BOMData - {"Level 1" Tree}
    <part_number> - {"Level 2" Tree [Dictionary]}
        Desc - {String}
        Cells - { "Level 3" Tree [Dictionary]}
    Top - {String}
    Bottom - {String}
            PartRefs - {"Level 3" Tree [Dictionary]}
    <ref-des> - { "Level 4" Tree [Dictionary]}
        XLoc - {Real}
        YLoc - {Real}
        Side - {String}
    <ref-des> - { "Level 4" Tree [Dictionary]}
        XLoc - {Real}
        YLoc - {Real}
        Side - {String}
        :
        etc.
    <part_number> - {"Level 2" Tree [Dictionary]}
        Desc - {String}
        Cells - { "Level 3" Tree [Dictionary]}
    Top - {String}
    Bottom - {String}
            PartRefs - {"Level 3" Tree [Dictionary]}
    <ref-des> - { "Level 4" Tree [Dictionary]}
        XLoc - {Real}
        YLoc - {Real}
        Side - {String}
    <ref-des> - { "Level 4" Tree [Dictionary]}
        XLoc - {Real}
        YLoc - {Real}
        Side - {String}
        :
        etc.
              :
            etc.
```

The root (top level) of the tree contains two sub-trees:

- **BoardRefs** — contains the board reference information.
- **BOMData** — contains the BOM data.

These two sub-trees correspond to the two main sections in the output file as illustrated in Figure 196: Using Excel to View the BOM.csv File. Each of these sub-trees contains their own sub-trees and sub-sub-trees as required.

Constant Definitions

Lines 12 through 23 specify the "key names" that the script uses to reference data in the tree structure.

```
 1  ' This script generates a BOM for a panel.
 2  ' It accounts for the contents of each board
 3  ' by opening Expedition and reading the board
 4  ' information.
 5  ' Author: Toby Rimes
 6
 7  Option Explicit
 8
 9  ' Constants
10
11  ' Key constants for acessing tree
12  Const BOMDATA_KEY = "BOMData"
13  Const BOARDREFMAP_KEY = "BoardRefs"
14  Const CELL_KEY = "Geom"
15  Const DESCRIPTION_KEY = "Desc"
16  Const PARTREFS_KEY = "PartRefs"
17  Const SIDE_KEY = "Side"
18  Const TOP_KEY = "Top"
19  Const BOTTOM_KEY = "Bottom"
20  Const XLOC_KEY = "XLoc"
21  Const YLOC_KEY = "YLoc"
22  Const FULLNAME_KEY = "FullName"
23  Const BOARDREF_KEY = "BoardRef"
```

Note:
The values assigned to these constants is arbitrary, and because they are not printed by the script, you can use anything. For this script, for example, it is as effective to assign numbers to these constants instead of strings. However, if it is desirable for a function to print these key names and values (such a function might be created to display and debug the contents of the tree), it is beneficial to use a descriptive string as is in this example.

Initialization/Setup

On Lines 26, 27, and 28 the script adds the type libraries for Xpedition Layout, Xpedition FabLink, and the Parts Editor.

Note:
It is not necessary to add type libraries for both Xpedition Layout and Xpedition FabLink because the automation interfaces are the same. The reason to add both type libraries is that this is good programming practice.

```
25  ' Add any type libraries to be used.
26  Scripting.AddTypeLibrary("MGCPCB.ExpeditionPCBApplication")
27  Scripting.AddTypeLibrary("MGCPCB.FablinkXEApplication")
28  Scripting.AddTypeLibrary ("MGCPCBLibraries.PartsEditorDlg")
```

On Lines 31 through 36 the script declares global variables and establishes the units the script uses. Observe that the script does not use the name *pcbAppObj* (for a board); instead it uses *pnlAppObj* (for a panel). Similarly, instead of using *pcbDocObj* the script uses *pnlDocObj*.

```
30  ' Global variables
31  Dim pnlAppObj                'Application object
32  Dim pnlDocObj                'Document object
33  ' Units to be used throughout scripts
34  Dim unitsEnum : unitsEnum = epcbUnitMils
35  ' Name of the .pnl file (passed on command line)
36  Dim designNameStr : designNameStr = ""
```

Lines 39 through 45 parse the command line arguments; if there are an incorrect number of arguments a *MsgBox()*, on Line 44, informs the user as to the correct usage for this script (see "Running Scripts from the Command Line" on page 133 for more details on techniques for parsing command line arguments).

```
38  ' Parse the arguments
39  Dim argColl
40  Set argColl = ScriptHelper.Arguments
41  If argColl.Count = 3 Then
42      designNameStr = argColl.item(3)
43  Else
44      MsgBox("Usage: mgcscript GeneratePanelBOM.vbs
                                            <panel_design_name>")
45  End If
```

Line 47 checks that the design name is valid (or that the value is not an empty string).

```
47  If Not designNameStr = "" Then
```

Assuming a valid design name, Line 50 creates a Xpedition FabLink server and Line 51 suppresses any trivial dialogs. Suppressing trivial dialogs allows the script to run as a batch process in a non-interactive manner.

```
49      ' Get the application object.
50      Set pnlAppObj = CreateObject("MGCPCB.FablinkXEApplication")
51      pnlAppObj.Gui.SuppressTrivialDialogs = True
```

Line 54 opens the panel document.

```
53      ' Open the document
54      Set pnlDocObj = pnlAppObj.OpenDocument(designNameStr)
```

Line 57 calls the *ValidateServer()* miscellaneous function to validate the panel document.

```
56      ' License the document
57      ValidateServer(pnlDocObj)
```

Line 60 sets the units.

```
59      ' Set the unitsEnum to match the hard coded values.
60      pnlDocObj.CurrentUnit = unitsEnum
```

Line 64 calls the main BuildBOMDataTree() Function, which opens any relevant Xpedition Layout board design documents, reads any required data, and builds the tree structure.

```
62      ' Collect the data
63      Dim bomDataObj
64      Set bomDataObj = BuildBOMDataTree()
```

Line 67 calls the main GenerateBOMFile() Subroutine, which generates the BOM file in the required format.

```
66      ' Generate the bom file.
67      Call GenerateBOMFile(bomDataObj, pnlDocObj.Path &
                                                "Output/BOM.csv")
```

Line 70 closes the Xpedition FabLink document, and Line 71 sets the *Panel* object to *Nothing*.

```
69      ' Done with Fablink close it
70      Call pnlAppObj.Quit()
71      Set pnlAppObj = Nothing
72
73  End If
```

BuildBOMDataTree() Function

Lines 80 through 187 declare the main routine that opens any relevant Xpedition Layout board documents, reads any required data, and builds the tree structure.

Lines 82 and 83 create the root of the tree structure in the form of a *Dictionary* object.

```
78  ' Creates a data tree containing BOM information and board
79  ' reference information
80  Function BuildBOMDataTree()
81      ' Initialize the data tree.
82      Dim dataTreeObj
83      Set dataTreeObj = CreateObject("Scripting.Dictionary")
```

Lines 86 through 88 call the InitTree() Function to create the two main sub-trees:

- the board references map
- the BOM data

The InitTree() Function takes the parent tree object and the key for the sub-tree and it returns the sub-tree.

```
85      ' Initialise sub trees.
86      Dim boardRefsTreeObj, bomTreeObj
87      Set boardRefsTreeObj = InitTree(dataTreeObj,
                                                BOARDREFMAP_KEY)
88      Set bomTreeObj = InitTree(dataTreeObj, BOMDATA_KEY)
```

Lines 91 and 92 acquire the collection of boards from the panel.

```
 90      ' Get the collection of boards
 91      Dim boardColl
 92      Set boardColl = pnlDocObj.Boards
```

Line 95 uses the *Sort* method to sort the collection by performing a standard alpha-numerical sort on the instance names. The reason to do this is to improve performance when working with a panel containing a heterogeneous collection of boards by grouping boards of the same type together.

```
 94      ' Sort the collection of boards.
 95      Call boardColl.Sort()
```

Lines 97 through 104 instantiate and initialize variables.

```
 97      Dim boardRef : boardRef = 0
 98      Dim fullNameStr : fullNameStr = ""
 99      Dim boardObj, pcbDocObj, pcbAppObj, partsEditorObj, partsDBObj
100      Set pcbDocObj = Nothing
101      Set pcbDocObj = Nothing
102      Set partsEditorObj = Nothing
103      Set partsDBObj = Nothing
104      Set pcbAppObj = Nothing
```

Lines 105 through 173 iterate through the collection of boards. For each board, lines 108 through 113 create a sub-tree corresponding to this board instance using its instance name as the key. Within the sub-tree the script adds the board's location, its full path name, and assigns a unique board reference (an integer number) using the appropriate keys.

```
105      For Each boardObj In boardColl
106
107          ' Create the board reference info.
108          Dim boardTreeObj
109          Set boardTreeObj = InitTree(boardRefsTreeObj,
                                         boardObj.InstanceName)
110          boardTreeObj(XLOC_KEY) = boardObj.OriginX
111          boardTreeObj(YLOC_KEY) = boardObj.OriginY
112          boardTreeObj(FULLNAME_KEY) = boardObj.FullName
113          boardTreeObj(BOARDREF_KEY) = boardRef
```

Observe the statement on Line 110, for example:

```
110          boardTreeObj(XLOC_KEY) = boardObj.OriginX
```

In other scripts containing a *Dictionary* object, the script explicitly used the *Item* property to perform this task. For example:

```
110          boardTreeObj.Item(XLOC_KEY) = boardObj.OriginX
```

Because the *Item* property is the default property of a *Dictionary* object, you can omit it if you choose, and it is done here because it makes the script easier to read and understand.

Note:
Usually in the case of *Collections* and *Collection*-like objects (where *Dictionary* objects can be considered to fall in the category of *Collection*-like objects), the *Item* property is the default and can be omitted. Similarly, in the case of singular objects, *Name* is the default property and can be omitted. Although this is generally the standard, these cases cannot be assumed to be true for all cases.

Line 115 checks if the board document is already open from a previous iteration , and if it is open, does not open it again. This check is a performance enhancement because opening a board document consumes a large amount of time. Lines 116 and 117 instantiates and initializes the Xpedition Layout and Parts Editor servers.

```
114
115         If Not fullNameStr = boardObj.FullName Then
116             Call GetPCBDocObj(pcbAppObj, pcbDocObj, boardObj.FullName)
117             Call GetPartsEditor(partsEditorObj, partsDBObj,
                                            boardObj.FullName)
118             fullNameStr = boardObj.FullName
119         End If
```

Lines 122 and 123 acquire the *Parts* collection from the Parts Editor *PartsDB* object associated with this board.

```
121         ' Get the collection of parts from the parts db
122         Dim pdbPartColl
123         Set pdbPartColl = partsDBObj.Partitions("*").Item(1).Parts
```

There is only one partition for databases associated with a board; this explains the first item in the *Partitions* collection on Line 123.

Lines 126 and 127 acquire the *Parts* collection from the Xpedition Layout *Document* object.

```
125         ' Get all the parts in the pcb design
126         Dim pcbPartColl
127         Set pcbPartColl = pcbDocObj.Parts
```

The *Parts* collection in the Xpedition Layout interface can be visualized as a collection of *References* pointed to by *Components* (parts in the Parts DB have more information associated with them, which is why this script accesses parts in the Parts DB).

Lines 130 through 166 iterate through all of the Xpedition Layout *Part* objects. Line 132, returns the Parts DB *Part* object that corresponds to the current Xpedition Layout *Part* object.

```
129         Dim pcbPartObj, pdbPartObj, partTreeObj, cellTreeObj
130         For Each pcbPartObj In pcbPartColl
131             ' Find the part in the parts db
132             Set pdbPartObj = pdbPartColl.Item(pcbPartObj.Name)
```

Line 135 calls the InitTree() Function to create a new sub-tree (*Dictionary* object) for the current part.

```
134             ' Initialize and/or get part tree for part
135             Set partTreeObj = InitTree(bomTreeObj, pcbPartObj.Name)
```

Line 138 adds a description to the new sub-tree using the appropriate key.

```
137                 ' Set the description
138                 partTreeObj(DESCRIPTION_KEY) = pdbPartObj.Description
```

Line 141 creates a new cell sub-tree under the current part sub-tree. Line 142 calls the GetCellReferenceName() Function to acquire the *top* cell name for the current part and then adds it to the cell sub-tree. Similarly, Line 143 acquires the *bottom* cell name for the current part and adds it to the cell sub-tree.

```
140                 ' Initialize and/or get cell tree for this part
141                 Set cellTreeObj = InitTree(partTreeObj, CELL_KEY)
142                 cellTreeObj(TOP_KEY) = GetCellReferenceName(pdbPartObj,
                                                            epdbCellRefTop)
143                 cellTreeObj(BOTTOM_KEY) = GetCellReferenceName(pdbPartObj,

 epdbCellRefBottom)
```

Lines 146 and 147 create a new part references sub-tree under the current part sub-tree (this sub-tree holds each component instance associated with this part).

```
145                 ' Initialize and/or get ref des tree for part
146                 Dim refDesTreeObj
147                 Set refDesTreeObj = InitTree(partTreeObj, PARTREFS_KEY)
```

Lines 150 and 151 acquire the collection of *Component* objects that reference the current part.

```
150                 Dim compColl
151                 Set compColl = pcbPartObj.Components
152                 Dim compObj, compObjTree
```

Lines 153 through 165 iterate through the collection of components. Line 155 creates a new reference designator sub-tree. Observe that in this case, the key is formed from a concatenation of the current board reference and the current component reference designator (refer to Figure 196: Using Excel to View the BOM.csv File to better visualize this).

```
149                 ' Get all instances of this part.
153                 For Each compObj In compColl
154                    ' Initialize tree for this component
155                    Set compObjTree = InitTree(refDesTreeObj,
                                         boardRef & "_" & compObj.Name)
```

Lines 158 and 159 add the current component's X and Y locations to the reference designator sub-tree.

```
157                    ' Get information for this component
158                    compObjTree(XLOC_KEY) =
                          boardObj.TransformPointX(compObj.PositionX,
                                                   compObj.PositionY)
159                    compObjTree(YLOC_KEY) =
                          boardObj.TransformPointY(compObj.PositionX,
                                                   compObj.PositionY)
```

Observe the *TransformPointX* and *TransformPointY* methods on Lines 158 and 159, respectively. These methods transform coordinates from their location on the original board to their corresponding coordinates on the panel. (Refer to the discussions on the related *TransformPointsArray* method in Exercise 2: Generating a Panel Side View for a more detailed explanation of this "transform" concept.)

Lines 160 through 164 check if this component is on the top of the board and if so adds the string "Top" to the reference designator sub-tree; otherwise adds the string "Bottom" to the sub-tree.

```
160                 If compObj.Side = epcbSideTop Then
161                     compObjTree(SIDE_KEY) = "Top"
162                 Else
163                     compObjTree(SIDE_KEY) = "Bottom"
164                 End If
165             Next
166         Next
```

Line 170 increments the board reference value for the next iteration/board.

```
168         ' Update the board reference
169         ' for the next iteration
170         boardRef = boardRef + 1
171
172     Next
```

Line 175 calls the ShutdownApps() Subroutine, which closes the Xpedition Layout and Parts Editor servers.

```
174     ' Close Expedition and the Parts Editor.
175     Call ShutdownApps(pcbAppObj, partsEditorObj)
```

Lines 178 through 181 explicitly release any references to the Xpedition Layout and Parts Editor servers.

```
177     ' Explicitly release all references
178     Set pcbDocObj = Nothing
179     Set pcbAppObj = Nothing
180     Set partsEditorObj = Nothing
181     Set partsDBObj = Nothing
```

Finally, Line 184 sets the return value from this function to be the full tree (in the form of the root *Dictionary* object).

```
182
183     ' Return tree.
184     Set BuildBOMDataTree = dataTreeObj
185
186 End Function
```

GenerateBOMFile() Subroutine

Lines 191 through 250 declare the main routine that accepts the tree structure and a string containing the name of the CSV to create; it then gathers the required data from the tree, formats it, and generates the output file.

```
188  ' Generates a BOM file from the data in dataTreeObj
189  ' dataTreeObj - Dictionary object (multi level)
190  ' fileNameStr - String
191  Sub GenerateBOMFile(dataTreeObj, fileNameStr)
```

Lines 193 through 196 create the output file (see “Creating Input/Output (I/O) Files” on page 256 for more details on creating, reading, and writing text files).

```
192      ' Create the file and get a text stream.
193      Dim fileSysObj
194      Set fileSysObj = CreateObject("Scripting.FileSystemObject")
195      Dim fileObj
196      Set fileObj = fileSysObj.CreateTextFile(fileNameStr, 1)
```

Line 199 writes a header for the board references section of the output file; Line 200 uses the *WriteBlankLines* method to output a blank line; and Line 201 writes the column headers associated with the board references section.

```
198      ' Write board reference mapping first
199      Call fileObj.WriteLine("Board References")
200      Call fileObj.WriteBlankLines(1)
201      Call fileObj.WriteLine("Instance Name,
                      Board Reference Number, Location, Full Name")
```

Lines 204 and 205 acquire the board references sub-tree.

```
203      ' Get the board refs sub tree.
204      Dim boardRefsTreeObj
205      Set boardRefsTreeObj = dataTreeObj(BOARDREFMAP_KEY)
```

Lines 209 through 213 iterate through all of the keys in the board references sub-tree (each key corresponds to a board instance name). Lines 210 and 211 acquire the board reference sub-tree associated with the current key.

```
207      ' Write the data for each board    instance
208      Dim keyStr
209      For Each keyStr In boardRefsTreeObj.Keys
210          Dim boardTreeObj
211          Set boardTreeObj = boardRefsTreeObj(keyStr)
```

Line 212 writes a single line to the output file; this line contains all of the comma-separated data values of interest (see “Exercise 1: Displaying a Single Routing Layer” on page 283 and “Exercise 6: Generating a Component CSV File” on page 374 for detailed discussions on formatting output strings).

```
212          fileObj.WriteLine(keyStr & "," &
                      boardTreeObj(BOARDREF_KEY) & ""<" &
                      boardTreeObj(XLOC_KEY) & "," &
                      boardTreeObj(YLOC_KEY) & ">"" & "," &
                      boardTreeObj(FULLNAME_KEY))
213      Next
```

Line 215 uses the *WriteBlankLines* method to output three blank lines.

```
215      Call fileObj.WriteBlankLines(3)
```

Line 218 writes a header for the BOM section of the output file; Line 219 uses the *WriteBlankLines* method to output a blank line; and Line 220 writes the column headers associated with the BOM section.

```
217      ' Write BOM data
218      Call fileObj.WriteLine("BOM")
219      Call fileObj.WriteBlankLines(1)
220      Call fileObj.WriteLine("Part Number, Count, Top Cell,
                                Bottom Cell, Description, References")
```

Lines 233 and 234 acquire the BOM data sub-tree.

```
222      ' Get the BOM data sub tree.
223      Dim bomTreeObj
224      Set bomTreeObj = dataTreeObj(BOMDATA_KEY)
```

Lines 227 through 246 iterate through all of the keys in the BOM data sub-tree (each key corresponds to a part name). Lines 228 and 229 acquire the part sub-tree associated with the current key.

```
226      ' Write the data for each part.
227      For Each keyStr In bomTreeObj.Keys
228          Dim partTreeObj
229          Set partTreeObj = bomTreeObj(keyStr)
```

Note:
In the case of the BOM format, this script ignores specific information relating to individual component instances; instead, it counts and lists the component reference designators.However, the data tree does contain instance-specific data, so it is relatively easy to modify this routine, or create a new routine that outputs a format containing this data.

Lines 235 through 242 iterate through all of the keys in the part references sub-sub-tree associated with the current part sub-tree. Line 236 counts this reference designator by incrementing the appropriate variable.

```
231          ' Make a list of ref-des
232          Dim refDesStr
233          Dim refDesCntInt : refDesCntInt = 0
234          Dim refDesKeyStr
235          For Each refDesKeyStr In partTreeObj(PARTREFS_KEY).Keys
236              refDesCntInt = refDesCntInt + 1
```

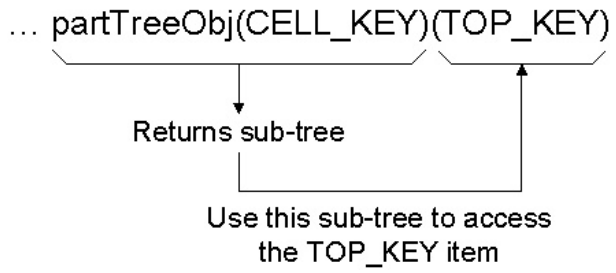
Lines 237 through 241 add the reference designator name to the reference designator string. If this is the first entry in the reference designator string, then Line 238 sets the string to contain this entry; otherwise, Line 240 appends a comma and the new entry to the end of the existing reference designator string.

```
237                 If refDesCntInt = 1 Then
238                     refDesStr = refDesKeyStr
239                 Else
240                     refDesStr = refDesStr & ", " & refDesKeyStr
241                 End If
242             Next
```

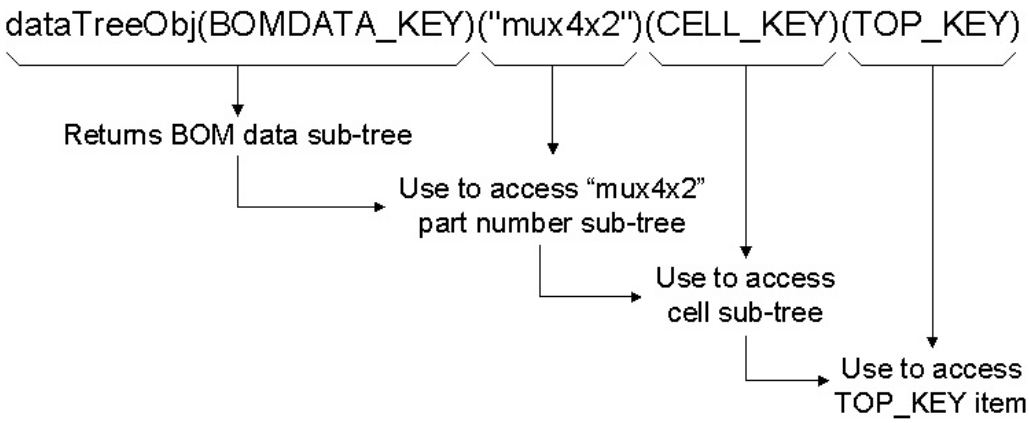
Line 245 writes the BOM line for the current part to the output file; this line contains all of the comma-separated data of interest (Refer to “Exercise 2: Documenting the Layer Stackup” on page 296 and “Exercise 6: Generating a Component CSV File” on page 374 for detailed discussions on formatting output strings in general; also, refer to Exercise 6: Generating a Component CSV File for discussions on generating comma-separated value output strings).

```
244             ' Write the parts line.
245             fileObj.WriteLine("""" & keyStr & """," &
                          refDesCntInt & ","""" &
                          partTreeObj(CELL_KEY)(TOP_KEY) & """,""" &
                          partTreeObj(CELL_KEY)(BOTTOM_KEY) & """,""" &
                          partTreeObj(DESCRIPTION_KEY) & """,""" &
                  refDesStr & """")
246         Next
```

Observe how nested trees and items in trees can be accessed as shown on Line 245; for example, consider the following:



The example starts in the middle of the tree. As an alternative scenario, you can access any item from the top of the tree using the list of keys that lead to that item. For example, if you wanted to get the top cell value for part *mux4x2*, you could do so using the following statement.



Note:
Values can be assigned to locations in the tree using the above syntax.

Line 249 closes the output file.

```
248      ' Close the output file.
249      Call fileObj.Close()
250  End Sub
```

Utility Functions and Subroutines

This section declares utility routines. These routines are similar in context to the generic "helper" routines, except that they are specific to this script and may not be used elsewhere without changes.

GetCellReferenceName() Function

This routine accepts two parameters: a Parts Editor *Part* object and a cell reference type enumerate that specifies "top" or "bottom". Line 263 acquires the *CellReferences* collection associated with the passed in enumerate.

```
256  ' Returns the cell reference for pdbPartObj
257  ' on cellRefTypeEnum side
258  ' pdbPartObj - Part Object (Parts Editor interface)
259  ' cellRefTypeEnum - Enumerate (Parts Editor interface,
260  '                              EPDBCellReferenceType)
261  Function GetCellReferenceName(pdbPartObj, cellRefTypeEnum)
262      Dim cellRefs
263      Set cellRefs = pdbPartObj.CellReferences(cellRefTypeEnum)
```

Line 265 checks if there are any cells in the collection and if there are, Line 266 returns the name of the first item; otherwise, Line 268 returns a string containing the string "None".

```
264
265      If cellRefs.Count > 0 Then
266          GetCellReferenceName = cellRefs.Item(1).Name
267      Else
268          GetCellReferenceName = "None"
269      End If
270  End Function
```

GetPCBDocObj() Subroutine

This routine accepts three parameters: an Xpedition Layout *Application* object, an Xpedition Layout *Document* object, and an Xpedition Layout design document path/filename. In this case the script is using the first two parameters—the *Application* and *Document* objects as bi-directional (in/out) parameters.

```
272  ' Creates an Expedition application if needed.
273  ' Opens a new document, docNameStr.
274  ' pcbAppObj - Application Object (Expedition) (in/out)
275  ' pcbDocObj - Document Object (Expedition) (in/out)
276  ' docNameString - String
277  Sub GetPCBDocObj(pcbAppObj, pcbDocObj, docNameStr)
```

Lines 279 through 281 check if a document is already open and if so, the script closes it.

```
278      ' Close the document if one is open.
279      If Not pcbDocObj Is Nothing Then
280          pcbDocObj.Close()
281      End If
```

Lines 284 through 288 check if the application has already been created and if not creates it (observe that Line 286 suppresses any trivial messages coming from this server).

```
283      ' Create the application if not already created.
284      If pcbAppObj Is Nothing Then
285          Set pcbAppObj =
                     CreateObject("MGCPCB.ExpeditionPCBApplication")
286          pcbAppObj.Gui.SuppressTrivialDialogs = True
287      End If
```

Line 290 opens the document and Line 291 validates it.

```
289      ' Open and license the document
290      Set pcbDocObj = pcbAppObj.OpenDocument(docNameStr)
291      ValidateServer(pcbDocObj)
```

Line 294 sets the units associated with this document and then Line 296 exits this routine.

```
293      ' Set the units.
294      pcbDocObj.CurrentUnit = unitsEnum
295
296  End Sub
```

GetPartsEditor() Subroutine

This routine accepts three parameters: a Parts Editor *PartsEditorDlg* object, a Parts Editor *PartsDB* object, and an Xpedition Layout design document path/filename.

```
298  ' Creates a Parts Editor object and opens a parts database.
299  ' partsEditorObj - PartsEditorDlg Object
300  ' partsDBObj - PartsDB Object
301  ' docNameStr - String
302  Sub GetPartsEditor(partsEditorObj, partsDBObj, docNameStr)
```

Lines 304 through 306 check if there is a *PartsEditorDlg* object and if not creates one.

```
303      ' Open the parts editor
304      If partsEditorObj Is Nothing Then
305          Set partsEditorObj =
                        CreateObject("MGCPCBLibraries.PartsEditorDlg")
306      End If
```

Line 308 opens the required Parts Editor database.

```
308      Set partsDBObj =
                         partsEditorObj.OpenDatabase(docNameStr, True)
309  End Sub
```

ShutdownApps() Subroutine

This routine accepts two parameters: an Xpedition Layout *Application* object and a Parts Editor *PartsEditorDlg* object. Lines 315 and 316 use the *Quit* method to exit both of these objects.

```
311  ' Close Expedition and PartsEditor.
312  ' pcbAppObj - Application Object (Expedition)
313  ' partsEditorObj - PartsEditorDlg Object
314  Sub ShutdownApps(pcbAppObj, partsEditorObj)
315      Call pcbAppObj.Quit()
316      Call partsEditorObj.Quit()
317  End Sub
```

Helper Functions and Subroutines

This script section declares several generic "helper" routines. These routines can be used in other scripts and are not focused on this script exclusively.

InitTree() Function

This routine accepts two parameters: a parent tree (in the form of a *Dictionary* object) and a key string under which to create the new sub-tree.

```
322  ' Create a new sub tree in paretnTreeObj under keyStr
323  ' parentTreeObj - Dictionary Object
324  ' keyStr - String
325  Function InitTree(parentTreeObj, keyStr)
```

Lines 327 through 329 check if a sub-tree already exists under this key and if not creates it.

```
326      ' Create the child tree if it doesn't exist
327      If Not parentTreeObj.Exists(keyStr) Then
328          Set parentTreeObj(keyStr) =
                                 CreateObject("Scripting.Dictionary")
329      End If
```

Line 332 sets the return value from this function to be the previously-existing or newly-created sub-tree.

```
331      ' Return the child tree.
332      Set InitTree = parentTreeObj(keyStr)
333  End Function
```

Creating and Testing the Script

1. Use the scripting editor of your choice to enter the script described and save it out as *GeneratePanelBOM.vbs*.

2. Close any instantiations of Xpedition Layout, Xpedition FabLink, and the Parts Editor that are currently running.

3. Launch a console (terminal) window and set its context to *C:\temp*.

4. Use the *mgcscript* engine to run the *GeneratePanelBOM.vbs* script passing the name of the panel file you want to process (*c:\jobs\Panel\CandyPanel\CandyPanel.pnl* in this example) as a parameter (see "Running Scripts from the Command Line" on page 133 for more details on using console windows).

   After a few seconds, during which the required output files are being generated by the engines called by the script, the console window prompt reappears, thereby indicating that the script has completed.

5. Open the *Output* directory/folder in the Xpedition FabLink *CandyPanel* design and note that it now contains a *BOM.csv* file.

6. Launch Excel and use it to view the contents of this file, and check that everything is as you expect.

Enhancing the Script

There are a number of ways in which you can enhance this script. Some suggestions are as follows (it is a good idea for you to practice your scripting skills by implementing these examples):

- For each application (Xpedition Layout, Xpedition FabLink, and the Parts Editor), modify the script such that it checks if the application is currently running and if it is, make the script attach to the existing process.

- As opposed to generating a CSV file, the script could be modified to automatically launch and populate an appropriate spreadsheet editor, such as Excel on Windows. This allows you to control the formatting of the output, such as varying font sizes, using bold text, and selecting different colors. (Refer to "Exercise 5: Integration with Excel" on page 358.)

Exercise 2: Generating a Panel Side View

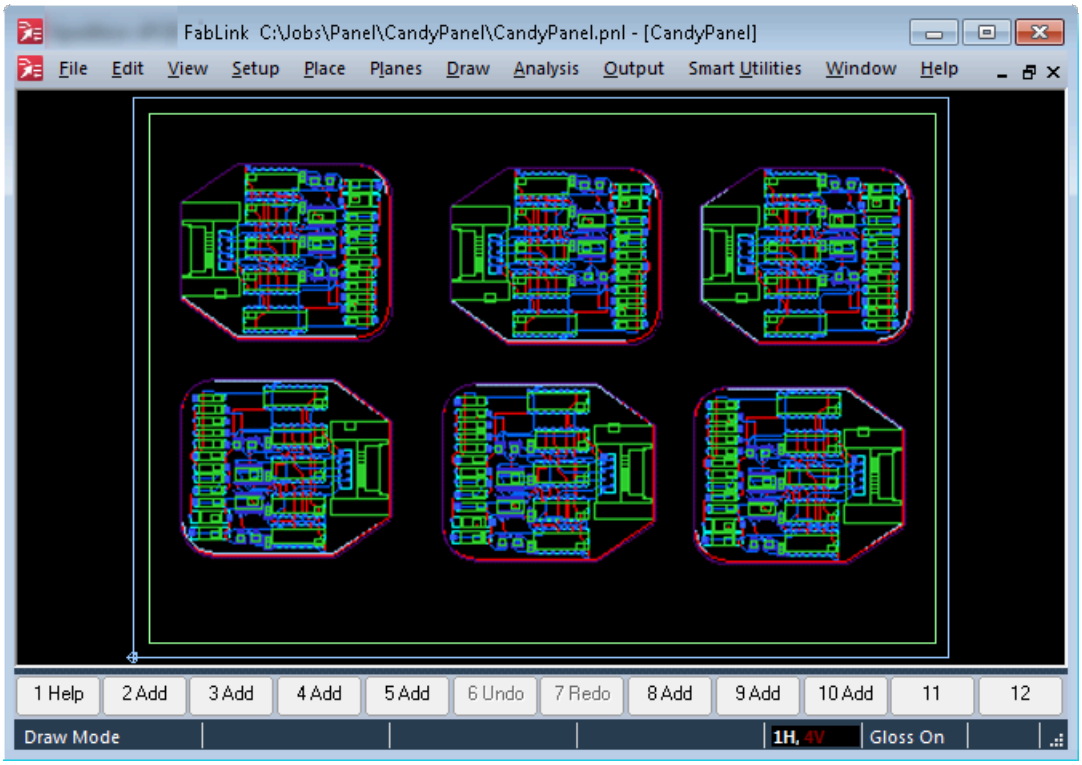
This script creates a graphic representation depicting the side view of a panel.

Approach:	To access board information from Xpedition FabLink, to access component outline and height information from Xpedition Layout, and to create the resulting side view on a user layer in Xpedition FabLink.
Points of Interest:	– Using a Xpedition FabLink script to access data in Xpedition Layout
	– Transforming component positional data (in the form of points arrays) from the board in Xpedition Layout to their corresponding locations on the panel in Xpedition FabLink.
Items Used:	– Xpedition Layout Server
	– TransformPointsArray Method (Xpedition FabLink)

Before Starting

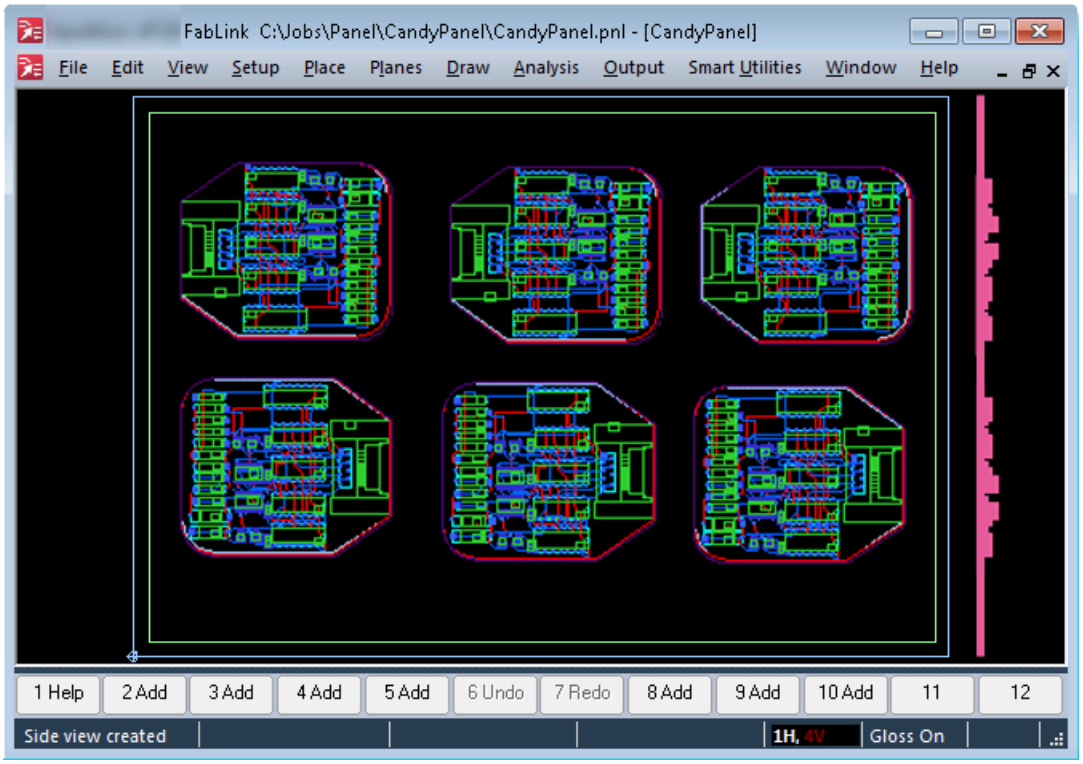
In order to see how this script works, launch Xpedition FabLink and open the *CandyPanel* panel design as illustrated in Figure 198: CandyPanel Panel in Xpedition FabLink.

Figure 198. CandyPanel Panel in Xpedition FabLink



Drag-and-drop the *PanelSideView.vbs* script onto the Xpedition FabLink design area. The result generates a vertical profile of the panel, and presents this profile to the right of the panel ("vertical" as used here refers to the manner in which the panel is presented in the Xpedition FabLink GUI).

Figure 199. Panel After the Script Runs



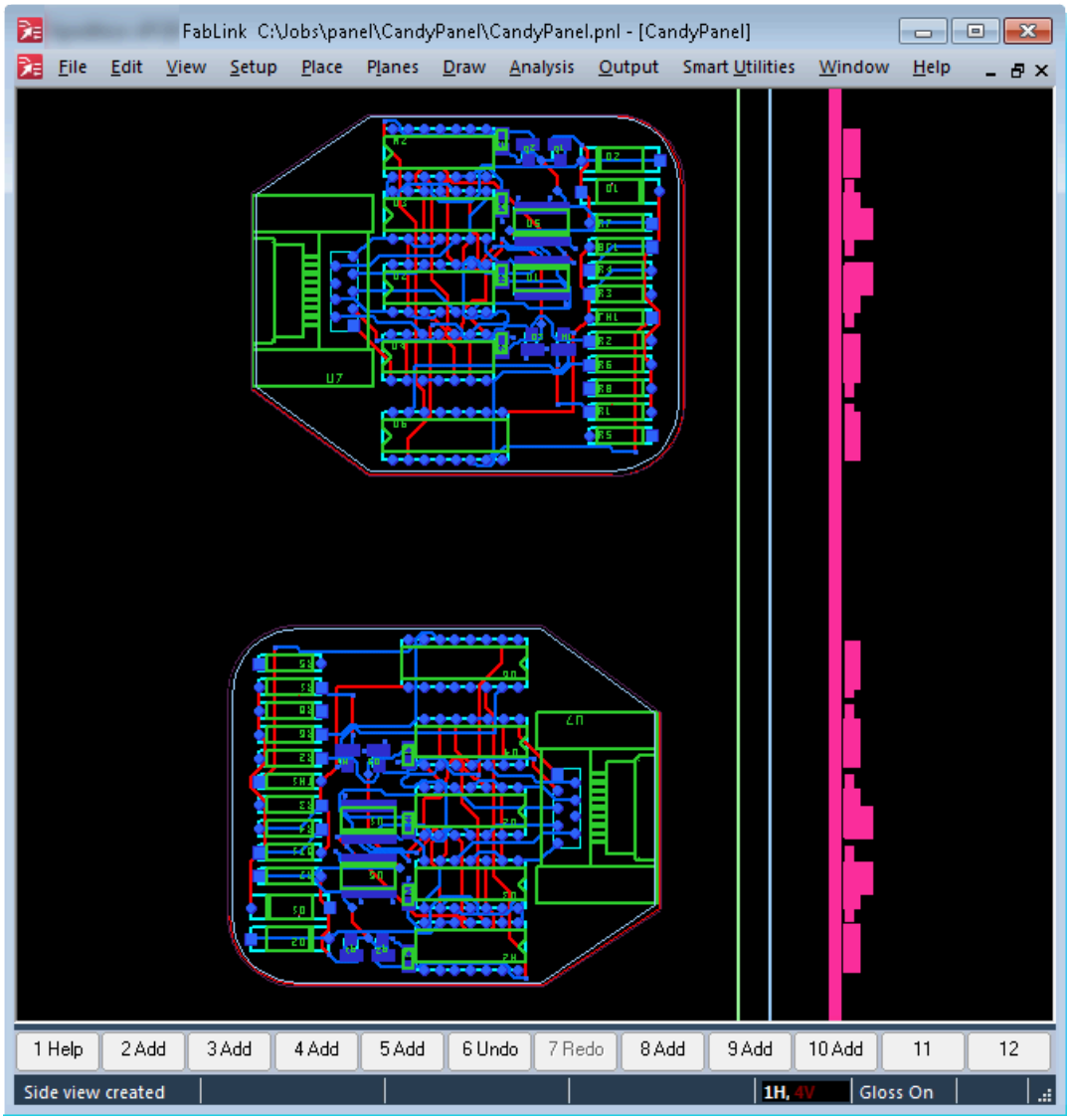
The graphic that runs the height of the panel is a graphical representation of the panel. The small rectangular objects to the right of the panel graphic represent the components on the boards forming the panel. In order to obtain the dimensions of these components, the script acquires this data from the board design in Xpedition Layout.

Zoom in on the right-hand-side of the panel as illustrated in Figure 200: Right-hand Side of the Panel. The components on the top of the panel are presented to the right of the panel.

The *Candy.pcb* boards comprising the *CandyPanel.pnl* panel have components only on their top sides. However, the script is capable of handling components on both the top and bottom sides (any bottom-side components appear to the left of the vertical panel graphical object).

The script is capable of handling both homogeneous panels (comprising multiple copies of the same board design) and heterogeneous panels (comprising a mixture of different boards).

Figure 200. Right-hand Side of the Panel



Script Sections

The key sections of this script are:

Constant Definitions	Defines the constants the script uses.
Initialization/Setup	Add type libraries, declare global variables, and perform any additional setup associated with this script.
Main Functions and Subroutines	The main functions and subroutines defined in the script.
	– CreatePanelSideView() Subroutine
Utility Functions and Subroutines	Special "workhorse" routines that are focused on this script.

	– AddPanel() Subroutine
	– AddComponents() Subroutine
	– AddComponent() Subroutine
	– GetPCBDocObj() Subroutine
Helper Functions and Subroutines	Generic "helper" routines that other scripts can use.
	– GetBoardThinkness() Function
	–ConfigureUserLayer() Function
	– GetYExtrema() Subroutine
Validate Server	Licenses the document and validates the server. Refer to "Running Your First Script" on page 59 for more details on this function.

Constant Definitions

Lines 9 through 13 declare the constants for this script.

```
 1  ' This script creates a side view of the panel
 2  ' on user layer SideView.  It accounts for
 3  ' component heights and board thickness.
 4  ' Author: Toby Rimes
 5
 6  Option Explicit
 7
 8  ' Constants
 9  Const SIDE_VIEW_OFFSET = 900
10  Const SIDE_VIEW_USER_LAYER = "SideView"
11  Const X = 0
12  Const Y = 1
13  Const R = 2
```

Initialization/Setup

Lines 16 and 17 add the type libraries for Xpedition Layout and Xpedition FabLink.

Note:
It is not necessary to add type libraries for both Xpedition Layout and Xpedition FabLink because the automation interfaces are the same. The reason to add both type libraries is that this is good programming practice.

```
15  ' Add any type libraries to be used.
16  Scripting.AddTypeLibrary("MGCPCB.FablinkXEApplication")
17  Scripting.AddTypeLibrary("MGCPCB.ExpeditionPCBApplication")
```

Lines 20 through 23 instantiate and initialize the global variables.

```
19  ' Global variables
20  Dim pnlAppObj                 'Application object
21  Dim pnlDocObj                 'Document object
22  ' Units to be used throughout scripts
23  Dim unitsEnum : unitsEnum = epcbUnitMils
```

Lines 26 through 29 acquire the Xpedition FabLink *Application* and *Document* objects.

```
25  ' Get the application object.
26  Set pnlAppObj = Application
27
28  ' Get the active document
29  Set pnlDocObj = pnlAppObj.ActiveDocument
```

Line 32 validates the document.

```
31  ' License the document
32  ValidateServer(pnlDocObj)
```

Line 35 starts a transaction to group all of the changes into a single **Undo** level.

```
34  ' Start a transaction to group changes for undo
35  Call pnlDocObj.TransactionStart()
```

Line 38 calls the main CreatePanelSideView() Subroutine.

```
37  ' Create the side view.
38  Call CreatePanelSideView()
```

Line 41 ends the transaction.

```
    40  ' End the transaction
41  Call pnlDocObj.TransactionEnd()
```

Finally, Line 44 displays a message in the status bar to inform the user as to the task just performed.

```
43  ' Let the user know we are done
44  pnlAppObj.Gui.StatusBarText("Side view created")
```

CreatePanelSideView() Subroutine

Lines 50 through 90 declare the main routine that generates the panel side view. Lines 52 and 53 acquire the collection of boards from the panel.

```
49  ' Creates the side view on the panel.
50  Sub CreatePanelSideView()
51      ' Get the collection of boards
52      Dim boardColl
53      Set boardColl = pnlDocObj.Boards
```

Line 56 uses the *Sort* method to sort the collection by performing a standard alpha-numerical sort on their instance names. The reason to do this is to improve performance when working with a panel containing a heterogeneous collection of boards by grouping boards of the same type together.

```
55      ' Sort the collection of boards.
56      Call boardColl.Sort()
```

Lines 60 through 65 instantiate and initialize local variables.

```
58      ' Create side views of all the components
59      ' on each board.
60      Dim topSurfaceXReal : topSurfaceXReal = 0
61      Dim bottomSurfaceXReal : bottomSurfaceXReal = 0
62      Dim fullNameStr : fullNameStr = ""
63      Dim pcbDocObj : Set pcbDocObj = Nothing
64      Dim pcbAppObj : Set pcbAppObj = Nothing
65      Dim boardObj
```

Lines 66 through 81 iterate through the collection of boards. Line 68 checks if the current board is already open in Xpedition Layout; if not, Line 69 calls the *GetPCBDocObj()* utility routine to create (launch) Xpedition Layout and to open the board design document.

```
66      For Each boardObj In boardColl
67          ' Open an Expedition document for this board
68          If Not fullNameStr = boardObj.FullName Then
69              Call GetPCBDocObj(pcbAppObj, pcbDocObj,
                                                boardObj.FullName)
70              fullNameStr = boardObj.FullName
71          End If
```

Lines 74 through 76 (if the script has not already done this) call the AddPanel() Subroutine to add the tall thin graphic element representing the panel to the user layer. The way to determine whether or not the script has already added this element is to check the *topSurfaceXReal* variable that was initialized to 0 on Line 60; if this is non-zero, then this means that the script previously called the AddPanel() Subroutine, because this routine assigns a non-zero value to this variable.

```
73          ' Add the board if we haven't already
74          If topSurfaceXReal = 0 Then
75              Call AddPanel(pnlDocObj.PanelOutline,
                    pcbDocObj, topSurfaceXReal, bottomSurfaceXReal)
76          End If
```

After adding the panel, the offset corresponding to the thickness of the panel is known. Line 79 calls the AddComponents() Subroutine to add the side view graphics corresponding to the components on the current board.

```
78          ' Add the components for this board instance
79          Call AddComponents(boardObj, pcbDocObj,
                                topSurfaceXReal, bottomSurfaceXReal)
80
81      Next
```

Note:
The way in which the script implements the loop defined by Lines 66 through 81 is not as performance-efficient as it can be. There is a small modification that you could make to increase the performance of the script. It it left an an exercise for the student. (**Hint:** This relates to the In-Process Client and Out-of-Process Client discussions; refer to Exercise 3: Creating a Via Mask).

Line 84 uses the *Quit* method on Xpedition Layout.

```
 83      ' Close Expedition
 84      pcbAppObj.Quit()
```

Lines 87 and 88 releases all references to the Xpedition Layout *Application* and *Document* objects., and then Line 90 exits the routine.

```
 86      ' Explicitly release all references
 87      Set pcbDocObj = Nothing
 88      Set pcbAppObj = Nothing
 89
 90  End Sub
```

Utility Functions and Subroutines

This section declares utility routines. These routines are similar in context to the generic "helper" routines, except that they are specific to this script and may not be used elsewhere without changes.

AddPanel() Subroutine

Lines 100 through 121 declare the routine that adds the side view graphic element corresponding to the panel to a user layer in Xpedition FabLink. There are four parameters to this routine: a Xpedition FabLink *PanelOutline* object, an Xpedition Layout *Document* object, and the X axis values corresponding to the top and bottom of the side view panel graphic, respectively (these last two parameters are *In-Out* parameters whose values are set by this routine).

```
 95  ' Adds the panel profile to the side view.
 96  ' pnlOutlineObj - PanelOutline Object
 97  ' pcbDocObj - Document Object (Xpedition Layout)
 98  ' topSurfaceXReal - Real (In/Out)
 99  ' bottomSurfaceXReal - Real (In/Out)
100  Sub AddPanel(pnlOutlineObj, pcbDocObj, topSurfaceXReal,
                                                bottomSurfaceXReal)
```

Lines 102 and 103 acquire the extrema (the extreme X and Y values) associated with the panel outline.

```
101      ' Get panel outline extrema
102      Dim extremaObj
103      Set extremaObj = pnlOutlineObj.Extrema
```

Line 107 sets the X value corresponding to the bottom surface of the panel side-view graphic to be a specified offset from the right-hand-side of the main panel (the value of this offset is defined by the *SIDE_VIEW_OFFSET* constant declared on line 9).

```
105      ' Determine the bottom surface of the
106      ' board for the side view.
107      bottomSurfaceXReal = extremaObj.MaxX + SIDE_VIEW_OFFSET
```

Line 110 calculates the X value corresponding of the top surface of the panel side-view graphic using the thickness of the panel.

```
109      ' Determine the top surface from the thickness
110      topSurfaceXReal = bottomSurfaceXReal +
                                    GetBoardThickness(pcbDocObj)
```

Lines 113 and 114 generate a points array to represent the panel side view graphic using the *CreateRectXYR* method (see the "Creating Objects with Geometries" on page 232 for more details on creating and using points arrays; refer to Exercise 2: Documenting the Layer Stackup for more details on using the *CreateRectXYR* method).

```
112      ' Create a rectangle to represent the board.
113      Dim pntsArr
114      pntsArr =
               pnlAppObj.Utility.CreateRectXYR(bottomSurfaceXReal,
                 extremaObj.MinY, topSurfaceXReal, extremaObj.MaxY)
```

Line 118 calls the ConfigureUserLayer() Function to create a new user layer with a specified name (or return an existing layer with that name) and to make the display of this layer active.

```
116      ' Add the side view to the panel.
117      Dim userLayerObj
118      Set userLayerObj = ConfigureUserLayer(SIDE_VIEW_USER_LAYER)
```

Line 119 uses the *PutUserLayerGrfx* method to add the tall, thin rectangular graphic element corresponding to the side view of the panel to the user layer.

```
119      Call pnlDocObj.PutUserLayerGfx(userLayerObj, 0,
                        UBound(pntsArr,2)+1, pntsArr, True, Nothing)
120
121  End Sub
```

AddComponents() Subroutine

Lines 128 through 134 declare the routine that adds the side view graphic elements corresponding to the components on a board to the user layer in Xpedition FabLink. There are four parameters to this routine: a Xpedition FabLink *Board* instance, the Xpedition Layout *Document* object associated with this board, and the X axis values corresponding to the top and bottom of the side view panel graphic, respectively.

```
123  ' Adds the components' profiles to the side view
124  ' boardObj - Board Object
125  ' pcbDocObj - Document Object (Xpedition Layout)
126  ' topSurfaceXReal - Real (In/Out)
127  ' bottomSurfaceXReal - Real (In/Out)
128  Sub AddComponents(boardObj, pcbDocObj, topSurfaceXReal,
                                              bottomSurfaceXReal)
```

Lines 131 through 133 iterate through each component in the Xpedition Layout *Document* object; for each component the script calls the AddComponent() Subroutine, which creates the graphic element associated with this component.

```
129      ' Create a side view for each component.
130      Dim compObj
131      For Each compObj In pcbDocObj.Components
132          Call AddComponent(boardObj, compObj, topSurfaceXReal,
                                                  bottomSurfaceXReal)
133      Next
134  End Sub
```

AddComponent() Subroutine

Lines 141 through 184 declare the routine that creates and displays the side view graphic element(s) corresponding to a particular component on the user layer in Xpedition FabLink. There are four parameters to this routine: a Xpedition FabLink *Board* (panel) object, an Xpedition Layout *Component* object, and the X axis values corresponding to the top and bottom of the side view panel graphic, respectively.

```
136  ' Adds the component's profile to the side view
137  ' boardObj - Board Object
138  ' pcbDocObj - Document Object (Xpedition Layout)
139  ' topSurfaceXReal - Real (In/Out)
140  ' bottomSurfaceXReal - Real (In/Out)
141  Sub AddComponent(boardObj, compObj, topSurfaceXReal,
                                                  bottomSurfaceXReal)
```

Lines 144 through 146 check whether or not this component has been placed and if so, exits the routine (we don’t want to process any unplaced components).

```
143     ' Don't process unplaced components
144     If compObj.Placed = False Then
145        Exit Sub
146     End If
```

Otherwise, Lines 148 and 149 instantiate three local variables.

```
148     Dim heightReal, underSideSpaceReal
149     Dim placeOutlineObj
```

Lines 150 through 183 iterate through all of the placement outlines associated with this component (although it's not common, a component can have multiple placement outlines associated with it; this may occur when a component has an uneven top surface comprising several levels, for example).

```
150     For Each placeOutlineObj In compObj.PlacementOutlines
```

Line 152 uses the *Height* property of the *PlacementOutline* object to determine the height of the placement outline associated with this component.

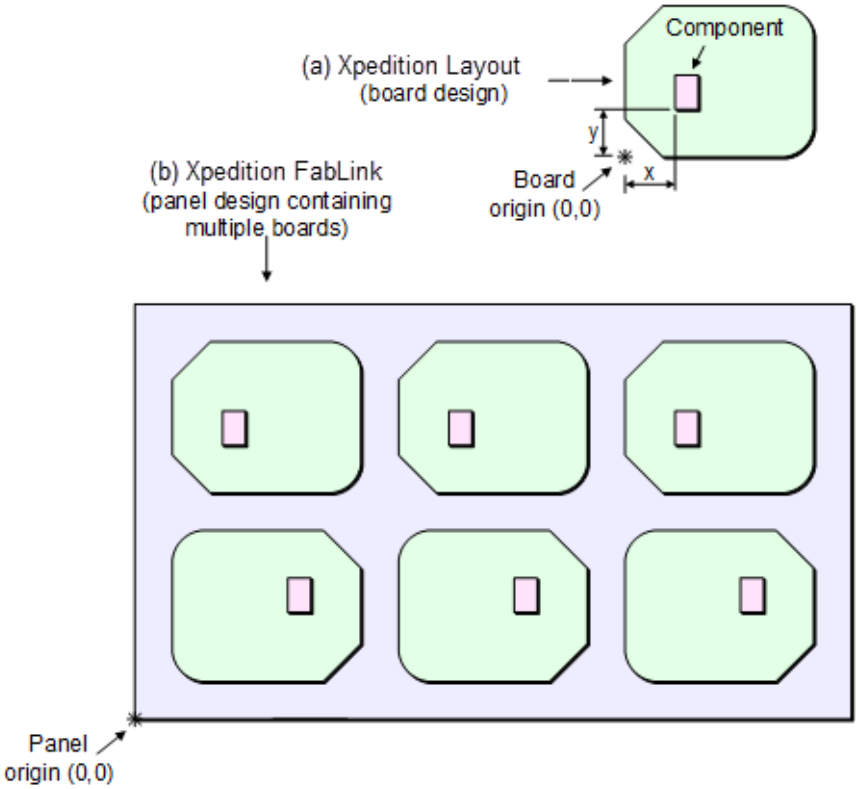
```
151       ' Get the height and underside space
152       heightReal = placeOutlineObj.Height
```

Line 153 uses the *UndersideSpace* property of the *PlacementOutline* object to determine the space (gap) under the component.

```
153        underSideSpaceReal = placeOutlineObj.UndersideSpace
```

Consider a component on a board in Xpedition Layout; the X/Y values specifying the location of the component on the board is relative to the board's origin as illustrated in Figure 201: Transforming values from the board domain to the panel domain (a).

Figure 201. Transforming values from the board domain to the panel domain



However, in order to construct the panel side view graphic associated with a particular component, the script must ransform the coordinates of that component on its local board into its corresponding coordinates on the panel.

This is a non-trivial problem, because in addition to X and Y values, you must account for any rotation of the board on the panel (observe that the bottom three boards on the panel shown in Figure 201: Transforming values from the board domain to the panel domain (b) have been rotated by 180 degrees). The Xpedition FabLink automation interface provides a *TransformPointsArray* method for just this purpose.

Line 160 acquires the points array associated with this component's placement outline.

```
155       ' If 0 height nothing to draw.
156       If Not heightReal = 0 Then
157         ' Translate the points array to
158         ' the boards position on the panel
159         Dim outlinePntsArr
160         outlinePntsArr = placeOutlineObj.Geometry.PointsArray
```

Line 163 uses the *TransformPointsArray* method associated with the Xpedition FabLink *Board* instance object to modify the points array.

```
162         ' Move the points array to its location on the panel.
163         outlinePntsArr =
                boardObj.TransformPointsArray(UBound(outlinePntsArr,
                                                2) + 1, outlinePntsArr)
```

Lines 166 and 167 uses the GetYExtrema() Subroutine to acquire the Y extrema of the transformed component points array.

```
165         ' Get the vertical extrema
166         Dim minYReal, maxYReal
167         Call GetYExtrema(outlinePntsArr, minYReal, maxYReal)
```

Lines 170 through 176, create the graphic elements to represent this component. Line 170 checks if the component is on layer 1 (the top of the board/panel) and if so, Line 172 creates a points array corresponding to the component being on top of the board/panel; otherwise, Line 175 creates a points array corresponding to the component being on the bottom of the board/panel.

```
169         Dim sideViewPntsArr
170         If placeOutlineObj.Layer = 1 Then
171           ' Create the side view in an XYR format
172           sideViewPntsArr =
                  pnlAppObj.Utility.CreateRectXYR(topSurfaceXReal +
                  underSideSpaceReal, minYReal, topSurfaceXReal +
                  heightReal, maxYReal)
173         ElseIf placeOutlineObj.Layer =
                                  pcbDocObj.LayerCount Then
174           ' Create the side view in an XYR format
175           sideViewPntsArr =
                pnlAppObj.Utility.CreateRectXYR(bottomSurfaceXReal –
                heightReal, minYReal, bottomSurfaceXReal –
                underSideSpaceReal, maxYReal)
176         End If
```

Lines 179 through 181 acquire the user layer upon which to display the side view and adds this graphic element to that layer.

```
178         ' Add the component side view to the panel.
179         Dim userLayerObj
180         Set userLayerObj =
                        ConfigureUserLayer(SIDE_VIEW_USER_LAYER)
181         Call pnlDocObj.PutUserLayerGfx(userLayerObj, 0,
                    UBound(sideViewPntsArr,2)+1, sideViewPntsArr,
```

```
                                                                True, Nothing)
182       End If
183    Next
184  End Sub
```

GetPCBDocObj() Subroutine

Lines 191 through 210 declare the routine that acquires an Xpedition Layout *Document* object. There are three parameters to this routine: an Xpedition Layout *Application* object, an Xpedition Layout *Document* object, and a path/filename string (the first two parameters are In-Out parameters whose values are assigned by this routine).

```
186  ' Creates an Expedition application if needed.
187  ' Opens a new document, docNameStr.
188  ' pcbAppObj - Application Object (Expedition) (in/out)
189  ' pcbDocObj - Document Object (Expedition) (in/out)
190  ' docNameString - String
191  Sub GetPCBDocObj(pcbAppObj, pcbDocObj, docNameStr)
```

Lines 193 through 195 check if a document is already open and if so, the script closes it.

```
192      ' Close the document if one is open.
193      If Not pcbDocObj Is Nothing Then
194          pcbDocObj.Close()
195      End If
```

Lines 198 through 201 check if the application has already been created and if not creates it (observe that Line 200 suppresses any trivial messages coming from this server).

```
197      ' Create the application if not already created.
198      If pcbAppObj Is Nothing Then
199          Set pcbAppObj =
                     CreateObject("MGCPCB.ExpeditionPCBApplication")
200          pcbAppObj.Gui.SuppressTrivialDialogs = True
201      End If
```

Line 204 opens the document and Line 205 validates it.

```
203      ' Open and license the document
204      Set pcbDocObj = pcbAppObj.OpenDocument(docNameStr)
205      ValidateServer(pcbDocObj)
```

Line 208 sets the units associated with this document and then Line 210 exits this routine.

```
207      ' Set the units.
208      pcbDocObj.CurrentUnit = unitsEnum
209
210  End Sub
```

Helper Functions and Subroutines

This script section declares several generic "helper" routines. These routines can be used in other scripts and are not focused on this script exclusively.

GetBoardThinkness() Function

Lines 217 through 230 declare the routine that acquires the thickness of the board/panel. The only parameter to this routine is an Xpedition Layout *Document* object.

```
215  ' Returns the total thickness of the board.
216  ' pcbDocObj - Document Object (Xpedition Layout)
217  Function GetBoardThickness(pcbDocObj)
```

Line 220 initializes the return value to 0.

```
219      ' Initialize the return value.
220      GetBoardThickness = 0
```

Lines 225 through 228 iterate through the *Layer* objects summing the thickness of all of the layers and returns the resulting total value.

```
222      ' Iterate through all the layers and add up the
223      ' thickness of each layer.
224      Dim lyrObjObj
225      For Each lyrObjObj In pcbDocObj.LayerStack(True)
226          ' add the layer thickness to the total
227          GetBoardThickness = GetBoardThickness +
                                 lyrObjObj.LayerProperties.Thickness
228      Next
229
230  End Function
```

ConfigureUserLayer() Function

Lines 235 through 253 declare the routine that creates and configures a new user layer (or return an existing layer). The only parameter to this routine is a string that defines the name of the required user layer.

```
232  ' Returns user layer object by this name and turns on display
233  ' of user layer.  If user layer doesn't exist it is created.
234  ' userLayerNameStr - String
235  Function ConfigureUserLayer(userLayerNameStr)
```

Line 238 acquires the user layer if it exists.

```
236      ' See if the user layer already exists.
237      Dim usrLyrObj
238      Set usrLyrObj = pnlDocObj.FindUserLayer(userLayerNameStr)
```

Lines 240 through 246 create the user layer if is does not exist; otherwise it clears any graphics from the layer if the user layer already exists.

```
240      If usrLyrObj Is Nothing Then
241          ' It doesn't exist, create it.
242           Set usrLyrObj =
             pnlDocObj.SetupParameter.PutUserLayer(userLayerNameStr)
243      Else
244          ' It does exist, remove any gfx on the user layer
245          pnlDocObj.UserLayerGfxs(, userLayerNameStr).Delete
246      End If
```

Line 249 ensures that display of the user layer is turned on.

```
248      ' ensure the user layer is turned on
249      pnlDocObj.ActiveViewEx.DisplayControl.UserLayer
                                          (userLayerNameStr) = True
```

Line 252 sets the return value from this routine to be the user layer.

```
251      ' Return the user layer object.
252      Set ConfigureUserLayer = usrLyrObj
253  End Function
```

GetYExtrema() Subroutine

Lines 259 through 275 declare the routine that determines the Y extrema values associated with a points array. This routine accepts three parameters: a points array, the minimum Y extreme, and the maximum Y extreme (the latter two parameters are *In-Out* parameters whose values are assigned by this routine).

```
255  ' Gets the vertical extrema of the points array.
256  ' pntsArr - Array
257  ' minYReal - Real (In/Out)
258  ' maxYReal - Real (In/Out)
259  Sub GetYExtrema(pntsArr, minYReal, maxYReal)
```

Lines 261 and 262 initialize both the minimum and maximum Y extreme values to be the Y value associated with the first point in the points array.

```
261      minYReal = pntsArr(Y, 0)
262      maxYReal = pntsArr(Y, 0)
```

Lines 265 through 273 iterate through all of the points in the points array starting with the second point. If the Y value associated with the current point is *less than* the current minimum Y, then resets the minimum Y to this new value; otherwise, if the new value is *greater than* the current maximum Y, then resets the maximum Y to this new value.

```
264      Dim i
265      For i = 1 To UBound(pntsArr, 2)
266          If pntsArr(Y, i) < minYReal Then
267              minYReal = pntsArr(Y, i)
268          ElseIf pntsArr(Y, i) > maxYReal Then
269              maxYReal = pntsArr(Y, i)
```

```
270          End If
271      Next
272
273  End Sub
```

Creating and Testing the Script

1. Use the scripting editor of your choice to enter the script described (including the license server function) and save it as *PanelSideView.vbs*.

2. Close any instantiations of Xpedition Layout and Xpedition FabLink that are currently running, and then launch a new instantiation of Xpedition FabLink.

3. Open the *CandyPanel.pnl* design.

4. Drag-and-drop the *PanelSideView.vbs* script into the middle of the Xpedition FabLink design area.

5. Observe the side view profile appear to the right of the panel. Zoom on this profile and observe the way the graphic elements are presented.

Enhancing the Script

There are a number of ways in which this script could be enhanced. Some suggestions are as follows (it is a good idea for you to practice your scripting skills by implementing these examples):

- This script generates both the panel and component graphic elements using a default color; you could modify the script to allow the user to select a color. Furthermore, you could allow the user to select one color for the panel and a different color for the components. You could also decide to support different colors for the top-side and bottom-side components. Alternatively, you could support different colors for different types of components … let your imagination roam wild and free!

- The script described in this chapter generates a vertical side view profile, which is displayed to the right of the panel in the Xpedition FabLink GUI. One modification might be to generate a horizontal profile, and to display this profile below the panel in Xpedition FabLink.

- Another modification is to first generate the side view as motion graphics that are attached to the mouse cursor. In this case, the user could move the side view with the mouse and then click the mouse button to place the graphics in their desired position (see Exercise 2: Documenting the Layer Stackup for more details on using motion graphics).

- Now, this one will really test you; the script could be modified to commence by presenting two white lines superimposed on the top of the panel: one splitting the panel in the vertical direction and the other splitting the panel in the horizontal direction. These two lines could be tied to the mouse as motion graphics, appearing like crosshairs on a gun sight (see Exercise 2: Documenting the Layer Stackup for more details on using motion graphics). A vertical side view profile could be presented to the right of the board; a horizontal side view profile could be presented under the board. Both of these profiles correspond to cross-sectional "slices" associated with the current positions of the vertical and horizontal white (motion graphics) lines. Moving the mouse causes the vertical and horizontal cross-sectional profiles to be updated on-the-fly. What do you think? Are you ready for the challenge?

Exercise 3: Creating a Via Mask

Scope: Xpedition Enterprise

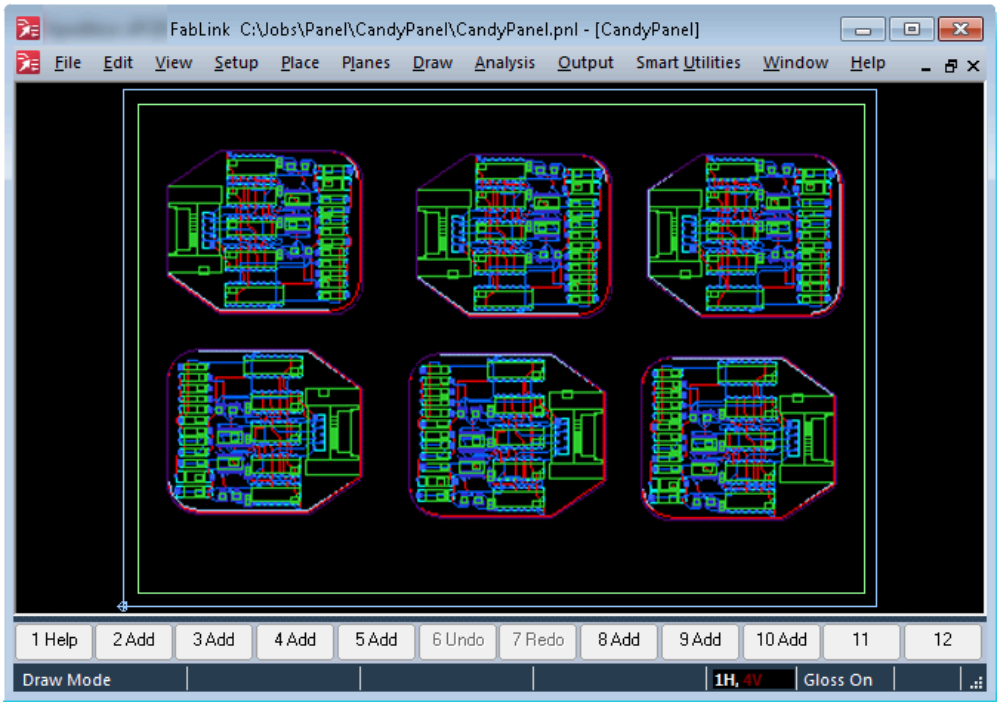
This script creates a mask for the top pads of specific vias on a Xpedition FabLink panel.

Approach:	To access board information from Xpedition FabLink, to access via information from Xpedition Layout, and to use this information to create a mask on a user layer in Xpedition FabLink.
Points of Interest:	– Using a Xpedition FabLink script to access data in Xpedition Layout
	– Transforming via pad data (in the form of points arrays) from the board in Xpedition Layout to their corresponding locations on the panel in Xpedition FabLink.
	– Using the Mask Engine to generate shapes
Items Used:	– Xpedition Layout Server
	– Mask Engine Server
	– TransformPointsArray Method (Xpedition FabLink)
	– AbsorbHoles Method (Mask Engine)

Before Starting

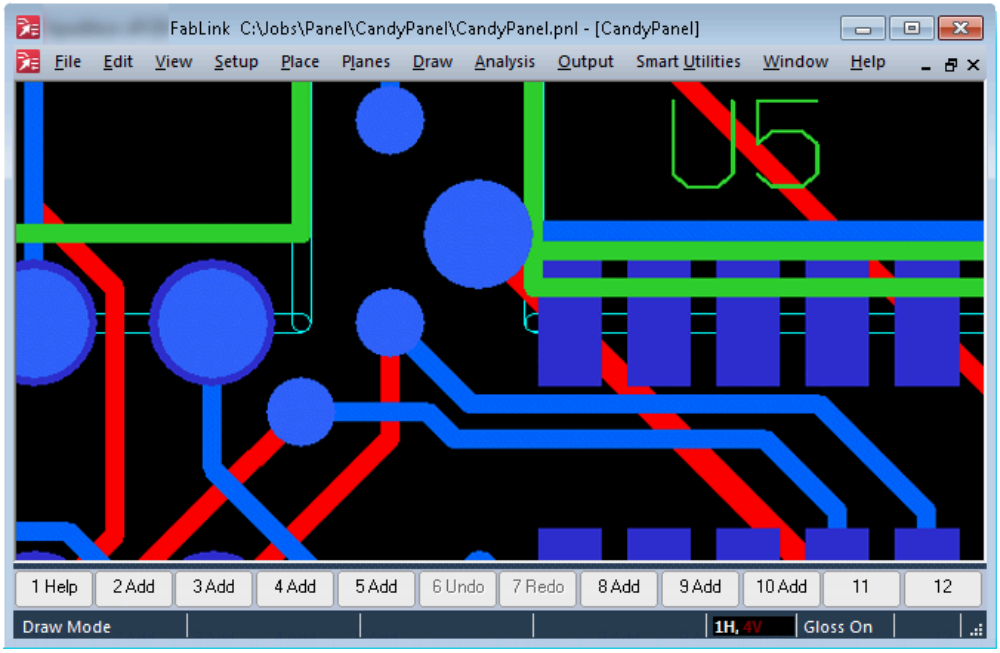
In order to see how this script works, launch Xpedition FabLink and open the *CandyPanel* panel design as illustrated in Figure 202: CandyPanel in Xpedition FabLink.

Figure 202. CandyPanel in Xpedition FabLink



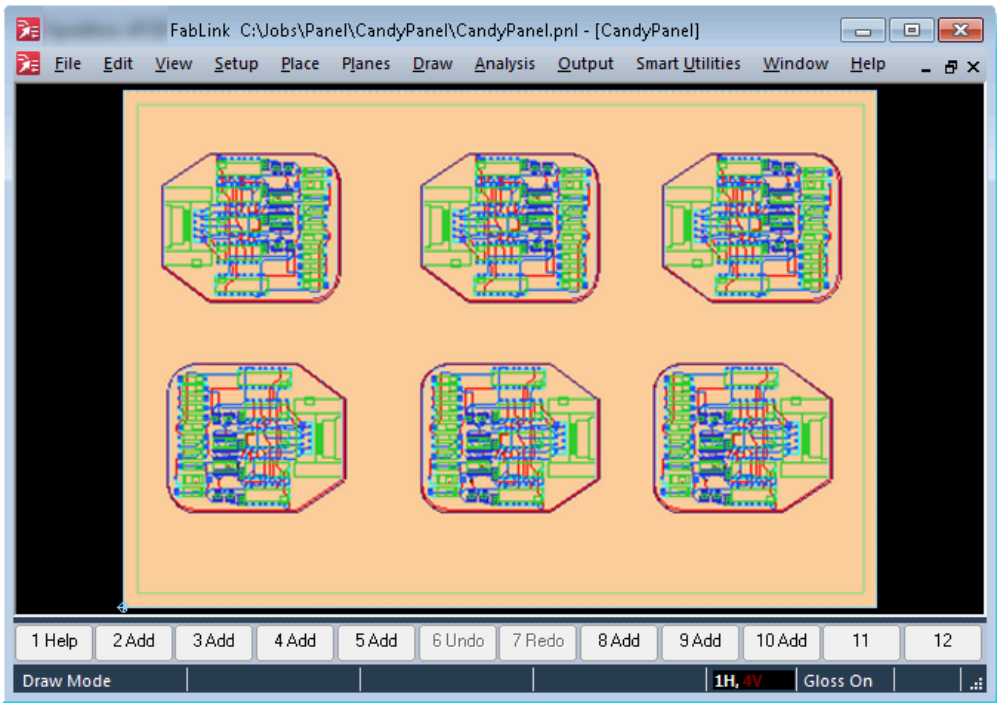
Zoom-in on one of the boards on the panel; the board contains a variety of vias as illustrated in Figure 203: VIA31 Near U5. In particular, observe the via located toward the bottom left-hand corner of component U5. This via uses the padstack of type "VIA31"; also, there are four such vias on each *Candy* board instance on the panel.

Figure 203. VIA31 Near U5



Drag-and-drop the *CreateViaMask.vbs* script into the Xpedition FabLink design area (Figure 204: CandyPanel After Running CreateViaMask.vbs).

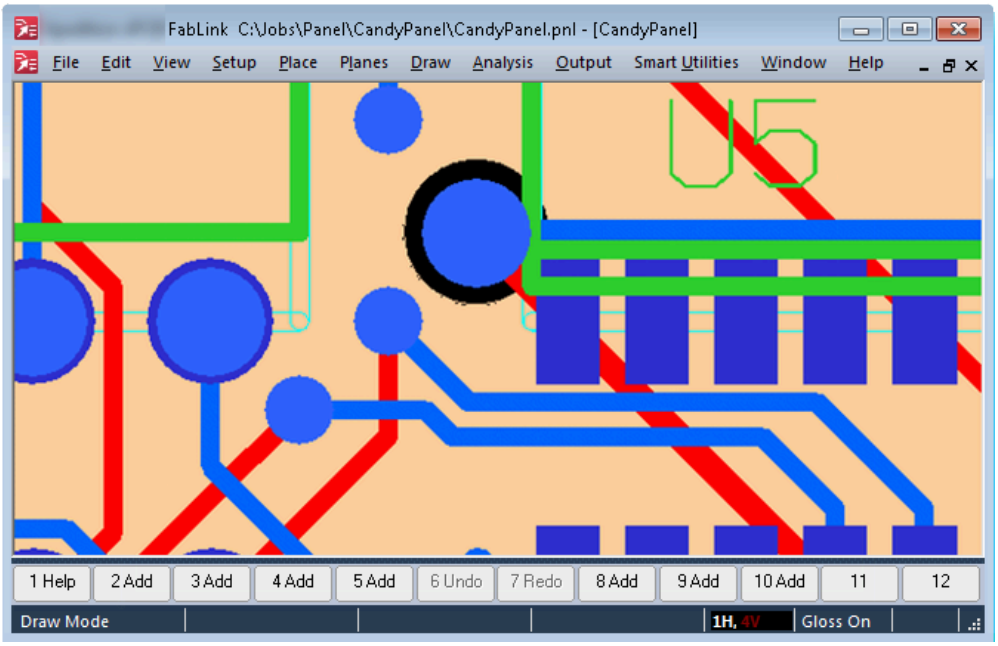
Figure 204. CandyPanel After Running CreateViaMask.vbs



The result is a mask that covers the entire panel except areas around the selected vias (plus a clearance of ten thousandths of an inch).

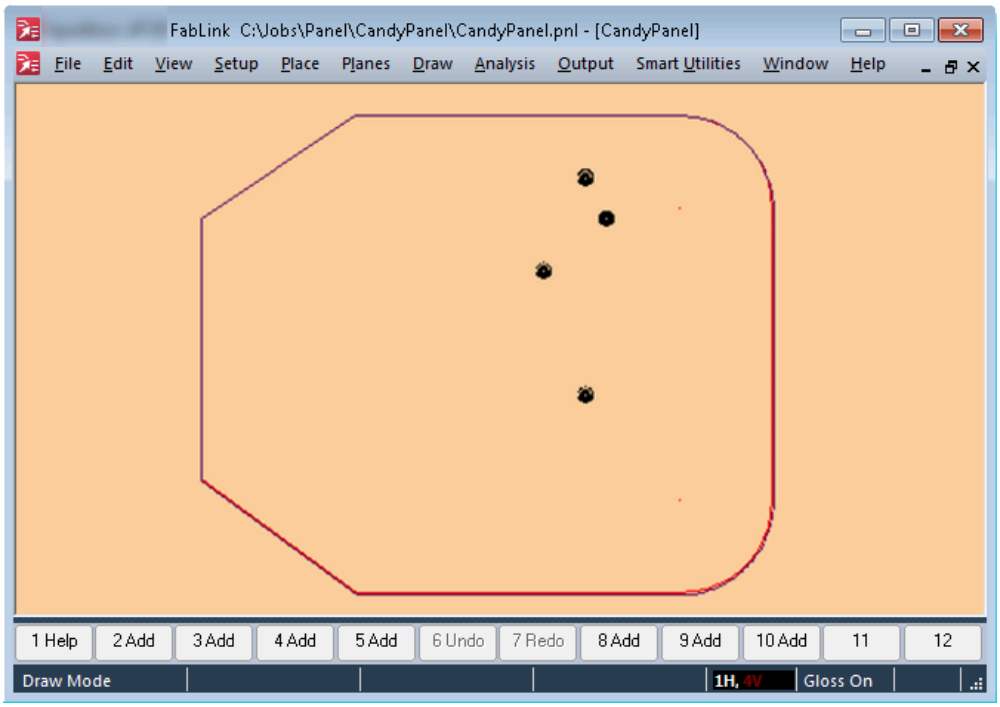
Zoom-in again (Figure 205: Zooming-in on VIA31 Padstack).

Figure 205. Zooming-in on VIA31 Padstack



Disable the display of everything except the mask and zoom-in on a single board instance on the panel. The result is as illustrated in Figure 206: Turning Everything Off Except the Mask (there are four vias that use padstack "VIA31" on each board instance).

Figure 206. Turning Everything Off Except the Mask



Script Sections

The key sections of this script are:

Constant Definitions	Defines the constants the script uses.
Initialization/Setup	Add type libraries, declare global variables, and perform any additional setup associated with this script.
Main Functions and Subroutines	The main functions and subroutines defined in the script.
	– CreateViaMask() Subroutine
Utility Functions and Subroutines	Special "workhorse" routines that are focused on this script.
	– AddViasMask() Subroutine
	– GetPCBDocObj() Subroutine
Helper Functions and Subroutines	Generic "helper" routines that other scripts can use.
	– ConfigureUserLayer() Function
	– AddMaskToUserLayer() Subroutine
Validate Server	Licenses the document and validates the server. Refer to "Running Your First Script" on page 59 for more details on this function.

Constant Definitions

Lines 9 through 13 declare the constants for this script.

```
 1  ' This script creates a side view of the panel
 2  ' on user layer SideView.  It accounts for
 3  ' component heights and board thickness.
 4  ' Author: Toby Rimes
 5
 6  Option Explicit
 7
 8  ' Constants
 9  Const VIA_PADSTACK = "VIA31"
10  Const OVERSIZE = 10
11  Const TOP = 1
12  Const BOTTOM = 2
13  Const USER_LAYER = "Via Mask"
```

Initialization/Setup

Lines 16 through 18 add the type libraries for Xpedition FabLink, Xpedition Layout, and the Mask Engine.

Note:
It is not necessary to add type libraries for both Xpedition Layout and Xpedition FabLink because the automation interfaces are the same. The reason to add both type libraries is that this is good programming practice.

```
15  ' Add any type libraries to be used.
16  Scripting.AddTypeLibrary("MGCPCB.FablinkXEApplication")
17  Scripting.AddTypeLibrary("MGCPCB.ExpeditionPCBApplication")
18  Scripting.AddTypeLibrary("MGCPCBEngines.MaskEngine")
```

Lines 21 through 30 instantiate and initialize the global variables and acquire the Xpedition FabLink *Application* and *Document* objects.

```
20  ' Global variables
21  Dim pnlAppObj                 'Application object
22  Dim pnlDocObj                 'Document object
23  ' Units to be used throughout scripts
24  Dim unitsEnum : unitsEnum = epcbUnitMils
25
26  ' Get the application object.
27  Set pnlAppObj = Application
28
29  ' Get the active document
30  Set pnlDocObj = pnlAppObj.ActiveDocument
```

Line 33 validates the document.

```
32  ' License the document
33  ValidateServer(pnlDocObj)
```

Line 36 starts a transaction to group all of the changes into a single **Undo** level.

```
35  ' Start a transaction to group changes for undo
36  Call pnlDocObj.TransactionStart()
```

Line 39 calls the main CreateViaMask() Subroutine.

```
38  ' Create the mask for top pads of vias with specificed padstack
39  Call CreateViaMask(VIA_PADSTACK, TOP)
```

Line 41 ends the transaction.

```
41  ' End the transaction
42  Call pnlDocObj.TransactionEnd()
```

Finally, Line 45 displays a message in the status bar to inform the user as to the task just performed.

```
44  ' Let the user know we are done
45  pnlAppObj.Gui.StatusBarText("Via mask created")
```

CreateViaMask() Subroutine

Lines 53 through 115 declare the main routine that generates the via mask. This routine accepts two parameters: a string defining the padstack name of interest and an integer, 1 or 2, that specifies the top or bottom of the board, respectively.

```
50  ' Creates the side view on the panel.
51  ' pastackNameStr - String
52  ' boardSideConst - Constant (TOP,BOTTOM)
53  Sub CreateViaMask(padstackNameStr, boardSideConst)
```

Lines 55 and 56 create the Mask Engine. Line 57 specifies the units to use with the Mask Engine.

```
54      ' Create the mask engine
55      Dim maskEngObj
56      Set maskEngObj = CreateObject("MGCPCBEngines.MaskEngine")
57      maskEngObj.CurrentUnit = unitsEnum
```

Each server has its own library of enumerates. Earlier in this script (on Line 24) the script assigns the *epcbUnitMils* enumerate from the Xpedition Layout type library to the *unitsEnum* variable. Line 57 associates the value of this variable with the Mask Engine.

Generally, it is not recommended to use enumerates defined in one type library with a different server. However, it is OK to use enumerates associated with units in any of the servers, because these enumerates are defined identically across all servers, and this allows you to use a single variable to specify the units for multiple servers.

Lines 60 and 61 create a mask to hold the panel outline.

```
59      ' Create a mask to hold the panel outline.
60      Dim panelMaskObj
61      Set panelMaskObj = maskEngObj.Masks.Add()
```

Lines 64 and 65 create a mask to hold the outlines of all of the vias of interest.

```
63      ' Create a mask to hold the via outlines.
64      Dim viaMaskObj
65      Set viaMaskObj = maskEngObj.Masks.Add()
```

Lines 52 and 53 acquire the collection of boards from the panel.

```
67      ' Get the collection of boards
68      Dim boardColl
69      Set boardColl = pnlDocObj.Boards
```

Line 56 uses the *Sort* method to sort the collection by performing a standard alpha-numerical sort on their instance names. The reason to do this is to improve performance when working with a panel containing a heterogeneous collection of boards by grouping boards of the same type together.

```
71      ' Sort the collection of boards.
72      Call boardColl.Sort()
```

Lines 75 through 78 instantiate and initialize local variables.

```
74      ' Create mask for all vias on each board.
75      Dim fullNameStr : fullNameStr = ""
76      Dim pcbDocObj : Set pcbDocObj = Nothing
77      Dim pcbAppObj : Set pcbAppObj = Nothing
78      Dim boardObj
```

Lines 79 through 89 iterate through the collection of boards. Line 81 checks if the current board is already open in Xpedition Layout; if not, Line 82 calls the GetPCBDocObj() Subroutine to create (launch) Xpedition Layout and to open the board design document. Line 83 resets the name variable to be that of the current board.

```
79      For Each boardObj In boardColl
80          ' Open an Expedition document for this board
81          If Not fullNameStr = boardObj.FullName Then
82              Call GetPCBDocObj(pcbAppObj, pcbDocObj,
                                                boardObj.FullName)
83              fullNameStr = boardObj.FullName
```

Line 87 calls the AddViasMask() Subroutine to add the via pad outlines associated with the current board instance to the vias mask.

```
85              ' Add the via shapes to this mask
86              ' for all boards of this type
87              Call AddViasMask(boardColl, pcbDocObj,
                       padstackNameStr, boardSideConst, viaMaskObj)
88          End If
89      Next
```

Note that the loop between lines 79 and 89 is similar to the loop used in the script in Exercise 2: Generating a Panel Side View, but there is a significant difference. Consider the snippet from Exercise 2: Generating a Panel Side View as follows:

```
65  ' Loop from Generating a Panel Side View
66  For Each boardObj In boardColl
67          ' Open an Expedition document for this board
68          If Not fullNameStr = boardObj.FullName Then
69             Call GetPCBDocObj(pcbAppObj, pcbDocObj,
70                                                boardObj.FullName)
70                fullNameStr = boardObj.FullName
71          End If
              :
77
78          ' Add the components for this board instance
79          Call AddComponents(boardObj, pcbDocObj, topSurfaceXReal,
                                                bottomSurfaceXReal)
80
81  Next
```

In this snippet, the *AddComponents()* utility routine (on Line 79) is called for every board instance (that is, for each iteration of the loop) and the first parameter to this routine is the Xpedition FabLink *Board* instance object.

By comparison, consider the corresponding loop in the current script:

```
     ' Loop from Creating a Via Mask
 79      For Each boardObj In boardColl
 80          ' Open an Expedition document for this board
 81          If Not fullNameStr = boardObj.FullName Then
 82              Call GetPCBDocObj(pcbAppObj, pcbDocObj,
                                                boardObj.FullName)
 83              fullNameStr = boardObj.FullName
 85              ' Add the via shapes to this mask
 86              ' for all boards of this type
 87              Call AddViasMask(boardColl, pcbDocObj,
                       padstackNameStr, boardSideConst, viaMaskObj)
 88          End If
 89      Next
```

In this case, the AddViasMask() Subroutine (on Line 87) is only called the first time the board is opened; in addition, as opposed to the single *Board* object used in the previous script, the first parameter to this routine is the collection of *Board* instances on the Xpedition FabLink panel.

The AddViasMask() Subroutine reads the data from Xpedition Layout a single time for all boards of the same type. This new script architecture has significantly better performance to the script in Exercise 3: Creating a Via Mask, because communicating with and accessing data from an Out-of-Process Client is much more time-consuming than working with an In-Process Client.

Line 84 uses the *Quit* method to close Xpedition Layout.

```
 91      ' Close Expedition
 92      pcbAppObj.Quit()
```

Lines 95 and 96 release all references to the Xpedition Layout *Application* and *Document* objects.

```
 94      ' Explicitly release all references
 95      Set pcbDocObj = Nothing
 96      Set pcbAppObj = Nothing
```

Line 100 acquires the points array for the panel outline. Line 101 adds this shape to the panel mask.

```
 98      ' Put the panel outline into a mask
 99      Dim pntsArr
100      pntsArr = pnlDocObj.PanelOutline.Geometry.PointsArray
101      Call panelMaskObj.Shapes.AddByPointsArray(UBound(pntsArr,
                                                  2) + 1, pntsArr)
```

Line 104 uses the *Oversize* method of the *ViaMask* object to increase the diameter of all of the via shapes in the via mask (this is to add the clearance required).

```
103      ' Grow the via mask
104      Call viaMaskObj.Oversize(OVERSIZE)
```

Lines 107 and 108 create a new mask to hold the result from the operation to perform.

```
106      ' Create a mask to store the result it.
107      Dim resultMaskObj
108      Set resultMaskObj = maskEngObj.Masks.Add()
```

Line 111 uses the *BooleanOp* method of the *MaskEngine* object to subtract the via mask from the panel mask.

```
110      ' Subtract the via mask from the panel outline mask.
111      Set resultMaskObj =
                        maskEngObj.BooleanOp(emeBooleanOpSubtract,
                                             panelMaskObj, viaMaskObj)
```

Line 114 calls the AddMaskToUserLayer() Subroutine to place the resulting mask graphics on the designated user layer.

```
113      ' Add the result to the panel.
114      Call AddMaskToUserLayer(pnlDocObj, USER_LAYER,resultMaskObj)
115
116  End Sub
```

Utility Functions and Subroutines

This section declares utility routines. These routines are similar in context to the generic "helper" routines, except that they are specific to this script and may not be used elsewhere without changes.

AddViasMask() Subroutine

Lines 127 through 169 declare the routine that adds the vias to the via mask. This routine accepts five parameters: the collection of *Board* instances on the Xpedition FabLink panel, the Xpedition Layout *Document* object containing the vias, the padstack name (as a string) associated with the pads to create masks, the side of the board of interest (1 = top, 2 = bottom), and the via mask (as an *In-Out* parameter).

```
121  ' Adds the vias to the via mask.
122  ' boardColl - Boards Collection
123  ' pcbDocObj - Document Object (Xpedition Layout)
124  ' padstackNameStr - String
125  ' boardSideConst - Constant (TOP,BOTTOM)
126  ' viaMaskObj - Mask Object (In/Out)
127  Sub AddViasMask(boardColl, pcbDocObj, padstackNameStr,
                                        boardSideConst, viaMaskObj)
```

Lines 131 through 136 resolve the board side of interest (top or bottom) to a layer number on the board.

```
129      ' Determine which side of the board we
130      ' are interested in.
131      Dim layerInt
132      If boardSideConst = BOTTOM Then
133          layerInt = pcbDocObj.LayerCount
134      Else
135          layerInt = 1
136      End If
```

Note:
Refer to "Exercise 4: Removing Unconnected Pads from Pins and Vias" on page 335 to review discussions on the differences/relationships between *Padstack* references and *PadstackObject* objects and the way in which they are associated with pins and vias.

Lines 140 through 168 iterate through all of the *Padstack* references used on the current board.

```
138      ' Process padstacks with this name.
139      Dim pstkObj
140      For Each pstkObj In pcbDocObj.Padstacks
```

Line 141 checks if this *Padstack* reference is the one of interest (recall that there may be multiple *Padstack* references with the same name if a padstack has been modified using the Padstack Processor).

```
141        If pstkObj.Name = padstackNameStr Then
```

If this *Padstack* reference *is* the padstack of interest then Lines 144 through 166 iterate through all of the vias that reference this padstack.

```
142          Dim viaObj
143          ' Process vias using this padstack
144          For Each viaObj In pstkObj.Vias
```

Lines 145 and 146 return the collection of pads for the current via on the desired layer (top or bottom).

```
145          Dim padColl
146          Set padColl = viaObj.Pads(layerInt)
```

Line 149 checks if there are any pads in this collection and if so, Lines 151 through 164 iterate through all of the geometries for this pad.

```
147          ' Process geometries of pads
148          ' on specified layer
149          If padColl.Count > 0 Then
150            Dim geomObj
151            For Each geomObj In padColl.Item(1).Geometries
```

Lines 152 and 153 acquire the points array associated with the current geometry.

```
152              Dim pntsArr
153              pntsArr = geomObj.PointsArray
```

Lines 156 through 163 iterate through all of the *Board* instances in the Xpedition FabLink *Board* collection.

```
154              ' Transform points array for board of each type
155              Dim boardObj
156              For Each boardObj In boardColl
```

Line 157 assures that the current *Board* instance is associated with the Xpedition Layout *Document* object that is passed into this routine.

```
157                 If boardObj.FullName = pcbDocObj.FullName Then
```

For each *Board* instance of the current type, Line 159 uses the *TransformPointsArray* method associated with the Xpedition FabLink *Board* instance object to modify the points array (refer to Exercise 1: Generating a BOM for more discussions on the *TransformPointsArray* method).

```
158                 Dim trasformPntsArr
159                 trasformPntsArr =
                        boardObj.TransformPointsArray(UBound(pntsArr,
                                                        2) + 1, pntsArr)
```

Line 161 adds the transformed points array to the via mask.

```
160                 ' Add the pad shape to the mask.
161                 Call viaMaskObj.Shapes.AddByPointsArray(
                                    UBound(trasformPntsArr, 2) + 1,
                                                        trasformPntsArr)
```

Lines 162 through 168 terminate the various control statements and then Line 169 exits the routine.

```
162               End If
163               Next
164             Next
165             End If
166           Next
167         End If
168       Next
169   End Sub
```

GetPCBDocObj() Subroutine

Lines 191 through 210 declare the routine that acquires an Xpedition Layout *Document* object. There are three parameters to this routine: an Xpedition Layout *Application* object, an Xpedition Layout *Document* object, and a path/filename string (the first two parameters are In-Out parameters whose values are assigned by this routine).

```
171   ' Creates an Expedition application if needed.
172   ' Opens a new document, docNameStr.
173   ' pcbAppObj - Application Object (Expedition) (in/out)
174   ' pcbDocObj - Document Object (Expedition) (in/out)
175   ' docNameString - String
176   Sub GetPCBDocObj(pcbAppObj, pcbDocObj, docNameStr)
```

Lines 178 through 180 check if a document is already open and if so, the script closes it.

```
177       ' Close the document if one is open.
178       If Not pcbDocObj Is Nothing Then
179           pcbDocObj.Close()
180       End If
```

Lines 183 through 186 check if the application has already been created and if not creates the application. Note that Line 185 suppresses any trivial messages coming from the server.

```
182      ' Create the application if not already created.
183      If pcbAppObj Is Nothing Then
184          Set pcbAppObj =
                     CreateObject("MGCPCB.ExpeditionPCBApplication")
185          pcbAppObj.Gui.SuppressTrivialDialogs = True
186      End If
```

Line 189 opens the document and Line 190 validates it.

```
188      ' Open and license the document
189      Set pcbDocObj = pcbAppObj.OpenDocument(docNameStr)
190      ValidateServer(pcbDocObj)
```

Line 193 sets the units associated with this document and then Line 195 exits this routine.

```
192      ' Set the units.
193      pcbDocObj.CurrentUnit = unitsEnum
194
195  End Sub
```

Helper Functions and Subroutines

This script section declares several generic "helper" routines. These routines can be used in other scripts and are not focused on this script exclusively.

ConfigureUserLayer() Function

Lines 203 through 221 declare the routine that creates and configures a new user layer (or returns an existing layer). The only parameter to this routine is a string that defines the name of the required user layer.

```
200  ' Returns user layer object by this name and turns on display
201  ' of user layer.  If user layer doesn't exist it is created.
202  ' userLayerNameStr - String
203  Function ConfigureUserLayer(userLayerNameStr)
```

Lines 205 and 206 acquire the user layer if it exists.

```
204      ' See if the user layer already exists.
205      Dim usrLyrObj
206      Set usrLyrObj = pnlDocObj.FindUserLayer(userLayerNameStr)
```

Lines 208 through 214 create the user layer if is does not exist; otherwise it clears any graphics from the layer if it already exists.

```
208      If usrLyrObj Is Nothing Then
209          ' It doesn't exist. Create it.
210          Set usrLyrObj =
             pnlDocObj.SetupParameter.PutUserLayer(userLayerNameStr)
```

```
211      Else
212          ' It does exist, remove any gfx on the user layer
213          pnlDocObj.UserLayerGfxs(, userLayerNameStr).Delete
214      End If
```

Line 217 ensures that display of the user layer is turned on.

```
216      ' ensure the user layer is turned on
217      pnlDocObj.ActiveViewEx.DisplayControl.UserLayer
                                          (userLayerNameStr) = True
```

Line 220 sets the return value from this routine to be the user layer.

```
219      ' Return the user layer object.
220      Set ConfigureUserLayer = usrLyrObj
221  End Function
```

AddMaskToUserLayer() Subroutine

Lines 227 through 246 declare the routine that adds the contents of a specified mask to a specified user layer. This routine accepts three parameters: a *Document* object (this can be an Xpedition Layout document or a Xpedition FabLink document), the name (as a string) of the user layer, and the *Mask* object to add to the user layer.

```
223  ' Adds the contents of a mask to a user layer.
224  ' docObj - Document Object (Xpedition FabLink or Xpedition Layout)
225  ' userLayerNameStr - String
226  ' maskObj - Mask Object
227  Sub AddMaskToUserLayer(docObj, userLayerNameStr, maskObj)
```

Line 231 calls the ConfigureUserLayer() Function to either create a new user layer (or return an existing user layer) and to ensure that the display of this user layer is on.

```
229      ' Get/Create user layer
230      Dim userLayerObj
231      Set userLayerObj = ConfigureUserLayer(userLayerNameStr)
```

Line 234 uses the *AbsorbHoles* method on the *Mask* object.

```
233      ' Make mentorgons
234      Call maskObj.AbsorbHoles()
```

Lines 239 through 243 iterate through all of the shapes in the mask and adds them to the user layer (in this example there is only a one shape).

Line 241 acquires the points array associated with the current shape; Line 242 uses the PutUserLayerGfx method to create the graphics corresponding to this points array on the user layer.

```
236      ' Iterate through all the shapes in the mask and add
237      ' them to the user layer.
238      Dim shpObj
239      For Each shpObj In maskObj.Shapes
240          Dim pntsArr
```

```
241          pntsArr = shpObj.PointsArray
242          Call pnlDocObj.PutUserLayerGfx(userLayerObj, 0,
                        UBound(pntsArr,2)+1, pntsArr, True, Nothing)
243      Next
```

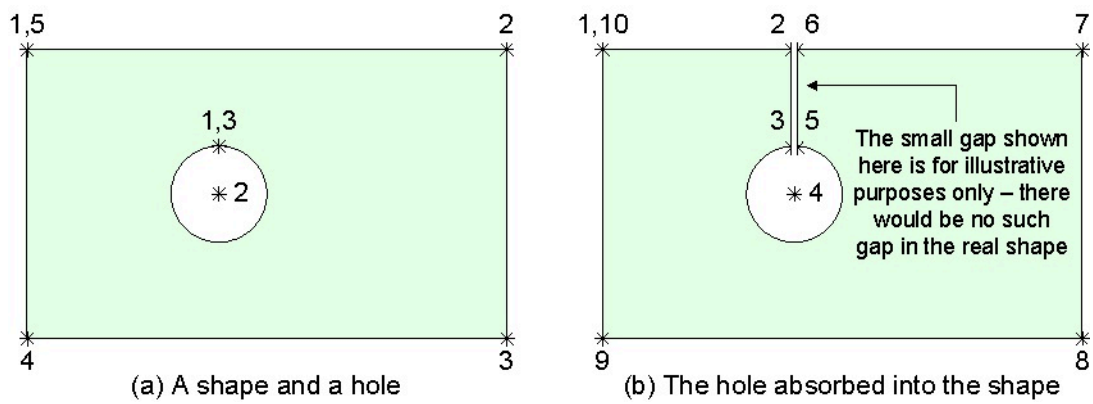
Finally, Line 245 exits this routine.

```
245  End Sub
```

Concept of Absorbing Holes Into Shapes

Generally, masks differentiate between shapes (outlines) and holes. For example, consider Figure 207: Absorbing Holes into Shapes(a) which comprises a single shape (a rectangular outline) and a single hole. In this case, the shape comprises five points, each of which is represented by X/Y values in a points array (the final point has the same X/Y location as the first point). By comparison, the hole comprises three points defining the start, center, and end (the hole's end point has the same X/Y location as its start point).

Figure 207. Absorbing Holes into Shapes



Two points arrays require manipulation to perform a task such as generating the graphics associated with this shape-hole combo on a user layer. The task becomes greater when considering masks containing hundreds or thousands of holes.

The solution is to use the *AbsorbHoles* method, which takes the points array associated with the shape and merges it with the points arrays associated with any holes that fall within the range of that shape as illustrated in Figure 207: Absorbing Holes into Shapes(b). The result is a single points array that encompasses both the shape and any holes. The narrow gap in this figure is for illustrative purposes only; there is no such gap in the real shape.

Note that a mask can contain multiple shapes, each of which may have holes associated with it. In this case, the *AbsorbHoles* method absorbs any holes into their corresponding shapes.

Creating and Testing the Script

1. Use the scripting editor of your choice to enter the script described (including the license server function) and save it as *CreateViaMask.vbs*.

2. Close any instantiations of Xpedition Layout and Xpedition FabLink that are currently running, and then launch a new instantiation of Xpedition FabLink.

3. Open the *CandyPanel.pnl* design.

4. Drag-and-drop the *CreateViaMask.vbs* script into the Xpedition FabLink design area.

5. Observe the via mask appear in front of the panel. Zoom on this mask and observe the manner the holes surrounding vias using padstack "VIA31" are displayed.

Enhancing the Script

There are a number of ways in which this script can be enhanced. One suggestion is as follows (it is a good idea for you to practice your scripting skills by implementing this example):

- Modify the script to create a mask for all of the pins and vias on the top of the panel and apply this mask to the solder mask layer.

Exercise 4: Generating Manufacturing Output Files

Scope: Xpedition Enterprise

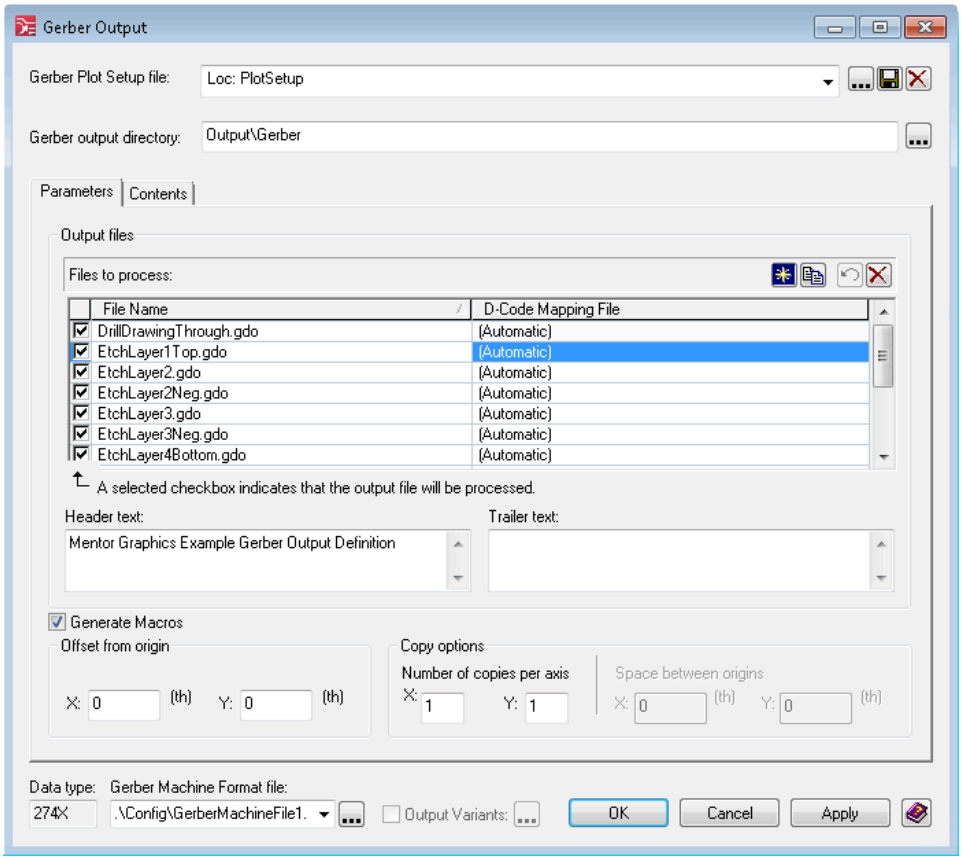
This script uses two different output engine automation interfaces; it configures their settings and instructs the interfaces to generate the appropriate output files.

Approach:	To use the Gerber and NC Drill interfaces to specify required options and generate associated manufacturing output files.
Points of Interest:	– Using the automation output engines
	– Running automation engines as batch processes
Items Used:	– Gerber Output Engine (Gerber Object)
	– NC Drill Output Engine (NCDrill Object)

Before Starting

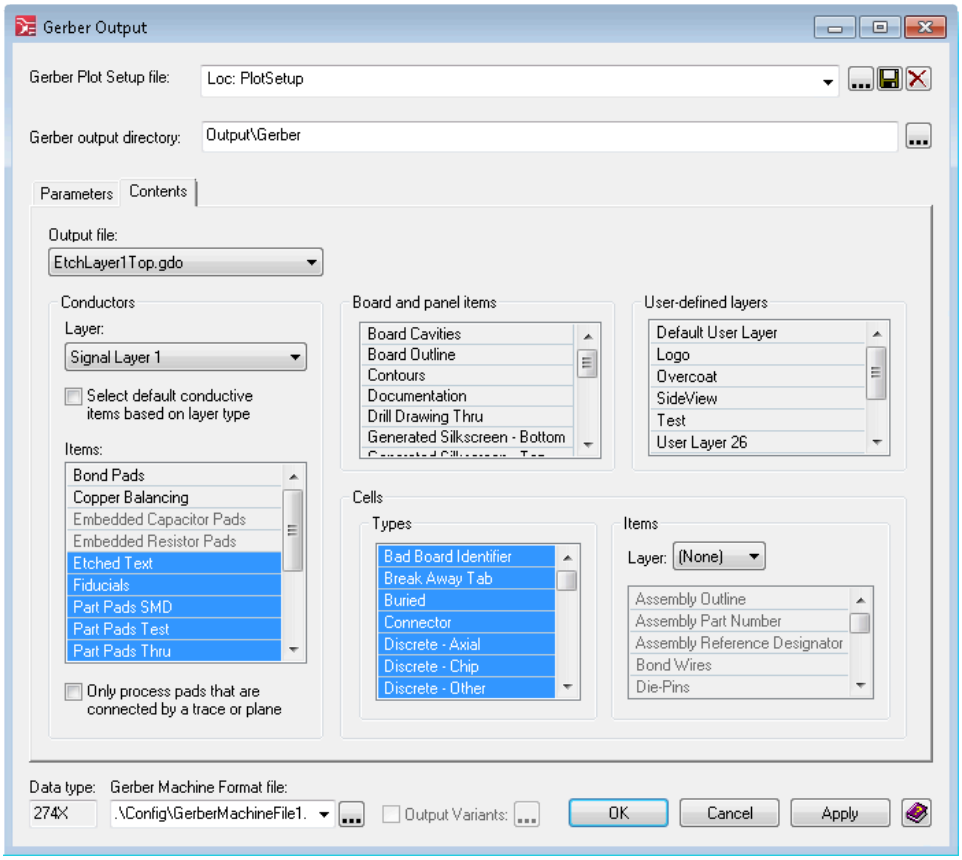
After completing a panel (or a board, because this particular script also works with Xpedition Layout), a suite of manufacturing files are to be generated to be passed to the board manufacturer. Xpedition FabLink provides many tools and much flexibility with regard to generating these files. For example: Choose **Output > Gerber** to access the *Gerber* engine (this is just one engine). This returns a dialog that allows you to set up an entire suite of files (Figure 208: Parameters Tab on the Gerber Output Dialog).

Figure 208. Parameters Tab on the Gerber Output Dialog



Each of these files has a set of options associated with it; for example, the options associated with the *EtchLayer1Top.gdo* file as illustrated in Figure 209: Options Associated With the EtchLayer1Top.gdo Output File (to access these options select the *EtchLayer1Top.gdo* item in the **File name** list under the **Parameters** tab as shown in Figure 208: Parameters Tab on the Gerber Output Dialog, and then select the **Contents** tab to display the associated options).

Figure 209. Options Associated With the EtchLayer1Top.gdo Output File



With the flexibility provided by Xpedition FabLink and its output engines, the possibility of human error becomes a major factor; for example, it is easy to incorrectly select/deselect one or more of these options. This is compounded by the generation of manufacturing output files occurring at the end of the board design process, by which time end-users may have forgotten all of the detailed settings required by their particular manufacturing process.

One solution is to create a script that automatically sets the desired options for the required output engines and then generates the appropriate manufacturing output files. The script described in this chapter uses the Gerber and NC Drill output engines; the *Prog IDs* for these, and other output engines are as follows:

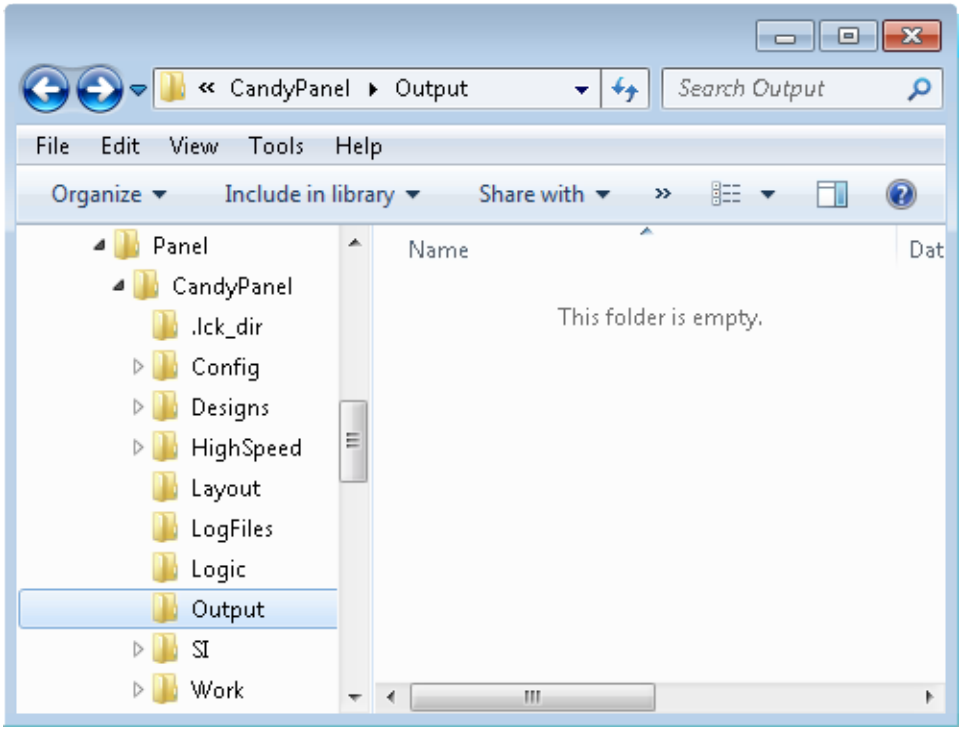
```
MGCPCBEngines.DFFDRC
MGCPCBEngines.DRC
MGCPCBEngines.GeneralInterfaceEngine
MGCPCBEngines.Gerber
MGCPCBEngines.IDFExport
MGCPCBEngines.ManufacturingOutputValidation
MGCPCBEngines.NCDrill
MGCPCBEngines.NeutralFileExport
MGCPCBEngines.ODBPPOutputEngine
MGCPCBEngines.PDFOutput
MGCPCBEngines.SilkscreenGen
```

Refer to the Automation Pro Batch Engines Reference for additional information on these engines.

The script in this chapter runs the Gerber and NC Drill automation engines as batch processes. This means that the process does not require Xpedition FabLink to be running.

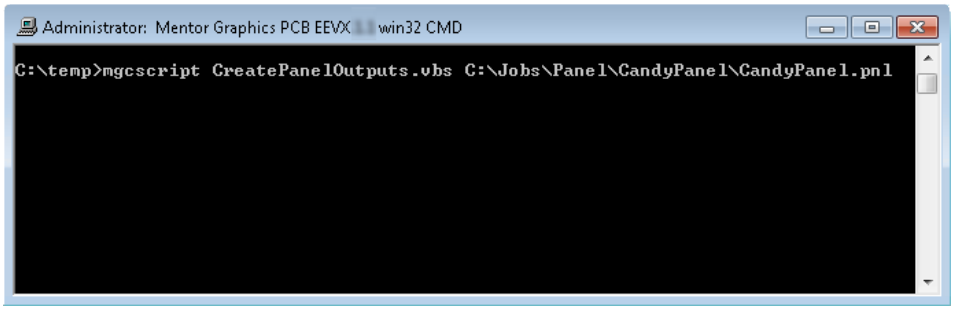
In order to see how the script works, view the *Output* directory/folder in the Xpedition FabLink *CandyPanel* design and note that it is empty as illustrated in Figure 210: Output Folder in the CandyPanel Design is Empty .

Figure 210. Output Folder in the CandyPanel Design is Empty



Launch a console (terminal) window and set its context to *C:\temp* (see "Running Scripts from the Command Line" on page 133 for more details on using console windows). Use the *mgcscript* engine to run the *CreatePanelOutputs.vbs* script in your *C:\temp* directory. The only argument to this script is the name of the panel file you want to process (*c:\jobs\Panel\CandyPanel\CandyPanel.pnl* in this example) as illustrated in Figure 211: Running CreatePanelOutputs.vbs Script. The script is written such that you must specify the full path.

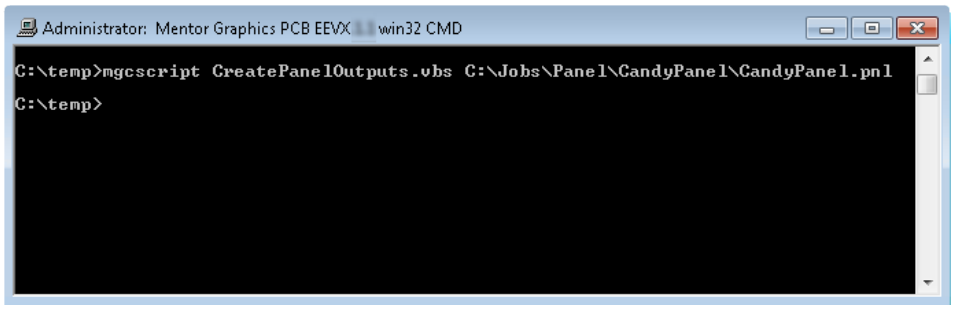
Figure 211. Running CreatePanelOutputs.vbs Script



After entering this command, press Enter. Observe that, after a few seconds the console window prompt reappears, thereby indicating that the script has completed as illustrated in Figure 212:

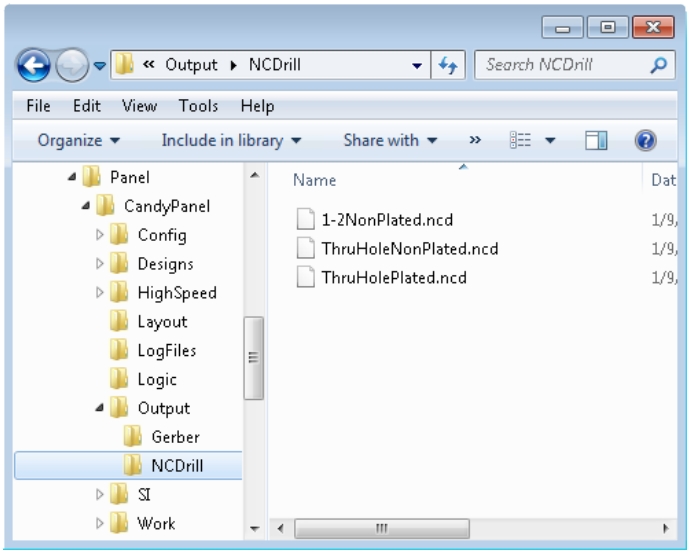
CreatePanelOutputs.vbs Script is Finished. The required output files are generated by the engines called by the script.

Figure 212. CreatePanelOutputs.vbs Script is Finished



View the *Output* directory/folder in the Xpedition FabLink *CandyPanel* design, and note that it now contains *Gerber* and *NCDrill* sub-folders, each of which contains appropriate manufacturing output files as indicated in Figure 213: Output Folder in the CandyPanel Design Is Populated.

Figure 213. Output Folder in the CandyPanel Design Is Populated



Script Sections

The key sections of this script are:

Initialization/Setup	Add type libraries, declare global variables, and perform any additional setup associated with this script.
Main Functions and Subroutines	The main functions and subroutines defined in the script.
	– RunGerber() Subroutine
	– RunNCDrill() Subroutine
Utility Functions and Subroutines	Special "workhorse" routines that are focused on this script.

	– SetupGerberMachineFormat() Subroutine
	– SetupGerberOutputFiles() Subroutine
	– SetupGerberSpecificOutputFile() Subroutine
	– SetupGerberAllOutputFile() Subroutine
	– SetupNCDrillMachineFormat() Subroutine
	– SetupNCDrillParameters() Subroutine
	– SetupNCDrillChart() Subroutine
Helper Functions and Subroutines	Generic "helper" routines that other scripts can use.
	– StripFile() Function
	– CorrectSlashes() Function
Validate Server	Licenses the document and validates the server. Refer to "Running Your First Script" on page 59 for more details on this function.

Initialization/Setup

Lines 10 and 11 add the required type libraries.

```
 1  ' This script sets-up and runs the NCDrill and Gerber
 2  ' output engines.  It is designed to be run externally.
 3  ' The script should be run as:
 4  ' mgcscript CreatePanelOutputs.vbs <design_name>
 5  ' Author: Toby Rimes
 6
 7  Option Explicit
 8
 9  ' Add type libraries
10  Scripting.AddTypeLibrary("MGCPCBEngines.Gerber")
11  Scripting.AddTypeLibrary("MGCPCBEngines.NCDrill")
```

Line 15 instantiates a local variable to hold the name of the design (the entire path including the file name); Line 17 instantiates a global variable to hold the path to the appropriate configuration directory/folder.

```
13  ' Global Variables
14  ' Name of the .pcb or .pnl file
15  Dim designNameStr
16  ' Location of the config directory
17  Dim designConfigPath
```

Lines 20 through 26 parse the command line arguments; if the number of arguments is incorrect a *MsgBox()*, on Line 25, informs the end-user as to the correct usage for this script (refer to "Running Scripts from the Command Line" on page 133 for more details on techniques for parsing command line arguments).

```
19  ' Parse the arguments
20  Dim argColl
21  Set argColl = ScriptHelper.Arguments
22  If argColl.Count = 3 Then
23      designNameStr = argColl.item(3)
24  Else
25      MsgBox("Usage: mgcscript CreatePanelOutputs.vbs
                                                    <design_name>")
26  End If
```

Line 30 calls the StripFile() Function to acquire directory/folder path containing the panel file. Line 31 uses this information to create a path to the appropriate configuration directory/folder.

```
28  ' Find the congif directory
29  Dim pnlPathStr
30  pnlPathStr = StripFile(designNameStr)
31  designConfigPath = pnlPathStr + "/Config/"
```

Finally, Lines 34 and 35 call the main RunGerber() Subroutine and the RunNCDrill() Subroutine, respectively.

```
33  ' Call main subroutines
34  Call RunGerber()
35  Call RunNCDrill()
```

Note:
This script does *not* require the standard license server function. If you view the code for the license server function (see the Creating and Running the Via Count Script topic in "Running Your First Script" on page 59), observe that the validation is performed on the Document object. The script in this chapter uses external engines but not the Document object ; therefore, no Document object validation is required.

An Automation Pro License is still required to use the automation engines. This license is acquired automatically when an engine's automation interface is accessed.

RunGerber() Subroutine

Lines 41 through 77 declare the main *RunGerber()* subroutine. Lines 44 and 45 create the *Gerber* engine/object.

```
40  ' Outputs the Gerber files.
41  Sub RunGerber()
42
43      ' Create Gerber Engine object
44      Dim gerberEngineObj
45      Set gerberEngineObj = CreateObject("MGCPCBEngines.Gerber")
```

Line 48 sets the *DesignFileName* property of the *Gerber* object to be the name of the design to process (this is the command line argument acquired on Line 23).

```
47      ' Set the design file name
48      gerberEngineObj.DesignFileName = designNameStr
```

Lines 51 and 52 call the SetupGerberMachineFormat() Subroutine and the SetupGerberOutputFiles() Subroutine, respectively.

```
50      ' Setup files for Gerber output
51      Call SetupGerberMachineFormat(gerberEngineObj)
52      Call SetupGerberOutputFiles(gerberEngineObj)
```

Because of a strong possibility for errors, Line 56 uses the *On Error Resume Next* statement and Line 57 clears any existing errors.

```
54      ' There are many possible errors when running the engine.
55      ' We will catch the case and display them on the own
56      On Error Resume Next
57      Err.Clear()
```

Line 60 uses the *Go* method to run the *Gerber* engine.

```
59      ' Run the Gerber Engine
60      gerberEngineObj.Go()
```

Because the script is performing many actions, there is the possibility of many errors. Each output engine supports its own set of errors, and each engine allows scripts to acquire its collection of errors after processing is complete. Line 62 uses the implicit Err object to determine if there was an error. Unlike the previous scripts, this script does not query the Err object to obtain the error description; instead, Line 65 uses the Errors property of the Gerber object to acquire the detailed collection of errors (if any) generated and gathered by the Gerber engine.

```
62      If Err Then
63          ' Check errors
64          Dim errorColl
65          Set errorColl = gerberEngineObj.Errors
```

Line 67 instantiates a variable to hold a string and initializes it to contain an empty string. Lines 68 through 70 iterate through all of the errors in the collection and adds each error to the string variable (each error is separated with a newline in the form of a *vbCrLf* constant).

```
66          Dim errObj
67          Dim errDisplayStr : errDisplayStr = ""
68          For Each errObj In errorColl
69              errDisplayStr = errDisplayStr & vbCrLf &
                                                errObj.ErrorString
70          Next
```

Line 72 checks if the string variable is empty or not; if not then there are errors, and Line 73 uses a *MsgBox()* to display all of the errors.

```
72          If Not errDisplayStr = "" Then
73              MsgBox("Error(s) running Gerber engine:" &
                                        vbCrLf & errDisplayStr)
74          End If
75      End If
76
77  End Sub
```

Note:
The scope of an *On Error Resume Next* statement is the routine in which it is called. When the calling routine is terminated, the system automatically returns to the default error handling mode (this is equivalent to the executing an *On Error Goto 0* statement before the end of the routine). Additionally, the system automatically returns to the default error handling mode within any *called* routines; however in this case, the system returns to the *On Error Resume Next* mode when the called routine returns control to the calling routine.

In summary, the effects of *On Error Resume Next* apply only within the routine in which this statement is found; any other routines (called or calling) use the default error mode unless otherwise specified within those routines themselves.

RunNCDrill() Subroutine

Lines 80 through 115 declare the main *RunNCDrill()* subroutine. Lines 83 and 84 create the *NCDrill* engine/object.

```
79  ' Outputs the NC Drill files.
80  Sub RunNCDrill()
81
82      ' Create NCDrill Engine object
83      Dim ncDrillEngineObj
84      Set ncDrillEngineObj =
                            CreateObject("MGCPCBEngines.NCDrill")
```

Line 87 sets the *DesignFileName* property of the *NCDrill* object to be the name of the design to process (this is the command line argument acquired on Line 23).

```
86      ' Set the design file name
87      ncDrillEngineObj.DesignFileName = designNameStr
```

Lines 89, 90, and 91 call the SetupNCDrillMachineFormat() Subroutine, the SetupNCDrillParameters() Subroutine and the SetupNCDrillChart() Subroutine, respectively.

```
89      Call SetupNCDrillMachineFormat(ncDrillEngineObj)
90      Call SetupNCDrillParameters(ncDrillEngineObj)
91      Call SetupNCDrillChart(ncDrillEngineObj)
```

Because of a strong possibility for errors while generating manufacturing output files, Line 95 uses the *On Error Resume Next* statement and Line 96 clears any existing errors.

```
93          ' There are many possible errors when running the engine.
94      ' We will catch the case and display them on the own
95      On Error Resume Next
96      Err.Clear
```

Line 99 uses the *Go* method to run the *NCDrill* engine.

```
98      ' Run the NCDrill Engine
99      ncDrillEngineObj.Go
```

Each output engine supports its own set of errors, and each engine allows scripts to acquire its collection of errors after processing is completed. Line 104 uses the *Errors* property of the *NCDrill* object to acquire the collection of errors (if any) generated and gathered by the NC Drill engine.

```
101     If Err Then
102         ' Check errors
103         Dim errorColl
104         Set errorColl = ncDrillEngineObj.Errors
```

Line 106 instantiates a variable to hold a string and initializes it to contain an empty string. Lines 107 through 109 iterate through all of the errors in the collection and adds each error to the string variable (the script separates each error with a newline in the form of a *vbCrLf* constant).

```
105         Dim errObj
106         Dim errDisplayStr : errDisplayStr = ""
107         For Each errObj In errorColl
108             errDisplayStr = errDisplayStr & vbCrLf &
                                                errObj.ErrorString
109         Next
```

Line 111 checks if the string variable is empty or not; if not then there are errors and Line 112 uses a *MsgBox()* to display all of the errors.

```
111         If Not errDisplayStr = "" Then
112             MsgBox("Error(s) running NC Drill engine:" &
                                          vbCrLf & errDisplayStr)
113         End If
114     End If
115 End Sub
```

Utility Functions and Subroutines

This section declares utility routines. These routines are similar in context to the generic "helper" routines (see the next topic), except that they are specific to this script and may not be used elsewhere without changes.

SetupGerberMachineFormat() Subroutine

This routine accepts a single parameter, which is the *Gerber* engine/object.

```
120  ' Sets up the machine format Gerber out
121  ' gerberEngineObj - Gerber Object
122  Sub SetupGerberMachineFormat(gerberEngineObj)
```

Lines 124 and 125 use the *MachineFormat* property of the *Gerber* object to acquire the *GerberMachineFormat* object.

```
124      Dim gerberMachineFormatObj
125      Set gerberMachineFormatObj = gerberEngineObj.MachineFormat
```

Line 127 specifies the file where all of the settings are written, and Lines 128 through 141 specify the required settings.

```
127      gerberMachineFormatObj.FileName = designConfigPath &
                                      "UserGerberMachineFile.gmf"
128      gerberMachineFormatObj.DataType = eengGerber274X
129      gerberMachineFormatObj.DataMode = TRUE
130      gerberMachineFormatObj.StepMode = eengStepIncremental
131      gerberMachineFormatObj.DataFormatLeadingDigits = 3
132      gerberMachineFormatObj.DataFormatTrailingDigits = 2
133      gerberMachineFormatObj.ZeroTruncation = eengZeroTruncationNone
134      gerberMachineFormatObj.CharacterSet = eengCharacterSetASCII
135      gerberMachineFormatObj.ArcStyle = eengArcStyleQuadrant
136      gerberMachineFormatObj.Delimiter = eengDataDelimiterStar
137      gerberMachineFormatObj.Comments = TRUE
138      gerberMachineFormatObj.SequenceNumbering = TRUE
139      gerberMachineFormatObj.Unit = eengUnitInch
140      gerberMachineFormatObj.PolygonFillMethod = eengPolygonFillDraw
141      gerberMachineFormatObj.RecordLength = 0
142
143  End Sub
```

SetupGerberOutputFiles() Subroutine

This routine accepts a single parameter, which is the *Gerber* engine/object.

```
145  ' Sets up the output files for the gerber engine
146  ' gerberEngineObj - Gerber Object
147  Sub SetupGerberOutputFiles(gerberEngineObj)
```

Line 150 acquires the collection of output files (note that this collection is initially empty).

```
149      Dim gerberoutputFileColl
150      Set gerberoutputFileColl = gerberEngineObj.OutputFiles
```

Lines 152 calls the SetupGerberSpecificOutputFile() Subroutine; Line 153 calls the SetupGerberAllOutputFile() Subroutine.

```
152      SetupGerberSpecificOutputFile(gerberoutputFileColl)
153      SetupGerberAllOutputFile(gerberoutputFileColl)
154
155  End Sub
```

SetupGerberSpecificOutputFile() Subroutine

This routine accepts a single parameter, which is the collection of Gerber output files acquired in the SetupGerberOutputFiles() Subroutine (recall that the collection is initially empty and that this is an *In-Out* parameter as are all VBScript function and subroutine parameters).

```
157  ' Sets up a Gerber output containing select items
158  ' outputFileColl - GerberOutputFiles Collection
159  Sub SetupGerberSpecificOutputFile(outputFileColl)
```

Line 162 uses the *Add* method on the output file collection to add a new output file to the collection and to specify the name of this output file.

```
161      Dim outFileAddEachObj
162      Set outFileAddEachObj =
                        outputFileColl.Add("SpecificElements.gdo")
```

Lines 163 through 166 specify the attributes to associate with the new output file.

```
163      outFileAddEachObj.FlashPads = FALSE
164      outFileAddEachObj.HeaderText = "Gerber Output Specific Elements"
165      outFileAddEachObj.TrailerText = "Gerber Output Specific Elements"
166      outFileAddEachObj.ProcessUnconnectedPads = FALSE
```

Line 170 acquires the *BoardItems* collection associated with this output file.

```
168      ' Add board items individually
169      Dim boardItemsObj
170      Set boardItemsObj = outFileAddEachObj.BoardItems
```

Lines 171 through 180 add specific board items to output to the collection.

```
171      boardItemsObj.Add(eengBoardItemContours)
172      boardItemsObj.Add(eengBoardItemDrillDrawingThru)
173      boardItemsObj.Add(eengBoardItemGeneratedSilkscreenBottom)
174      boardItemsObj.Add(eengBoardItemGeneratedSilkscreenTop)
175      boardItemsObj.Add(eengBoardItemMountingHoles)
176      boardItemsObj.Add(eengBoardItemSoldermaskBottom)
177      boardItemsObj.Add(eengBoardItemSoldermaskTop)
178      boardItemsObj.Add(eengBoardItemSolderpasteBottom)
179      boardItemsObj.Add(eengBoardItemSolderpasteTop)
180      boardItemsObj.Add(eengBoardItemToolingHoles)
```

Line 184 acquires the *CellTypes* collection associated with this output file.

```
182      ' Add all cell types individually
183      Dim cellTypesObj
184      Set cellTypesObj = outFileAddEachObj.CellTypes
```

Lines 185 through 208 add specific cell types to output to the collection.

```
185      cellTypesObj.Add(eengCellTypeBuried)
186      cellTypesObj.Add(eengCellTypeConnector)
187      cellTypesObj.Add(eengCellTypeDiscreteAxial)
188      cellTypesObj.Add(eengCellTypeDiscreteChip)
189      cellTypesObj.Add(eengCellTypeDiscreteOther)
190      cellTypesObj.Add(eengCellTypeDiscreteRadial)
191      cellTypesObj.Add(eengCellTypeEdgeConnector)
192      cellTypesObj.Add(eengCellTypeGeneral)
193      cellTypesObj.Add(eengCellTypeGraphic)
194      cellTypesObj.Add(eengCellTypeICBareDie)
195      cellTypesObj.Add(eengCellTypeICBGA)
196      cellTypesObj.Add(eengCellTypeICDIP)
197      cellTypesObj.Add(eengCellTypeICFlipChip)
198      cellTypesObj.Add(eengCellTypeICLCC)
199      cellTypesObj.Add(eengCellTypeICOther)
200      cellTypesObj.Add(eengCellTypeICPGA)
201      cellTypesObj.Add(eengCellTypeICPLCC)
202      cellTypesObj.Add(eengCellTypeICSIP)
203      cellTypesObj.Add(eengCellTypeICSOIC)
204      cellTypesObj.Add(eengCellTypeJumper)
205      cellTypesObj.Add(eengCellTypeMechanical)
206      cellTypesObj.Add(eengCellTypePanelIdentifier)
207      cellTypesObj.Add(eengCellTypeTestCoupon)
208      cellTypesObj.Add(eengCellTypeTestPoint)
```

Line 210 specifies to output cells on the top side.

```
210      outFileAddEachObj.CellItemsSide = eengCellSideTop
```

Line 214 acquires the *CellItems* collection associated with this output file.

```
212      ' Add cell items individually
213      Dim cellItemsObj
214      Set cellItemsObj = outFileAddEachObj.CellItems
```

Lines 215 through 222 add specific cell items to output to the collection.

```
215      cellItemsObj.Add(eengCellItemAssemblyOutline)
216      cellItemsObj.Add(eengCellItemAssemblyPartNumber)
217      cellItemsObj.Add(eengCellItemAssemblyReferenceDesignator)
218      cellItemsObj.Add(eengCellItemInsertionOutline)
219      cellItemsObj.Add(eengCellItemPlacementOutline)
220      cellItemsObj.Add(eengCellItemSilkScreenOutline)
221      cellItemsObj.Add(eengCellItemSilkScreenPartNumber)
222      cellItemsObj.Add(eengCellItemSilkScreenReferenceDesignator)
```

Line 225 specifies the conductive layer to output.

```
224      ' Add conductive items individually
225      outFileAddEachObj.ConductiveLayer = 1
```

Line 227 acquires the *ConductiveItems* collection associated with this output file.

```
226      Dim conductiveItemsObj
227      Set conductiveItemsObj = outFileAddEachObj.ConductiveItems
```

Lines 228 through 239 add specific conductive items to output to the collection.

```
228      conductiveItemsObj.Add(eengCondItemActualPlaneShapes)
229      conductiveItemsObj.Add(eengCondItemCopperBalancing)
230      conductiveItemsObj.Add(eengCondItemEtchedText)
231      conductiveItemsObj.Add(eengCondItemFiducials)
232      conductiveItemsObj.Add(eengCondItemPartPadsSMD)
233      conductiveItemsObj.Add(eengCondItemPartPadsTest)
234      conductiveItemsObj.Add(eengCondItemPartPadsThru)
235      conductiveItemsObj.Add(eengCondItemPlaneData)
236      conductiveItemsObj.Add(eengCondItemResistorAreas)
237      conductiveItemsObj.Add(eengCondItemTraces)
238      conductiveItemsObj.Add(eengCondItemViaHoles)
239      conductiveItemsObj.Add(eengCondItemViaPads)
240
241  End Sub
```

SetupGerberAllOutputFile() Subroutine

This routine accepts a single parameter, which is the collection of Gerber output files acquired in the SetupGerberOutputFiles() Subroutine and modified in the SetupGerberSpecificOutputFile() Subroutine.

```
243  ' Sets up a Gerber output containg all data
244  ' outputFileColl - GerberOutputFile Collection
245  Sub SetupGerberAllOutputFile(outputFileColl)
```

Line 249 uses the *Add* method on the output file collection to add a new output file to the collection and to specify the name of this output file.

```
247      ' Add all items using "All" constants
248      Dim outFileAddAllObj
249      Set outFileAddAllObj = outputFileColl.Add("AllElements.gdo")
```

Lines 250 through 253 specify the attributes to be associated with the new output file.

```
250      outFileAddAllObj.FlashPads = TRUE
251      outFileAddAllObj.HeaderText = "Gerber Output All Elements"
252      outFileAddAllObj.TrailerText = "Gerber Output All Elements"
253      outFileAddAllObj.ProcessUnconnectedPads = True
```

Lines 255 through 260 specify to add everything.

```
255      outFileAddAllObj.BoardItems.Add(eengBoardItemAll)
256      outFileAddAllObj.CellTypes.Add(eengCellTypeAll)
257      outFileAddAllObj.CellItemsSide = eengCellSideTop
258      outFileAddAllObj.CellItems.Add(eengCellItemAll)
259      outFileAddAllObj.ConductiveLayer = 1
260      outFileAddAllObj.ConductiveItems.Add(eengCondItemAll)
261
262  End Sub
```

SetupNCDrillMachineFormat() Subroutine

This routine accepts a single parameter, which is the *NCDrill* engine/object.

```
264  ' Sets up an NC Drill machine format
265  ' ncDrillEngineObj - NCDrill Object
266  Sub SetupNCDrillMachineFormat(ncDrillEngineObj)
```

Lines 267 and 268 use the *MachineFormat* property of the *NCDrill* object to acquire the *NCDrillMachineFormat* object.

```
267      Dim ncDrillMFObj
268      Set ncDrillMFObj = ncDrillEngineObj.MachineFormat
```

Lines 271 through 284 setup the required settings.

```
270      ' Setup EnglishDrillMachineFormat.dff
271      ncDrillMFObj.DataType = eengExcellon
272      ncDrillMFObj.Unit = eengUnitInch
273      ncDrillMFObj.DataFormatLeadingDigits = 2
274      ncDrillMFObj.DataFormatTrailingDigits = 4
275      ncDrillMFObj.StepMode = eengStepAbsolute
276      ncDrillMFObj.ZeroTruncation = eengZeroTruncationTrailing
277      ncDrillMFObj.DataMode = TRUE
278      ncDrillMFObj.ArcStyle= eengArcStyleRadius
279      ncDrillMFObj.SequenceNumbering = FALSE
280      ncDrillMFObj.CharacterSet = eengCharacterSetASCII
281      ncDrillMFObj.Delimiter = ""
282      ncDrillMFObj.Comments = TRUE
283      ncDrillMFObj.CommentStr = "; "
284      ncDrillMFObj.RecordLength = 0
```

Line 287 specifies a file to which the settings are saved (this is optional, if this file is not specified the NCDrill engine uses these settings but does not save them).

```
286      ' Optional, where to write the format file
287      ncDrillMFObj.FileName = designConfigPath +
                                    "EnglishDrillMachineFormat.dff"
288  End Sub
```

SetupNCDrillParameters() Subroutine

This routine accepts a single parameter, which is the *NCDrill* engine/object.

```
290  ' Sets up NC Drill parameters
291  ' ncDrillEngineObj - NCDrill Object
292  Sub SetupNCDrillParameters(ncDrillEngineObj)
```

Line 295 uses the *Parameters* property of the *NCDrill* object to acquire the *NCDrillParameters* object.

```
294      Dim ncDrillParametersObj
295      Set ncDrillParametersObj = ncDrillEngineObj.Parameters
```

Lines 299 through 306 setup the required NC Drill parameters.

```
293
296
297      ' Same as the options in "Drill Options"
298      ' tab of the "NC Drill Generation" dialog
299      ncDrillParametersObj.SweepAxis = eengSweepaxisHorizontal
300      ncDrillParametersObj.Bandwidth(eengUnitInch) =  0.1
301      ncDrillParametersObj.PreDrillHolesLargerThan(eengUnitInch) = 1
302      ncDrillParametersObj.OutputFileExtension = ".ncd"
303      Call ncDrillParametersObj.ClearFileHeader()
304      Call ncDrillParametersObj.ClearFileNotes()
305      Call ncDrillParametersObj.AddFileHeader("Drill Setup")
306      Call ncDrillParametersObj.AddFileNotes ("This file was
                  generated by the CreatePanelOutputs.vbs script.")
307  End Sub
```

SetupNCDrillChart() Subroutine

This routine accepts a single parameter, which is the *NCDrill* engine/object.

```
309  ' Sets up the NCDrill chart.
310  ' ncDrillEngineObj - NCDrill Object
311  Sub SetupNCDrillChart(ncDrillEngineObj)
```

Line 315 uses the *Chart* property of the *NCDrill* object to acquire the *NCDrillChart* object.

```
313      Dim ncDrillChartObj
314      Set ncDrillChartObj = ncDrillEngineObj.Chart
```

Lines 318 through 329 setup the required NC Drill chart options.

```
316      ' Same as the options in "Drill Chart Options"
317      ' tab of the "NC Drill Generation" dialog
318      ncDrillChartObj.FontName = "VeriBest Gerber 0"
319      ncDrillChartObj.FontSize(eengUnitInch) =  0.05
320      ncDrillChartObj.LineSpacing(eengUnitInch) = 0.05
321      ncDrillChartObj.PenWidth(eengUnitInch) =  0.005
```

```
322      ncDrillChartObj.PrecisionLeadingDigits = 1
323      ncDrillChartObj.PrecisionTrailingDigits = 4
324      ncDrillChartObj.Unit = eengUnitsEnglish
325      ncDrillChartObj.PositiveTolerance(eengUnitInch) =  1
326      ncDrillChartObj.NegativeTolerance(eengUnitInch) =  1
327      ncDrillChartObj.AssignDrillCharacters = FALSE
328      ncDrillChartObj.Title = "title"
329      ncDrillChartObj.SpecialNotes = "notes"
```

Line 332 acquires the *Columns* collection associated with the *NCDrillChart* object.

```
331      Dim ncDrillChartObjColumns
332      Set ncDrillChartObjColumns = ncDrillChartObj.Columns
```

Lines 334 through 340 specify what columns should appear in the NC Drill chart.

```
334      ncDrillChartObjColumns.Add(eengColumnSymbol)
335      ncDrillChartObjColumns.Add(eengColumnDiameter)
336      ncDrillChartObjColumns.Add(eengColumnTolerance)
337      ncDrillChartObjColumns.Add(eengColumnPlated)
338      ncDrillChartObjColumns.Add(eengColumnPunched)
339      ncDrillChartObjColumns.Add(eengColumnHolename)
340      ncDrillChartObjColumns.Add(eengColumnQuantity)
341  End Sub
```

Helper Functions and Subroutines

This script section declares several generic "helper" routines. These routines can be used in other scripts and are not focused on this script exclusively.

Tip
It is a good idea to collect both of the helper routines in this section into a library along with any related routines (see also “Creating and Using a Library” on page 529).

StripFile() Function

This routine accepts a string parameter containing a path that includes a file name; it strips the file name off the end of the path and returns a string containing only the path to the directory/folder containing the file. (Also note that this routine calls the CorrectSlashes() Function.)

```
347  ' Removes the file name from a file path string
348  ' fileNameStr - String
349  Function StripFile(fileNameStr)
350
351      Dim pathStr: pathStr = Replace(fileNameStr, "\", "/")
352
353      ' If there is no . return the string
354      If InStrRev(pathStr, ".") = 0 Then
355          StripFile = pathStr
356      Else
357          ' Strip the file
```

```
358         StripFile = Left(pathStr, InStrRev(pathStr, "/") - 1)
359     End If
360
361     ' Correct the slashes
362     CorrectSlashes(StripFile)
363
364 End Function
```

CorrectSlashes() Function

This routine accepts a string parameter containing a path that may or may not contain a file name. If the current platform is UNIX or Linux, the routine returns a string with the path specified using '/' directory/folder delimiting characters; otherwise, if the current platform is Windows, the routine returns a string with the path specified using '\' directory/folder delimiting characters.

```
366 ' Ensures a path string is using the appropriate
367 ' slashes for the current platform
368 ' pathStr - String
369 Function CorrectSlashes(pathStr)
370
371     If Scripting.IsUnix Then
372         CorrectSlashes = Replace(pathStr, "\", "/")
373     Else
374         CorrectSlashes = Replace(pathStr, "/", "\")
375     End If
376
377 End Function
```

Creating and Testing the Script

1. Use the scripting editor of your choice to enter the script described and save it as *CreatePanelOutputs.vbs*.

2. Close any instantiations of Xpedition FabLink that are currently running.

3. Launch a console (terminal) window and set its context to *C:\temp*. Use the *mgcscript* engine to run the *CreatePanelOutputs.vbs* script passing the name of the panel file you want to process (*c:\jobs\Panel\CandyPanel\CandyPanel.pnl* in this example) as a parameter (see "Running Scripts from the Command Line" on page 133 for more details on using console windows).

4. Observe that, after a few seconds during which the required output files are being generated by the engines called by the script, the console window prompt reappears, thereby indicating that the script has completed.

5. View the *Output* directory/folder in the Xpedition FabLink *CandyPanel* design and observe that it now contains *Gerber* and *NCDrill* sub-folders, each of which contains appropriate manufacturing output files.

Enhancing the Script

There are a number of ways in which this script could be enhanced. One suggestion is as follows (it is a good idea for you to practice your scripting skills by implementing this example):

- This script uses a *MsgBox()* to display any errors. The script can be modified to display error messages directly in the console (terminal) window (refer to the “Using mgcscript” topic of the Layout Automation Reference for more details on how this may be achieved).

- This script can also be run with Xpedition Layout on a board. In reality, there are a few options specified in the script that are not valid for Boards *objects*. If you run the script "as-is" (without modification) and pass in a **.pcb* file, the resulting error messages direct you to these options. Commenting out these lines allow the script to work with a **.pcb* file.

Exercise 5: Automated Board Placement on a Panel

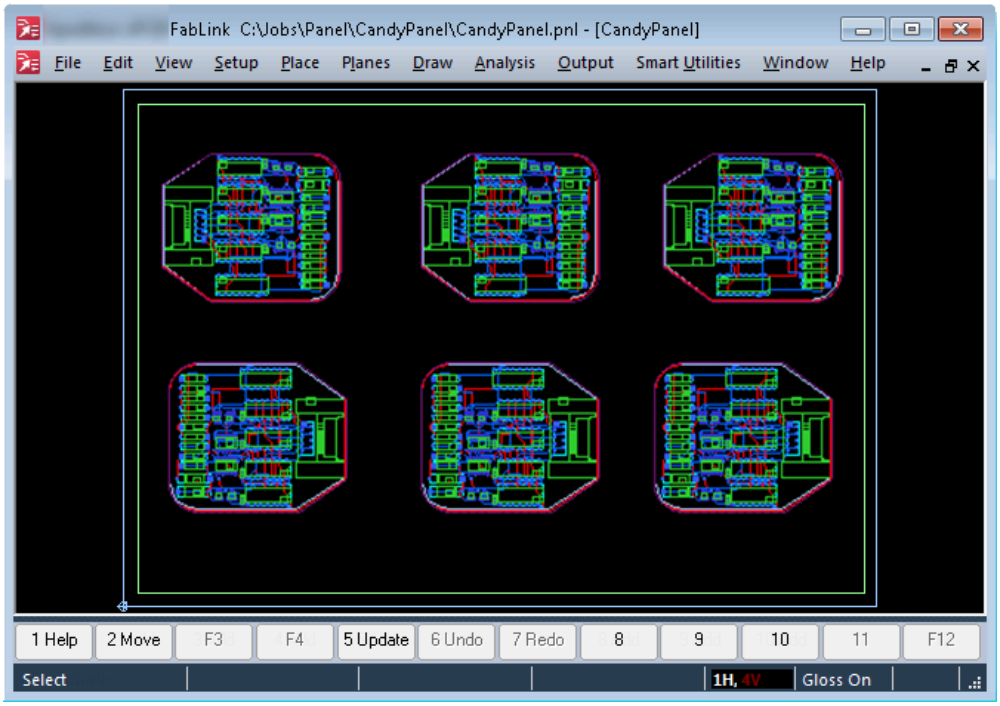
This script attempts to place the optimal (maximum) number of copies of a specified board in Xpedition Layout onto an existing panel in Xpedition FabLink.

Approach:	To access the board extreme in Xpedition Layout; to calculate the maximum number of boards that can fit within the panel border (try both 0º and 90º board rotations); and to then automatically place the boards on the panel.
Points of Interest:	– Using a Xpedition FabLink script to access data in Xpedition Layout.
	– Launching a dialog/form from within a script.
	– Implementing script-to-script communication.
Items Used:	– Xpedition Layout Server
	– ProcessScript Method
	– PutBoard Method
	– Scripting.Globals Property

Before Starting

Although arbitrary panel sizes are sometimes used, panels generally come in standard sizes, so it is common to try to fit that maximum number of boards into a panel. This script allows the end-user to specify a board design in Xpedition Layout along with a minimum clearance value. The script then determines the maximum number of boards that can be placed within the panel border on an existing panel in Xpedition FabLink (it tries both 0º and 90º board rotations). Finally, the script automatically places the boards on the panel.

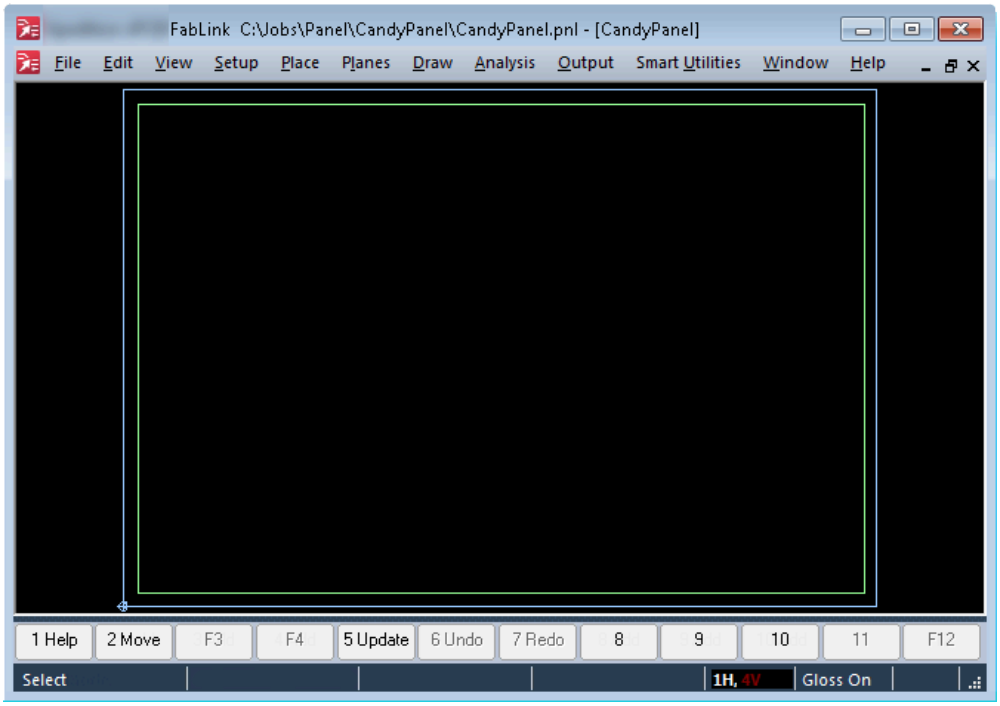
Figure 214. CandyPanel Panel



In order to see how this script works, launch Xpedition FabLink and open the existing *CandyPanel* panel design, it appears as illustrated in Figure 214: CandyPanel Panel. Note that only six boards are presented in this design.

Select and delete the six existing boards, leaving the bare panel as illustrated in Figure 215: Bare Panel.

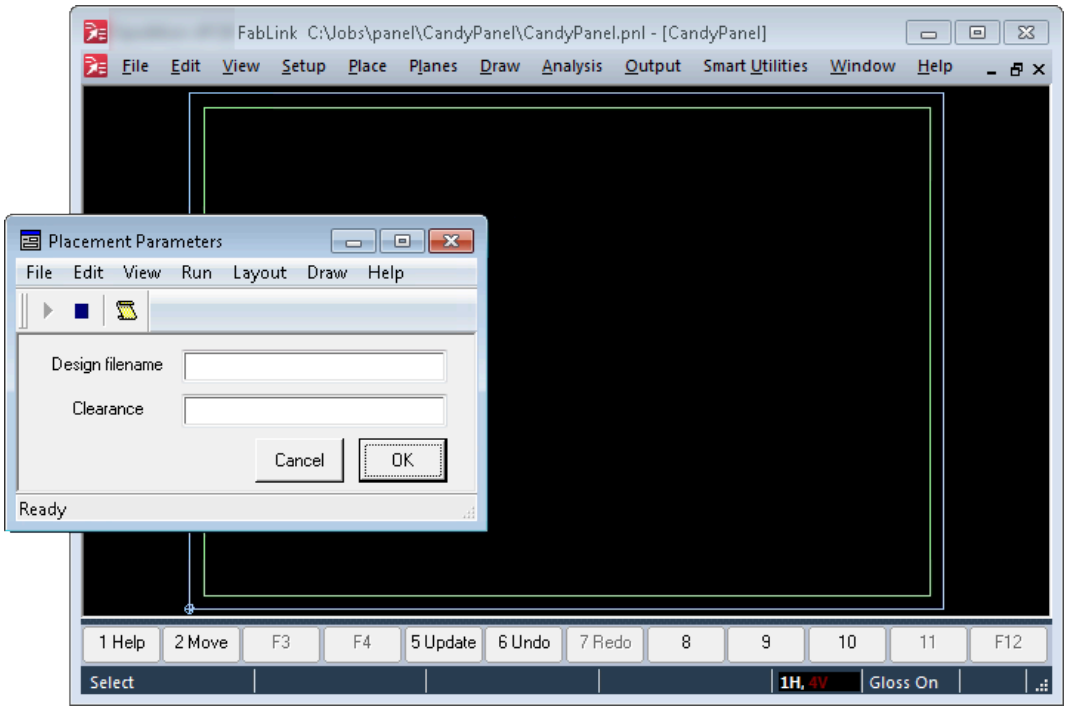
Figure 215. Bare Panel



Drag-and-drop the *PlaceBoards.vbs* script into the Xpedition FabLink design area. This script automatically launches the **Placement Parameters** dialog/form illustrated in Figure 216: Script invokes the Placement Parameters Dialog.

Note:
You must ensure that the *PlaceBoardInput.efm* dialog/form file is in one of your WDIR folders/paths before running the *PlaceBoards.vbs* script.

Figure 216. Script invokes the Placement Parameters Dialog



Enter the path and filename to the *Candy.pcb* on your system in the **Design filename** field (this path is *C:\temp\Candy\PCB\Candy.pcb* on the system, but it may be different on your system). Enter a clearance value of 100 in the **Clearance** field (the script assumes that these are the same units as in the Xpedition FabLink panel design).

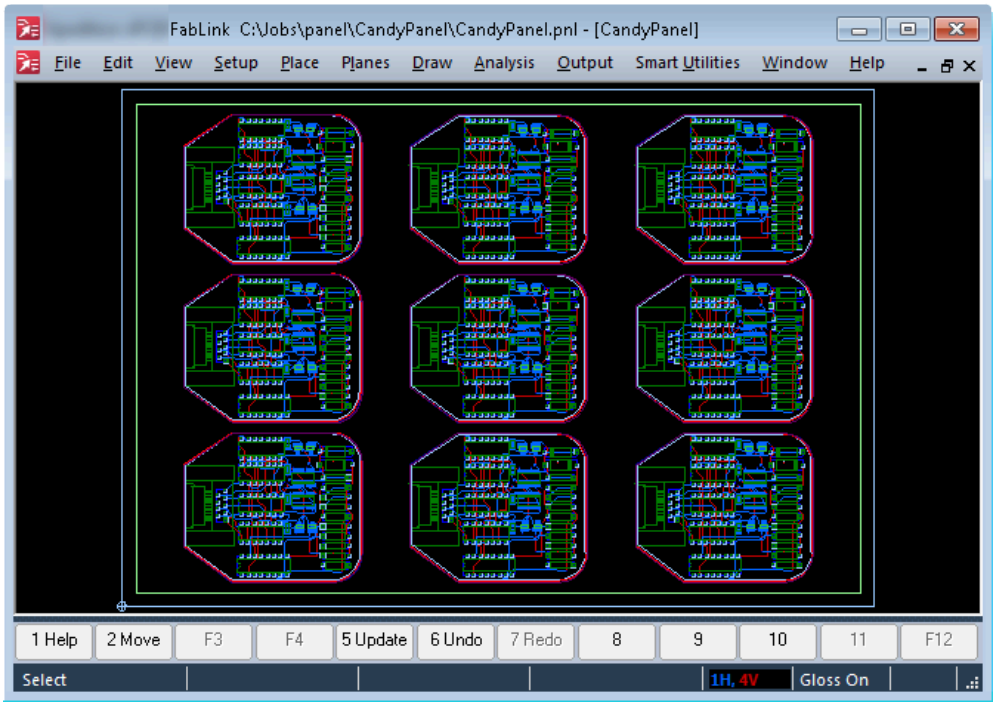
Note:
When the script creates the **Placement Properties** dialog/form, in addition to clicking the **Cancel** or **OK** buttons, you also have to accommodate the possibility of the user clicking the red 'X" (**Close**) button in the upper right-hand corner of the dialog.

Click the **OK** button.

The script evaluates the maximum number of rows and columns of this board design using a 0º board rotation. It then evaluates the maximum number of rows and columns of this board using a 90º board rotation. Based on the result, the script then automatically place the boards, three rows by three columns at 0º rotation, for this example. This is illustrated in Figure 217: Maximum Number of Boards on the Panel.

Note:
When the boards first appear in the panel, they are presented only as outlines. After the script runs choose **Place > Upload All Design Data** to populate the boards with all graphics.

Figure 217. Maximum Number of Boards on the Panel



Script Sections

The key sections of this script are:

Constant Definitions	Defines the constants the script uses.
Initialization/Setup	Add type libraries, declare global variables, and perform any additional setup associated with this script.
Main Functions and Subroutines	The main functions and subroutines defined in the script.
	– GetParameters() Function
	– PlaceBoards() Subroutine
Utility Functions and Subroutines	Special "workhorse" routines that are focused on this script.
	– FindBestMatrix() Subroutine
Helper Functions and Subroutines	Generic "helper" routines that other scripts can use.
	– GetBoardDimensions() Subroutine
	– EqualizeClearance() Function
	– FindMaxFit() Function
	– FindInWDIR() Function

Validate Server	Licenses the document and validates the server. Refer to "Running Your First Script" on page 59 for more details on this function.

Constant Definitions

Lines 10 through 14 define the constants for implementing script-to-script communications. Line 16 defines the name of the dialog/form to invoke from within the main script.

```
 1  ' This script take a clearance and a board name.  Given
 2  ' a predefined panel it places the largest number of
 3  ' boards possible at either all 0 or all 90 degree
 4  ' orientations.
 5  ' Author: Toby Rimes
 6
 7  Option Explicit
 8
 9  ' Constants
10  Const CANCEL = "Cancel"
11  Const OK = "OK"
12  Const ANSWER = "place_param_ans"
13  Const DESIGNNAME = "place_param_designname"
14  Const CLEARANCE = "place_param_clearance"
15
16  Const INPUT_FORM = "PlaceBoardInput.efm"
```

Initialization/Setup

Lines 19 and 20 add the type libraries for Xpedition Layout and Xpedition FabLink.

Note:
It is not necessary to add type libraries for both Xpedition Layout and Xpedition FabLink because the automation interfaces are the same. The reason to add both type libraries is that this is good programming practice.

```
18  ' Add any type libraries to be used.
19  Scripting.AddTypeLibrary("MGCPCB.FablinkXEApplication")
20  Scripting.AddTypeLibrary("MGCPCB.ExpeditionPCBApplication")
```

Lines 23 and 24 instantiate global variables to hold the Xpedition FabLink *Application* and *Document* objects.

```
22  ' Global variables
23  Dim pnlAppObj                 'Application object
24  Dim pnlDocObj                 'Document object
```

Line 27 acquires the Xpedition FabLink *Application* object; Line 30 acquires the Xpedition FabLink *Document* (panel) object; and Line 32 licenses the document (validates the server).

```
26  ' Get the application object.
27  Set pnlAppObj = Application
28
29  ' Get the active document
30  Set pnlDocObj = pnlAppObj.ActiveDocument
31  ' License the document
32  ValidateServer(pnlDocObj)
```

Line 35 instantiates a variable to hold the units to use, and then assigns the current design units associated with the panel to this variable.

```
34  ' Units to be used throughout scripts
35  Dim unitsEnum : unitsEnum = pnlDocObj.CurrentUnit
```

Other scripts explicitly defined the units to use prior to calling the *ValidateServer()* function. In this case, the script uses the current design units associated with the panel.

Line 38 calls the main GetParameters() Function as part of an *If … Then* statement. This is the function that invokes the dialog/form, waits for the end-user to input the required values, and then accesses these values. If this function returns *True* (meaning it successfully acquired the path/name of the board in Xpedition Layout and the desired clearance), then Line 39 calls the main PlaceBoards() Subroutine.

```
37  Dim designNameStr, clearanceReal
38  If GetParameters(designNameStr, clearanceReal) Then
39      Call PlaceBoards(designNameStr, clearanceReal)
40  End If
```

GetParameters() Function

Lines 50 through 72 declare the function that invokes the dialog/form, waits for the user to input the required values, and then accesses these values. This function accepts two parameters (both of these are *In-Out* parameters whose values are set within the function): the path/name of the Xpedition Layout board document and the required clearance value.

```
45  ' Invokes the form PlaceBoardInput.efm and waits for it to exit
46  ' After exit it pulls variables from Scripting.Globals that
47  ' were set by the form.
48  ' designNameStr - String (In/Out)
49  ' clearnaceReal - Real (In/Out)
50  Function GetParameters(designNameStr, clearanceReal)
```

Line 52 initializes the *ANSWER* key of the *Scripting.Globals* object to be an empty string (recall that the *ANSWER* key was one of the constants declared at the beginning of the script). This is used as a "flag" that lets the script know when the end-user exits the dialog/form by clicking its OK or Cancel buttons.

```
51      ' Initialize answer
52      Scripting.Globals(ANSWER) = ""
```

Note:
Scripting.Globals is a *Dictionary* object that can be accesses by any in-process script/client/form. This is the mechanism by which one script/client/form can communicate with another script/client/form.

Line 55 uses the *ProcessScript* method to invoke the dialog/form. Before using the *ProcessScript* method to invoke the dialog/form, the script calls FindInWDIR() Function to acquire the full path to the form.

```
54      ' Display the form
55      Call pnlAppObj.ProcessScript(FindInWDIR(INPUT_FORM), False)
```

Line 59 through 61 use a *While … Wend* control loop to wait for the end-user to enter the data and/or exit the form. The *Scripting.Sleep* statement on Line 60 prevents this loop from consuming all of the CPU cycles; the sleep value of 300 is understood by the system to be in milliseconds.

```
57      ' Wait until the form exits. This is flagged by
58      ' the form assigning a value to the Globals "ANSWER"
59      While Scripting.Globals(ANSWER) = ""
60          Call Scripting.Sleep(300)
61      Wend
```

Line 65 uses the *Scripting.Globals* object to check whether the dialog/form set the *ANSWER* key to *OK* or *Cancel* indicating whether the end-user clicks **OK** or **Cancel**.

If the end-user clicks **OK**, then Line 66 uses the *DESIGNNAME* key to retrieve the path/name of the board design in Xpedition Layout; Line 67 uses the *CLEARANCE* key to retrieve the clearance value requested by the end-user; and Line 68 sets the return value for this function to be *True*. Alternatively, if the end-user clicks **Cancel** (or the 'X' **Close** button in the upper-right-hand corner of the form), then Line 70 sets the return value for this function to be *False*.

```
63      ' If the user hit ok get the parameters
64      ' and return true.  Otherwise return false.
65      If Scripting.Globals(ANSWER) = OK Then
66          designNameStr = Scripting.Globals(DESIGNNAME)
67          clearanceReal = Int(Scripting.Globals(CLEARANCE))
68          GetParameters = True
69      Else
70          GetParameters = False
71      End If
72  End Function
```

PlaceBoards() Subroutine

Lines 79 through 126 declare the subroutine that (a) determines the optimal (maximum) number of boards that fit on the panel and (b) places the boards on the panel. This routine accepts two parameters: the path/name of the Xpedition Layout board document and the required clearance value (these are identical to the GetParameters() Function, except that these are treated as input parameters by this routine).

```
74  ' Determines the optimal matrix placement of boards using
75  ' a given clearance and the existing panel border.
76  ' Places the boards in the calculated matrix.
77  ' designNameStr - String
78  ' clearanceReal - Real
79  Sub PlaceBoards(designNameStr, clearanceReal)
```

Lines 81 and 82 call the GetBoardDimensions() Subroutine to determine the width and height of the board in Xpedition Layout.

```
80      ' Get the board dimensions
81      Dim boardXDimReal, boardYDimReal
82      Call GetBoardDimensions(designNameStr, boardXDimReal,
                                                   boardYDimReal)
```

Lines 85 and 86 acquire the extrema of the panel border in Xpedition FabLink.

```
84      ' Get the extrema of the panel border
85      Dim borderExtremaObj
86      Set borderExtremaObj = pnlDocObj.PanelBorder.Extrema
```

Lines 90 and 91 use the extrema of the panel border to calculate the width and height of the usable area in the panel.

```
88      ' Get the horizontal and vertical dimensions of border
89      Dim borderXDimReal, borderYDimReal
90      borderXDimReal = borderExtremaObj.MaxX – borderExtremaObj.MinX
91      borderYDimReal = borderExtremaObj.MaxY – borderExtremaObj.MinY
```

Lines 95 and 96 call the FindBestMatrix() Subroutine to determine the optimal (maximum) number of boards that fit on the panel. This function *returns* three parameters: the number of boards that fit in the horizontal plane; the number of boards that fit in the vertical plane; and the orientation/rotation (0 or 90, corresponding to 0º or 90º) to be applied to all of the boards.

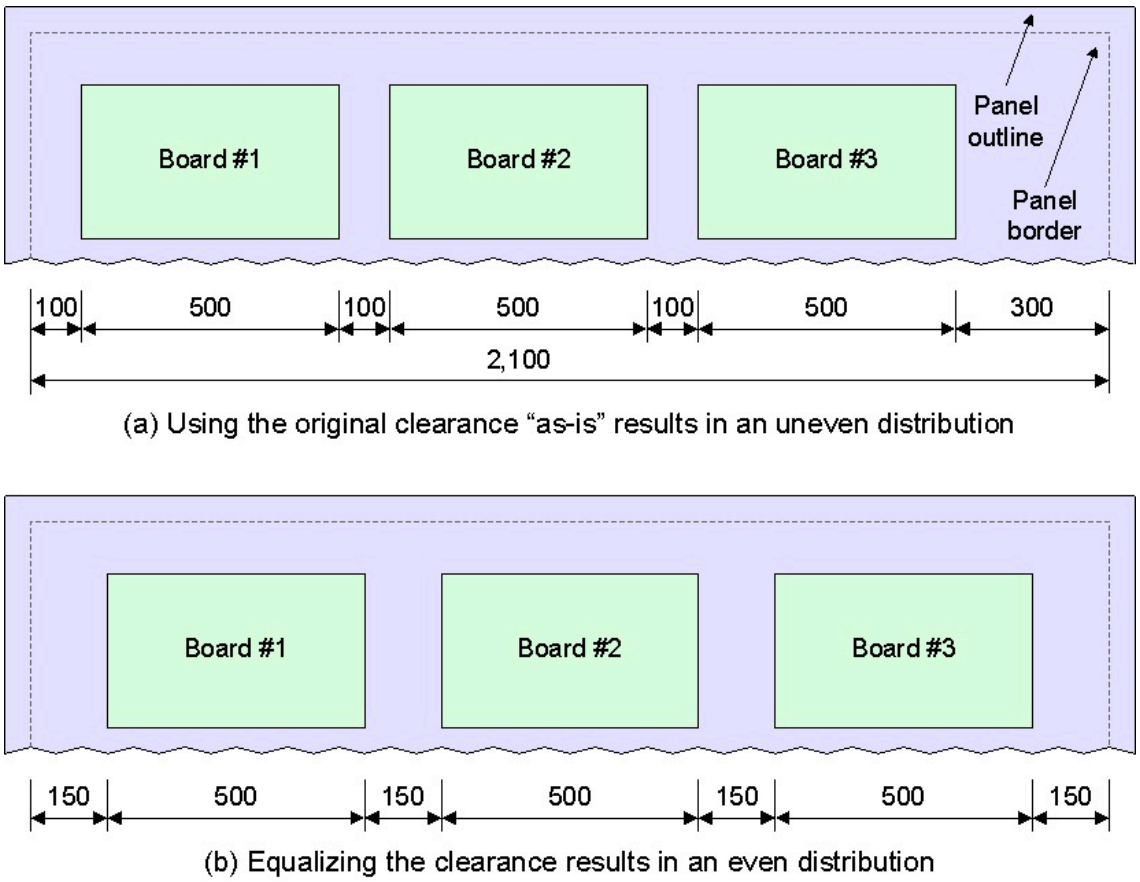
```
93      ' Get the optimal matrix in terms of column count
94      ' row count and orientation
95      Dim numColInt, numRowInt, orientationInt
96      Call FindBestMatrix(boardXDimReal, boardYDimReal,
                    clearanceReal, borderXDimReal, borderYDimReal,
                             numColInt, numRowInt, orientationInt)
```

The end-user originally specified a minimum required clearance value, which fits the boards in both the horizontal and vertical directions. Based on the combination of this clearance value and the size of the boards, the FindBestMatrix() Subroutine calculates the maximum number of boards that fit in the horizontal direction and the maximum that fit in the vertical direction.

It is extremely unlikely that the size of the boards and the clearance values that exactly fit the space inside the panel border. As an example, assume that the original clearance specified by the user is 100 units, that the boards are 500 units wide, and that the width of the panel (border to border) is 2,100 units. In this case, if there is a clearance of 100 on the left of the panel, then there is an uneven distribution with a clearance of 300 on the right of the panel as illustrated in Figure 218: Equalizing the Clearance(a).

For this script, it is better to equalize the clearances so as to provide a more uniform distribution as illustrated in Figure 218: Equalizing the Clearance(b). Of course, you must apply a similar equalization with regard to the height of the panel, and it is unlikely that the new vertical clearances are the same as the horizontal clearances. Therefore, although you start with a single clearance value, you now need to generate new equalized clearance values in the X and Y dimensions (the variables used to hold these values in the script are named *xClearanceReal* and *yClearanceReal*, respectively).

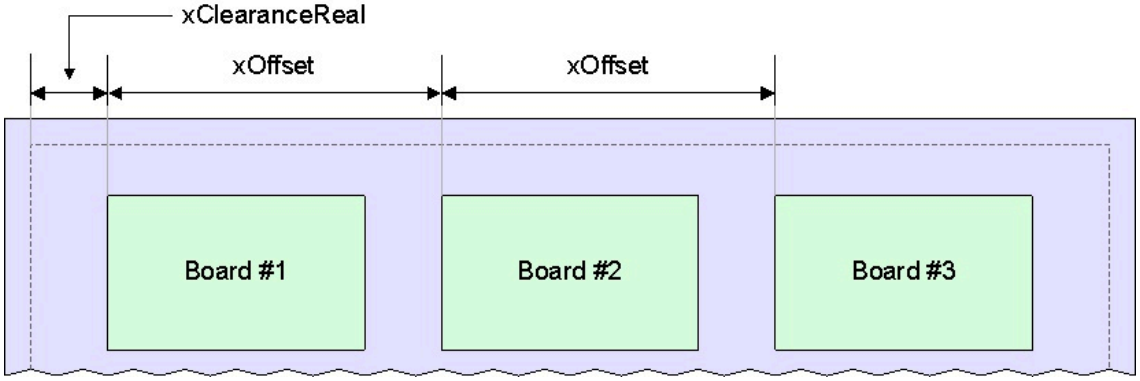
Figure 218. Equalizing the Clearance



In the following lines of code, the script instantiates and uses variables called *xOffset* and *yOffset*. In order to understand how these are used, consider the illustration in Figure 219: Concept of "Offsets". In this case, the new horizontal clearance value (*xClearanceReal*) determines the location of the left-hand edge of the first board.

The left-hand edge of the second board is calculated as an offset from the left-hand edge of the first board (where this offset is the sum of the width of the board plus the new horizontal clearance value); the left-hand edge of the third board is calculated as an offset from the left-hand edge of the second board; and so forth for subsequent boards (if there are any). A similar treatment is performed in the vertical direction using the new vertical clearance value.

Figure 219. Concept of "Offsets"



Lines 100 and 101 instantiate the variables to use to hold the new X and Y clearance values and offsets.

```
 98      ' Get the offset and equalize the clearance
 99      ' accounting for orientation.
100      Dim xClearanceReal, yClearanceReal
101      Dim xOffSet, yOffSet
```

Line 102 checks if the orientation of the boards returned by the FindBestMatrix() Subroutine is 0 or 90, corresponding to 0º or 90º. If the orientation is 0, then Lines 103 through 106 calculate the new X and Y clearance values and X and Y offsets accordingly.

Line 103 calls the EqualizeClearance() Function to calculate the new X clearance. The parameters passed into this routine are the X dimension (the *width*) of the board; the X dimension (the border-to-border *width*) of the panel; and the number of columns returned by the FindBestMatrix() Subroutine.

Similarly, Line 104 calls the same EqualizeClearance() Function to calculate the new Y clearance. In this case, the parameters passed into the routine are the Y dimension (the *height*) of the board; the Y dimension (the border-to-border *height*) of the panel; and the number of rows returned by the FindBestMatrix() Subroutine.

Line 105 calculates the X offset as the sum of the X dimension (the *width*) of the board and the new X clearance. Line 106 calculates the Y offset as the sum of the Y dimension (the *height*) of the board and the new Y clearance.

```
102      If orientationInt = 0 Then
103          xClearanceReal = EqualizeClearance(boardXDimReal,
                                        borderXDimReal, numColInt)
104          yClearanceReal = EqualizeClearance(boardYDimReal,
                                        borderYDimReal, numRowInt)
105          xOffSet = boardXDimReal + xClearanceReal
106          yOffSet = boardYDimReal + yClearanceReal
```

By comparison, if the orientation of the boards in 90, then Lines 108 through 111 calculate the new X and Y clearance values and X and Y offsets accordingly.

In this example, the boards are rotated by 90º. Therefore, when the script calls the EqualizeClearance() Function to calculate the new X clearance, the script passes in the Y dimension (the *height*) of the board. Similarly, when the script calls the EqualizeClearance() Function to calculate the new Y clearance, the script passes in the X dimension (the *width*) of the board. Additionally, Line 110 calculates the X offset as

the sum of the Y dimension (the *height*) of the board and the new X clearance; Line 111 calculates the Y offset as the sum of the X dimension (the *width*) of the board and the new Y clearance

```
107      Else
108          xClearanceReal = EqualizeClearance(boardYDimReal,
                                        borderXDimReal, numColInt)
109          yClearanceReal = EqualizeClearance(boardXDimReal,
                                        borderYDimReal, numRowInt)
110          xOffSet = boardYDimReal + xClearanceReal
111          yOffSet = boardXDimReal + yClearanceReal
112      End If
```

Finally, Lines 116 through 125 use the new X and Y clearance values and X and Y offsets in conjunction with two nested loops and the *PutBoard* method to add the boards into the panel.

```
113
114      ' Add the board matrix according to the
115      ' specified parameters.
116      Dim i, j, currXReal, currYReal
117      currYReal = borderExtremaObj.MinY + yClearanceReal
118      For i = 1 To numRowInt
119          currXReal = borderExtremaObj.MinX + xClearanceReal
120          For j = 1 To numColInt
121              Call pnlDocObj.PutBoard(designNameStr, currXReal,
                     currYReal, orientationInt, False,
                                    epcbBoardAttachPointLowerLeft)
122              currXReal = currXReal + xOffSet
123          Next
124          currYReal = currYReal + yOffSet
125      Next
126  End Sub
```

Utility Functions and Subroutines

This section declares utility routines. These routines are similar in context to the generic "helper" routines, except that they are specific to this script and may not be used elsewhere without changes.

FindBestMatrix() Subroutine

Lines 143 through 162 declare the routine that determines the optimal (maximum) number of boards that can fit into a panel. This is currently a *utility* routine, one could argue that it should be offered as a *helper* routine, because it is written in a very generic way.

The reason for this is that this routine does not understand anything about "boards" and "panels". Instead, you provide it with the X and Y dimensions associated with some "item", a required clearance value, and the X and Y dimensions of a "border". The routine then determines how many "items" (including clearance values) fit within the boundary of the "border". As part of this task, the routine performs its evaluations with the "items" presented at both 0º and 90º rotations to determine which returns the best result. If the results are equal the routine defaults to the 0º rotation of the "items".

The parameters to this routine are the X and Y dimensions of the "items"; the minimum required clearance between adjacent items and between items and the "border"; the X and Y dimensions of the border; the number of columns, the number of rows, and the rotation of the items (the last three parameters are *In-Out* parameters whose values are specified by this routine).

```
131  ' Calculates optimal matrix given dimensions of elements to
132  ' to place and dimensions of the container (rectangles assumed)
133  ' Results are returned in the form of the number of columns,
134  ' the number of rows, and the orientation of the elements.
135  ' itemXDimReal - Real
136  ' itemYDimReal - Real
137  ' clearanceReal - Real
138  ' borderXDimReal - Real
139  ' borderYDimReal - Real
140  ' numColInt - Int (In/Out)
141  ' numRowInt - Int (In/Out)
142  ' orientationInt - Int (In/Out)
143  Sub FindBestMatrix(itemXDimReal, itemYDimReal, clearanceReal,
             borderXDimReal, borderYDimReal, numColInt, numRowInt,
                                                   orientationInt)
```

Line 145 assumes that a rotation of 0º provides the best fit.

```
144      ' Try orientation of 0 degrees first.
145      orientationInt = 0
```

Line 146 calls the FindMaxFit() Function to determine the maximum number of columns; that is, the number of "items" (plus clearances) that fit in the "border" in the horizontal direction. Because the script is currently working with a rotation of 0º, the first parameter to the FindMaxFit() Function is the X dimension (the *width*) of the "item".

Similarly, Line 147 calls the FindMaxFit() Function to determine the maximum number of rows; that is, the number of "items" (plus clearances) that fit in the "border" in the vertical direction. Because the script is currently working with a rotation of 0º, the first parameter to the FindMaxFit() Function is the Y dimension (the *height*) of the "item".

```
146      numColInt = FindMaxFit(itemXDimReal, clearanceReal,
                                                   borderXDimReal)
147      numRowInt = FindMaxFit(itemYDimReal, clearanceReal,
                                                   borderYDimReal)
```

Next, the script determines the result of rotating the items through 90º. In order to do this, Line 152 first instantiates two variables to hold temporary column and row values.

```
149      ' Try orientation of 90 degrees.  This is accomplished
150      ' by using the vertical board dimension with the horizontal
151      ' panel border dimension
152      Dim tempNumColInt, tempNumRowInt
```

Line 153 calls the FindMaxFit() Function to determine the maximum number of columns. Because the script is assuming the "items" have been rotated by 90º, the first parameter to the FindMaxFit() Functionis the Y dimension (the *height*) of the "item".

Similarly, Line 154 calls the FindMaxFit() Function to determine the maximum number of rows. Because the script is assuming the "items" have been rotated by 90º, the first parameter to the FindMaxFit() Function is the X dimension (the *width*) of the "item".

```
153      tempNumColInt = FindMaxFit(itemYDimReal, clearanceReal,
                                                     borderXDimReal)
154      tempNumRowInt = FindMaxFit(itemXDimReal, clearanceReal,
                                                     borderYDimReal)
```

Line 156 checks if the product of the number of rows and columns for a 90º rotation is greater than the number of rows and columns for a 0º rotation. If true, then Lines 157 through 159 set the values of the column, row, and rotation parameters that are returned by this routine to be the values associated with a 90º rotation of the items.

```
156      If tempNumColInt * tempNumRowInt >
                                      numColInt * numRowInt Then
157          numColInt = tempNumColInt
158          numRowInt = tempNumRowInt
159          orientationInt = 90
160      End If
161
162  End Sub
```

Helper Functions and Subroutines

This script section declares several generic "helper" routines. These routines can be used in other scripts and are not focused on this script exclusively.

GetBoardDimensions() Subroutine

Lines 172 through 202 declare the routine that determines the X and Y dimensions of the board in Xpedition Layout. There are three parameters to this routine: the path/filename string for the board in Xpedition Layout, and the X/Y dimensions of the board (the last two parameters are In-Out parameters whose values is assigned by this routine).

```
167  ' Returns the dimenstions of a board given the .pcb filename
168  ' Opens Expedition to get this data.
169  ' docNameString - String
170  ' xDimReal - Real (In/Out)
171  ' yDimReal - Real (In/Out)
172  Sub GetBoardDimensions(docNameStr, xDimReal, yDimReal)
```

Lines 173 and 174 instantiate variables to hold the Xpedition Layout *Application* and *Document* objects.

```
173      Dim pcbAppObj
174      Dim pcbDocObj
```

Line 177 creates the Xpedition Layout server. Line 178 prevents any trivial messages from being displayed.

```
176      ' Create Expedition.
177      Set pcbAppObj =
                    CreateObject("MGCPCB.ExpeditionPCBApplication")
178      pcbAppObj.Gui.SuppressTrivialDialogs = True
```

Lines 181 and 182 open the Xpedition Layout *Document* object and licenses it (validate the server).

```
180      ' Open and license the document
181      Set pcbDocObj = pcbAppObj.OpenDocument(docNameStr)
182      ValidateServer(pcbDocObj)
```

Line 185 sets the units associated with this Xpedition Layout board to be the same as the units associated with the Xpedition FabLink panel.

```
184      ' Set the units.
185      pcbDocObj.CurrentUnit = unitsEnum
```

Lines 189 and 190 acquire the manufacturing outline extrema associated with the board in Xpedition Layout.

```
187      ' Use the manufacturing outline to calculate the
188      ' dimensions of the board.
189      Dim extremaObj
190      Set extremaObj = pcbDocObj.ManufacturingOutline.Extrema
```

Lines 192 and 193 use these extrema to calculate the horizontal and vertical dimensions (the width and height) of the manufacturing outline.

```
192      xDimReal = extremaObj.MaxX - extremaObj.MinX
193      yDimReal = extremaObj.MaxY - extremaObj.MinY
```

Line 196 uses the *Quit* method to exit Xpedition Layout.

```
195      ' Exit Expedetion
196      Call pcbAppObj.Quit()
```

Lines 199 and 200 explicitly release the Xpedition Layout *Application* and *Document* objects.

```
198      ' Explicitly release all references.
199      Set pcbAppObj = Nothing
200      Set pcbDocObj = Nothing
201
202  End Sub
```

EqualizeClearance() Function

Lines 209 through 213 declare the routine that recalculates (equalizes) the clearances. There are three parameters to this function: the size of a generic "item" (in this script, this is the width or height of the board, but it could be anything); the total size available (in this script, this is the border-to-border width or height of the panel, but it could be anything); and the number of "items" to fit into the total size available.

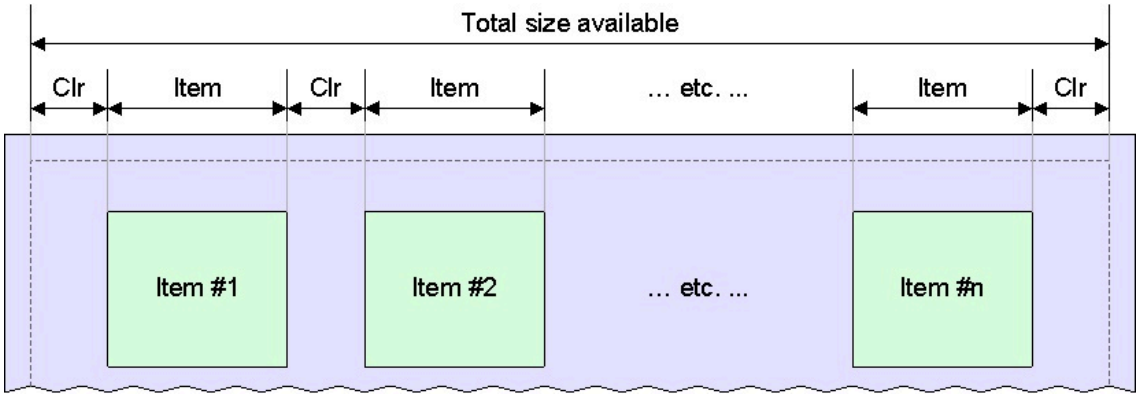
```
204  ' Returns the equal clearance that can be used to space
205  ' numItemInt of itemWidthReal within totalWidthReal.
206  ' itemSizeReal - Real
207  ' totalSizeReal - Real
208  ' numItemsInt - Int
209  Function EqualizeClearance(itemSizeReal, totalSizeReal,numItemsInt)
```

Line 211 calculates the new (equalized) clearance value.

```
211    EqualizeClearance = ( totalSizeReal - (itemSizeReal *
                                numItemsInt) )  / ( numItemsInt + 1 )
212
213  End Function
```

The formula used to calculate the new clearance value is simple, however, tricky to understand at first. Therefore, consider the illustration presented in Figure 220: Basis for the "Equalize Clearance" Algorithm.

Figure 220. Basis for the "Equalize Clearance" Algorithm



The value of the total size available and the size and number ('n') of the items are known. Therefore, the space available for any clearances is calculated as follows:

```
Remaining space available = Total size available – (size of items x n)
```

If there are 'n' items, then there are 'n + 1' clearances (one between the border and first board, one between each pair of adjacent boards, and one between the last board and the border). Therefore, the size of the new clearance is calculated as follows:

```
        Clearance = Remaining space available / (n + 1)
```

These two equations are captured in a single formula in Line 211.

FindMaxFit() Function

Lines 220 through 224 declare the routine that determines the maximum number of items that fit in a given distance with a specified clearance. There are three parameters to this function: the size of a generic "item" (in this script, this is the width or height of the board, but it could be anything); the minimum required clearance between adjacent items and between items and the outside border; and the total size available (in this script, this is the border-to-border width or height of the panel, but it could be anything).

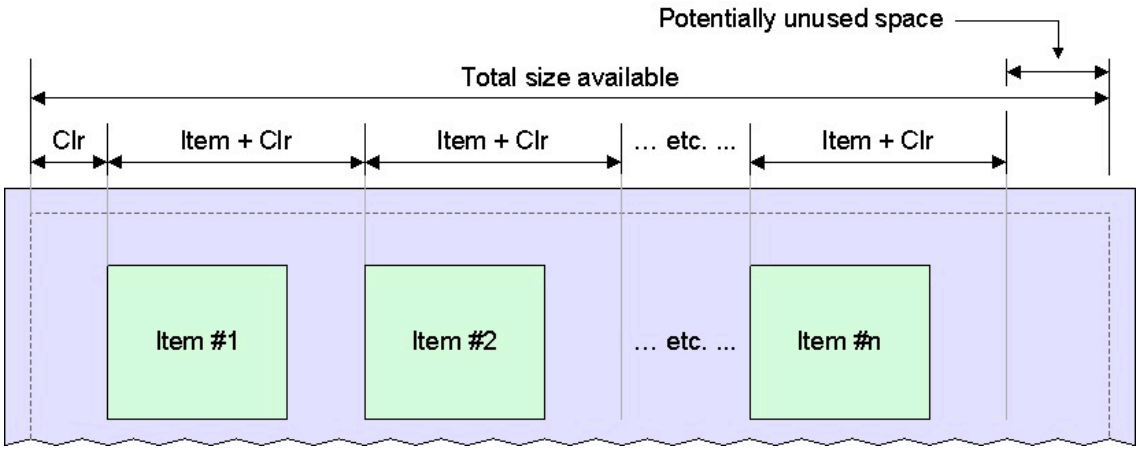
```
215  ' Returns the maximum number of items that will fit in
216  ' the total size with the given clearance.
217  ' itemSizeReal - Real
218  ' clearanceReal - Real
219  ' totalSizeReal - Real.
220  Function FindMaxFit(itemSizeReal, clearanceReal, totalSizeReal)
```

Line 222 calculates the maximum number of "items" that fit within the specified dimension.

```
221
222      FindMaxFit = Int((totalSizeReal - clearanceReal) /
                                    (clearanceReal + itemSizeReal))
223
224  End Function
```

This equation is simple, however it can be tricky to understand at first. Therefore, consider the illustration presented in Figure 221: Basis for the "Maximum Number" Algorithm.

Figure 221. Basis for the "Maximum Number" Algorithm



The value of the total size available, the size of the "item", and the minimum required clearance are known. Assume starting on the left-hand side of the total size available. As a starting point, there is a clearance on the left hand side, which means that the remaining space available is calculated as follows:

```
Remaining space available = Total size available – Clearance
```

Next, for every "item" there is an associated clearance on the right-hand side as shown in Figure 221: Basis for the "Maximum Number" Algorithm. Therefore, the number of items ("Real n") that fit in the remaining space available is:

```
        Real n = Remaining space available / (Size of item + Clearance)
```

This returns a real number such as 3.12 (hence the use of the term 'Real n'), where the fractional 0.12 equates to the "potentially unused space" shown in Figure 221: Basis for the "Maximum Number" Algorithm. In order to determine the whole (integer) number of boards ('n'), use the *Int()* function as follows:

```
        n = Int (Real n)
```

These three equations are captured in a single formula in Line 222.

FindInWDIR() Function

The *FindInWDIR()* helper function accepts a single parameter, which is the name of a file (just the file name and extension, not the path to that file).

```
226  ' Finds specified file in WDIR directory. Returns full path.
227  ' filenameStr - String
228  Function FindInWDIR(filenameStr)
```

Line 229 creates a *FileSystemObject* object.

```
229      Dim fileSysObj: Set fileSysObj =
                            CreateObject("Scripting.FileSystemObject")
```

Line 231 specifies the separator character (a semicolon) used in environment variables when running on the Windows operating system.

```
231      Dim separatorStr: separatorStr = ";"
```

Consider a simple WDIR value comprising two values (paths) separated by a semicolon as illustrated in Figure 222: WDIR Example.

Figure 222. WDIR Example

However, the separator character employed by the UNIX or Linux operating systems is a colon. Therefore, Lines 232 through 234 use the *Scripting.IsUnix* method to determine if the script is running on UNIX or Linux and if so, to change the separator character to be a colon (the *Scripting.IsUnix* method is discussed in "Exercise 6: Generating a Component CSV File" on page 374).

```
232      If Scripting.IsUnix Then
233          separatorStr = ":"
234      End If
```

Line 235 uses the separator character to split the WDIR environment variable and to create an array of directory/folder paths.

```
235      Dim wdirArr: wdirArr =
               Split(Scripting.GetEnvVariable("WDIR_<release>"),
 separatorStr)
```

Where *<release> is the root name of the product release or other unique identifier.*

Lines 237 through 243 iterate through all of these directory/folder paths checking if the required library file exists in any of them. When the file is found, the script returns its complete path/name and exits the function.

```
236      Dim i
237      For i = 0 To UBound(wdirArr)
238          Dim pathStr: pathStr = wdirArr(i) & "/" & filenameStr
239          If fileSysObj.FileExists(pathStr) Then
```

```
240                 FindInWDIR = pathStr
241                 Exit Function
242             End If
243         Next
```

If the script does not find the specified library file in any of the WDIR directory/folder paths, then assume that the *filenameStr* parameter to this function defined an absolute path to the library file, in which case this is the name the script returns as specified on Line 244.

```
244         FindInWDIR = filenameStr
245     End Function
```

Placement Parameters Dialog

The procedure for creating the dialog is to:

- create the dialog
- populate the various elements on the dialog with their portions of the script

The key scripting sections "behind" the form are as follows (these are shown in the order in which they are presented):

- Initialization and Setup
- "OK" Button Event/Click
- "Cancel" Button Event/Click
- "Close" (Terminate) Event/Click

Adding Elements to the Form

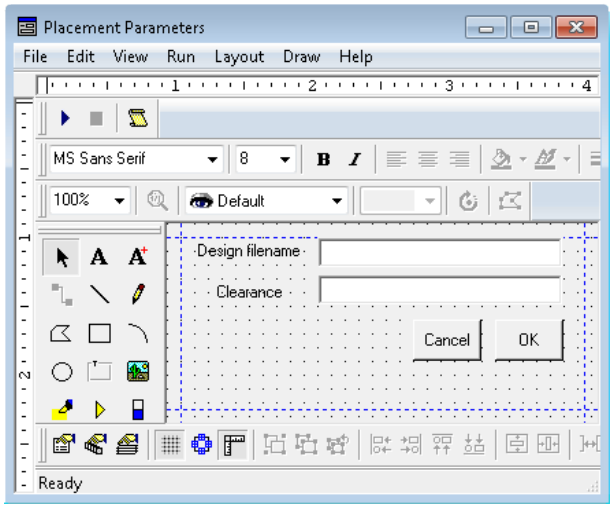
1. Launch the IDE in form editor mode as discussed in “Creating Forms and Dialogs Using the IDE” on page 69 and then create the form as illustrated in Figure 223: Creating the Placement Parameters Form.

2. Choose *File* > *Save As* to save the form in the folder you are using to store your scripts; save the form with the name *PlaceBoardInput.efm*

Note:
The **.efm* extension denotes a form, as compared to the **.vbs* extension used to denote a pure VBScript.

Tip
: As you add the code behind the elements on the form, it is a good idea to continually save your work.

Figure 223. Creating the Placement Parameters Form



Note that there are two text boxes on this form. Each text box has a free text item placed to its left side: one labeled **Design filename** and the other labeled **Clearance**. At the bottom of the form are two buttons with text annotations of **OK** and **Cancel**.

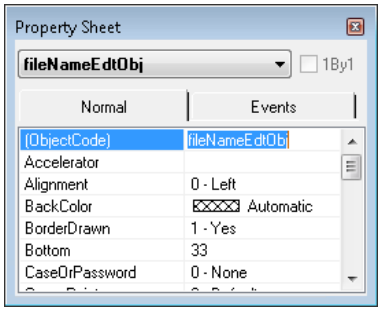
Naming Elements on the Form

In this procedure you will name any of the objects on the form with which you associate scripting code.

1. Right-click on the upper text boxes and select the **Object Properties** item from the resulting pop-up menu.

2. Change the assignment to the **(Object Code)** item from its default value to the name *fileNameEditObj* as illustrated in Figure 224: Object Property Sheet Dialog.

Figure 224. Object Property Sheet Dialog



Note that this name automatically appears in the field at the top of the form. This is the name that the script uses to access this text box element.

3. Click the red 'X' button located in the upper right-hand corner to dismiss this dialog.

Repeat this procedure for the other items on the form as follows:

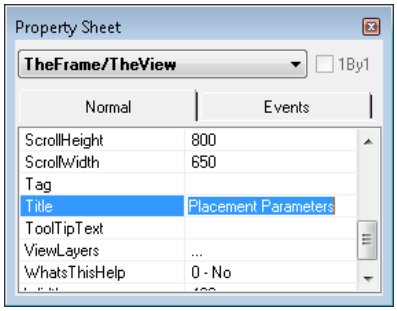
- Change the name of the lower text box control to *clearanceEditObj*.
- Change the name of the **Cancel** button to *cancelBtnObj*.
- Change the name of the **OK** button to *okBtnObj*.

Naming the Form

In this procedure you are change the title of the form from its default to “Placement Parameters”.

1. Right-click on any blank area on the form and select the **Object Properties** item from the resulting pop-up menu.
2. Change the assignment to the **Title** item from its default value to *Placement Parameters* as illustrated in Figure 225: Form Property Sheet Dialog .

Figure 225. Form Property Sheet Dialog



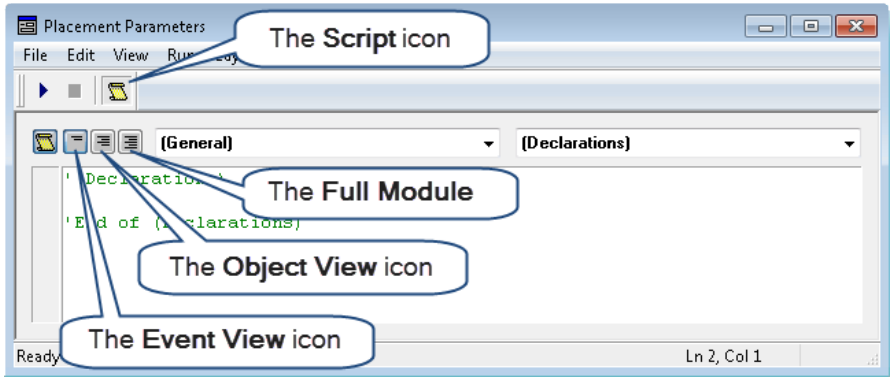
3. Click the red 'X' button located in the upper right-hand corner to dismiss this dialog.

Adding Code "Behind" the Elements on the Form

In this procedure you add the code (sub-scripts) "behind" the elements on the form. There are a number of mechanisms by which this may be achieved.

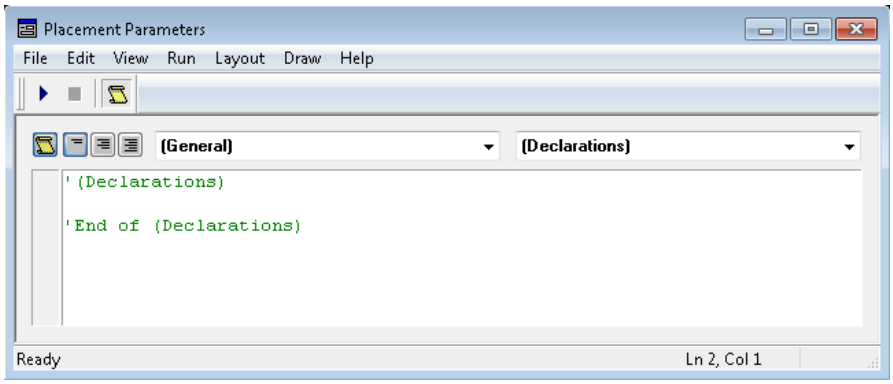
1. Click the **Script** icon and then assure that the **Event View** icon is active as illustrated in Figure 226: Adding Code "Behind" the Elements on the Form.

Figure 226. Adding Code "Behind" the Elements on the Form



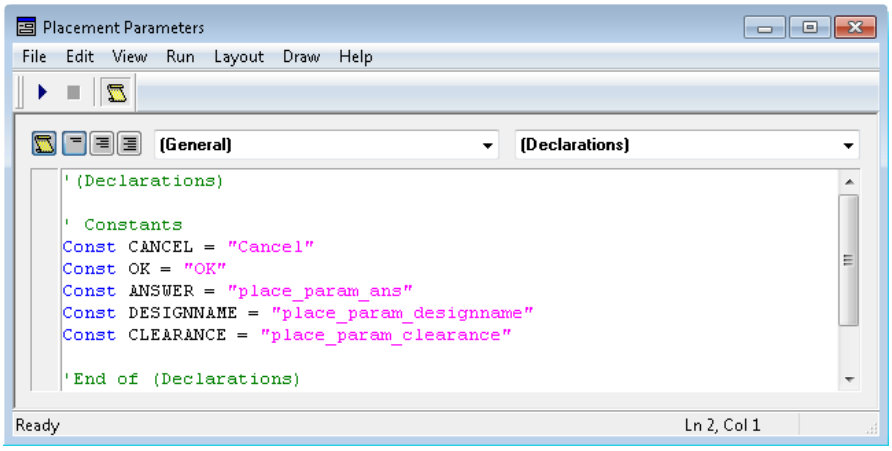
2. Assure that **(General)** is selected in the left-hand (Object) list at the top of the form. This causes the right-hand list to be automatically populated with the **(Declarations)** item as illustrated in Figure 227: Adding the Initialization/Setup Code.

Figure 227. Adding the Initialization/Setup Code



3. Place the cursor between the *'(Declarations)* and *'End of (Declarations)* comments and enter the code shown in Figure 228: Initialization and Setup Code.

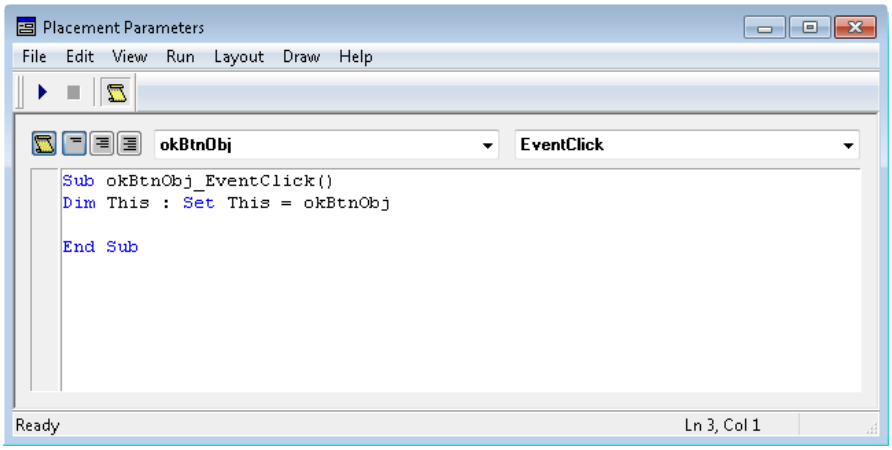
Figure 228. Initialization and Setup Code



Note that these are exactly the same constant values declared in the main script and that these are the keys that implement script/form-to-script/form communications by means of the *Scripting.Globals Dictionary* object.

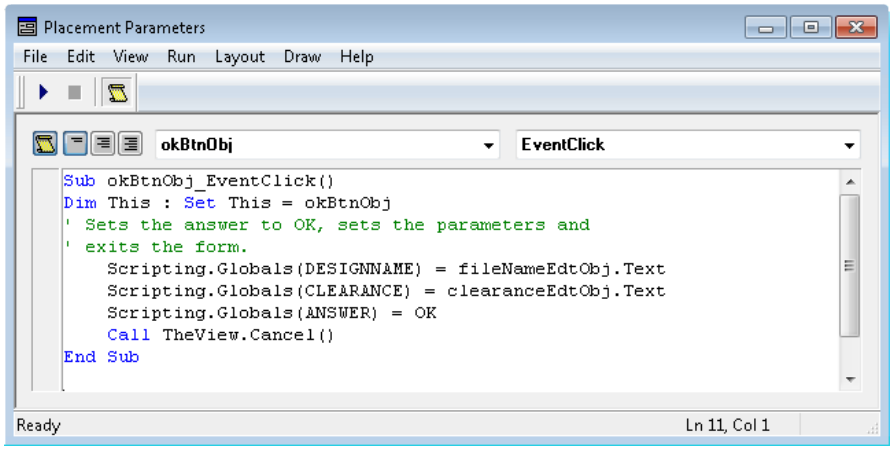
4. Assure that **okBtnObj** is selected in the left-hand (Object) list at the top of the form; also that **EventClick** is selected in the right-hand list at the top of the form. The result is as illustrated in Figure 229: Adding the OK Button Code.

Figure 229. Adding the OK Button Code



5. Place your cursor on the blank line between the *Dim* and *End Sub* statements and then enter the code shown in Figure 230: OK Button Code.

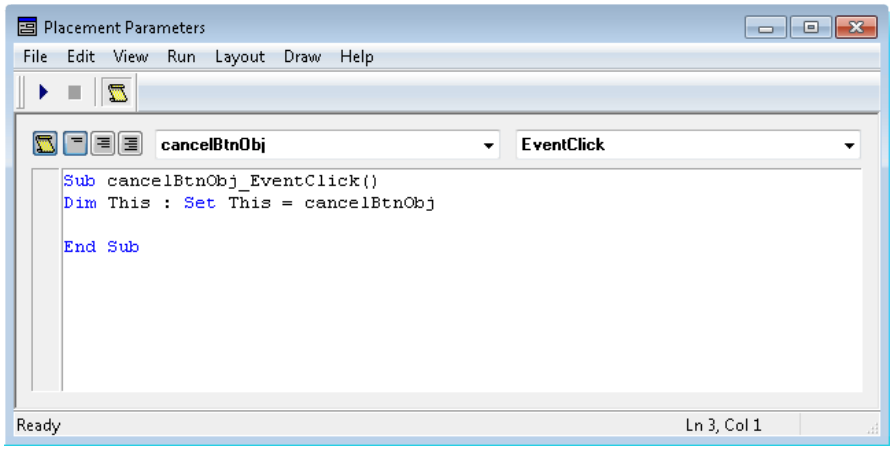
Figure 230. OK Button Code



Note that the script first uses the *Scripting.Globals Dictionary* object to associate the *DESIGNNAME* key with the contents of whatever text string the user enters into the upper text box; next it associates the *CLEARANCE* key with the contents of whatever text string the user enters into the lower text box; next it associates the *ANSWER* key with a text string value of "*OK*"; and finally the script uses the *Cancel* method associated with *TheView* to exit the form. *TheView* refers to the form itself.

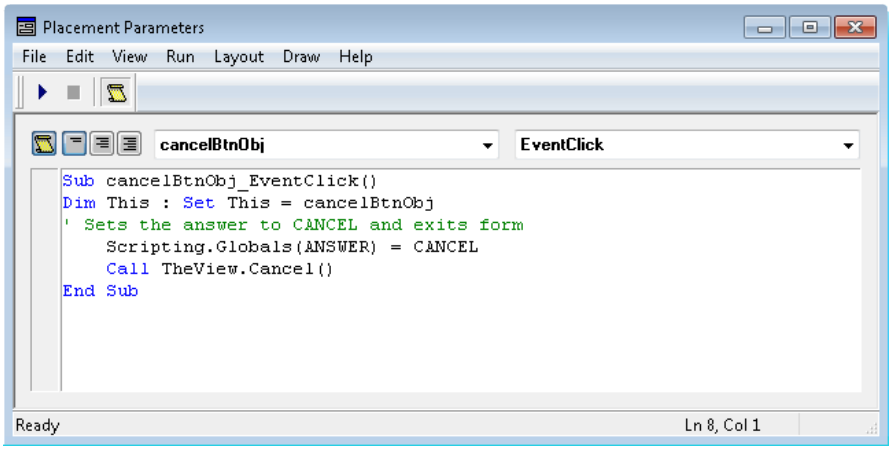
6. Ensure that **CancelBtnObj** is selected in the left-hand (Object) list at the top of the form; also that **EventClick** is selected in the right-hand list at the top of the form. The result is as illustrated in Figure 231: Adding the Cancel Button Code.

Figure 231. Adding the Cancel Button Code



7. Place your cursor on the blank line between the *Dim* and *End Sub* statements and then enter the code shown in Figure 232: Cancel Button Code.

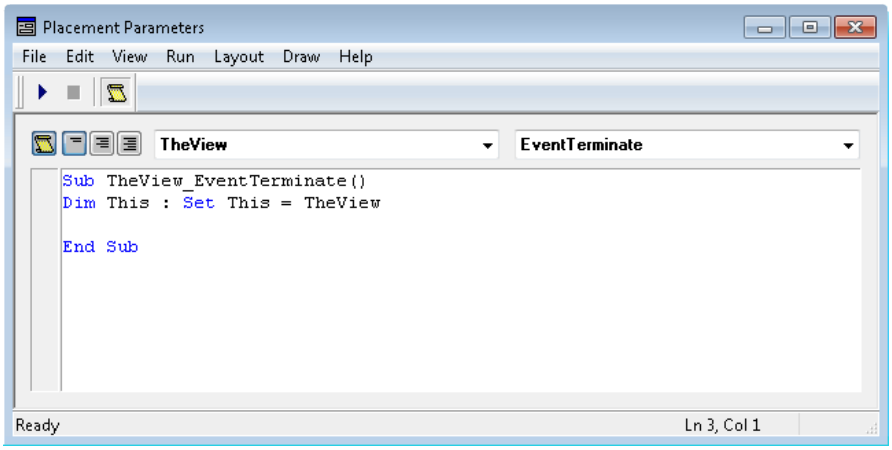
Figure 232. Cancel Button Code



The script first uses the *Scripting.Globals Dictionary* object to associate the *ANSWER* key with a text string value of "*Cancel*". Next, the script uses the *Cancel* method associated with *TheView* to exit the form.

8. Ensure that **TheView** is selected in the left-hand (Object) list at the top of the form (this refers to the form itself); also that **EventTerminate** is selected in the right-hand list at the top of the form. The result is as illustrated in Figure 233: Adding the Close (Terminate) Button Code.

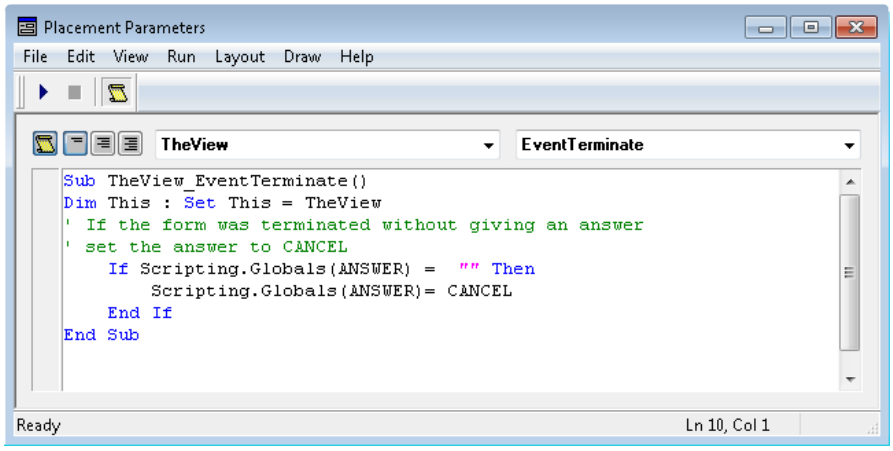
Figure 233. Adding the Close (Terminate) Button Code



The *EventTerminate* event associated with *TheView* is called whenever the form exits. This is independent of the mechanism and means that this event is called whenever the end-user clicks the **OK** or **Cancel** buttons (each of which uses the *Cancel* method associated with *TheView* to exit the form) or the **Close** (terminate) button in the upper-right-hand corner of the form.

9. Place your cursor on the blank line between the *Dim* and *End Sub* statements and then enter the code shown in Figure 234: Close (Terminate) Button Code.

Figure 234. Close (Terminate) Button Code



10. Note that the script first uses the *Scripting.Globals Dictionary* object to check if the *ANSWER* key is associated with an empty string. If so, this means that the end-user clicked the **Close** (Terminate) in the upper-right-hand corner of the form, in which case the script associates the *ANSWER* key with whatever string is assigned to the *CANCEL* constant ("Cancel" in this script and form).

11. Save the results.

Creating and Testing the Script

1. Use the scripting editor of your choice to enter the script described (including the license server function) and save it as *PlaceBoards.vbs*.

2. Use the IDE to create the dialog/form described and save it as *PlaceBoardInput.efm*. Ensure that this file is in one of your WDIR folders/paths before running the *PlaceBoards.vbs* script.

3. Close any instantiations of Xpedition Layout and Xpedition FabLink that are currently running, and then launch a new instantiation of Xpedition FabLink.

4. Open the *CandyPanel.pnl* design.

5. Delete the six existing boards from the panel.

6. Drag-and-drop the *PlaceBoards.vbs* script into the Xpedition FabLink design area.

7. Enter the path and filename to the *Candy.pcb* on your system in the **Design filename** field, enter a clearance value of 100 in the **Clearance** field, and click the OK button.

8. Observe that nine boards (three rows of three columns) appear in the panel.

Enhancing the Script

There are a number of ways in which this script could be enhanced. Some suggestions are as follows (it is a good idea for you to practice your scripting skills by implementing these examples):

- For the pop-up dialog/form, you could add some error checking to ensure that the end-user enters appropriate values. For example, that the value entered into the Clearance text box is a number.

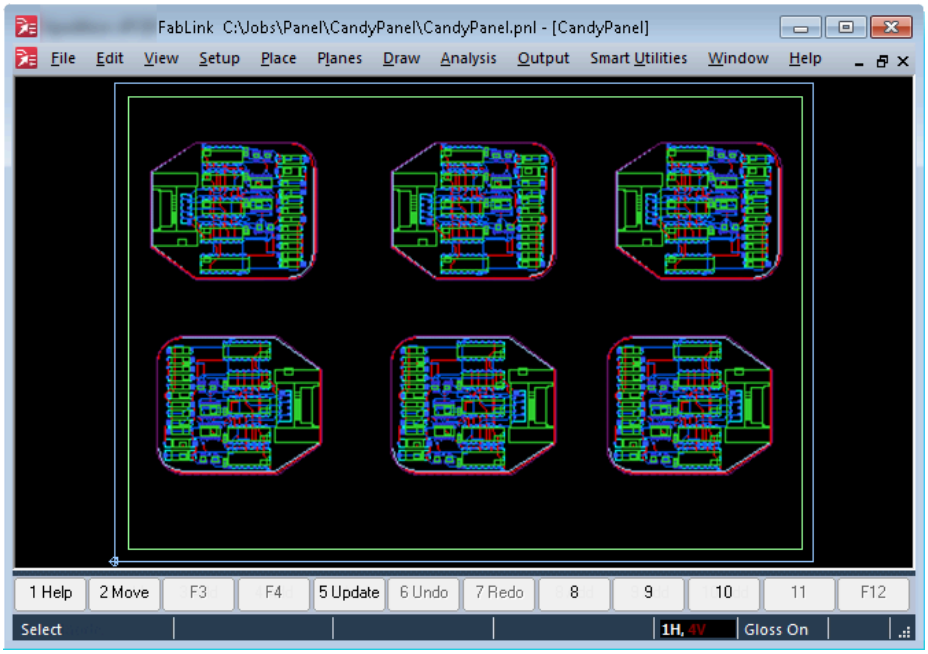
- For the pop-up dialog/form, you have to specify the full path to the Xpedition Layout board design. It is useful to add a Browse button to this dialog/form that allows you to easily locate and select a board design file (this is relatively easy on Windows because there is already an automation server Browse dialog available; implementing this functionality on UNIX or Linux is a little more challenging!)

- The script presents all of the boards in the same orientation; that is, either 0º or 90º. However, there could be a process requirement to have different boards in different rotations. Consider the default CandyPanel panel, for example, in which the top row of boards are presented with 0º rotation and the bottom row of boards are presented with 180º rotation as illustrated in Figure 235: Pre-defined CandyPanel Panel.

  Once the script has determined the optimal number of boards, it could automatically flip alternate rows or columns. If the script initially decided that the optimal fit was N-rows by M-columns with a 0º rotation, for example, then it could flip the boards on odd rows to have a rotation of 180º; or it could flip the boards on odd columns to have a rotation of 180º; or it could create a checkerboard pattern of boards at 0º and 180º.

  Alternatively, if the script initially decided that the optimal fit was N-rows by M-columns with a 90º rotation, then it could flip the boards on odd rows to have a rotation of 270º (or –90º) or it could flip the boards on odd columns to have a rotation of 270º (or –90º); or it could create a checkerboard pattern of boards at 90º and 270º.

Figure 235. Pre-defined CandyPanel Panel



- The pop-up dialog/form could be modified to allow the user to select whether all of the boards have the same rotation (this could be the default), or whether to flip alternate rows, or to flip alternate columns, or to create a checkerboard pattern.

- The script could be enhanced to perform a better "best fit". For example, it may be possible to fit more boards on the panel if some rows used a 0º board rotation while others used a 90º board rotation (it is best to implement this using a loop that tried boards with differing rotations as opposed to a formula as is the case in the existing script).

Chapter 16
Recommended Scripting Practices

This section presents recommended scripting practices. "Good scripting practices" encompasses many things, such as employing a standard naming convention and doing "things" a certain way.

In the case of a scripting language like VBScript, you are often presented with different options; for example, you may include or omit a space between a method name and the parentheses surrounding any parameters of that method. In such a case, it is not important which option you choose, it is important to be consistent.

The recommended practices in this appendix have evolved over many engineer-years and the writing of many scripts. It is recommended that you use these practices, because doing so facilitates sharing your scripts with others and understanding the scripts others create (assuming that they also choose to use these practices).

Naming Conventions
Option Explicit Statement
Comments
Parentheses
Space Characters
Dim and Dim()

Naming Conventions

Using a standard naming convention provides obvious advantages with regard to sharing scripts with — and understanding scripts from— third parties; everything is much easier if everyone is following the same convention.

Variable Names
Function and Subroutine Names
Method and Property Names
Event-Handler Function Names
Object Names in IDE Forms
File Names for Scripts and Forms

Variable Names

VBScript is a loosely-typed language. For example, you may declare a variable and initially assign an integer value to it. Later, you may assign a real number or a string to the same variable. When you declare a variable, you usually have a certain type of assignment in mind. Therefore, in order to keep track of things and improve the legibility of your code, it is recommended that you post-fix the variable name with a short string specifying its primary type.

The following are the strings recommended and used in this tutorial:

```
BoolBoolean (e.g. 0 / 1 or True / False)' Primitive typeIntInteger (e.g.
 10)' Primitive typeRealReal number (e.g. 3.142)' Primitive typeStrString
 (e.g. "Hello World!")' Primitive typeEnumEnumerate ' Special primitive
 (named integer)ObjObject' Object typeCollCollection (of objects)' Object
 typeClsClass (user-defined object/structure)' Object typeConstA variable
 assigned a constant value' Object type
```

For example, you can name a *Trace* object, *traceObj,* and a collection of *Trace* objects, *traceColl*. Two special objects that appear at the top of the data hierarchy, and in most scripts, are the *Application* and *Document* objects. It is common to include these object types in their corresponding variable names; for example, *pcbAppObj* and *pcbDocObj*.

The first letter of a variable name should be lowercase. Each new sub-word in the variable name should be uppercase (any remaining letters should be lowercase). For example, you might name a *Trace* object that is part of a function that finds the longest trace in a design, *longestTraceObj*.

If the variable represents an array, you should append *Arr*. For example, *nameStr* indicates a single string variable, while *nameStrArr* indicates an array of string variables.

Function and Subroutine Names

Function and subroutine names are similar to variable names. The main differences are that they typically do not require the post-fix type string, and also that the first letter in the name should be uppercase. The uppercase first letter makes it easy to differentiate between variable and function/subroutine names. Consider the following statement:

```
Set longestTraceObj = FindLongestTrace(traceColl)
```

In this case, the name of the function (whose contents is described elsewhere) is *FindLongestTrace()*; observe the initial uppercase letter.

Method and Property Names

It is recommended to capitalize the first letter of method and property names. Observe the following example lines of code:

```
Dim viasColl
Set viasColl = pcbDocOBJ.Vias
```

The first letter of *Vias* is in uppercase, because *Vias* is a property that is defined in the interface.

Event-Handler Function Names

Assume that you are interested in the events that are associated with the *Document* object.Use the *AttachEvents* method of the *Scripting* object as follows:

```
Call Scripting.AttachEvents(pcbDocObj, "pcbDocObj")
```

The *AttachEvents* method instructs the script engine to respond to any events associated with the object that is specified by the first parameter. In this example, the *Document* object variable *pcbDocObj* is this parameter.

The second parameter to the *AttachEvents* method is a string. This string is a common prefix for any of the event-handler function names associated with events originating from the object assigned to the *pcbDocObj* variable. The string itself is arbitrary – you can call it "Fred" if you choose – however, it is recommended programming practice to set this string to be identical to the name of the first parameter. Because the first parameter in the example is the *pcbDocObj* variable, use "pcbDocObj" as the prefix string.

The name of an event-handler function has three parts as follows:

- The string specified as the second parameter to the *AttachEvents* method (described following).
- An underscore '_ ' character.
- The name of the event as defined in the Layout Automation Reference documentation.

The following shows an example event-handler function:

```
Function pcbDocObj_OnPreClose()
   ' Some statements go here
End Function
```

Object Names in IDE Forms

The names of objects in IDE forms are similar to the names of variables. When in the IDE's form editor mode, you can add controls to the form such as buttons, radio buttons, combo-boxes, check-boxes. As an example, you can name a *Button* object that counts all vias in a design, *countAllViasButObj* (meaning "Count All Vias Button Object").

The following abbreviations are recommended for objects used in forms:

File Names for Scripts and Forms

File names for VBScript scripts have the extension **.vbs*. File names for IDE forms have the extension **.efm*. The file name should follow the same naming rules as function and subroutine names; that is, the first letter of the file name should be uppercase and each new sub-word in the variable name should be uppercase. The following illustrates both examples:

```
HelloWorld.vbs
ViaCountForm.efm
```

For a library (see “Creating and Using a Library” on page 529), it is recommended to use the *Lib* qualifier as part of the file name, as this indicates that this is a library rather than a standalone script. The following is an example library:

```
MenuLib.vbs
```

Option Explicit Statement

By default, VBScript does not require *Dim* (dimension) statements. For the following statements the *Dim* statement is optional:

```
Dim floppyInt
floppyInt = 10
```

The problem with this is that it is easy for misspelled variables to go unnoticed in a script. For example, suppose you want to assign a new value of 20 to the *floppyInt* value as follows:

```
floopyInt = 20
```

The variable name is misspelled (it is floopyInt not floppyInt). By default, VBScript accepts this new statement and creates a new variable.

This type of problem can be extremely time-consuming to track down and resolve. Therefore, it is strongly recommended to include the Options Explicit as the first statement in your script. For example:

```
Option Explicit

Dim floppyInt
floppyInt = 10
     :
    etc.
```

Using *Option Explicit* requires you to declare your variables using *Dim* statements. In turn, this means that if you misspell a variable name, the script host reports an error and guides you directly to the problem.

Comments

Comments begin with a single quote and continue from that point to the end of the line. For example, consider the following snippet of code:

```
' This is a demo script showing the use of comments
Option Explicit  ' This optional statement is recommended

Dim floppyInt    ' You should use a lot of comments
floppyInt = 10   ' They make scripts easier to understand
```

It is recommended that a space character follow the ' character.

Parentheses

VBScript allows you to enclose function, property and method parameters in parentheses or to omit the parentheses. For example, VBScript accepts both of the following statements:

```
MsgBox "Hello World"     ' This is not recommended
MsgBox("Hello World")    ' This IS recommended
```

For consistency you should use one technique throughout your script. Using parentheses is recommended.

Space Characters

VBScript does not allow space characters in variable names. However, VBScript allows the use of a space between a function, method, or property name and the opening parenthesis enclosing any parameters. For example, VBScript accepts both of the following statements:

```
MsgBox ("Hello World")   ' This is not recommended
MsgBox("Hello World")    ' This IS recommended
```

For consistency you should use one technique throughout your script. *Not* using a space character is recommended.

Dim and Dim()

Consider the following statements:

```
Dim tempAStr      ' (a) See notes below
Dim tempBStr(3,4) ' (b) Static array (cannot be resized)
Dim tempCStr()    ' (c) Dynamic array (can be resized using ReDim)
```

The first statement, (a), appears to be a scalar (single) variable; the second statement, (b), instantiates a two-dimensional static array whose size cannot change; the third statement, (c), instantiates a dynamic array whose size can change with a *ReDim* statement.

The first and third statements are functionally identical; that is, it is possible to apply a *ReDim* statement to *tempAStr* and convert it into a single or multi-dimensional array. This is *not recommended*, because anyone reading the script assumes that *tempAStr* is a scalar variable. Instead, if you create a dynamic array, you should use parentheses when dimensioning the variable with the Dim statement, as shown with the *tempCStr()* example. This makes it obvious to anyone reading the script that this is an array whose size can change.

Chapter 17
Recommended Script Architecting Process

This section presents a number of tricks and techniques to make the script easy to understand, use, re-use, and maintain.

Creating a Template
Re-using Functions and Subroutines
Internal and External Scripts
Calling Other Scripts
Script to Script Communication
Delivering Scripts

Creating a Template

Creating a standard template (as a starting point) is one of the most effective things you can do. A simple example is illustrated:

```
 1  ' Comments describing the script and its overall function go
 2  ' here. These may include author details and a change history
 3  ' (so as to make it easy to "assign the blame" in the future).
 4
 5  Option Explicit
 6
 7  ' Constants Declarations (if any)
 8
 9  ' Add any type librarys to be used.
10  Scripting.AddTypeLibrary("MGCPCB.ExpeditionPCBApplication")
11
12  ' Global Variables
13  Dim pcbDocObj
14  Dim pcbAppObj
15
16  ' Get the application object
17  Set pcbAppObj = Application
18
19  ' Get the active document
20  Set pcbDocObj = pcbAppObj.ActiveDocument
21
22  ' License the document
23  ValidateServer(pcbDocObj)
24
41  ' Include File helper functions go here (see notes)
42
43  ' Server validation function goes here
44  :
45 Etc.
```

Observe the comment on Line 41. This is where to insert the "Include File" helper functions (these functions are described in “Creating and Using a Library” on page 529).

Re-using Functions and Subroutines

Re-use functions and subroutines to keep from "re-inventing the wheel".

You can achieve this by creating libraries of commonly used functions and subroutines and then calling these routines from your scripts (see also “Creating and Using a Library” on page 529). The two main advantages to this approach are:

- You can fix bugs that effect multiple scripts by modifying a single routine.
- Using well-defined and well-known library routines across multiple scripts, makes those scripts easier to read and maintain.

Internal and External Scripts

Generally, you should always try to use internal scripts because these have the best performance. However, you can connect to Xpedition Layout using an external script and to then run an internal script that does the performance-intensive work.

Calling Other Scripts

When creating a script, you can use *Application.ProcessScript* to invoke other scripts or forms. In turn, these "sub-scripts" and "sub-forms" can use the same technique to call other scripts and forms. This technique allows you to quickly and easily pull multiple scripts and forms together to create more complex functionality.

Script to Script Communication

Internal scripts and forms can communicate with each other using the Globals object as illustrated in the following example:

Script #1: Writes to a *Globals* object

```
Scripting.Globals("MyGlobalName") = "MyGlobalVal"
```

Script #2: Reads from a *Globals* object

```
1  MsgBox Scripting.Globals("MyGlobalName")
```

Observe that, when assigning a value to a *Globals* object as illustrated in Script #1, the parameter ("MyGlobalName" in this example) is an arbitrary string (or string variable or string constant) you define. You must choose a unique name, because anyone can use this.

Delivering Scripts

The most flexible way to add functionality to an application such as Xpedition Layout, is using one or more startup scripts. There are several approaches to use. For example:

- You can create one startup script that adds a group of menu items and/or accelerator keys, where this script then calls other scripts to implement the functionality associated with these entities (one sub-script per menu item).

- You can create a number of startup scripts, each of which adds an individual menu item or defines an accelerator key, and each of which defines the functionality associated with that particular entity.

The second approach is preferred (keeping the menu or key-binding together with its functionality), but you must choose which technique best meets your requirements. The advantage of the first approach is that it is easier to layout a menu with a large number of additions when all the changes to the menu are made in a single location. If you are doing significant menu customizing, this might be the best approach.

The most flexible place to put these scripts is in a WDIR directory. In addition, if you have a server install, you can configure all users to have a network share defined in the WDIR environment variable, in which case anything you place in that directory is automatically picked up by all users (see Running Scripts from an Application and Running Startup Scripts for more details on using the WDIR environment variable).

Chapter 18
Creating and Using a Library

This section shows you how to create and re-use a library of help functions.

You often create useful helper functions and you then re-use these helper function by copying and pasting them to new scripts. There are two problems with this approach as follows:

- As you create more and more helper-type functions, it become increasingly time-consuming to remember where they are located, to retrieve them, and to re-use them.
- If you subsequently find an error in a helper routine that you have copied into multiple scripts, it can be extremely time-consuming to fix the same bug in multiple locations. It can also be difficult to remember the scripts in which you use the routine, and it is easy to lose consistency (that is, to "fix" the bug in different ways in different scripts).

The solution is to create a library of helper functions and subroutines that you call from multiple scripts. In this case, if you detect a bug, a single modification to the library routine fixes all scripts calling that routine. Another advantage of this approach is that using well-defined and well-known library routines across multiple scripts makes those scripts easier to read and maintain.

Creating a Library
Calling Routines From a Library

Creating a Library

VBScript does not inherently support the concept of libraries, but there is a way to work around this problem.

First, create a directory/folder called *\Lib* (for library) somewhere on your system or on a server. This folder should be in a path that is defined by your WDIR environment variable (refer to Using the WDIR Environment Variable for more details).

Next, consider the *AddMenuAfter()* and *GetMenuNumber()* helper functions. Put these functions into a single file as shown below and then save this file as *MenuLib.vbs* in the *\Lib* directory. This file is implemented as a library (the *Lib* qualifier in the file name indicates that this is a library rather than a standalone script).

```
 1  ' This script is implemented as a library.
 2
 3
 4  ' Menu manipulation functions.
 5
 6  ' Creates a new menu entry, menuToAdd, on menuBar after the
 7  ' afterMenuEntry entry.  If the entry is not found the menu is
 8  ' added at the end.
 9  ' menuToAddStr - String
10  ' afterMenuEntryStr - String
11  ' menuBarObj - CommandBarSvr menu object
12  Function AddMenuAfter(afterMenuEntryStr, menuBarObj)
13      Dim entryNumInt
```

```
14      entryNumInt = GetMenuNumber(afterMenuEntryStr, menuBarObj) + 1
15      If entryNumInt > menuBarObj.Controls.Count Then
16           entryNumInt = -1
17      End If
18
19      Set AddMenuAfter =
             menuBarObj.Controls.Add(cmdControlButton,,,entryNumInt)
20  End Function
21
22  ' Function that returns a menu index for a given
23  ' menu entry string.
24  ' menuToFindStr - String
25  ' menuBarObj - CommandBarSvr menu object
26  Function GetMenuNumber(menuToFindStr, menuBarObj)
27      Dim ctrlColl : Set ctrlColl = menuBarObj.Controls
28      Dim ctrlObj
29
30      Dim menuCntInt : menuCntInt = 1
31      GetMenuNumber = -1
32
33      For Each ctrlObj In ctrlColl
34         Dim captStr: captStr = ctrlObj.Caption
35         captStr = Replace(captStr, "&", "")
36         If captStr = menuToFindStr Then
37             GetMenuNumber = menuCntInt
38             Exit For
39         End If
40         menuCntInt = menuCntInt + 1
41      Next
42  End Function
```

In this case, these routines are always used together. However, even routines that do not call each other, should be put them all into a single file if the routines serve a common or related purpose. For example, if you create any additional menu-related helper routines in the future, it might make sense to include them in this file (an example of such a routine is a function to determine if a menu item has already been added).

Calling Routines From a Library

After you create a library you call these library routines from a script. As a simple example, consider the following script. The script adds a new **Hello** item to the **View** pull-down menu. When the end-user chooses **View > Hello**, a *MsgBox()* appears on the screen displaying "Hello".

The following is the script in its entirety (apart from the standard *ValidateServer* function):

```
1  Option Explicit
2
3  Include("Lib\MenuLib.vbs")
4
5  ' add any type librarys to be used.
6  Scripting.AddTypeLibrary("MGCPCB.ExpeditionPCBApplication")
7
8  ' Get the application object
```

```
 9  Dim pcbAppObj
10  Set pcbAppObj = Application
11
12  ' Get the active document
13  Dim pcbDocObj
14  Set pcbDocObj = pcbAppObj.ActiveDocument
15
16  ' License the document
17  ValidateServer(pcbDocObj)
18
19  ' Get the document menu bar.
20  Dim docMenuBarObj
21  Set docMenuBarObj =
                  pcbAppObj.Gui.CommandBars("Document Menu Bar")
22
23  ' Get the view menu
24  Dim viewMenuObj
25  Set viewMenuObj = docMenuBarObj.Controls.Item("&View")
26
27  ' Call the library function
28  Dim helloBtnObj
29  Set helloBtnObj = AddMenuAfter("Mouse Mapping", viewMenuObj)
30
31  helloBtnObj.Caption = "Hello"
32  helloBtnObj.Target = ScriptEngine
33  helloBtnObj.ExecuteMethod = "OnHello"
34
35  Scripting.DontExit = True
36
37  Function OnHello(id)
38      MsgBox("Hello")
39  End Function
40
41  '''''''
```

```
59          separatorStr = ":"
60      End If
61      Dim wdirArr:
62      wdirArr = Split(Scripting.GetEnvVariable("WDIR_<release>"),
separatorStr)
63      Dim i
64      For i = 0 To UBound(wdirArr)
65          Dim pathStr: pathStr = wdirArr(i) & "/" & filenameStr
66          If fileSysObj.FileExists(pathStr) Then
67              FindInWDIR = pathStr
68              Exit Function
69          End If
70      Next
71      FindInWDIR = filenameStr
72  End Function
73
74  ' Server validation function
75       :
76      etc.
```

Where *<release>* is the root name of the product release or other unique identifier.

The script calls the *AddMenuAfter()* library helper function on Line 29.

```
27  ' Call the library function
28  Dim helloBtnObj
29  Set helloBtnObj = AddMenuAfter("Mouse Mapping", viewMenuObj)
```

Line 3 calls a local *Include()* helper function and passes it the name of the library file to open. In this example, the string "Lib\MenuLib.vbs". You can also specify an full path/filename.

```
 1  Option Explicit
 2
 3  Include("Lib\MenuLib.vbs")
```

The following topics cover the script's *Include()* and *FindInWDIR()* routines. It is strongly recommended that you include these routines in all of your scripts (the best way to do this is to add them into your "Template Script" as discussed in "Creating a Template" on page 525).

Include() Function

This function accepts a single parameter, which is the name of the library file to open. This parameter may either specify an absolute path to a file or a file that is in a path defined by your WDIR environment variable.

```
44  ' Includes the contents of WDIR file.
45  ' fileNameStr - String
46  Sub Include(fileNameStr)
```

On Line 47 the script creates a *FileSystemObject* object.

```
47       Dim fileSysObj: Set fileSysObj =
                        CreateObject("Scripting.FileSystemObject")
```

Line 48 calls the *FindInWDIR()* local helper function to locate the required file and return its path. The '1' parameter at the end of Line 48 instructs the *OpenTextFile* method to open this file for reading.

```
48        Dim fileObj: Set fileObj =
     fileSysObj.OpenTextFile(FindInWDIR(fileNameStr, fileSysObj),1)
```

Line 49 reads the entire contents of this file into a string variable named *scriptStr*.

```
49        Dim scriptStr: scriptStr = fileObj.ReadAll
```

Line 50 calls the built-in VBScript *ExecuteGlobal()* routine to run the code from our library file.

```
50         Call ExecuteGlobal(scriptStr)
51   End Sub
```

FindInWDIR() Function

The *FindInWDIR()* helper function accepts two parameters. The first is typically a relative path to a library file in a directory specified by the WDIR environment variable; however, this parameter can be an absolute path to a library file. The second parameter is the *FileSystemObject* object instantiated in the *Include()* helper function.

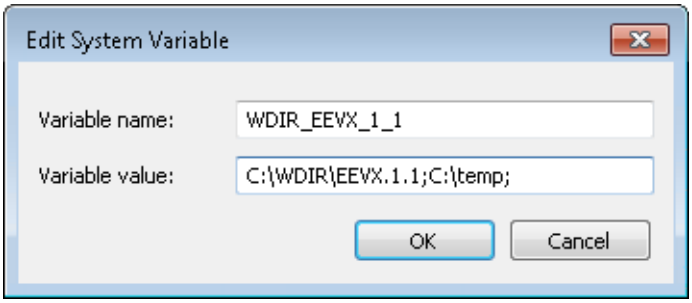
```
53   ' Finds specified file in WDIR directory. Returns full path.
54   ' filenameStr - String
55   ' fileSysObj - FileSystemObject Object
56   Function FindInWDIR(filenameStr, fileSysObj)
```

On Line 57 the script specifies the separator character (a semicolon) in environment variables on the Windows operating system.

```
57       Dim separatorStr: separatorStr = ";"
```

Consider a simple WDIR value comprising two values (paths) separated by a semicolon as illustrated in Figure 236: Simple WDIR Example .

Figure 236. Simple WDIR Example



The separator character for UNIX or Linux operating systems is a colon. Therefore, on Lines 58 through 60 the *Scripting.IsUnix* method determines if the script is running on UNIX or Linux and if so changes the separator character to be a colon.

```
58      If Scripting.IsUnix Then
59          separatorStr = ":"
60      End If
```

Lines 61 and 62 use the separator character to split the WDIR environment variable and create an array of directory paths.

```
61      Dim wdirArr:
62      wdirArr = Split(Scripting.GetEnvVariable("WDIR_<release>"),
separatorStr)
```

Where *<release>* is the root name of the product release or other unique identifier.

Lines 64 through 70 iterate through all of the directory paths checking if the required library file exists. When the script finds the file it returns the complete path name and exits the function.

```
63      Dim i
64      For i = 0 To UBound(wdirArr)
65          Dim pathStr: pathStr = wdirArr(i) & "/" & filenameStr
66          If fileSysObj.FileExists(pathStr) Then
67              FindInWDIR = pathStr
68              Exit Function
69          End If
70      Next
```

Finally, if the script does not find the specified library file in any of the WDIR directory paths, the *filenameStr* parameter to this function is assumed to be an absolute path to the library file. Therefore, the name is returned on Line 71.

```
71      FindInWDIR = filenameStr
72  End Function
```

Note:
It is difficult to debug code that is in a library. This is because the error messages do not output the line number of the error, if the error occurs in the script library. For this reason, it is strongly recommended that the first time you create new helper routines, you should embed them in the main script for testing and debugging. After you determine that your routines are correct, you should move them to a library file to include by other scripts.

Chapter 19
VBScript References and Tools

This section presents a list of scripting references and scripting tools.

General References
Text and Scripting Editors
Object Browsers

General References

For general information regarding scripting, refer to these sources.

- For general scripting information:

  http://www.msdn.microsoft.com/library/default.asp?url=/library/en-us/dnanchor/html/scriptinga.asp

- For a VBScript language reference:

  http://www.msdn.microsoft.com/library/default.asp?url=/library/en-us/script56/html/vbscripttoc.asp site

- The book *VBScript Programmer's Reference* by Susanne Clark, Antonio De Donatis, and more authors (ISBN: 0-7645-4367-9) provides a complete VBScript reference.

- A method for importing VBScript libraries can be found at:

  http://www.source-code.biz/snippets/vbscript/5.htm

- For information on making languages like Perl and Tcl automation-aware:

  http://www.activestate.com

Text and Scripting Editors

There are a number of text editors with automatic syntax; some examples are as follows:

- VI users can load **gVim** to get syntax coloring (Free).

- The **Vbsedit** tool from *http://www.AderSoft.com* is a text editor that is similar to the Windows base editor (Free). This product includes a debug environment (Not Free).

- Another editor is **TextPad** from *http://www.textpad.com*. You can load a module with this tool to obtain syntax coloring for various languages (Not Free).

- **PrimalScript** from Sapien Technologies at *http://www.sapien.com* is another editor that can be integrated with debuggers to give a complete development environment (Not Free).

Object Browsers

Object Browsers can be found in Microsoft Office tools (Word, Excel, etc.) and also in the Visual Basic 6.0 application. Note that Object Browsers can be used to view any type library and they are not specific to the MGCPCB type library.

The following steps describe how to view the MGCPCB type library using one of the Microsoft Office applications.

1. Choose **Tools > Macro > Visual Basic Editor**.
2. In the Visual Basic Editor choose **Tools > References**.
3. In the Available References list, check the box associated with the MGCPCB (AutoActive Series) Type Library item.
4. Click **OK** in the **Available References** dialog.
5. In the Visual Basic Editor choose **View > Object Browser**.
6. In the **Object Browser** dialog, filter the type libraries to only show the **MGCPCB** type library. To do this, select the **MGCPCB** item in the drop down combo box that reads **<All Libraries>**
    - The pane labeled **Classes** shows all of the objects in the MGCPCB interface.
    - Selecting an **Object** will display that object's methods and properties in the right-hand pane.
    - Selecting a **Method/Property/Event** in the right pane will display any corresponding parameters in the bottom pane along with a brief description as to what this **Method/Property/Event** does and any tasks it performs.

Third-Party Information

Details on open source and third-party software that may be included with this product are available in the *<your_software_installation_location>/legal* directory.