



PCB Automation Tutorial

For Software Version 2005 Spac 1 and greater

© 2007 Mentor Graphics Corporation
All rights reserved.

This document contains information that is proprietary to Mentor Graphics Corporation. The original recipient of this document may duplicate this document in whole or in part for internal business purposes only, provided that this entire notice appears in all copies. In duplicating any part of this document, the recipient agrees to make every reasonable effort to prevent the unauthorized use and distribution of the proprietary information.

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

The terms and conditions governing the sale and licensing of Mentor Graphics products are set forth in written agreements between Mentor Graphics and its customers. No representation or other affirmation of fact contained in this publication shall be deemed to be a warranty or give rise to any liability of Mentor Graphics whatsoever.

MENTOR GRAPHICS MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

MENTOR GRAPHICS SHALL NOT BE LIABLE FOR ANY INCIDENTAL, INDIRECT, SPECIAL, OR CONSEQUENTIAL DAMAGES WHATSOEVER (INCLUDING BUT NOT LIMITED TO LOST PROFITS) ARISING OUT OF OR RELATED TO THIS PUBLICATION OR THE INFORMATION CONTAINED IN IT, EVEN IF MENTOR GRAPHICS CORPORATION HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

RESTRICTED RIGHTS LEGEND 03/97

U.S. Government Restricted Rights. The SOFTWARE and documentation have been developed entirely at private expense and are commercial computer software provided with restricted rights. Use, duplication or disclosure by the U.S. Government or a U.S. Government subcontractor is subject to the restrictions set forth in the license agreement provided with the software pursuant to DFARS 227.7202-3(a) or as set forth in subparagraph (c)(1) and (2) of the Commercial Computer Software - Restricted Rights clause at FAR 52.227-19, as applicable.

Contractor/manufacturer is:

Mentor Graphics Corporation

8005 S.W. Boeckman Road, Wilsonville, Oregon 97070-7777.

Telephone: 503.685.7000

Toll-Free Telephone: 800.592.2210

Website: www.mentor.com

SupportNet: supportnet.mentor.com/

Send Feedback on Documentation: supportnet.mentor.com/user/feedback_form.cfm

TRADEMARKS: The trademarks, logos and service marks ("Marks") used herein are the property of Mentor Graphics Corporation or other third parties. No one is permitted to use these Marks without the prior written consent of Mentor Graphics or the respective third-party owner. The use herein of a third-party Mark is not an attempt to indicate Mentor Graphics as a source of a product, but is intended to indicate a product from, or associated with, a particular third party. A current list of Mentor Graphics' trademarks may be viewed at: www.mentor.com/terms_conditions/trademarks.cfm.

End-User License Agreement: You can print a copy of the End-User License Agreement from: www.mentor.com/terms_conditions/enduser.cfm

Contents

SECTION 1: INTRODUCTION AND PRIMERS	11
Chapter 1: Introduction	12
Introduction	13
Documentation Conventions	13
Icons.....	13
Fonts	14
Terminology.....	14
Client/Server	14
Enumerate.....	15
Object.....	16
Property.....	16
Method	16
Collection	17
Event	17
Type Library	18
Prog ID	18
Script	19
Script Host.....	19
Scripting Language	19
Accelerator Key Binding.....	19
Menu Button/Item.....	19
Menu Popup.....	20
In-Process Client.....	20
Out-of-Process Client.....	20
Chapter 2: VBS Primer.....	22
Introduction	23
General-Purpose Statements	24
Option Explicit	24
Dim [Dimension a variable]	24
Dim (x,y,z,...) [Dimension an array of variables]	25
ReDim (x,y,z,...) [Re-dimension an array of variables]	26
UBound(...) [Return the upper-bound(s) of an array]	27
Set [Setting/assigning an object to a variable].....	28
Call [Calling a method or function]	28
Arithmetic, Comparison, and Logical Operators.....	29
Arithmetic Operators	29
Comparison Operators.....	30
Logical Operators.....	31
String Operators.....	31
Control Statements	32
If ... Then ... End If	32
If ... Then ... Else ... End If	32
For ... To ... Next	32
For Each ... In ... Next.....	33
While ... Wend	34
Select Case ... End Select	34
Functions and Subroutines	35
Function <name of function>(parameters to function)	35
Sub <name of subroutine>(parameters to subroutine)	36
Input/Output (I/O) Functions	37
InputBox().....	37
MsgBox().....	37
Conversion and Formatting Functions	38
Conversion Functions [CInt(), CDbl(), CStr(), ...]	39
String Manipulation Functions [Len(), Left(), Right(), Replace(), ...]	39
Special Characters.....	40
FormatNumber().....	41
Date(), Time(), Now(), and FormatDateTime() Functions	42
Error Handling.....	43

Chapter 3: Running Your First Script	47
Introduction	48
Creating and Running the Hello World Script.....	48
Creating the Hello World Script.....	48
Running the Hello World Script.....	48
Creating and Running the Via Count Script	49
Visualizing the Data Hierarchy.....	50
Using the Help System	50
Creating the Via Count Script	52
Running the Via Count Script.....	55
An Alternative Technique.....	56
Chapter 4: Introducing the IDE	58
Introduction	59
Using the IDE to Create a Script	59
The IDE's Error Handling Capability.....	62
Using the IDE to Create a Form/Dialog	63
Launching the IDE in its Form Editor Mode.....	63
Adding Graphical Buttons to the Form/Dialog	65
Adding Code "Behind" the Cancel Button.....	67
Adding Code "Behind" the Count All Button	69
Adding Code "Behind" the Count Selected Button.....	76
Removing Duplicated Code	77
Assigning a Title.....	78
Going into Production	79
Chapter 5: Associating an Accelerator (Shortcut) Key with a Script	81
Introduction	82
Associating an Accelerator (Shortcut) Key with a Script	82
Creating the Script	82
Running the Script	83
Chapter 6: Adding a Menu and/or Menu Button/Item.....	86
Introduction	87
Tying an Existing Script to a New Button/Item on an Existing Pull-Down Menu	87
Creating the Script	87
Running the Script	91
Modifying the Previous Example to Use a Single Script	93
Creating the Script	94
Running the Script	97
Creating a New Top-Level Menu Item.....	99
Creating the Script	99
Running the Script	101
Chapter 7: Adding a Tool Bar Icon.....	103
Introduction	104
Adding a New Icon	104
Creating the Script	105
Running the Script	107
Chapter 8: Running Scripts from the Command Line.....	109
Introduction	110
Running Scripts from the Command Line (no Arguments)	111
Running Scripts from the Command Line (with Arguments)	114
Example #1: Simply Display a Series of Arguments	114
Example #2: Actually Using the Arguments.....	117
More-Sophisticated Error Handling.....	120
Attaching and/or Creating Applications and Opening Documents	122
Attaching (Application Open, Document Open).....	123
Attaching (Application Open, Document Closed)	125
Creating Invisible (Application Closed, Document N/A)	129
Creating Visible (Application Closed, Document N/A).....	131
Chapter 9: Running a Script from Within an Application	134

Introduction	135
Using the WDIR Environment Variable	136
Creating/Modifying the WDIR Environment Variable.....	136
Chapter 10: Running Startup Scripts.....	141
Introduction	142
Introducing scripts.ini Files	142
Startup Scripts when an Application is Launched	143
Startup Scripts when a Document is Opened	145
Debugging Startup Scripts.....	146
Chapter 11: Introducing Events and the Scripting Object.....	147
Introduction	148
Events and In-Process Scripts	148
Events and Out-Of-Process Scripts Using mgcscript.exe.....	153
Chapter 12: Basic Building-Block Examples	155
Introduction	156
Creating a Skeleton Script.....	156
Traversing Objects.....	156
Traversing Nets.....	157
Traversing Components.....	159
Traversing Pins	163
Collection Filters	171
Finding Unplaced Components	171
Deleting Traces	173
Filtering Padstack Objects	175
Display Control	177
Toggling the Display of Placement Outlines On/Off	177
Changing the Layer Color.....	178
Load Scheme.....	180
Executing Menu Commands	181
Executing the Fit Board Command.....	181
Executing the Route Command	181
Creating Objects with Geometries.....	185
Put Trace.....	186
Put User Layer Graphics.....	194
Interactive Commands.....	198
Put Vias.....	198
File Input/Output (I/O).....	204
Creating a Net Report	204
Reading a Points Array from a File	207
Running Executables.....	215
Using NotePad to View a Net Report	215
Compressing a File and Emailing the Result.....	218
Reading, Analyzing, and Updating a Design.....	222
Generate Via Caps	222
SECTION 2: DETAILED EXAMPLES – EXPEDITION PCB	228
Chapter 13: Display a Single Routing Layer	229
Introduction	230
Before We Start	230
The Script Itself.....	231
Constant Definitions.....	232
Initialization/Setup	232
Main Subroutine.....	233
Event Handler Generator Function	236
Display Control Subroutine	237
Helper Functions and Subroutines	238
Creating and Testing the Script.....	239
Enhancing the Script	241

Chapter 14: Documenting the Layer Stackup	242
Introduction	243
Before We Start	243
The Script Itself.....	245
Constant Definitions.....	247
Initialization/Setup	248
Event Handlers.....	249
Generate Motion Graphics Function.....	250
Generate User-Layer Graphics Function.....	251
Utility Functions and Subroutines	252
Helper Functions and Subroutines	257
Miscellaneous Functions and Subroutines	261
Creating and Testing the Script.....	262
Enhancing the Script	262
Chapter 15: Performing Assembly DRC	263
Introduction	264
Before We Start	264
The Assign Placement Order Script	266
Constant Definitions.....	266
Initialization/Setup	267
Assign Placement Order Function	267
The Main Script.....	268
Constant Definitions.....	268
Initialization/Setup	269
Event Handler	270
Assembly Violation Checking Function.....	271
Utility Functions and Subroutines	273
Helper Functions and Subroutines	277
Creating and Testing the Script.....	280
Enhancing the Script	280
Chapter 16: Removing Unconnected Pads from Pins and Vias.....	281
Introduction	282
Before We Start	282
The Form Itself.....	285
Adding Elements to the Form	286
Naming Elements on the Form	287
Preparing to Add Code "Behind" the Elements on the Form.....	288
Initialization and Setup.....	288
Exclude List Initialization.....	290
Include List Initialization	292
Event Change Handler for Pins Checkbox	293
Event Change Handler for Vias Checkbox	293
Include All Button	294
Exclude All Button	295
Include Selected Button	296
Exclude Selected Button	297
Remove Unconnected Pads Button	297
Helper Functions	300
Creating and Testing the Script	304
Enhancing the Script	305
Chapter 17: Integration with Excel	306
Introduction	307
Before We Start	307
The Script Itself.....	309
Constant Definitions.....	309
Initialization/Setup	310
Event Handler	311
Load Excel Subroutine	311
Utility Functions and Subroutines	312
Helper Functions and Subroutines	317
Creating and Testing the Script.....	320

Enhancing the Script	320
Chapter 18: Generating a Component CSV File.....	322
Introduction	323
Before We Start	323
The Script Itself.....	324
Constant Definitions.....	324
Initialization/Setup	324
Create CSV File Subroutine.....	325
Utility Functions and Subroutines	326
Helper Functions and Subroutines	330
Creating and Testing the Script.....	330
Enhancing the Script	331
Chapter 19: Finalizing a Design.....	332
Introduction	333
Before We Start	333
The Script Itself.....	335
Constant Definitions.....	335
Initialization/Setup	336
Add Board Dimensions Subroutine.....	337
Add Company Logo Subroutine.....	338
Utility Functions and Subroutines	339
Helper Functions and Structures	346
Creating and Testing the Script.....	348
Enhancing the Script	348
Chapter 20: Route by Layer	349
Introduction	350
Before We Start	350
The Script Itself.....	351
Constant Definitions.....	352
Initialization/Setup	352
Run Fanout Route Pass Subroutine	353
Run Angle-Bias-Based Route Pass Subroutine	354
Utility Functions and Subroutines	356
Helper Functions and Subroutines	359
Creating and Testing the Script.....	362
Enhancing the Script	363
Chapter 21: Show Maximum Length/Delay	364
Introduction	365
Before We Start	365
The Script Itself.....	368
Constant Definitions.....	368
Initialization/Setup	368
Event Handlers.....	370
Utility Functions and Subroutines	371
Helper Functions and Subroutines	374
Creating and Testing the Script.....	377
Enhancing the Script	377
SECTION 3: DETAILED EXAMPLES – FABLINK XE	378
Chapter 22: Introducing FabLink XE.....	379
Introducing FabLink XE	380
FabLink XE From an Automation Perspective	380
Chapter 23: Generating a BOM	382
Introduction	383
Before We Start	383
The Script Itself.....	386
Constant Definitions.....	388

Initialization/Setup	388
Build Bom Data Tree Function.....	390
Generate BOM Output Subroutine	394
Utility Functions and Subroutines	397
Helper Functions and Subroutines	400
Creating and Testing the Script.....	400
Enhancing the Script	401
Chapter 24: Generating a Panel Side View.....	402
Introduction	403
Before We Start	403
The Script Itself.....	405
Constant Definitions.....	406
Initialization/Setup	406
Generate Panel Side View Subroutine	407
Utility Functions and Subroutines	409
Helper Functions and Subroutines	414
Creating and Testing the Script.....	415
Enhancing the Script	416
Chapter 25: Creating a Via Mask	417
Introduction	418
Before We Start	418
The Script Itself.....	421
Constant Definitions.....	422
Initialization/Setup	422
Create Via Mask Function.....	423
Utility Functions and Subroutines	426
Helper Functions and Subroutines	429
Creating and Testing the Script.....	431
Enhancing the Script	431
Chapter 26: Generating Manufacturing Output Files	433
Introduction	434
Before We Start	434
The Script Itself.....	437
Initialization/Setup	438
Main Run Gerber Engine Subroutine.....	439
Main Run NC Drill Engine Subroutine	440
Utility Functions and Subroutines	442
Helper Functions and Subroutines	447
Creating and Testing the Script.....	448
Enhancing the Script	449
Chapter 27: Automated Board Placement on a Panel.....	450
Introduction	451
Before We Start	451
The Script Itself.....	454
Constant Definitions.....	455
Initialization/Setup	455
Main Get Parameters Function	456
Main Place Boards Subroutine	457
Utility Functions and Subroutines	461
Helper Functions and Subroutines	463
The Dialog/Form Itself	467
Adding Elements to the Form	468
Naming Elements on the Form	468
Naming the Form Itself.....	469
Preparing to Add Code "Behind" the Elements on the Form.....	470
Initialization and Setup.....	470
The OK Button Event/Click	471
The Cancel Button Event/Click	472
The Close (Terminate) Button Event Click	473
Creating and Testing the Script.....	474

Enhancing the Script	474
Appendix A: Good Scripting Practices.....	477
Introduction	478
Naming Conventions	478
Variable Names.....	478
Function and Subroutine Names	479
Method and Property Names	479
Event-Handler Function Names.....	479
Object Names in IDE Forms	480
File Names for Scripts and Forms/Dialogs	480
Option Explicit.....	480
Comments	481
Parentheses.....	481
Space Characters.....	482
Dim versus Dim().....	482
Appendix B: Recommended Script Architecting Process.....	483
Introduction	484
Creating a Template	484
Re-using Functions and Subroutines	484
Internal versus External Scripts.....	485
Calling other Scripts	485
Script to Script Communication	485
Delivering Scripts.....	485
Appendix C: Creating and Using a Library.....	486
Introduction	487
Creating a Library	487
Calling Routines from a Library	488
Appendix D: General VBScript References and Tools.....	493
General References	494
Text/Scripting Editors	494
Object Browsers	494
Appendix E: Accessing Example Scripts and Designs	496
Introduction	497
Index	498

SECTION 1: INTRODUCTION AND PRIMERS

Chapter 1: Introduction

Introduction

Expedition PCB 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 personnel) to augment, customize, and drive the capabilities of Expedition PCB 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 user having the ability to automate a series of actions that are available in the application.)

Expedition PCB supports *Component Object Model (COM)* automation. Developed by Microsoft®, COM may be considered to be 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 application does not necessarily infer the use of 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 user to pre-select any data before running an automation script. (When we say "access to the application's data," this is not to 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* as described later in this chapter.)
- **Function-Centric Automation:** This form of automation relies on a set of built-in functions that are included with the application. These functions operate on whatever data has been pre-selected. It is possible for the user to create new functions, but these can only execute a series of existing functions, so this form of automation is limited to whatever has been built-in the application by its architects. (An example of this form of automation is presented by Mentor's Falcon Framework and applications such as Board Station; in this case, the Falcon Framework provides the function-centric automation, while AMPLE is the scripting language that is used to drive the automation.)
- **Flow-Centric Automation:** In addition to using automation to augment and customize the capabilities of individual applications, it can also be used to control data as is passed through multiple applications throughout a design and verification flow (different types of data and automation may be available at different points in the flow).

The automation featured in Expedition PCB is predominantly data-centric, but it also offers a significant amount of function-centric and flow-centric capabilities.

Documentation Conventions

Icons



- The information icon is used to annotate important information.



- The exclamation icon is used to annotate cautionary information.

Fonts

- ❑ ***Italics*** font may be used to emphasize text. It is also used for book, document, chapter, and topic titles/names and for path and file names.
- ❑ **Bold** may be used to emphasize text, highlight menu items, and denote the titles of (and fields in) dialog boxes.
- ❑ **Menu > Command** identifies the path used to select a menu command by hand.
- ❑ **Menu -> Command** identifies the path used to select a menu command in a script.
- ❑ **Courier** font is used for program and script listings and for any text messages that the software displays on the screen.
- ❑ **Note:** describes important information, warnings, or unique commands.
- ❑ "Click-left" (or just "click") means click the left mouse button on the indicated item.
- ❑ "Click-middle" or "middle-click" means click the middle mouse button on the indicated item.
- ❑ "Click-right" or "right-click" means click the right mouse button on the indicated item.
- ❑ "Double-click" means click twice consecutively with the left mouse button.
- ❑ "Drag-left-and-drop" (or just "drag-and-drop") means press and hold the left mouse button on the indicated item, then move the cursor (pointer) to the destination and release the button.
- ❑ "Shift-click-left" means press and hold the <Shift> key then click the left mouse button on the indicated item.
- ❑ "Ctrl-click-left" means press and hold the <Ctrl> key then click the left mouse button on the indicated item.
- ❑ "Select" is another way of saying "click left."

Terminology

Automation involves a large number of concepts and terms. For the purposes of this introductory section, we shall briefly define some fundamental terminology as follows:

Client/Server

At its simplest level, a *server* is a software module or application that satisfies requests made by a client. By comparison, a *client* is a software module or application that makes requests to a server (or servers) to perform certain actions. As an example, consider the following illustration:

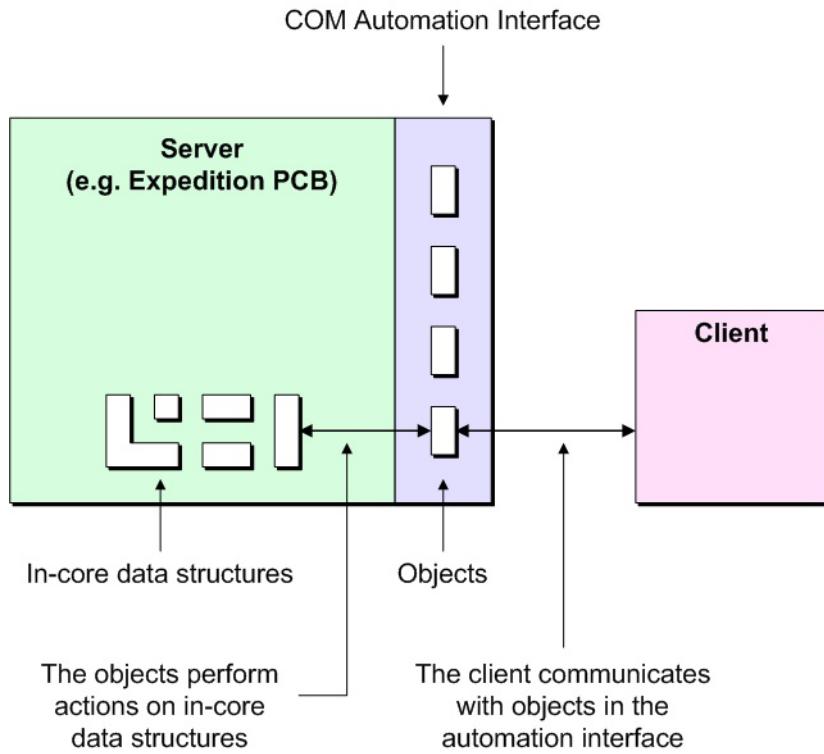


Figure 1-1. Client-server-based automation.

In this case, Expedition PCB is acting in the role of the server. The external (out-of-process) client communicates with objects in Expedition PCB's automation interface. These objects perform actions on Expedition PCB's in-core data structures. All of this is largely transparent to the user. From the user's perspective, it *appears* as though actions are being directly performed on the in-core data structures; indeed, things *should* be visualized as working this way.



Note: There may be multiple clients communicating with a single server, or there may be a single client communicating with multiple servers, or there may be multiple clients and multiple servers (see also the *In-Process Client* and *Out-of-Process Client* topics later in this chapter).

Enumerate

An *enumerate* may be regarded as a "descriptive number." For example, assume that we've declared a variable called *traceObj* and that we've already assigned a *Trace* object to this variable. One of the properties of a *Trace* object is its *anchor type*. Inside the application, the anchor type is represented as an integer:

0	=	None
1	=	Semi-fixed
2	=	Fixed
3	=	Locked

If we wanted to set the anchor type associated with the object pointed to by our *trace* variable to *Fixed*, for example, then we could use the following statement:

```
traceObj.Anchor = 2
```

However, this is less than intuitive and it makes this statement difficult for other programmers to parse and understand. For this reason, the Expedition PCB architects have defined an enumerate called `epcbAnchorType`. Also, they have defined four possible assignments for this enumerate and associated each of these assignments with an integer as follows:

<code>epcbAnchorNone</code>	<code>= 0</code>
<code>epcbAnchorSemiFixed</code>	<code>= 1</code>
<code>epcbAnchorFixed</code>	<code>= 2</code>
<code>epcbAnchorLocked</code>	<code>= 3</code>

Now, if we once again want to set the anchor type associated with the object pointed to by our trace variable to *Fixed*, we can use the following statement:

```
traceObj.anchor = epcbAnchorFixed
```

Object

An *object* is an entity (in memory) encapsulating methods and/or properties (see below). These entities perform actions on data and/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. By comparison, a *Button object* would be a GUI object that operates on a button in a dialog.

Property

A *property* is an attribute associated with an object. For example, a *Trace object* will have a *width* associated with it, and this width is a property of the *Trace object*.

Properties are set and modified by means of the "`=`" assignment operator. For example, assume that we've declared a variable called `traceObj` and that we've already assigned a *Trace object* to this variable. In this case, we could set the width to a specific constant value of say 10 using the following statement:

```
traceObj.width = 10
```

Alternatively, if we assume that we've already declared an integer variable called `widthInt` and assigned this variable a value of 10, we could use the following statement to achieve the same effect:

```
traceObj.width = widthInt
```

Method

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

Assume that we've declared a variable called `cmpObj` and that we've already assigned a *Component object* to this variable. Further assume that this component has already been placed. In this case, we could move the component by say 10 units in the X direction and 15 units in the Y direction relative to its original location using the following statement (we won't concern ourselves with the definition of "units" at this time):

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



Note: The reason for (and use of) the *Call* statement in the above example is discussed in *Chapter 2: VBS Primer*.

Alternatively, if we assume that we've already declared two integer variables called *xInt* and *yInt* and assigned these variables values of 10 and 15, respectively, we could use the following statement to achieve the same effect:

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

Contrast the way we apply a method with the way we assigned a value to a property in the previous topic. Although it may appear easy to spot the difference between properties and methods, the line can become somewhat blurred in certain cases. For example, assume that we've already declared a variable called *documentObj* and that we've assigned a *Document* (layout design) object to this variable. Further assume that we've declared a variable called *netObj*. Now consider the following statement, whose intent is to locate a net named *Fred*:

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

This looks like we're manipulating a property because this statement returns an object, but we're actually performing an action on the document (we're asking the document to find an object). Thus, in this case, we would regard this statement as applying a method.



Note: The reason for (and use of) the *Set* statement in the above example is discussed in *Chapter 2: VBS Primer*.

Collection

A *collection* may be regarded as a container for multiple objects. Furthermore, a collection can be used and treated as just another object. For example, assume that we've already declared a variable called *documentObj* and that we've assigned a *Document* (layout design) object to this variable. Further assume that we've declared a collection called *placedCompsColl*. Now consider the following statement, whose intent is to locate all of the placed components in the document and to put them in our collection:

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

In this case, *epcbSelectPlaced* is one of the pre-defined options associated with an enumerate as discussed earlier in this chapter.



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 as discussed earlier in this chapter, it is the client that initiates communication with the server. In many cases it is useful for the server to provide notification of some occurrence (something that happened). In such a case, *events* provide a mechanism for the server to notify one or more clients as to selected occurrences.

To think of this another way, let's remember that – in the context of these discussions – we are regarding the client as being the script and the server as being the application (e.g. Expedition PCB). In this case, events provide a mechanism by which clients (scripts) can react to changes in the server (application) and to modify the behavior of the server (application). 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 the

script includes an *OnClose* function, and that the user wants this function to be run after the document has been closed. In this case, the script has to notify the application that it has this function. When the application eventually closes the document, it will call the *OnClose* function in the script.



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

The previous example reflects an event that causes the client to react to a change in the server; that is, the client knows not to attempt to perform any more operations on this document because it's been closed (possibly under the direction of another client or perhaps as the result of an action by the user).

An example of the client using an event to modify the behavior of the server could be a *PreClose* function. Assume that the client has informed the server that it has a *PreClose* function. Now assume that the user instructs the server to close the document. At this time, the server will issue an event to inform the client that it [the server] intends to close the document, but that this action has not happened yet. The client's *PreClose* event handler now has the opportunity to return a value of *True* (thereby allowing the server to close the document) or a value of *False* (thereby preventing the close from occurring).

Type Library

A *type library* defines the *enumerates*, *objects*, *methods*, *properties*, and *events* associated with – and supported by – a particular server. (See *Creating the Via Count Script* topic in *Chapter 3* for an example of adding a type library.)

Prog ID

A *Prog ID* (program identifier) is a predefined string that is used to uniquely identify a server. Such an identifier can be used to launch a server or to attach a client to an already running server. Furthermore, a *Prog ID* is also required in order to access and extract the enumerates from the type library associated with a server (see *Chapter 3: Running your First Script* for an example of this usage).

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

```
MGCPCB.ExpeditionPCBAplication  
MGCPCB.BoardStationREApplication  
MGCPCB.FablinkXEApplication  
MGCPCB.PlannerPCBAplication  
MGCPCB.ViewerPCBAplication
```

The following list details some additional *Prog IDs* that may be of interest:

```
MGCSSDD.CommandBarsEx  
MGCSSDD.KeyBindings  
  
MGCPCBEngines.MaskEngine  
MGCPCBEngines.NCDrill  
MGCPCBEngines.PDFOutput  
MGCPCBEngines.DFFDRC  
MGCPCBEngines.DRC  
MGCPCBEngines.Gerber  
MGCPCBEngines.ManufacturingOutputValidation  
  
MGCPCBLibraries.MaterialsDB  
MGCPCBLibraries.PadstackEditorDlg  
MGCPCBLibraries.PartsEditorDlg
```

Script

In the most general of terms, a *script* is a set of ASCII (text) commands that are processed and executed by a *script host*.

Script Host

In the most general of terms, a *script host* (sometimes called a *script engine*) is an application that can run a *script*.

Scripting Language

A *scripting language* is a special programming language used to create scripts. As discussed in *Chapter 2: VBS Primer*, there are multiple scripting languages available. For the purposes of this tutorial we shall focus on *Visual Basic Script* (also known as *VBScript* or *VBS*).

Accelerator Key Binding

In generic usage, the term *accelerator* refers to a shortcut key on the keyboard that is used to execute 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 is used to launch a script.

Menu Button/Item

The term *menu button* or *menu item* refers to a single entry in a popup window. For example, consider the following screenshot:

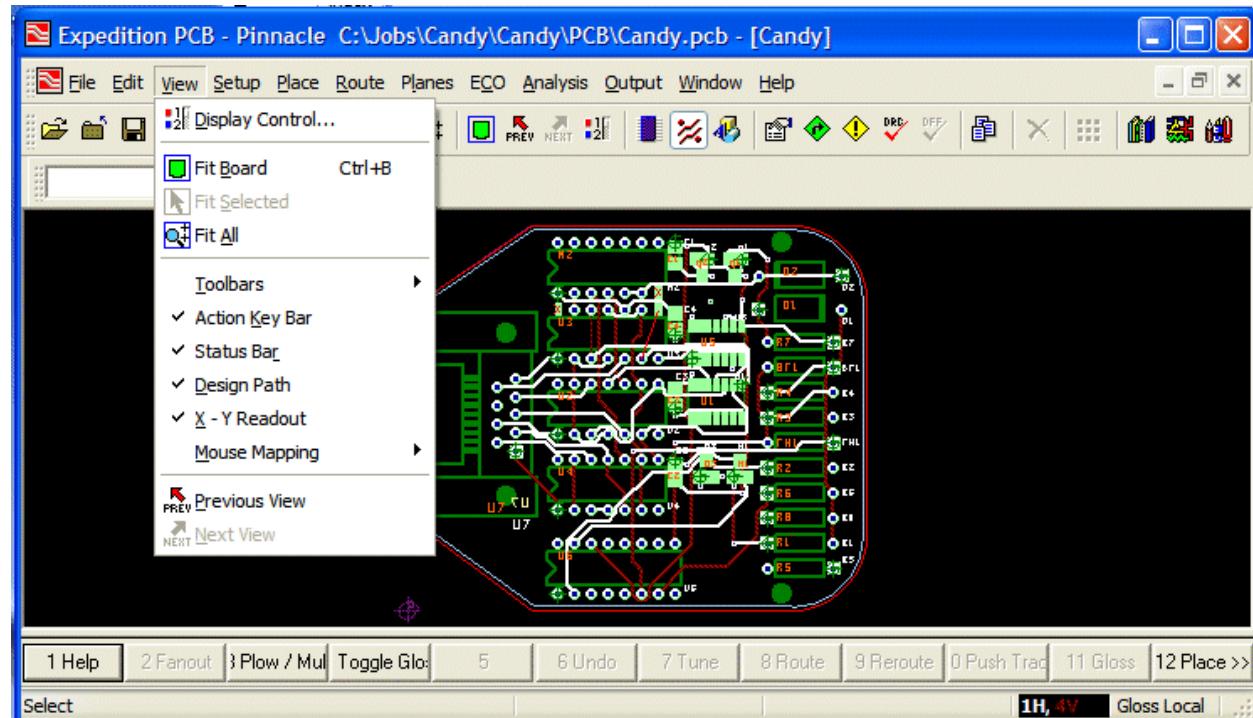


Figure 1-2. Screenshot of one of Expedition PCB's popup menus.

In this case, items like **Display Control**, **Fit Board**, **Fit Selected**, **Fit All**, and so forth would be considered to be menu buttons.

Menu Popup

The term *menu popup* refers to a list of items that appear in a popup window. Such a list may be obtained by selecting a main menu item (clicking-left) or by accessing a context-sensitive menu (clicking-right).

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, Expedition PCB is a script host in its own right as is illustrated in Figure 1-3.

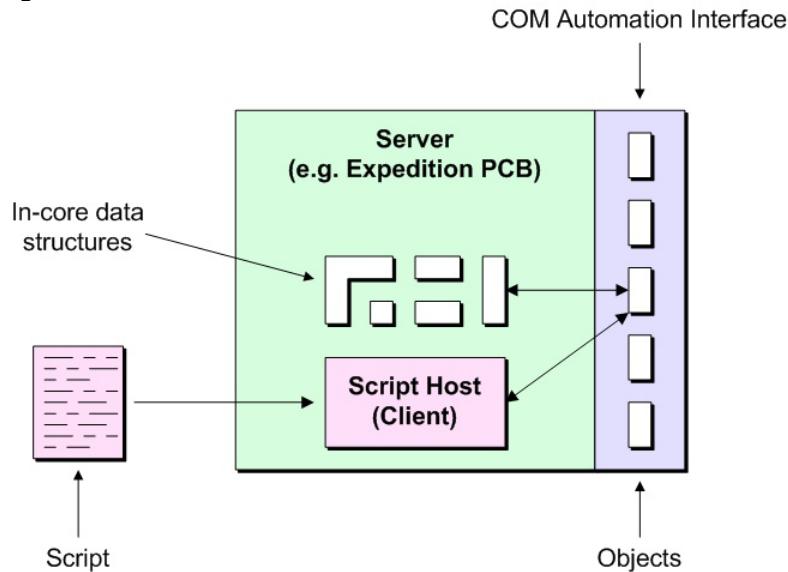


Figure 1-3. An in-process client.

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 FabLink XE is acting as an out-of-process client that is accessing and manipulating data in Expedition PCB (Figure 1-4).

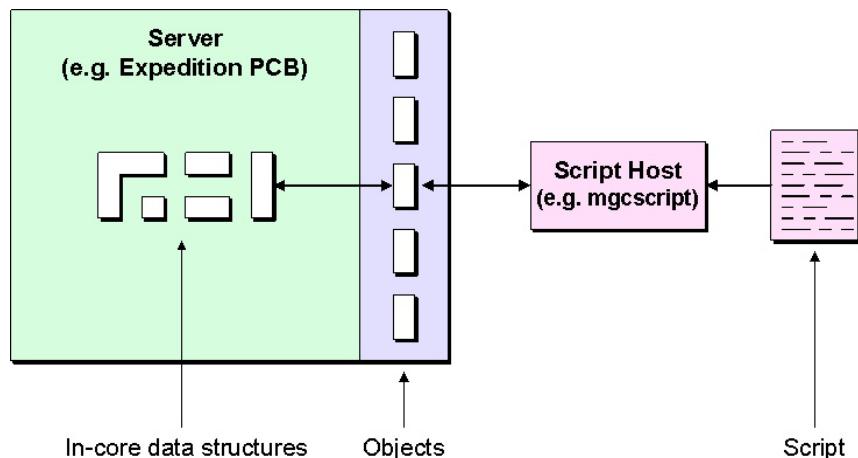


Figure 1-4. An out-of-process client.

Due to the fact that applications such as Expedition PCB and FabLink XE both support COM automation, it is possible to create a single script than can be run by both script hosts. That is, the same script that was run by the in-process client in Figure 1-3 could be run on the out-of-process client in Figure 1-4.



Note: Performance degrades significantly when a script communicates across process boundaries. For example, consider the following scenario, in which a script host in Expedition PCB is running *Script-A* and is acting as an in-process client to Expedition PCB. At the same time, a script host in FabLink XE is running *Script-B* and is acting as both an in-process client to FabLink XE and as an out-of-process client to Expedition PCB (Figure 1-5).

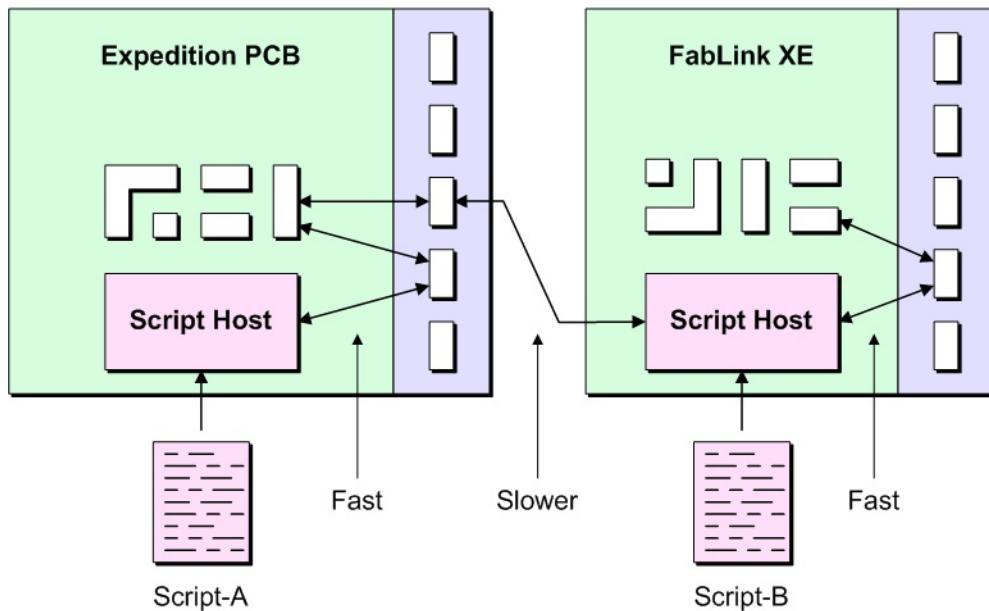


Figure 1-5. Performance degrades when running across process boundaries.

Since the script host running in Expedition PCB is only accessing and manipulating data inside Expedition PCB, there are no performance issues. Similarly, in the case of the script host running in FabLink XE, there are no performance issues with regard to any manipulation of data within FabLink XE. It is only when the script host running in FabLink XE acts as an out-of-process client and attempts to access and manipulate data in Expedition PCB that there is a negative impact on performance.

Chapter 2: VBS Primer

Introduction

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 to be used to build 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. Scripting languages are used 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 one moment, then a real, and then a string, because this makes it easier for them 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 are often used to perform a series of actions over and over again; 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's 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 the use of VBScript because it is easier to learn for folks who aren't coming from a programming or scripting background (languages like Perl and Tcl tend to be somewhat cryptic and very difficult to read and understand for the uninitiated).

The script host featured in Mentor's AutoActive tools – such as Expedition PCB and FabLink XE – is capable of working with all of the above scripting languages.



Note: Tcl, Perl, and Python can be used to drive automation only on Windows® platforms because COM technology is native to Windows® (this is ironic, since these languages were originally created for use in Unix/Linux environments).

When requested to run a script, the script host will determine 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 will check your computer's registry to determine whether or not you

have the appropriate parser/executer 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 a number of compiled languages that can be used for automation. These languages include Visual Basic, C++, the .Net languages (C#, J#, and VB.Net), and Java (with the use of a special Java/COM bridge). Once again, these languages can be used to drive automation only on Windows® platforms because COM technology is native to Windows®.

General-Purpose Statements

VBScript is an extremely rich and powerful language. The remainder of this chapter is intended to provide only a brief overview of some of the more common statements (see also *Appendix D: General VBScript References and Tools*).

Option Explicit

By default, VBScript does not require the use of *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 use a misspelled variable name, the script host will report an error and guide you directly to the problem (see also *Appendix A: Good Scripting Practices*).

Dim [Dimension a variable]

The *Dim* statement is used to instantiate (declare) 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
```



Note: VBScript is an un-typed language. If a *string* is assigned to a variable, for example, the variable becomes a string at that time. If an *integer* is later assigned to the same variable, the variable becomes an integer at that time. 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
```



Note: There are two types of entities that can be assigned to variables: *primitive types* (boolean, integer, real, string) and *object types* (object, collection). Also, there are some implicit object types, which include *Application* and *Scripting* (see also *Chapter 3: Running Your First Script* 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 our scripts:

Bool	Boolean (e.g. 0 / 1 or True / False)	' Primitive type
Int	Integer (e.g. 10)	' Primitive type
Real	Real number (e.g. 3.142)	' Primitive type
Str	String (e.g. "Hello World!")	' Primitive type
Obj	Object	' Object type
Coll	Collection (of objects)	' Object type

For example, a *Trace* object could be named something like *traceObj*, while a collection of *Trace* objects might be called *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. Due to the fact that 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 declares a one-dimensional array comprising four items (numbered from 0 to 3) and then assigns strings to each of these items. We can visualize this as illustrated in Figure 2-1.

0	1	2	3
0 Fred	1 Beth	2 Max	3 Janet

Figure 2-1. A one-dimensional array.



Note: The value in parenthesis is the *upper-bound* of the array. The items in an array are numbered from 0 (zero). Thus, 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 declares a two-dimensional array comprising four "columns" and three "rows", and then assigns strings to these items. We can visualize this as illustrated in Figure 2-2.

	0	1	2	3
0	(0,0) Fred	(1,0) Beth	(2,0) Max	(3,0) Janet
1	(0,1) Margaret	(1,1) Cuthbert	(2,1) ...	(3,1) ...
2	(0,2) ...	(1,2) ...	(2,2) ...	(3,2) ...

Figure 2-2. A two-dimensional array.

Similarly, `Dim(6,4,8)` would declare 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 in the future. For example, consider the following statement:

```
Dim nameStrArr( 4 )
```

This declares a one-dimensional array whose size has been set in stone as comprising only five items numbered from 0 to 4. Now consider the following statement:

```
Dim anotherNameStrArr( )
```

In this case, parentheses are used to indicate that we are instantiating an array, but no parameters were associated with these parentheses. The result is a *dynamic array* whose size can be modified in the future by means of the `ReDim` statement. For example, let's assume that – at some stage in our script – we want our `anotherNameStrArr` to become a two-dimensional array comprising a matrix of six by six items; this can be achieved using the following statement:

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

Once we've applied a `ReDim` statement, we 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 will be lost when a new `ReDim` statement is executed. 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've 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.



Note: Using the *Preserve* statement can negatively impact 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(s) of an array]

The *UBound* statement is used to determine 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* will end up containing 2, which is the upper bound of the *nameStrArr* array. Note that we could also have written the assignment to *sizeInt* as follows:

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

In this case, the second parameter associated with *UBound* specifies the dimension in which we are interested. As *nameStrArr* is a one-dimensional array in this example, our only option was 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)
```

In this case, the variables *sizeXInt*, *sizeYInt*, and *sizeZInt* will end up containing 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/assigning an object to a variable]

The *Set* statement must be used when a method or function returns an object type (*object* or *collection*) that is assigned to a variable. For example, assume that we've already declared and established a collection of *Trace* objects (which we may refer to as a "*Traces collection*") called *traceColl* and that we want to access the first item in this collection; now, consider the following snippet of code:

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



Note: Collections commence at item 1, while arrays are numbered from item 0.



Note: The *Set* statement must also be used when a method or function returns an implicit object type such as an *Application* (see also *Chapter 3: Running Your First Script* for more details on these implicit object types). For example, consider the following snippet of code:

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



Note: The *Set* statement **cannot** be used when assigning primitive types. For example, consider the following snippet of code:

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

Furthermore, we should 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 must be used when calling a *method* that has parameters in parentheses. For example, assume that we've declared a variable called *traceObj* and that we've already assigned a *Trace* object to this variable. Further assume that this trace has already been assigned a location. In this case, we could move the trace by say 10 units in the X direction and 15 units in the Y direction relative to its original location using the following statement (we won't concern ourselves with the definition of "units" at this time):

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



Note: The *Call* statement **cannot** be used when calling a *method* that has parameters when you choose not to enclose these parameters in parentheses; for example:

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

However, it is recommended that you do use the parentheses, and hence the *Call* statement (see also *Appendix A: Good Scripting Practices*).



Note: 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 don't want to assign the returned value to a variable. In such a case, it's optional whether or not any parameters associated with that function are enclosed in parentheses (once again, however, it is recommended that you *do use* the parentheses).



Note: With regard to the previous point, unlike a *function*, a *subroutine* never returns a value. Thus, in the case of calling a subroutine, the use of the *Call* statement is optional. Furthermore, irrespective as to whether or not you use the *Call* statement, it's optional whether or not any parameters associated with that subroutine are enclosed in parentheses (once again, however, it is recommended that you *do use* the parentheses).

Arithmetic, Comparison, and Logical Operators

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

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

In the case of expressions that include operators from more than one category, any arithmetic operators are executed first, then comparison operators, and finally logical operators.

Arithmetic Operators

VBScript supports a suite of arithmetic operators as shown in Table 2-1. If an expression contains multiple arithmetic operators, they will be evaluated in the following order of precedence; that is, exponentiation will be performed before unary negation, which will be performed before multiplication, and so forth.

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

Table 2-1. The arithmetic operators supported by VBScript.

```
' Arithmetic/Assignment Operator example

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

ansInt = xInt + yInt
```



Note: In the code snippet above, observe how colons can be used to separate multiple statements on the same line.

Comparison Operators

VBScript supports a suite of comparison operators as shown in Table 2-2. Comparison operators all have equal precedence, so if an expression contains multiple comparison operators they will be evaluated from left-to-right in the order in which they appear

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

Table 2-2. The comparison operators supported by VBScript.



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 Table 2-3. If an expression contains multiple logical operators, they will be evaluated in the following order of precedence; that is, logical negation will be performed before logical conjunction, which will be performed before logical disjunction, and so forth.

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

Table 2-3. The logical operators supported by VBScript.

```

' 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 Table 2-4.

Symbol	Description
&	String concatenation
+	String concatenation

Table 2-4. The string concatenation operators supported by VBScript.

```

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 (see also the discussions on the *I/O Functions* later in this chapter).

Control Statements

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

If ... Then ... End If

Assume that you've already declared a string variable called *nameStr* and you perform the following test:

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

If *nameStr* does equal "Fred", then all of the statements between the *Then* and the *End If* statements will be executed.



Note: The *If ... Then ... End If* statement can all be performed on a single line; for example, consider the following:

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

In this case, the *End If* portion of the statement is optional.



Note: Multiple *If ... Then ... End If* statements can be nested; for example, assume that you've already declared two string variables called *firstNameStr* and *lastNameStr*. Now consider the following:

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

If ... Then ... Else ... End If

Assume that you've already declared a string variable called *nameStr* and you perform the following test:

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

If *nameStr* does equal "Fred", then all of the statements between the *Then* and the *Else* statements will be executed; otherwise, all of the statements between the *Else* and the *End If* statements will be executed.

For ... To ... Next

The *For ... To ... Next* control statement causes a group of statements to be repeated a specified number of times; for example, consider the following:

```
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 snippet above – the *Dim* statement – observe how commas can be used to separate multiple variable declarations on the same line



Note: In the second line of the code snippet above, observe how colons can be used to separate multiple statements on the same line.



Note: The *Step* portion of the *For ... To* statement is optional. If this is omitted, the step will default to increments of +1. However, if the *startInt* value is less than the *endInt* value and you want to count down then you must use a *Step* qualifier; for example, *Step -1*.



Note: Wherever a variable (except *counterInt*) was used in the example code snippet above, this could be replaced with a constant integer. For example, the following would provide 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
```



Note: If you are in the middle of a *For ... To ... Next* loop and you use a conditional statement to determine that you've finished whatever it was you were trying to do, you can use the *Exit For* statement to terminate the loop and continue to the next portion of your script. For example, consider the following:

```
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
```

In this case, when *counterInt* is incremented to equal 5, any statements used to replace the "Before" comment will be executed, while any statements used to replace the "After" comment will not be executed. (With regard to the above example, remember that the *End If* portion of the *If ... Then ... End If* statement is optional if everything is on the same line.)

For Each ... In ... Next

This is a special case of the *For ... To ... Next* control statement that is used to process collections of objects. For example, assume that you have already declared (and assigned) a collection of *Trace* objects called *traceColl*. Now consider the following snippet of code:

```
Dim traceObj
```

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

The *For Each ... In ... Next* statement shown above will commence with the first item in the collection and iterate to the last item in the collection. (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 in the previous sub-topic.)

While ... Wend

This form of control statement repeats while a specified condition exists. As a simple example, consider the following snippet of code:

```
Dim tempInt
tempInt = 10

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

Any statements in the body of the loop will be executed over and over again until the final iteration when the *Wend* statement is reached and the controlling condition is no longer true.



Note: There are also *Do ... Until* statements that repeat until a condition exists and *Do ... While* statements that repeat while a condition exists, but these are used less often than the other control statements and so are not discussed here (see also *Appendix D: General VBScript References and Tools*).

Select Case ... End Select

This control statement is of particular use when there are a large number of conditions to be checked. 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 above 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 actually resolves to an integer). For example, assume that you've already declared and initialized a string variable called *nameStr*; now consider the following:

```
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
```



Note: The *Case Else* portion of this statement – which is optional – acts as a "catch-all". Any statements associated with the *Case Else* will be executed if none of the other case conditions are met.

Functions and Subroutines

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 <name of function>(parameters to function)

Assume that you want to create a function called *AddTwoInts* that accepts two integers as parameters, adds these integers together, and displays the result. You could achieve this as follows:

```
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 the use of VBScript classes is an advanced topic that is outside the scope of this tutorial.

Now, let's assume that you want to call your function from elsewhere in the script. You could do this as follows:

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

In this example we've passed two integer constants into the function, but these could of course be integer variables.

If you want your function to return a value, you can achieve this by assigning this value to the function name itself; for example:

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

Observe the last line before the *End Function* statement (in **bold**). Here, the value associated with the variable *tempInt* is assigned to the function name, so this value will be returned to the outside world. In this case, you might call your function as follows:

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

In some cases you may desire to terminate the function before completion. You can achieve this by means of 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; now consider the following:

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



Note: 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 will remain unchanged.



Note: Even if a function does return a value, you aren't obliged to use it. For example, assume that you're working with the last version of our *AddTwoInts* function and consider the following code snippet:

```
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*; by comparison, the second time *AddTwoInts* is called (b), its return value will simply be discarded by the script.



Note: In some cases you may create a function (or a subroutine) that does not accept any parameters. In this case, the use of parenthesis when declaring the function is optional; however, it is recommended that they are used for the sake of consistency. For example, consider the following function:

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

Similarly, it's good programming practice to use the parenthesis when calling this function from elsewhere in your script, because this helps to emphasize the fact that you are calling a function; for example:

```
SillyFunction()
```

Sub <name of subroutine>(parameters to subroutine)

Subroutines are very similar to functions; all that is required is to make the following changes:

Function is replaced with *Sub*

End Function is replaced with *End Sub*

Exit Function (if used) is replaced with *Exit Sub*

The main difference between subroutines and functions is that a subroutine cannot be used to return a value (it can, however, modify any *In-Out* parameters passed into the subroutine – and all parameters to functions and subroutines in VBScript are *In-Out* parameters).

Input/Output (I/O) Functions

There are two very common VBScript functions that are used to perform input and output operations: *InputBox()* [for input] and *MsgBox()* [for output].

InputBox()

The simplest way to use the *InputBox()* function is to call it with its first parameter set to be your query string; that is, the question you want to ask. For example, consider the following snippet of code:

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

When the script reaches the *InputBox()* statement, it will display the query on the screen and then pause until the user clicks the **OK** button (Figure 2-3).

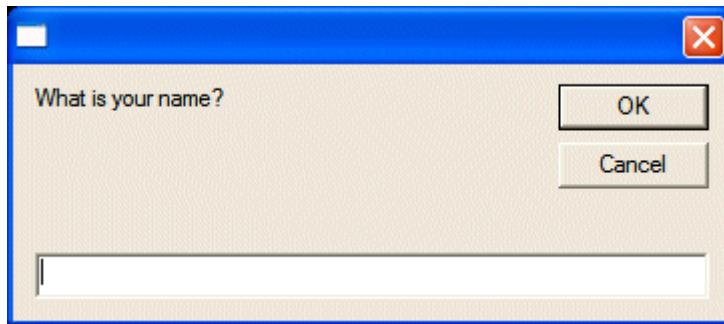


Figure 2-3. The form/dialog displayed by the *InputBox()* function.



Note: There are other parameters to the *InputBox()* function. These allow you to perform actions such as setting the title of the *InputBox()* form, specifying its location on the screen, and so forth (see also Appendices E and F for more details on VBScript references and third-party documentation).

MsgBox()

The simplest way to use the *MsgBox()* function is to call it with its first parameter set to be your message string; for example, consider the following statement:

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

This will cause the script to display the string "You are magnificent!" on the screen and then stop until the user clicks the **OK** button (Figure 2-4).



Figure 2-4. The form/dialog displayed by the *MsgBox()* function.

Now consider the following snippet of code that combines the use of the *InputBox()* and *MsgBox()* functions:

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

Observe the use of 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.

Let's assume that the user enters the name "Max" into the *InputBox()* function. In this case, the *MsgBox()* will display "You are magnificent Max" on the screen (Figure 2-5).

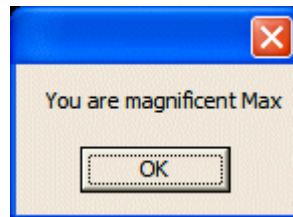


Figure 2-5. The form/dialog displayed by the *MsgBox()* function.



Note: There are other parameters to the *MsgBox()* function. These allow you to perform actions such as setting the title of the *MsgBox()* form, specifying its location on the screen, and so forth (see also Appendices E and F for more details on VBScript references and third-party documentation).



Note: 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 will automatically convert the value represented by the integer variable *tempInt* 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 below (see also Appendices E and F for more details on VBScript references and third-party documentation).

Conversion Functions [CInt(), CDbl(), CStr(), ...]

Not surprisingly, the conversion functions are used to convert data from one type to another. For example, suppose you have a string "2468" that you want to convert into an integer; you could achieve this using the *CInt()* function as demonstrated in the following snippet of code:

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

By comparison, the *CDbl()* function converts strings into real (double-precision) numbers. For example, suppose you have a string "3.142" that you want to convert into a real number; you could achieve this using the *CDbl()* function as demonstrated in the following snippet of code:

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

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

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



Note: The *CStr()* function will convert from any format into a string; for example, a real number like 3.142 will be automatically converted into its string equivalent of "3.142", a Boolean variable will be 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 [Len(), Left(), Right(), Replace(), ...]

VBScript provides a variety of functions that can be used to manipulate strings. For example, the *Len()* function returns the length of a string:

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

In this case, the *lenInt* integer variable will end up containing 11, which is the number of characters in the *testStr* string.

The *Left()* function returns a substring comprising some number of characters copied from the left-hand side of a source string; for example:

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

In this case, the substring *subStr* will end up containing the leftmost five characters from *mainStr*; that is, *subStr* will end up containing the word *Hello*.

The *Right()* function is the counterpart to the *Left()* function; that is, the *Right()* function returns a substring comprising some number of characters copied from the right-hand side of a source string; for example:

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

In this case, the substring *subStr* will end up containing the rightmost five characters from *mainStr*; that is, *subStr* will end up containing the word *World*.

In the examples above, the first parameter to the *Len()*, *Left()*, and *Right()* functions was shown as being a string variable, but this could also have been a string. Similarly, the second parameter to the *Left()* and *Right()* functions was shown as being an integer, but this could also have been an integer variable. Consider the following snippet of code, which will generate exactly the same result as the previous example:

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

Last but not least, the *Replace()* function can be used to replace 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 intellect"  
mainStr = Replace(mainStr, "big", "small")
```

The first parameter is the source string, the second parameter is the old substring to be replaced, and the third parameter is the new substring. Thus, in the case of this example, *mainStr* will end up containing: *You have a small intellect*.

As usual, the parameters to the *Replace()* function may be strings or string variables. Consider the following snippet of code, which will generate exactly the same result as the previous example:

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

Special Characters

There are a number of special characters of which you should be aware if you are processing textual data or formatting output using the string manipulations discussed in the previous topic.

First however, we should note that the *Chr()* function converts an integer (or an integer variable) into a character. For example, *Chr(9)* will return a horizontal <Tab> character. This means that, if we wanted to replace a group of four spaces in a string with a <Tab> character, for example, we could do so using the following snippet of code:

```

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

```

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

vbCr	Chr(13)	Carriage return
vbLf	Chr(10)	Line feed
vbCrLf	Chr(13)Chr(10)	Carriage return and line feed
vbTab	Chr(9)	Horizontal tab

There is also the very useful *vbNewLine* constant, which is platform-independant. That is, it either equates to *Chr(10)* or *Chr(13)Chr(10)* depending on the operating system under which your script is running (Windows® or UNIX/Linux) without your having to change anything.

FormatNumber()

The *FormatNumber()* function is used to format a number for display and to return a string. The general form of function is demonstrated in the following pseudo-code snippet:

```

Dim numberStr
numberStr = FormatNumber(<p1>,<p2>,<p3>,<p4>)

```

The first parameter – shown as *<p1>* above – is an integer (e.g. 1234) or a real number (e.g. 3.142) or an integer or real variable, or an expression based on numbers or variables that resolves into a number (e.g. 3.0 + 0.142). Only the first parameter is required, the rest are optional.

The second parameter – shown as *<p2>* above – specifies the number of digits that are to be displayed after the decimal point; consider the following snippet of code, for example:

```

Dim numberStr
numberStr = FormatNumber(3.142857,3)

```

This will leave *numberStr* containing 3.143 (observe that this is 3.143 rather than 3.142 because the value is rounded during the process of discarding the unwanted digits).

The third parameter is a Boolean (or a Boolean Variable); a value of *True* instructs the function to display a leading zero, while a value of *False* will prevent 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 (or a Boolean Variable); a value of *True* instructs the function to display negative values in parentheses, while a value of *False* will cause negative values to be displayed 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 above) will assume their default values, which are defined as part of the host computer's regional settings.

Date(), Time(), Now(), and FormatDateTime() Functions

The *Date()* and *Time()* functions are useful when writing out reports. The *Date()* function returns the current date and can be used as a string; for example:

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

As opposed to assigning the output from the *Date()* function to a string as shown above, you can use the function as part of the argument to the *MsgBox()*; for example:

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

The way in which the *Date()* is formatted into a string 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). In the USA, the string displayed on the *MsgBox()* from either of the code snippets above would appear as illustrated in Figure 2-6.

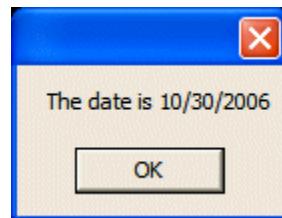


Figure 2-6. Using the *MsgBox()* function to display the date.

Similarly, the *Time()* function returns the current time and can be used as a string as illustrated in the following 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). Thus, the string displayed on the *MsgBox()* from the statement above would appear as illustrated in Figure 2-7.

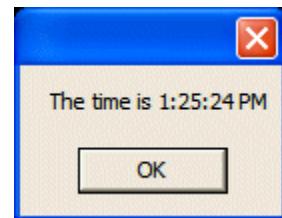


Figure 2-7. Using the *MsgBox()* function to display the time.

The `Now()` function returns the date followed by the time, so the following two statements will provide identical results (Figure 2-8):

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

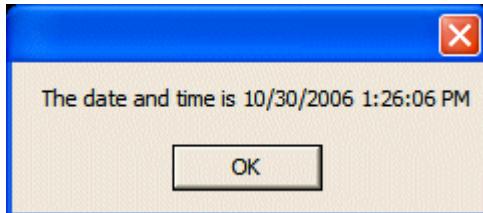


Figure 2-8. Using the `MsgBox()` function to display the date and time.

The `FormatDateTime()` function can be used to change the format of the date or the time. If you want to format the date, the first parameter to the `FormatDateTime()` function will typically be either the `Date()` or `Now()` functions. By comparison, if you want to format the time, the first parameter to the `FormatDateTime()` function will typically be either the `Time()` or `Now()` functions; 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 to the `FormatDateTime()` function is an enumerate (or an integer) that specifies the type of formatting to be performed. The supported enumerate names, equivalent integer values, and corresponding results are as follows:

Enumerate Name	Value	Use With	Example Output
<code>vbGeneralDate</code>	0	Date or Now	9/7/2006
<code>vbLongDate</code>	1	Date or Now	Thursday, September 07, 2006
<code>vbShortDate</code>	2	Date or Now	9/7/2006
<code>vbLongTime</code>	3	Time or Now	11:30:59 AM
<code>vbShortTime</code>	4	Time or Now	11:30



Note: the first parameter to the `FormatDateTime()` function could 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; for example, if the act of moving a component causes a short, which – in turn – causes the server to issue a *Design Rule Check (DRC)* error.

If such an error occurs and you haven't added any code into your script, then by default the script will abort and the error will either be displayed 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).

The point is that, in some cases you may not want the script to abort; that is, you recognize that what you are trying to do may cause an error and you want to handle it yourself. 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

```

With regard to the *If... Then... End If* test, a value of 0 is equivalent to a Boolean value of *False*, while 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

```

Note that *Err* is an object that is simply there; that is, there's no need to use a *Dim* statement to declare it. Meanwhile, *.Number* is a property of the *Err* object. Different integers correspond to different errors, and these integers are mapped on to enumerates. For example, the enumerate for a DRC error in Expedition PCB is *epcbErrCodeDRCViolation*. This means you could 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 MGCPBCB interface by the *EPcbErrCode* (see also *Appendix D: General VBScript References and Tools* for more information on using Object Browsers to locate this type of information). A list of these enumerates, their values, and their descriptions can be found in the help file under the *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

```

As opposed to comparing the *Description* string, it is recommended to compare the *EPcbErrCode* enumerate because this is guaranteed to remain the same from release to release (while it is unlikely that the *Description* string will change, there is no guarantee that it will not change). An alternative use of *Err.Description* is when writing the error to a log file; for example, assume that there is a *WriteErrorToFile()* function that writes errors to a log file and consider 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

```

Once the script has handled an error, we could use the *Err.Clear* statement to reset the *Err* object and clear it to a null error value; 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 we've handled an error might be viewed as "*cleaning up our own mess*". This will work if (a) we always do things this way and (b) our script never calls anyone else's scripts and/or library routines (the concepts of scripts calling other scripts and/or library routines are discussed later in this tutorial).

However, if our script has already called someone else's script(s) and/or library routines, then we can't be sure that they have "*cleaned up their mess*." In order to address this, we could use the *Err.Clear* statement just after the *On Error Resume Next* statement (before we generate and handle the error) to ensure that we don't accidentally pick up and 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 we handle 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 our calling someone else's scripts, and it also works for our own scripts so long as we always do things this way.

But a potential problem still remains with the above technique. Supposing that our script generates and responds to an error, and then it calls someone else's script(s) and/or library routines. If this "third party code" attempts to perform error handling – and if it uses the technique of applying the *Err.Clear* statement after it's handled an error – then it may mistakenly see a "false error" from our portion of the script (that is, an error that we addressed but didn't clear afterwards).

Thus, the absolute safest practice would be to use the *Err.Clear* statement both before we handle an error (thereby cleaning up anyone else's mess) and also after we've handled 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 everyone will 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 will cause it 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

```



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 our executing an *On Error Goto 0* statement before the end of the routine). Furthermore, the system will automatically return to the default error handling mode within any *called* routines; in this case, however, the system will return to the *On Error Resume Next* mode when the called routine returns control to the calling routine.

To put this in a nutshell, the effects of *On Error Resume Next* apply only within the routine in which this statement is found; any other routines (called or calling) will use the default error mode unless otherwise specified within those routines themselves.

Chapter 3: Running Your First Script

Introduction

This chapter will first guide you through the process of running a simple "Hello World" type script. Next, you will create a slightly more complex script that counts the number of vias in the currently open layout design. This script, which requires the use of the automation license server, will be used as the basis for some of the subsequent introductory chapters in this section of the tutorial.

Creating and Running the Hello World Script

This script simply calls the VBScript `MsgBox()` function to display the string *Hello World* on your screen. For the purposes of these discussions, you will be calling your script from within the Expedition PCB application. Since this script does not access or modify any of the application's internal data structures, it does not require any special server validation code (compare this to the *ViaCount.vbs* script later in this chapter).

Creating the Hello World Script

Expedition PCB and other products in the flow support a built-in editor called the *Integrated Development Environment (IDE)* (see also *Chapter 4: Introducing the IDE*). There are 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, automation scripts are ASCII files than can be created and modified using any text editor. If you are running under the Windows® operating system, for example, invoke the Notepad editor and use it to key-in the following statement:

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

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

Running the Hello World Script

For the purposes of this example, you will run your script from within Expedition PCB (this technique is covered in greater detail in *Chapter 9: Running a Script From Within an Application* – see also *Chapter 8: Running a Script From the Command Line*).

- 1) Launch Expedition PCB.



Note: Some scripts require a design to be open in order to perform their desired function; others (like this one) 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 user, or you can write the script in such a way that it checks to see if a design is already open and – if not – to either open a design automatically or to prompt the user to do so.

- 2) Right-click on the toolbar and ensure that the **Keyin Command** option is ticked (enabled).
- 3) Enter the string *run C:\Temp\HelloWorld.vbs* into the **Keyin Command** field (Figure 3-1).

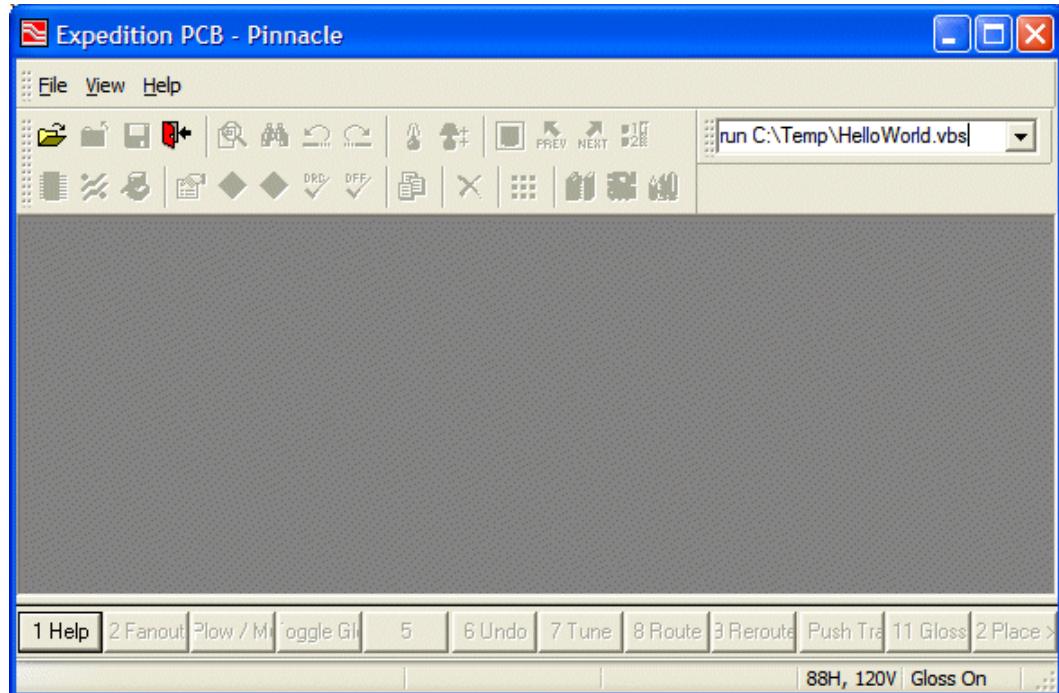


Figure 3-1. The Keyin Command Field.



Note: This string is *case-insensitive* on Windows platforms and *case-sensitive* on UNIX and Linux platforms.



Note: If there are any spaces in the path name to the file, then the entire path must begin and end with double quotes (the space between the *run* keyword and the path name doesn't count).

- 4) Press the <Enter> key on your keyboard to execute your script. Observe the *MsgBox()* form/dialog containing the Hello World string appear on your screen (Figure 3-2).



Figure 3-2. The *MsgBox()* displaying the "Hello World" string.

- 5) Click the **OK** button to dismiss the *MsgBox()* form/dialog (this also terminates this particular script, because there's nothing else to do).

Creating and Running the Via Count Script

This script counts and displays the number of vias in whatever layout document is currently open and active in the Expedition PCB application. As part of the process of creating this script, you will discover how to use the Help system to work your way through the application's data structures. Due to the fact that this script will be accessing the application's

internal data structures, it does require some special server validation code (compare this to the *HelloWorld.vbs* script earlier in this chapter).

Visualizing the Data Hierarchy

The data structures inside Mentor applications are organized in a hierarchy. Each application may represent its data structures and hierarchy in a different way. For the purposes of this portion of our discussions, we shall consider the data structures used by Expedition PCB.

At the top of the hierarchy is the *Application*. You may think of this as Expedition PCB running without having a layout design open. Next in the hierarchy is a *Document*, for example, a layout design in Expedition PCB. You can visualize the data residing in the Document in multiple ways; two examples are as illustrated in Figure 3-3.

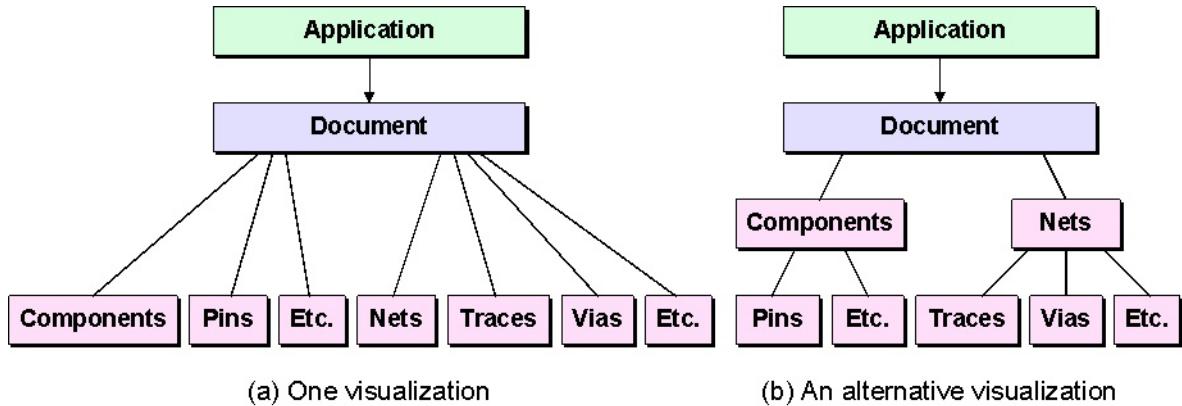


Figure 3-3. Two ways of visualizing the data hierarchy in Expedition PCB.

In reality, everything is "owned" by the *Document*; this means that you can access the *Vias*, for example, using *Document* > *Vias* as shown in Figure 3-3(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 3-3(b).



Note: The set of *Vias* returned by the *Document* and the *Net* are different. This is due to the fact that when accessing the *Document*, all of the *Vias* on the board are returned; by comparison, when accessing a *Net*, only the *Vias* on that *Net* are returned.

Using the Help System

As was previously noted, you are soon going to create a script to count all of the vias in a layout document. Let's assume, however, that you are not familiar with the data hierarchy – including the methods and properties associated with the various data elements – in Expedition PCB. In this case, you may use the Help system as follows:

- 1) Launch Expedition PCB.
- 2) Use the **Help > Contents > Automation** command.
- 3) Since you want to navigate through the data hierarchy, select the **Hierarchical Overview** item in the dialog, double-click on this item to open it up, and observe that it contains a single sub-item called **Application Object** (Figure 3-4).

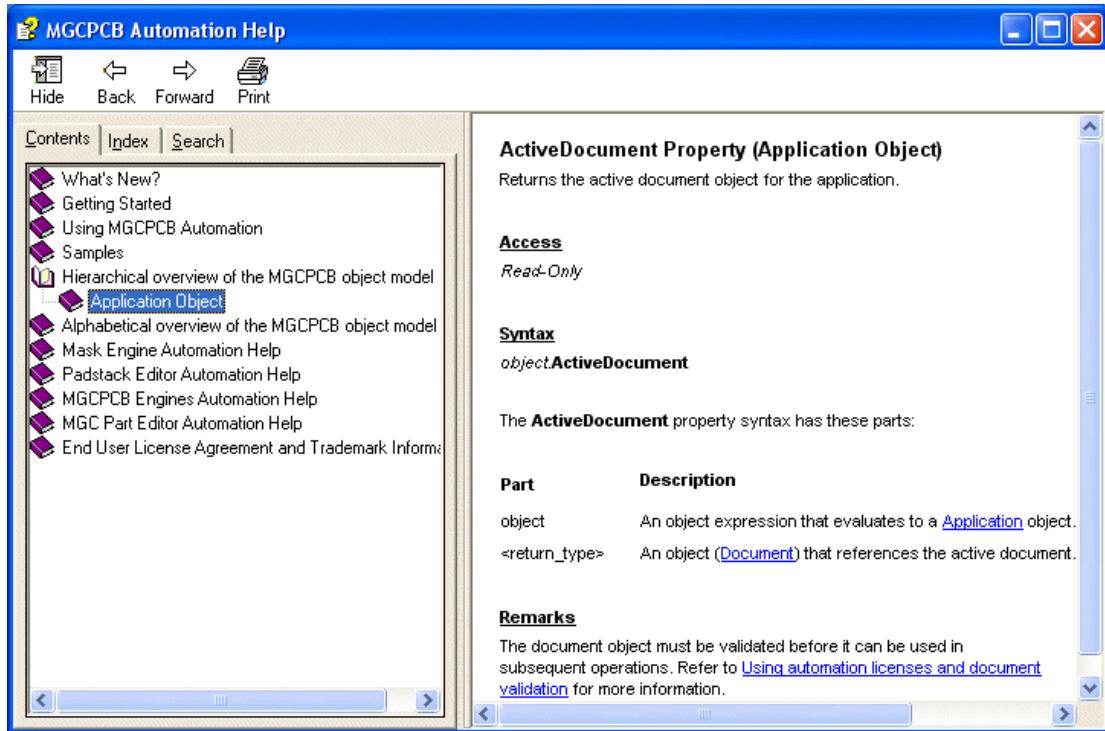


Figure 3-4. Locating the Application Object.

- 4) Double-click on the **Application Object** to see all of the methods and properties associated with this object. One of these is the **Active Document** property; double-click on this and observe the **Property Description** that informs you how to use it; also observe the **Document Object**.
- 5) Double-click on the **Document Object** to see all of the methods and properties associated with this object.
- 6) Double-click on the **Vias** property and observe the **Property Description** that informs you how to use it; also observe that the return value from this property is a **Collection** (of vias).
- 7) Double-click on the **Collection** item and observe that there is a **Count** property associated with it.
- 8) Click on the **Count** property and observe the description on how to use it as illustrated in Figure 3-5. By performing the above steps, you now know that the hierarchy your script will use to count the vias in the design is as follows:

Application > Document > Vias > Count Property

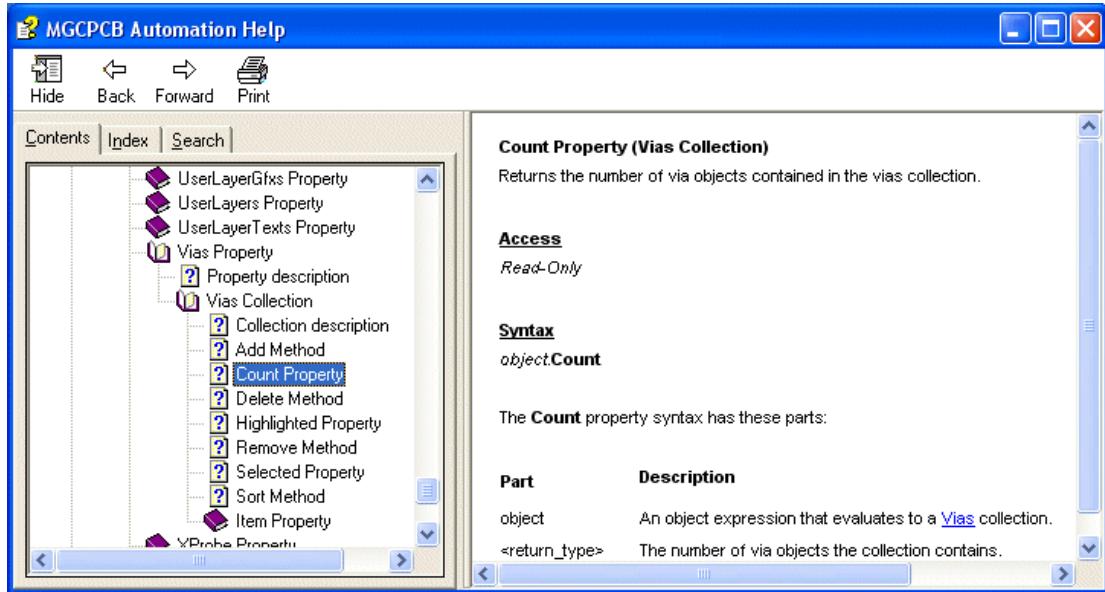


Figure 3-5. Working one's way through the hierarchy.

Creating the Via Count Script

Use the NotePad editor or the text/scripting editor of your choice to enter the following script in its entirety and then use the **File > Save As** command to save this file as *ViaCount.vbs* in the *C:\Temp* folder on your system (alternatively, you can access a pre-defined version of this script as discussed in *Appendix E*).

```

1 Option Explicit
2
3 ' Add any type libraries to be used.
4 Scripting.AddTypeLibrary( "MGCPBCB.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 =
42          CreateObject("MGCPBCBAutomationLicensing.Application")
43
44      ' Ask the license server for the license token
45      licenseTokenInt = licenseServerObj.GetToken(keyInt)
46
47      ' Release license server
48      Set licenseServerObj = nothing
49
50      ' Turn off error messages (validate may fail if the
51          token is incorrect)
52      On Error Resume Next
53      Err.Clear
54
55      ' Ask the document to validate the license token
56      docObj.Validate(licenseTokenInt)
57      If Err Then
58          ValidateServer = 0
59      Else
60          ValidateServer = 1
61      End If
62
63  End Function

```

Observe the statement that starts on line 41 and continues on the following line (without a line number). This actually a single statement that would occupy a single line in the script; it is displayed here using two lines only for purposes of formatting and presentation in this document. Similarly for the statement that starts on line 49.

Now, let's step through the first half of this script line-by-line. The script commences with the *Option Explicit* statement, whose function and use is discussed in *Chapter 2: VBS Primer* and *Appendix A: Good Scripting Practices*.

1 Option Explicit

On line 4 we see one use of the *Scripting* object. This is a special implicit object that is always present and available to scripts running in (or on) Mentor's applications; that is, you do not have to define or declare the *Scripting* object, because it's already there. There are a number of methods and properties associated with the *Scripting* object (see the **Automation Help** for more details as discussed in the previous topic).

```

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

```

The method of interest to us here is *AddTypeLibrary*, which provides access to any enumerate values associated with a specified server. Observe that the Expedition PCB

server is specified using its *Prog ID* (this is *MGCPCB.ExpeditionPCBAApplication* as was discussed in *Chapter 1: Introduction*).

Remember that, for the purpose of this portion of our discussions, we are assuming that this script will be launched from within the Expedition PCB application. Based on this, on line 7 we declare a PCB *Application* object called *pcbAppObj*, and on line 8 we assign the application to this object.

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

Like the *Scripting* object, *Application* is a special implicit object that is always present and available to scripts running in (or on) Mentor's applications; that is, you do not have to define or declare the *Application* object, because it's already there. This is the simplest and best way to approach things if you know that your script is always going to be run from within Expedition PCB (we'll consider an alternative later in this chapter); this is because you inherently know to which instantiation of Expedition PCB the script will be attached (if the script is run from outside the application and there are multiple instantiations of Expedition PCB running, then the instantiation to which the script will attach itself is arbitrary).

Now, assume that the user will have opened a layout design document prior to running the script. Based on this. On line 11 we declare a PCB *Document* object called *pcbDocObj*, and on line 12 we assign the *ActiveDocument* property associated with the *Application* object to this *Document* object.

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

On line 15 the script calls the *ValidateServer* function using the *pcbDocObj* object as a parameter. This is a user-declared function that handshakes with the license server in order to acquire any necessary licenses required to run automation such that your script can access and modify design data. This action *must* be performed 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 you've formally declared them in your script.

On line 18 we declare a *Via* collection object, called *viaColl*, and on line 19 we assign the collection of the vias associated with the *Document* object to the *Via* collection object.

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

From our examination of the **Automation Help**, we know that there is a *Count* method associated with a via collection that returns the number of vias in the collection. Thus, on line 22 we declare an integer variable called *countInt*, and on like 23 we use the *viaColl.Count* statement to load *countInt* with the number of vias in the design.

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

Last but not least (for the body of the script), on line 25 we use a *MsgBox()* function to present a form/dialog on the screen displaying the total number of vias in the design.

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

The server validation itself is to be found between lines 31 and 61 in this script. We could walk you through this function, but there's no point, because all you will do in the future is to *copy* the function from this script, *paste* it into your new script, and *forget* about it thereafter (we're not kidding).

Running the Via Count Script

Once again, you will run your script from within Expedition PCB (this technique is covered in greater detail in *Chapter 9: Running a Script From Within an Application*).

- 1) Launch Expedition PCB.
- 2) Open a layout design of your choice.
- 3) Right-click on the toolbar and ensure that the **Keyin Command** option is ticked (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 interface (Figure 3-6).

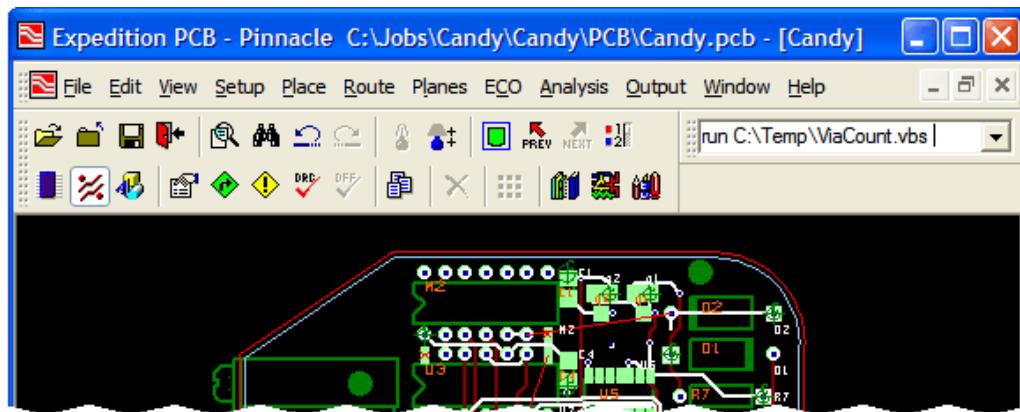


Figure 3-6. The Keyin Command Field.

- 5) Press the <Enter> key on your keyboard to execute your script. Observe the *MsgBox()* form/dialog displaying the number of vias in your design appear on your screen (Figure 3-7).



Figure 3-7. The *MsgBox()* displaying the number of vias in the design.

- 6) Click the **OK** button to dismiss the *MsgBox()* form/dialog (this also terminates this particular script, because there's nothing else to do).

An Alternative Technique

In the original version of the Via Count script as discussed above, the way in which we declared and accessed the *Application* and *Document* objects was as follows:

```
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
```

An alternative technique is as shown below. Observe that, as opposed to storing the *Application* object in our own variable, this new version uses this 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 here only for the sake of completeness. Although it may seem more efficient, this technique is not recommended. This is because this approach makes it harder to take a script that was originally intended to run on an in-process client and modify it to run on an out-of-process client. For example, consider our original version of the script, which was created under the assumption that it was going to run as an in-process client on Expedition PCB:

```
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, let's assume that you decide to use this same script to extract vias from an Expedition PCB document, but that you want to run the script as an out-of-process client. Out-of-process clients are generally run from the command line, but they could also be scripts running in another product – FabLink XE for example. The point is that the implicit *Application* object is not available to out-of-process scripts. In order to address this, you could quickly and easily modify the script as follows:

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

In this case, *GetObject* is a function in the VBScript language. You can use this function to attach to an instantiation of Expedition PCB that is already running using its *Prog ID* (as was discussed in *Chapter 1: Introduction*, the Prog ID used to attach to Expedition PCB is

MGCPCB.ExpeditionPCBAApplication).



Note: If you have multiple instantiations of the same application open, then it's arbitrary as to which instantiation the *GetObject* function will attach.

Alternatively, if an instantiation of Expedition PCB were not already running, you could use the *CreateObject* function to automatically launch the application as shown below.

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

Note that, in this case, once the script has launched the Expedition PCB application, the script would either have to direct Expedition PCB to open a layout design document or prompt the user to do so.

Chapter 4: Introducing the IDE

Introduction

Forms (dialogs) are useful in cases where more data needs to be entered than can be supported by primitive *InputBox()* functionality. Forms are also useful for displaying information to the user in an organized manner, and to have that information update dynamically. Scripts are good at pulling things together and automating tasks, but most scripting languages have limited graphical interface support built into them. In order to address this, Mentor products feature an Integrated Development Environment (IDE) that allows users to create forms. The IDE is the preferred mechanism for creating scripts and forms (dialogs) in the Expedition PCB tool flow.

In this chapter, you will first discover how to use the IDE to create an example script in VBScript. In fact, this script will be based on the *CountVias* script that was presented in the previous chapter.

Next, you will use the IDE to create a simple form/dialog with three options/buttons named *Count All*, *Count Selected*, and *Cancel*. Initially, you will implement the *Cancel* functionality to simply cancel out of the form. You will then implement the *Count All* functionality; once again, this will be based on the *CountVias* script that was presented in the previous chapter. Next, you will slightly modify the original script and use the new version to perform the *Count Selected* functionality. And, finally, you will perform some simple "housekeeping" tasks.

Using the IDE to Create a Script

- 1) Invoke Expedition PCB and open up a circuit board design containing some number of traces and vias. Now use the **File > New Script Form** command to access the **Select Script Language** dialog (Figure 4-1)



Figure 4-1. The Select Script Language dialog.

- 2) Use the pull-down list to select VBScript in the **Form Language** field and ensure that there's a check in the **Script** checkbox (this informs the IDE that you are intending to create a script only).
- 3) Click the **OK** button to accept these settings and launch the IDE in its *text editor mode* (Figure 4-2). Observe the *(Declarations)* and *End of (Declarations)* comments in the main editing window. All of your script statements must be entered between these two comment lines.

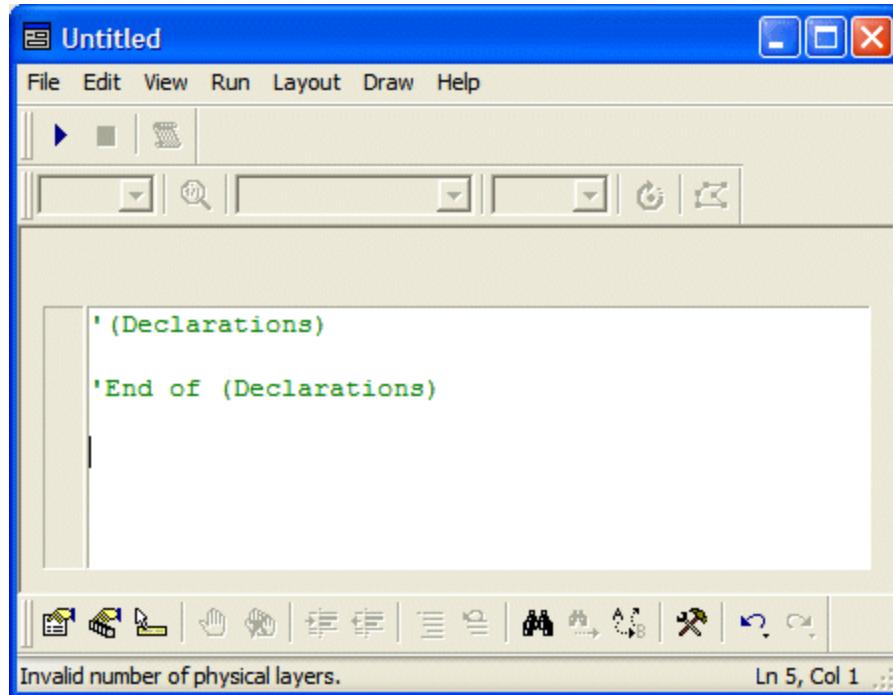


Figure 4-2. The IDE in its text editor mode.

- 4) As you will discover, 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).

Normally, you will type your script directly into the main editing window. For the purposes of this example, however, you will use the script you created in the previous chapter. Use the editor of your choice to open your *ViaCount.vbs* script; use the **Edit > Select All** command to select the entire script in this editor; use the **Edit > Copy** command to copy the script into the paste buffer; then use the **Edit > Paste** command in the IDE to paste the script between the *(Declarations)* and *End of (Declarations)* comments as illustrated in Figure 4-3.

- 5) Use the **File > Save As** command to save this IDE-based version of the script as *ViaCount-IDE.vbs*.

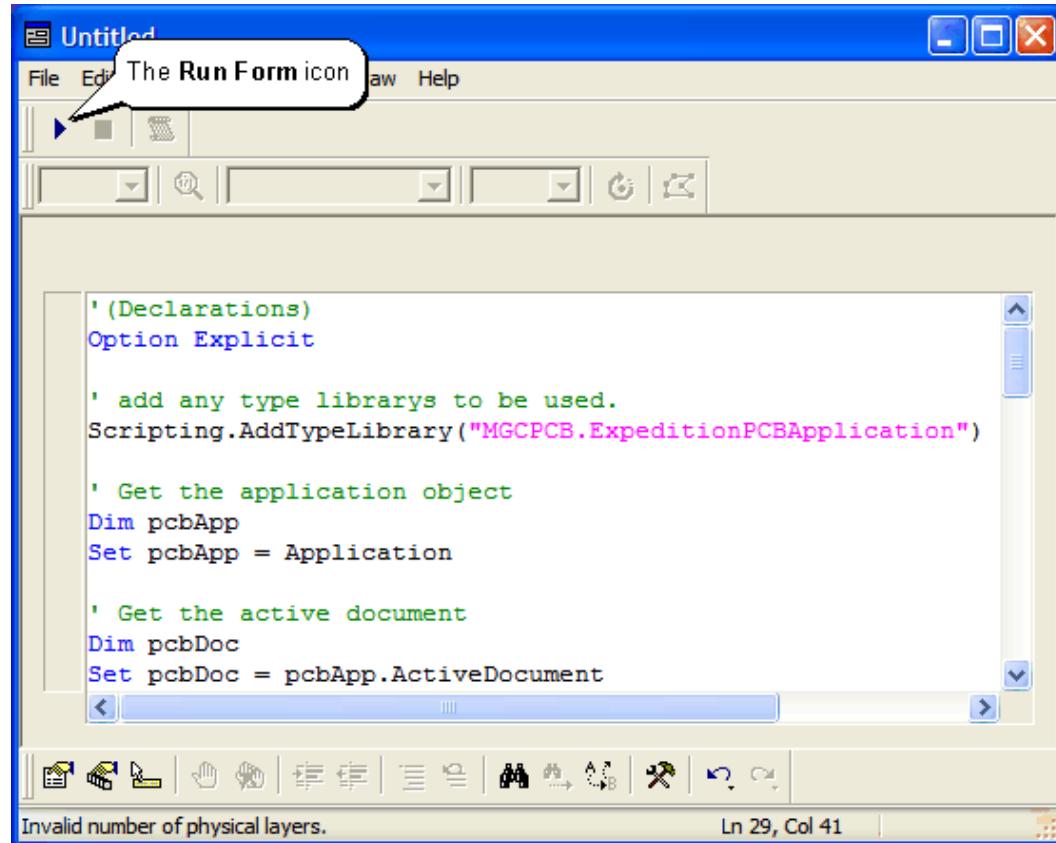


Figure 4-3. The IDE in its text editor mode.

- 6) You can now run your new script directly from the IDE by clicking the **Run Form** icon (Figure 4-3); this is the right-pointing arrow that looks like the "Play" button on a VCR or DVD player and that is located in the upper-left-hand corner of the IDE. Alternatively, you can use the **Run > Run** command or the **F5** function key to achieve the same effect.

The result is shown in Figure 4-4 (your actual via count value may be different depending on the design you are working with).



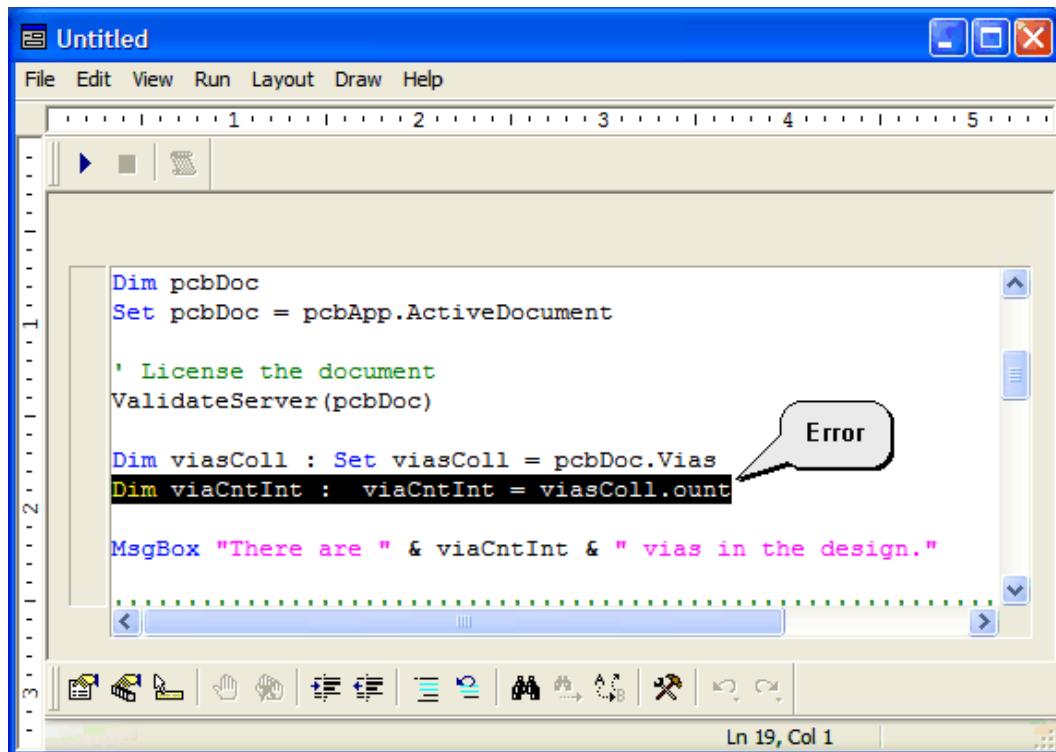
Figure 4-4. The output from the script.

- 7) Remember that your script calls a *MsgBox()* function to display the total number of vias in the design. Observe that the IDE brings up a dummy form – currently named *Untitled* – in which the script is run; the *MsgBox()* is presented in the middle of Figure 4-4.
- 8) Click the **OK** button to dismiss the *MsgBox()*.
- 9) Observe that the *MsgBox()* disappears, but that the IDE form continues running. Click the **Stop** icon (Figure 4-4); this is the small square that looks like the "Stop" button on a VCR or DVD player and that is located in the upper-left-hand corner of the IDE form. This will terminate the form and return you to the IDE's *text editor mode*.

The IDE's Error Handling Capability

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* as illustrated in Figure 4-5.



The screenshot shows the IDE window titled "Untitled". The menu bar includes File, Edit, View, Run, Layout, Draw, Help. The toolbar has icons for Run Form, Stop, and others. The code editor contains the following VBScript:

```

Dim pcbDoc
Set pcbDoc = pcbApp.ActiveDocument

' License the document
ValidateServer(pcbDoc)

Dim viasColl : Set viasColl = pcbDoc.Vias
Dim viaCntInt : viaCntInt = viasColl.ount

MsgBox "There are " & viaCntInt & " vias in the design."

```

A speech bubble labeled "Error" points to the line "Dim viaCntInt : viaCntInt = viasColl.ount". The status bar at the bottom right says "Ln 19, Col 1".

Figure 4-5. Introducing a deliberate error.

- 2) Run this new version of your script directly from the IDE by clicking the **Run Form** icon.
- 3) Observe the resulting **Run-Time Error** dialog (Figure 4-6). This tells you what your error is and presents you with two options: **Continue** or **End**.
 - Selecting the **Continue** option will instruct the IDE to ignore this error and attempt to continue running the script.
 - Selecting the **End** option will launch the **VBS Mini-Editor** that you can use to correct the error.

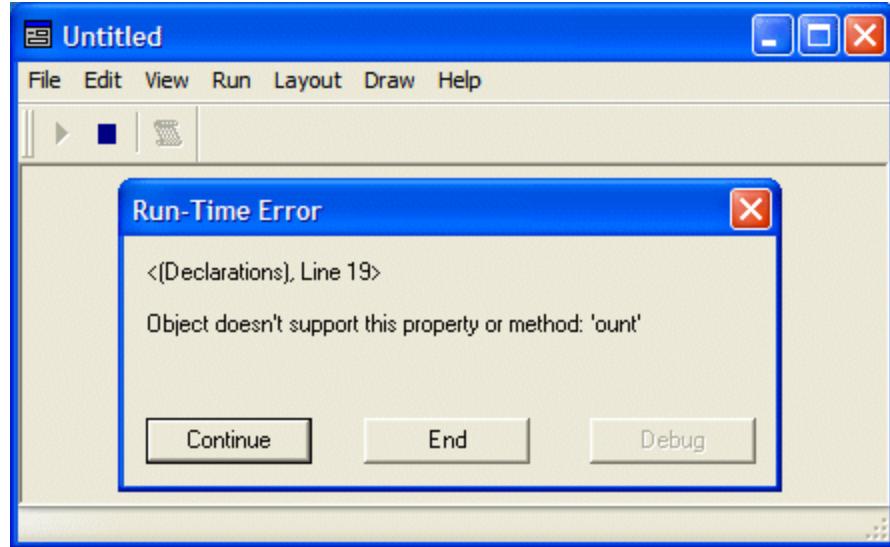


Figure 4-6. The Run-Time Error dialog.

- 4) Click the **End** button to access the **VBS Mini-Editor**. This will take you to the statement containing the error and highlight that statement as illustrated in Figure 4-7.

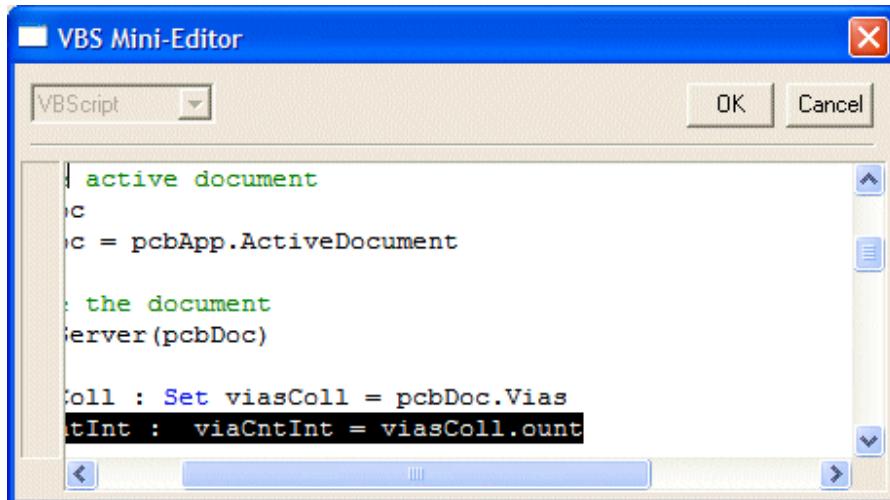


Figure 4-7. Using the VBS Mini-Editor to correct an error.

- 5) Correct the error (change *viasColl.ount* back to *viasColl.Count*) and click the **OK** button. This dismisses the mini editor and returns you to the main script editor where you will discover that the changes you made using the mini editor have been implemented.

Using the IDE to Create a Form/Dialog

Launching the IDE in its Form Editor Mode

- 1) Invoke Expedition PCB and open up a circuit board design containing some number of traces and vias. Now use the **File > New Script Form** command to access the **Select Script Language** dialog (Figure 4-8)



Figure 4-8. The Select Script Language dialog.

- 2) Use the pull-down list to select VBScript in the **Form Language** field and ensure that there the **Script** checkbox remains *unchecked* (this informs the IDE that you are intending to create a form/dialog).
- 3) Click the **OK** button to accept these settings and launch the IDE in its *form editor mode* (Figure 4-9).

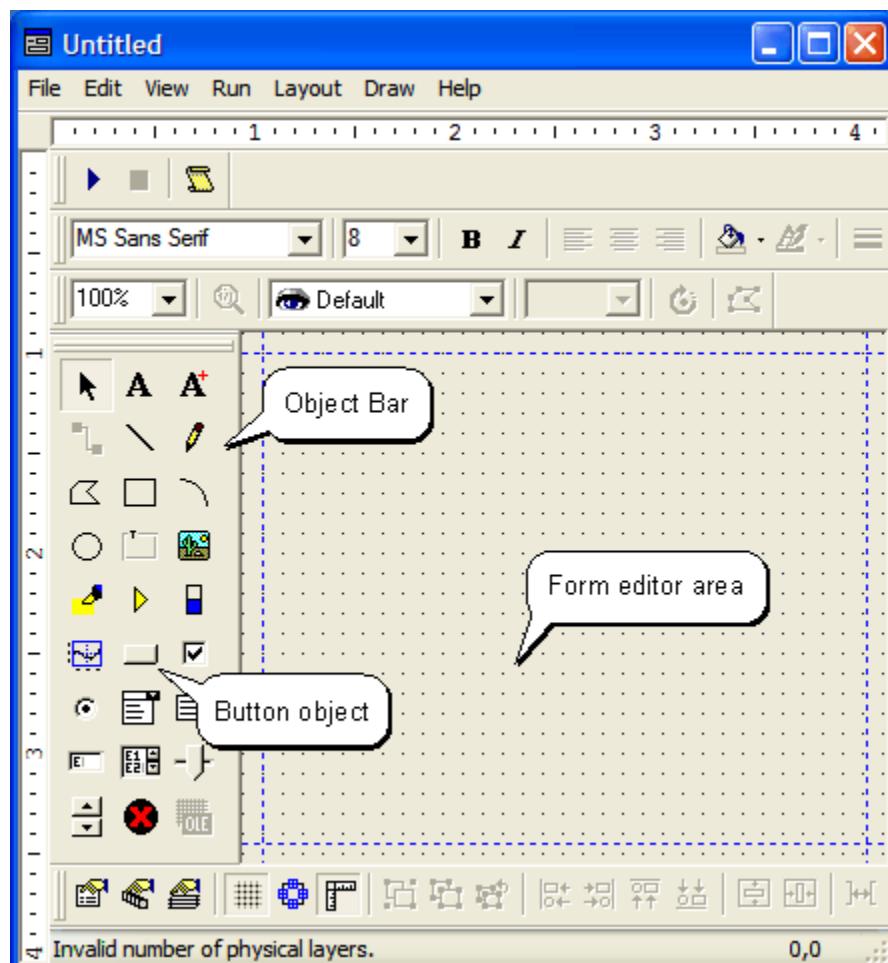


Figure 4-9. The IDE in its form editor mode.

Adding Graphical Buttons to the Form/Dialog

- 1) On the left of the form editor is the **Object Bar**, which presents graphical depictions of the various objects you can add to your forms. For the purposes of this example, you are going to add three buttons (*Count All*, *Count Selected*, and *Cancel*) to your form. Click the **Button** object (the icon that looks like a button in the middle of the **Object Bar**) and then move the mouse cursor into the main form editor area and click again to place your button (Figure 4-10).

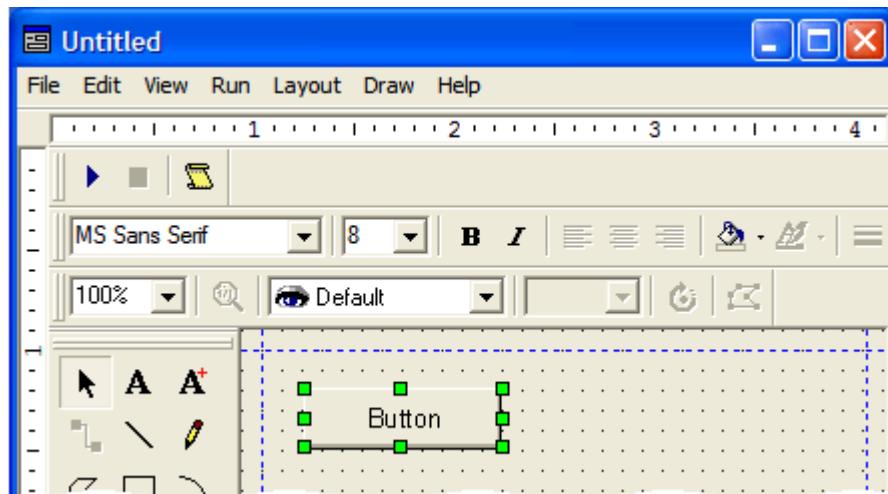


Figure 4-10. Oh, the excitement of placing your first button!



Note: When in the *form editor mode*, you can add controls such as buttons, radio buttons, combo boxes, check boxes, and list boxes to your form. Also, you can add static text, lines, and geometric shapes as required.



Note: Observe the green "handles" associated with the button (four in the corners and four on the sides). You can use these handles to resize the button as required. Also, if you move the mouse cursor over the top of the button, it will change into a four-headed arrow, at which point you can drag-and-drop the button to locate it wherever you want in the form.

- 2) Right-click on the button and then select the **Object Properties** item from the resulting pop-up menu. This brings up the **Property Sheet** dialog as shown in Figure 4-11.
- 3) Make sure you are on the **Normal** tab. The two items of interest for the purposes of this portion of our discussions are the **(Object Code)** and **Text** items:
 - The attribute associated with the **(Object Code)** item is the name of the button that will be "seen" by the underlying script. To put this another way, this is the variable name used to access the control from the script.
 - The attribute associated with the **Text** item is the name of the button that will be presented to the user as an annotation on the button on the form.
- 4) Set the attribute associated with the **(Object Code)** item to *countAllBtnObj* (this stands for "Count All Button Object"). Observe that this name automatically appears in the pull-down list field at the top of the **Property Sheet** dialog.
- 5) Set the attribute associated with the **Text** item to *Count All*.

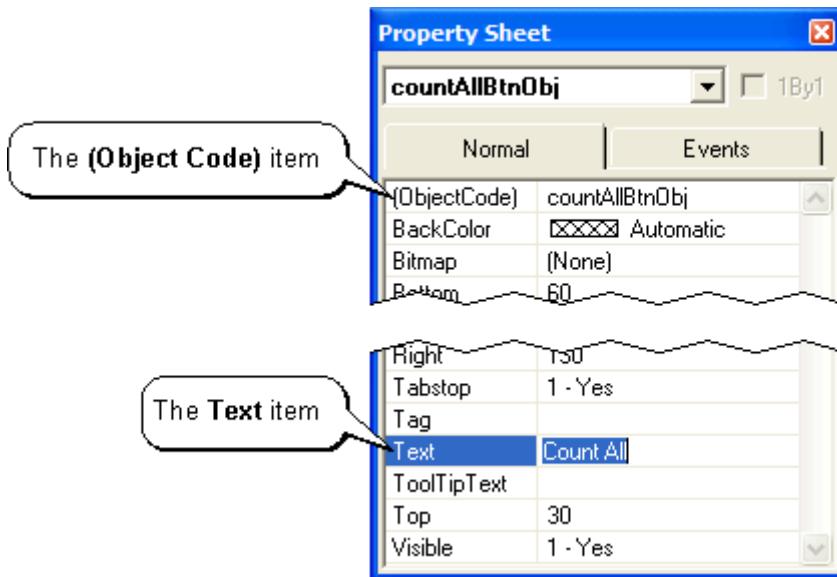


Figure 4-11. The Property Sheet dialog.



Note: The following abbreviations are recommended for objects used in forms:

- *BtnObj* = "Button object"
- *RadObj* = "Radio button object"
- *CmbObj* = "Combo-box object"
- *ChkObj* = "Check-box object"
- *LstObj* = "List box object"

- 6) Once you've made these changes, either press the <Esc> key or click the small red 'X' button on the right-hand side of the title bar to dismiss this form (this may seem strange, but there's no need to formally apply these changes because they are automatically applied as you are keying them in).
- 7) Repeat steps (1) through (6) to create two more buttons with the following attributes (the result should look something like Figure 4-12):

Text	(Object Code)
Count Selected	<i>countSelBtnObj</i>
Cancel	<i>cancelButObj</i>

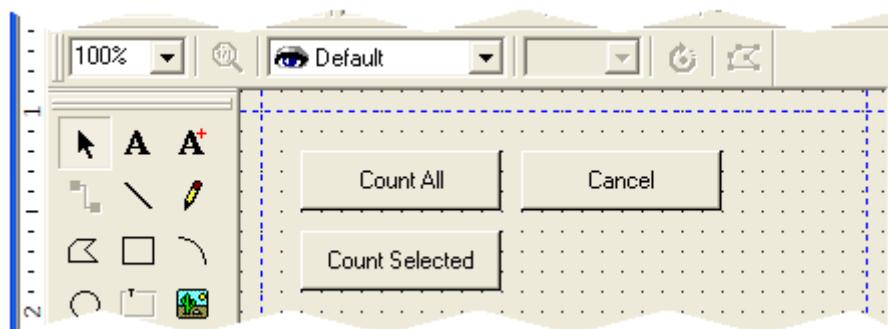


Figure 4-12. The Count All, Count Selected, and Cancel buttons.

Adding Code "Behind" the Cancel Button

Now that you've designed your form, it's time to add the code (scripts) "behind" the various buttons. There are a number of mechanisms by which this may be achieved. In general, the simplest technique is to double-click on the button of interest in the form editor. This will automatically take you to the event handler routine for that button's click event (we'll explain these terms shortly).

- 1) Double-click on the **Cancel** button in the form editor and observe the resulting dialog as illustrated in Figure 4-13.

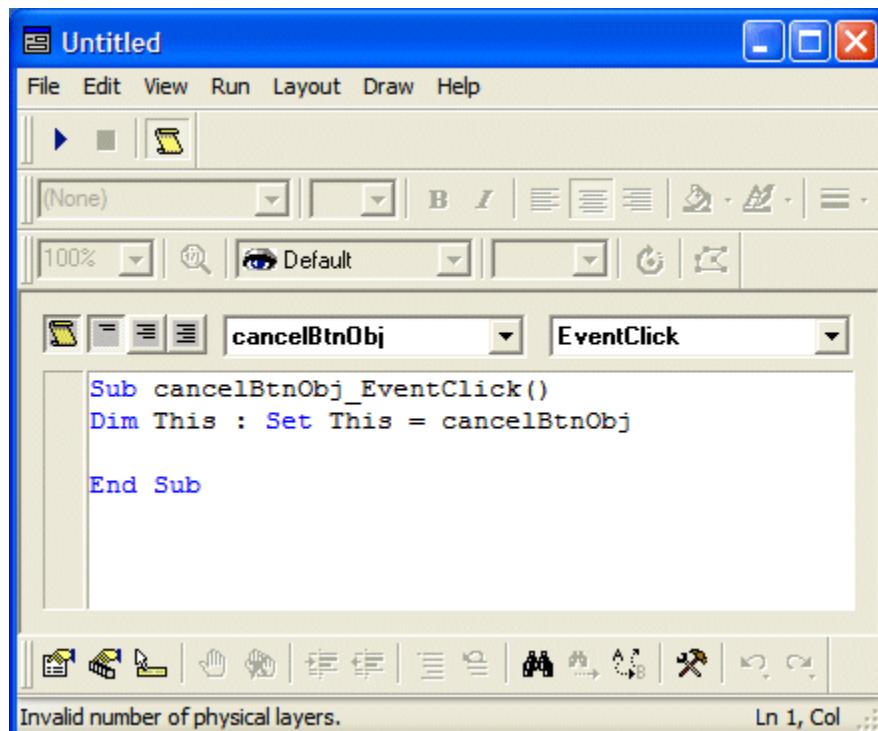


Figure 4-13. The event handler for the Cancel button's click event.

In particular, observe the drop-down box located just above the main edit window to the left-hand side. This lists the objects available to this form. By default, the name you previously entered in the attribute field for the **(Object Code)** item in the **Property Sheet** dialog associated with this button is already selected for you (*cancelBtnObj*, in this case).

Now observe the drop-down box located just above the main edit window to the right-hand side. This lists the event handlers available for use with this object. By default, the **EventClick** event handler is selected; this is the one that will service the user executing a single-click on the **Cancel** button. There are a variety of such 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 is executed when the control is initially created at the time the form is loaded)
etc.

- 2) Now, this is a tricky bit, so sit up and pay attention. From the perspective of the form as a whole, the attribute associated with the **(Object Code)** item is known as *TheView*. Place your cursor in the edit window on the blank line between the *Dim* and *End Sub*

statements. Start entering the following text: *TheView*. As soon as you key in the period following *TheView*, a pop-up window appears listing all of the properties and methods associated with this form (Figure 4-14).

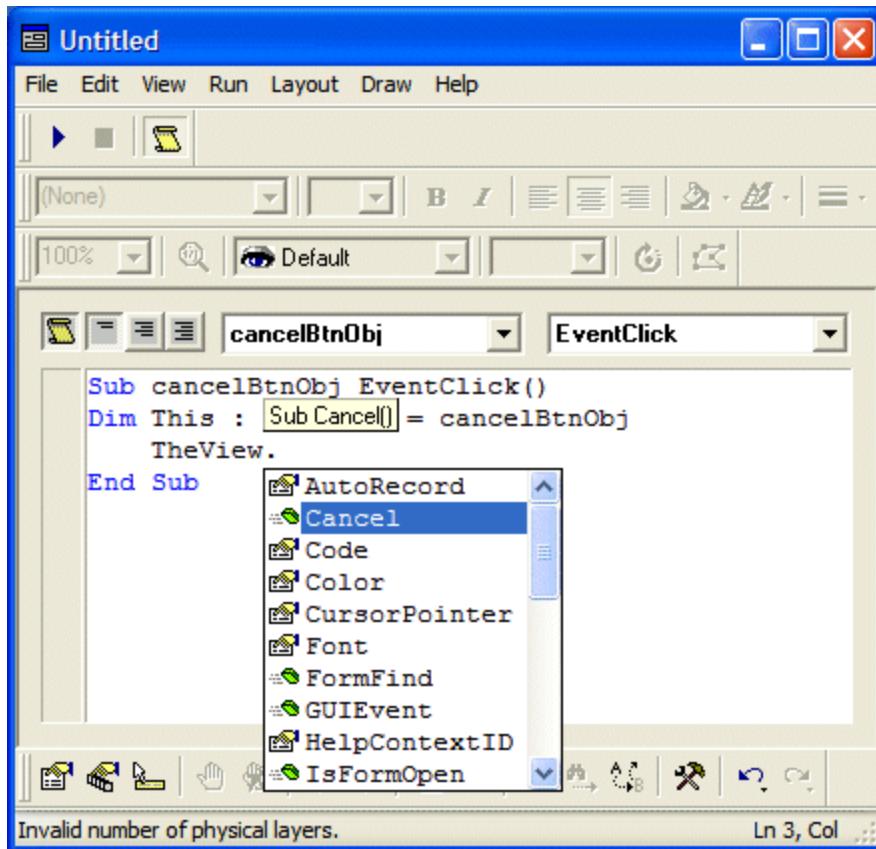


Figure 4-14. The IDE has an automated statement-completion feature.

- 3) Select the **Cancel** item and press the <Enter> key to confirm this selection. This will automatically complete the statement in the editor window (Figure 4-15).

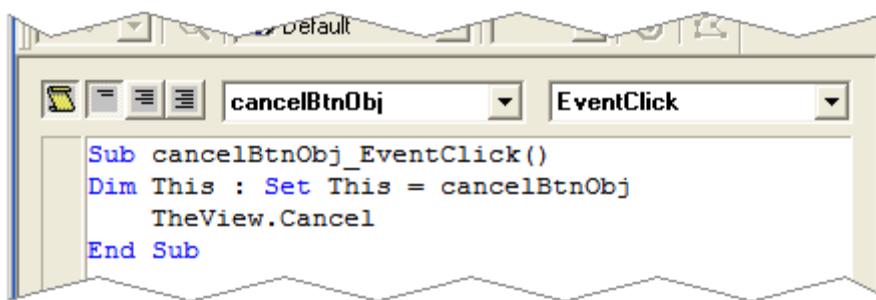


Figure 4-15. The completed statement.

- 4) Use the **File > Save As** command to save the form as *ViaCountForm.efm* (the *.efm extension denotes a form, as compared to the *.vbs extension we used in the previous chapter to denote a pure text script).
- 5) Click the **Run Form** icon to set the form running (the right-pointing arrow that looks like the "Play" button on a VCR or DVD player). The result will be as shown in Figure 4-16.

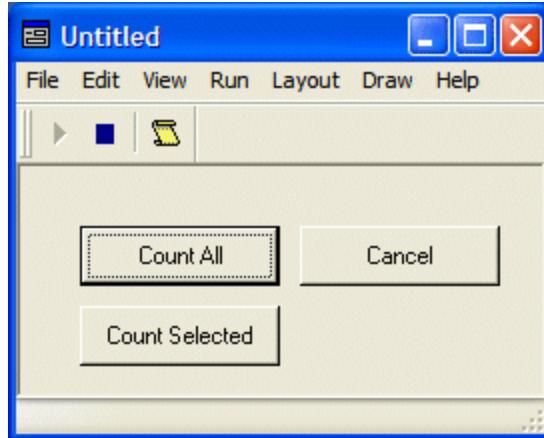


Figure 4-16. The form is running.



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

- 6) Click the **Count All** and the **Count Selected** buttons a couple of times. Nothing happens because you haven't put the code "behind" these buttons. Now click the **Cancel** button to dismiss the form.

Adding Code "Behind" the Count All Button

- 1) Your last action at the end of the previous topic was to click the **Cancel** button to dismiss the form. As you will observe, this button performed its task magnificently because the form was terminated and has completely disappeared (this is why regularly using the **File > Save** or **File > Save As** commands is such a good idea).

If you haven't already done so, invoke Expedition PCB and open up a circuit board design containing some number of traces and vias. Now use the **File > Open Script Form** command in Expedition PCB to locate and open the *ViaCountForm.efm* file you created in the previous topic.

- 2) By default, the IDE opens in its *run mode*. Click the **Stop** icon (the square that looks like the "Stop" button on a VCR or DVD player) to return the IDE to its *edit mode*.



Note: If you want to open the IDE directly in *edit mode*, press and hold the **<Ctrl>** key when clicking the **OK** button on the **File > Open Script Form** dialog.

- 3) Double-click on the **Count All** button in the form editor to access the event handler for the **Count All** button's click event as illustrated in Figure 4-17.

As before, observe the drop-down box located just above the main edit window to the left-hand side. This lists the objects available to this form. By default, the name you previously entered in the attribute field for the (Object Code) item in the Property Sheet dialog associated with this button is already selected for you (*countAllBtnObj*, in this case).

Also observe the drop-down box located just above the main edit window to the right-hand side. This lists the event handlers available for use with this object. By default, the **EventClick** event handler is selected; this is the one that will service the user executing a single-click on the **Count All** button.

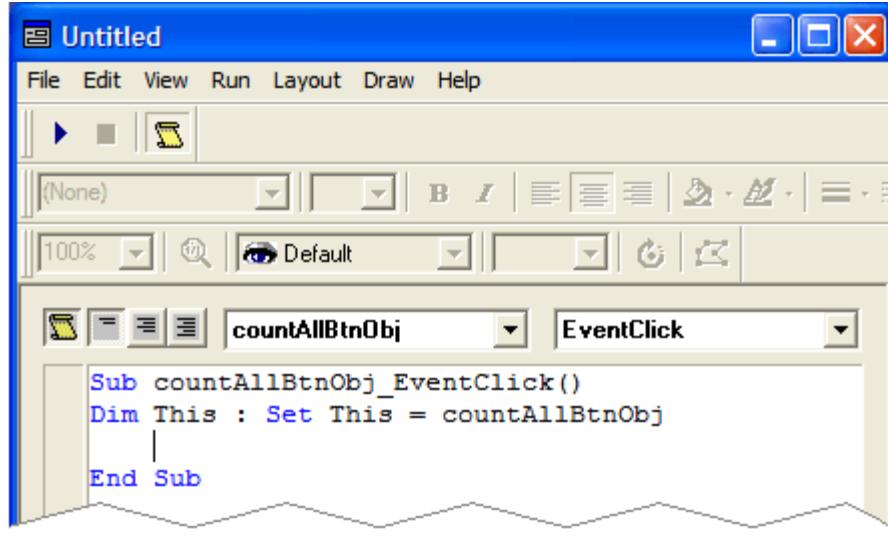


Figure 4-17. The event handler for the Count All button's click event.

- Now, use the text or script editor of your choice to open up the *ViaCount.vbs* file you created during the previous chapter. At the moment, we are interested only in the code to be associated with the **Count All** button. 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("MGCPBC.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)

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

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



---


"Local functions"

```

- Paste this code between the *Dim* and *End Sub* statements in the IDE editor as illustrated in Figure 4-18.

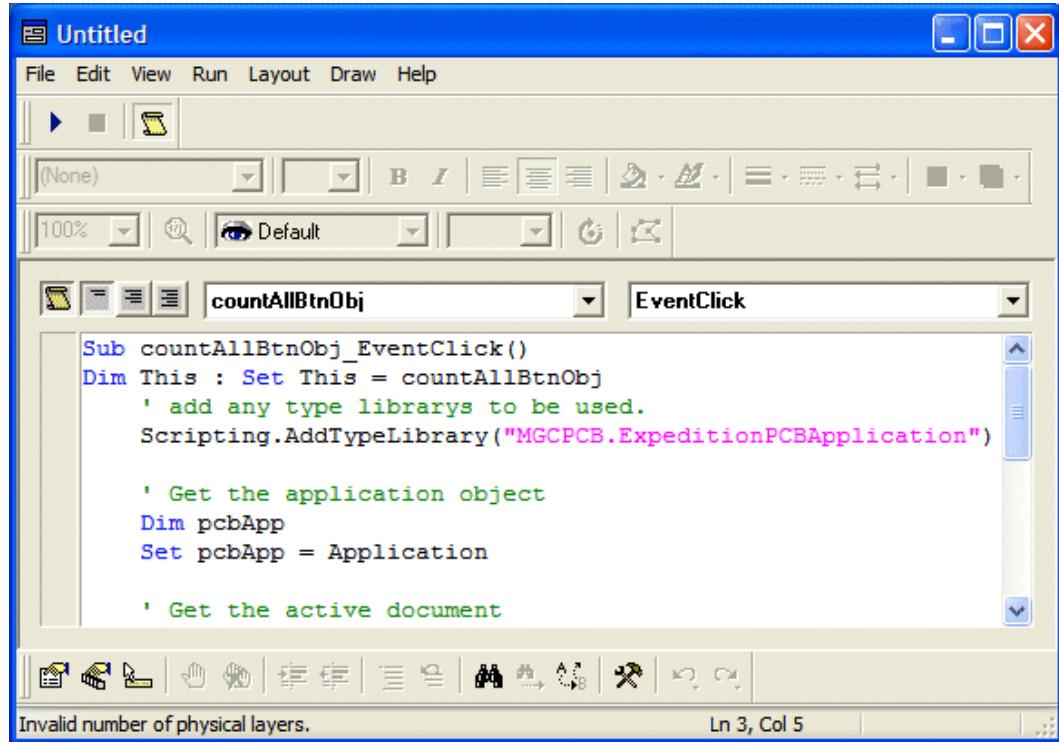


Figure 4-18. The event handler for the Count All button's click event.

- 6) Next, you need to include the server validation function in your form. Return to the *ViaCount.vbs* file you created during the previous chapter and select (and copy) the contents of the function (including the *Function* statement at the top and the *End Function* statement at the bottom) as shown below:

```

' Server validation function
Function ValidateServer(doc)

    Dim key, licenseServer, licenseToken

    ' Ask Expedition's document for the key
    key = doc.Validate(0)

    ' Get license server
    Set licenseServer =
        CreateObject( "MGCPBCAutomationLicensing.Application" )

    ' Ask the license server for the license token
    licenseToken = licenseServer.GetToken(key)

    ' Release license server
    Set licenseServer = nothing

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

    ' Ask the document to validate the license token
    doc.Validate(licenseToken)
    If Err Then

```

```

        ValidateServer = 0
    Else
        ValidateServer = 1
    End If

End Function

```

- 7) If you wanted, you could simply include this code as part of the **Count All** button's single click event handler, but this would mean that you would have to include the same validation code in the **Count Selected** button's event handler.

As an alternative, click the down-arrow icon associated with the drop-down box that lists the objects available to this form as illustrated in Figure 4-19.

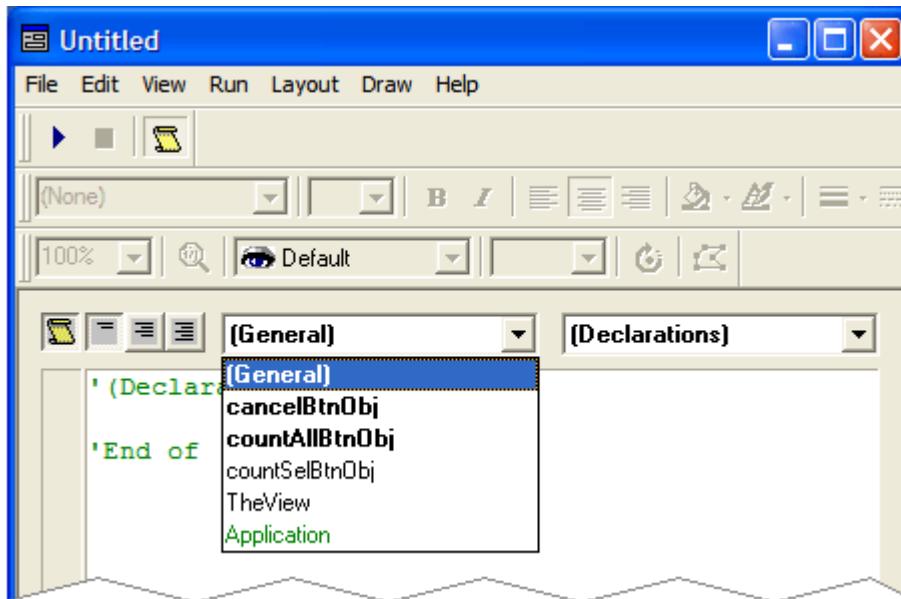


Figure 4-19. Selecting the **(General)** item.

- 8) Select the **(General)** item. Observe the *(Declarations)* and *End of (Declarations)* comments that appear in the main editing window as illustrated in Figure 4-20. These are the same comments you saw when you were creating your pure text script earlier in this chapter (you may want to refer back to Figure 4-2 for a moment).



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

Any code in this section (with the exception of *functions* and *subroutines* as discussed below) will be automatically executed a single time on start-up when the form is first launched.

Functions and subroutines in the **(General)** section are not executed until they are called. The reason for defining any functions and subroutines here is that this makes them global such that they can be called from anywhere in the current form.

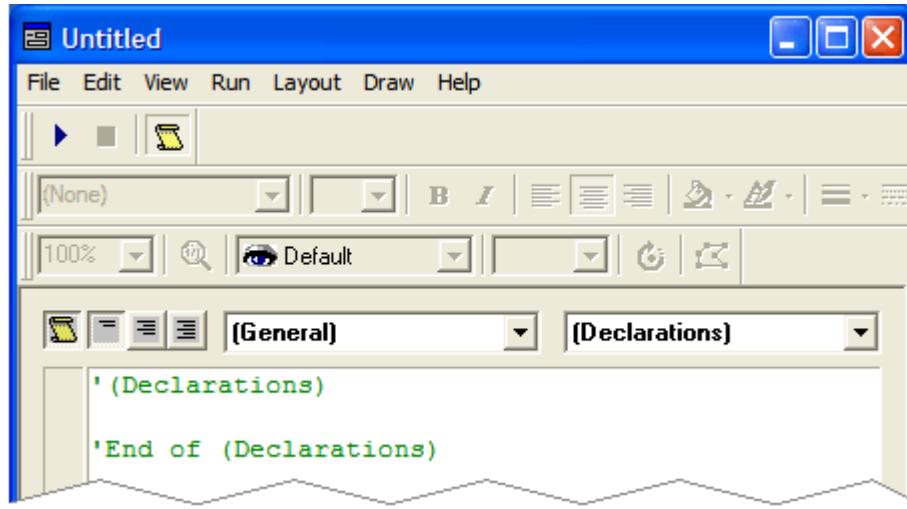


Figure 4-20. The comments associated with the (General) section.

- 9) Place your cursor between the *(Declarations)* and *End of (Declarations)* comments, key-in the *Option Explicit* command (this was introduced in Chapter 2) and then paste in the server validation code (Figure 4-21).

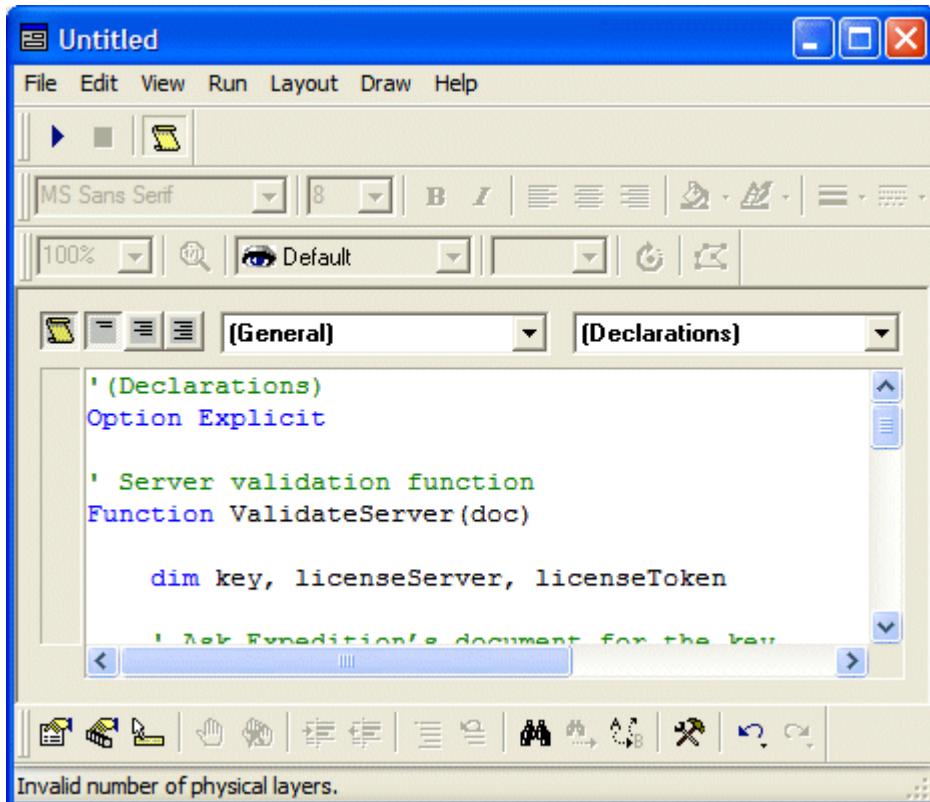


Figure 4-21. Including the server validation code in the (General) section.

- 10) Now, observe the three icon buttons located just above the editing area on the left-hand side (Figure 4-22). The leftmost of these three icons is known as the **Event View**. While in this mode, the edit window will display only the code associated with the combination of the object selected in the left-hand drop-down list and the event type

selected in the right-hand drop-down list. If you look back at the illustrations above, you will see that this is the mode in which you have been working.

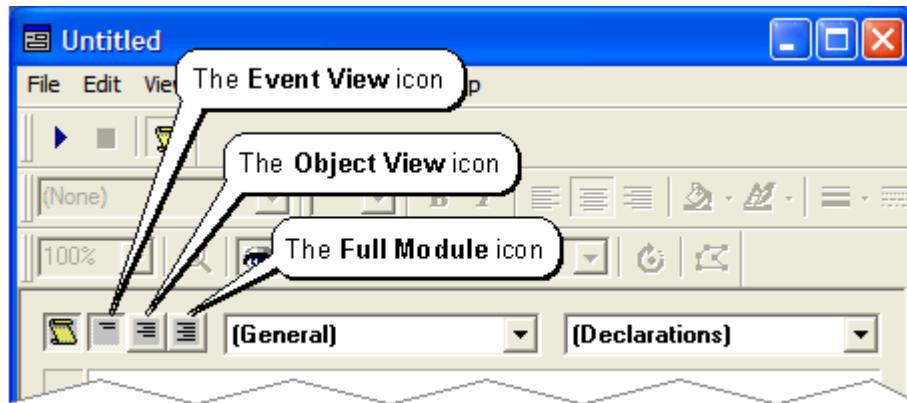


Figure 4-22. The Event View, Object View, and Full Module icons.

The middle icon is known as the **Object View**. If this is selected, the edit window will display all of the event handlers – for which you have actually defined code – associated with the selected object. To put this another way, if you define code for both single-click and double-click event handlers for an object, for example, and if this object were selected in the left-hand drop-down list, then clicking the **Object View** icon would cause the code for both of these event handlers to appear in the main edit window.

The right-hand icon is known as the **Full Module** view. If this is selected, the edit window will display all of the code associated with this form. Click the Full Module icon now and observe the results as illustrated in Figure 4-23.

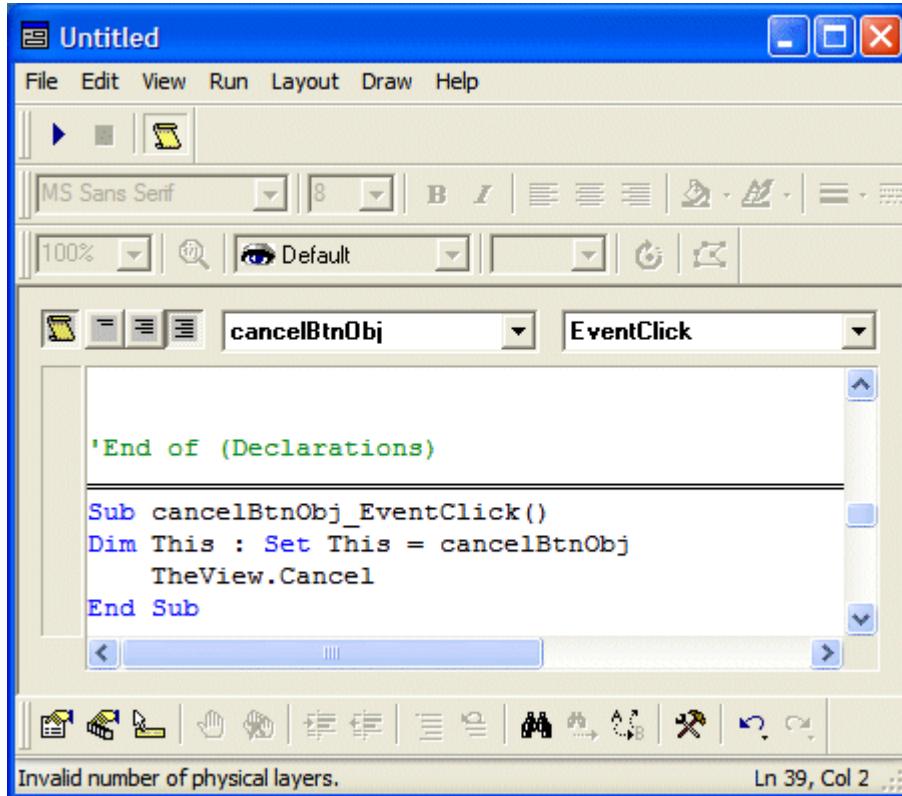


Figure 4-23. The Full Module view.

If you scroll to the top of the edit area you will find yourself in the code for the (**General**) section. As you scroll down, you will see the code for the **Cancel** and **Count All** objects. Observe that, 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 left-hand drop-down list automatically updates to display the name of the object whose code you are currently viewing.



Note: Even when in the **Full Module** view, the edit window will only display the code for any event handlers you have actually defined. This is why you will see only the single-click event handlers for the **Cancel** and **Count All** buttons, because you haven't defined any other event handlers (such as a double-click handler) for these buttons and you have not yet defined even a single-click event handler for your **Count Selected** button.

- 11) Click the **Event View** icon to return the editor to only displaying the code associated with a single event handler.
- 12) Use the **File > Save** command to save the current version of your *ViaCountForm.efm* file, and then click the **Play Form** icon.
- 13) Click the **Count Selected** button. As usual, nothing happens because you have not yet added the code "behind" this button.
- 14) Now click the **Count All** button. The script associated with this button is executed and the *MsgBox()* appears on the screen (Figure 4-24).

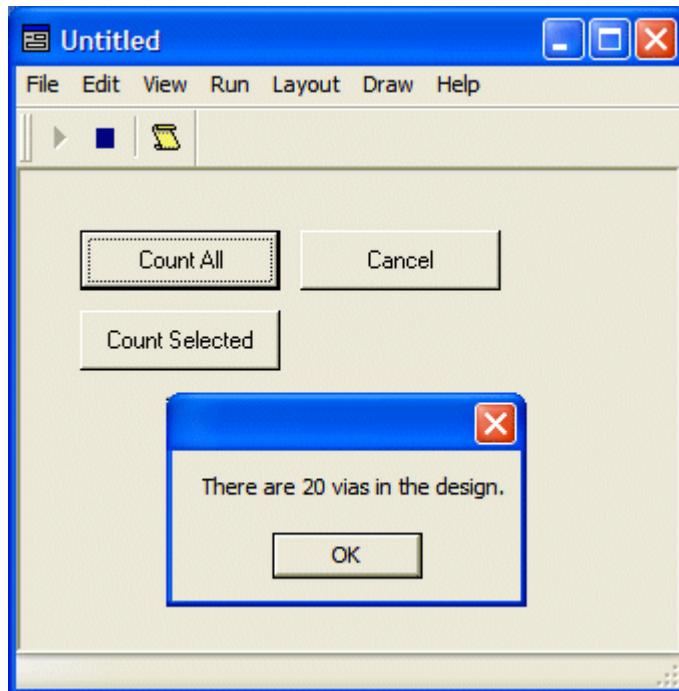


Figure 4-24. The Count All button now works – hurray!

- 15) Click the **OK** button to dismiss the *MsgBox()*.
- 16) Click the **Stop** icon to return the IDE to its *form editor* mode.

Adding Code "Behind" the Count Selected Button

- 1) Double-click the **Count Selected** button in the form editor to access the event handler for the **Count Selected** button's click event. Observe that *countSelBtnObj* appears in the left-hand drop-down list.
- 2) Use the left-hand drop-down list to select the *countAllBtnObj* (this is the **Count All** button).
- 3) Select the code between the *Dim* and *End Sub* statements.
- 4) Use the left-hand drop-down list to return to the *countSelBtnObj*.
- 5) Place the cursor between the *Dim* and *End Sub* statements and then paste a copy of the code from the **Count All** button into this area.
- 6) Edit this code as shown below. The first edit is to add the enumerate *epcbSelectSelected* as the first parameter to the *pcbDoc.Vias* method. This will cause the *viasColl* object to end up containing only the selected vias. The second edit is to modify the message associated with the *MsgBox()* to reflect the fact that this script is counting only *selected* vias.

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

' Get the application object
Dim pcbApp
Set pcbApp = Application
'Set pcbApp = CreateObject("MGCPCB.ExpeditionPCBAApplication")

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

' License the document
ValidateServer(pcbDoc)

Dim viasColl : Set viasColl = pcbDoc.Vias
Dim viasColl : Set viasColl = pcbDoc.Vias(epcbSelectSelected)
Dim viaCntInt : viaCntInt = viasColl.Count

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

- 7) Use the **File > Save** command to save the current version of your *ViaCountForm.efm* file, and then click the **Play Form** icon.
- 8) Use your mouse to select a subset of the vias in the design.
- 9) Click the **Count All** button. The script associated with this button is executed and the *MsgBox()* appears on the screen displaying the total number of vias.
- 10) Click the **OK** button to dismiss the *MsgBox()*.
- 11) Now click the **Count Selected** button. The script associated with this button is executed and the *MsgBox()* appears on the screen displaying the number of selected vias.
- 12) Click the **OK** button to dismiss the *MsgBox()*.

-
- 13) Use your mouse to select *all* of the vias in the design.
 - 14) Click the **Count Selected** button again. Check that the number of selected vias as displayed in the *MsgBox()* is the same as the total number of vias reported when you previously used the **Count All** button.
 - 15) Click the **OK** button to dismiss the *MsgBox()*.
 - 16) Click the **Stop** icon to return the IDE to its *form editor* mode.

Removing Duplicated Code

The way in which you were instructed to populate the code "behind" the **Count All** and **Count Selected** buttons was designed to introduce the IDE in as understandable way as possible.

However, this resulted in some code being duplicated. This is not recommended, because it increases the effort involved in maintaining the code in the future. As a general rule of thumb, any code that is common to multiple functions should be declared a single time as part of the **(General)** section.



Note: All variables dimensioned in the **(General)** section can be accessed from any event handler; similarly, all functions and subroutines defined in the **(General)** section can be called from any event handler.



Note: The act of adding a type library should never be performed in a function, because this is a global statement that applies to the entire form, so type libraries should always be added in the **(General)** section.

- 1) Click the **Full Module** icon to see all of the code associated with this form.
- 2) Scroll through the code and compare the single-click event handlers for the **Count All** and **Count Selected** buttons. In fact, all of the code – except for the final three lines – is common to both event handlers. For example, consider the code associated with the **Count All** event handler as shown below (the code shown in **bold** font is common).

```
' Add any type libraries to be used.  
Scripting.AddTypeLibrary("MGCPCB.ExpeditionPCBAApplication")  
  
' 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 the common code from the **Count All** event handler and use the **Edit > Cut** command to cut it out and copy it into the paste buffer.
- 4) Scroll up to the code associated with the **(General)** section, place your mouse cursor on the blank line below the *Option Explicit* statement, and use the **Edit > Paste** command to paste the common code as illustrated in Figure 4-25.

```

' (Declarations)
Option Explicit

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

' Get the application object
Dim pcbApp

```

Figure 4-25. Gathering common code in the (General) section.

- 5) Scroll back down to the code associated with the **Count Selected** object; select the same common code as before, and use the **Edit > Cut** command to simply cut it out and discard it.
- 6) Use the **File > Save** command to save the current version of your *ViaCountForm.efm* file, and then click the **Play Form** icon and check that the **Count All** and **Count Selected** buttons still function as required (don't click the **Cancel** button, because there remain some edits you still have to make to the form).



Note: Thus far, all of our scripts have assumed that a PCB Layout document has already been opened. What happens if you attempt to run the script without having first opened a layout document? One solution would be to augment your scripts with a test to see if a layout document is already open and – if not – prompts the user to open such a document.

Assigning a Title

If you look at the blue title bar in the previous illustrations, you will observe that the names of these forms is "Untitled" (this is true in both the *edit mode* and the *run mode* as illustrated in Figures 4-23 and 4-24, respectively). Thus, before a form/dialog is placed in production, it has to be assigned a title. In order to do this, perform the following steps:

- 1) Click the **Stop** icon to return the IDE to its edit mode.

- 2) Right-click anywhere in the edit except over a button.
- 3) On the resulting pop-up menu select the **Form Properties** item. This will open up a **Property Sheet** dialog similar to the dialogs we used on the individual buttons. In this case, however, the dialog is associated with **TheFrame/TheView** (Figure 4-26)

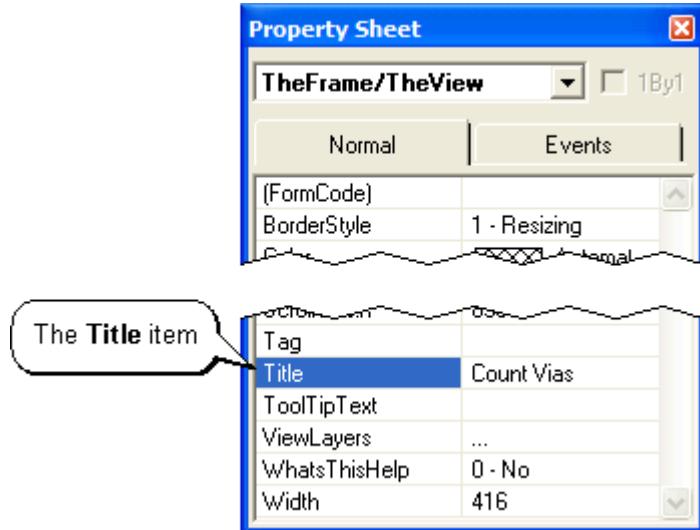


Figure 4-26. The Property Sheet dialog.

- 4) Change the attribute associated with the **Title** Item to “Count Vias”.
- 5) Once you've made these changes, either press the <Esc> key or click the small red 'X' button on the right-hand side of the title bar to dismiss this form (as we previously noted, although this may seem a little strange, there's no need to formally apply these changes because they are automatically applied as you are keying them).
- 6) Observe that the title bar in the editor is automatically updated to reflect this new value (this title will also be displayed when the form is running).

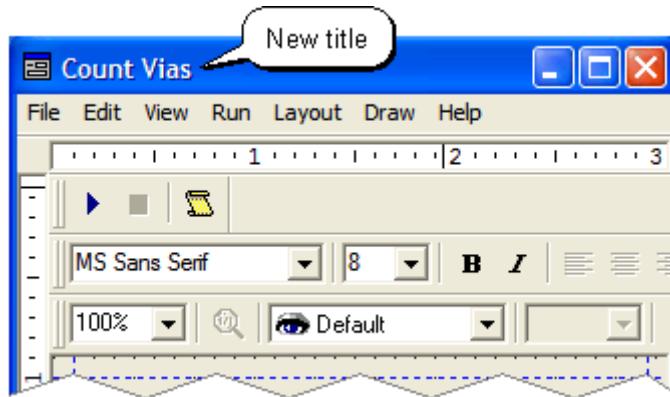


Figure 4-27. Changing the text in the form's title bar.

Going into Production

- 1) Click the **Run Form** icon to enter the IDE's *run mode*.
- 2) Not surprisingly, even when the IDE is in its run mode, the form still displays the **Stop** icon that will return the IDE into its edit mode as illustrated in Figure 4-28.

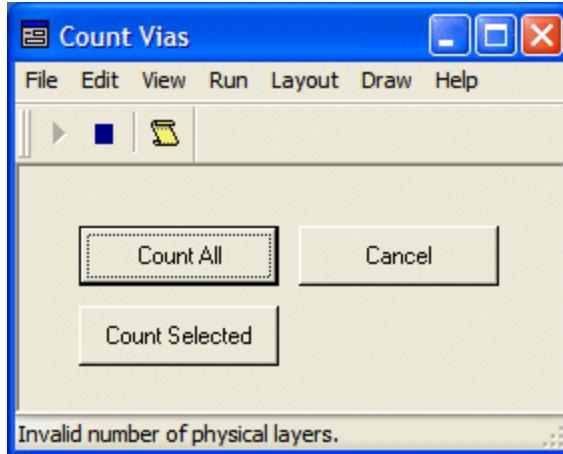


Figure 4-28. By default, the form presents users with the Stop button.

- 3) Of course, you need to be able to access the **Stop** icon while you are developing your form, but you don't want users to have the ability to access and modify the form's code, so click the **Cancel** button to dismiss the form.
- 4) Use your computer's browser to locate the *ViaCountForm.efm* file, right-mouse click on the file, select the **Properties** item from the resulting pop-up window, and set the attributes of this file to **Read-Only**.
- 17) Use the **File > Open Script Form** command in Expedition PCB to locate and open the *ViaCountForm.efm* file you created in the previous topic (Figure 4-29).

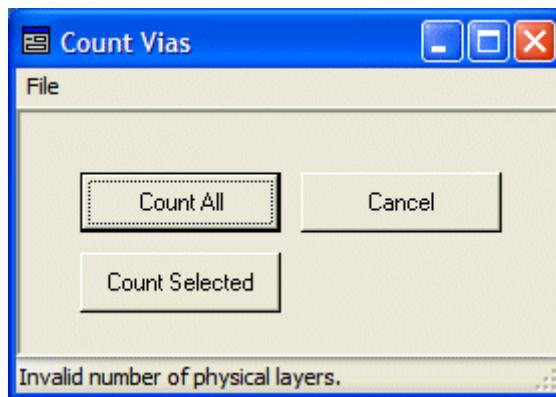


Figure 4-29. With file properties set to Read-Only there is no Stop button.

- 5) As usual, the IDE opens in its *run mode*. Observe that the **Stop** icon is no longer present.
- 6) Click the **Cancel** button to dismiss the form.

Chapter 5: Associating an Accelerator (Shortcut) Key with a Script

Introduction

The Key Bind Server is a separate server that's used by applications such as Expedition PCB and FabLink XE to establish and execute key bindings; that is, the Key Bind Server is used to configure accelerator (shortcut) keys.

The Key Bind Server is accessed through a scripting or programming language. For example, you can set things up such that when the user presses <Ctrl>+N (that's pressing the 'N' key while pressing and holding the <Ctrl> key), some script will be run.

This chapter presents an example of using the Key Bind Server.

Associating an Accelerator (Shortcut) Key with a Script

An accelerator (shortcut) key can be bound to one of two things:

- **A menu item:** Pressing the appropriately bound accelerator key launches a script that invokes and runs an existing menu command; this scenario is not discussed further in this chapter.
- **Any valid key-in command:** Pressing the appropriately bound 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 chapter. For example, in *Chapter 3: Running Your First Script*, you created a *ViaCount.vbs* script that you stored in a folder called *C:/Temp*. At that time, you ran 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, then rather than continually keying it in, it would be preferable to associate it with an accelerator (shortcut) key such as <Ctrl>+N.

Creating the Script

Use the scripting 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 is used to indicate a script that is associated with an accelerator key).

On Line 1 we see the *Option Explicit* statement (the use of this statement is discussed in *Chapter 2: VBS Primer and Appendix A: Good Scripting Practices*).

```
1 Option Explicit
```

On Line 4 we use 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 *Chapter 11*). Any time you use the Key Bind Server – which is returned from the *Bindings* property as discussed below – you will want to add this type library.

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

On Lines 7 and 8 we access the *Application* (this was introduced in *Chapter 3: Running your First Script*). Observe that this script doesn't need to access a *Document*. Also observe that there's no need to perform any validation at this stage, 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
```

From the Application, on Lines 11 and 12 we access the *Gui* object by calling the *Gui* property. As opposed to storing the *Gui* object in a variable (which we could certainly do), we use the very common practice to use it immediately to obtain the *Bindings* object (this is known as the *BindingsTables* object in the help file).

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

Observe that the string "Document" is specified as a parameter at the right-hand side of the statement on Line 12; this means that we want to access and use the table associated with having a *Document* open. In fact, there are three such options 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, in Lines 15 and 16, we use the *AddKeyBinding* method associated with 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 is continued 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)
```

Also observe that there are four parameters associated with the *AddKeyBinding* method as follows:

- **Parameter 1:** This is the accelerator key (or key combination) to be used. In this example we've assigned the string "Ctrl+N", which equates to pressing the 'N' key while pressing and holding the <Ctrl> key. Had we required, however, we could have simply assigned the string "N", which would cause our script to be executed whenever the 'N' key was pressed on its own.
- **Parameter 2:** This is the command to be run; it can be any valid key-in command. In this example, we've chosen to use the same key-in command we employed in *Chapter 3* to launch our *ViaCount.vbs* script.
- **Parameter 3:** The *BindCommand* attribute indicates that we are working with a key-in command as opposed to a menu entry (see also *Chapter 6: Adding a Menu and/or Menu Button/Item*).
- **Parameter 4:** Last but not least, *BindAccelerator* is a required attribute that currently has no impact.

Running the Script

For the purposes of this example, you will run your script from within Expedition PCB (this technique is covered in greater detail in *Chapter 9: Running a Script From Within an Application*).

- 1) Launch Expedition PCB.
- 2) At this stage you may open a layout design of your choice, however, you are not obliged to do so – see also point (6).
- 3) Right-click on the toolbar and ensure that the **Keyin Command** option is ticked (enabled).
- 4) Enter the string `run C:\Temp\ViaCountAcc.vbs` into the **Keyin Command** field (Figure 5-1) and press the <Enter> key to execute this script.

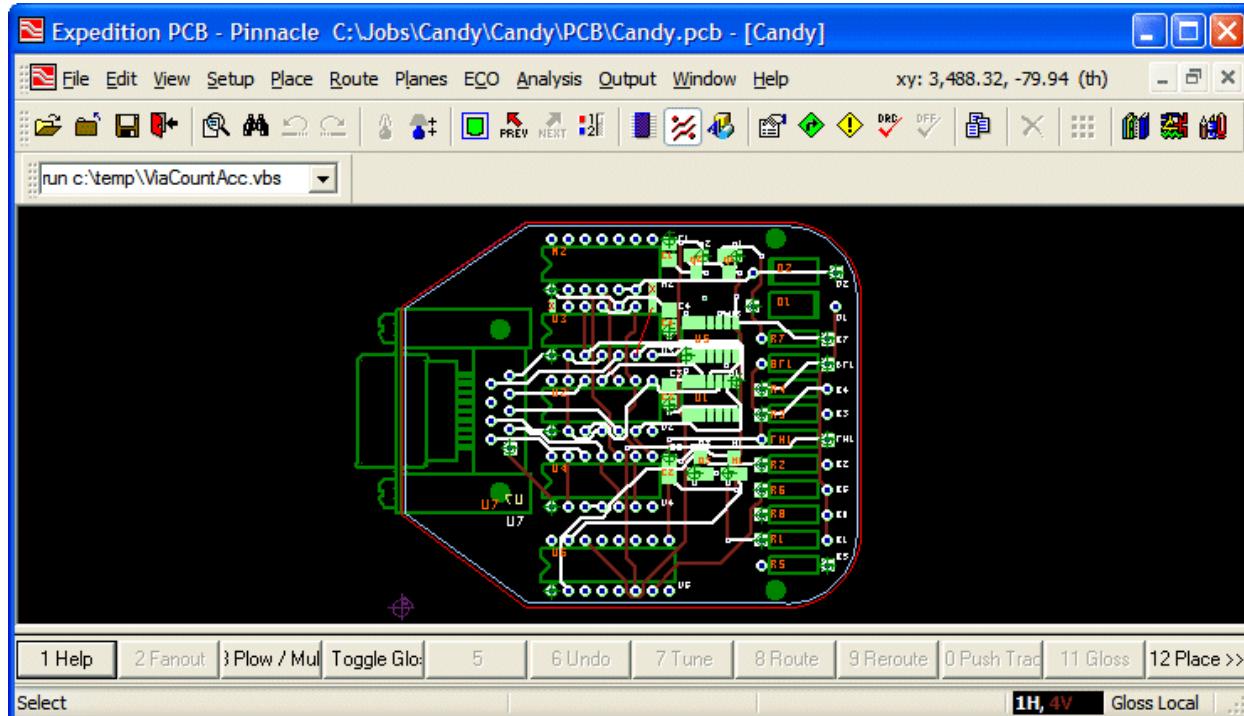


Figure 5-1. Running the *ViaCountAcc.vbs* script.

- 5) Observe that after you've run this script it exits and nothing seems to have happened. In reality, however, the script has established a binding for the <Ctrl>+N key combination.
- 6) Now we want to test the accelerator key to check that it does indeed run our *ViaCount.vbs* script. If you had not already opened a layout design document in point (2), then this would be the time to do so.
- 7) Press the <Ctrl>+N key combination and observe that your accelerator key does indeed work as illustrated in Figure 5-2.



Note: Your new <Ctrl>+N binding will remain in existence for the duration of this session.



Note: You may be wondering as to the point of all this. After all, rather than first having to run the *ViaCountAcc.vbs* script and then pressing the <Ctrl>+N key combination, wouldn't it be easier to simply run our original *ViaCount.vbs* script? Well, the answer to this is two fold: first, it's possible to establish lots of key accelerator (shortcut) bindings in a single script; second, this type of script would typically be set to run automatically at startup (see also *Chapter 10: Running Startup Scripts*).

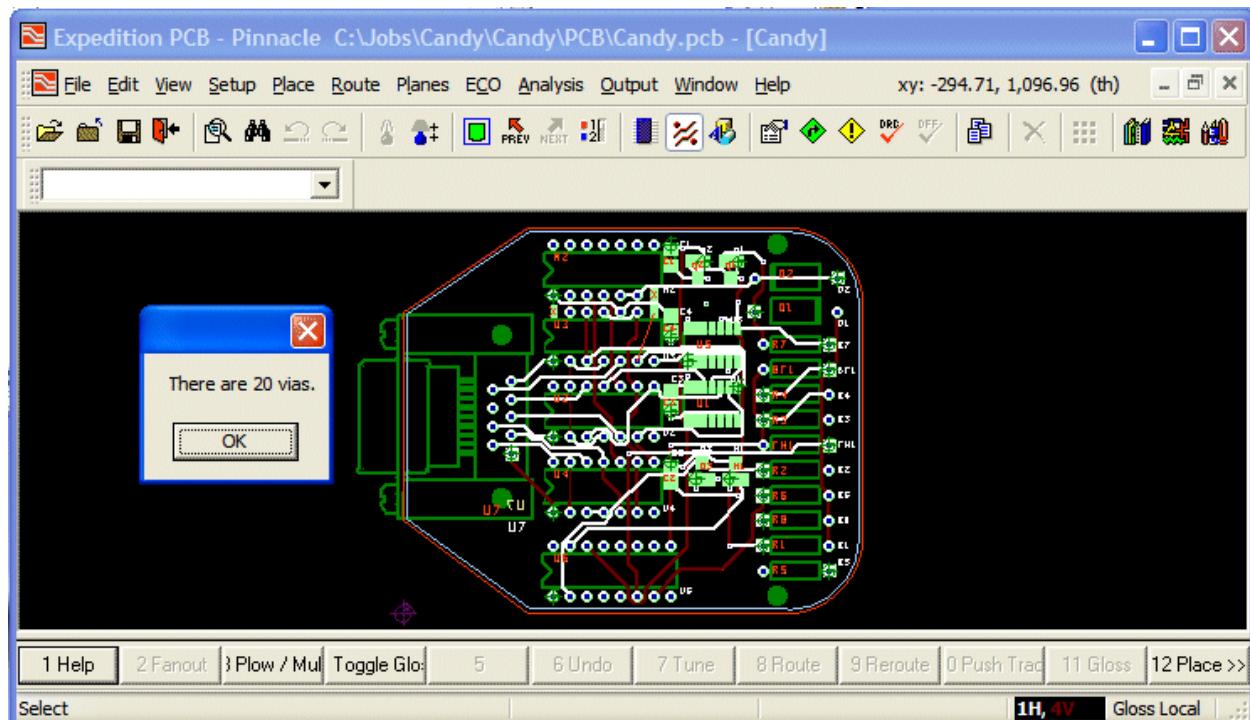


Figure 5-2. Using the <Ctrl>+N accelerator to run the ViaCount.vbs script.

Chapter 6: Adding a Menu and/or Menu Button/Item

Introduction

The Command Bar Server is a separate server that's used by applications such as Expedition PCB and FabLink XE to configure pull-down menus on the menu bar (it can also be used to configure the icons/actions associated with the various tool bars as discussed in *Chapter 7: Adding a Tool Bar Icon*). For example, you can add a button/item to an existing pull-down menu on the menu bar and you can then associate this button with a script. Alternatively, you can add a completely new (top-level) pull-down menu to the menu bar, add a button/item to this new pull-down menu, and then associate *this* button with a script.

The Command Bar Server is accessed through a scripting or programming language. This chapter presents three examples of using the Command Bar Server. The first example involves you adding a new button/item to an existing pull-down menu and then associating this button/item with a small script that calls the *ViaCount.vbs* script you created in *Chapter 3: Running Your First Script*. In the second example, you will modify the script associated with your new button/item such that it performs the counting of the vias itself. The third example involves you creating a completely new pull-down menu.

Tying an Existing Script to a New Button/Item on an Existing Pull-Down Menu

In this example, you are going to create a script that first adds a new button/item to the bottom of the **View** pull-down menu and then ties this button to the *ViaCount.vbs* script you created in *Chapter 3: Running Your First Script*.

Creating the Script

Use the scripting 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 is used to indicate a script that is associated with a menu – the "Dual" portion of the filename is required only to distinguish this script from our next example).

On Line 1 we see the *Option Explicit* statement (the use of this statement is discussed in *Chapter 2: VBS Primer and Appendix A: Good Scripting Practices*).

```
1 Option Explicit
```

On Line 4 we use 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 *Chapter 11*). Any time you use the Command Bar Server – which is returned from the *Bindings* property as discussed below – you will want to add this type library.

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

On Lines 7 and 8 we access the *Application* (this was introduced in *Chapter 3: Running your First Script*). Observe that this script doesn't need to access a *Document*. Also observe that there's no need to perform any validation at this stage, 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
```

From the Application, on Lines 11 and 12 we access the *Gui* object by calling the *Gui* property. As opposed to storing the *Gui* object in a variable (which we could certainly do), we use the very common practice to use it immediately to obtain the *CommandBars* object.

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

Observe that the string parameter at the right-hand side of the statement on Line 12. This parameter is used to specify whether you want to access the menu bar or the tool bar. The string "Document Menu Bar" is used to specify the menu bar, and this returns the appropriate *CommandBars* object.

In order to manipulate (and add items to) this *CommandBars* object, we need to access a list of controls that equate to the various drop-down menus (each drop-down menu is a single control). Thus, on Line 16 we instantiate a collection called *docMenuBarCtrlColl* and on Line 17 we associate a list of menu bar controls with this collection.

```
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, we want to locate the **View** drop-down menu in our collection. On Line 20 we instantiate a variable called *viewMenuObj* and on line 21 we use the *Item* property on our collection of controls to access a specific control.

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

As its parameter, the *Item* property can accept an index (in the form of an integer or integer variable) or a string or string variable. In this particular example we've used the string "&View" as the parameter to locate the **View** drop-down menu. Observe that the '&' character is required before whichever character appears underlined in the menu bar as illustrated in Figure 6-1 and 6-2. (Any underlined characters can be used as shortcuts to access the various menus; for example, under the Windows® operating system, you can use "<Alt>+V" in Expedition PCB to access the contents of the **View** menu.)

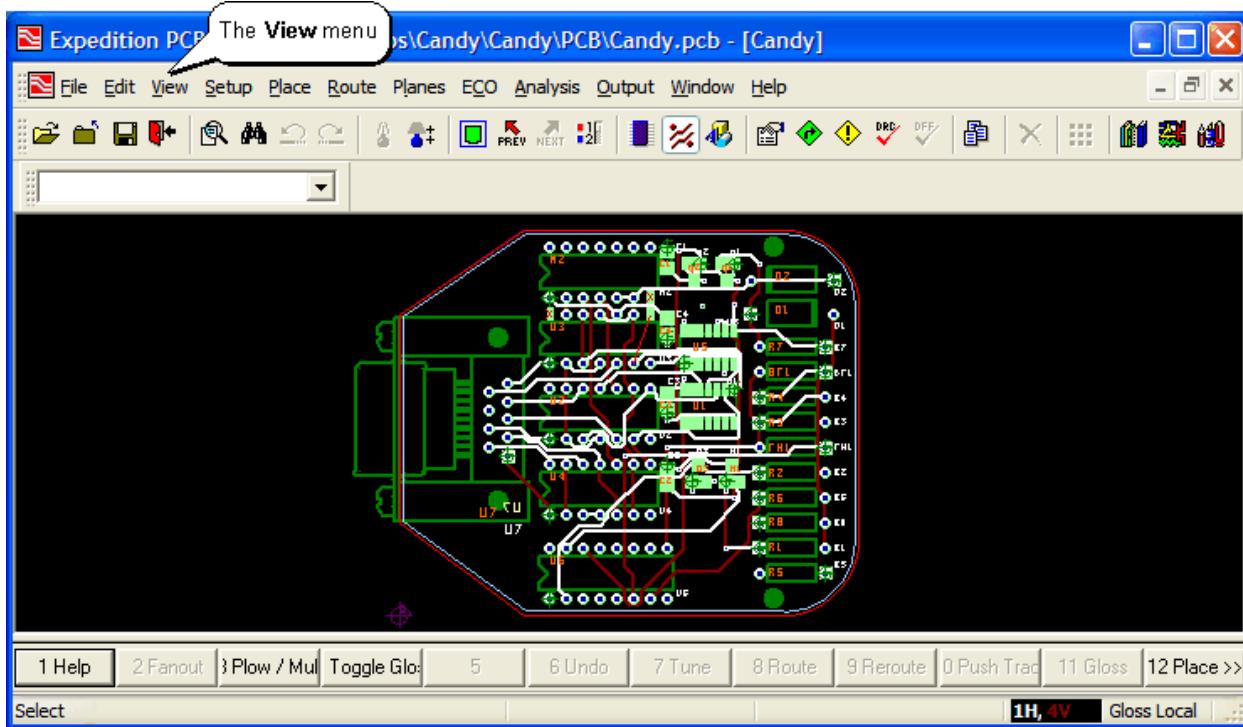


Figure 6-1. The View menu item.

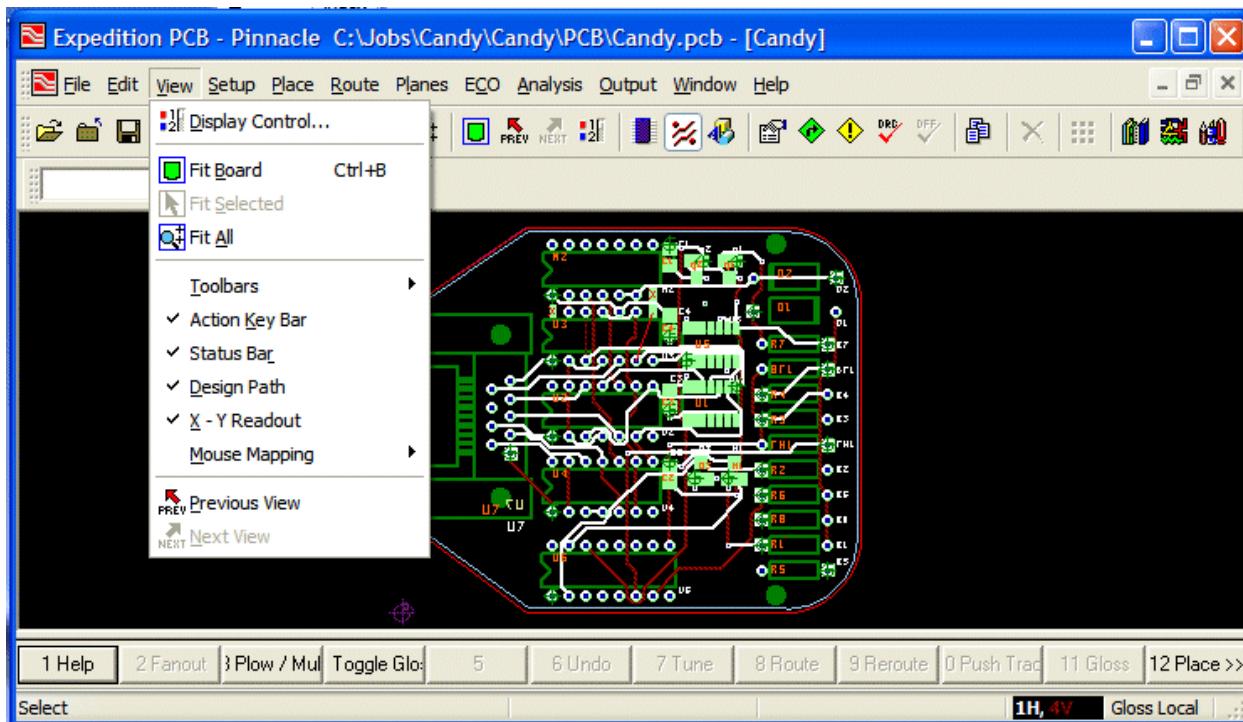


Figure 6-2. The contents of the View pull-down menu.

Observe that the last button/item in this menu (the one at the bottom) is currently **Next View**. We want to add a new **Via Count** button/item underneath this existing button/item.

Previously, we accessed a list of the main menu controls (**File**, **Edit**, **View**, **Setup**, etc.). Now we want to access the collection of sub-controls associated with the main **View** control. Thus, on Line 24 we instantiate a collection called `viewControlsColl` and on Line 25 we associate this collection with a list of sub-controls.

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

We now want to add a new button/item to this collection. In Line 28 we instantiate a variable called `viaCntBtnObj` that we will use to represent our new button/item and on Line 29 we use the *Add* method to add this button/item to the list of **View** sub-controls. Observe that there are four parameters associated with the *Add* method as follows:

- **Parameter 1:** This parameter specifies what we're adding; the options are:
 - 0 cmdControlButton ' A simple button
 - 1 cmdControlPopup ' A sub-pop-up menu
 - 2 cmdControlButtonSeparator ' A horizontal line
- **Parameter 2:** This is an optional command ID (it is not generally used).
- **Parameter 3:** This is reserved for future use.
- **Parameter 4:** The position before which you would like your new button/item to appear in the collection (a value of -1 equates to the end of the existing collection of button/items).

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

Once our new button/item has been added to the list of **View** sub-controls, we need to configure it. On Line 32 we use the string "&Via Count" to define the caption that will appear in the drop-down menu; once again, the '&' character is used to inform the system that we want the following character ('V' in this case) to be underlined and to act as a shortcut key). On Line 33 we specify the description text to be associated with our button/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"
```

On Lines 34 and 35 we define the function that is to be called when our new button/menu item is selected. First, on Line 34, we specify the "Target," which we can visualize as being the location where this function "lives." In this example we use `ScriptEngine`, which is a special implicit object that basically equates to: *"The same engine that's running the current script."* Next, on Line 35, we use `ExecuteMethod` to instruct the system to run a function called `OnDisplayViaCount` when our new button/menu item is selected.

In fact, the name of this function is arbitrary. In this example, we've simply prefixed the name of our original function with `OnDisplay`, where "On" is commonly used to signify that a function is an event handler (in this case the event is the button press) and "Display" is used to indicate (to ourselves) that we're going to display something.

Now, observe the statement on Line 38. In all of the example scripts we've run in previous chapters, the script has terminated once we've completed execution. This would be a problem in this case, because if our script terminates the button press handler – which is in

the script – cannot be called. Thus, on Line 38 we set the *DontExit* property of the *Scripting* object to the Boolean value of *True*, which means that this script will continue 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. This cannot be used for scripts executed with the *mgcscript* script engine (see also *Chapter 8*). In such a case, an infinite loop must be used to keep the script from exiting (see also *Chapter 11*)

Finally, on Lines 40 to 42, we define the contents of our *OnDisplayViaCount* function. Observe that all handlers for menu entries have a single integer parameter called *nID*. The body of this function contains the statement(s) to be executed when our new button/item is selected. In this case, we have only a single statement that employs the *ProcessKeyin* method associated with the *Gui* object. The *ProcessKeyin* method executes any valid key-in command; in this case, we're using the same key-in command (the string parameter "run C:\Temp\ViaCount.vbs") that we were required to enter by hand in *Chapter 3: Running Your First Script*.

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

Running the Script

For the purposes of this example, you will run your script from within Expedition PCB (this technique is covered in greater detail in *Chapter 9: Running a Script From Within an Application*).

- 1) Launch Expedition PCB.
- 2) At this stage you may open a layout design of your choice, however, you are not obliged to do so – see also point (6).
- 3) Right-click on the toolbar and ensure that the **Keyin Command** option is ticked (enabled).
- 4) Enter the string *run C:\Temp\ViaCountMenuDual.vbs* into the **Keyin Command** field and press the <Enter> key to execute this script.
- 5) Observe that after you've run this script it exits and nothing seems to have happened. In reality, however, this script has established a new menu button/item as illustrated in Figure 6-3.

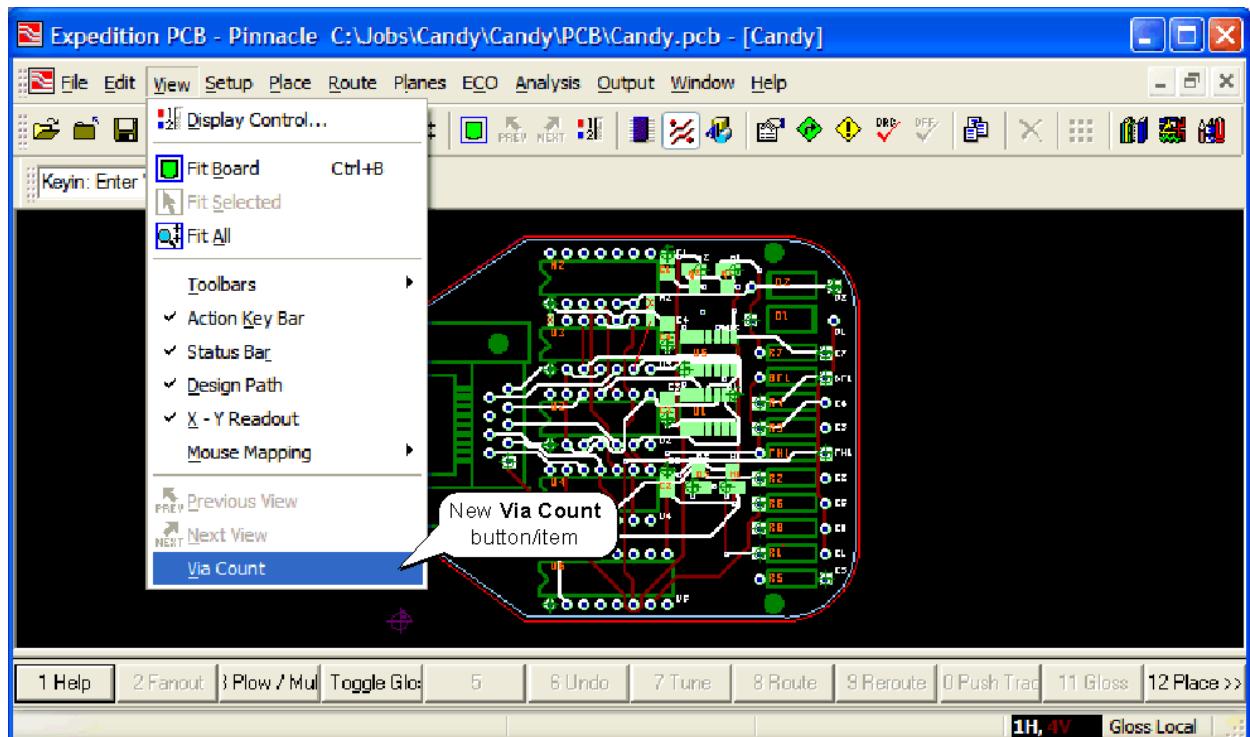


Figure 6-3. The new Via Count button/item.

- 6) Now we want to test this new menu button/item to check that it does indeed run our *ViaCount.vbs* script. If you had not already opened a layout design document in point (2), then this would be the time to do so.
- 7) Click the View menu item and then select your new button/item and observe that it does indeed launch your *ViaCount.vbs* script as illustrated in Figure 6-4.

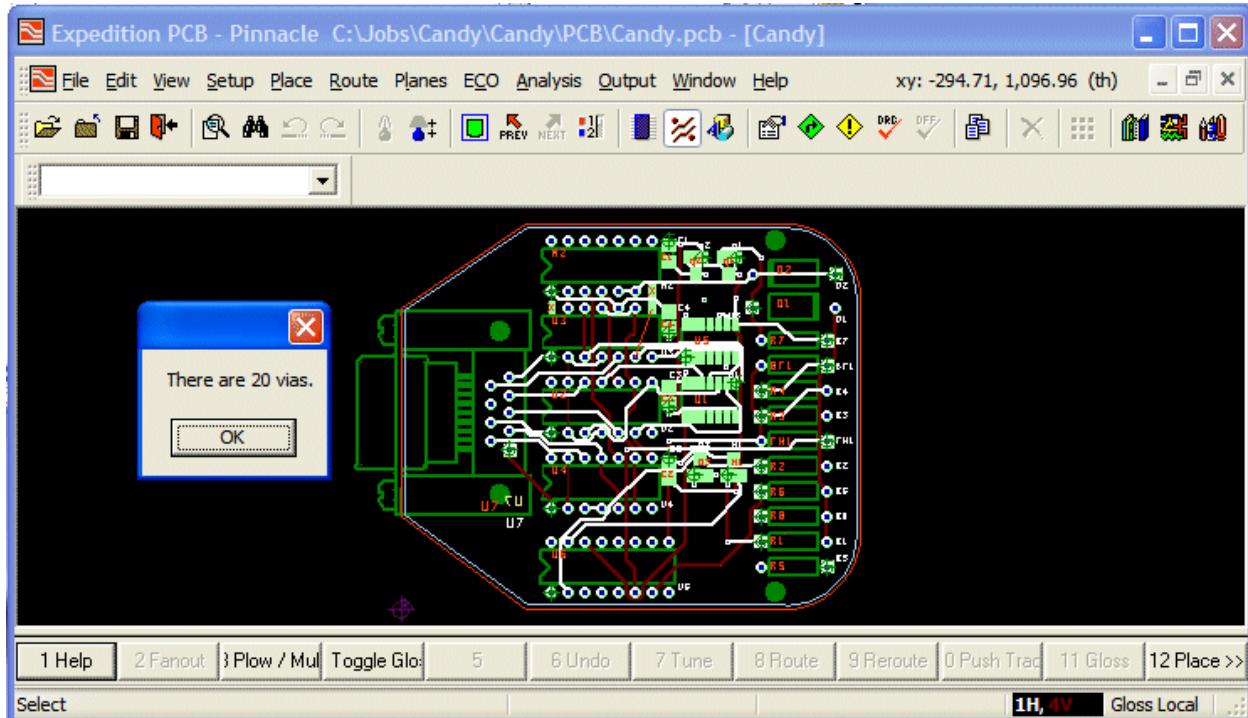


Figure 6-4. Using the View > Via Count command to run the *ViaCount.vbs* script.



Note: Your new menu button/item will remain in existence only for the duration of this session.



Note: As we discussed in the previous chapter, you may be wondering as to the point of all this. After all, rather than first having to run the *ViaCountMenuDual.vbs* script and then selecting your new **View > Via Count** command, wouldn't it be easier to simply run our original *ViaCount.vbs* script? As before, the answer to this is that this type of script would typically be set to run automatically at startup (see also *Chapter 10: Running Startup Scripts*).



Note: In this particular example we added the new menu button/item to the bottom of the existing menu. In some cases you may want to add the new button/item somewhere in the middle of an existing menu. You could use an integer to specify the location of the new button/item, but this is not an ideal solution because someone else may run a script that changes the number of items in the menu. In order to address this issue, it is possible to create some "helper functions" (see also *Chapter 12: Basic Building Block Examples*).

Modifying the Previous Example to Use a Single Script

The previous example employed two scripts: *ViaCountMenuDual.vbs* created the new menu button/item which could be used to invoke our original *ViaCount.vbs* script. This approach has its advantages in that it's very flexible and facilitates reuse, but there's always a tradeoff, because it means you now have to keep track of two scripts. The alternative is to create a single *CommandBar* script as discussed below.



Note: The script you created in the previous example is still running. In order to terminate this script and remove its associated button/item, close Expedition PCB.

Creating the Script

Use the scripting editor of your choice to open the *ViaCountMenuDual.vbs* script you created in the previous example and save it out as *ViaCountMenuSingle.vbs* (this new version should also be stored in your C:\Temp folder).

Now open your original *ViaCount.vbs* script, which should look something like the following:

```
0  ' ORIGINAL ViaCount.vbs SCRIPT
1  Option Explicit
2
3  ' Add any type libraries to be used.
4  Scripting.AddTypeLibrary( "MGCPBCB.ExpeditionPCBAutomation" )
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 ' Local functions
28
29 ' Server validation function
30 Function ValidateServer(docObj)
31
32     Dim keyInt
33     Dim licenseTokenInt
34     Dim licenseServerObj
35
36     ' Ask Expedition's document for the key
37     keyInt = docObj.Validate(0)
38
39     ' Get license server
40     Set licenseServerObj =
41         CreateObject( "MGCPBCBAutomationLicensing.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

```

Select Line 4 from the original *ViaCount.vbs* script (the line that adds the Expedition PCB type library) and paste it in *between* Lines 3 and 4 in your new *ViaCountMenuSingle.vbs* script (this will become Line 4 in the new script as shown below). As you'll see, this means that we're now opening two type libraries. The reason we didn't worry about opening the Expedition PCB type library in the previous script was that this type library was opened in the *ViaCount.vbs* script; now that we are combining the two scripts, we need to open both type libraries in our new script.

Select Lines 10 through 25 from the original *ViaCount.vbs* script and use them to replace the contents of the *OnDisplayViaCount* function in your new *ViaCountMenuSingle.vbs* script (these will become Lines 42 through 56 in the new script as shown below).

Observe that you do not require Lines 7 and 8 from the original script (the statements that acquire the *Application* object) because these statements are already present in Lines 8 and 9 of your new script as shown below.

Finally, select Lines 27 through 61 from your original script (these are the statements that perform the server validation function) and paste them onto the end of your new script where they will appear as Lines 60 through 94 as shown below.

```

0  ' NEW ViaCountMenuSingle.vbs SCRIPT
1 Option Explicit
2
3  ' Add any type libraries to be used.
4  Scripting.AddTypeLibrary( "MGCPCB.ExpeditionPCBAplication" )
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 .....
61 'Local functions
62
63 ' Server validation function
64 Function ValidateServer(docObj)
65
66     Dim keyInt
67     Dim licenseTokenInt
68     Dim licenseServerObj
69
70     ' Ask Expedition's document for the key
71     keyInt = docObj.Validate(0)
72
73     ' Get license server
74     Set licenseServerObj =
75         CreateObject( "MGCPCBAutomationLicensing.Application" )
76
77     ' Ask the license server for the license token
78     licenseTokenInt = licenseServerObj.GetToken(keyInt)
79
80     ' Release license server
81     Set licenseServerObj = nothing

```

```

82      ' Turn off error messages.
83      On Error Resume Next
84      Err.Clear
85
86      ' Ask the document to validate the license token
87      docObj.Validate(licenseTokenInt)
88      If Err Then
89          ValidateServer = 0
90      Else
91          ValidateServer = 1
92      End If
93
94  End Function

```

Running the Script

- 1) Launch Expedition PCB.
- 2) At this stage you may open a layout design of your choice, however, you are not obliged to do so – see also point (6).
- 3) Right-click on the toolbar and ensure that the **Keyin Command** option is ticked (enabled).
- 4) Enter the string *run C:\Temp\ViaCountMenuSingle.vbs* into the **Keyin Command** field and press the <Enter> key to execute this script.
- 5) As before, once you've run this script it exits and nothing seems to have happened. In reality, of course, this script has established a new menu button/item as illustrated in Figure 6-5.
- 6) As usual, you will want to test this new menu button/item to check that it performs as planned. If you had not already opened a layout design document in point (2), then this would be the time to do so.
- 7) Click the **View** menu item and then select your new button/item and observe that it does indeed count the vias as illustrated in Figure 6-6.

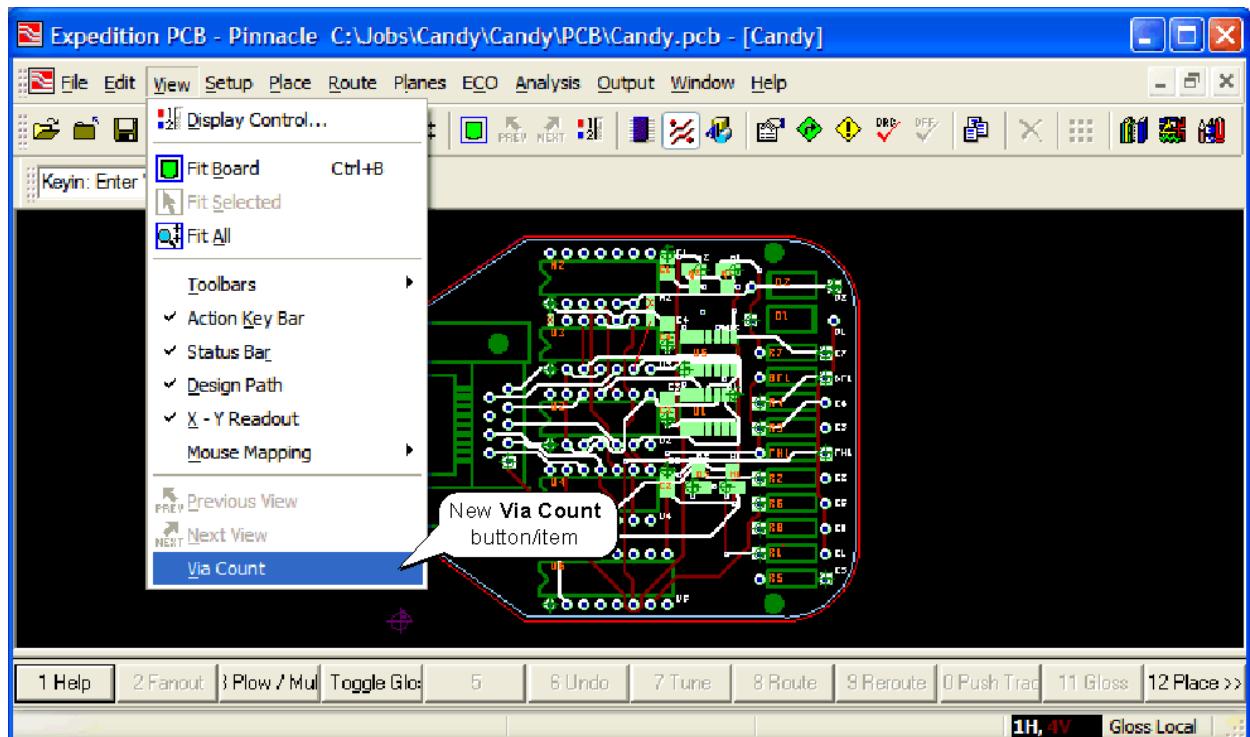


Figure 6-5. The new Via Count button/item.

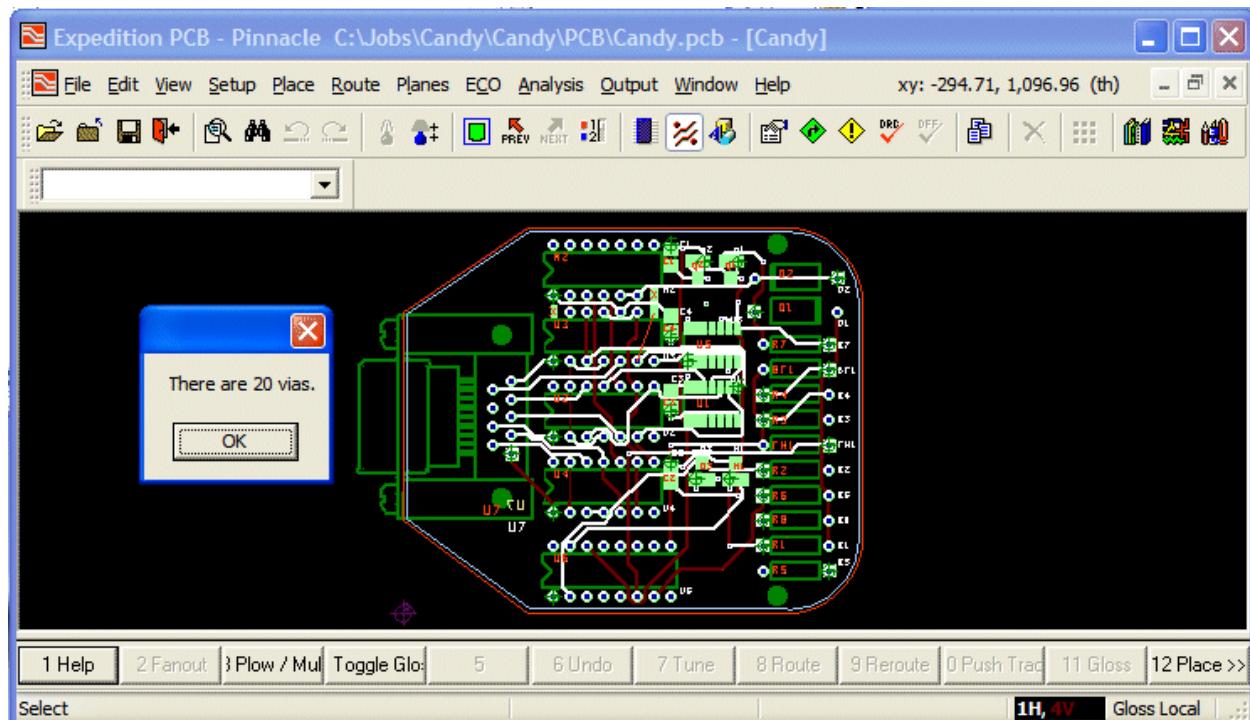


Figure 6-6. Using the new View > Via Count command.



Note: Your new menu button/item will remain in existence for the duration of this session.

Creating a New Top-Level Menu Item

As opposed to adding a button/item to an existing top-level menu, this example introduces the process of creating a new top-level menu item.



Note: The script you created in the previous example is still running. In order to terminate this script and remove its associated button/item, close Expedition PCB.

Creating the Script

Use the scripting editor of your choice to open the *ViaCountMenuSingle.vbs* script you created in the previous example and save it out as *ViaCountMenuTop.vbs* (this new version should also be stored in your *C:\Temp* folder).

As you will see, it is necessary to perform only a few modifications to the existing script. First, modify the statements at Lines 20 through 22 as shown below (observe that our original *viewMenuObj* variable has been renamed to *countMenuObj*); these modifications mean that – as opposed to finding the existing **View** menu item – we are adding a new main menu item (the fourth parameter of *-1* at the end of Line 22 instructs the Command Bar Server to add this new item to the right-hand end of the existing main menu items).

Next, add the comment in Line 24 and the statement at Line 25 as shown below. This adds the caption "Count" to the new main menu item.

All that remains is to change the variable name *viewControlsColl* to *countControlsColl* in Lines 28, 29, and 33, and also to change the variable name *viewMenuObj* to *countMenuObj* in line 29. The resulting script is as shown below:

```
1 Option Explicit
2
3 ' Add any type libraries to be used.
4 Scripting.AddTypeLibrary( "MGCPBC.ExpeditionPCBAplication" )
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
39  viaCntBtnObj.ExecuteMethod = "OnDisplayViaCount"
40
41  ' Keep this script running so that the handler can be executed.
42  Scripting.DontExit = True
43
44  Function OnDisplayViaCount(nID)
45      ' Get the active document
46      Dim pcbDoc
47      Set pcbDoc = pcbApp.ActiveDocument
48
49      ' License the document
50      ValidateServer(pcbDoc)
51
52      ' Get the vias collection
53      Dim viaColl
54      Set viaColl = pcbDoc.Vias
55
56      ' Get the number of vias in collection
57      Dim countInt
58      countInt = viaColl.Count
59
60      MsgBox( "There are " & countInt & " vias." )
61  End Function
62
63
64  .....
65  'Local functions
66
67  ' Server validation function
68  Function ValidateServer(docObj)
69
70      Dim keyInt
71      Dim licenseTokenInt
72      Dim licenseServerObj
73
74      ' Ask Expedition's document for the key
75      keyInt = docObj.Validate(0)
76
77      ' Get license server
78      Set licenseServerObj =
79          CreateObject( "MGCPCBAutomationLicensing.Application" )
80
81      ' Ask the license server for the license token
82      licenseTokenInt = licenseServerObj.GetToken(keyInt)
83
84      ' Release license server
85      Set licenseServerObj = nothing
86
87      ' Turn off error messages.

```

```

87      On Error Resume Next
88      Err.Clear
89
90      ' Ask the document to validate the license token
91      docObj.Validate(licenseTokenInt)
92      If Err Then
93          ValidateServer = 0
94      Else
95          ValidateServer = 1
96      End If
97
98  End Function

```

Running the Script

- 1) Launch Expedition PCB.
- 2) At this stage you may open a layout design of your choice, however, you are not obliged to do so – see also point (6).
- 3) Right-click on the toolbar and ensure that the **Keyin Command** option is ticked (enabled).
- 4) Enter the string *run C:\Temp\ViaCountMenuTop.vbs* into the **Keyin Command** field and press the <Enter> key to execute this script.
- 5) In this case, as soon as you run this script your new main menu item appears as illustrated in Figure 6-7.

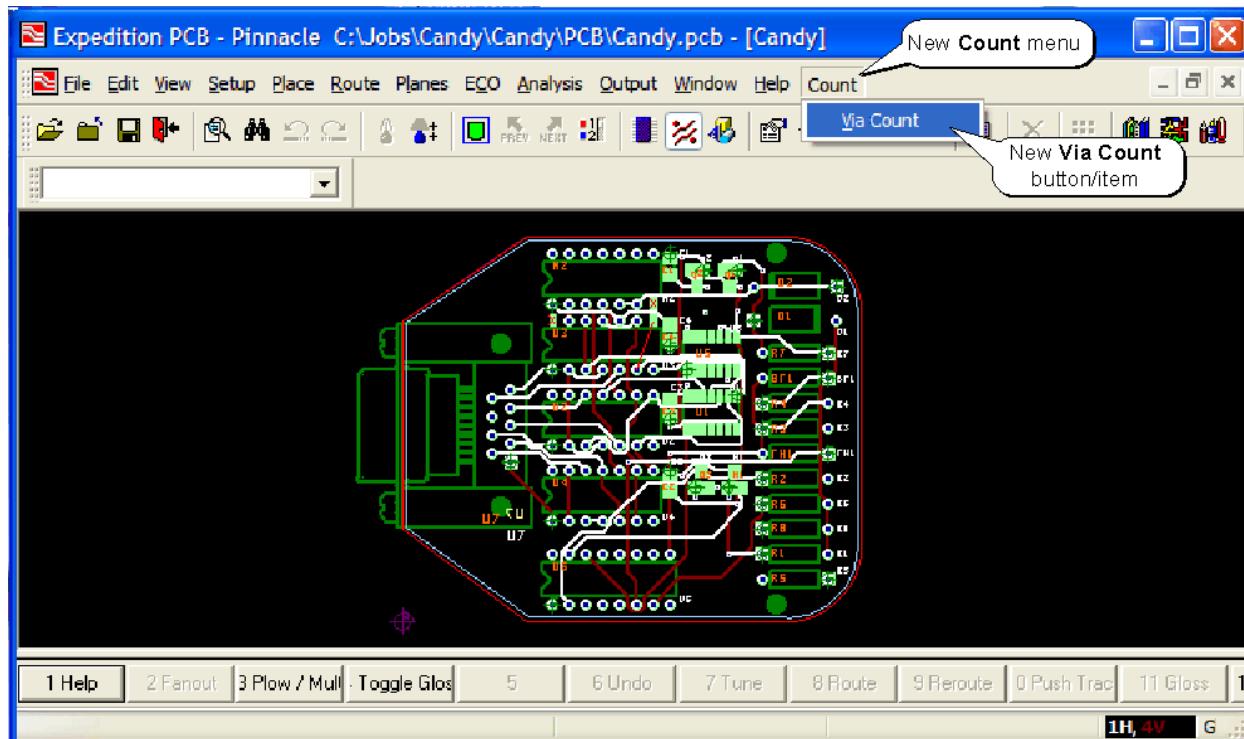


Figure 6-7. The new Count main menu item.

-
- 6) As usual, you will want to test this new menu button/item to check that it performs as planned. If you had not already opened a layout design document in point (2), then this would be the time to do so.
 - 7) Click your new **Count** main menu item and then select its associated Via Count button/item and ensure that it does indeed perform the expected function.

Chapter 7: Adding a Tool Bar Icon

Introduction

The Command Bar Server is a separate server that's used by applications such as Expedition PCB and FabLink XE to configure the icons/actions associated with the various tool bars (it is also used to modify pull-down menus on the menu bar as discussed in *Chapter 6: Adding a Menu and/or Menu Button/Item*). For example, you can add an icon to an existing tool bar and you can then associate this icon with a script.

The Command Bar Server is accessed through a scripting or programming language. This chapter describes how to create a script that adds a new icon to an existing tool bar and causes this icon (when clicked) to call the *ViaCount.vbs* script you created in *Chapter 3: Running Your First Script*.

Adding a New Icon

In this example, you are going to create a script that first adds a new icon to the end of the **Utilities** tool bar and then ties this icon to the *ViaCount.vbs* script you created in *Chapter 3: Running Your First Script*.

- 1) Launch Expedition PCB (note that we're only doing this here to look at the tool bars).
- 2) At this stage you may open a layout design of your choice (you are not obliged to open a layout design at this time – you can do so later as described in the *Running the Script* topic below. For the purposes of these discussions, however, we shall assume that you have opened a design).
- 3) Use the **View > Toolbars** command to ensure that the **Utilities** tool bar is turned on (has a checkmark) as illustrated in Figure 7-1.

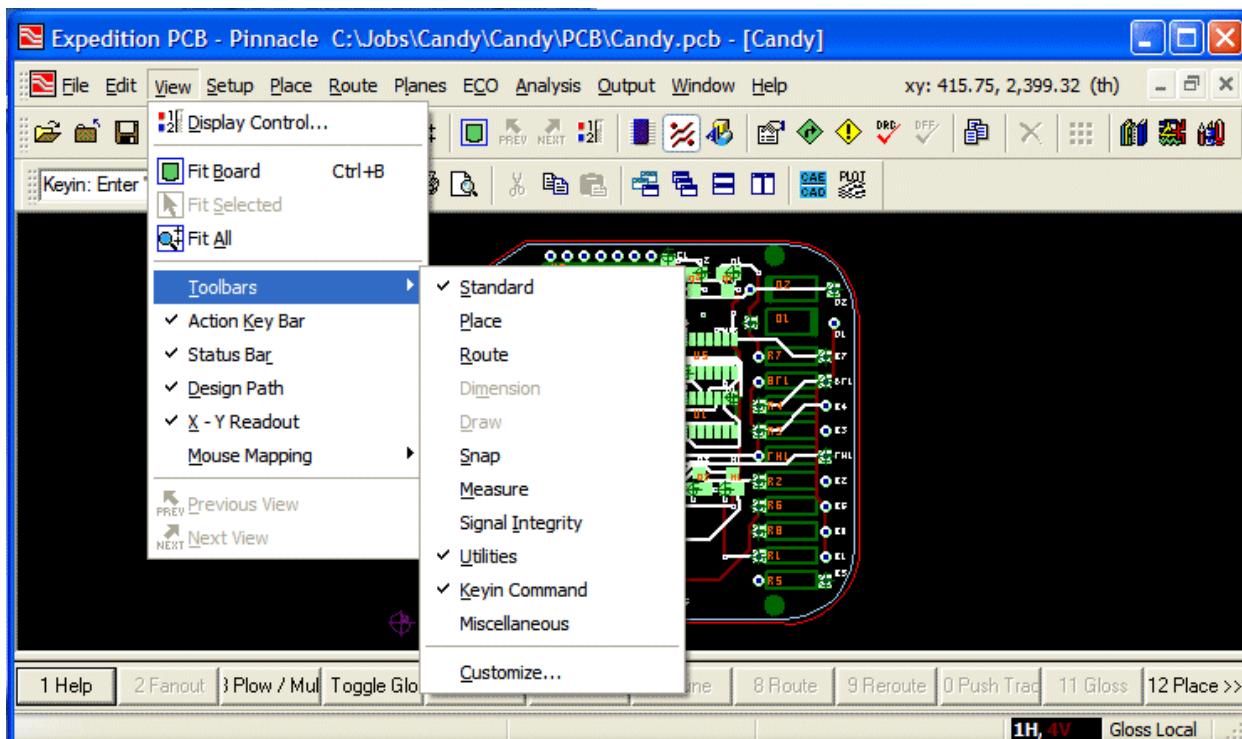


Figure 7-1. Ensuring the **View > Toolbars > Utilities** tool bar is turned on.

- 4) Observe that the last icon in the **Utilities** tool bar is currently the **Plot** icon as illustrated in Figure 7-2. We want to add a new **Via Count** icon to the right of this existing icon.

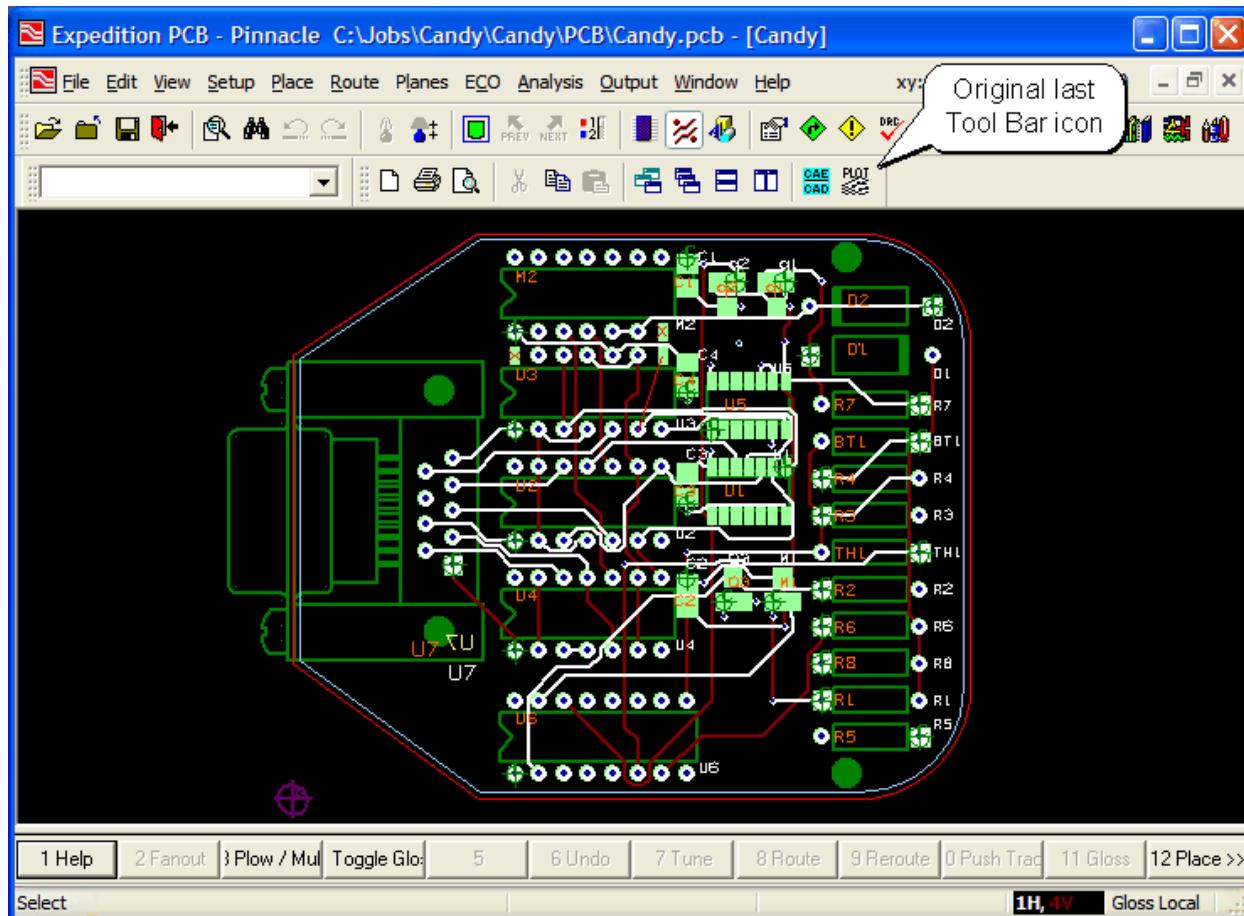


Figure 7-1. Ensuring the View > Toolbars > Utilities tool bar is turned on.

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.

On Line 1 we see the *Option Explicit* statement (the use of this statement is discussed in *Chapter 2: VBS Primer* and *Appendix A: Good Scripting Practices*).

```
1 Option Explicit
```

On Line 4 we use 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 *Chapter 11*). Any time you use the Command Bar Server – which is returned from the *Bindings* property as discussed below – you will want to add this type library.

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

On Lines 7 and 8 we access the *Application* (this was introduced in *Chapter 3: Running your First Script*). Observe that this script doesn't need to access a *Document*. Also observe that

there's no need to perform any validation at this stage, 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
```

From the Application, on Lines 11 and 12 we access the *Gui* object by calling the *Gui* property. As opposed to storing the *Gui* object in a variable (which we could certainly do), we use the very common practice to use it immediately to obtain the *CommandBars* object.

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

Observe that the string parameter at the right-hand side of the statement on Line 12. This parameter is used to specify whether you want to access one of the tool bars or the menu bar. In the case of a tool bar, this string is the name of that tool bar – "Utilities" in this example – and this returns the appropriate *CommandBars* object.

In order to manipulate (and add items to) this *CommandBars* object, we need to access a list of controls that equate to the various icons. Thus, on Line 15 we instantiate a collection called *utilitiesControlsColl* and on Line 16 we associate a list of controls from the **Utilities** tool bar with this collection.

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

We now want to add a new icon to this collection. In Line 19 we instantiate a variable called *viaCntBtnObj* that we will use to represent our new button/item and on Line 20 we use the *Add* method to add this button/item to the list of **Utilities** controls. Observe that there are four parameters associated with the *Add* method as follows:

- **Parameter 1:** This parameter specifies what we're adding; in this case, a value of *cmdControlButton* equates to a simple icon.
- **Parameter 2:** This is an optional command ID (it is not generally used).
- **Parameter 3:** This is reserved for future use.
- **Parameter 4:** The position before which you would like your new icon to appear in the collection (a value of -1 equates to the end/right-hand-side of the existing collection).

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

Once our new button/item has been added to the list of **Utilities** controls, we need to configure it. On Line 23 we specify the description text to be associated with our button/item; on Line 24 we define the "tool tip" text that will automatically appear if the user "hovers" the cursor over this icon, and on Line 25 we specify the filename for the image (a bit map *.bmp file) of the icon.



Note: Using a graphics editor of your choice, you have to have previously created a suitable 16 x 16 pixel *.bmp icon.

```

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"

```

On Line 26 and 27 we define the function that is to be called when our new button/menu item is selected. First, on Line 26, we specify the "Target," which we can visualize as being the location where this function "lives." In this example we use *ScriptEngine*, which is a special implicit object that basically equates to: *"The same engine that's running the current script."* Next, on Line 27, we use *ExecuteMethod* to instruct the system to run a function called *OnDisplayViaCount* when our new button/menu item is selected.

In fact, the name of this function is arbitrary. In this example, we've simply prefixed the name of our original function with *OnDisplay*, where "On" is commonly used to signify that a function is an event handler (in this case the event is the button press) and "Display" is used to indicate (to ourselves) that we're going to display something.

Now, observe the statement on Line 30. In most of the example scripts we've run in previous chapters (apart from those in *Chapter 6*), the script has terminated once we've completed execution. This would be a problem in this case, because if our script terminates the icon press handler – which is in the script – cannot be called. Thus, on Line 30 we set the *DontExit* property of the *Scripting* object to the Boolean value of *True*, which means that this script will continue running until the current session ends.

```

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

```

Finally, on Lines 32 to 34, we define the contents of our *OnDisplayViaCount* function. Observe that all handlers for tool bar icon entries have a single integer parameter called *nID*; this parameter must be present in order for the handler to work, but you can ignore the parameter within the function. The body of this function contains the statement(s) to be executed when our new button/item is selected. In this case, we have only a single statement that employs the *ProcessKeyin* method associated with the *Gui* object. The *ProcessKeyin* method executes any valid key-in command; in this case, we're using the same key-in command (the string parameter "run C:\temp\ViaCount.vbs") that we were required to enter by hand in *Chapter 3: Running Your First Script*.

```

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

```

Running the Script

For the purposes of this example, you will run your script from within Expedition PCB (this technique is covered in greater detail in *Chapter 9: Running a Script From Within an Application*).

- 1) If you haven't already done so, launch Expedition PCB.
- 2) At this stage you may open a layout design of your choice, however, you are not obliged to do so – see also point (6).
- 3) Right-click on the toolbar and ensure that the **Keyin Command** option is ticked (enabled).

- 4) Enter the string `run C:\Temp\ViaCountToolBar.vbs` into the **Keyin Command** field and press the <Enter> key to execute this script.
- 5) Observe that after you've run this script a new icon appears on the **Utilities** tool bar as illustrated in Figure 7-3.

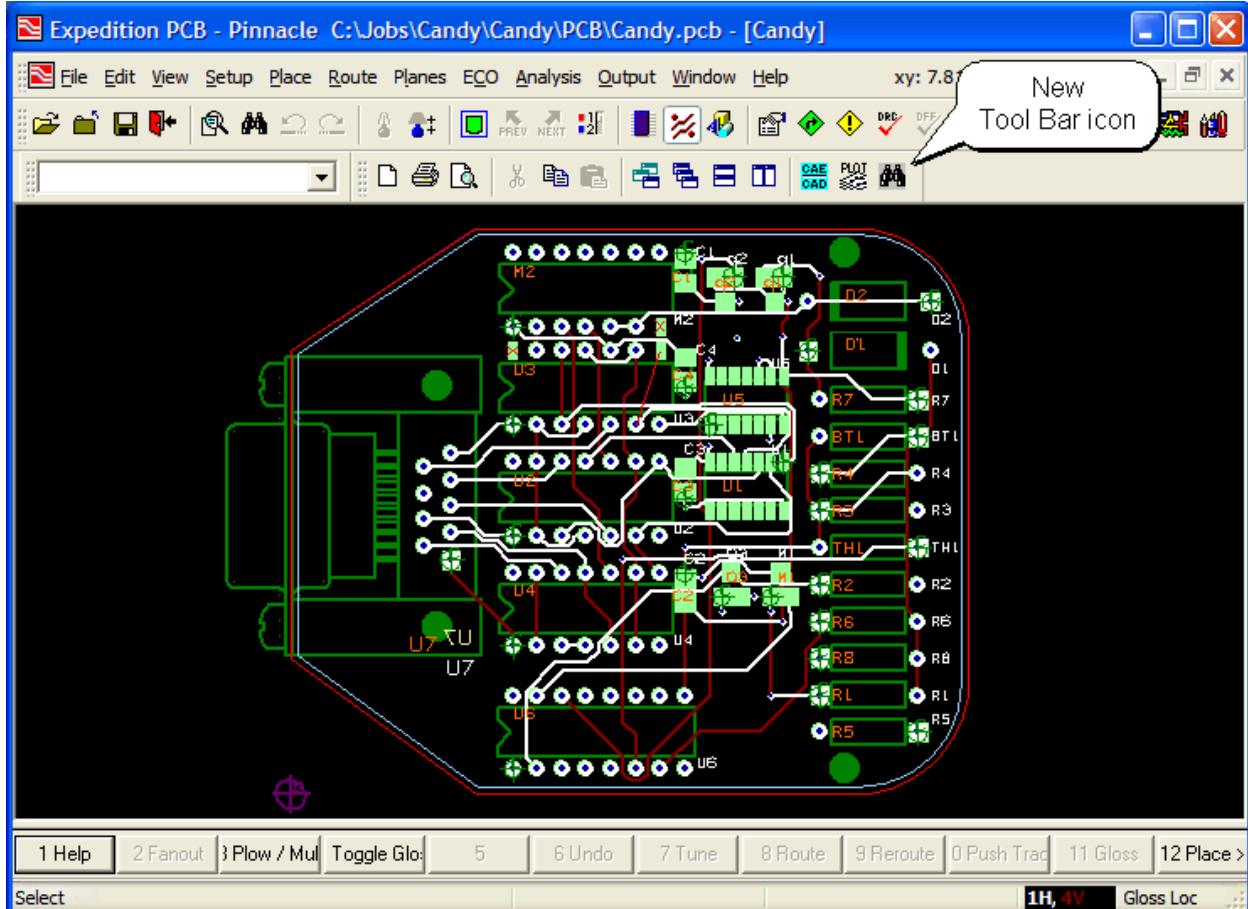


Figure 7-3. The new Via Count icon on the Utilities menu.

- 6) Of course, you will want to test this new tool bar icon to check that it does indeed run our *ViaCount.vbs* script. If you had not already opened a layout design document in point (2), then this would be the time to do so.
- 7) Click the new tool bar icon and observe that it does indeed launch your *ViaCount.vbs* script.



Note: Your new menu button/item will remain in existence only for the duration of this session.



Note: As discussed in the previous chapter, this type of script would typically be set to run automatically at startup (see also *Chapter 10: Running Startup Scripts*).



Note: In this particular example we added the new icon to the end (right-hand side) of an existing tool bar. In some cases you may want to add the icon somewhere in the middle of an existing tool bar. In order to facilitate this, it is possible to create some "helper functions" (see also *Chapter 12: Basic Building Block Examples*).

Chapter 8: Running Scripts from the Command Line

Introduction

For the purpose of these discussions, we'll assume that we're working under the Windows® operating system, which has two built-in scripting engines as follows:

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

By default VBScript (*.vbs) files are associated with the *Wscript.exe* scripting engine. If you want, you can change this association to another scripting engine such as *Cscript.exe* or Mentor Graphic's *MGCscript.exe*.



Note: There is no *.exe file extension on UNIX or Linux platforms. Also, on UNIX and Linux operating systems you will only have the choice of the *MGCScript* script engine.

The point we are interested in here is the fact that each scripting engine has its own unique set of implicit objects. In order to illustrate what this means, perform the following simple experiment:

- 1) Open an Explorer window and use it to locate and view the contents of the C:\Temp folder (directory), which is where you have stored all of the scripts you have created thus far as illustrated in Figure 8-1.

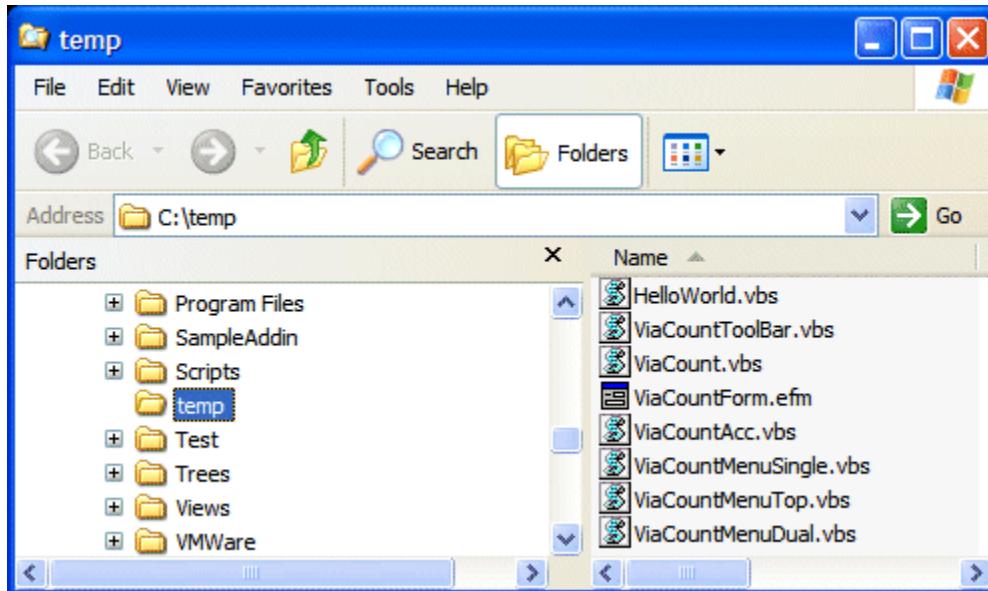


Figure 8-1. The contents of the C:\Temp folder.

- 2) Double-click on the *HelloWorld.vbs* script, which you created in *Chapter 2: VBS Primer*. By default, this runs on the *Wscript.exe* scripting engine as discussed above. The result is the expected *MsgBox()* (Figure 8-2).



Figure 8-2. The *HelloWorld* script runs in *Wscript.exe* without any problems.

-
- 3) If you haven't already done so, launch Expedition PCB and open a layout design of your choice.
 - 4) Now, double-click on the *ViaCount.vbs* script, which you also created in *Chapter 3*. This script fails as illustrated in Figure 8-3, because it is attempting to access the *Scripting* object and this object is not supported by the *Wscript.exe* scripting engine.

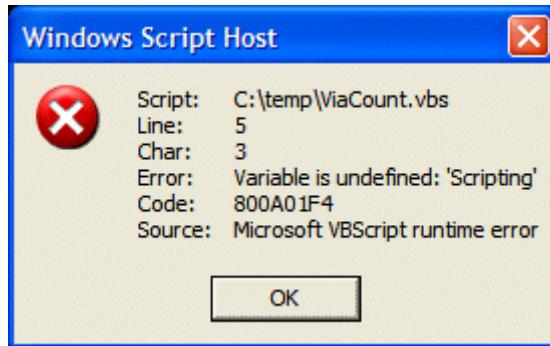


Figure 8-3. The *ViaCount* script fails in *Wscript.exe*

As an alternative, you can run scripts from the command line, in which case you can specify the scripting engine you want to use to execute the script as discussed below.

Running Scripts from the Command Line (no Arguments)

Once again, for the purpose of these discussions, we'll assume that we're working under the Windows® operating system.

- 1) Click the Start button, then select the Run option, then type "cmd" (short for "command") as illustrated in Figure 8-4.

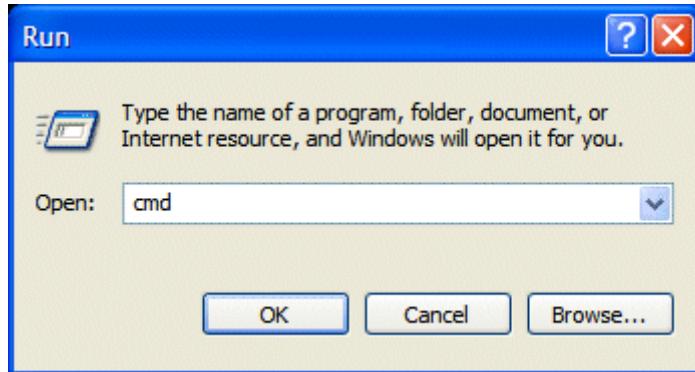


Figure 8-4. Using the Run dialog.

- 2) Click the **OK** button or press the <Enter> key to execute this command, which will launch a *console window* (sometimes referred to as a *terminal*) as illustrated in Figure 8-5.

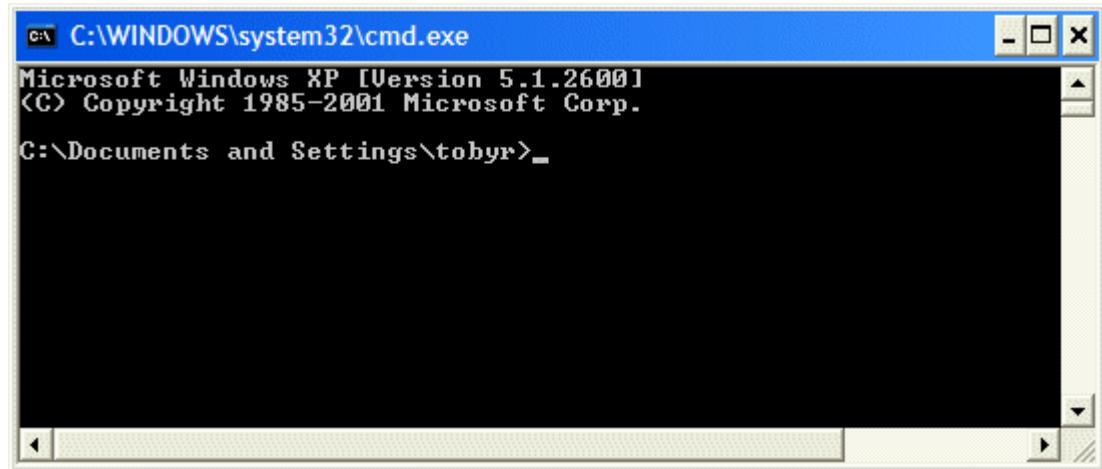


Figure 8-5. Launching a console (terminal) window.

- 3) Use the **cd** ("change directory") command to change the context to your **C:\Temp** directory (more commonly referred to as a *folder* in Windows®) as illustrated in Figure 8-6 (this is where you've been storing all of your scripts thus far).

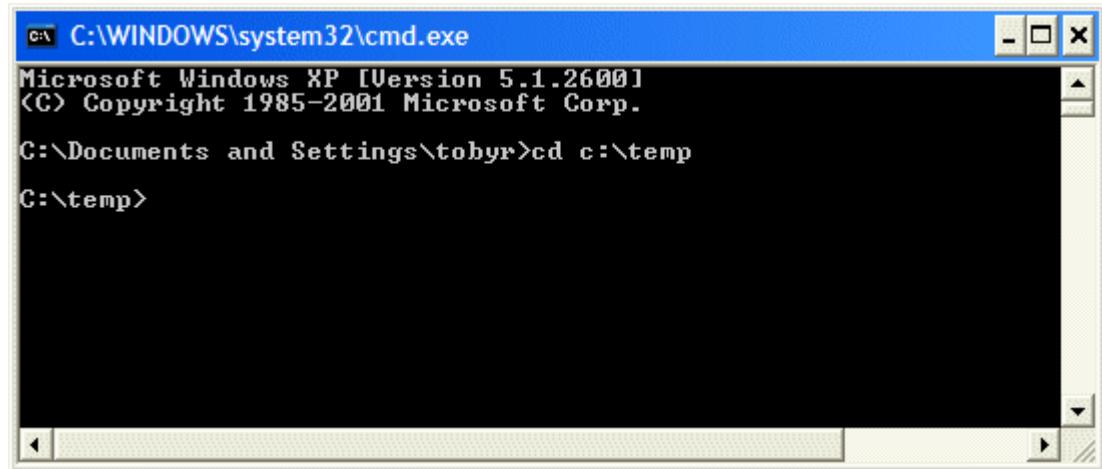


Figure 8-6. Changing the context to your C:\Temp folder.

- 4) Mentor's scripting engine is called *MGCscript.exe*. Type *mgcscript HelloWorld.vbs* and press the <Enter> key. This calls the *MGCscript.exe* scripting engine and passes the *HelloWorld.vbs* script to it as an argument. The script runs as illustrated in Figure 8-7.

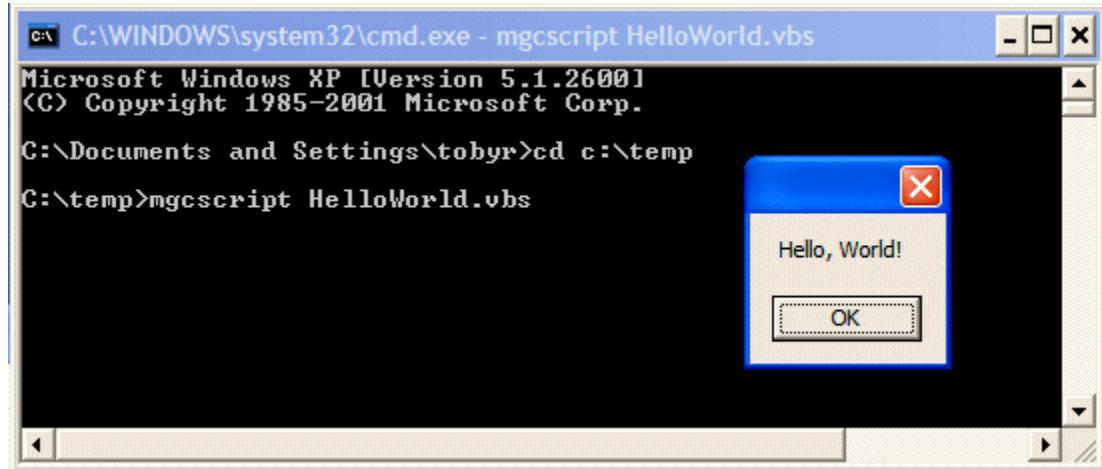


Figure 8-7. Running the `HelloWorld` script from the command line.



Note: The reason for entering `mgcscript` rather than `mgcscript.exe` is that it's common practice to omit the `.exe` portion of the filename when calling an executable program. This is because – in this type of situation – Windows® will assume a `*.exe` or `*.bat` file.



Note: Generally speaking, the terms "argument" and "parameter" can be used interchangeably. For the purpose of this tutorial, however, we will use "parameter" to refer to values being passed into a function, while "argument" will be used to refer to values being passed into a script from the command line.

- 5) Observe that the command line prompt has not reappeared in Figure 8-7. This is because the script is still running. Click the **OK** button to dismiss the `MsgBox()` and terminate the script, and observe that the command line prompt reappears (Figure 8-8).

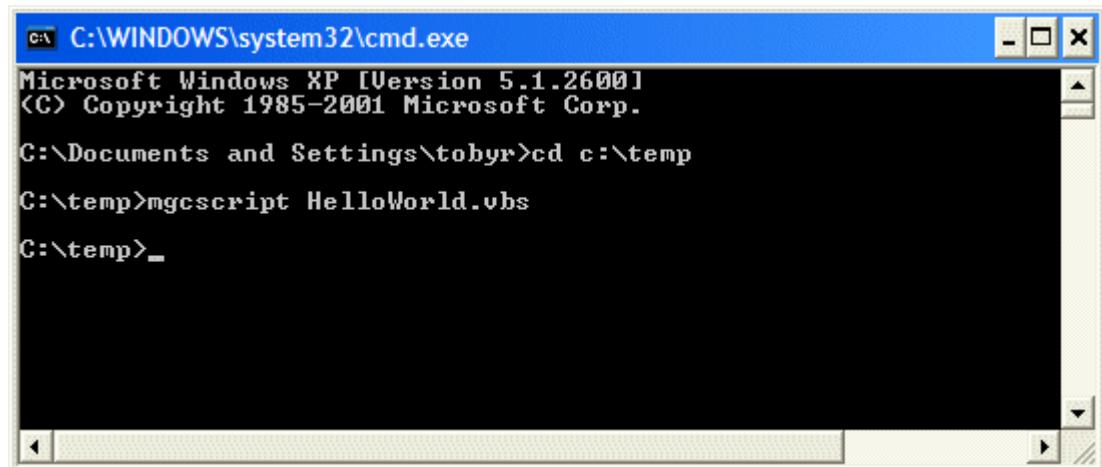


Figure 8-8. The command line prompt reappears when the script stops running.

Running Scripts from the Command Line (with Arguments)

This topic presents two simple examples involving arguments being passed into scripts and subsequently used by those scripts.

Example #1: Simply Display a Series of Arguments

Use the scripting editor of your choice to enter the following script and save it as *DisplayArguments.vbs* in your 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
```

On Line 3 we declare the variable *argumentColl*, in which we're going to store a collection of arguments. On Line 4 we declare the variable *argumentObj*, to which we can assign individual argument objects.

The *ScriptHelper* object is another explicit scripting object that is specific to the *MGCscript.exe* scripting engine. The predominant (read-only) property associated with the *ScriptHelper* object is *Arguments*. We use this property on Line 7 to assign the collection of command line arguments to our *argumentColl* variable.

Finally, in Lines 10 through 12, we cycle around using a *MsgBox()* to display each argument in turn. (Remember that the *For Each ... In ... Next* statement is a special control statement that is used to process collections of objects. This statement was introduced in *Chapter 2: VBS Primer*.)

- 1) Type *mgcscript DisplayArguments.vbs Hello Sailor 3* into the command line as illustrated in Figure 8-9.



Note: Programmers who are performing 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 West Coast 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*).



Note: Both the call to the *MGCscript.exe* scripting engine and the name of the *DisplayArguments.vbs* script are considered to be arguments from the perspective of the command line. Thus, our *mgcscript DisplayArguments.vbs Hello Sailor 3* actually has five arguments as illustrated in Figure 8-9.



Note: Every argument on the command line is considered to be (and is treated as) a string; for example, our last argument '3' will be treated as a string, not an integer. We will return to consider this point in more detail in later examples in this topic.

```

C:\WINDOWS\system32\cmd.exe
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\toby\>cd c:\temp

C:\temp>mgcscript HelloWorld.vbs
C:\temp>mgcscript DisplayArguments.vbs Hello Sailor 3_
Argument 1 Argument 2 Argument 3 Argument 4 Argument 5

```

Figure 8-9. Passing arguments into a script

- 2) Press the <Enter> key to execute this command and run the *DisplayArguments.vbs* script. The first argument immediately appears in a *MsgBox()* as illustrated in Figure 8-10 [observe that the full path name for this file is presented in the *MsgBox()*].

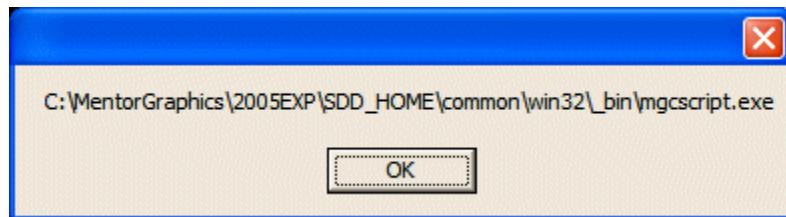


Figure 8-10. Displaying the first argument.

- 3) Click the **OK** button to dismiss this *MsgBox()*. The script now displays the second argument as illustrated in Figure 8-11.

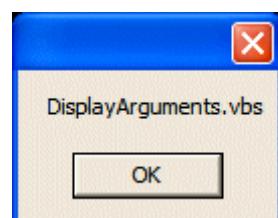


Figure 8-11. Displaying the second argument.

- 4) Click the **OK** button to dismiss this *MsgBox()*. The script now displays the third argument as illustrated in Figure 8-12.



Figure 8-12. Displaying the third argument.

-
- 5) Click the **OK** button to dismiss this *MsgBox()*. The script now displays the fourth argument as illustrated in Figure 8-13.

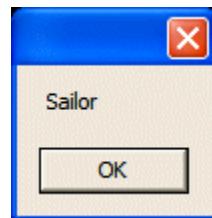


Figure 8-13. Displaying the fourth argument.

- 6) Click the **OK** button to dismiss this *MsgBox()*. The script now displays the fifth argument as illustrated in Figure 8-14.

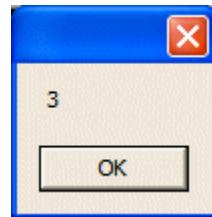


Figure 8-14. Displaying the fifth argument.

- 7) Click the **OK** button to dismiss this *MsgBox()* and terminate the script.
- 8) Observe that – in the previous example – the words *Hello* and *Sailor* were considered by the system to be two separate arguments. This was due to the space between them. If we want these two words to be treated as a single argument, we have to "gather them together" using double quotes as illustrated in Figure 8-15.

A screenshot of a Windows command prompt window titled "C:\WINDOWS\system32\cmd.exe". The window shows the following text:

```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\toby\>cd c:\temp

C:\temp>mgcscript HelloWorld.vbs

C:\temp>mgcscript DisplayArguments.vbs Hello Sailor 3
C:\temp>mgcscript DisplayArguments.vbs "Hello Sailor" 3
```

Below the command line, four green callout boxes point to the arguments: "Argument 1" points to "Hello", "Argument 2" points to "Sailor", "Argument 3" points to "3", and "Argument 4" points to the closing double quote of the string "Hello Sailor".

Figure 8-15. Using double quotes to combine words into a single argument.

- 9) Press the <Enter> key to execute this command and re-run the *DisplayArguments.vbs* script with this new set of arguments. The first argument immediately appears in a *MsgBox()* as illustrated in Figure 8-16.

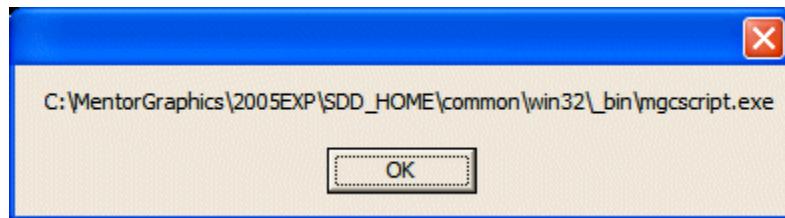


Figure 8-16. Displaying the first argument.

- 10) Click the **OK** button to dismiss this *MsgBox()*. The script now displays the second argument as illustrated in Figure 8-17.

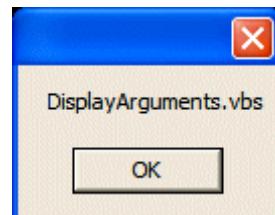


Figure 8-17. Displaying the second argument.

- 11) Click the **OK** button to dismiss this *MsgBox()*. The script now displays the third argument as illustrated in Figure 8-18.

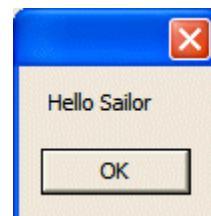


Figure 8-18. Displaying the third argument.

- 12) Click the **OK** button to dismiss this *MsgBox()*. The script now displays the fourth argument as illustrated in Figure 8-19.

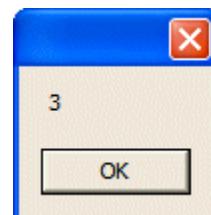


Figure 8-19. Displaying the fourth argument.

- 13) Click the **OK** button to dismiss this *MsgBox()* and terminate the script.

Example #2: Actually Using the Arguments

Before we start, it's probably worth noting that the goal of this script is to accept two user-defined arguments; the first will be a string, while the second will be an integer (actually, as noted above, all of the arguments are considered to be strings, but we will eventually use the second argument in the role of an integer). The script will repeatedly display the string forming the first argument a number of times as defined by the second argument.

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

On Line 1 we see our usual Option Explicit statement. On Line 3 we declare the variable *argumentColl*, in which we're going to store a collection of arguments. Then, on Line 6, we use the *Arguments* property associated with the *ScriptHelper* object to assign the collection of command line arguments to our *argumentColl* variable.

```
1  Option Explicit
2
3  Dim argumentColl
4
5  ' Get the collection of arguments
6  Set argumentColl = ScriptHelper.Arguments
```

Now, just to aid us to visualize how this works, assume that when we eventually run this script we're going to pass it two user-defined arguments as follows:

```
mgcscript ParsingArguments.vbs "Hello Sailor" 3
(1)           (2)           (3)           (4)
```

The numbers shown in parentheses reflect the order of the arguments. The two user-defined arguments that we are passing into the script are the string *Hello Sailor* and the string 3 (that we intend to use as an integer).

Of course, we'd really have problems if we declared too few arguments when running our script (having too many arguments wouldn't be a problem, because we could simply ignore any "extras"). Thus, the next thing we do (on Line 9) is to use the *Count* property on our collection of arguments to ensure that the expected four arguments are present. If there is a problem, the statements in Lines 24 through 28 will display an error message explaining what the user is required to enter; otherwise we will proceed to line 10.

On Line 11 we instantiate a variable called *textStr*, and then we use the *Item* property on our collection to assign the third argument – our *Hello Sailor* string – to this variable. (Observe the colon ":" character on Line 11. This is used as a separator that allows multiple statements to be used on a single line. This construct was introduced in *Chapter 2: VBS Primer*.)

Similarly, on Line 12 we instantiate a variable called *rptInt* (where rpt stands for "repeat") and then we use the *Item* property on our collection to assign the fourth argument – our 3 string – to this variable. The reason we use the "Int" suffix on the variable *rptInt* – even though we know this is actually being stored as a string – is to remind ourselves that we are going to be using it in the role of an integer.

The clever part of all of this is that, even though VBScript knows that the value that is currently assigned to *rptInt* is being stored as a string, it will automatically cast (convert) this value into an integer when this variable is used in the context of an integer variable (we will see this in action on Line 17).

```
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)
```

On Line 15 we instantiate a variable called *displayStr* in which we intend to hold the string to be displayed. Observe that we also initialize this string with a null value (an empty string in this case). Generally speaking, we don't need to initialize strings with null values; the reason

we do so here is that our script features a loop in which *displayStr* is concatenated with itself, which means that we need to establish an initial value.

```
14      ' Loop to build up string to be displayed
15      Dim displayStr: displayStr = ""
```

On Line 16 we instantiate an iteration variable called *i* that we intend to use to control our loop. Theoretically, following the conventions laid down in *Appendix A: Good Scripting Practices*, we should actually name this variable *iInt*, but the use of *i* in this role is almost universal.¹

On Line 17 we start a *For ... Next* loop that iterates from 1 to whatever number is represented by our *rptInt* variable. This is where VBScript automatically casts the string assigned to *rptInt* into an integer for the purposes of controlling this loop. For each iteration of the loop, the current string assigned to our *displayStr* variable is concatenated with itself, and with a copy of the third (string) argument that was passed into the script, and with a special "carriage-return line-feed" (newline) constant. (The use of the *vbCrLf* constant was introduced in *Chapter 2: VBS Primer*.)

```
16      Dim i
17      For i = 1 To rptInt
18          displayStr = displayStr & textStr & vbCrLf
19      Next
```

On Line 22, the resulting string is displayed in a *MsgBox()*.

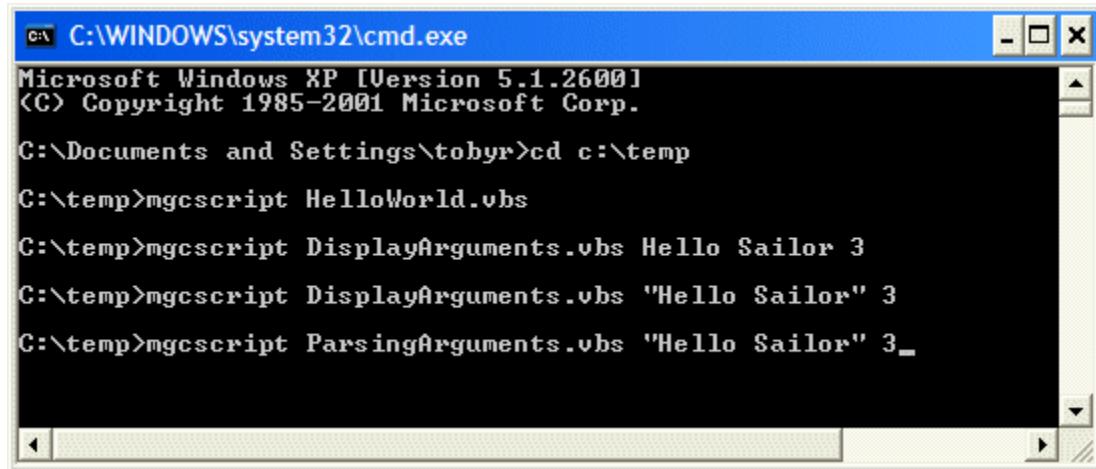
```
21      'Display string
22      MsgBox displayStr
```

Last but not least, Lines 26 and 27 define the error message we want to display if the user enters an 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
```

¹ Also, as Group Captain Sir Douglas Robert Steuart Bader (a fighter pilot in the Royal Air Force during WWII) famously said: "*Rules are for the obedience of fools and the guidance of wise men.*"

-
- 1) Type in the command as illustrated in Figure 8-20.



C:\WINDOWS\system32\cmd.exe
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.
C:\Documents and Settings\toby>cd c:\temp
C:\temp>mgcscript HelloWorld.vbs
C:\temp>mgcscript DisplayArguments.vbs Hello Sailor 3
C:\temp>mgcscript DisplayArguments.vbs "Hello Sailor" 3
C:\temp>mgcscript ParsingArguments.vbs "Hello Sailor" 3

Figure 8-20. Preparing to run the ParsingArguments.vbs script

- 2) Press the <Enter> key to execute the script and observe the resulting *MsgBox()* as illustrated in Figure 8-21.



Figure 8-21. Hurrah! It Works!

- 3) Click the **OK** button to dismiss the *MsgBox()* and terminate the script.
- 4) Experiment running the script with two few and/or too many parameters.

More-Sophisticated Error Handling

In the second script example presented above, we included a simple error test to check that the correct number of arguments is present. In order to make this script of "production-level" quality, however, it would be good programming practice to at least ensure that the last argument (the second user-defined argument) is an integer; that is, a string that can be cast as an integer.

The "poor man's" way of doing this is to explicitly cast the second user-defined argument as an integer before using it. For example, consider our original script:

```
1 Option Explicit
2
3 Dim argumentColl
4
5 ' Get the collection of arguments
6 Set argumentColl = ScriptHelper.Arguments
7
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)
13
14      ' Loop to build up string to be displayed
15      Dim displayStr: displayStr = ""
16      Dim i
17      For i = 1 To rptInt
18          displayStr = displayStr & textStr & vbCrLf
19      Next
20
21      'Display string
22      MsgBox displayStr
23
24 Else
25     ' Display an error
26     MsgBox "Usage: ParsingArguments.vbs " &
27         "<string to display> <number to repeat>"
28 End If

```

There are two locations in which we could cast the second user-defined 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. The first is that – when you come to look at your code in the event of a problem – this makes it clear what you are trying to do; that is, that you expect to cast and use this argument in the form of an integer. The second is that the error message displayed by the system can better explain just what it is that's gone wrong. For example, let's assume that you make the modification on Line 12 as shown above and that you then pass the text string *three* instead of 3 as the last argument into your *ParsingArguments.vbs* script (as illustrated in the last entry in Figure 8-22).

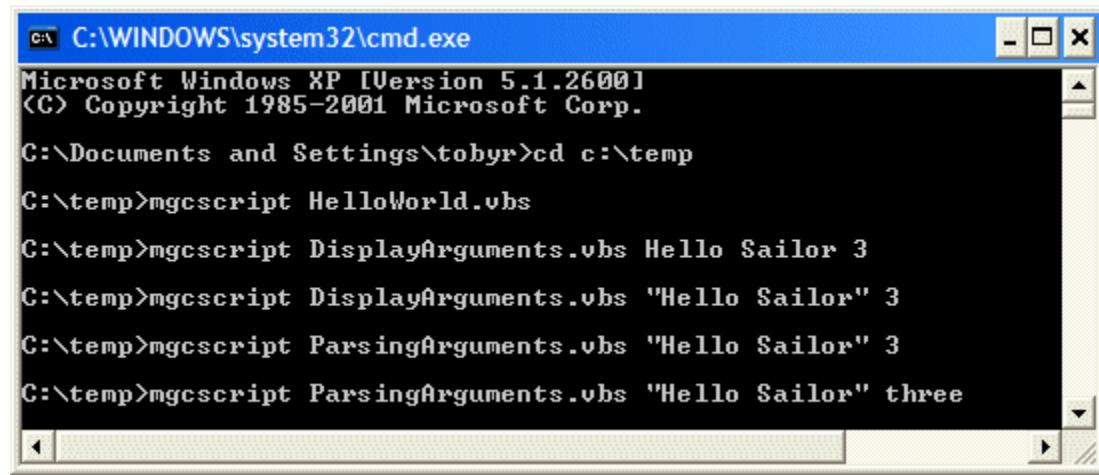
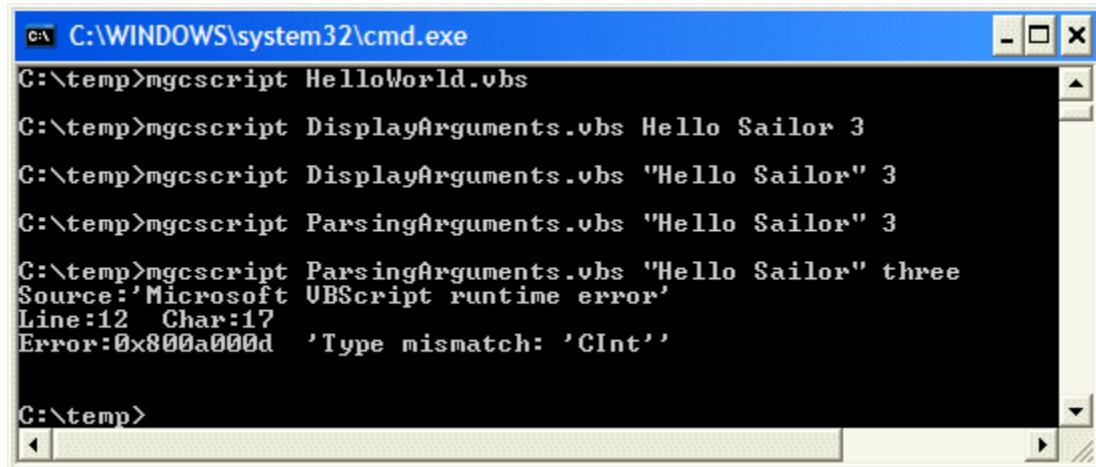


Figure 8-22. Changing the last argument from '3' to 'three'.

In this case, the system would respond with the error message shown in Figure 8-23.

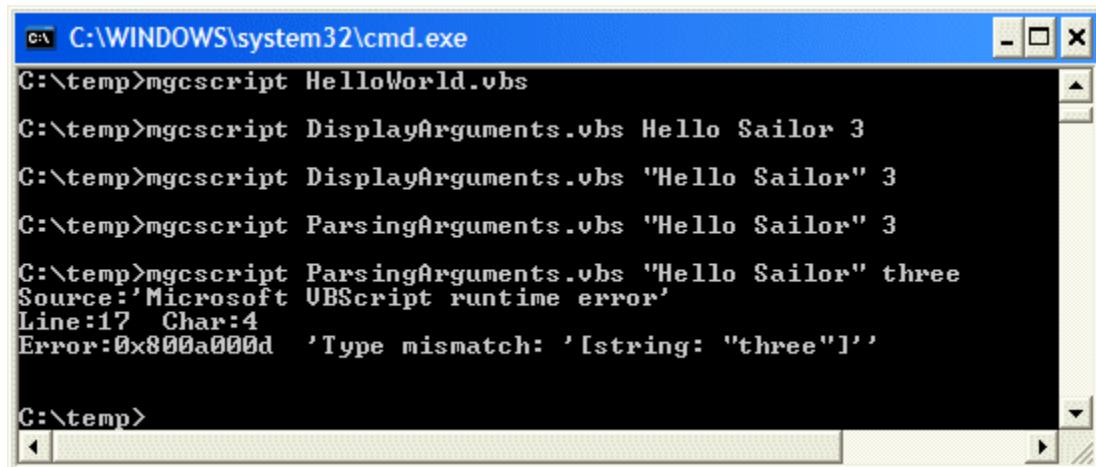


```
C:\WINDOWS\system32\cmd.exe
C:\temp>mgcscript HelloWorld.vbs
C:\temp>mgcscript DisplayArguments.vbs Hello Sailor 3
C:\temp>mgcscript DisplayArguments.vbs "Hello Sailor" 3
C:\temp>mgcscript ParsingArguments.vbs "Hello Sailor" 3
C:\temp>mgcscript ParsingArguments.vbs "Hello Sailor" three
Source:'Microsoft VBScript runtime error'
Line:12 Char:17
Error:0x800a000d 'Type mismatch: 'CInt''

C:\temp>
```

Figure 8-23. Observe 'CInt' being displayed as part of the error message.

As we see, the error message indicates that we were expecting an integer argument (or at least, a string that could be cast as an integer). If we had not performed this explicit casting, then the error would have occurred on Line 17 as illustrated in Figure 8-24.



```
C:\WINDOWS\system32\cmd.exe
C:\temp>mgcscript HelloWorld.vbs
C:\temp>mgcscript DisplayArguments.vbs Hello Sailor 3
C:\temp>mgcscript DisplayArguments.vbs "Hello Sailor" 3
C:\temp>mgcscript ParsingArguments.vbs "Hello Sailor" 3
C:\temp>mgcscript ParsingArguments.vbs "Hello Sailor" three
Source:'Microsoft VBScript runtime error'
Line:17 Char:4
Error:0x800a000d 'Type mismatch: '[string: "three"]''
```

Figure 8-24. A slightly less helpful error message.

This is slightly less helpful, because all the error message is telling us is the script was *not* expecting to see a string, but it doesn't tell us what the script was expecting to see.

Of course, the real ("industrial strength") way to handle this would be to combine the explicit *CInt* cast technique on Line 12 with an *On Error Resume Next* statement (this statement was introduced in *Chapter 2: VBS Primer*). In the case of an error, this would allow us to use the *Err* object to determine the type of error and react accordingly (note that the *Err* object can be used to provide both the error number and description).

Attaching and/or Creating Applications and Opening Documents

None of the test scripts discussed thus far in this chapter have required access to an application such as Expedition PCB and a document such as a layout design document. But suppose we create a script that does require access to an application and a document? What

happens if the application isn't running? What happens if the application is running but a document has not yet been opened? In fact there are three cases we have to consider as follows:

Application	Document
Open	Open
Open	Closed
Closed	Closed

Attaching (Application Open, Document Open)

Use the scripting editor of your choice to open the original *ViaCount.vbs* script you created in *Chapter 3* (this should be located in the *C:\Temp* folder on your system) and save it as *ViaCountAttachDO.vbs*, where the "DO" portion of this filename stands for "Document Open". This script will initially appear as follows:

```
1
2 Option Explicit
3
4 ' Add any type libraries to be used.
5 Scripting.AddTypeLibrary( "MGCPCB.ExpeditionPCBAplication" )
6
7 ' Get the Application object
8 Dim pcbAppObj
9 Set pcbAppObj = Application
10
11 ' Get the active document
12 Dim pcbDocObj
13 Set pcbDocObj = pcbAppObj.ActiveDocument
14
15 ' License the document
16 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 .....
29 'Local functions
30
31 ' Server validation function
32 Private Function ValidateServer(docObj)
   :
   etc.
```

Comment out the original Line 9 and insert a new Line 10 as shown below:

```
9 'Set pcbAppObj = Application
10 Set pcbAppObj = GetObject(, "MGCPCB.ExpeditionPCBAplication")
```

The `GetObject` function is used to attach to an application that is already running. The first parameter is defaulted and the second parameter is a string that is used to specify the *ProgID* that is used to uniquely identify a server – Expedition PCB in this case.



Note: If you have multiple instantiations of the same application open, then it's arbitrary as to which instantiation the `GetObject` function will attach.



Note: With regard to the previous point, the `GetObject` function is part of VBScript and cannot be changed. However, Mentor has provided a way to attach to a specific application; this is discussed in more detail in *Chapter 12: Basic Building Block Examples*.



Note: The `GetObject` function is not specific to the *mgcscript* engine. For example, it can be employed by scripts running in one application (say FabLink XE) that need to attach to another application (such as Expedition PCB).



Note: If we had left the script "as-is" (that is, not commented out Line 9 and replaced it with Line 10 as discussed above), and if we had then run this script with the *mgcscript* engine, we would receive the error message: *Variable is undefined: 'Application'*. The reason for this is that the *mgcscript* engine doesn't have the context of a server (unlike running a script in the Expedition PCB or FabLink XE scripting hosts).

- 1) If you haven't already done so, launch Expedition PCB.
- 2) Open a layout design of your choice.
- 3) Open a Console/Terminal window and set the context to *C:\Temp*.
- 4) Run the *ViaCountAttachDO.vbs* script from the command line and observe that the script attaches to the currently running instantiation of Expedition PCB (Figure 8-25).

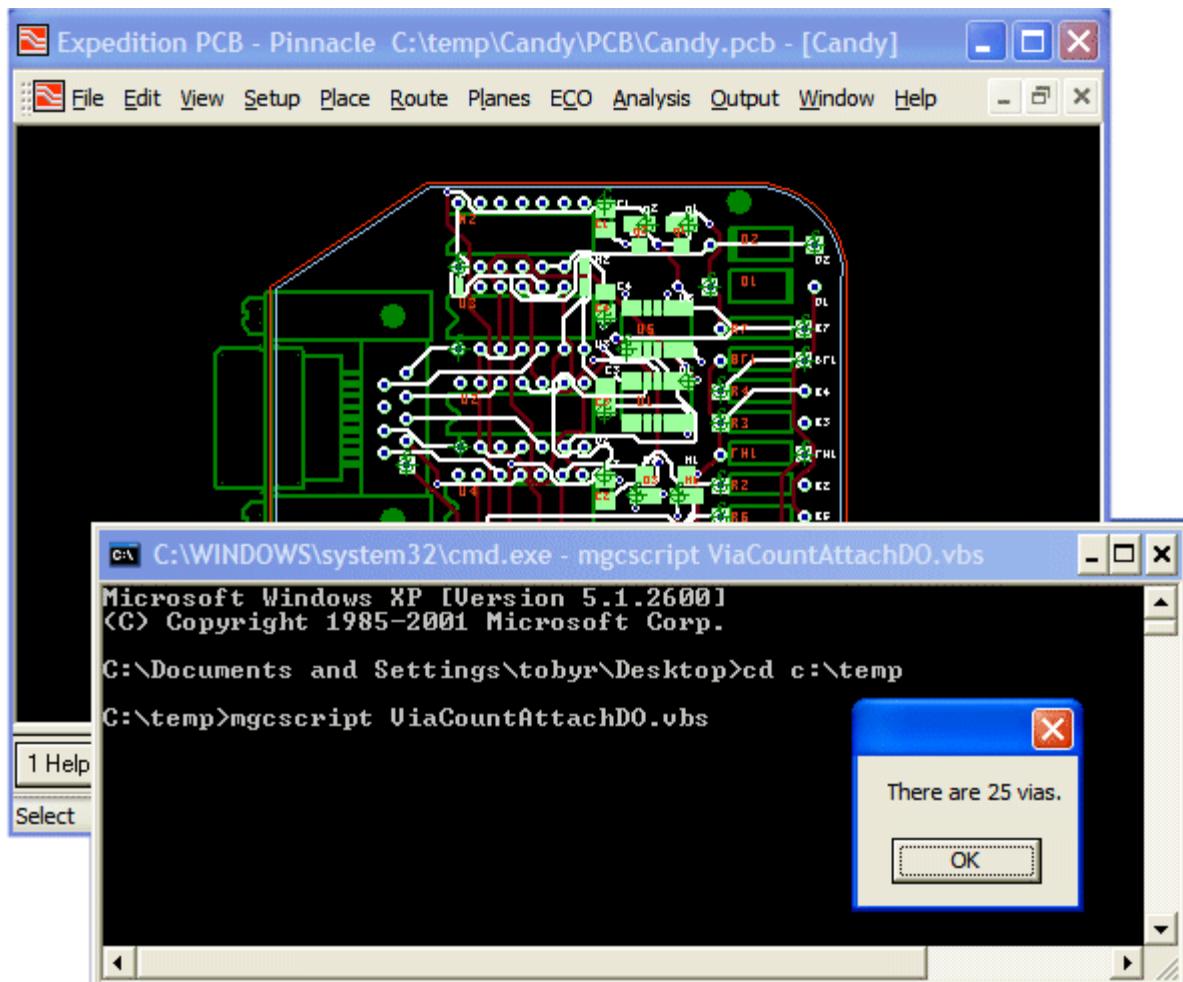


Figure 8-25. Attaching to a running application with an open document.

Attaching (Application Open, Document Closed)

Use the scripting editor of your choice to open the *ViaCountAttachDO.vbs* file you created in the previous exercise and save it out as *ViaCountAttachDC.vbs*, where the "DC" portion of this filename stands for "Document Closed". This script will initially appear as follows:

```

1
2 Option Explicit
3
4 ' Add any type libraries to be used.
5 Scripting.AddTypeLibrary( "MGCPBC.ExpeditionPCBAplication" )
6
7 ' Get the Application object
8 Dim pcbAppObj
9 'Set pcbAppObj = Application
10 Set pcbAppObj = GetObject(, "MGCPBC.ExpeditionPCBAplication" )
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.

```

Let's assume that the layout document in which we are interested is called *Candy.pcb*, and that it's located in your *C:\Temp\Candy\PCB* folder (see also *Chapter 9* for details regarding more typical locations of such documents). Comment out the original Line 14, insert a new Line 15 as shown below, and save and close the file.

```

14 'Set pcbDocObj = pcbAppObj.ActiveDocument
15 Set pcbDocObj =
   pcbAppObj.OpenDocument( "C:\Temp\Candy\PCB\Candy.pcb" )

```

Note that this new statement should all be on one line, it's shown as being split over two lines here for the purposes of formatting. As we see, this new Line 15 explicitly opens our *Candy.pcb* layout document (see also the notes below).

- 1) Close any instantiations of Expedition PCB that are currently running, and then launch a new instantiation of this application.
- 2) Do NOT open a layout document.
- 3) Open a Console/Terminal window, set the context to *C:\Temp*, and prepare to run your *ViaCountAttachDC.vbs* script as shown in Figure 8-26 (don't actually run it yet, just prepare yourself for the excitement to come).

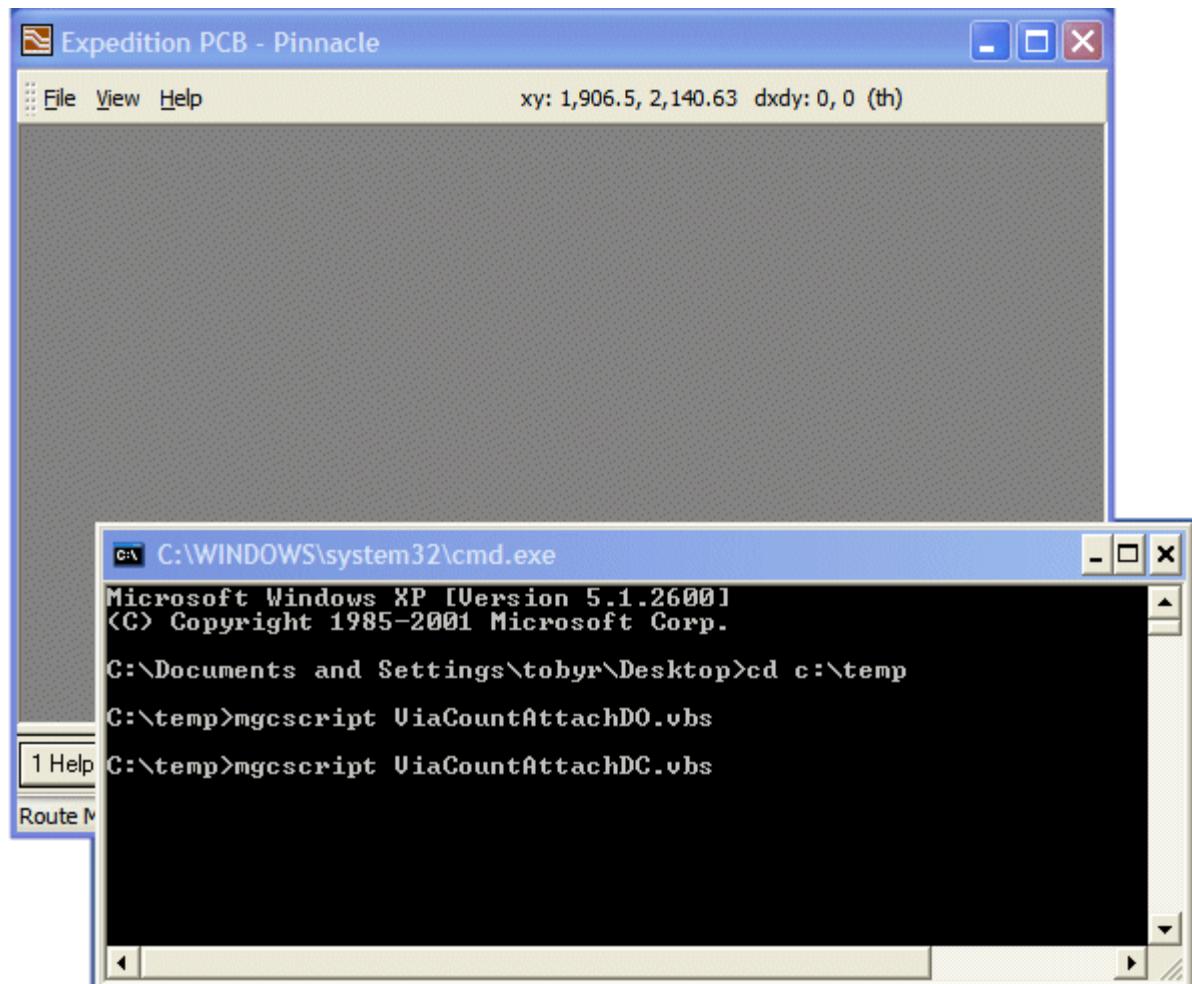


Figure 8-26. Preparing to run the script.

- 4) Press the <Enter> key to run the script. Observe that the script opens the *Candy.pcb* layout document and counts the vias as illustrated in Figure 8-27.

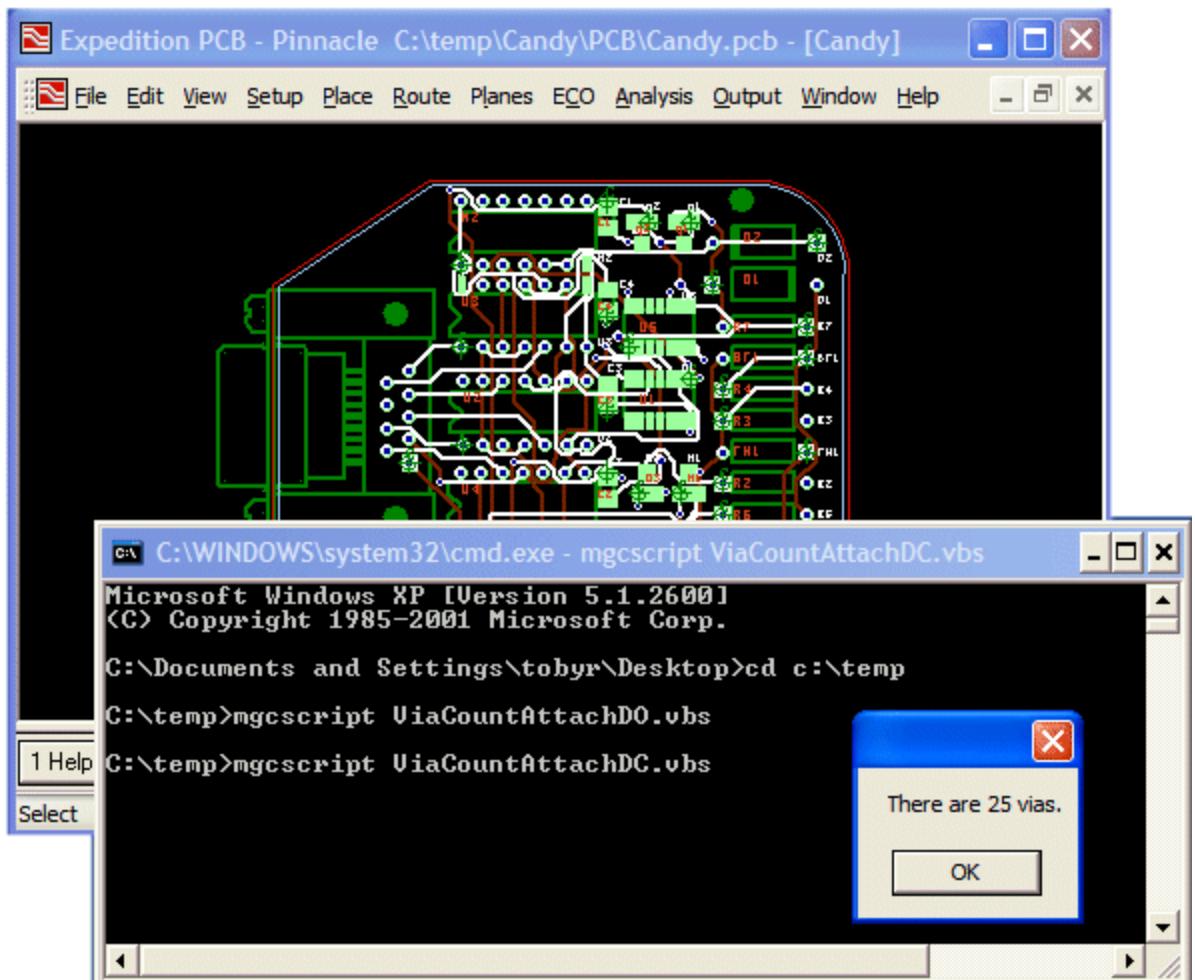


Figure 8-27. Attaching to a running application and opening a document.



Note: There is an easy way by which your script can test to see whether or not a document is already open. As you will recall, Lines 14 and 15 in your script are currently as follows:

```

12  ' Get the active document
13  Dim pcbDocObj
14  'Set pcbDocObj = pcbAppObj.ActiveDocument
15  Set pcbDocObj =
    pcbAppObj.OpenDocument( "C:\Temp\Candy\PCB\Candy.pcb" )

```

Let's start by un-commenting Line 14, which means that this statement will access the active document if there is such a beast. Next, insert a new Line 15 before the existing Line 15 (so your original line 15 now becomes Line 16) followed by a new Line 17 as shown below:

```

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

```

On our new Line 15 we perform a test by comparing our *Document* object to the keyword *Nothing*. This keyword equates to a null object and is often used for this type of comparison (it 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), this test will pass and the statement on Line 16 will be used to explicitly open the document as we previously discussed. (If you do perform these modifications, for goodness sake save the modified file with a different filename, otherwise chaos will ensue because we're going to want to reuse the original version in the not-too-distant future.)



Note: As opposed to explicitly specifying the document to be opened in the script as presented above, you could pass the name of the document into the script in the form of an argument as was discussed earlier in this chapter (this would be a good experiment for you to perform on your own to see if you actually understand all of this as much as you think you do).

Creating Invisible (Application Closed, Document N/A)

Use the scripting editor of your choice to open the *ViaCountAttachDC.vbs* file you created in the previous exercise and save it out as *ViaCountCreateInvisible.vbs*. This script will initially appear as follows:

```
1  Option Explicit
2
3
4  ' Add any type libraries to be used.
5  Scripting.AddTypeLibrary( "MGCPBC.ExpeditionPCBApplication" )
6
7  ' Get the Application object
8  Dim pcbAppObj
9  'Set pcbAppObj = Application
10 Set pcbAppObj = GetObject(, "MGCPBC.ExpeditionPCBApplication" )
11
12 ' Get the active document
13 Dim pcbDocObj
14 Set pcbDocObj =
15     pcbAppObj.OpenDocument( "C:\temp\Candy\PCB\Candy.pcb" )
16
17 ' License the document
18 ValidateServer(pcbDocObj)
19
20 ' Get the vias collection
21 Dim viaColl
22 Set viaColl = pcbDocObj.Vias
23
24 ' Get the number of vias in collection
25 Dim countInt
26 countInt = viaColl.Count
27
28 MsgBox( "There are " & countInt & " vias." )
29
30 ' Local functions
31
32 ' Server validation function
33 Private Function ValidateServer(docObj)
34     :
35     etc.
```

Observe the current Line 10 in which we use the *GetObject* function to attach to an instantiation of the application that is already running as shown below (remember that the *Prog ID* is the second parameter to this function):

```
10 Set pcbAppObj = GetObject(, "MGCPBC.ExpeditionPCBApplication" )
```

Edit this line such that it now calls the *CreateObject* function to launch an instantiation of the application as shown below (observe that the *Prog ID* is now the first parameter to this function):

```
10 Set pcbAppObj = CreateObject("MGCPCB.ExpeditionPCBAApplication")
```



Note: If you already have one or more instantiations of the application running when you use the *CreateObject* function, you will end up launching an additional instantiation. If you want to test whether or not an instantiation of the application is already running before calling the *CreateObject* function, you could try first using the *GetObject* function in conjunction with the *On Error Resume Next* statement as discussed in Chapter 2: VBS Primer.

Theoretically, the change to Line 10 shown above is the only modification we need to make to our script; however, there is something else we need to think about. By default, when a script uses the *CreateObject* function to launch an application, that application will not be visible on the screen (you will understand why we don't show you a screenshot of this).

This may be the preferred scenario in many cases – such as batch processing and analysis – because having applications suddenly appearing on the screen can be somewhat distracting if you're not expecting it.

One consideration, however, is that when the script terminates, the (invisible) application will continue to run in the background. Furthermore, since – by definition – this (invisible) application doesn't present a Graphical User Interface (GUI), there is no easy way to close the application. The only solution in this case would be to use the operating system to "Kill" the application process (the actual mechanism behind this is dependent on the operating system being used).

The preferred solution in this case would be for the script to explicitly release any objects it's used from the application and to then terminate the application. Now, in order to release an object, you have to set the variable to which the object was originally assigned to the *Nothing* keyword.

Observe the current Line 27, which contains the last statement in the body of our script prior to the validation function:

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

It is considered to be good programming practice to close everything down *before* displaying the result, so 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.")
```

First, on Line 28, we release the *viaColl* variable by setting it to the *Nothing* keyword. Next, on Line 31, we use the *Close* method on the *Document* object to close the document (this is

equivalent to our using the **File > Close** command in the GUI). Even after the document has been closed, we still need to release the *Document* object by setting its variable (*pcbDocObj*) to the *Nothing* keyword as seen on Line 33.

Finally, on Line 36 we use the *Quit* method on the *Application* object to close the application (this is equivalent to our using the **File > Exit** command in the GUI). As usual, even after the application has been terminated, we still need to release the *Application* object by setting its variable (*pcbAppObj*) to the *Nothing* keyword as seen on Line 38.

Once you've made the above changes, save them, close the script, and exit the script editor (just to make sure).

- 1) Ensure that you don't have any instantiations of Expedition PCB running.
- 2) Open a Console/Terminal window, set the context to your C:\Temp folder, run your *ViaCountCreateInvisible.vbs* script, and observe the results as shown in Figure 8-28.

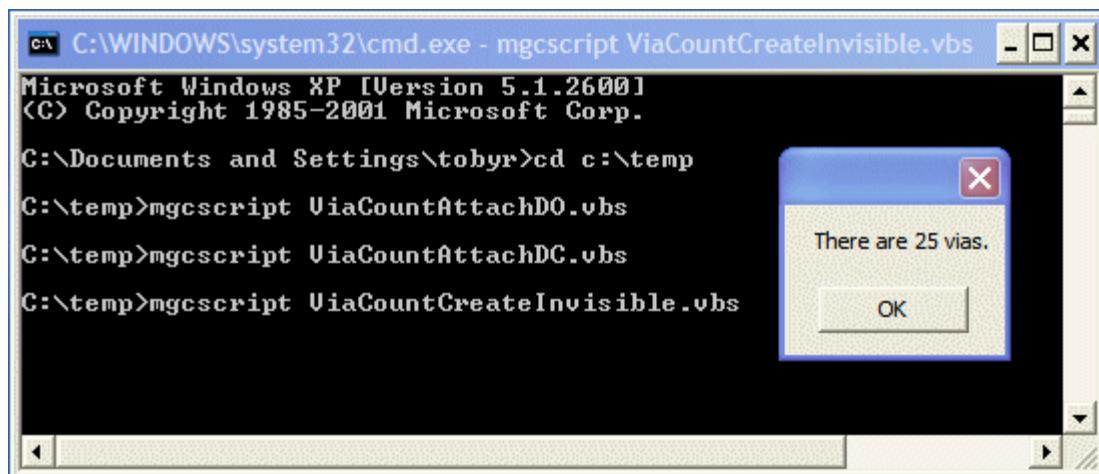


Figure 8-28. Running a script that automatically launches the application.

- 3) Observe that the Expedition PCB GUI does *not* appear (which is, of course, exactly what we expected to happen).
- 4) Click the **OK** button to dismiss the *MsgBox()* dialog and terminate the script.

Creating Visible (Application Closed, Document N/A)

Use the scripting editor of your choice to open the *ViaCountCreateInvisible.vbs* file you created in the previous exercise and save it out as *ViaCountCreateVisible.vbs*. This script will initially appear as follows (see the note following the script with regard to Lines 27 through 38):

```
1 Option Explicit
2
3
4 ' Add any type libraries to be used.
5 Scripting.AddTypeLibrary( "MGCPCB.ExpeditionPCBAplication" )
6
7 ' Get the Application object
8 Dim pcbAppObj
9 'Set pcbAppObj = Application
10 Set pcbAppObj = CreateObject( "MGCPCB.ExpeditionPCBAplication" )
11
12 ' Get the active document
13 Dim pcbDocObj
14 Set pcbDocObj =
```

```

15         pcbAppObj.OpenDocument( "C:\temp\Candy\PCB\Candy.pcb" )
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 .....
43 'Local functions
44
45 ' Server validation function
46 Private Function ValidateServer(docObj)
        :
        etc.

```

First of all, you can delete Lines 27 through 38 (which is why they are shown as being crossed out in the script above). As you will recall, we only added these as part of the process we used to terminate the application when it was launched without a GUI. In the case of this experiment, however, we are going to make the application visible and display its GUI, in which case we almost certainly don't want to close the application down using the script. Otherwise the application would flicker on and off the screen in a very annoying manner (feel free to experiment with this at your leisure).

Now, in order to launch the application *and* display its GUI, all that is required is to set the *Visible* property on the *Application* object to *True*. This is illustrated by inserting three new Lines just before the existing Line 12 as shown below (the additions are shown in **bold**):

```

8 Dim pcbAppObj
9 'Set pcbAppObj = Application
10 Set pcbAppObj = CreateObject( "MGCPCB.ExpeditionPCBApplication" )
11
12 ' Make the application visible
13 pcbAppObj.Visible = True
14
15 ' Get the active document
16 Dim pcbDocObj

```

- 1) Ensure that you don't have any instantiations of Expedition PCB running.

-
- 2) Open a Console/Terminal window, set the context to your *C:\Temp* folder, run your *ViaCountCreateVisible.vbs* script, and observe the results.
 - 3) Click the **OK** button to dismiss the *MsgBox()* dialog and terminate the script.
 - 4) Use the **File > Close** command to close the layout document and then use the **File > Exit** command to terminate the application.

Chapter 9: Running a Script from Within an Application

Introduction

There are a variety of ways by which a script can be run from within an application such as Expedition PCB; the main ones are as follows:

- 1) Using the key-in command field (this was described in detail in *Chapter 3: Running Your First Script*).
- 2) Tying a script to a menu button/item or to a tool bar icon (these techniques were presented in *Chapter 6* and *Chapter 7*, respectively).
- 3) Automatically executing a script when the application is launched or when a document is opened (these *startup scripts* are discussed in *Chapter 10: Running Startup Scripts*).
- 4) Using the mouse to drag a script file from a folder (directory) – or from the desktop – and dropping it on top of an already open document as illustrated in Figure 9-1.

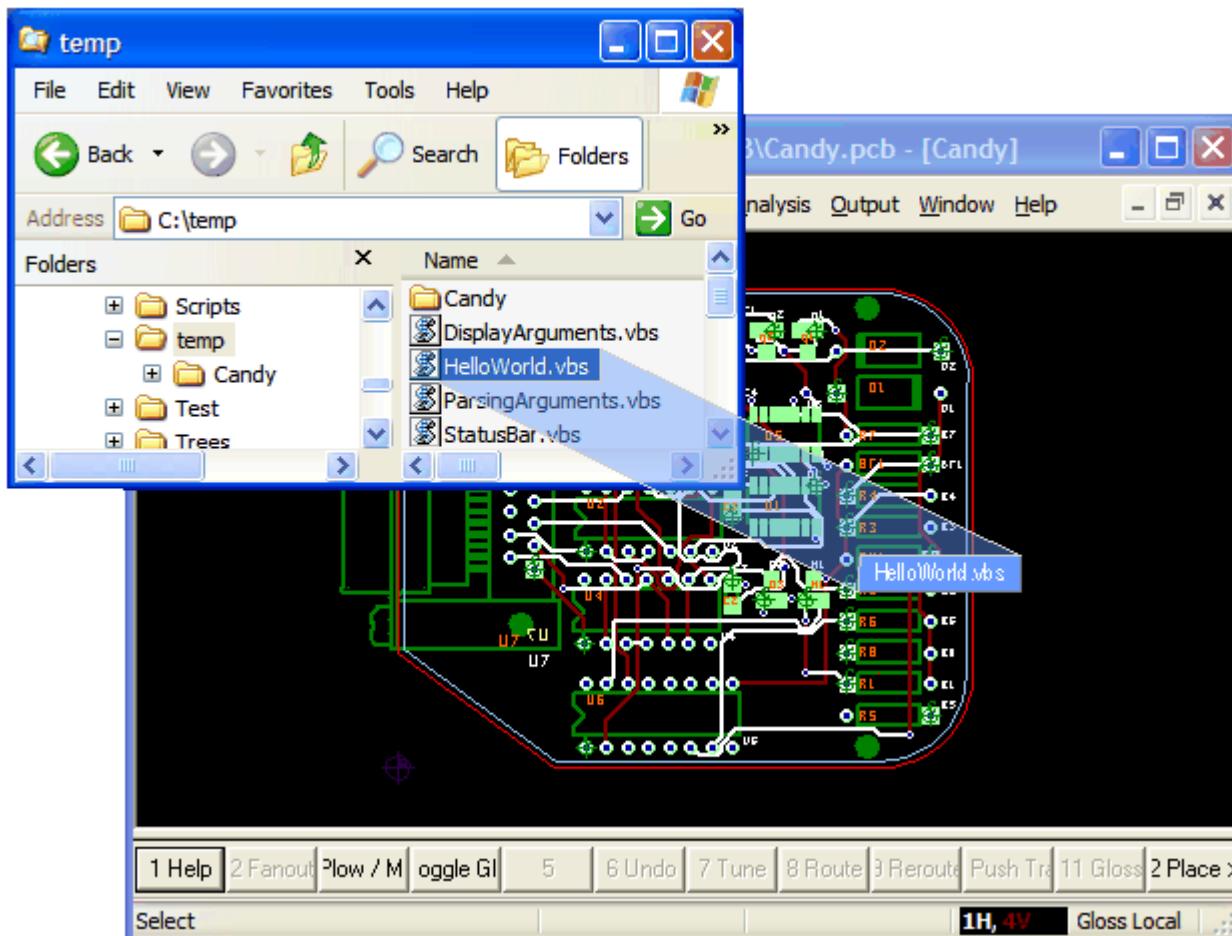


Figure 9-1. Running a script using a drag-and-drop technique.

Using the WDIR Environment Variable

In all of our previous examples we have stored our scripts in the C:\Temp folder. This has obliged us to specify the entire path – including this folder – when running these scripts.

In fact, when the full path is not specified as part of a script's filename, an application such as Expedition PCB will automatically look in a number of different folders on the system in the following order:

- 1) **The 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 (see also *Chapter 10: Running Startup Scripts*).
- 2) **The Design Area:** In the case of Expedition PCB, this is the folder containing the currently open *.pcb layout design document. If no document is open, then the application will not have this context and this option will be excluded from the search.
- 3) **User-Defined Areas:** These are folders that are specified using the WDIR environment variable.



Note: The WDIR Environment variable allows multiple folders to be defined. These are separated by semi-colons (;) when running under the Windows® operating system and by colons (:) when running under the UNIX and Linux platforms.



Note: As soon as a script with the correct name is discovered in any of the above folders, the search will terminate and that script will be run.

Creating/Modifying the WDIR Environment Variable

For the purposes of this portion of our discussions, we shall assume that we are running under the Windows® operating system.

- 1) Click the **Start** button, right-click **My Computer**, and select the **Properties** item from the resulting pop-up menu as illustrated in Figure 9-2.



Figure 9-2. Accessing the System Properties dialog.

- 2) Observe the resulting **System Properties** dialog as illustrated in Figure 9-3.

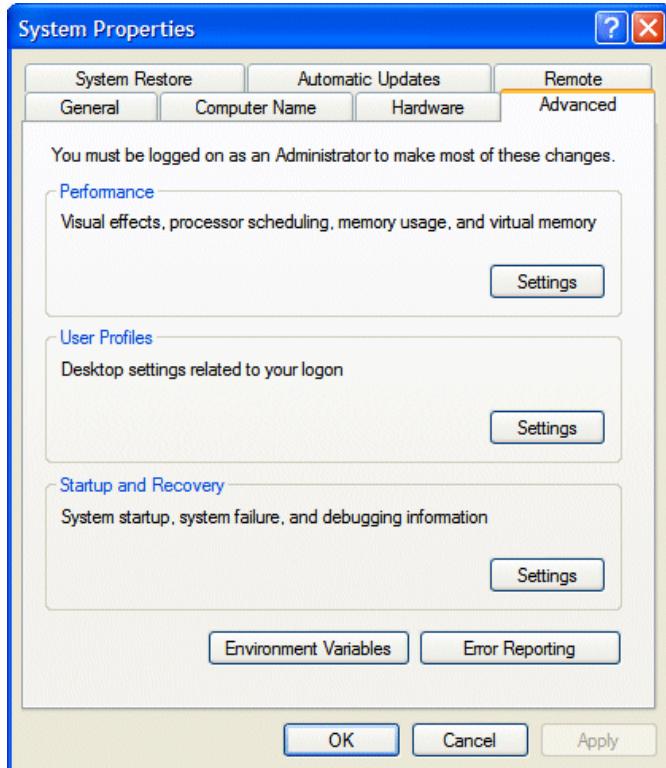


Figure 9-3. The System Properties dialog.

- 3) Select the **Advanced tab** and click the **Environment Variables** button to access the **Environment Variables** dialog as illustrated in Figure 9-4.

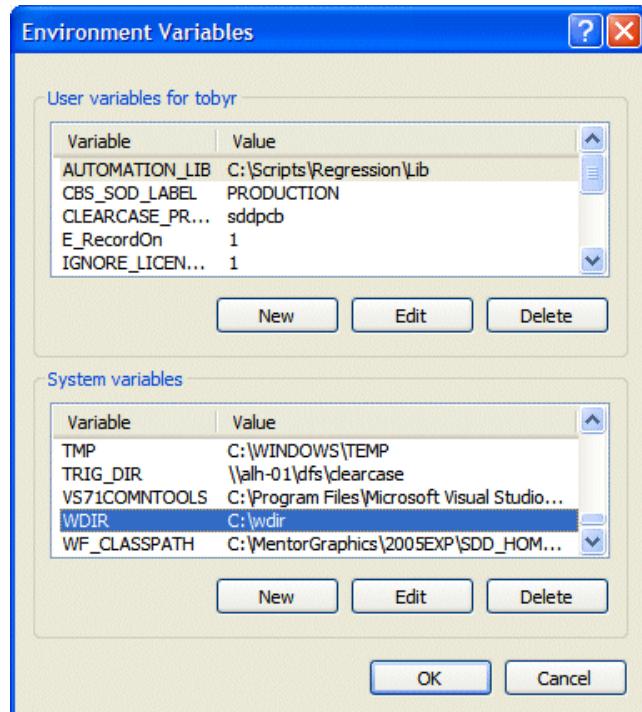


Figure 9-4. The Environment Variables dialog.

- 4) If an application's installation has already configured a *WDIR* variable for you, it will be located in the **System Variables** scrolling list as illustrated in Figure 9-4. If you want to add a new folder – such as our *C:\Temp* folder – to an existing *WDIR* variable, then select (highlight) the *WDIR* item in the scrolling list and click the **Edit** button to access the **Edit System Variable** dialog as illustrated in Figure 9-5.

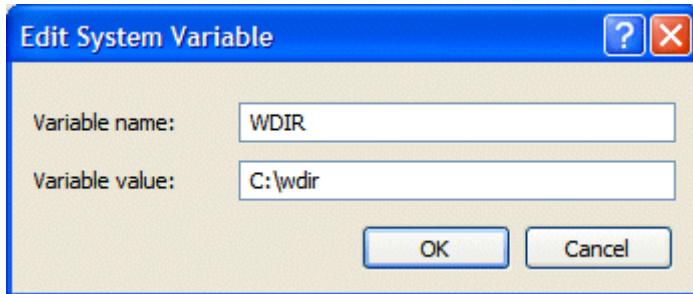


Figure 9-5. The Edit System Variable dialog.

- 5) Add the new path(s) to the **Variable Value** field as illustrated in Figure 9-6.

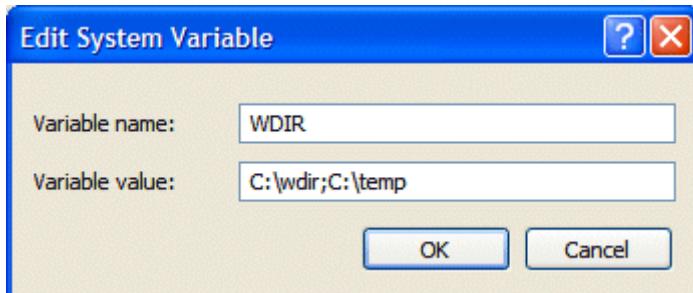


Figure 9-6. Editing an existing *WDIR* variable.

- 6) Before we proceed, we should note that if there had not been an existing *WDIR* variable as illustrated in Figure 9-4, then we would have clicked the **New** button on the **Environment Variables** dialog, which would have returned the New System Variable dialog (Figure 9-7).

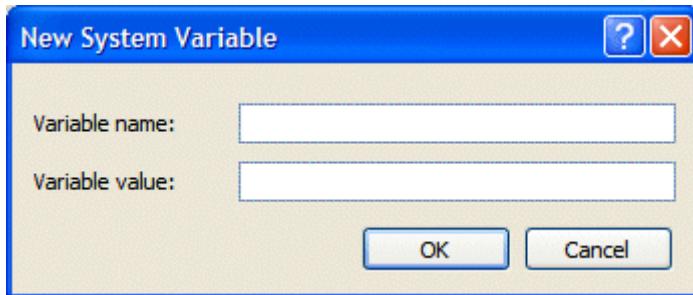


Figure 9-7. The New System Variable dialog.

- 7) In this case, enter *WDIR* in the **Variable name** field and then type in the path(s) you want to add in the **Variable value** field as illustrated in Figure 9-8.

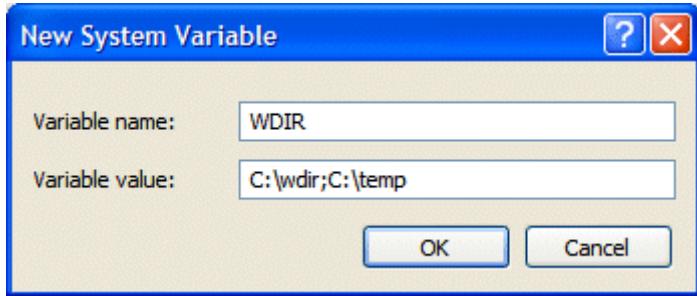


Figure 9-8. Creating a new WDIR variable.

- 8) Click the **OK** button to exit the **Edit System Variable** dialog (or the **New System Variable** dialog); then click the **OK** button to exit the **Environment Variables** dialog; and finally click the **OK** button to exit the **System Properties** dialog.
- 9) Close any currently open instantiations of Expedition PCB.
- 10) Launch a new instantiation of Expedition PCB.
- 11) Right-click on the toolbar and ensure that the **Keyin Command** option is ticked (enabled).
- 12) If you cast your mind back to *Chapter 3: Running Your First Script*, you will recall that you had to enter the string *run C:\Temp\HelloWorld.vbs* into the **Keyin Command** field in order to run your *HelloWorld.vbs* script (Figure 9.9).

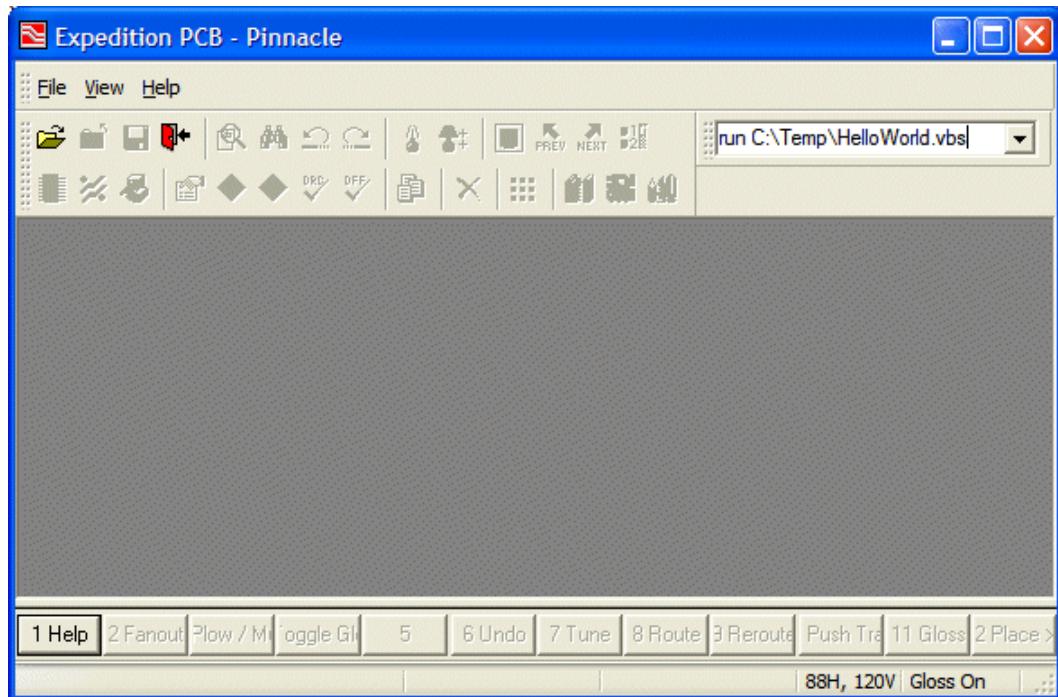


Figure 9-9. The way we used to do things (specifying the entire path).

Now that you've added the *C:\Temp* folder to your *WDIR* environment variable, you can simply enter the string *run HelloWorld.vbs* into the **Keyin Command** field (actually performing this task and observing the results will be left to the reader).



Note: In addition to absolute paths, it is possible to specify relative paths with regard to any folder defined by the *WDIR* environment variable. For example:

- a) Assume that your *WDIR* environment variable already "points" to the *C:\Temp* folder.
- b) Further assume that you have a folder called *Test* under your *C:\Temp* folder (that is, you have a folder called *C:\Temp\Test*).
- c) Further assume that you have a script called *MyTest.vbs* in your *Test* folder. In this case, entering either of the following two strings **Keyin Command** field will run your *MyTest.vbs* script:

run <i>C:\Temp\Test\MyTest.vbs</i>	{ Absolute }
run <i>\Test\MyTest.vbs</i>	{ Relative }



Note: You can also use environment variables as part of the key-in path; for example:

run *%HOME%\MyTest.vbs*

Once again, the above assumes that we are running under the Windows® operating system; the equivalent when running under the UNIX or Linux operating systems would be as follows:

run *\$HOME/MyTest.vbs*

Chapter 10: Running Startup Scripts

Introduction

As their name would suggest, "startup scripts" are scripts that are automatically run when an application is first launched and/or when a document is opened (for example, a layout design document in the case of Expedition PCB).

Introducing `scripts.ini` Files

The scripts to be run at startup are specified in a special `scripts.ini` file. The format of this file will be discussed in the following sections; for the moment, we need only be aware that the file can contain multiple lists of scripts, where each list is associated with a particular application. For example, there could be one list of scripts associated with Expedition PCB, another list of scripts associated with FabLink XE, and so forth.

In fact, there can be multiple `scripts.ini` files, but only one such file in each of the default folders. The order in which the default folders are searched is as follows:

- 1) **The 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` (see also *Chapter 9: Running a Script from Within an Application*).

This folder, which is automatically established when you install your Mentor products, is the preferred location for your `script.ini` file when the products are installed on a server and are accessible by multiple users, all of whom want to use the same startup scripts. (See also the note below.)

- 2) **The Design Area:** In the case of Expedition PCB, this is the folder containing the currently open `*.pcb` layout design document. By comparison, in the case of FabLink XE, this is the folder containing the `*.pn1` document. There is a separate design area associated with each design. (See also the note below).
- 3) **User-Defined Areas:** These are folders that are specified using the `WDIR` environment variable (the process of creating this variable and modifying the folders associated with it was detailed in *Chapter 9: Running a Script from Within an Application*).

This is the most common and flexible option when it comes to running one's personal startup scripts.



Note: Even after the application has located and processed its first `scripts.ini` file, it will continue to search all of the remaining default folders (and process any `scripts.ini` files it finds there) in the order described above.



Note: In the majority of cases, it is desired that startup scripts will be run when an application is launched (these are typically referred to as "application startup scripts"); in some cases, however, it is required to run startup scripts when a design document is opened (these are typically referred to as "document startup scripts"). Both of these scenarios are discussed in more detail later in this chapter.

The point is that `scripts.ini` files stored in the design areas discussed above are only applicable in the case of startup scripts that are run when a design document is opened. When the application is first launched, it will not process any `script.ini` files in any of the design areas, because it doesn't have any context until a design document is actually opened.

Startup Scripts when an Application is Launched

In the case of startup scripts that are to be run when an application is launched, the format of the *scripts.ini* file 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.
```



Note: As was discussed earlier, a *script.ini* file can contain multiple lists of scripts, where each list is associated with a particular application. The application names as they are to be used in *script.ini* files are as follows:

Product	Key (name) used in <i>scripts.ini</i> file
Expedition PCB	Expedition PCB
Board Station RE	Board Station RE
Board Station XE	Board Station XE
Viewer PCB	Discovery PCB { See *note }
Planner PCB	Discovery PCB { See *note }
FabLink XE	Fablink XE Pro
ICX Pro Verify	ICX Pro Verify
SFX RE	SFX RE
DxDesigner	ViewDraw { See **note }
CES	Workbench { See **note }

* All of the startup scripts associated with both the *ViewerPCB* and *PlannerPCB* executables must be specified under a single *Discovery PCB* application list in the *scripts.ini* file.

** These products use a slightly different format and search different product areas. Please see the automation documentation for these products for further details.

When an application is launched it will search for *scripts.ini* files as discussed above. For each *scripts.ini* file, the application will look for the list of scripts associated with its own key (name) and it will then process those scripts.



Note: Once an application locates an associated list of startup scripts in the *scripts.ini* file, these scripts will be processed in the order in which they are defined. This may be important in certain cases; for example, two scripts that each add a new menu button/item to the "end" of a pull-down menu. In this case, the relative positioning of the new buttons/items in the final menu will depend on the order in which these scripts are specified and processed.

Furthermore, the *script.ini* parser (reader) automatically starts at *Script#0* and proceeds until it detects a break in the sequence. Thus, it is very important for the first script to be associated with *Script#0*, and also to ensure that there are no breaks in the sequence (which might occur if you delete an entry from the *scripts.ini* file). For example, consider the following:

```
[<Application One>]
Script#1=<script path>
Script#2=<script path>
Script#3=<script path>
:
etc.

[<Application Two>]
Script#0=<script path>
Script#1=<script path>
Script#3=<script path>
:
etc.
```

In this case, none of the scripts associated with *Application One* would run, because this list doesn't start with *Script#0*. Meanwhile, in the case of *Application Two*, only *Script#0* and *Script#1* would be processed; following *Script#1*, the parser would detect a break in the sequence, and would therefore cease processing any further scripts associated with this list.

- 1) Create a folder called *Startup* under your *C:\Temp* folder.
- 2) Use a text editor of your choice to open a new / empty file and enter the following:

```
[Expedition PCB]
Script#0=C:\Temp\ViaCountMenuSingle.vbs
```

As you may recall, the *ViaCountMenuSingle.vbs* script is one that you previously created in *Chapter 6: Adding a Menu and/or Menu Button/Item*. This script adds a new **Via Count** button/item to the end of the **View** pull-down menu.



Note: Observe that we specified the full path for the script in the above example. If we were to add the *C:\Temp* folder to the *WDIR* environment variable (this was discussed in *Chapter 9: Running a Script from Within an Application*), then we could use 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 your new *C:\Temp\Startup* folder to its list of folders (again, this was discussed in *Chapter 9: Running a Script from Within an Application*).
- 5) Make sure no instantiations of Expedition PCB are currently running, and then launch a new instantiation of Expedition PCB.
- 6) Open a layout design document of your choice, then click the **View** pull-down menu and observe that your **Via Count** button/item has been automatically added to the end of the list as illustrated in Figure 10-1.

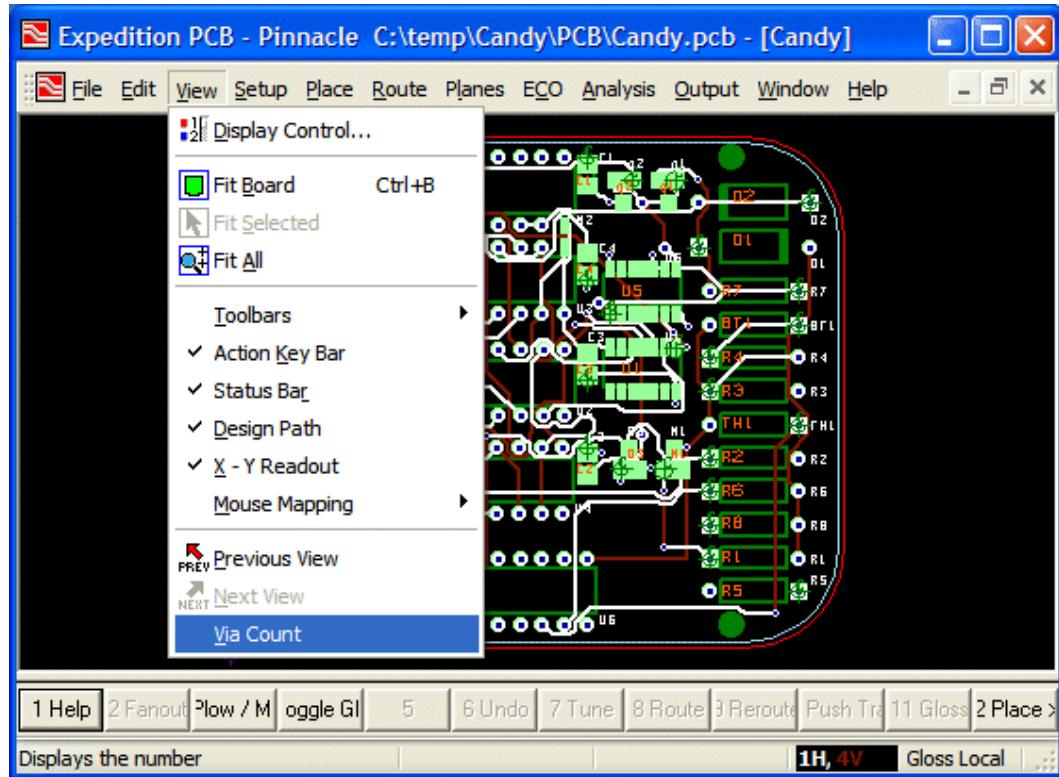


Figure 10.1. The new Via Count button/item.

Startup Scripts when a Document is Opened

As was previously noted, it may be required for some startup scripts to be run only when a document is opened. In this case, these scripts are specified in a separate section of the *scripts.ini* file. As before, the *scripts.ini* file can contain multiple lists of scripts, where each list is associated with a particular application. For example, there could be one list of scripts associated with Expedition PCB, another list of scripts associated with FabLink XE, and so forth. These *document startup scripts* are distinguished from their *application startup script* counterparts by appending the " – Document" qualifier to the application name.

In order to provide a simple example, let's assume that we want to run our original *ViaCount.vbs* script whenever a layout design document is opened in Expedition PCB (you created this script as part of *Chapter 3: Running Your First Script*):

- 1) Use a text editor of your choice to open the *C:\Temp\Startup\scripts.ini* file you created in the previous section. This file, which contained only an *application startup script*, should look like the following:

```
[Expedition PCB]
Script#0=C:\Temp\ViaCountMenuSingle.vbs
```

- 2) Now, add a document startup script as follows and then re-save and exit this file:

```
[Expedition PCB]
Script#0=C:\Temp\ViaCountMenuSingle.vbs
```

```
[Expedition PCB - Document]
Script#0=C:\Temp\ViaCount.vbs
```

- 3) Make sure no instantiations of Expedition PCB are currently running, and then launch a new instantiation of Expedition PCB.
- 4) Open a layout design document of your choice and observe that the *ViaCount.vbs* script is automatically run as soon as the document is opened as illustrated in Figure 10-2.

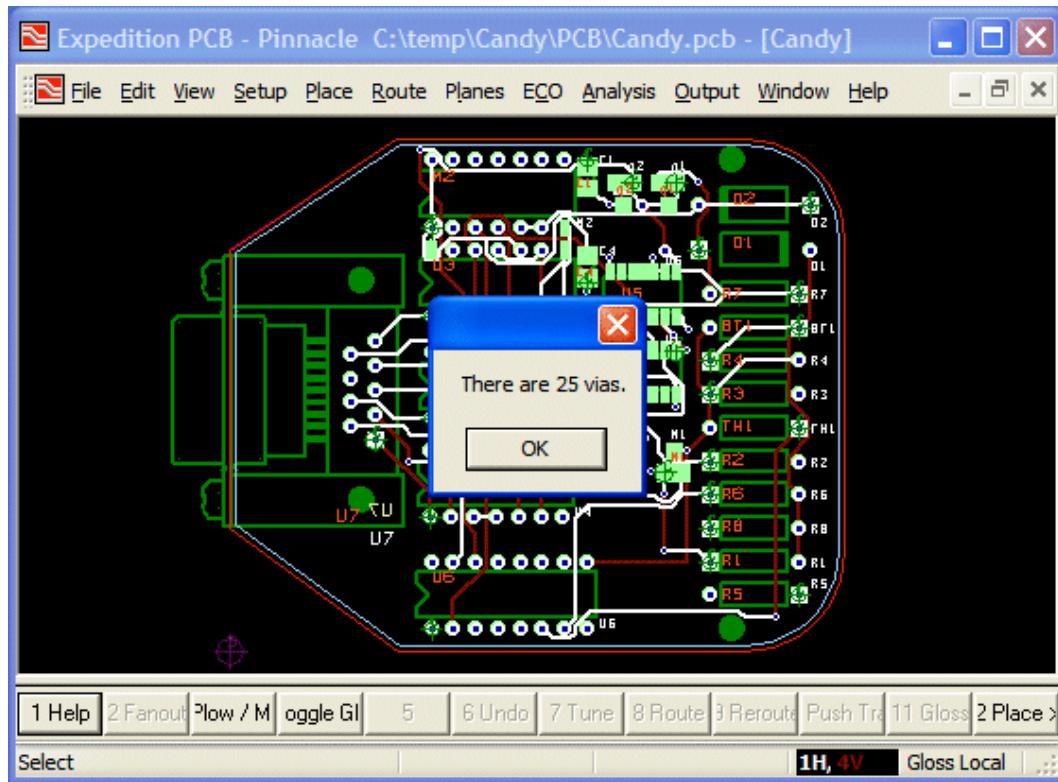


Figure 10.2. ViaCount.vbs is automatically run when the document is opened.

- 5) Click the **OK** button to dismiss the `MsgBox()`.
- 6) Use the **File > Close** command to close the document followed by the **File > Exit** command to exit Expedition PCB.
- 7) Rename the `scripts.ini` file in your `C:\Temp\Startup` folder to `scripts-(do-not-run).ini`. If you didn't do this, the *ViaCount.vbs* script would automatically run every time you opened a layout design document in the future, and this can become very annoying very quickly.

Debugging Startup Scripts

In some cases, you may want to debug your startup scripts by either (a) seeing a list of which scripts were actually run or (b) but not running them at all. This can be accomplished by launching Expedition PCB from a Console/Terminal window and specifying the command line options `-dbgstart` or `-nostart`, respectively.

See the *Expedition Automation Help* for more details on how to use these options and techniques.

Chapter 11: Introducing Events and the Scripting Object

Introduction

With the exception of the mechanisms underlying the use of menu buttons/items and tool bar icons as discussed in Chapters 6 and 7, respectively, our *ViaCount.vbs* script (in all of its many incarnations) has simply extracted information from the application (Expedition PCB in these examples) and processed this information using methods and properties defined in the automation interface.

In some cases, however, we would like the application to notify our scripts when specific occurrences take place; for example:

- When a document is opened or closed (or prepares to close).
- When the application exits (or prepares to exit).
- When an object selection changes (something is selected or deselected).
- When changes are made to the design (moving or modifying objects).
- Etc.

The way in which this works is that we define a function in a script that will act as an event handler. This function will be called (executed) by the application when a specific event occurs. As we shall see, the names of these event handler functions are – to a large extent – predefined.

The *Scripting* object has many uses. The two we've experienced thus far have been to add type libraries to scripts and to keep scripts from exiting. As we shall soon discover, one very important use of the *Scripting* object is to facilitate the use of events in scripts.

Events and In-Process Scripts

Use the scripting editor of your choice to enter the following script and then save it in your C:\Temp folder as *EventsInProcess.vbs*:

```
1
2 Option Explicit
3
4 ' Add any type libraries to be used.
5 Scripting.AddTypeLibrary("MGCPBCB.ExpeditionPCBAApplication")
6
7 ' Get the Application object
8 Dim pcbAppObj
9 Set pcbAppObj = Application
10
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 *****
26 ' Event handlers
27
```

```

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
38
39  .....
40  'Local functions
41
42  ' Server validation function
43  Private Function ValidateServer(docObj)
        :
        etc.

```

The first portion of this script – Lines 1 through 16 – involve accessing the application, accessing the design document, and licensing the document in a similar manner to our previous scripts. The only point of interest with regard to these discussions occurs in Lines 11 through 13:

```

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

```

On Line 12 we instantiate a variable called *pcbDocObj* and on Line 13 we assign the active document to this variable. It is the name of this variable which is of interest, because – as we shall soon see – this will form part of the event handler's function name.

Now, before we proceed, it is important to understand that different types of events are associated with different objects. The most commonly used events are those associated with applications and documents. The events available for each type of object are described in detail in the Expedition Automation Help.

For example, a quick peek at the help shows that *Application* objects have two types of events associated with them – an *OpenDocument* event and a *Quit* event – as illustrated in Figure 11-1.

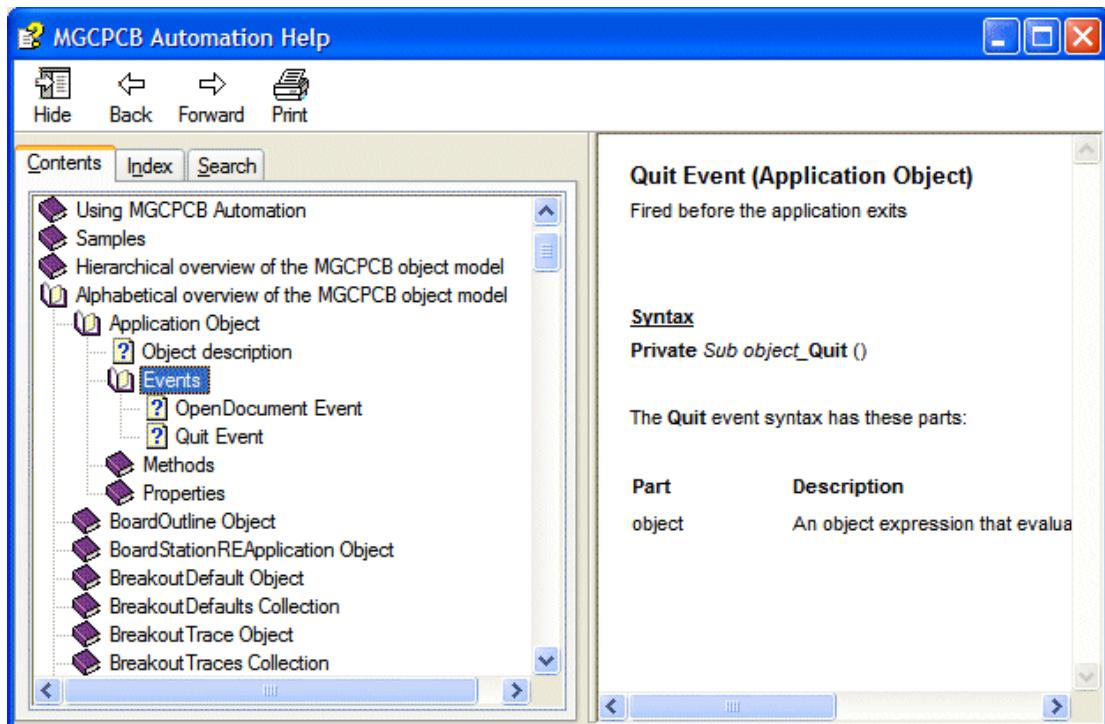


Figure 11-1. Using the automation help to research different types of events.

Now, 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 associated with the *Scripting* object to instruct the script engine that we are interested in any events associated with the object that is specified by the first parameter. In this case, we're using the *Document* object variable *pcbDocObj* as this parameter (this is the variable we instantiated on Line 12).

Observe that the second parameter is a string. This string will eventually be used as a common prefix for any of the event handler functions we declare in the future (that is, for any event handler functions associated with events originating from the object assigned to the *pcbDocObj* variable). The string itself is arbitrary – you could call it "Fred" if you wanted – however, it is recommended programming practice to set this string to be identical to the name of the first parameter. Thus, as the first parameter in our example is the *pcbDocObj* variable, we will use "pcbDocObj" as our string.

Next, 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
```

Observe the statement on Line 23. With the exception of the scripts associated with menu buttons/items and tool bar icons as discussed in Chapters 6 and 7, respectively, the scripts we've run in previous chapters have terminated once they've completed execution. This would be a problem in this case, because if our script terminates the event handler (which we declare later in the script) cannot be called. Thus, on Line 23 we set 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.

Finally, we set up the event handler functions for any events in which we are interested. For the purposes of this example, we are only going to consider the event that occurs just prior to a document closing. This is known as the *OnPreClose* event.



Note: Some events occur *after* an action has occurred, in which case it's too late to do anything about that action. By comparison, other events – such as *OnPreClose* – occur just *before* an action occurs; the document closing in this case. This type of event allows the script to control whether or not the action will be permitted to occur. In the case of this particular example – and depending on input from the user – the script may allow the document to close or it may prevent the document from closing. This form of allowing / disallowing is controlled by the return values associated with the event handlers, and events that don't have ability to change behavior would not return a value.

Now, 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 key to making sure this function gets fired. If either the name or the number of parameters differ in any way from the documentation, the event handler function will not get called.

In the case of this example, there are no parameters to be passed into the function. Meanwhile, the name of the function (*pcbDocObj_OnPreClose*) comprises three parts as follows:

- The string we previously specified as the second parameter to the *AttachEvents* method (this was the "pcbDocObj" string we used on Line 19).
- An underscore '_' character.
- The name of the event as defined in the Expedition Automation Help documentation, which is *OnPreClose* in this example.

On Line 30 we instantiate a variable called *ansEnum*, to which we are soon to assign a Yes/No answer. On Line 31 we see a new use of the *MsgBox()* function. In this case, we set the second parameter to the enumerated type *vbYesNo*. This is part of the VBScript language and will cause the *MsgBox()* to present the user with two buttons. One of these buttons will carry the annotation **Yes** – clicking this button will cause the *MsgBox()* function to return the enumerated type *vbYes*. By comparison, the other button will carry the annotation **No** – clicking *this* button will cause the *MsgBox()* function to return the enumerated type *vbNo*.

The value returned from the *MsgBox()* function [*vbYes* or *vbNo*] is assigned to our *ansEnum* variable. On Line 32 we compare the value represented by *ansEnum* to the enumerated type *vbYes*. If the user selected the **Yes** button, then the script allows the document to close by causing our *pcbDocObj_OnPreClose* event handler function to return a Boolean value of *True*. Alternatively, if the user selected the **No** button, then the script prevents the document from closing by returning a Boolean value of *False*.

- 1) Make sure no instantiations of Expedition PCB are currently running, and then launch a new instantiation of Expedition PCB.
- 2) Open a layout design document of your choice.

- 3) Drag the *EventsInProcess.vbs* script from your C:\Temp folder and drop it on top of the layout design document (this technique of running a script was discussed in *Chapter 9: Running a Script from Within an Application*).
- 4) Use the **File > Close** command to instruct Expedition PCB to close the layout design document.
- 5) Observe the *MsgBox()* with its **Yes** and **No** buttons appear on the screen as illustrated in Figure 11-2.

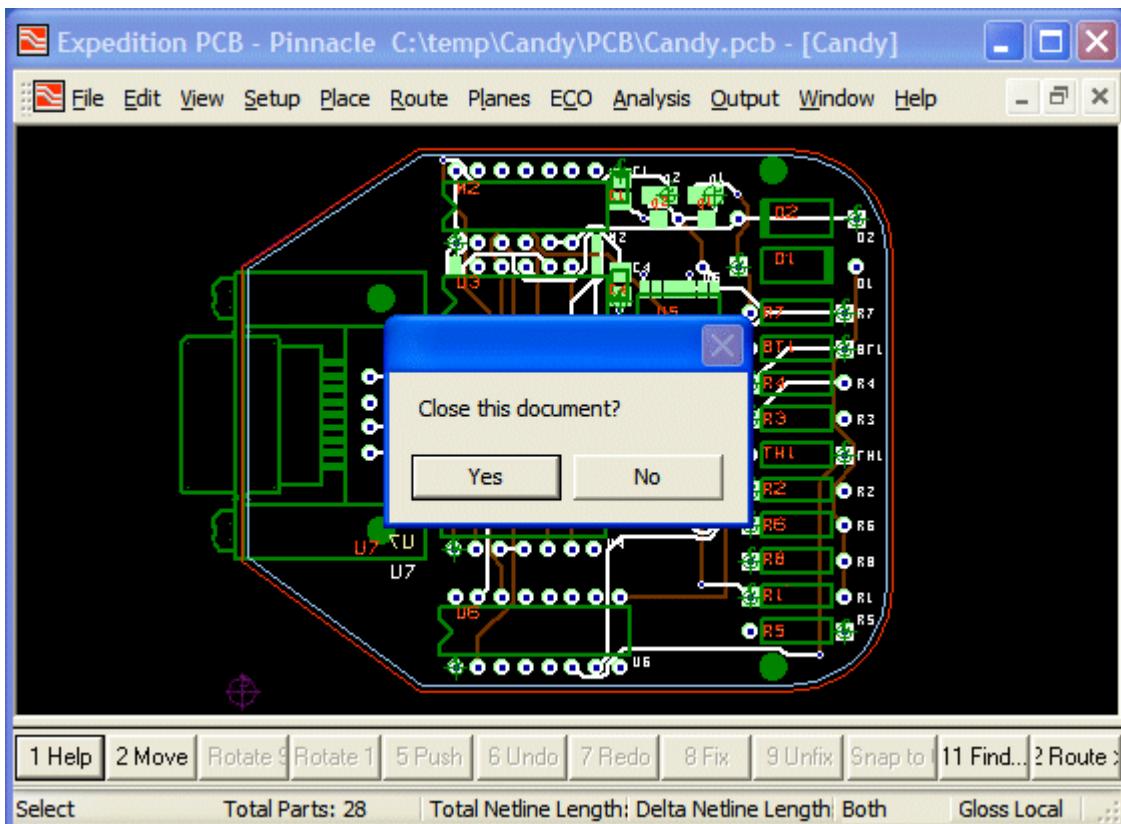


Figure 11-2. Using an event to invoke an in-process event handler script.

- 6) Click the **No** button and observe that the document does not close.
- 7) Once again, use the **File > Close** command to instruct Expedition PCB to close the layout design document.
- 8) Click the **Yes** button and observe that the document closes.



Note: It is common for script writers to talk about "sinking events". This refers to the process of receiving event notifications from a server and firing an event handler. When the event handler starts processing the event we say it is *sinking* the event.

Events and Out-Of-Process Scripts Using mgcscript.exe

Consider Lines 21 through 23 from our previous script:

```
21  ' Set the Scripting.DontExit property to
22  ' true to keep the script running.
23  Scripting.DontExit = True
```

On Line 23 we used the *DontExit* property of the *Scripting* object to prevent the script from exiting. (Remember that this does not mean the script will run for ever; just that it will continue to run until the current application session is terminated.)

This technique will also work if the script is run as an out-of-process script from within another application such as FabLink XE. However, this approach will not work if the script is run from the command line using a scripting engine such as *mgcscript.exe*, because – in this case – the scripting engine does not have the context of the application closing (the fact that the scripting engine is not running inside the application means that it doesn't understand when the application exits).

In order to address this problem, use the scripting editor of your choice to open the *EventsInProcess.vbs* script in your C:\Temp folder and save it as *EventsOutOfProcess.vbs*. Now replace the existing Lines 21 through 23 with the new Lines 21 through 26, which implement an infinite loop as shown below:

```
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
```

On Line 22 we instantiate a *dontExitBool* variable to which we assign a Boolean value of *True* on Line 23. Next, we enter an infinite loop that will cycle endlessly around as long as the *dontExitBool* variable is *True*.

Now, observe the statement on Line 25. This causes the script to "go to sleep" for 300 milliseconds every time we go around the loop. In turn, this keeps the script from consuming CPU cycles and frees the host computer to perform other tasks.

Next, we have to 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
```

As we see, this modification simply involves adding Line 37, which sets the *dontExitBool* variable to *False*. The idea here is that, when the document is eventually allowed to close, setting this variable to *False* will allow the infinite loop (Lines 24 through 26) to terminate, which – in turn – will allow the script to exit.

- 1) Make sure no instantiations of Expedition PCB are currently running and then launch a new instantiation of Expedition PCB.
- 2) Open a layout design document of your choice.
- 3) Open a Console/Terminal window, set the context to your C:\Temp folder, and run your *EventsOutOfProcess.vbs* script.
- 4) Use the **File > Close** command to instruct Expedition PCB to close the layout design document.
- 5) Observe the *MsgBox()* with its **Yes** and **No** buttons appear on the screen as illustrated in Figure 11-3.

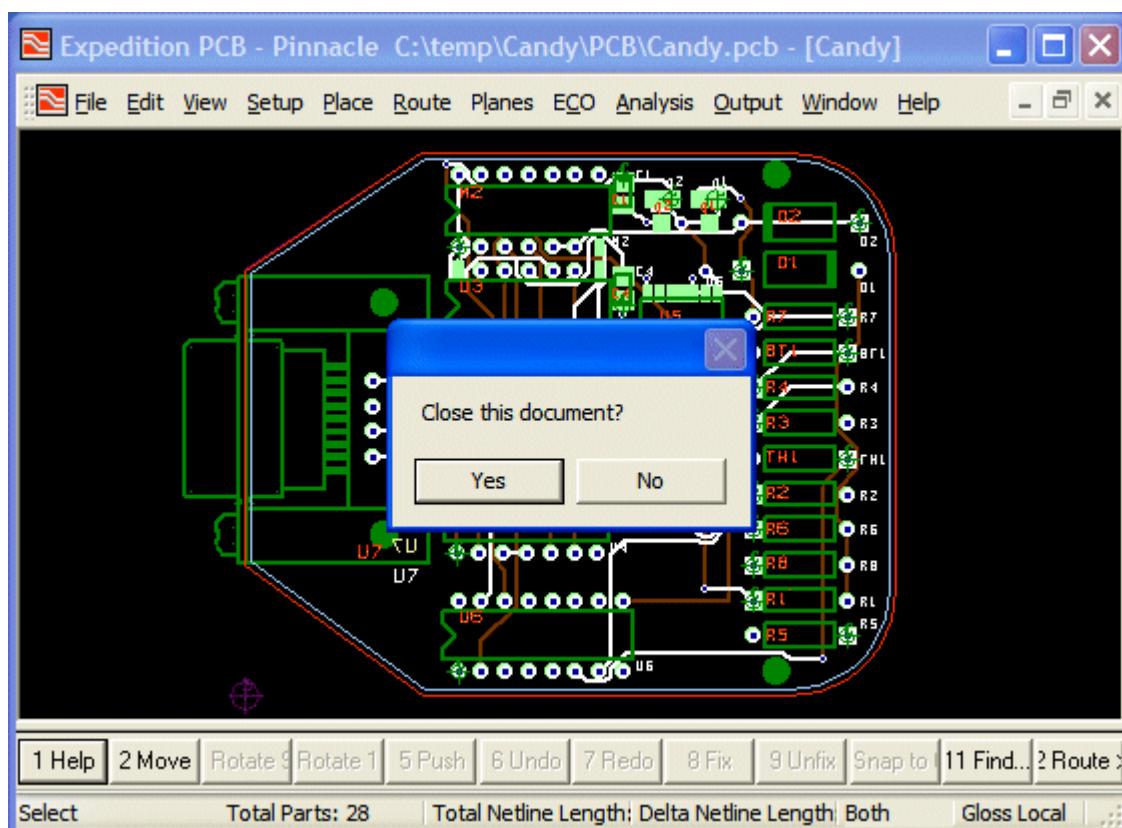


Figure 11-3. Using an event to invoke an out-of-process event handler script.

- 6) Click the **No** button and observe that the document does not close.
- 7) Once again, use the **File > Close** command to instruct Expedition PCB to close the layout design document.
- 8) Click the **Yes** button and observe that the document closes.



Note: The command prompt returns in the console window at this point. The script is exited when **Yes** is selected because the *dontExitBool* variable was set to false on Line 37.



Note: These discussions provided only a very simple introduction to events and event handling functions. More sophisticated versions are presented in *Chapter 12: Basic Building Block Examples*.

Chapter 12: Basic Building-Block Examples

Introduction

This chapter presents a suite of basic “building-block” automation examples. Each of these building-block scripts performs some fundamental task in isolation; these tasks can subsequently be combined in different ways to form more scripts.

Creating a Skeleton Script

Before we proceed, use the scripting editor of your choice to create the following “skeleton script” and save it out as *Skeleton.vbs* in your C:\Temp folder.

```
1 Option Explicit
2
3 ' Add any type libraries to be used.
4 Scripting.AddTypeLibrary( "MGCPBC.ExpeditionPCBAApplication" )
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 ' ##### YOUR CODE WILL GO HERE
18
19
20 .....
21 'Local functions
22
23 ' Server validation function
24 Private Function ValidateServer(docObj)
25   :
26   etc.
```

By this stage, Lines 1 through 16 should be very familiar. Following the *Option Explicit* statement, we add any type libraries to be used, access an *Application* object, access a *Document* object, and then call the *ValidateServer* function to license the document.

Meanwhile, Line 20 onwards contains the same server validation function we introduced in *Chapter 3: Running Your First Script* and that we've been using in all of our subsequent scripts.

You will be using this skeleton script as the starting point for all of the test cases in the remainder of this chapter. In each case, you will replace the comment on Line 18 with the new code associated with 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 of interest.
- Verify the existence of one or more objects of interest.
- Report information with regard to one or more objects of interest.
- Etc.

In this topic, we are going to show three examples that demonstrate different facets with regard to traversing through nets, components, and pins.

Traversing Nets

Goal: To locate any nets with opens and to present the user with a list of these nets (or inform the user if no such nets exist).

Approach: Use a *For Each ... Next* control loop to iterate through all of the nets in a layout design document.

Assumptions: None.



Note: Unlike *components*, *pins*, and *traces* – each of which correspond to physical entities in the real world – *nets* are logical constructs that are used to organize electrical connectivity. With regard to the Expedition PCB Automation Interface, nets are regarded as being objects with associated methods and properties. Furthermore, nets can be treated in a similar manner to other objects such as traces and pins.

Use the scripting editor of your choice to open the *Skeleton.vbs* script in your C:\Temp folder, save it 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)

```

On Line 19 we instantiate a variable called *netColl*, which will be used to hold a collection of nets. Similarly, on Line 20 we instantiate a variable called *netObj*, which will be used to represent a single *Net* object. And on Line 21 we retrieve all of the nets from the document and assign them to our *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
```

On line 24 we instantiate a variable called *netListStr*, which we will use to store a list of nets with opens. Also on this line, we initialize the *netListStr* variable with a null string.

```
23  ' Instantiate and initialize string to hold net names  
24  Dim netListStr: netListStr = ""
```

In Lines 26 through 32 we use a *For Each ... Next* control loop to iterate through all of the nets in the collection. Whenever a net with an open is detected on Line 28, its name is added to the list stored in the *netListStr* variable.

```
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
```



Note: Observe the sub-string of spaces (" ") in the middle of Line 30. These are used to make the output more visually appealing. Also on Line 30, observe the use of the *vbCrLf* character string/constant (see also *Chapter 2: VBS Primer*). This equates to a carriage return and line feed, which means that it will appear as a new line when the string is eventually displayed or printed.

Lines 34 through 42 are used to decide exactly what it is that we want to display. On line 35 we instantiate a new variable called *displayStr*; this will hold the final string to be displayed.

```
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
```

On line 36 we test our *netListStr* to see what it contains. If *netListStr* still equates to a null string, then this means that no nets with opens were detected, in which case – as seen on Line 38 – *displayStr* is assigned a string saying "No opens found." Otherwise, on Line 41, our *displayStr* variable is assigned a string comprising three elements: the string "Open Net List:", a *vbCrLf* character string/constant to force a new line, and the contents of our *netListStr* variable.

Finally, on Line 45, we call the *MsgBox()* function to display the result.

```
44  ' Display the result.
```

Let's assume that the layout document in which we are interested is called *Candy.pcb*, and that it's located in your *C:\Temp\Candy\PCB* folder (see also *Appendix E: Accessing Example Scripts and Designs* with regard to obtaining a copy of this design).

- 1) Close any instantiations of Expedition PCB that are currently running, and then launch a new instantiation of this application.
- 2) Open the *Candy.pcb* document. This document doesn't actually contain any nets with opens, so modify the design to create two or more opens (do not save these changes).
- 3) Drag the *TraverseNets.vbs* script and drop it on top of the layout design document. Observe the result as illustrated in Figure 12-1 (the actual opens reported will depend on the changes you make to your design).

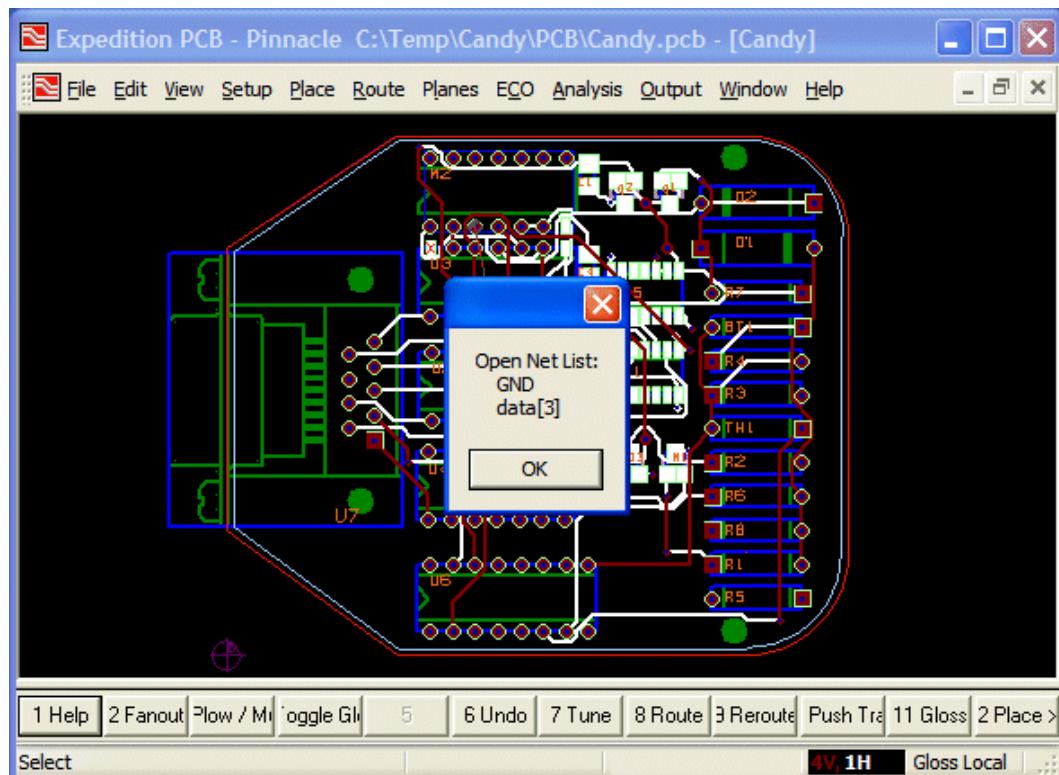


Figure 12-1. Running the *TraverseNets.vbs* script.

- 4) Click the **OK** button to dismiss the *MsgBox()* and terminate the script, then use the **File > Close** command to close the design – once again, do not save any changes you made as part of testing this script.

Traversing Components

Goal: The goal of this script is to filter a collection of components so as to un-place all of the design's bottom-side components.

Approach: Use a *For ... Next* control loop to iterate through all of the components in a layout design document; identify the components of interest and then delete (un-place) them.

Assumptions: For the purposes of providing an interesting example, we will assume that we are working with a design that is populated with both top-side and bottom-side components.

Also of Interest: This script uses the Expedition PCB status bar to inform the user as to what actions are occurring.

Use the scripting editor of your choice to open the *Skeleton.vbs* script in your C:\Temp folder, save it as *TraverseComponents.vbs*, and replace the comment on Line 18 with the following:

```
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, we use the *StatusbarText* method associated with the *Gui* object that we retrieve from the *Application* object (*pcbAppObj*) to display a message on the status bar. Note that we've shown Line 19 as being split into two lines for aesthetic reasons; in the real script this would be 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 is the string to be displayed. The second parameter is an enumerate that specifies the target field on the status bar as illustrated in Figure 12-2. The S3 field doubles up as the *Error* field (this field turns yellow when it's displaying an error).

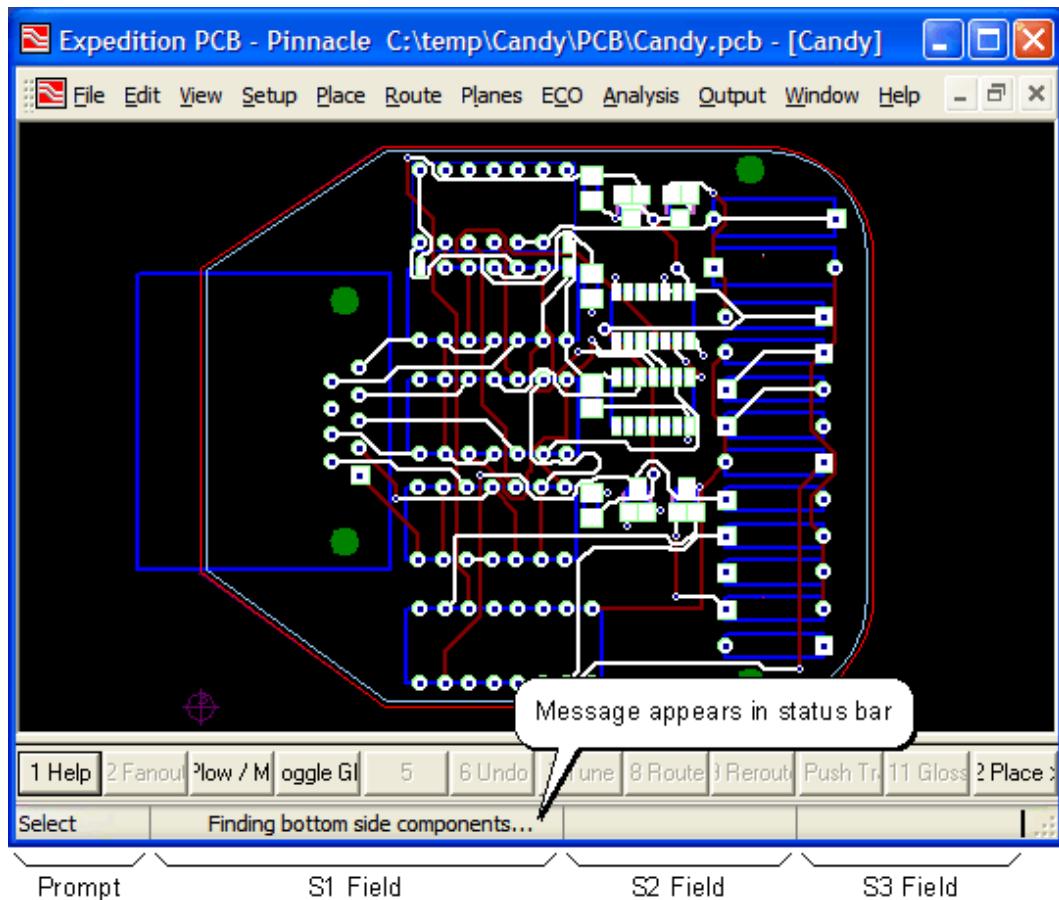


Figure 12-2. Displaying messages in the status bar.

On Line 22 we instantiate a variable called *cmpsColl*, which will be used to hold a collection of components. On Line 21 we retrieve all of the components from the document and assign them to our *cmpsColl* variable.

```
21  ' Get the component collection
22  Dim cmpsColl
23  Set cmpsColl = pcbDocObj.Components
```

The next stage in the process is to filter this collection to remove any components that are not on the bottom-side of the board. This may seem counter-intuitive in that our goal is to un-place all of the bottom-side components. The point, however, is that once we've removed all of the components that are *not* on the bottom-side, our collection will contain only bottom-side components, and we can then un-place the entire collection using a single operation.



Note: Another valid approach would be to delete the bottom side components as they are found.

In Lines 25 through 34 we use a *For ... Next* control loop to iterate through all of the components in the collection. In particular, observe Line 28, in which we iterate from the last item in the collection to the first (as opposed to iterating from the first item to the last). The reason for doing it this way is that we intend to remove items from the collection; we remove items based on their item number; and the act of removing items will change the remaining item numbers, which could potentially confuse 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

```

On Line 30 we check to see if the current item is *not* on the bottom-side and – if this is the case – on Line 32 we use the *Remove* method on our collection to extract this item from the collection. Thus, following Line 34, the collection will contain only bottom-side components.



Note: The *Remove* method does not delete the item from the board – it simply takes it out of the collection.



Note: The *For ... Next* loop used in this *Traversing Components* example behaves in a different manner to the *For Each ... Next* loop we used in Lines 26 through 32 of the previous *Traversing Nets* example. In the case of the *For Each ... Next* loop, the object in question was already assigned to a variable when it came to performing the *If ... Then* test inside the loop.

By comparison, in this example we are using an integer variable to control the *For ... Next* loop. This is why, on Line 30, we use the item number to retrieve the individual *Component* object and compare its *Side* property to the enumerate *epcbSideBottom* in order to determine whether or not this component is to be removed from the collection.

Lines 36 through 46 are used to (a) delete (un-place) the bottom-side components if there are any and (b) to use the status bar to inform the user as to what actions are 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

```

On Line 37 we check to see if there's anything left in our collection; if not, then there are no bottom-side components in this design, in which case the statement on Line 45 is used to inform the user accordingly.

Alternatively, if there are some bottom-side components, then on Line 39 we inform the user that we are about to start deleting components; on Line 41 we call the *Delete* method on our *cmpsColl* component collection, which un-places all of the components in the collection; and on Line 43 we inform the user how many components were deleted (un-placed).

Before we proceed to the next example, there is a final point that we need to consider. Generally speaking, a single call to a method or property in a script corresponds to a single **Undo** action in the GUI. The exception to this rule comes when we perform actions on

collections of objects. If our script above were to be run as-is, then if we wanted to use the **Undo** command in the GUI, we would have to use a separate **Undo** for each of the deleted components.

The way to think about this is that the *Delete* method un-places the components one at a time; this explains why we would have to **Undo** these actions one at a time. The solution is to encapsulate the delete operation in a transaction. Replace Lines 40 and 41 above 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
```

On Line 41 we declare the start of a transaction; on Line 43 we perform the delete; and on Line 45 we declare the end of the transaction. Transactions can be used to group multiple actions into a single **Undo** action in the GUI. In this case, we've used a transaction to gather all of the delete actions into a single action.

Traversing Pins

Goal: This script has three goals and it comprises three tasks. First, the script is used to select all pins with the name '1'. Second, the script is used to select all of the pins on a particular net. Third, the script is used to select all of the pins on a particular component.

Approach: To use the context associated with different types of objects (*Document*, *Net*, and *Component*) to access a collection of pins.

Assumptions: None.

Also of Interest: Using pairs of double-quotes in strings.

Use the scripting editor of your choice to open the *Skeleton.vbs* script in your C:\Temp folder, save it as *TraversePins.vbs*, and replace the comment on Line 18 with the following:

```
18  ' Common variables
19  Dim pinColl
20  Dim pinObj
21
22  .....
23  ' Select pins named '1'
24
25  Set pinColl = pcbDocObj.Pins
26
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
34
35  ' Let the user know what has been selected
36  MsgBox("All pins name ""1"" have been selected.")
37
38  ' Clean up by unselecting
39  pcbDocObj.UnselectAll
40
41  .....
```

```

42  ' Select all pins on a net
43
44  ' Ask the user for a net to process
45  Dim netNameStr: netNameStr = InputBox("Enter a net name.")
46
47  ' Get the net object
48  Dim netObj
49  Set netObj = pcbDocObj.FindNet(netNameStr)
50
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
54      pinColl.Selected = True
55
56  ' Let the user know what has been selected
57  MsgBox("All pins on net " & netObj.Name &
58          " have been selected.")
58 Else
59  ' Tell the user their entry was invalid
60  MsgBox(netNameStr & " was not found.")
61 End If
62
63  ' Clean up by unselecting
64  pcbDocObj.UnselectAll
65
66  .....
67  ' Select all pins on a component
68
69  ' Ask the user for a component ref des to process
70  Dim cmpRefDesStr: cmpRefDesStr =
71                  InputBox("Enter a component ref des.")
71
72  ' Get component object
73  Dim cmpObj
74  Set cmpObj = pcbDocObj.FindComponent(cmpRefDesStr)
75
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
79      pinColl.Selected = True
80
81  ' Let the user know what has been selected
82  MsgBox("All pins on net " & cmpRefDesStr &
83          " 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

```

First, on Lines 18 through 20 we declare some common variables that we will use for all three tasks. On Line 19 we instantiate a variable called *pinColl*, which will be used to hold a collection of pins. Similarly, on Line 20 we instantiate a variable called *pinObj*, which will be used to represent a single *Pin* object.

```

18  ' Common variables
19  Dim pinColl
20  Dim pinObj

```

Task #1: Select all pins with the name '1'

On Line 25 we retrieve all of the pins from the document and assign them to our *pinColl* variable as shown below"

```
25 Set pinColl = pcbDocObj.Pins
```

On Lines 28 through 33 we use a For Each ... Next control loop to iterate through all of the pins in our collection. With each iteration, on Line 30 we check the individual Pin object associated with that iteration to see if its name is the string '1'. If the pin's name is '1', then on Line 31 we set the *Selected* property associated with this *Pin* object to *True*. In turn, this will cause 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
```

On Line 36 we use a *MsgBox()* function to inform the user as to what we've just done.

```
35 ' Let the user know what has been selected  
36 MsgBox("All pins name ""1"" have been selected.")
```

On Line 39 we use the *UnselectAll* method on the *Document* object function to deselect everything in the design.

```
38 ' Clean up by unselecting  
39 pcbDocObj.UnselectAll
```

Task #2: Select all of the pins on a particular net

On Line 45 we instantiate a variable called *netNameStr*, which we will use to hold the name of a *Net* object. Also on Line 45, we use an *InputBox()* function to ask the user which net he or she wants to process. The value returned from this *InputBox()* function is assigned to our *netNameStr* variable.

```
44 ' Ask the user for a net to process  
45 Dim netNameStr: netNameStr = InputBox("Enter a net name.")
```

On Line 48, we instantiate a variable called *netObj*, which we will use to hold a *Net* object. Next, on Line 49, we apply the *FindNet* method to the *Document* object using our *netNameStr* variable as the search parameter. If a net with the user-specified name exists, it will be assigned to our *netObj* variable; otherwise, our *netObj* variable will automatically be set to the null object *Nothing*.

```
47 ' Get the net object  
48 Dim netObj  
49 Set netObj = pcbDocObj.FindNet(netNameStr)
```

On Line 52, we compare *netObj* to *Nothing*. If *netObj* is not equal to *Nothing*, then the net exists. In this case, on Line 53 we access the collection of pins from the *Net* object represented by *netObj* and we assign them to our *pinColl* variable (this is the common variable we instantiated on Line 19).

```
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
```

Next, we want to select all of the pins associated with this *Net* object. Thus, on Line 54, we set the *Selected* property on our *Pins* collection (*pinColl*) to *True*. In turn, this will cause the affected pins to be selected in the GUI.

```
54     pinColl.Selected = True
```

On Line 57 we use a *MsgBox()* function to inform the user as to what we've just done.

```
56     ' Let the user know what has been selected
57     MsgBox("All pins on net " & netObj.Name &
              " have been selected.")
```

Alternatively, if the test on Line 52 had shown that *netObj* was equal to *Nothing*, then the net specified by the user did not exist. In this case, on Line 60 we inform the user accordingly.

```
58 Else
59     ' Tell the user their entry was invalid
60     MsgBox(netNameStr & " was not found.")
61 End If
```

Finally (for this task), on Line 64 we use the *UnselectAll* method on the *Document* object function to deselect everything in the design.

```
63 ' Clean up by unselecting
64 pcbDocObj.UnselectAll
```

Task #3: Select all of the pins on a particular component

On Line 69 we instantiate a variable called *cmpRefDesStr*. This stands for "component reference designator string," where the reference designator is essentially the name of the component. We will use this variable to hold the name of a *Component* object. Also on Line 69, we use an *InputBox()* function to ask the user which component he or she wants to process. The value returned from this *InputBox()* function is assigned to our *cmpRefDesStr* variable.

```
69 ' Ask the user for a component ref des to process
70 Dim cmpRefDesStr: cmpRefDesStr =
              InputBox("Enter a component ref des.")
```

On Line 73, we instantiate a variable called *cmpObj*, which we will use to hold a *Component* object. Next, on Line 74, we apply the *FindComponent* method to the *Document* object using our *cmpRefDesStr* variable as the search parameter. If a component with the user-specified name exists, it will be assigned to our *cmpObj* variable; otherwise, our *cmpObj* variable will automatically be set to the null object *Nothing*.

```
72 ' Get component object
73 Dim cmpObj
74 Set cmpObj = pcbDocObj.FindComponent(cmpRefDesStr)
```

On Line 76, we compare *cmpObj* to *Nothing*. If *cmpObj* is not equal to *Nothing*, then the net exists. In this case, on Line 78 we access the collection of pins from the *Component* object represented by *cmpObj* and we assign them to our *pinColl* variable (this is the common variable we instantiated on Line 19).

```
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
```

Next, we want to select all of the pins associated with this *Component* object. Thus, on Line 79, we set the *Selected* property on our *Pins* collection (*pinColl*) to *True*. In turn, this will cause the affected pins to be selected in the GUI.

```
79      pinColl.Selected = True
```

On Line 82 we use a *MsgBox()* function to inform the user as to what we've just done.

```
81      ' Let the user know what has been selected
82      MsgBox("All pins on net " & cmpRefDesStr &
                 " have been selected.")
```

Alternatively, if the test on Line 77 had shown that *cmpObj* was equal to *Nothing*, then the component specified by the user did not exist. In this case, on Line 85 we inform the user accordingly.

```
83 Else      ' Tell the user their entry was invalid
84     MsgBox(cmpRefDesStr & " was not found.")
85 End If
```

Finally, on Line 89 we use the *UnselectAll* method on the *Document* object function to deselect everything in the design.

```
88 ' Clean up by unselecting
89 pcbDocObj.UnselectAll
```



Note: All three tasks in this script use the *Pins* property to access a collection of pins. In each case, however, this property is associated with a different object type. On Line 25 we use the *Document* object; on Line 53 we use a *Net* object; and on Line 78 we use a *Component* object. As we've seen, 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
```

Let's assume that the layout document in which we are interested is called *Candy.pcb*, and that it's located in your *C:\Temp\Candy\PCB* folder (see also *Appendix E: Accessing Example Scripts and Designs* with regard to obtaining a copy of this design).

- 1) Close any instantiations of Expedition PCB that are currently running, and then launch a new instantiation of this application.
- 2) Open the *Candy.pcb* document and zoom in on component *M2* as illustrated in Figure 12-3.

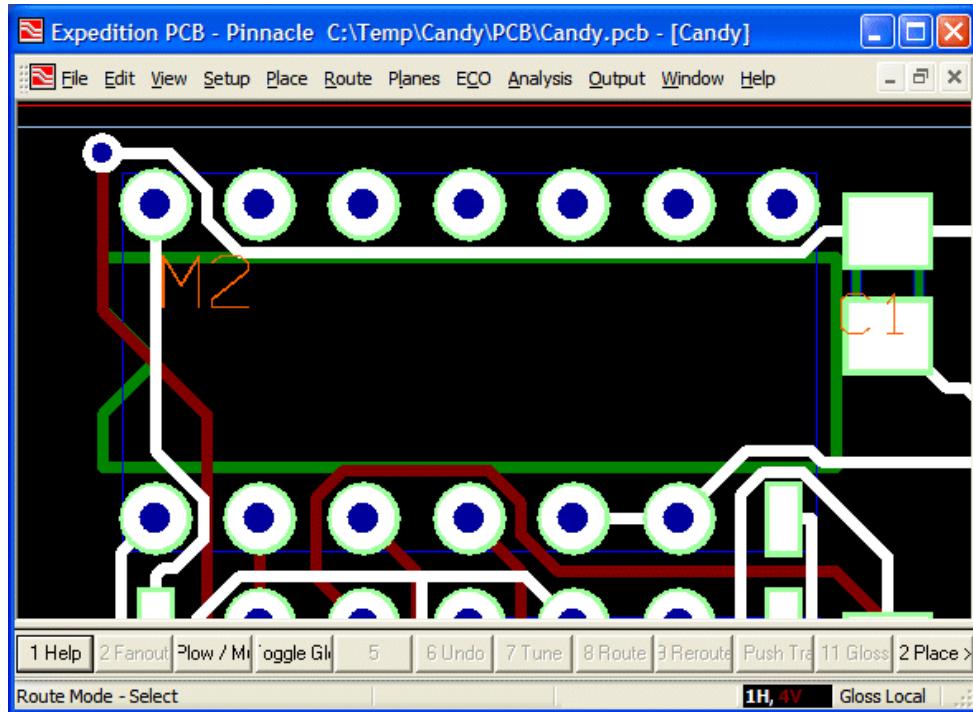


Figure 12-3. Zoom in on component M2 in the Candy.pcb design document.

- 3) Drag the *TraversePins.vbs* script and drop it on top of the layout design document. Observe the *MsgBox()* associated with **Task #1** appear as illustrated in Figure 12-4. Also observe that one pin in the lower-left-hand corner of component M2 has been selected similarly one pin on capacitor C1 has been selected.

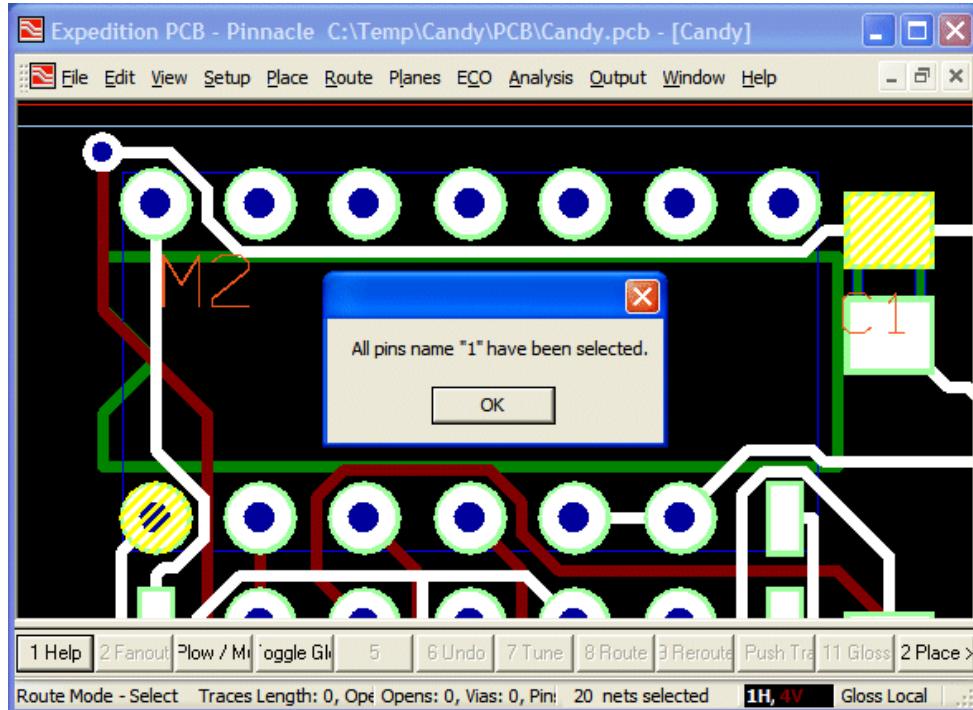


Figure 12-4. Result from Task #1 in the TraversePins.vbs script.

- 4) Click the **OK** button to dismiss this dialog and proceed to **Task #2**. Observe the *InputBox()* used to request the name of a net. Enter the name "GND" as illustrated in Figure 12-5.

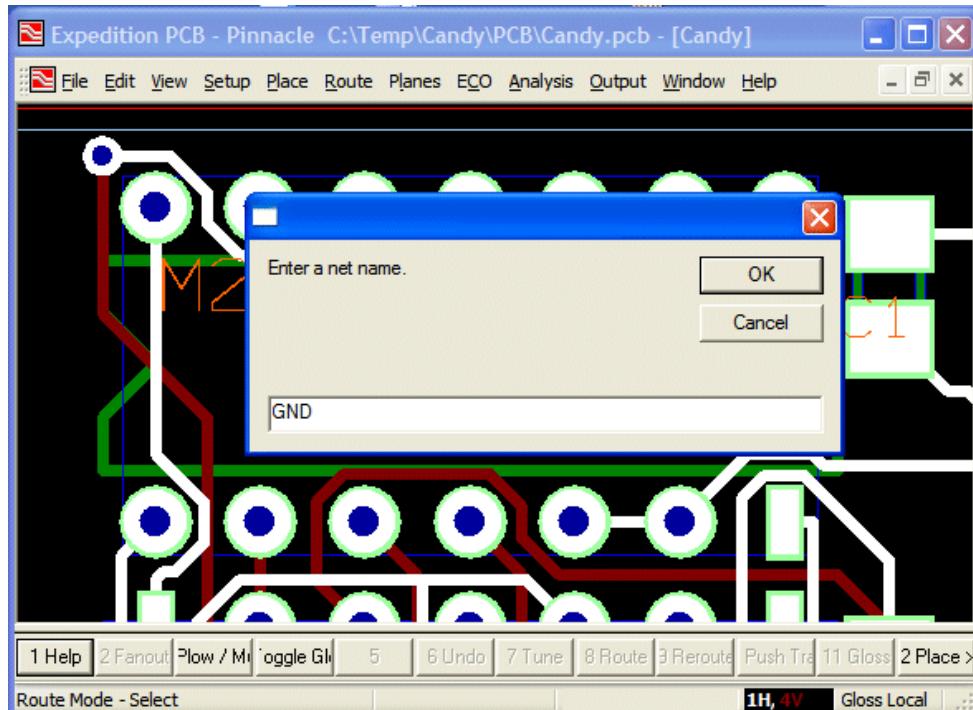


Figure 12-5. Query at start of Task #2 in the TraversePins.vbs script.

- 5) Click the **OK** button and observe the results from **Task #2**; that is, two new pins have been selected and a new *MsgBox()* appears as illustrated in Figure 12-6). Click this **OK** button and then enter the name of the "M2" component into the **Task #3 MsgBox()** (Figure 12-7).

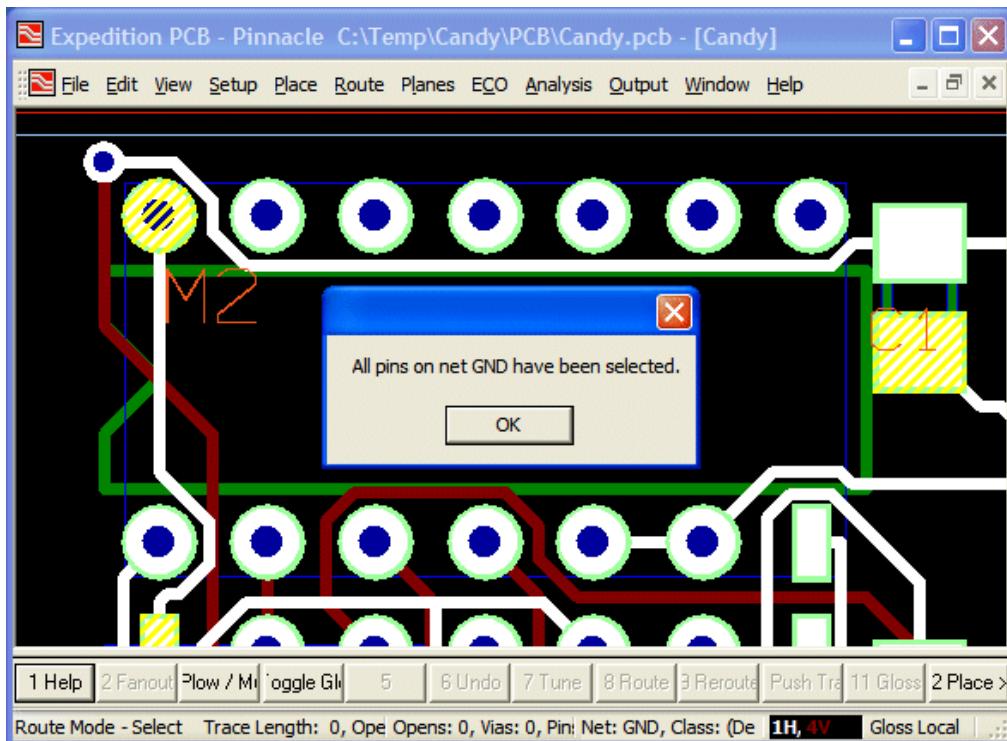


Figure 12-6. Result from Task #2 in the TraversePins.vbs script.

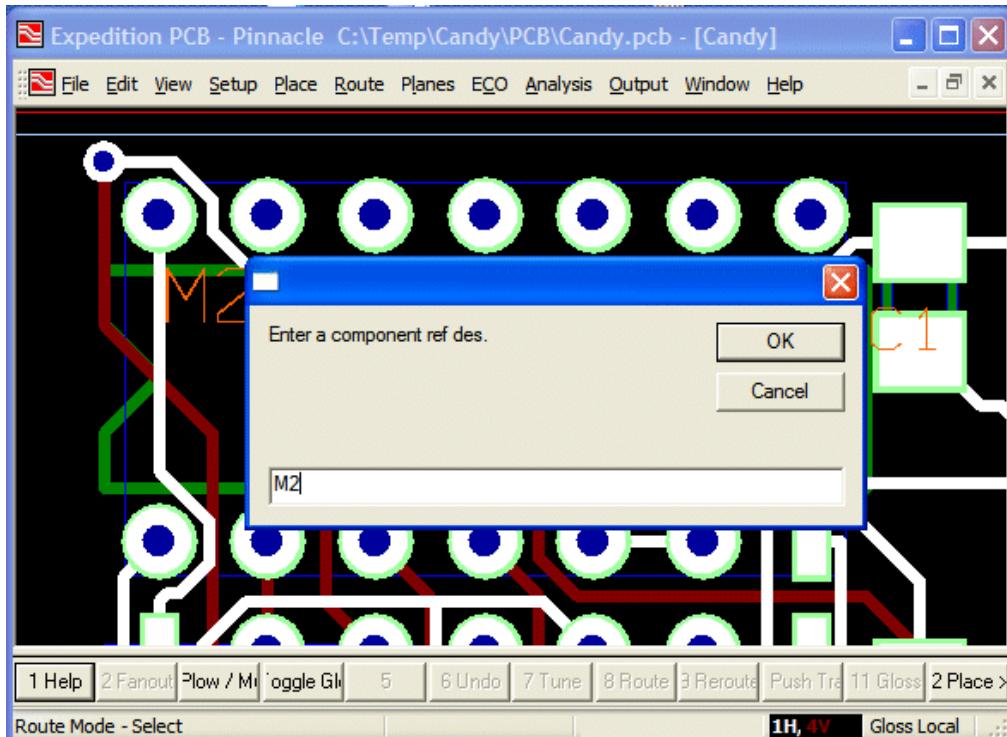


Figure 12-7. Query at start of Task #3 in the TraversePins.vbs script.

- 6) Click the **OK** button to accept and apply the "M2" component name and observe the results from **Task #3** (Figure 12-8).

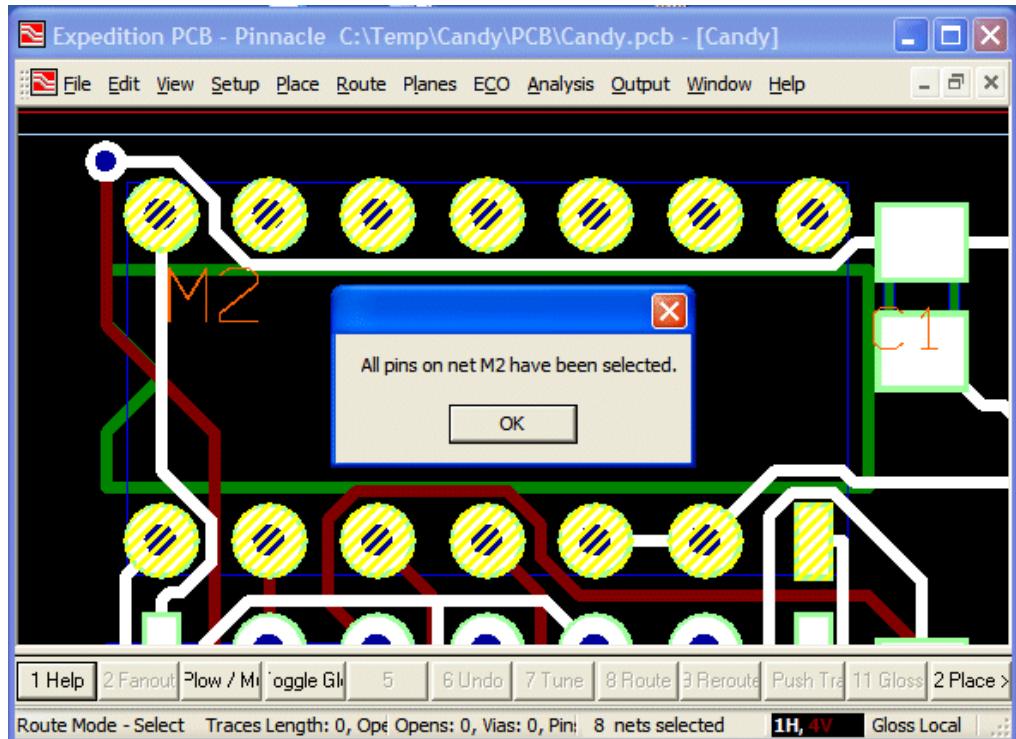


Figure 12-8. Result from Task #3 in the TraversePins.vbs script.

- 7) Click the **OK** button to dismiss this *MsgBox()* and terminate the script.

Collection Filters

In the previous topic we iterated through collections of objects and – in some cases – we explicitly filtered these collections. In fact, the Automation Interface can be requested to automatically filter collections for us. In this topic, we are going to show three examples that demonstrate different facets with regard to collection filters.

Finding Unplaced Components

Goal: 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.

Assumptions: None.

Use the scripting editor of your choice to open the *Skeleton.vbs* script in your *C:\Temp* folder, save it 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: " &
39                      vbCrLf & cmpListStr
40
41  ' Display the result.
42  MsgBox(displayStr)

```

On Line 19 we instantiate a variable called *cmpsColl*, which will be used to hold a collection of components. Similarly, on Line 20 we instantiate a variable called *cmpObj*, which will be used to represent a single *Component* object. And on Line 21 we use the *Components* property of the *Document* object to retrieve a collection of components and assign them to our *cmpsColl* variable.

```

18  ' Get the component collection
19  Dim cmpsColl
20  Dim cmpObj
21  Set cmpsColl = pcbDocObj.Components(epcbSelectUnplaced)

```



Note: We have used the *Components* property in previous scripts, but without an associated parameter, which has resulted in all of the components in the design being returned in the collection. In fact, the *Components* property supports four parameters, each of which can provide different filtering capabilities. In the case of the statement on Line 21, we are passing the enumerate *epcbSelectUnplaced* as the first parameter that will filter the collection so as to leave it containing only unplaced components (we are using default values for the other three parameters – see the Expedition PCB Automation Help for more details.)

On line 24 we instantiate a variable called *cmpListStr*, which we will use to store a list of unplaced components. Also on this line, we initialize the *cmpListStr* variable with a null string.

```

23  ' Instantiate and initialize string to hold net names
24  Dim cmpListStr: cmpListStr = ""

```

In Lines 26 through 32 we use a *For Each ... Next* control loop to iterate through all of the components in the collection. For each component, Line 28 causes its name to be added to the list stored in the *cmpListStr* variable.

```

26  ' List all unplaced components.
27  For Each cmpObj In cmpsColl
28      cmpListStr = cmpListStr & "      " & cmpObj.Name & vbCrLf
29  Next

```



Note: Observe the sub-string of spaces (" ") in the middle of Line 28. These are used to make the output more visually appealing. Also on Line 28, observe the use of the *vbCrLf* character string/constant (see also *Chapter 2: VBS Primer*). This equates to a carriage return

and line feed, which means that it will appear as a new line when the string is eventually displayed or printed.

Lines 31 through 39 are used to decide exactly what it is that we want to display. On line 32 we instantiate a new variable called *displayStr*; this will hold the final string to be displayed.

```
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: " &
39          vbCrLf & cmpListStr
39 End If
```

On line 33 we test our *cmpListStr* to see what it contains. If *cmpListStr* still equates to a null string, then this means that no unplaced components were detected, in which case – as seen on Line 35 – *displayStr* is assigned a string saying "No unplaced components found." Otherwise, on Line 38, our *displayStr* variable is assigned a string comprising three elements: the string "Unplaced Components List:", a *vbCrLf* character string/constant to force a new line, and the contents of our *cmpListStr* variable.

Finally, on Line 42, we call the *MsgBox()* function to display the result.

```
41  ' Display the result.
42  MsgBox(displayStr)
```

Deleting Traces

Goal: To delete all unfixed and unlocked traces in a design.

Approach: To use a filter on the *Traces* property in order to locate – and subsequently delete – any unfixed and unlocked traces.

Assumptions: None.

Also of Interest: This script uses the Expedition PCB status bar to inform the user as to what actions are occurring. It also provides an example of using the bit-mask properties associated with certain enumerates.

Use the scripting editor of your choice to open the *Skeleton.vbs* script in your C:\Temp folder, save it 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
21                                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(
32          "No unfixed or unlocked traces found.", epcbStatusFieldError)
```

```
32 Else
33     Call pcbAppObj.Gui.StatusBarText
            (cntInt & " traces deleted.", epcbStatusField1)
34 End If
```

On Line 19 we instantiate a variable called *trcColl*, which will be used to hold a collection of traces. On Line 21 we use the *Traces* property of the *Document* object to retrieve a collection of traces and assign them to our *trcColl* variable.

```
19 Dim trcColl
20 Set trcColl = pcbDocObj.Traces(epcbSelectUnfixed Or
                           epcbSelectUnlocked)
```



Note: We have used the *Traces* property in previous scripts, but without an associated parameter, which has resulted in all of the traces in the design being returned in the collection. In fact, the *Traces* property supports two parameters, both of which can provide different filtering capabilities. In the case of the statement on Line 20, we are specifying the first parameter and using the default value for the second (see also the following note).



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 as 1, 2, 4, 8, 16, etc. The *epcbSelectUnfixed* and *epcbSelectUnlocked* enumerates shown in Line 20 above are of this latter type. This allows us to logically OR them together to form a single parameter; in turn, this means that our collection will end up containing only unfixed **and** unlocked traces. It's 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. That is, *Or* operator allows us to combine filters and pass them as a single parameter, but both filters must be satisfied independently. (See the Expedition PCB Automation Help for more details on these enumerates and the parameters associated with the *Traces* property.)

On Line 23 we instantiate a variable called *cntInt*, which we're going to use to store the number of unfixed and unlocked traces in our collection. On Line 24, we use the *Count* property associated with our traces collection to determine how many unfixed and unlocked traces there are in the design and we assign this value to our *cntInt* variable.

```
22 ' Store the number of traces in collection to report later
23 Dim cntInt
24 cntInt = trcColl.Count
```

On Line 27 we use the *Delete* method on our trace collection to delete all of the traces in the collection (this actually deletes these traces from the design).

```
26 ' Delete traces
27 trcColl.Delete
```

Finally, on Lines 29 through 34, we use the Expedition PCB status bar to inform the user as to what occurred. The use of the status bar was discussed in detail in the *Traversing Components* script earlier in this chapter. Of particular interest here is the fact that – on Line 31 – we use the *Error* status field, which will be highlighted yellow when the message is displayed.

```
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
```

```
34  End If          (cntInt & " traces deleted.", epcbStatusField1)
```

Filtering Padstack Objects

- Goal:** To count 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 retrieve a collection of vias or pins.
- Assumptions:** None.
- Also of Interest:** This script makes use of the fact that the *PadstackObject* object is a "general object" that can represent a variety of different object types, including pins and vias. It also provides an example of using the bit-mask properties associated with certain enumerates.

In previous examples, our scripts have focused on specific object types (*Pins*, *Nets*, *Traces*, *Components*, etc.). In some cases, however, we want to access objects in a more general fashion. For example, pins and vias have common characteristics, such as the fact that they are both defined by a padstack.

In order to facilitate this sort of thing, the Expedition PCB Automation Interface provides a *PadstackObject* object, which has methods and properties that are common to all padstack-based objects. This script makes use of this *PadstackObject* object.

Use the scripting editor of your choice to open the *Skeleton.vbs* script in your C:\Temp folder, save it as *PadstackObjects.vbs*, and then replace the comment on Line 18 with the following:

```
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 we instantiate a variable called *pstkObColl* ("padstack object collection"), which will be used to hold a collection of objects. Similarly, on Line 20 we instantiate a variable called *pstkObObj* ("padstack object object"), which will be used to represent a single object. And on Line 21 we use the *PadstackObjects* property of the *Document* object to retrieve a collection of objects and assign them to our *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 objects associated with the padstack will be retrieved. In this case, we specify a parameter that will select 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 as 1, 2, 4, 8, 16, etc. The *epcbPadstackObjectVia* and *epcbPadstackObjectVia* enumerates shown in Line 21 are of this latter type. This allows us to logically OR them together to form a single parameter; in turn, this means that our collection will end up containing only vias and pins (See the Expedition PCB Automation Help for more details on these enumerates and the parameters associated with the *PadstackObjects* property.)

On Line 23 we instantiate a variable called *cntInt*, which we're going to use to store the number of pins and vias in our collection that have a pad on layer 1 of the board.

```
24 Dim cntInt
```

In Lines 23 through 30, we iterate through the collection counting and pins and vias that have a pad on layer 1 of the board. Observe Line 27 where we use the *Pads* property associated with the *PadstackObject* object (represented by our variable *pstkObObj*). In particular, observe the way in which we use the *(1)* qualifier to retrieve 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, on Line 33 we use a *MsgBox()* to inform the 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.")
```

Display Control

When using the GUI in Expedition PCB, the **View > Display Control** dialog allows the user to select which items are to be displayed and also the colors to be used for the various 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. Thus, these controls may be considered to be "local" to each view. By comparison, if the user changes the color associated with an item type in one view, this change will be automatically applied to all views. Thus, these controls are considered to be "global".

Similarly, in the Automation Interface, there are *DisplayControl* and *GlobalDisplayControl* objects, which can be used to control local and global display characteristics, respectively.

In this topic, we are going to consider three examples that demonstrate different facets of display control.

Toggling the Display of Placement Outlines On/Off

Goal: To toggle the display of component outlines from on to off and vice versa.

Approach: To use the *DisplayControl* object to manipulate the display of placement outlines.

Assumptions: None.

Use the scripting editor of your choice to open the *Skeleton.vbs* script in your *C:\Temp* folder, save it 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.ActiveView.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 we instantiate a variable called *dispCtrlObj*, which we will use to hold a *DisplayControl* object. On Line 20 we use the *ActiveView* property associated with the *Document* object to retrieve the active *View* object from the document. As part of the same statement, we immediately use the *DisplayControl* property of this active *View* object to retrieve the *DisplayControl* object associated with the active view, and we assign this object to our *dispCtrlObj* variable.

```
18  ' Get the display control object.  
19  Dim dispCtrlObj  
20  Set dispCtrlObj = pcbDocObj.ActiveView.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 – generally handles the display of any text and graphics associated with components.

Lines 23 through 29 are used to toggle the display of the top-side placement outlines on and off. As we see, the *PartItems* property supports two parameters. The first is used to define the item type, while the second is used to specify the board side of interest. In this example, we are using these parameters to specify the placement outlines (by means of the enumerate *epcbDCPlacementOutlines*) and the top-side of the board (by means of the 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
```

Also of interest is that the *PartItems* property can be used to perform both "read" and "write" tasks. In the case of the *If ... Then* statement on Line 23, for example, the *PartItems* property is used to query ("read") the current state of the specified item. Depending on this current state, Lines 25 and 28 use the *PartItems* property to assign ("write") a new value to the specified item.

Launch Expedition PCB, open a design document, 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 (see also Chapter 5: *Associating an Accelerator (Shortcut) Key with a Script*).

Changing the Layer Color

Goal: To change 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 use of the *Utility* object.

Use the scripting editor of your choice to open the *Skeleton.vbs* script in your C:\Temp folder, save it 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.ActiveView.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 we instantiate a variable called *dispCtrlObj*, which we will use to hold a *DisplayControl* object. On Line 20 we use the *ActiveView* property associated with the *Document* object to retrieve the active *View* object from the document. As part of the same statement, we immediately use the *DisplayControl* property of this active *View* object to retrieve the *DisplayControl* object associated with the active view, and we assign this object to our *dispCtrlObj* variable.

```

18  ' Get the display control object.
19  Dim dispCtrlObj
20  Set dispCtrlObj = pcbDocObj.ActiveView.DisplayControl

```

Similarly, on Line 23 we instantiate a variable called *dispCtrlGlobalObj*, which we will use to hold a *GlobalDisplayControl* object. Next, on Line 24, we use the *Global* property associated with our *DisplayControl* object (as stored in our *dispCtrlObj* variable) to access the *GlobalDisplayControl* object and assign it to our *dispCtrlGlobalObj* variable (try saying that ten times quickly!).

```

22  ' Get the global display control object.
23  Dim dispCtrlGlobalObj
24  Set dispCtrlGlobalObj = dispCtrlObj.Global

```

Now, in order to associate a new color with a display item, it is necessary to first create a new *ColorPattern* object. Such an object has *RGB* (red, 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 is used to create new objects or empty collections as required. Thus, in Line 27 we instantiate a variable called *clrObj*, which we will use to hold our new *ColorPattern* object. On Line 28, we use the *NewColorPattern* property associated with the *Utility* object to create a new *ColorPattern* object and assign it to our *clrObj* variable.

```

26  ' Create a new color object
27  Dim clrObj
28  Set clrObj = pcbAppObj.Utility.NewColorPattern

```

Lines 31 through 33 are used to assign *RGB* values to the *Red*, *Green*, and *Blue* properties associated with the *ColorPattern* object stored in our *clrObj* variable.

```

30  ' Set the color.
31  clrObj.Red    = 200
32  clrObj.Green = 100
33  clrObj.Blue  = 200

```

Just in case you were wondering, these values correspond to a rather pleasing shade of mauve/purple as shown below:

On Line 36 we use the *UserLayerColor* property associated with the *GlobalDisplayControl* object (held in our *dispCtrlGlobalObj* variable) to assign our new *ColorPattern* object (held in our *clrObj* variable) to the user layer "Test" (observe that the *Set* keyword is used in this case because an object is being assigned to the *UserLayerColor* property).

```
35  ' Apply the color to the user layer
36  Set dispCtrlGlobalObj.UserLayerColor("Test") = clrObj
```

Finally, on Line 38, we use a *MsgBox()* to inform the user what we've done.

```
38  MsgBox("Color changed for user layer ""Test"".")
```

The example design *Candy.pcb* includes a user layer called "Test". Let's assume that this design is located in your *C:\Temp\Candy\PCB* folder (see also *Appendix E: Accessing Example Scripts and Designs* with regard to obtaining a copy of this design).

Launch Expedition PCB, open the *Candy.pcb* design, access the **View > Display Control** command, select the **General** tab, and ensure the checkbox associated with the user layer "Test" is selected. Now, try running the above script to make sure it behaves as expected.

Load Scheme

- Goal:** To apply 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 called *PlacementOutlines* has already been defined for the design.

Use the scripting editor of your choice to open the *Skeleton.vbs* script in your *C:\Temp* folder, save it 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.ActiveView.DisplayControl
21
22  ' Load a display scheme
23  dispCtrlObj.LoadScheme("Loc: PlacementOutlines")
```

On Line 19 we instantiate a variable called *dispCtrlObj*, which we will use to hold a *DisplayControl* object. On Line 20 we use the *ActiveView* property associated with the *Document* object to retrieve the active *View* object from the document. As part of the same statement, we immediately use the *DisplayControl* property of this active *View* object to retrieve the *DisplayControl* object associated with the active view, and we assign this object to our *dispCtrlObj* variable.

```
18  ' Get the display control object.
19  Dim dispCtrlObj
20  Set dispCtrlObj = pcbDocObj.ActiveView.DisplayControl
```

On Line 23 we use the *LoadScheme* method associated with our *DisplayControl* object (stored in our *dispCtrlObj* variable) and pass it the new scheme name as a string.

```
22  ' Load a display scheme
```

```
23 dispCtrlObj.LoadScheme( "Loc: PlacementOutlines" )
```

Executing Menu Commands

Generally speaking, one writes scripts that use the Automation interface to directly query documents for information and/or manipulate objects. In some cases, however, it is efficacious to create scripts that execute existing menu commands/entries. In this topic, we are going to demonstrate two different aspects of this capability.

Executing the Fit Board Command

- Goal:** To use a script to execute the pre-existing **View > Fit Board** menu command/entry.
- Approach:** Use the *ProcessCommand* method associated with the *Gui* property to execute menu commands/entries.
- Assumptions:** None.

Use the scripting editor of your choice to open the *Skeleton.vbs* script in your C:\Temp folder, save it as *FitBoard.vbs*, and then replace the comment on Line 18 with the following line of code:

```
18 pcbAppObj.Gui.ProcessCommand( "View->Fit Board" )
```

As we see, in Line 18 we use the *ProcessCommand* method associated with the *Gui* property (which is itself associated with the *Application* object stored in our *pcbAppObj* variable) to execute the **View > Fit Board** command; we do this by passing it a string parameter of "View->Fit Board" (observe the use of the "->" characters and also that there are no spaces before or after these characters; also that a space is allowed in the "Fit Board" portion

That's it – this is all there is to this script (were you expecting something more?).



Note: In this example we used the string "View->Fit Board" to explicitly define the full menu path. However, since we know that there is only one **Fit Board** menu command in the interface (assuming you haven't added your own version), we could have simply passed the string "Fit Board" as the parameter to Line 18 as follows:

```
18 pcbAppObj.Gui.ProcessCommand( "Fit Board" )
```

Executing the Route Command

- Goal:** To route any open net lines (these open net lines may also be referred to as "From-Tos").
- Approach:** To individually select each open net line and route it using the **Route > Interactive > Route** menu command/entry.
- Assumptions:** None.

Use the scripting editor of your choice to open the *Skeleton.vbs* script in your C:\Temp folder, save it 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 we instantiate a variable called *fromToColl*, which will be used to hold a collection of *FromTo* objects. Similarly, on Line 20 we instantiate a variable called *fromToObj*, which will be used to represent a single *FromTo* object. And on Line 21 we use the *FromTo* property of the *Document* object to retrieve a collection of *FromTo* objects and assign them to our *fromToColl* variable.

```

18  ' Get the collection of netlines
19  Dim fromToColl
20  Dim fromToObj
21  Set fromToColl = pcbDocObj.FromTos

```

On Lines 23 through 30 we used a *For Each ... Next* control loop to iterate through all of the *FromTo* objects in our collection. On Line 25 we use the *Selected* property to select the current *FromTo* object (this will highlight the object in the GUI). On Line 17 we execute the **Route > Interactive > Route** menu command/entry by passing the string "Route->Interactive->Route" as a parameter to the *ProcessCommand* method. And on Line 29 we deselect 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 are used to inform the user what has happened by displaying messages in the status bar. (In particular, observe the statement spanning lines 39 and 40; this displays a message in the status bar's Error field, which will flash with 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 -

```

```

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: Observe the test on Line 33, in which we use the *Count* property on our collection of *FromTo* objects. The point is that, even though all of the *FromTo* objects have been routed by this state in the script, our original *FromTo* collection still exists. This means that we can still perform certain actions on the collection itself such as using its *Count* property; however, we cannot apply any methods or properties on objects *inside* the collection, because those objects are no longer valid as the from-to they represented no longer exists.

Let's assume that the layout document in which we are interested is called *Candy.pcb*, and that it's located in your *C:\Temp\Candy\PCB* folder (see also *Appendix E: Accessing Example Scripts and Designs* with regard to obtaining a copy of this design).

- 1) Close any instantiations of Expedition PCB that are currently running, and then launch a new instantiation of this application.
- 2) Open the *Candy.pcb* document, zoom in on component *U3*, and select the four traces crossing the middle of this component as illustrated in Figure 12-9.

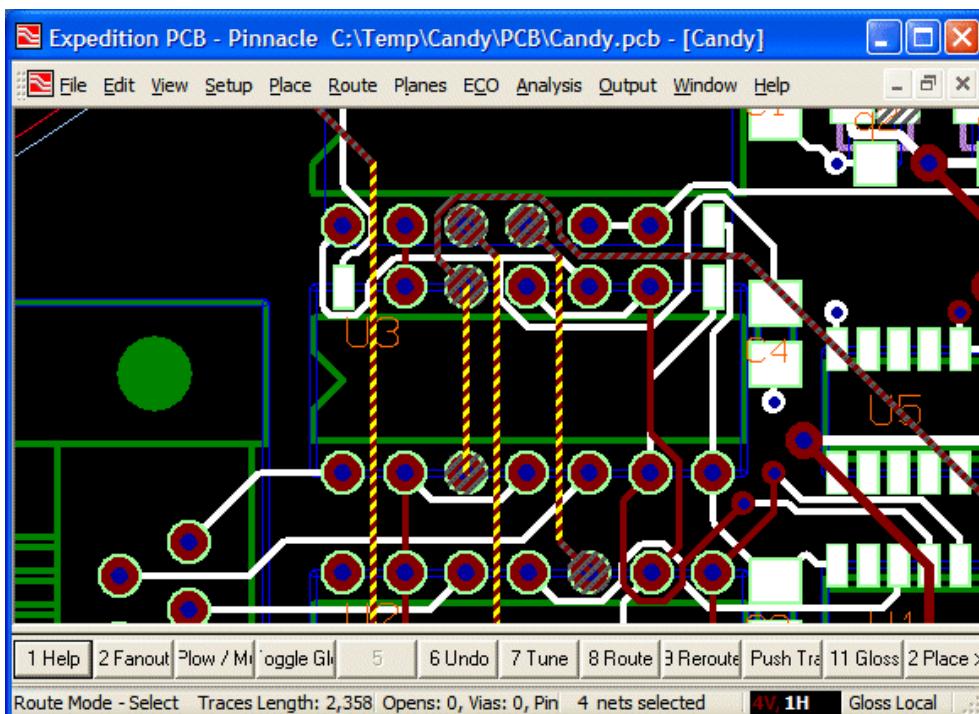


Figure 12-9. Zoom in on component U3 in the Candy.pcb design document.

- 3) Delete these traces, resulting in the design appearing as shown in Figure 12-10.

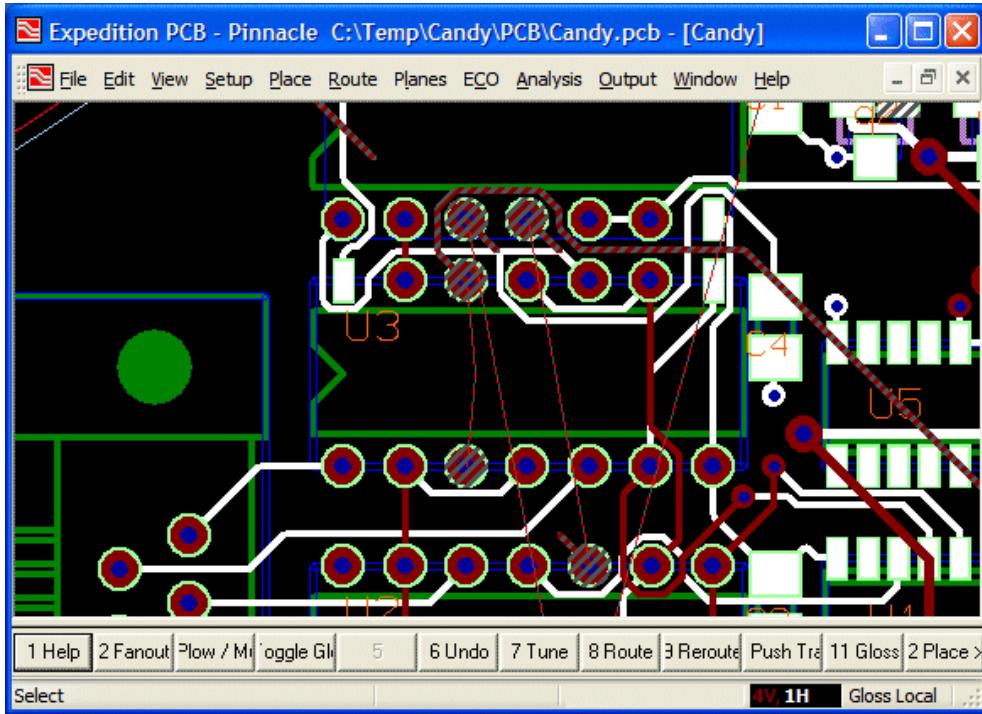


Figure 12-10. Delete the four traces crossing the middle of component U3.

- 4) Drag the *RouteNetlines.vbs* script from your *C:\Temp* folder, drop it on top of the design, and observe that these four "From-Tos" have been rerouted; also a message is displayed to this effect in the status bar as illustrated in Figure 12-11.

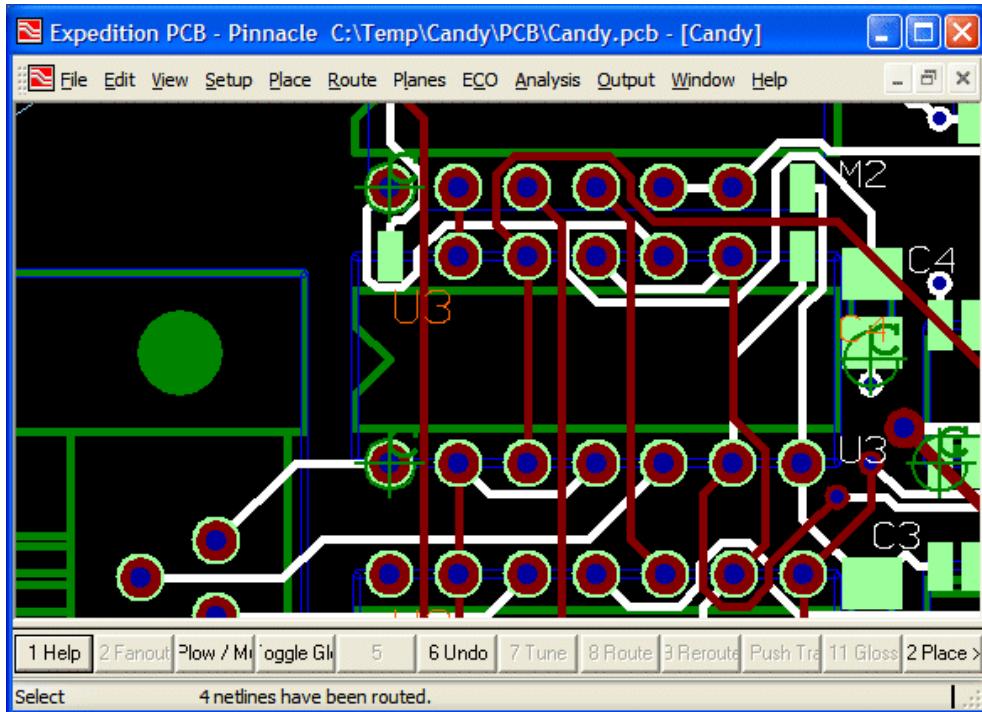


Figure 12-11. The *RouteNetlines.vbs* script reroutes the four From-Tos.

Creating Objects with Geometries

Some objects have geometries associated with them; for example, traces, placement outlines, and the board outline itself. In order to create and manipulate objects of this type, it is necessary to specify a list of vertices. Such a list is referred to as a "points array".

Every item 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. First, let's consider a geometry formed from a series of straight line segments as illustrated in Figure 12-12.

Observe that 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. Thus, assuming – for the sake of these discussions – that our geometry commences with the point located toward the bottom-left of the line (X,Y,R = 100, 100, 0), this graphical element could be 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
```

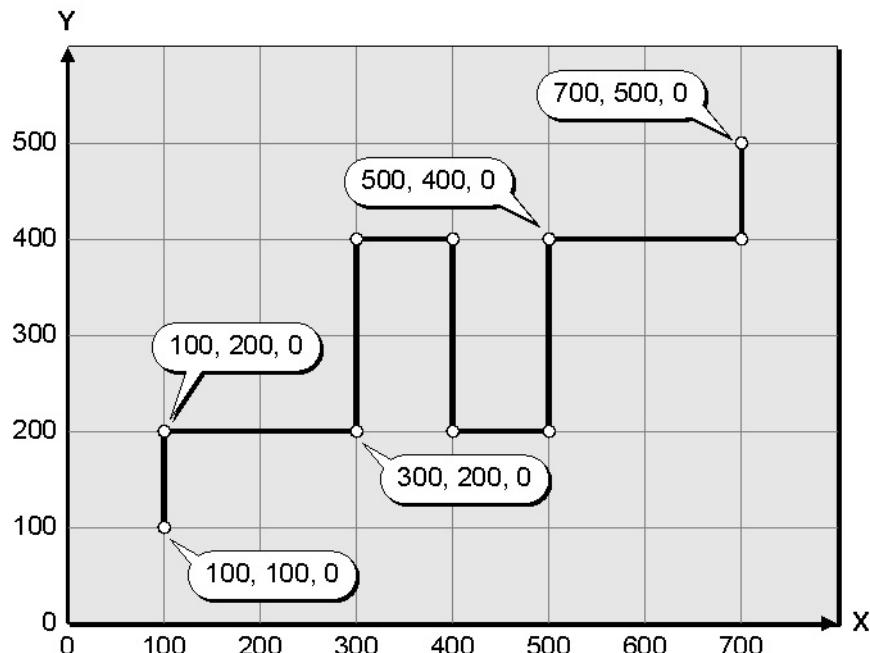


Figure 12-12. A geometry formed from a series of straight line segments.

By comparison, 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). (In the case of a circle, the

"End" point will have the same X/Y coordinates as the "Start" point). For example, consider the geometry illustrated in Figure 12-13.

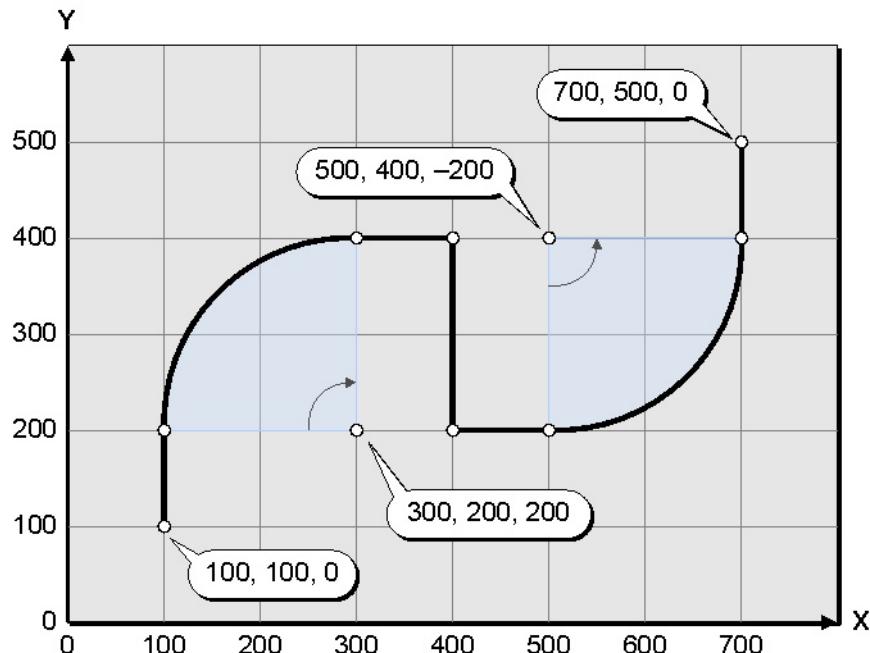


Figure 12-13. A geometry formed from straight lines and two arcs.

Once again, if we assume that our geometry commences with the point located toward the bottom-left of the line ($X,Y,R = 100, 100, 0$), this shape could be 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

```

Observe that the two geometries illustrated in Figures 12-12 and 12-13 contain exactly the same number of points and that these points are defined by identical X/Y values. The only difference is that two of the points now 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 "anticlockwise path").



Note: The values of the first and last points in the points array will be identical in the case of closed geometries such as board outlines.

Put Trace

Goal: To add a trace between two pins.

Approach: Use the *PutTrace* method to add a trace to the board.

Assumptions: We will be working with a specific design (*Candy.pcb*) from which a trace is missing.

Also of Interest: This script introduces the use of specific units of measure; also, the use of transactions to control DRC is demonstrated.

Use the scripting editor of your choice to open the *Skeleton.vbs* script in your *C:\Temp* folder, save it 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                      epcbUnitCurrent, epcbAnchorNone)
```

Before we begin to consider this script, we should first note that the X/Y locations of vertices associated with geometric objects inside a layout design document are – at the most fundamental level – represented and stored in the form of "database units" (these are largely invisible to the user).

When we acquire a document object, it will already have a *CurrentUnit* property value assigned to it. This value reflects the current design units as presented in the GUI. The related 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
3	epcbUnitInch	Inches
4	epcbUnitMM	Millimeters
5	epcbUnitUM	Micrometers

The point of interest to us here is that if the value associated with the *CurrentUnit* property is changed in the script it does not change anything in the design. This is a property that is local to your *Document* object. If, for example, the *CurrentUnit* property were set to *epcbUnitMM*, the result would be that any methods/properties accessed on your *Document* object or on any object retrieved from that *Document* object will – by default – return values in millimeters. Furthermore, every method/property that returns or accepts coordinates or lengths has a *Unit* parameter that allows you to specify the units for that method/property on an individual basis. The default value for the *Unit* parameter is *epcbUnitCurrent*, which would be *epcbUnitMM* in our theoretical case.

In this particular example, we are going to hard-code the vertices for our geometric shape, and we intend for these values to be interpreted in Mills. As it happens, we know that the design units in the *Candy.pcb* design are also in Mills; however, we cannot discount the possibility that the user may change this prior to running our script. Thus, on Line 21 we assign the *epcnUnitMils* enumerate to the *CurrentUnit* property associated with the active *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
```

Next, on Lines 26 through 28 we define our points array. On Line 26 we instantiate an array that is three columns "wide" by two rows "deep" (remember that array dimensions start at 0; the dimensioning of arrays was discussed in *Chapter 2: VBS Primer*.)

```
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
```

As we see, our 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 (remember that all X/Y values are relative to the board origin).

Lines 31 through 43 are used to establish the values of the parameters that will be required when we actually come to add our trace to the board. First, on Line 31, we instantiate a variable called *numPntsInt*, in which we will store the number of vertices defined by our points array.

```
30  ' Get the number of points in the array
31  Dim numPntsInt : numPntsInt = UBound(pntsArr, 2) + 1
```

Also on this line, we use the *UBound* function to determine the number of vertices; the value returned by the *UBound* function is incremented by 1 and assigned to our *numPntsInt* variable. The first parameter to the function is the name of our points array, the second parameter is the dimension of interest. As you will recall, we originally instantiated our points array on Line 26 using the statement *Dim pntsArr(2, 1)*. The second dimension has a value of 1, but we know that array dimensions commence at zero, so in order to determine the number of vertices, we add a value of '1' to the value of this second dimension (it all makes sense when you think about it).

Next, we have to have a *Net* object with which the *Trace* object we intend to create will be associated. In this example we are going to connect our trace to the GND net, so on Line 34 we instantiate a variable called *netObj* and then we use the *FindNet* method associated with the *Document* object to find the net called "GND" and assign this net to our *netObj* variable.

```
33  ' Get the Net object for the net the trace will be on.  
34  Dim netObj: Set netObj = pcbDocObj.FindNet("change")
```

Similarly, on Line 36 we first instantiate a variable called *widthReal* in which to hold the width of our trace, and we then assign a value of 10 to this variable (this will of course be 10 mils, because we set the units associated with this script to be mils).

```
36  ' Provide a width  
37  Dim widthReal : widthReal = 10
```

For the purposes of this example, we do not want our new trace to be associated with any particular component. Thus, on Line 40 we first instantiate a variable called *cmpObj* in which to hold a *Component* object; or not as the case might be, because we then assign a value of *Nothing* to this variable.

```
39  ' Don't add this trace to any component  
40  Dim cmpObj : Set cmpObj = Nothing
```

Last but not least, on Line 43 we instantiate a variable called *layerInt*, which we will use to hold the number of the layer on which the trace will be created, and to which we assign a value of 1.

```
42  ' Specify the layer to put the trace on  
43  Dim layerInt : layerInt = 1
```

Now we're ready to rock-and-roll. Lines 46 through 48 contain a single statement that uses the *PutTrace* method associated with the *Document* object to add a new trace to the document.

```
45  ' Put a new trace in the document  
46  Call pcbDocObj.PutTrace(layerInt, netObj, widthReal, _  
47  numPntsInt, pntsArr, cmpObj, _  
48  epcbUnitCurrent, epcbAnchorNone)
```

As we see, the *PutTrace* method requires a suite of parameters as follows:

Parameter	Description
1	Target layer for the trace
2	Net object with which the trace is to be associated
3	Width of the trace
4	Number of vertices in the points array
5	The points array itself
6	The component object with which this trace is to be associated
7	An enumerate defining the units to be used
8	An enumerate defining whether the trace is to be fixed or locked

Note that Lines 30 through 43 were actually used only for the purposes of clarity. Also that the enumerates assigned to the last two parameters were actually the default values associated with these parameters. This means that we could comment out lines 30 through 43, and that we could replace the statement on lines 46 through 48 with the following, which would act in an identical manner:

```
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)
```

The layout design document we intend to use for this example is called *Candy.pcb*. By this stage in our discussions, this design should be located in your *C:\Temp\Candy\PCB* folder

(see also *Appendix E: Accessing Example Scripts and Designs* with regard to obtaining a copy of this design).

- 1) Close any instantiations of Expedition PCB that are currently running, and then launch a new instantiation of this application.
- 2) Open the *Candy.pcb* document and zoom in on component *U3*.
- 3) While in the *Route* mode, double-click on the trace connecting pins *U3-11* and *U3-9* as illustrated in Figure 12-14. (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 our points array.)

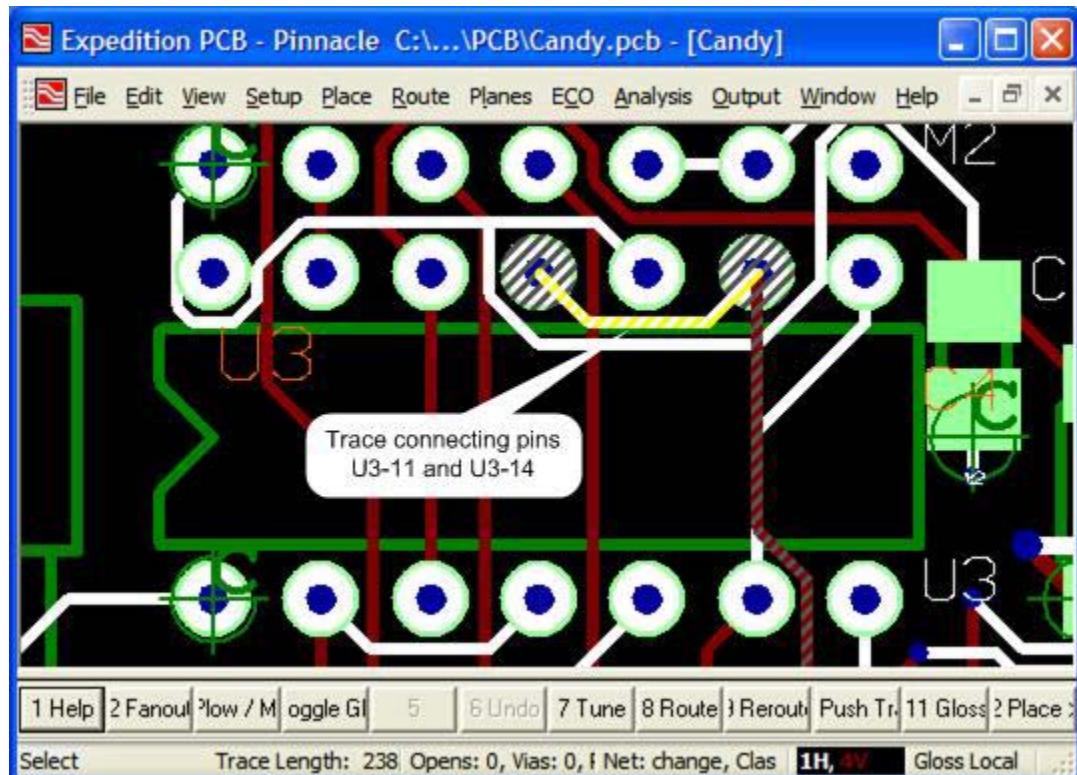


Figure 12-14. Select the trace connecting pins U3-11 and U3-9.

- 4) Delete this trace, resulting in the design appearing as shown in Figure 12-15.

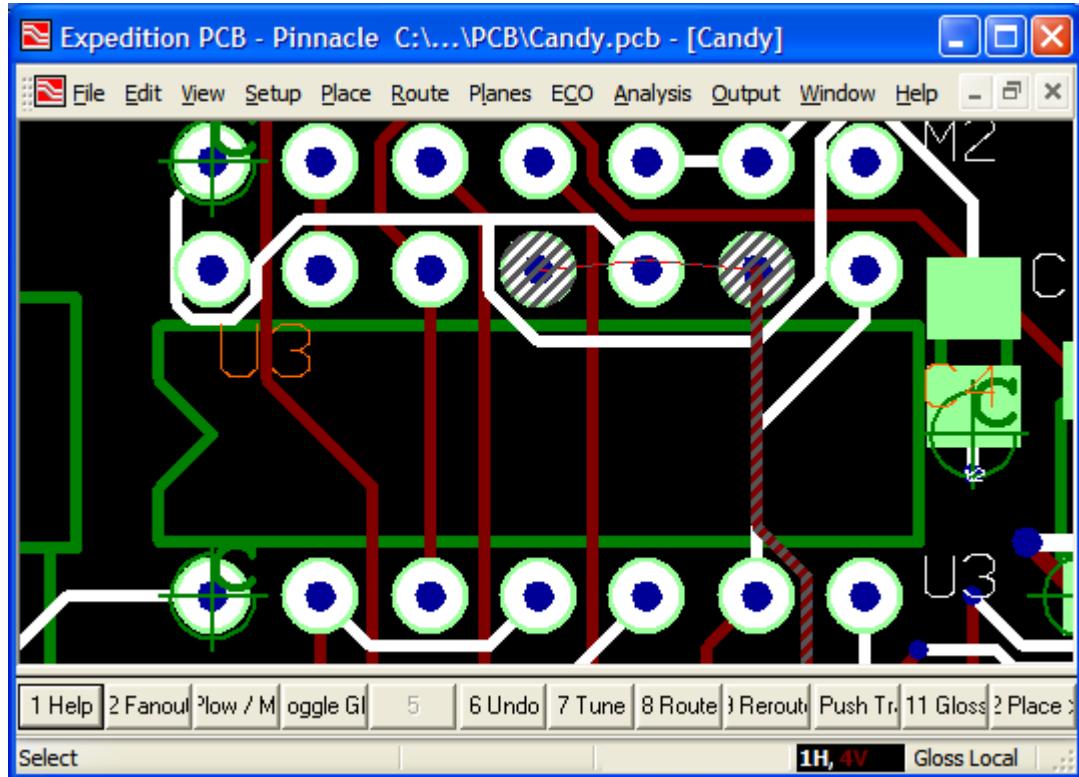


Figure 12-15. The trace connecting pins U3-11 and U3-9 has been deleted.

- 5) Drag the *PutTraceNoTransac.vbs* script from your C:\Temp folder, drop it on top of the design, and observe the resulting error message as illustrated in Figure 12-16.

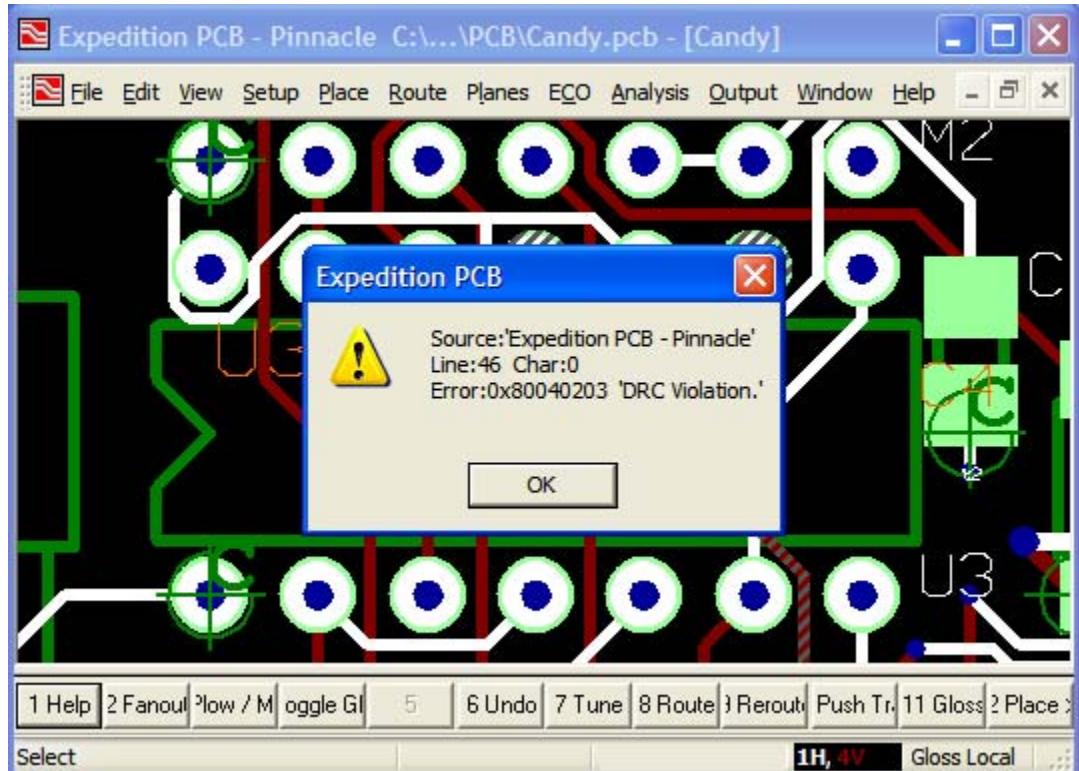


Figure 12-16. The *PutTraceNoTransac.vbs* script results in an error.

As we see, our new trace was not inserted into the design because there was a DRC fail. If we return to Figure 12-14, we can easily identify the source of the problem. Our script is attempting to route a trace directly between pins *U3-11* and *U3-9*, but this will cause the trace to pass over pin *U3-10*, which will cause a DRC failure.

The way in which to address this is to use a transaction. Use the scripting editor of your choice to open the *PutTraceNoTransac.vbs* you created earlier and save it as *PutTraceTransac.vbs*, and then 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          epcbUnitCurrent, epcbAnchorNone)
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
```

On Line 46 we use the *TransactionStart* method on the *Document* object to initiate a transaction. The *epcbDrcModeResolve* enumerate is passed as a parameter to specify the level of DRC required for this transaction. In this case, we are asking the system to resolve any DRC errors for us, if possible.

In Lines 48 through 50, we use the *PutTrace* method associated with the *Document* object in exactly the same manner as for our original script. Then on Line 53 we end the transaction.

The *Else* clause and associated statements on Lines 54 through 57 are used to inform the user if the *If ... Then* statement fails to initiate the transaction for any reason (see note below).



Note: The *If ... Then* statement "wrapping" the transaction on Line 46 is not strictly necessary. We could initiate the transaction directly if we so desired (this technique was demonstrated in the *Traversing Components* script earlier in this chapter). However, the use of the *If ... Then* statement shown in this example is good programming practice in case the database is currently being written to by something else (the user, another script, etc.).

- 1) Drag the *PutTraceTransac.vbs* script from your C:\Temp folder, drop it on top of the design, and observe that our trace is now routed as illustrated in Figure 12-17.

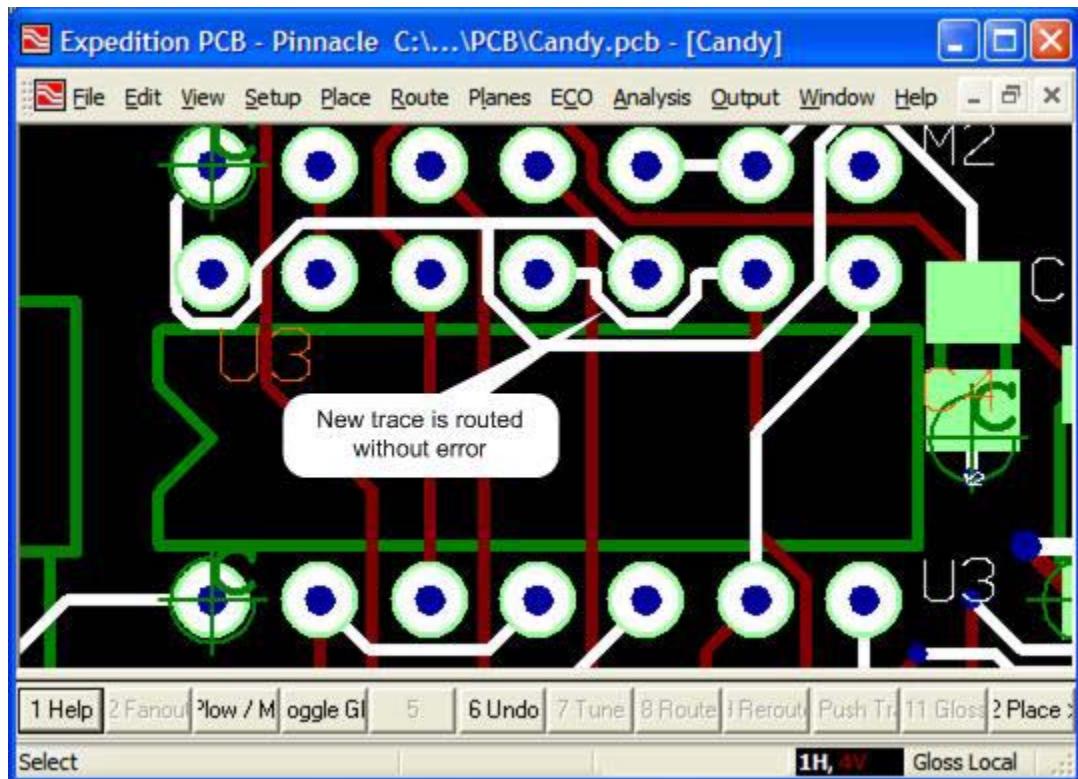


Figure 12-17. The PutTraceTransac.vbs script runs without error.



Note: There are three DRC modes: `epcbDRCModeDRC` (used by default in the original script), `epcbDRCModeResolve`, and `epcbDRCModeNone`. It is suggested that you experiment with these modes in this script and observe the results.

Put User Layer Graphics

- Goal:** To add a graphical element to a user layer.
- Approach:** Use the *PutUserLayerGfx* method to add a graphical element to a user layer on the board.
- Assumptions:** We will be working with a specific design (*Candy.pcb*) that contains a user layer named *Test*.
- Also of Interest:** The graphical element generated by this script includes two arcs.

Before we commence, it's worth noting that there is a difference between a *path* and a *shape*. Paths are graphical items, such as traces, that are made up of a series of points and that have a width (paths may or may not be closed). By comparison, shapes are graphical items, such as a conductive areas, that are made up of a series of points which must be closed and which have no width.

For example, 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 18(a). (For the purposes of these discussions, we will ignore any practical issues or potential DRC problems that might arise from creating such a trace.) If we were to zoom-in on this trace, its width would scale up as a factor of the amount of zoom.

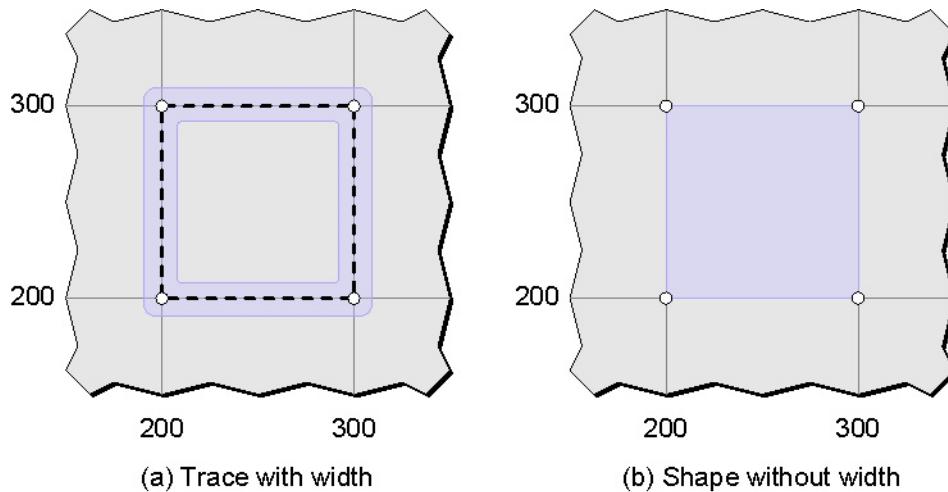


Figure 12-18. Comparison of graphics entities with/without width.

By comparison, consider a pure graphical element defined using the same four points, but with a width set to zero. In this case – and assuming that its fill parameter is enabled – the shape will be filled with the user layer color. Furthermore, the shape's outline (which will be the same user layer color and which will only be apparent when the fill is disabled or in certain modes where the fill is modified) will always be one pixel wide, irrespective of the amount of zoom.

Different types of objects may support either one or both of these geometric types. For example, *contours* are paths in which the width is the diameter of the hole used to create that contour, while the board outline is a path but the width is used only for display purposes (physically a board outline has zero width). Graphical elements on a user layer can be either a path or shape; they have no restrictions as there is no physical equivalent in the real world.

Now, use the scripting editor of your choice to open the *Skeleton.vbs* script in your *C:\Temp* folder, save it 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.ActiveView.SetExtentsToSelection
```

In this particular example, we are going to hard-code the vertices for our geometric shape, and we intend for these values to be interpreted in Mills. Thus, on Line 21 we assign the *epcbUnitMils* enumerate to the *CurrentUnit* property associated with the active *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
```

On Lines 24 through 34, we instantiate the points array and specify the points array that we are going to use to generate our graphical object.

```

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 are used to establish the values of the parameters that will be required when we actually come to add our graphical element to the board. First, on Line 37, we instantiate a variable called *numPntsInt*, in which we will store the number of vertices defined by our points array. Also on this line, we use the *UBound* function to determine the number of vertices; the value returned by the *UBound* function (plus 1) is assigned to our *numPntsInt* variable.

```

36  ' Get the number of points in the array
37  Dim numPntsInt : numPntsInt = UBound(pntsArr, 2) + 1

```

On Line 40 we instantiate a variable called *userLayerObj*, which will be used to hold a *UserLayer* object. Next, on Line 41, we use the *FindUserLayer* method associated with the *Document* object to locate and retrieve the user layer called "Test", and we assign this object to our *userLayerObj* variable.

```

39  ' Get user layer object
40  Dim userLayerObj
41  Set userLayerObj = pcbDocObj.FindUserLayer("Test")

```

On Line 44 we first instantiate a variable called *widthReal* in which to hold the width associated with our graphical shape, and we then assign a value of 0 (zero) to this variable. This will result in a shape with a 1-pixel outline as discussed at the beginning of this topic.

```

43  ' Provide a width
44  Dim widthReal : widthReal = 0

```

On Line 47 we instantiate a variable called *filledBool*. Also on this line, we assign this variable a Boolean value of True; this will enable the fill when we eventually generate our graphical element.

```

46  ' Set shape as filled
47  Dim filledBool : filledBool = True

```

For the purposes of this example, we do not want our new graphical element to be associated with any particular component. Thus, on Line 50 we first instantiate a variable called *cmpObj* in which to hold a *Component* object, and we then assign a value of *Nothing* to this variable.

```

49  ' Don't add this shape to a component
50  Dim cmpObj : Set cmpObj = Nothing

```

On line 53, we instantiate a variable called *usrLyrGfxObj*, which will be used to hold our new graphics object. Lines 54 through 56 contain a single statement that uses the *PutUserLayerGfx* method (in conjunction with all of our parameters) to add the new graphic to the document.

```

52  ' Add the user layer graphics
53  Dim usrLyrGfxObj
54  Set usrLyrGfxObj = pcbDocObj.PutUserLayerGfx(userLayerObj, _
55      widthReal, numPntsInt, pntsArr, filledBool, _
56      cmpObj, epcbUnitCurrent)

```

Last but not least, on Line 59 we use the *Selected* property on our *userLayerObj* variable to select this object in the GUI. And on Line 62, we use the *SetExtentsToSelection* method on the view returned by the *ActiveView* property to zoom in on our new graphical shape.

```

58  ' Select the newly added object
59  usrLyrGfxObj.Selected = True
60
61  ' Fit the selected object
62  Call pcbDocObj.ActiveView.SetExtentsToSelection

```

The layout design document we intend to use for this example is called *Candy.pcb*. This design already contains a user layer called "Test" (see also *Appendix E: Accessing Example Scripts and Designs* with regard to obtaining a copy of this design).

- 1) Close any instantiations of Expedition PCB that are currently running, and then launch a new instantiation of this application.
- 2) Open the *Candy.pcb* document. Then use the **View > Display Control** menu command, select the **General** tab, and then ensure that the check box next to the user layer called "Test" is turned on.
- 3) Drag the *PutUserLayerGfx.vbs* script from your *C:\Temp* folder, drop it on top of the design, and observe the resulting shape appear as illustrated in Figure 12-19.

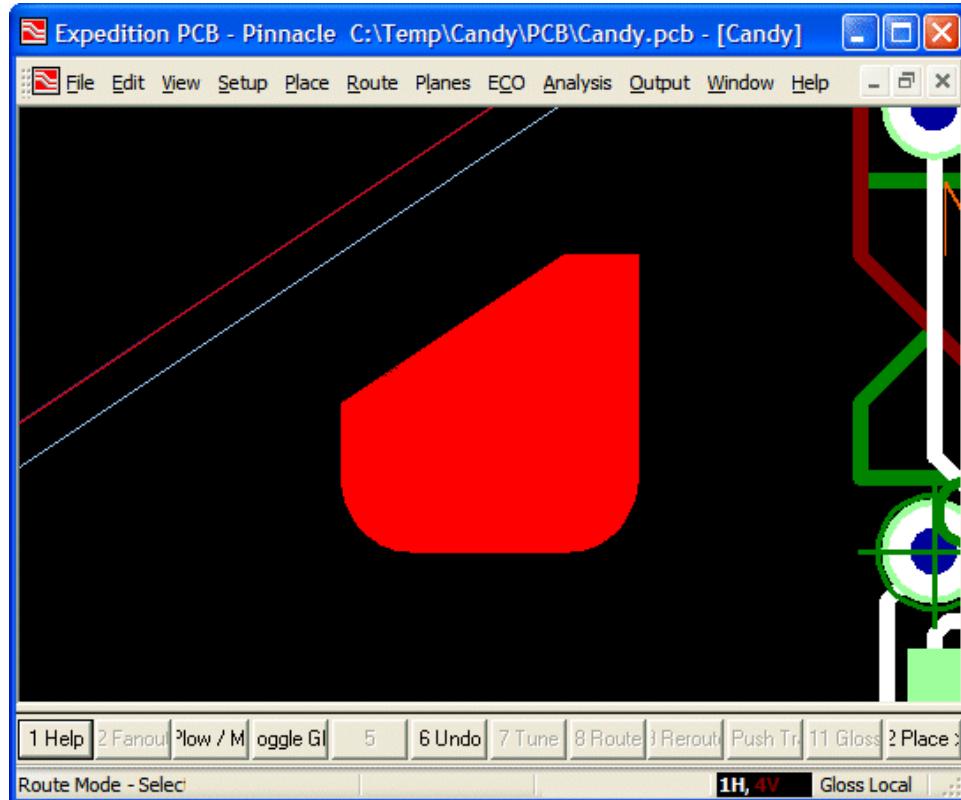


Figure 12-19. Using the PutUserLayerGfx.vbs script to add a graphical element.



Note: This screenshot shows the shape as being unselected. If the shape is selected, its appearance may vary depending on the current mode (*Route*, *Draw*, *Place*, etc.)

Interactive Commands

There may be cases where we want to write scripts that interact with the user; for example, by performing specific actions associated with mouse clicks or mouse moves. In this topic, we are going to demonstrate a classic aspect of this capability.



Note: This script makes use of the *Command* object. Although there are a few methods and properties associated with this object, it is mostly composed of events. In particular, the *Command* object can be used to receive any mouse or keyboard events.

Put Vias

- Goal:** To use a script to place vias at locations where the mouse is clicked.
- Approach:** Use the *Command* object to handle mouse click events by placing a via at the current cursor location when the click occurs.
- Assumptions:** That a "VIA20" padstack exists in the design; also that there is a net named "GND".

Use the scripting editor of your choice to open the *Skeleton.vbs* script in your *C:\Temp* folder, save it 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  .....
31  ' Command event handlers
32
33  Function cmdObj_OnTerminate()
34      ' Release the command
35      Set cmdObj = Nothing
36  End Function
37
38  Function cmdObj_OnMouseClk(button, flags, x, y)
39
40      ' Get all the parameters needed to put a via
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                                         pcbStatusFieldError)
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 we instantiate a variable called *cmdObj*, which will be used to hold the Command object.

```

18  ' Variable to hold the Command object
19  Dim cmdObj

```

On Line 22 we create a Command object by using the *RegisterCommand* method associated with the *Gui* object, and we assign this to our *cmdObj* variable. The parameter passed to the *RegisterCommand* method is a string that will appear 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")

```

On Line 25 we use the *AttachEvents* method associated with the *Scripting* object to instruct the script engine that we are interested in any events associated with the object that is specified by the first parameter. In this case, we're using the *Command* object variable called *cmdObj* that we instantiated on Line 12. (See also *Chapter 11: Introducing Events and the Scripting Object*.)

Observe that the second parameter is a string. This string will eventually be used as a common prefix for any of the functions we declare to handle events originating from the *Command* object assigned to the *cmdObj* variable. The string itself is arbitrary, but it is recommended programming practice to set this string to be identical to the name of the first parameter. Thus, as the first parameter in our example is the *cmdObj* variable, we will use "cmdObj" as our string.

```

24  ' Attach events to the command object
25  Call Scripting.AttachEvents(cmdObj, "cmdObj")

```

With the exception of the scripts associated with Chapters 6, 7, and 11, the majority of our scripts have terminated once they've completed execution. This would be a problem in this case, because if our script terminates the event handler (which we declare later in the script) cannot be called. Thus, on Line 28 we set 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 has a specific lifespan that can be ended in a number of ways; these include:

- The invocation of another command.
- Pressing the <Esc> ("Escape") key.
- Terminating the object in the script itself.

Whenever any of these occur, the *OnTerminate* event will be triggered. You should always have a handler for this event; in turn, this handler should – at the very least – release the *Command* object (additional cleanup may be required for other commands – see also *Chapter 14*). The event handler on Lines 33 through 36 is used to set the *Command* object stored in our *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
```

Remember that the name of the event handler function (on Line 33) is key to making sure this function gets fired. If either the name or the number of parameters differ in any way from the documentation, the event handler function will not get called.

In the case of this example, there are no parameters to be passed into the function. Meanwhile, the name of the function (*cmdObj_OnTerminate*) comprises three parts as follows:

- The string we previously specified as the second parameter to the *AttachEvents* method (this was the "cmdObj" string we used on Line 25).
- An underscore '_' character.
- The name of the event as defined in the Expedition Automation Help documentation, which is *OnTerminate* in this example.

The main event handler function is described in Lines 38 through 71. On Line 38, we declare the function itself. In this case, the name of the event in which we are interested is *OnMouseClk*, which is fired when the mouse button is released after being pressed (there are a variety of other mouse click events, such as *OnMouseDown*, which fires when the mouse is first pressed).

```
38  Function cmdObj_OnMouseClk(button, flags, x, y)
```

The *OnMouseClick* event requires four parameters. The first, called *button* in this example, can be used to determine which mouse button was pressed. The second, called *flags* in this example, can be used to determine if one or more of the <Alt>, <Shift>, and/or <Ctrl> keys are being pressed when the mouse click takes place. The event handler described in this example does not make use of these first two parameters. Finally, the third and fourth parameters describe the X/Y location of the mouse cursor when the click event takes place.

Following the click event, we need to gather all of the parameters required to create a via. First, we require a padstack to be used for the via. On Line 41 we instantiate a variable to hold a *Padstack* object. Next, the statement spanning Lines 42 and 43 uses the *PutPadstack* method associated with the *Document* object to access and load a *Padstack* object and to associate this object with our *pstkObj* variable. The first two parameters to the *PutPadstack* method specify the range of layers of interest (from 1 to the maximum number of layers in this case), while the third parameter is the name of the padstack ("VIA20" in this example).

```
40      ' Get all the parameters needed to put a via
41      Dim pstkObj
42      Set pstkObj = pcbDocObj.PutPadstack(1,
43                                         pcbDocObj.LayerCount, "VIA20")
```



Note: The *PutPadstack* method will load a padstack from the local library if that padstack does not already exist in the design. By comparison, if the padstack does already exist in the design, the *PutPadstack* method simply returns that padstack.

Next, we have to have a *Net* object with which the *Padstack* object we intend to create will be associated. In this example we are going to connect our trace to the GND net, so on Line 46 we instantiate a variable called *netObj* and then we use the *FindNet* method associated with the *Document* object to find the net called "GND" and assign this net to our *netObj* variable.

```
45      ' Get the net to put the via on
46      Dim netObj: Set netObj = pcbDocObj.FindNet("GND")
```

For the purposes of this script, we do not want our new trace to be associated with any particular component. Thus, on Line 49 we first instantiate a variable called *cmpObj* in which to hold a *Component* object, and then we assign a value of *Nothing* to this variable.

```
48      ' Don't attach it to a component
49      Dim cmpObj: Set cmpObj = Nothing
```

Next, on Line 53, we turn off the automatic error handling so as to enable us to handle any DRC errors from within our script (see also the *Error Handling* topic in *Chapter 2: VBS Primer*).

```
51      ' Turn off error handling so the script
52      ' can handle DRC violations
53      On Error Resume Next
54      Err.Clear
```

On Line 56 we use the *PutVia* method associated with the *Document* object to add a via to the design. In this case, we use the X/Y location generated by the mouse click event along with the other parameters we've collected along the way.

```
55      ' Add the via where the user specified.
56      Call pcbDocObj.PutVia(x, y, pstkObj, netObj, cmpObj)
```

On Lines 59 through 71, we check for any errors and also inform the user as to what has taken place. On Line 59 we use the *Number* property associated with the *Err* object to determine if there was an error. If this value is 0 then there was no error; otherwise, on Line 61 we inform the user about the error by using the *Description* property associated with 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                                pcbStatusFieldError)
```

On Line 64 we use the *Clear* method associated with the *Err* object to clear this error so that we don't see it again on the next click of the mouse button.

```
63      ' Clear the error
64      Err.Clear
```

Alternatively, if there was no error in the first place, we use the statement spanning Lines 67 and 68 to display a message on the status bar informing the user that a via was placed.

```
66      ' No error. Let the user know what happened
67      Call pcbAppObj.Gui.StatusBarText("Placed via.", _
68                                epcbStatusField3)
```

The layout design document we intend to use for this example is called *Candy.pcb*. This design – which contains a padstack called "VIA20" – should be located in your C:\Temp\Candy\PCB folder (see also *Appendix E: Accessing Example Scripts and Designs* with regard to obtaining a copy of this design).

- 1) Close any instantiations of Expedition PCB that are currently running, and then launch a new instantiation of this application.
- 2) Open the *Candy.pcb* document.
- 3) Zoom in on an empty area of the design.
- 4) Drag the *PutVia.vbs* script from your C:\Temp folder and drop it on top of the design. Observe that the "Custom Place Via" text appears in the *Prompt* field on the status bar as illustrated in Figure 12-20 (this was the string parameter we specified in Line 22).

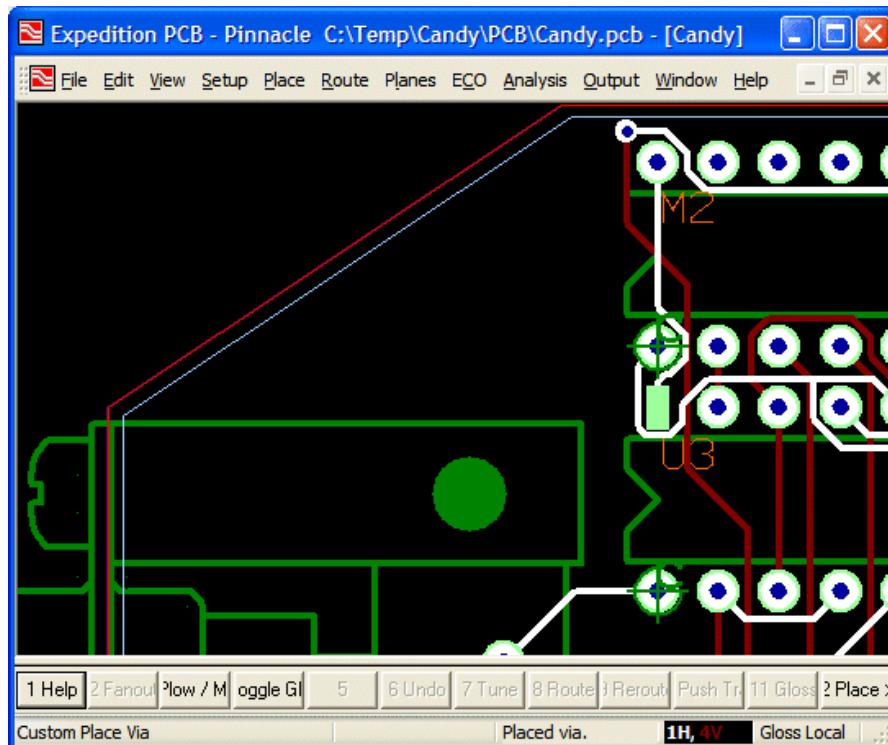


Figure 12-20. Zoom in on a blank area of the design.

- 5) Click your mouse cursor a few times in the blank area of the design and observe a new via appear on each click as illustrated in Figure 12-21.

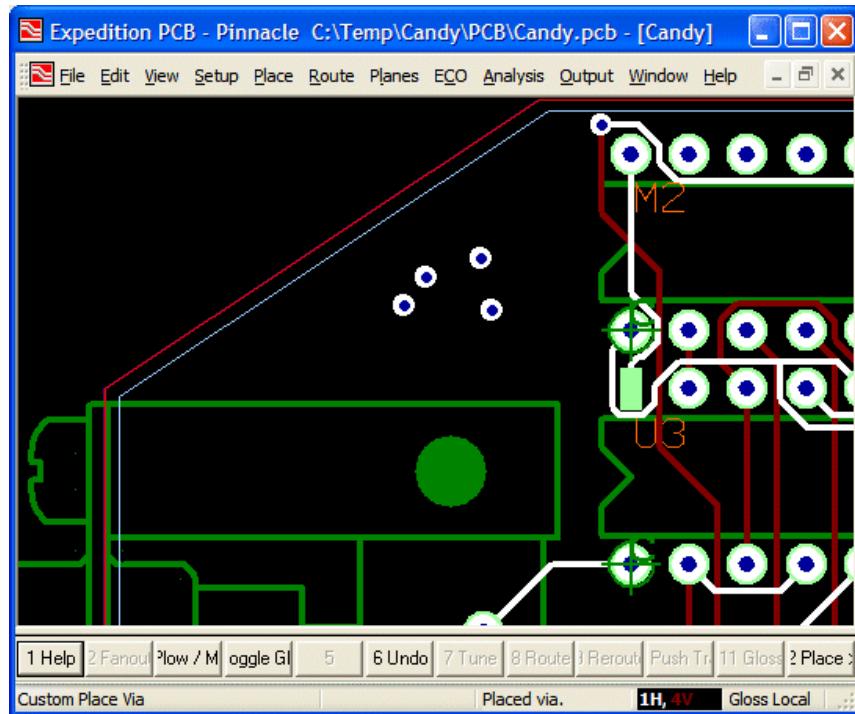


Figure 12-21. Zoom in on a blank area of the design.

- 6) Now try clicking your mouse cursor over an existing pin (say M2-1, for example) and observe an error message appear in the *Error* field of the status bar as illustrated in Figure 12-22.

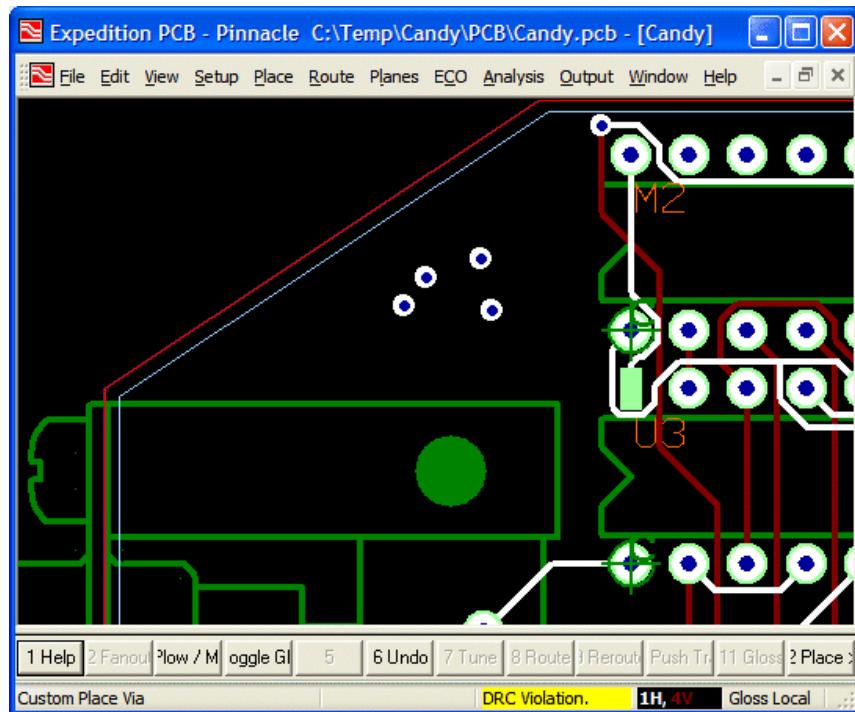


Figure 12-22. An error occurs is we try to place a via on top of an existing pin.

-
- 7) Finally, click the <Esc> key to terminate this script and observe that the "Custom Place Via" text disappears from the *Prompt* field on the status bar.

File Input/Output (I/O)

File Input/Output (I/O) does not have any direct relationship to Expedition PCB Automation. As you can imagine, however, it can be very useful to be able to write information to, or read information from, files on the host system. In this topic we present two examples that involve (a) writing a report to a file and (b) reading and parsing data from a file.



Note: File I/O is not built into the VBScript language. It is actually provided by another COM server, which will need to be invoked by the script. Thus, this same mechanism can be used by any COM-aware language such as JScript.

Creating a Net Report

- Goal:** To use a script to create and output 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:** Use the *FileSystemObject* object to create a new file and to then write data to this file.
- Assumptions:** None.

Use the scripting editor of your choice to open the *Skeleton.vbs* script in your C:\Temp folder, save it 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 we instantiate a variable called *overWriteInt* and assign it a value of 1. In the fullness of time, we will use the value in this variable to cause the system to overwrite any existing file with the same name as the file we want to create.

```
18  Dim overWriteInt: overWriteInt = 1
```

On Line 21 we instantiate a variable called *fileSysObj*, which will be used to hold our *FileSystemObject* object. On Line 22 we use the *CreateObject* function and pass it the *Prog ID* of the *FileSystemObject* object as a parameter, where this *Prog ID* is the string "Scripting.FileSystemObject" (see also the *Terminology* topic in *Chapter 1: Introduction* for more information on the concept of *Prog IDs*).

```
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* (see also *Appendix D: General VBScript References and Tools* for more information on using Object Browsers).

In addition to the *FileSystemObject* object, we also need to create a *TextStream* object. A *TextStream* object is used to actually perform the tasks of reading/writing text from/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 in the form of the *File* object, which is another object associated with the File System Object server.

On Line 23 we instantiate a variable called *txtStreamObj* which will be used to hold our *TextStream* object. On Lines 24 and 25 we use the *CreateTextFile* method associated with the *FileSystemObject* object to create a new file and return the *TextStream* object associated with this file. We then assign this *TextStream* object to our *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. In this example, we form this path by combining two strings. The first string is retrieved from the *Path* property associated with the *Document* object. This is the path to the folder containing the current PCB document. The second string specifies a combination of the *LogFiles* folder and the name of the report file to be created, which is *MyNetReport.txt*.

The second parameter to the *CreateTextFile* method is our *overWriteInt* variable, to which we assigned a value of 1 on Line 18. This value instructs the *CreateTextFile* method to overwrite any existing file with the same name as the file we are attempting to create.

Lines 27 through 30 are used to write a header to our report file. This header consists of a blank line, followed by a line containing the title of the report and the current date², followed by another blank line.

```
27  ' Add a header
28  txtStreamObj.WriteLine()
29  txtStreamObj.WriteLine("My Net Report : " & Date)
30  txtStreamObj.WriteLine()
```

On Line 33 we instantiate a variable called *netColl* in which we intend to hold a collection of *Net* objects; on Line 34 we instantiate a variable called *netObj* to hold a single *Net* object; and on Line 35 we retrieve a collection of *Net* objects and assign them to our *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
```

On Lines 38 through 47, we use a *For Each ... Next* control loop to iterate through each net in our collection. For each net we retrieve, format, and output the data in which we are interested.

```
37  ' Iterate through all nets
38  For Each netObj In netColl
39      ' Report information on each net.
40      txtStreamObj.WriteLine(netObj.Name)
41      txtStreamObj.WriteLine()
42      txtStreamObj.WriteLine("    Opens: " & netObj.FromTos.Count)
43      txtStreamObj.WriteLine()
44      txtStreamObj.WriteLine("    Pins: " & netObj.Pins.Count)
45      txtStreamObj.WriteLine()
46      txtStreamObj.WriteLine()
47  Next
```



Note: The *Write* statement simply outputs its associated data to the file. By comparison, the *Writeline* statement writes the same data and then automatically appends a new line character (or new line characters) on the end. Thus, Lines 42 and 43 could have been combined into a single *Writeline* statement (we showed them separated into two statements here only for the purposes of this example). Alternatively, we could delete the *Writeline* statement on Line 43 and append a *VbCrLf* ("carriage return, line feed") constant to the end of the string being outputted. The same thing applies to the two statements on Lines 44 and 45.

Finally, on Line 50, we use the *Close* method associated with the *TextStream* object to close the file.

```
49  ' Close the file
50  txtStreamObj.Close()
```

- 1) Close any instantiations of Expedition PCB that are currently running, and then launch a new instantiation of this application.
- 2) Open the *Candy.pcb* design we've been using throughout these building block examples.
- 3) Drag the *NetReport.vbs* script from your *C:\ITemp* folder and drop it on top of the design.
- 4) Use your browser to locate the following file:

² See the *Conversion and Formatting Functions* topic in *Chapter 2: VBS Primer* for more information on the *Date* function.

C:\Temp\Candy\Pcb\LogFiles\MyNetlist.txt

- 5) Open this file in the text editor of your choice and observe its contents, which should look something like the following:

```
My Net Report : 2/5/2007

GND
    Opens: 0
    Pins: 18

VCC
    Opens: 0
    Pins: 19

Battery
    Opens: 0
    Pins: 6
    :
etc.
```

Reading a Points Array from a File

Goal: To open a text file, to read and parse its contents, and to use these contents to create a points array.

Approach: Use the *FileSystemObject* object to open an existing file and to then read data from this file

Assumptions: The design contains a user layer called *Test*; also, a file called *PointsArrayString.txt* is created and placed in your C:\Temp folder prior to this script being run.

First, 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. (Observe that this text describes the same X-Y-R values used in the points array featured in the *Put User Layer Graphics* script presented earlier in this chapter.)

```
(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)
```

Next, use the scripting editor of your choice to open the *Skeleton.vbs* script in your C:\Temp folder, save it 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.ActiveView.SetExtentsToSelection()
71 End If
72
73 '.....
74 'Local functions
75
76 ' Creates a points array from a string
77 Function GetPointsArrayFromString(inputStr)

```

```

78      ' Split string into a multiple strings. One for each line
79      Dim linesArr: linesArr = Split(inputStr, vbCrLf)
80
81      ' Instantiate the array to return
82      Dim pntsArr
83      Redim pntsArr(2, UBound(linesArr))
84
85      ' Loop through each line reading the values
86      Dim i
87      For i = 0 To UBound(linesArr)
88          ' Split on the comma to find each value
89          Dim strArr: strArr = Split(linesArr(i), ", ")
90          Dim xReal: xReal = CDbl(Split(strArr(0), "(")(1))
91          Dim yReal: yReal = CDbl(strArr(1))
92          Dim rReal: rReal = CDbl(Split(strArr(2), ")")(0))
93
94          ' Put the values in the return array.
95          pntsArr(0,i) = xReal
96          pntsArr(1,i) = yReal
97          pntsArr(2,i) = rReal
98
99      Next
100
101     ' Return points array
102     GetPointsArrayFromString = pntsArr
103
104 End Function

```

On Line 19, we use the *AddTypeLibrary* method associated with 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: We didn't include this statement in the previous example, because we knew that we were not going to use and enumerates associated with the *FileSystemObject* object in that example.



Note: It would actually be common practice to add all of the type libraries required by a script in the same location at the beginning of the script. The only reason we didn't do that in this case is because we are basing all of these building-block examples on the common *Skeleton.vbs* script.

On Line 22 we instantiate a variable called *fileSysObj* which will be used to hold our *FileSystemObject* object. On Line 23 we use the *CreateObject* function and pass it the *ProgID* of the *FileSystemObject* object as a parameter.

```

21  ' Create a FileSystemObject
22  Dim fileSysObj
23  Set fileSysObj = CreateObject("Scripting.FileSystemObject")

```

On Line 26 we instantiate a variable called *pntsFileStr* and we then assign the name of the file with which we will be working to this variable.

```

25  ' Create variable to hold name of input file
26  Dim pntsFileStr: pntsFileStr = "c:/temp/PointsArrayString.txt"

```



Note: In the previous example we hard-coded the name of the file as part of the function call. In the case of this script, however, we will need to use the same file name several times, so it makes more sense to assign the name of the file to a variable and then use this variable throughout the rest of the script. This allows us to modify the name of the file in the future by making a single change to the script (should we want to do so).

On Line 29 we use the *FileExists* method associated with the *FileSystemObject* object to determine whether or not the file in which we are interested actually exists. If the file does not exist, then we use the statement on Line 30 to report an error; otherwise, we proceed to the *Else* case on Line 31.

```
29  If Not fileSysObj.FileExists(pntsFileStr) Then  
30      MsgBox("Can't find file " & pntsFileStr & ".")
```

On Line 33 we instantiate a variable called *txtStreamObj* which will be used to hold our *TextStream* object. On Line 34 we use the *OpenTextFile* method associated with the *FileSystemObject* object to create a new file and return the *TextStream* object associated with this file. We then assign this *TextStream* object to our *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 file, which includes the name of the file. In this example, we are using the value stored in the *pntsFileStr* variable we declared on Line 26.

The second parameter to the *OpenTextFile* method is the enumerate *ForReading*. It probably comes as no great surprise to discover that this enumerate is used to inform the *OpenTextFile* method that we want to open this file for reading. Other related enumerates we might want to use are *ForAppending* and *ForWriting* (this latter case is different to the previous script in which we used the *CreateTextFile* method to create a new file from scratch – by comparison, the *ForWriting* enumerate informs the *OpenTextFile* method that we want to open an *existing* file and then overwrite its contents).

On Line 37 we instantiate a variable called *pntsArrStr*. Also on this line, we use the *ReadAll* method associated with the *TextStream* object to read the entire contents of our text file, and we assign the result to our *pntsArrStr* variable.

```
36      ' Read the whole file.  
37      Dim pntsArrStr: pntsArrStr = txtStreamObj.ReadAll
```

Since we've now read everything out of the file, we no longer require it to remain open, so on Line 39 we use the *Close* method associated with the *TextStream* object to close the file.

```
38      ' Close the file  
39      txtStreamObj.Close()
```

On Line 42 we instantiate a variable called *pntsArr*, which we will use to hold our points array. Also on this line, we call the function *GetPointsArrayFromString* and we pass this function our *pntsArrStr* variable as a parameter.

```
41      ' Convert the string to a points array
```

```
42     Dim pntsArr: pntsArr = GetPointsArrayFromString(pntsArrStr)
```

Note that *GetPointsArrayFromString* is a user-defined function that we are going to create shortly. For the moment we need only understand that this function returns a points array, which we assign to our *pntsArr* variable.

Lines 44 through 70 are taken directly from our *Put User Layer Graphics* script earlier in this chapter. These lines simply generate a group of parameters required to create a graphic element using the points array, actually generate the graphic, and zoom in on the graphic 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.ActiveView.SetExtentsToSelection()
```

Lines 77 through 104 describe the *GetPointsArrayFromString* function. On Line 77 we declare the function name and also the name of its single parameter, which we've decided to call *inputStr*.

```
77     Function GetPointsArrayFromString(inputStr)
```

On Line 80 we instantiate a variable called *linesArr*. We are going to use this variable to hold an array of strings, where each string is a line from our original text file. Also on Line 80, we use the VBScript *Split* function to split the string we read in from our text file into an array of strings, and to assign this array of strings to our *linesArr* variable.

```
79     ' Split string into a multiple strings. One for each line
80     Dim linesArr: linesArr = Split(inputStr, vbCrLf)
```

Now, this is slightly tricky to describe, so let's take it step-by-step. When we used a text editor to create our original *PointsArrayString.txt* file, it looked like the following on the screen:

```
(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 carriage-return and line-feed characters, which are – in turn – represented by *vbCrLf* constants as follows:

(700,1950,0)vbCrLf(650,1950,50)vbCrLf(650,1900,0)...etc.
--

The point is that the carriage-return and line-feed characters are present in both the text editor and in the string we read into our script; it is just that they are displayed as new lines in the text editor and they exist as *vbCrLf* constants in the string.

The *Split* function always returns an array (this aspect of this function will become important to us shortly). When we call the *Split* function on Line 80, we pass it two parameters: the first is a string variable containing the entire contents of our original text file; the second is the string or string constant we want to use as the target upon which to split the original string (in this case we are using the *vbCrLf* string constant).

Every time the *Split* function detects a *vbCrLf* constant it splits the string at that point (it also discards any copies of the character/string used to control the split – *vbCrLf* in this case). The result is an array of strings, which we 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)

Now, this is where things start to get interesting. On line 83 we instantiate a variable called *pntsArr*, which we will use to hold our points array. On Line 84 we re-dimension this variable with two dimensional parameters. The first parameter says that each row in our array will contain 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 our array of strings.

```

82      ' Instantiate the array to return
83      Dim pntsArr
84      Redim pntsArr(2, UBound(linesArr))

```



Note: The name we use to instantiate this points array on Line 83 is the same name we used for the variable we instantiated in the body of the script on Line 42. However, the fact that we've re-instantiated it here with a new *Dim* statement (as opposed to simply using the existing variable name) means that this new copy is local to this function and distinct from the original variable used in the body of the script.

On Line 87 we instantiate a variable called *i*, which we use to iterate around the *For* loop declared on Line 88. This loop is going to be used to iterate through each item in our array of strings.

```
87      Dim i  
88      For i = 0 To UBound(linesArr)
```

As most programmers quickly come to appreciate, parsing text files can be surprisingly tricky and time-consuming. Consider the first iteration around the loop. In this case, we know that the first item in our array of strings looks like the following:

(700,1950,0)

On Line 90 we declare a variable called *strArr*. Also on this line, we call the *Split* function and ask it to split the contents of the current item in our string array using commas as the characters to split on.

```
89          ' Split on the comma to find each value  
90          Dim strArr: strArr = Split(linesArr(i), ",")
```

This will result in our *strArr* variable containing three items as follows:

(700
1950
0)

The first item (700 will be stored in *strArr(0)*, the second item 1950 will be stored in *strArr(1)*, and the third item 0) will be stored in *strArr(2)*.

Once again, observe that when the *Split* function performs its magic, it discards any copies of the character/string used to control the split (the comma character in this case). The reason this is important to us here is that – as we previously noted – the *Split* function always returns an array.

In order to understand what this means, consider what happens when we call the *Split* function on Line 91.

```
91          Dim xReal: xReal = CDbl(Split(strArr(0), "( )(1))
```

We start by instantiating a variable called *xReal* (we consider this to be a real number because we don't know whether the user will specify values as integers or reals). Also on this line, we ask the *Split* function to use the "(" character to split the (700 stored in *strArr(0)*). The result will be an array containing two items as follows:

700

The first (empty) item has an index value of (0), while the second item – 700 – has an index value of (1). This explains why we use the (1) located toward the right-hand side of the

statement on Line 91 to access the second item – 700 in this example – in the resulting array. We then cast the result as a real value and assign this to our *xReal* variable.

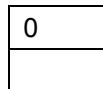
By comparison, Line 92 is relatively simple, because all we have to do is to cast the contents of *strArr(1)* as a real and assign the result to the *yReal* variable.

```
92      Dim yReal: yReal = CDbl(strArr(1))
```

Now consider the statements on Line 93:

```
93      Dim rReal: rReal = CDbl(Split(strArr(2), " ")(0))
```

In this case, we start by instantiating a variable called *rReal*. Also on this line we ask the *Split* function to use the ")" character to split the 0) stored in *strArr(2)*. The result will be an array containing two items as follows:



The first item – 0 in this case – has an index value of (0), while the second (empty) item has an index value of (1). This explains why we use the (0) located toward the right-hand side of the statement on Line 91 to access the first item – 0 in this case – in the resulting array. We then cast the result as a real value and assign this to our *rReal* variable.

On Lines 96 through 98, we take the X, Y, and R values we just extracted from this item in our array of strings and we assign these values to the appropriate locations in our points array.

```
96      pntsArr(0,i) = xReal
97      pntsArr(1,i) = yReal
98      pntsArr(2,i) = rReal
```

We then repeat the above – iterating around our *For ... Next* loop – for each item in our array of strings. Finally, on Line 102, we return the points array, and on Line 104 we terminate this function.

```
101      ' Return points array
102      GetPointsArrayFromString = pntsArr
103
104  End Function
```

The layout design document we intend to use for this example is called *Candy.pcb*. This design already contains a user layer called "Test" (see also *Appendix E: Accessing Example Scripts and Designs* with regard to obtaining a copy of this design).

- 1) Close any instantiations of Expedition PCB that are currently running, and then launch a new instantiation of this application.
- 2) Open the *Candy.pcb* document. Use the **View > Display Control** menu command, select the **General** tab, and then ensure that the check box next to the user layer called "Test" is turned on.
- 3) Drag the *ReadPointsArray.vbs* script from your C:\Temp folder, drop it on top of the design, and observe the resulting shape appear as illustrated in Figure 12-23.

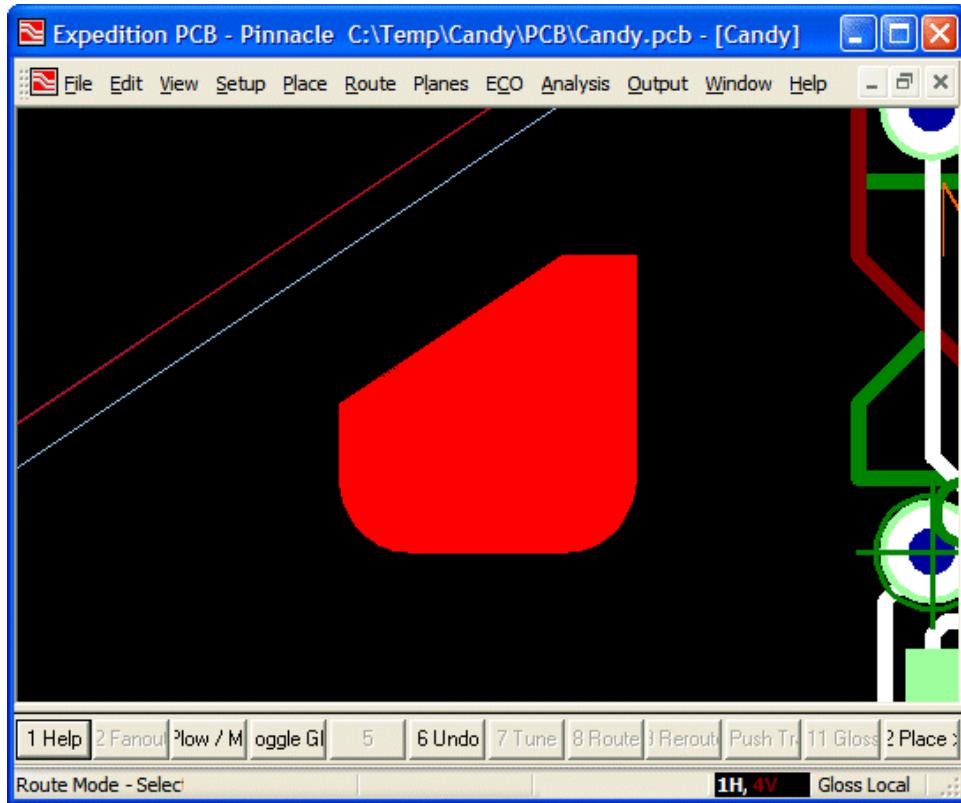


Figure 12-23. Using the `ReadPointsArray.vbs` script to read a text file and generate/display a graphical element.

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.

Using NotePad to View a Net Report

Goal: To generate a net report as described in the *Creating a Net Report* script earlier in this chapter, and to then use the Expedition PCB Automation Interface to launch the NotePad® editor in order to view this report.

Approach: To use the *Exec* object from the *Viewlogic* COM server to execute a system command that launches the NotePad editor.

Assumptions: That we are running under the Windows® operating system; that the NotePad.exe application is available; and that we've already created the *NetReport.vbs* script as described in the *Creating a Net Report* topic earlier in this chapter.

Use the scripting editor of your choice to open the *NetReport.vbs* script in your *C:\Temp* folder, save it as *ViewNetReport.vbs*. The portion of this script of interest to us here 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 we did in this script occurred on Line 50; this is where we closed our *MyNetReport.txt* file after we'd finished outputting it. Now 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")

```

On Line 53 we instantiate a variable called *execObj*, which we will use to hold our *Exec* object (where *Exec* is short for "Execute" and refers to an object that can run system commands). On Line 54 we call the *CreateObject* function that is built in to VBScript, and we pass it the "Viewlogic.Exec" Prog ID.

```

53 Dim execObj
54 Set execObj = CreateObject("Viewlogic.Exec")

```



Note: Be careful anytime you use *CreateObject* to access a COM server, because this is machine-specific. Even when moving from one Windows® machine to another, the two machines could have a different set of COM servers. The reason we know the "Viewlogic.Exec" COM server is available is that it is delivered with Mentor's products.

On Line 56 we use the *WorkingDirectory* of the *Exec* object to set the working directory/folder to the directory/folder containing the *MyNetReport.txt* net report file.

```

55 ' Set the working directory
56 execObj.WorkingDirectory = pcbDocObj.Path + "LogFiles/"

```

On Line 58, we use the *Run* method associated with the *Exec* object to execute the *notepad.exe* application and to pass it the name of our net report file – *MyNetReport.txt* – as an argument.

```
57  ' Open notepad on the net report.  
58  Call execObj.Run("notepad.exe MyNetReport.txt")
```



Note: You can use the *Exec* object to execute any command line string. Following on from this, you can substitute the *notepad.exe* application for the editor of your choice.

- 1) Close any instantiations of Expedition PCB that are currently running, and then launch a new instantiation of this application.
- 2) Open the *Candy.pcb* design we've been using throughout these building block examples.
- 3) Drag the *NetReport.vbs* script from your *C:\Temp* folder and drop it on top of the design.
- 4) Observe that as soon as the net report has been generated, this new script launches the NotePad editor and uses this editor to display your *MyNetReport.txt* file as illustrated in Figure 12-24.

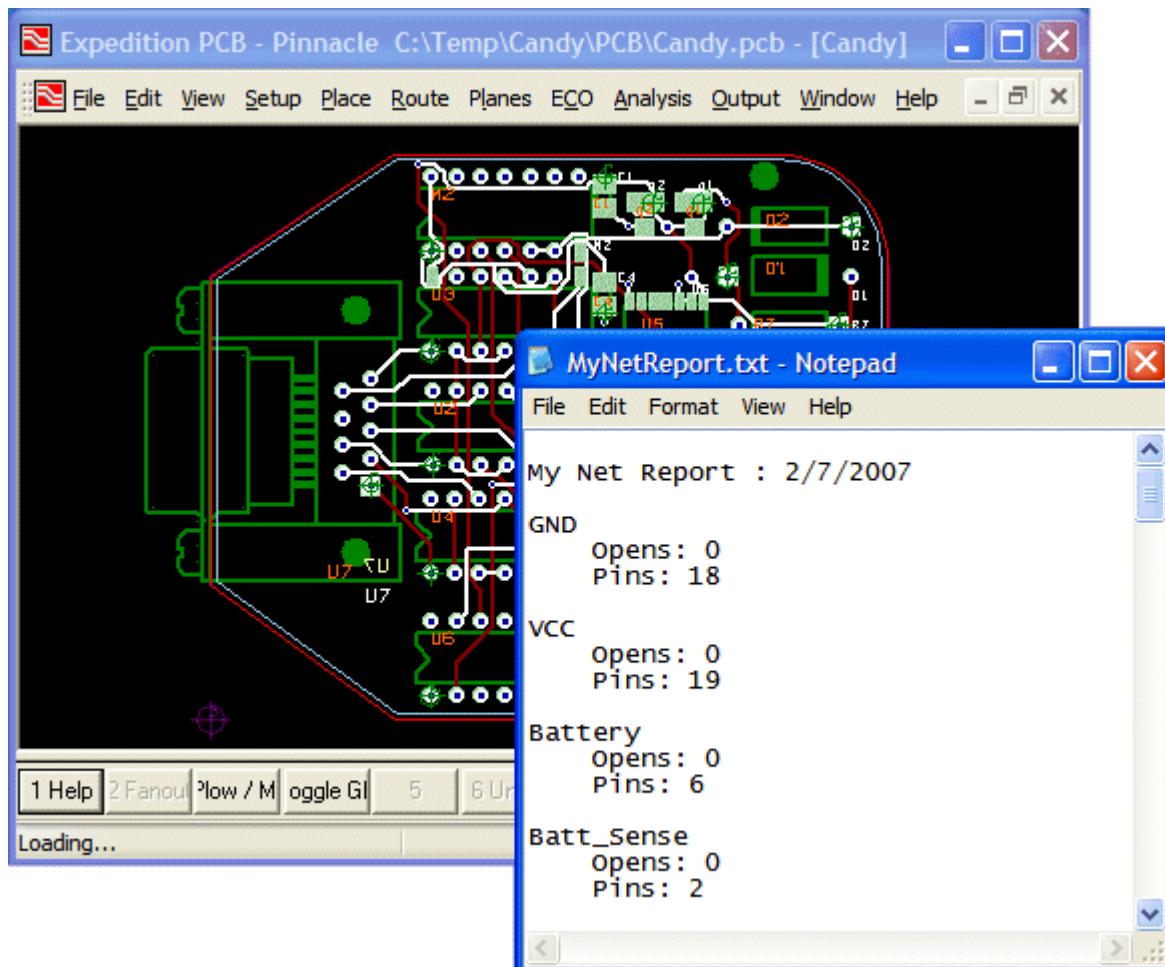


Figure 12-24. The *ViewNetReport.vbs* script launches the NotePad editor.

Compressing a File and Emailing the Result

Goal: To generate a net report as described in the *Creating a Net Report* script earlier in this chapter, and to then use the Expedition PCB Automation Interface to compress and email 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 email.

Assumptions: That we are running under the Windows® operating system because the *Outlook®* COM server is not available on UNIX/Linux systems. Also that we've already created the *NetReport.vbs* script as described in the *Creating a Net Report* topic earlier in this chapter.

Use the scripting editor of your choice to open the *NetReport.vbs* script in your *C:\Temp* folder, save it as *ZipAndMailNetReport.vbs*. The portion of this script of interest to us here 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 we did in this script occurred on Line 50; this is where we closed our *MyNetReport.txt* file after we'd finished outputting it. Now 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  ' Tell the user what is going on
59  Call pcbAppObj.Gui.StatusBarText("Zipping net report...", 
60                                     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"""
66          "-min -a -r NetReport.zip MyNetReport.txt", visibleInt, waitBool)
67
68  ' Tell the user what is going on
69  Call pcbAppObj.Gui.StatusBarText("Creating email...", 
70                                     epcbStatusField1)
71
72  ' Create an outlook object
73  Dim outlookObj
74  Set outlookObj = CreateObject("Outlook.Application")
75
76  ' Create a new email
77  ' AddTypeLibrary doesn't work so hardcode the enumerate
78  Const olMailItem = 0
79  Dim mailObj
80  Set mailObj = outlookObj.CreateItem(olMailItem)
81
82  ' Setup the email
83  mailObj.Subject = "Net report for " & pcbDocObj.Name
84  mailObj.Attachments.Add(pcbDocObj.Path +
85                           "LogFiles/NetReport.zip")
86  mailObj.Display
87
88  ' Let the user know we have finished.
89  Call pcbAppObj.Gui.StatusBarText("Processing complete.", 
90                                     epcbStatusField1)

```

On Line 53 we instantiate a variable called `execObj` to hold our `Exec` object. On Line 54 we call the `CreateObject` function and we pass it the "Viewlogic.Exec" Prog ID.

```

53  Dim execObj
54  Set execObj = CreateObject("Viewlogic.Exec")

```

On Line 56 we use the `WorkingDirectory` of the `Exec` object to set the working directory/folder to the directory/folder containing the `MyNetReport.txt` net report file.

```

55  ' Set the working directory
56  execObj.WorkingDirectory = pcbDocObj.Path + "LogFiles/"

```

On Line 60 we display a message on the Expedition PCB status bar to inform the user as to what we are doing.

```

59  ' Tell the user what is going on
60  Call pcbAppObj.Gui.StatusBarText("Zipping net report...", 
61                                     epcbStatusField1)

```

When we use the `Exec` object to execute a command, it automatically invokes a console / terminal window, and it uses this window to actually execute the command. On Line 63 we instantiate a variable called `visibleInt` and we assign a value of 0 to this variable. As we shall see, this value will be used to inform the system that we do not want to see the console / terminal window appear on the screen (the reason we didn't do this in the previous script is that this is the default, we're only doing it here so that you can see how it works – if you do

want to see the console / terminal window, simply set the value associated with the *visibleInt* variable to be 1).

```
62 ' Zip the file in the Output directory
63 Dim visibleInt: visibleInt = 0
```

On Line 63 we instantiate a variable called *waitBool* and we assign a Boolean value of *True* to this variable. As we shall see, this value will be used to inform the system that we want to wait for the *Run* method on Line 65 to finish performing its actions before we proceed with the remainder of the script.

```
64 Dim waitBool: waitBool = True
```

On Line 65, we use the *Run* method associated with the *Exec* object to execute the *WinZip32.exe* application and pass it the following arguments:

- -min Run WinZIP in the background (not visible)
- -a Add the specified file to the compressed file
- -r Include subfolders (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)
```



Note: You can Google® "WinZip32" for more details on these (and other) arguments. In the case of the input file(s) to be compressed, we could specify a space-separated list of file names (we could also use regular expressions like *.txt).

After the file has been compressed, on Line 68 we display a message on the Expedition PCB status bar to inform the user as to what we are doing now.

```
67 ' Tell the user what is going on
68 Call pcbAppObj.Gui.StatusBarText("Creating email...", 
                                   epcbStatusField1)
```

On Line 71 we instantiate a variable called *outlookObj* to hold our *Outlook* object. On Line 72 we call the *CreateObject* function, we pass it the "Outlook.Application" Prog ID, and we assign the resulting *Outlook* object to our *outlookObj* variable.

```
70 ' Create an outlook object
71 Dim outlookObj
72 Set outlookObj = CreateObject("Outlook.Application")
```

After investigation, it was discovered that – for some unknown reason – adding the Microsoft® type library to the script did not provide access to the Outlook enumerates. In order to circumvent this problem, on Line 76 we declare a constant called *olMailItem*, to which we assign a value of 0. The reason we did this is that – looking at the documentation for Outlook – the *CreateItem* method associated with the *Outlook Application* object takes an enumerate as a parameter. We used the documentation to determine that a value of 0 instructs the *CreateItem* method to create a *Mail* object (as opposed to a *Calendar* object or some other Outlook-related object).

On Line 77 we instantiate a variable called *mailObj* to hold a *Mail* object. On Line 78 we use the *CreateItem* method associated with the *Outlook Application* object to create a new *Mail* object, and we then assign this object to our *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)
```

On Line 81, we use the *Subject* property associated with the *Mail* object to assign text to the **Subject** field. On Line 82 we use the *Add* method of the *Attachments* collection returned by the *Attachments* property associated with the *Mail* object to add an attachment to our email (we do this by passing a parameter that specifies the path to our 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")
```

On Line 83, we use the *Display* method associated with the *Mail* object to display the email on the screen so as to allow the user to fill in the **To** and **CC** fields and also add a message to the body of the email if required (all of this could be done in the script if we knew this information beforehand).

```
83  mailObj.Display
```

Finally, on Line 86 we display a message on the Expedition PCB status bar to inform the user that we've finished.

```
85  ' Let the user know we have finished.
86  Call pcbAppObj.Gui.StatusBarText("Processing complete.",  
                                   epcbStatusField1)
```

- 1) Close any instantiations of Expedition PCB that are currently running, and then launch a new instantiation of this application.
- 2) Open the *Candy.pcb* design we've been using throughout these building block examples.
- 3) Drag the *ZipAndMailNetReport.vbs* script from your *C:\Temp* folder and drop it on top of the design.
- 4) Observe that as soon as the net report has been generated, this new script compresses the file and composes the email as illustrated in Figure 12-25.

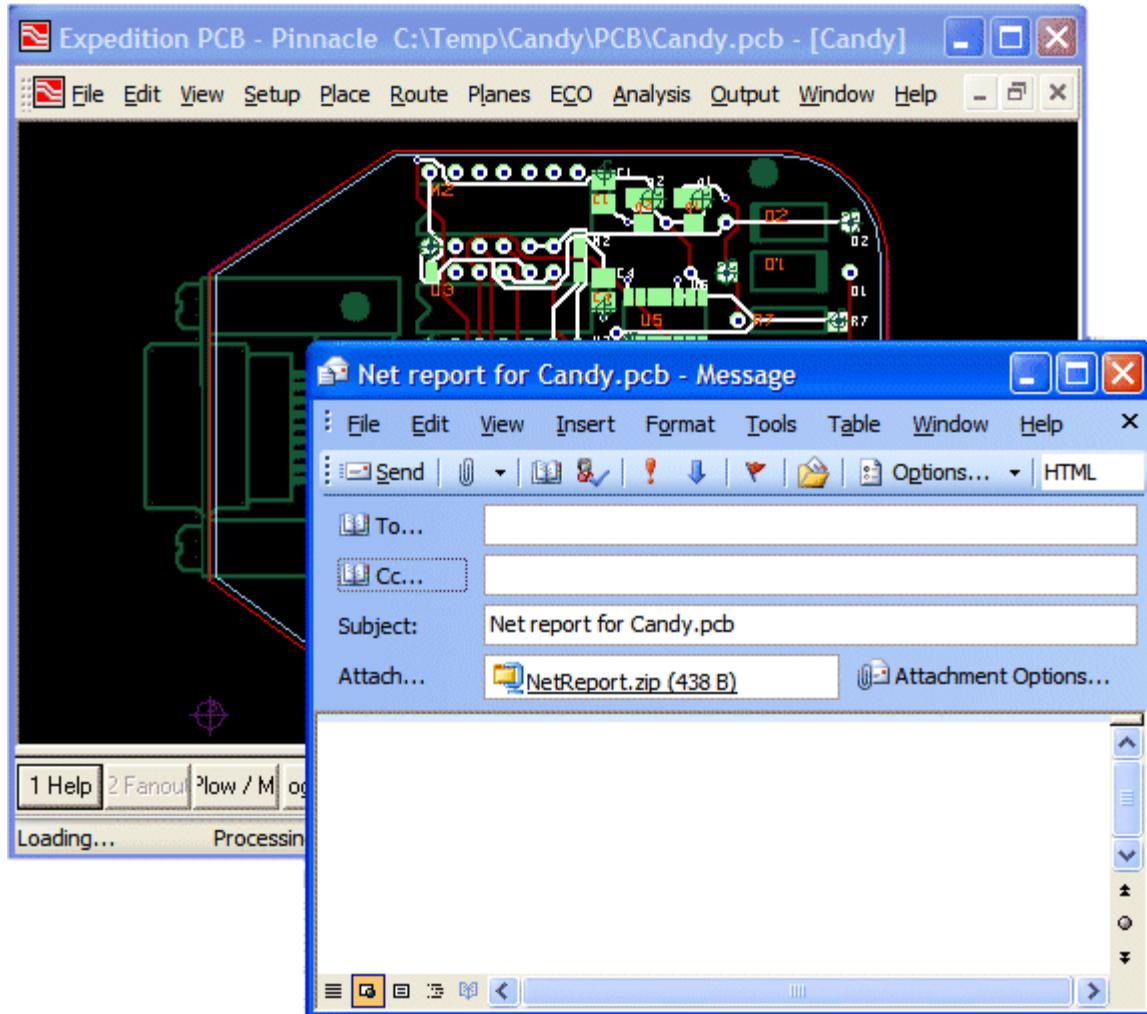


Figure 12-25. The ZipAndMailNetReport.vbs script creates an email.

Reading, Analyzing, and Updating a Design

It is very common to create a script that reads information out of a design, analyzes this information, and then modifies and/or augments the design with new data.

Generate Via Caps

- Goal:** To create graphics indicating via locations on a user layer called "Test".
- Approach:** 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 called "Test" exists in the design.

Use the scripting editor of your choice to open the *Skeleton.vbs* script in your C:\Temp folder, save it 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 =
59         pcbAppObj.Utility.CreateCircleXYR(viaObj.PositionX, _
60                                         viaObj.PositionY, radiusDbl)
61     Dim numPntsInt
62     numPntsInt = UBound(pntsArr, 2) + 1
63
64     ' Add the graphics to the user layer
65     Call pcbDocObj.PutUserLayerGfx(userLayerObj, widthReal, _
66                                     numPntsInt, pntsArr, filledBool, _
67                                     cmpObj, epcbUnitCurrent)
68 Next
69 Call pcbAppObj.Gui.StatusBarText("Added " & viaColl.Count & _
70                                 " via caps on user layer ""Test""", epcbStatusField1)
71
72 pcbDocObj.TransactionEnd

```

On Line 19 we assign the *epcbUnitMils* enumerate to the *CurrentUnit* property associated with the active *Document* object.

```
18  ' Set the unit to be used
19  pcbDocObj.CurrentUnit = epcbUnitMils
```

On Line 23, we start a transaction so as to ensure that any changes made by our script will be gathered into a single **Undo** command/level.

```
21  ' Start a transaction so that all changes
22  ' are on a single undo level
23  pcbDocObj.TransactionStart
```

On Line 26 we instantiate a variable called *usrLayerGfxColl*, which will be used to hold a collection of graphic objects. On Line 27 we retrieve all of the graphics objects associated with the user layer called "Test" and we assign them to our collection. On Line 29 we delete all of the graphics from the collection (we do this to ensure that we won't end up with duplicate graphics if we rerun our script).

```
25  ' Delete all the existing gfx on Test
26  Dim usrLayerGfxColl
27  Set usrLayerGfxColl = pcbDocObj.UserLayerGfxs(epcbSelectAll,
                                                 "Test")
28
29  usrLayerGfxColl.Delete
```

On Line 32 we instantiate a variable called *vaiColl*, which will be used to hold a collection of vias. On Line 33 we retrieve all of the vias in the design and assign them to our collection.

```
31  ' Get the collection of Vias
32  Dim viaColl
33  Set viaColl = pcbDocObj.Vias
```

On Lines 35 through 51 we acquire/collect/specify all of the parameters we're going to need in order to specify our graphics, including the user layer "Test" (Lines 38 and 39), the fact that we want these graphics to be filled (Line 42), the width associated with these graphics (Line 45), the fact that these graphics are not to be associated with any particular component (Line 48), and the radius of the graphics, which are to be circles (Line 51).

```
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
```

On Line 54 we instantiate a variable called *viaObj*, which will be used to hold a single *Via* object. Then on Lines 55 through 67 we loop around iterating through each via in the collection; for each via we add a circular graphic to the user layer "Test".

On Line 57 we instantiate a variable called *pntsArr*, which will be used to hold a points array. On Line 58 we use the *CreateCircleXYR* method associated with the *Utility* object to generate our points array (observe that we use the *PositionX* and *PositionY* properties associated with the current *Via* object to determine its X/Y location).

On Lines 60 and 61 we determine the number of points in the points array. On Line 64 we use the *PutUserLayerGfx* method associated with the *Document* object to add a graphic to our 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 =
59      pcbAppObj.Utility.CreateCircleXYR(viaObj.PositionX, _
60                                         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
```

On Line 69 we display a message on the Expedition PCB status bar to inform the user as to how many via caps we've added to user layer "Test".

```
69  Call pcbAppObj.Gui.StatusBarText("Added " & viaColl.Count & _
70                                     " via caps on user layer ""Test""", epcbStatusField1)
```

Finally, on Line 72, we end the transaction we initiated on Line 23.

```
72  pcbDocObj.TransactionEnd
```

- 1) Close any instantiations of Expedition PCB that are currently running, and then launch a new instantiation of this application.
- 2) Open the *Candy.pcb* design we've been using throughout these building block examples.
- 3) Use the **View > Display Control** menu command, select the **General** tab, and then ensure that the check box next to the user layer called "Test" is turned on.
- 4) Zoom in on a portion of the design containing vias as illustrated in Figure 12-26.

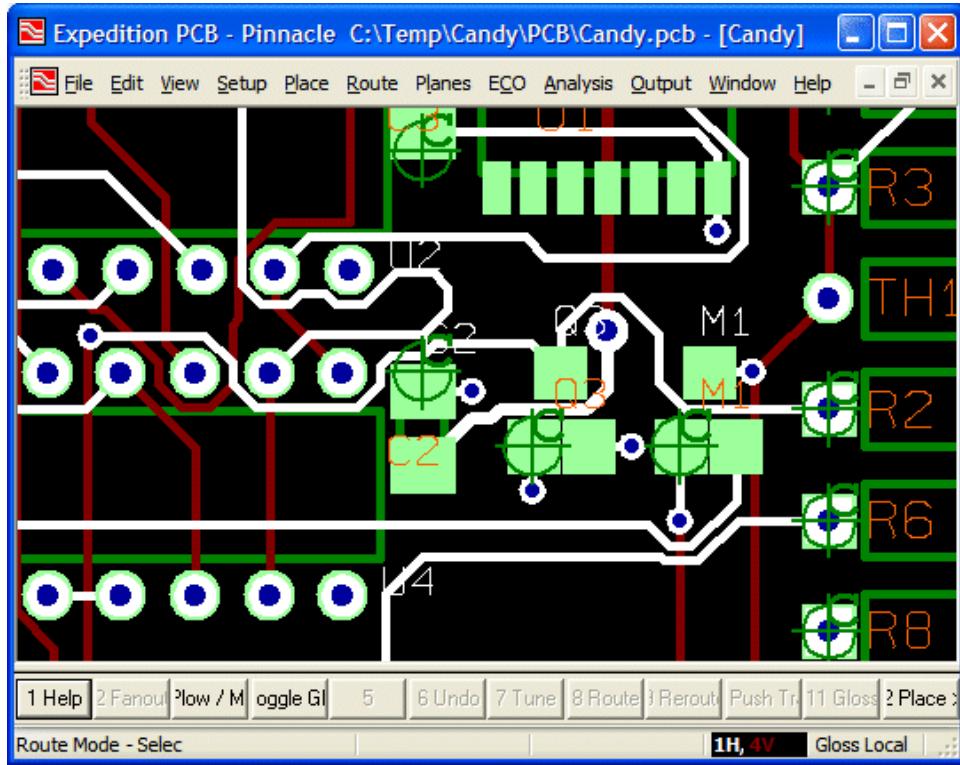


Figure 12-26. Zoom in on a portion of the design containing vias.

- 5) Drag the *ViaCaps.vbs* script from your C:\Temp folder, drop it on top of the design, and observe the via cap graphics appear as illustrated in Figure 12-27.

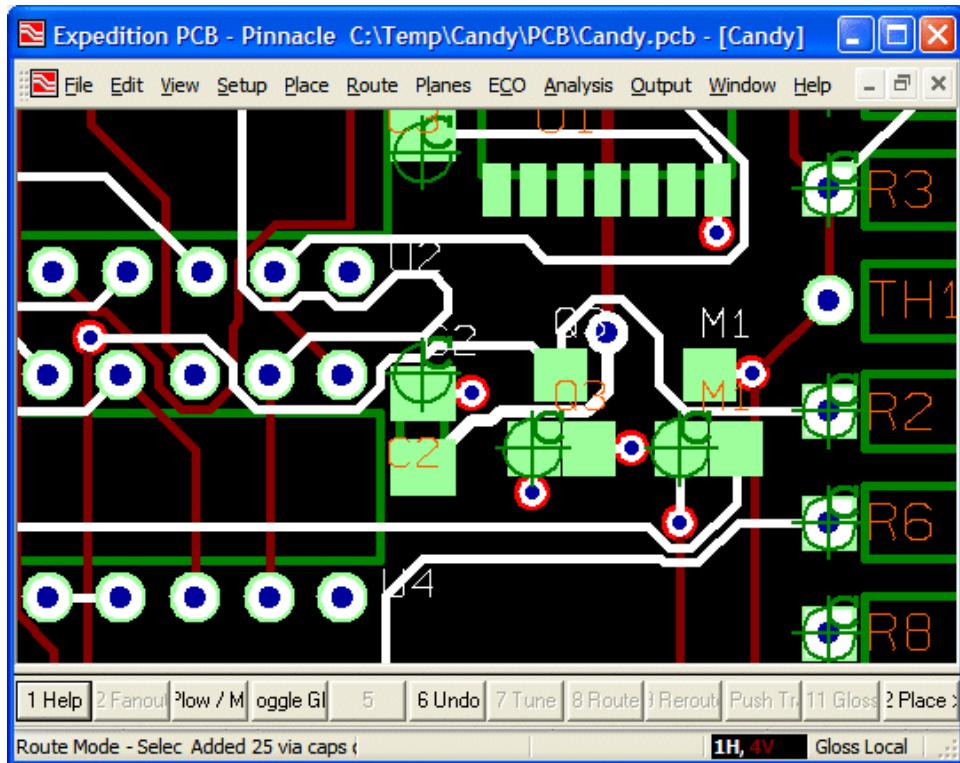


Figure 12-27. The *ViaCap.vbs* script adds via caps to the user layer called "Test".

-
- 6) In order to see these new graphics more clearly, use the **View > Display Control** menu command to turn off vias, holes, traces, pads, and so forth as illustrated in Figure 12-28.

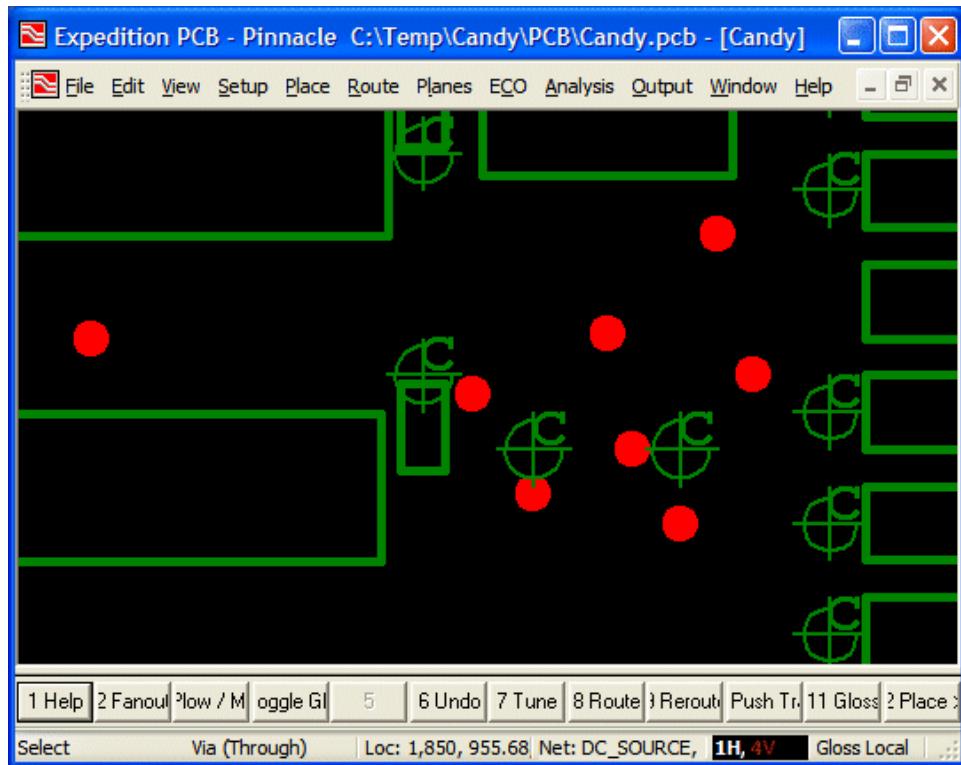


Figure 12-28. Turning other graphics off makes the new via caps easier to see.

SECTION 2: DETAILED EXAMPLES – EXPEDITION PCB

Chapter 13: Display a Single Routing Layer

Introduction

- Overview:** This script allows the user to easily enable the visibility of a single routing layer (and automatically disable the visibility of any other routing layers).
- Approach:** To automatically build and add a collection of menu entries and accelerators that modify the **View > Display Control** settings.
- Points of Interest:**
- Using helper functions to add menu entries
 - Dynamic (run-time) code generation
- Items Used:**
- Command Bar Server
 - Key Bind Server
 - LayerObject Object
 - DisplayControl Object

Before We Start

Different board designs may have different numbers of routing layers. This script searches through the design database to identify the various routing layers. The script adds a series of buttons/items to the **View** pull-down menu – one for each routing layer (the name of each button/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 buttons/items or accelerator keys will cause the associated routing layer to be displayed and all of the other routing layers to be disabled.

Just to give ourselves a feel for the way in which this will work, assume that you've just launched Expedition PCB and opened a layout design document. If you click on the **View** pull-down menu, you should see something similar to that shown in Figure 13-1.

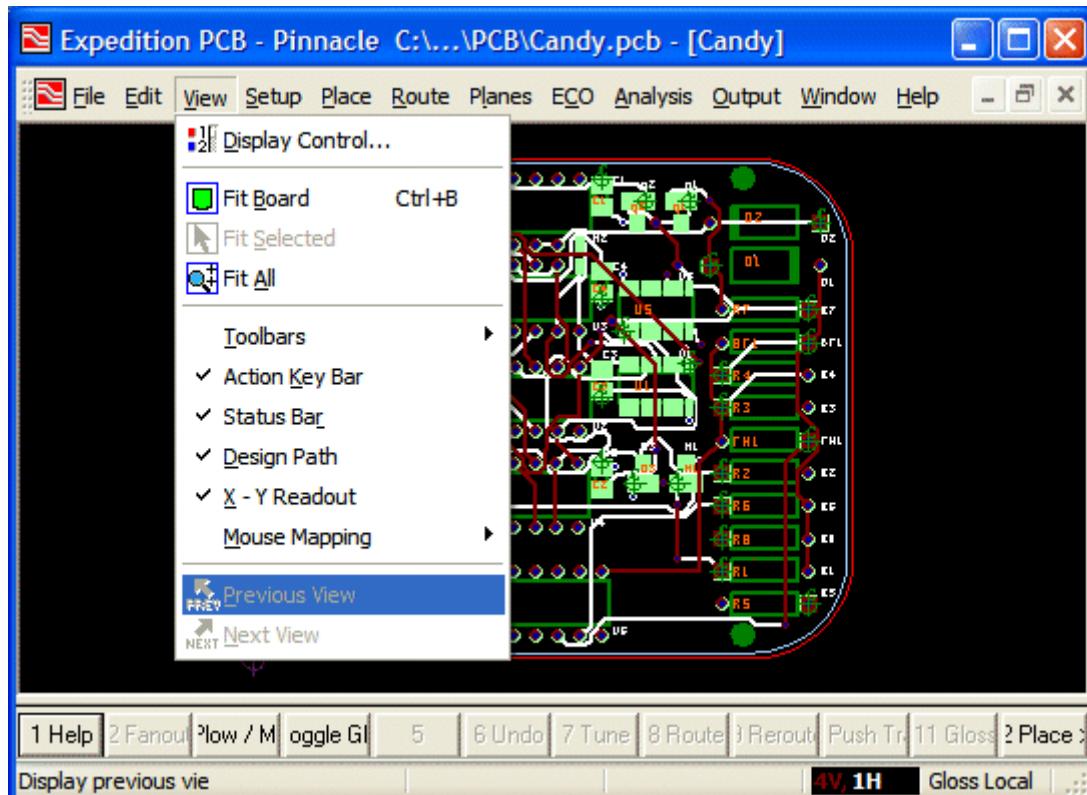


Figure 13-1. Buttons/items under the **View** menu before the script is run.

Observe that the **Previous View** button/item currently follows the **Mouse Mapping** button/item.

Now assume that you execute the script we're about to create and access the **View** pull-down menu once more. In the case of this particular example design, there are two routing layers on Layer 1 and layer 4, which means that the script will modify the **View** pull-down menu to appear as shown in Figure 13-2.

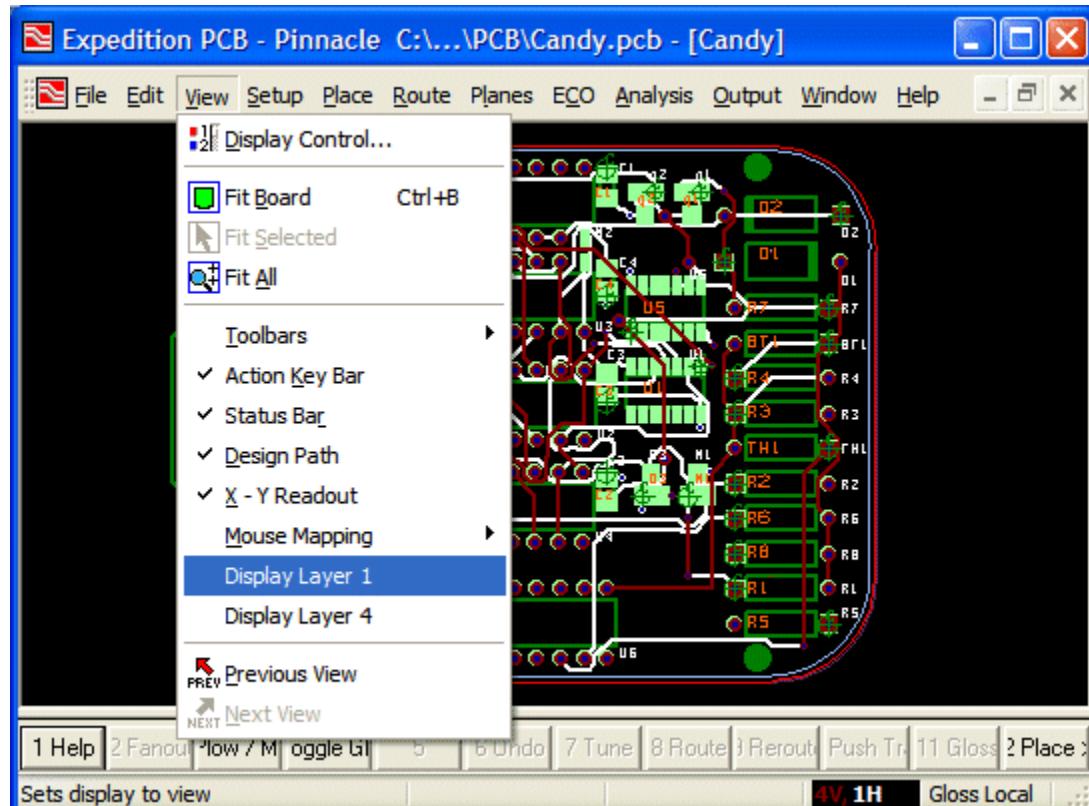


Figure 13-2. Buttons/items under the **View** menu before the script is run.

If you were to execute the **View > Display Layer 1** command, any elements on Layer 1 would be displayed and any elements on Layer 4 would be disabled. Similarly, if you were to execute the **View > Display Layer 4** command, any elements on Layer 4 would be displayed and any elements on Layer 1 would be disabled.

Also, the script will establish 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 will have the same effect as using their corresponding menu commands.

The Script Itself

The key sections forming this script are as follows:

- **Constant Definitions** Define the constants to be used by the script.
- **Initialization/Setup** Get the *Application* and *Document* objects, license the document, add the type libraries, and call the subroutine that will add the menu entries and accelerators.
- **Main Subroutine** This is the subroutine that actually adds the menu entries and accelerators.

- **Event Handler Generator Function** This is the function that dynamically generates event handlers.
- **Display Control Subroutine** This subroutine is used to manipulate the **View > Display Control** settings.
- **Helper Functions and Subroutines** These menu manipulation "helper routines" are used to aid with dynamic code generation and adding menu entries.
- **Validate Server** The standard function used to license the document and validate the server. (Note that we won't go through this function in this chapter – for more details on this function see *Chapter 3: Running Your First Script*.)

Constant Definitions

Observe Lines 8 through 13. On line 8 we instantiate a variable and on Line 9 we assign a value to it. Similarly, on Line 10 we instantiate a variable and on Lines 11 through 13 we assign a value to it.

As was previously noted, this script will feature dynamic code generation, and the variables declared on Lines 8 and 10 will be used to facilitate this. In fact, these variables will be used as constants, so why did we use the *Dim* keyword as opposed to the *Const* (for "Constant") keyword. Well, the assignment on Lines 11 through 13 involves us building a string, and we can't do this with a *Const*; this explains why we use a *Dim* on line 10.

By comparison, the assignment on Line 9 doesn't require us to build anything, so we could have combined Lines 8 and 9 into a single *Const* declaration. The only reason we split it into a *Dim* on Line 8 and an assignment on Line 9 was 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 my adding a menu
3  ' entry for each route layer and tying and 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 are used to get the *Application* object; Lines 20 and 21 are used to get the *Document* object; on Line 24 we license the document by calling the standard *ValidateServer()* function; and Lines 27 and 28 are used to add the type libraries required by this script.

```

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("MGCPBC.ExpeditionPCBAApplication")
28  Scripting.AddTypeLibrary("MGCSDD.KeyBindings")
29

```

On line 31 we call the subroutine *AddLayerMenus()*. This subroutine – which we are about to create – is the one that adds the new menu buttons/items and establishes the various accelerator keys.

Even after we've added the new menu buttons/items and programmed the accelerator keys, we don't want this script to terminate, because this would prevent the various menu button/item and accelerator event handlers – which are defined in the script – from being called. Thus, on Line 33 we set the *DontExit* property of the *Scripting* object to the Boolean value of *True*, which means that this script will continue running until the current session ends.

```

30  ' Add the menu entries and accelerators.
31  Call AddLayerMenus()
32
33  Scripting.DontExit = True
34

```

Main Subroutine

Lines 35 through 80 are where we declare the subroutine that actually adds the menu entries and accelerators. On Lines 39 and 40 we get 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

```

On Lines 43 and 44 we get 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

```

We also want to add accelerator key bindings, so on Lines 47 and 48 we get the *Bindings* object associated with the document.

```

46      ' Get the key bind object.
47      Dim docKeyBindTableObj
48      Set docKeyBindTableObj = pcbAppObj.Gui.Bindings("Document")
49

```

Now, we want to add a menu button/item for each of the routing layers, so on Lines 51 and 52 we get a collection of the *LayerObjects* that define the layer stackup. Observe the *False* parameter used on Line 52 – this indicates that we desire any insulation layers to be excluded from our collection, which will therefore contain only routing (signal) and plane layers.

```

50      ' Get the layer stack
51      Dim layerObjColl
52      Set layerObjColl = pcbDocObj.LayerStack(False)
53

```

Now we are going to iterate through all of the layers in our layer collection. On Line 56 we instantiate a variable to control the iteration loop and on Line 57 we define the iteration order. Observe that we're iterating in *reverse* order so that we can build our new menu buttons/items from the bottom up; that is, each time we add a new button/item, we will add it directly under the **View > Mouse Mapping** button/item.

```

54      ' Process 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 we get the *LayerObject* object associated with the current iterator.

```

58          Dim layerObjObj
59          Set layerObjObj = layerObjColl.Item(i)
60

```

On Lines 61 and 62 we get the layer number associated with the current *LayerObject* object.

```

61          Dim layerInt
62          layerInt = layerObjObj.ConductorLayer
63

```

On Line 65 we check to see that this 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

```

In the case where this is a routing layer (as per the test above), on Lines 67 and 68 we declare a *CommandBarButton* object and then use the *AddMenuAfter()* helper function to create an associated button/item object at the correct location in the menu.

The *AddMenuAfter()* helper function is presented later in this chapter. The first parameter to this function specifies the existing menu entry after which the new menu entry should be inserted; the second parameter is the *Menu* object to which the new button/item should be added.

```

66          ' Add the menu entry.
67          Dim displayLayerBtnObj
68          Set displayLayerBtnObj = AddMenuAfter("Mouse
                                         Mapping", viewMenuObj)

```

Line 69 defines the name to be used for the menu button/item. This name is built dynamically using the number of the current layer. Line 70 defines the description text to be associated 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. Now, this is where things start to get interesting. We need a button press event handler function for each layer – that is, for each button/item we generate. Due to the fact that the number of layers isn't known until run

time, we have to dynamically create these functions at run time. Thus, on Line 72 we call the `GenerateDisplayHandler()` function and pass it the layer number for which the event handler should be generated. The return value from this function is the name of the event handler, which we assign to the `ExecuteMethod` property.

```

71                               displayLayerBtnObj.Target = ScriptEngine
72                               displayLayerBtnObj.ExecuteMethod =
73                                     GenerateDisplayHandler(layerInt)

```

Also, we want to tie an accelerator key to our newly created menu entry. These accelerator keys will be `<Ctrl>+<Layer Number>`; that is, press the number of the layer while pressing and holding the `<Ctrl>` key. We do this on Lines 75 through 77 by means of the keyin command `xm` ("execute menu").

```

74             ' Add an accelerator.
75             Call docKeyBindTableObj.AddKeyBinding("Ctrl+" &
76                                         layerInt, _
77                                         "xm ""View->Display Layer "& layerInt & """", _
78                                         BindCommand, BindAccelerator)
79             Next
80 End Sub
81

```

Observe the use of single, dual, and quadruple double-quotation characters (" , "", and "") on Line 76. This can take a little effort to wrap one's brain around the first time you see it. The easiest way to visualize what's happening here is by means of an example, such as the one shown in Figure 13-3.

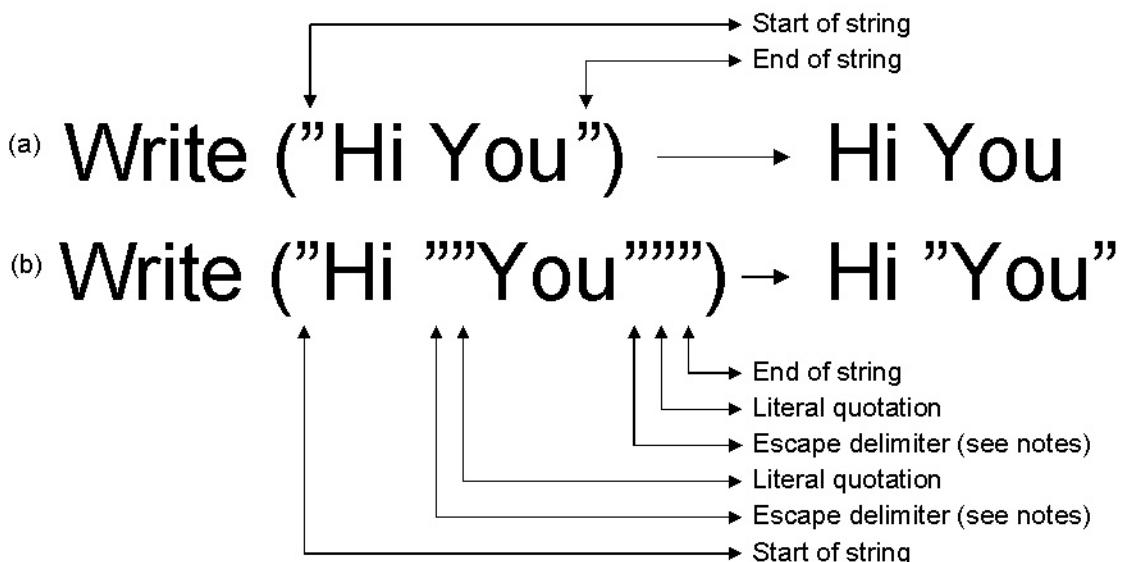


Figure 13-3. Understanding single, double, and quadruple double-quotes.

First, consider the simple example shown in Figure 13-3(a). In this case the first double-quote indicates the start of the string, then we have some text (*Hi You*), and finally we have a second double-quote to terminate the string. In this case, the text generated by this statement will be *Hi You*.

Next, consider the slightly more complex example shown in Figure 13-3(b). Once again, the first double-quote indicates the start of the string. When the system comes to consider the

second double-quote it sees that this character is immediately followed by another double-quote; thus, the second double-quote acts as an escape delimiter, which informs the system that the following character is not intended to close the string, but should instead be treated as a literal character that we want to output. Similarly, the fourth double-quote acts as an escape delimiter informing the system that the fifth double-quote should instead be treated as a literal character that we want to output. Finally, the sixth double-quote terminates the string. In this case, the text generated by this statement will be *Hi "You"*.

All this means that, as we cycle around the loop in the code above, the strings generated by Line 76 will appear as follows:

```
xm "View->Display Layer 1"  
xm "View->Display Layer 4"
```

(For more information on the use of multiple double-quotes, see also the discussions that are associated with Figure 18-6 in *Chapter 18*.)

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 executes it in the context of the running script. In the case where this string defines a function, *ExecuteGlobal()* doesn't actually execute the function, it just makes that function available to be called.

Also observe the parameters passed into the *ExecuteGlobal()* function on Line 89. In order to generate this parameter, we use the "constant" we declared on Lines 10 through 13 as a template. We substitute a substring in the template with the layer number to generate the unique handler. We use a similar trick on Line 92 when we generate the function name that will be returned with this handler.

```
82  ' Function for generating the command handlers at runtime.  
83  ' This is required because 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(handlerTemplateNameStr, "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, so – generally speaking – it should be used only for very simple functions. As we see in the above example, we're simply creating a function that will call a more sophisticated helper function, and it's this more sophisticated (but non-dynamically created) helper function that will actually perform the display manipulations.

In the case of the *Candy.pcb* design, the routing layers are 1 and 4. Thus, if we were to explicitly create the handlers by hand, they would appear as follows (the line numbers would

be different). The function presented above generates these handlers dynamically (apart from the 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 will set a specified layer to be the active layer, turn on the display of this active layer, and disable the display of all of the other routing (signal) layers. Observe that on Line 101 we lock the server to improve performance since we're about to perform display manipulations. This will prevent the display from updating until the *UnlockServer* method is called in Line 130.

```
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 we determine the number of layers in the design, then on Line 107 we verify that this is a valid number (there's no reason why it shouldn't be, we're just being careful).

```
102     ' Figure out 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
```

On Lines 109 and 110 we get the *DisplayControl* object.

```
108         ' Get the display control object
109         Dim dispCntrlObj
110         Set dispCntrlObj = pcbDocObj.ActiveView.DisplayControl
111
```

On Line 114 we make the layer of interest the active layer and on Line 117 we make the display of this layer active.

```
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) =
118                                         True
```

On lines 120 through 126 we iterate through a loop turning everything but the active layer off. In fact, it's not possible to disable the display of the active layer, so we don't really require the test on line 123, but we include this test to make it apparent to other programmers what we're 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) =
125                  False
126          End If
127      Next
128  End If
```

Finally, on Line 130 we unlock the server and on Line 132 we terminate the subroutine.

```
129      ' Unlock the server
130      pcbAppObj.UnlockServer
131
132  End Sub
133
```

Helper Functions and Subroutines

This is where we declare some generic "helper" routines. These routines are not focused on this script per se; they can easily be used in other scripts.

AddMenuAfter() Function

Lines 142 through 149 declare the *AddMenuAfter()* function, which can be used to add 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 to be added into the menu. Observe that, on line 145, this function calls the *GetMenuNumber()* helper function, which we will consider in a moment.

```
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 - CommandBarSrv menu object
142  Function AddMenuAfter(afterMenuEntryStr, menuBarObj)
143      Dim entryNumInt
144      entryNumInt = GetMenuNumber(afterMenuEntryStr,
145                                  menuBarObj) + 1
146      If entryNumInt > menuBarObj.Controls.Count Then
147          entryNumInt = -1
148      End If
149      Set AddMenuAfter =
149      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" (sort of like a "gentleman's gentleman"). It performs its task by cycling through all of the buttons/items and comparing their captions with the one it's looking 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
155  Function GetMenuNumber(menuToFindStr, menuBarObj)
156      Dim ctrlColl : Set ctrlColl = menuBarObj.Controls
157      Dim ctrlObj
158
159      Dim menuCntInt : menuCntInt = 1
160      GetMenuNumber = -1
161
162      For Each ctrlObj In ctrlColl
163          Dim captStr: captStr = ctrlObj.Caption
164          captStr = Replace(captStr, "&", "")
165          If captStr = menuToFindStr Then
166              GetMenuNumber = menuCntInt
167              Exit For
168          End If
169          menuCntInt = menuCntInt + 1
170      Next
171
172  End Function
```

Creating and Testing the Script

- 1) Use the scripting editor of your choice to enter the script described above (including the license server function which is not shown above) and save it out as *LayerDisplay.vbs*.
- 2) Close any instantiations of Expedition PCB that are currently running, and then launch a new instantiation of this application.
- 3) Open the *Candy.pcb* design we've been using throughout these building block examples.
- 4) Run the *LayerDisplay.vbs* script.
- 5) Execute the new **View > Display Layer 1** menu button command (or use the new <Ctrl>+1 accelerator) and observe the results as illustrated in Figure 13-4.

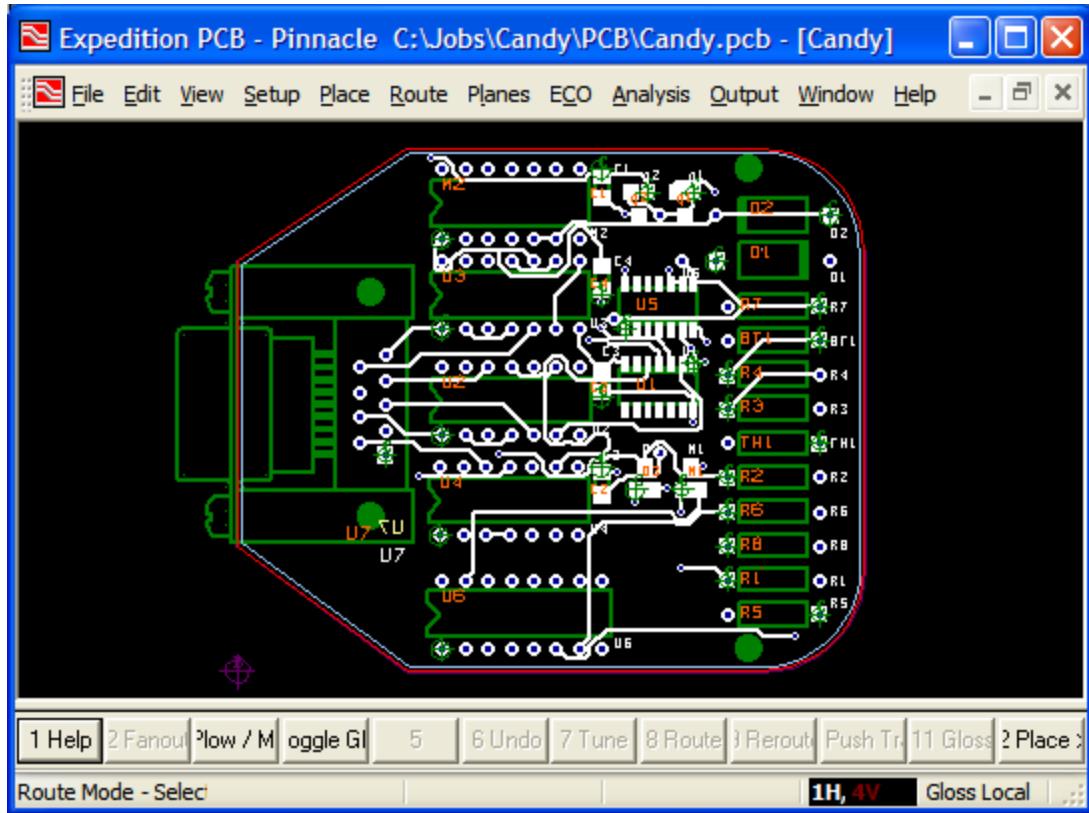


Figure 13-4. After executing the **View > Display Layer 1** command.

- 6) Observe that only white signals from routing layer 1 are displayed, while any red signals from routing layer 4 have been disabled.
- 7) Execute the new **View > Display Layer 4** menu button command (or use the new **<Ctrl>+4** accelerator) and observe the results as illustrated in Figure 13-5.
- 8) Observe that only red signals from routing layer 4 are displayed, while any white signals from routing layer 1 have been disabled.

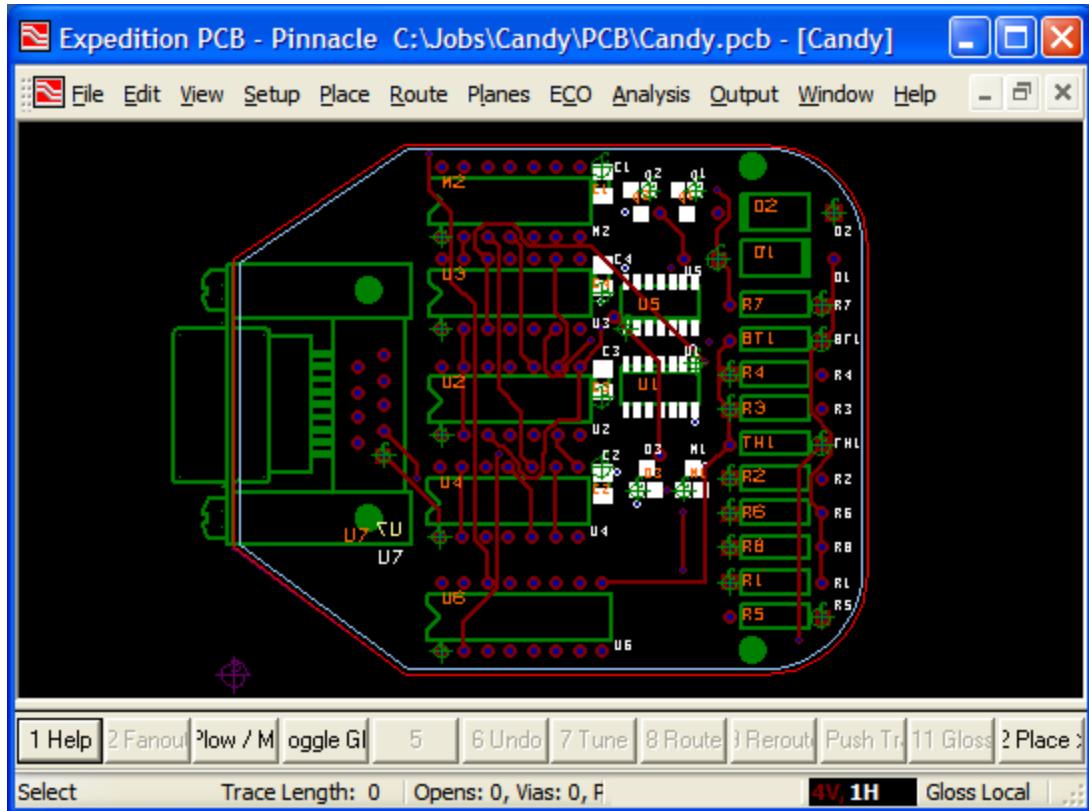


Figure 13-5. After executing the View > Display Layer 4 command.

- 9) Close this instantiation of Expedition PCB.
- 10) Launch a new instantiation of this application.
- 11) Open one of your own designs that contains more than two routing layers.
- 12) Run the *LayerDisplay.vbs* script.
- 13) Check that all of the new menu buttons/items and accelerator keys are created and function as expected.

Enhancing the Script

There are a number of ways in which this script could be enhanced. Some suggestions are as follows (it would be a good idea for you to practice your scripting skills by implementing these examples):

- Set this script to run as a startup script (see *Chapter 10: Running Startup Scripts*).
- Modify the script to turn display of pads on or off for each layer.
- Modify the script such that whenever a new document is opened the script automatically removes any menu entries (and accelerators) associated with the previous design and then adds appropriate menu entries (and accelerators) for the newly opened design.

Chapter 14: Documenting the Layer Stackup

Introduction

Overview:	This script documents the layer stackup by creating text and graphics on a user layer, which represents the layer stackup.
Approach:	To use motion graphics to present the user with a view of the layer stackup that can be moved around and positioned as required and then placed on a user layer.
Points of Interest:	– Implementing a command using motion graphics
Items Used:	– Command Object – Motion Graphics Methods

Before We Start

The term "motion graphics" refers to special graphical representations of text and objects that can be "attached" to the mouse cursor. When the mouse is moved the motion graphics follow the cursor. Motion graphics can be used for a variety of purposes, including documenting the layer stackup as illustrated in this example script.

When you are working with a layout design document (which may have been created by yourself or by someone else), you can use the **Setup > Setup Parameters** command to examine the layer stackup. However, it can be useful to graphically display this information in the document itself.

Just to give ourselves a feel for how this script is going to work, let's assume that we've already created the script and that we've just opened a layout design document (the *Candy.pcb* design for the purposes of this example) and run the script on it. As soon as we run the script, motion graphics will appear on the screen as illustrated in Figure 14.1.

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 were to move the mouse, these motion graphics would follow the cursor (hence their "motion" moniker).

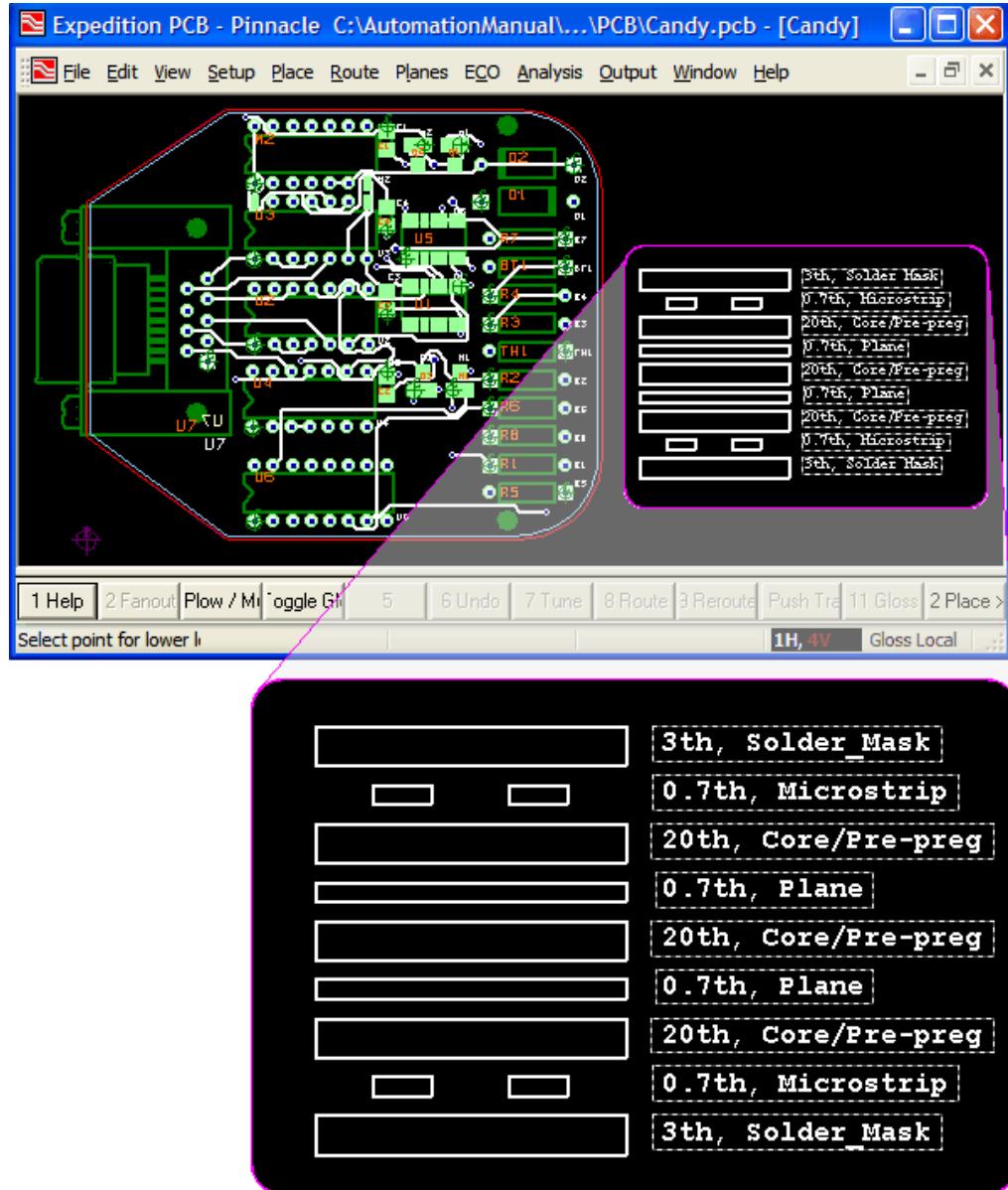


Figure 14-1. Motion graphics generated by our script.

When you have placed the motion graphics in the desired location, clicking the mouse button will cause the script to create "real" versions of these graphics on a user layer as illustrated in Figure 14.2.

Observe that some of the graphical items now show a fill pattern; also that the text items no longer have dotted lines around them; also that we've now added some header information.

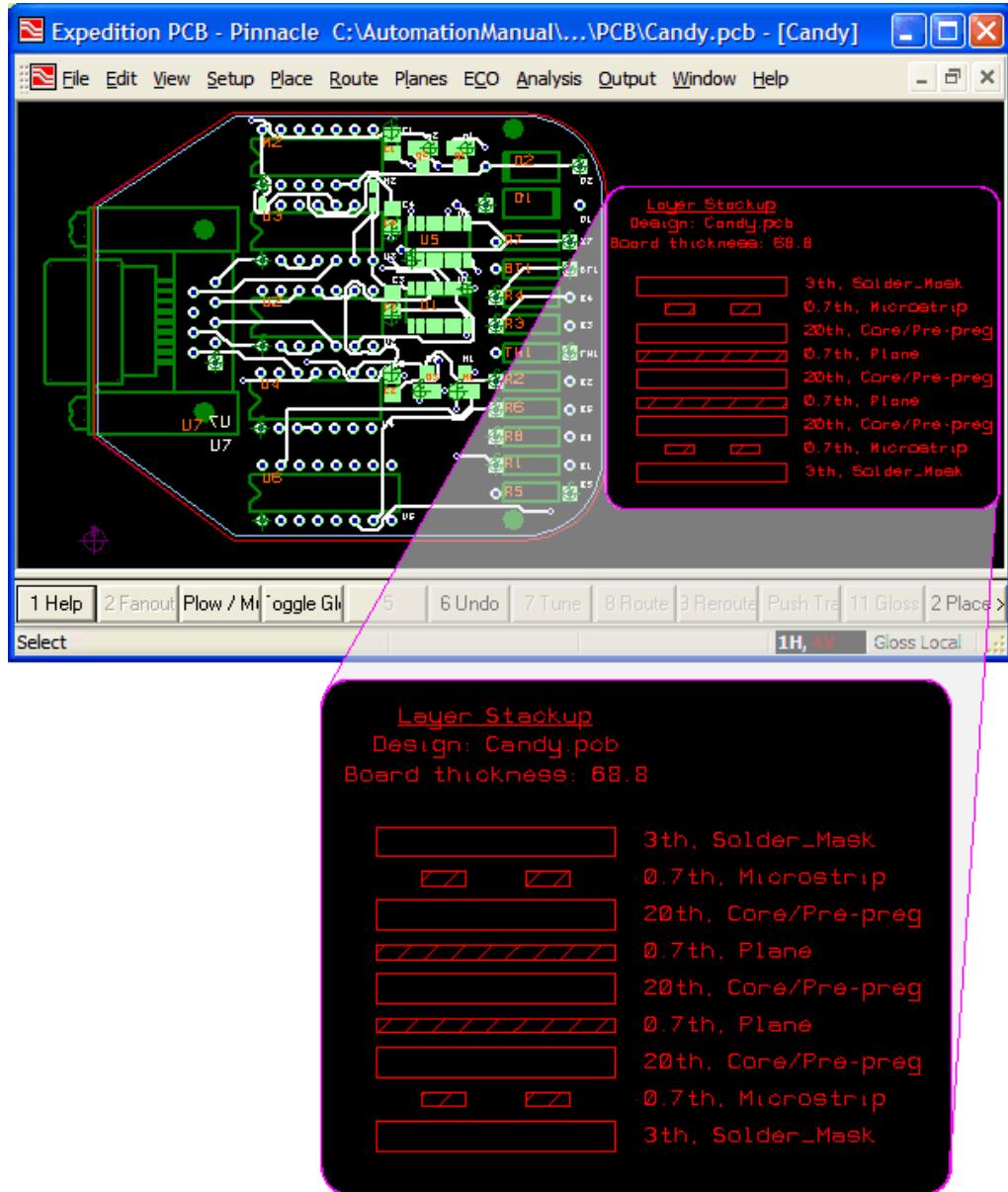


Figure 14-1. "Real" graphics generated by our script and placed on a user layer.

The Script Itself

The key sections forming this script are as follows:

- **Constant Definitions** Define the constants to be used by the script.
- **Initialization/Setup** Get the *Application* and *Document* objects, license the document, and add the type libraries. Also perform any additional setup associated with this script.
- **Event Handlers** These are the event handlers for the *Command* object, which is used to detect and process mouse clicks.

-
- | | |
|--|--|
| <ul style="list-style-type: none"> ▪ Generate Motion Graphics Function ▪ Generate User-Layer Graphics Function ▪ Utility Functions and Subroutines ▪ Helper Functions and Subroutines ▪ Miscellaneous Functions/Routines ▪ Validate Server | <p>This is the function that will generate the motion graphics on the screen.</p> <p>This is the function that will generate the "real" graphics and place them on the user layer.</p> <p>Special "workhorse" routines that are focused on this script.</p> <ul style="list-style-type: none"> – GenerateHeaderOnUserLayer() – GenerateStackupGraphics() – DrawDielectricLayer() – DrawSignalLayer() – DrawPlaneLayer() <p>Generic "helper" routines that can be used in other scripts.</p> <ul style="list-style-type: none"> – ConfigureUserLayer() – GetLayerDescription() – DrawText() – DrawRectangle() <p>Additional routines that don't fit in the "utility" or "helper" classes</p> <ul style="list-style-type: none"> – ClearMotionGraphics() <p>The standard function used to license the document and validate the server. (Note that we won't go through this function in this chapter – for more details on this function see <i>Chapter 3: Running Your First Script</i>.)</p> |
|--|--|

As we shall come to discover, there are lots of fiddly details with regard to generating and manipulating graphics. Thus, it will make things easier if we first consider a high-level (pseudo-code) representation of the actions to be performed by the script as follows:

```

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()

```

As we see, 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 forming the board. In this case these items are created as motion graphics, which means they will automatically follow the mouse cursor until a mouse button is clicked.

The second portion of the script addresses what to do when a mouse button is clicked, which is to clear any existing motion graphics, to configure a user layer, and to regenerate the graphic and text items associated with each dielectric, signal, and plane layers forming the board. This graphics and text generation process is identical to the first portion of the script except that – in this case – these items will be presented on the user layer. We finish by displaying some header information on the user layer.

Constant Definitions

In the first part of the script (Lines 8 through 24) we declare the constants we're going to use throughout the remainder of the script. These constants define the size (width and height) of the various graphical rectangles that are to be 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 in one place at the beginning of the script makes it easy to play with them until the results look the way we require

```

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 TEXTTHEIGHT = 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 we add the type library we want to use:

```

26 ' Add any type libraries to be used.
27 Scripting.AddTypeLibrary("MGCPBC.ExpeditionPCBApplication")
28

```

On Lines 30 through 34 we declare our global variables. In addition to the usual *Application* and *Document* objects, we also declare 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 we declare a "tag" and initialize it with a value of -1. As we shall see, when we create any motion graphics we are going to associate this tag with them; this will allow us to access them and delete them later.

```

33 Dim tagMotionInt : tagMotionInt = -1 ' Tag for identifying
                                         motion graphics.

```

On Line 34 we define the specific units to be associated with this script (apart from anything else, these units will be applied to the constants we declared on Lines 8 through 24).

```

34 Dim unitsEnum : unitsEnum = epcbUnitMils ' Units to be used
                                             throughout

```

On Lines 36 through 43 we 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)

```

On Line 46 we specify the units that are to be associated with the document.

```

45 ' Set the unitsEnum to match our hard coded values.
46 pcbDocObj.CurrentUnit = unitsEnum

```

On Line 49 we use the *RegisterCommand* method to create a *Command* object, which will become the active command at this time. Essentially we are saying: "This is now the active command, so any events should come to this command." Also, the string parameter "Place Layer Stackup" is the text that will appear 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 our command is active; that is, as long as our **Place Layer Stackup** command is the active command, the main 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")
```

On Line 52 we define the units to be associated with the *Command* object. This means that when we get a mouse-click event, the coordinates associated with the mouse cursor will be reported in terms of these units. The reason we have to do this is that the *Command* object doesn't inherit anything from the *Document* object.

```
51  ' Command unitsEnum are separate from the doc so set those too.
52  cmdObj.Unit = unitsEnum
```

On Line 55 we formally attach any events to our *Command* object.

```
54  ' attach the events to the command
55  Call Scripting.AttachEvents(cmdObj, "cmdObj")
```

On Line 58 we call the function that will create the stackup motion graphics and we pass it the current position of the mouse cursor. This is because we require the initial position of the motion graphics to be defined by the current location of the mouse cursor. At this point the motion graphics will appear on the screen, after which they will automatically follow the position of the mouse cursor (until the user clicks a mouse button).

```
57  ' Create the motion graphics
58  Call GenerateStackupMotionGraphics(pcbDocObj.ActiveView.
      MousePositionX, pcbDocObj.ActiveView.mousePositionY)
```

On Line 62 we display a message in the status bar to prompt the 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)
```

On Line 66 we set the *DontExit* property of the *Scripting* object to the Boolean value of *True*. This means this script will continue 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 discussed in detail in the *Interactive Commands* topic of *Chapter 12: Basic Building-Block Examples*.

On Line 79 we call a function to delete any existing motion graphics. On Line 81 we call a function to recreate the graphics on the user layer. On Line 83 we set the return value of this function to *True* to tell the calling script that this function was used. Now, we only care about the first mouse click, so on Line 85 we terminate the top-level command (which is our command) and return control to whatever was active prior to us running this script.

```
68 '
69 ' Event Handlers
70 '
71 ' Mouse click event handler (defined in MGCPCB interface)
72 ' buttonEnum - Enumerate (EPcbMouseButton)
73 ' flagsInt - Integer
74 ' xReal - Real
75 ' yReal - Real
76 Function cmdObj_OnMouseClk(buttonEnum, flagsInt, xReal, yReal)
77     If buttonEnum = epcbMouseButtonLeft Then
78         ' User has selected a location place the graphics.
79         Call ClearMotionGraphics()
80         ' Add the layer stack to a user layer
81         Call GenerateStackupGraphicsOnUserLayer(xReal, yReal)
82         ' Return true to indicate the event was processed.
83         cmdObj_OnMouseClk = True
84         ' We only needed the one click so exit the command now.
85         Call Gui.TerminateCommand()
86     End If
87 End Function
```

On Lines 90 through 95 we "clean things up" with an *OnTerminate* handler that terminates our script. This termination event could occur for a number of reasons, such as the user pressing the <Esc> ("Escape") key, or another command being invoked, or the user clicking a mouse button.

```
89 ' Command termination event handler (defined in MGCPBC I/F)
90 Function cmdObj_OnTerminate()
91     ' The command has been exited clear motion gfx.
92     Call ClearMotionGraphics()
93     ' We are done with the command.
94     Set cmdObj = Nothing
95 End Function
```

Generate Motion Graphics Function

Lines 101 through 120 represent the main function used to generate the motion graphics. On Line 108 we call a function to clear any existing motion graphics (just to be sure). On Line 111 we acquire a new tag that will be associated with the motion graphics we are about to create.

On Line 115 we call a function to generate the stackup graphics; the first parameter to this function defines the user layer to which any graphics should be added. By passing a value of *Nothing* we instruct the function that the graphics should be created as motion graphics.

On Line 119 we use the *MotionGfxDrawAnyUndrawn* method to force the graphics to be drawn without our having to wait for the user to move the mouse cursor.

```
101 ' Motion Graphics Generation Function
102
103 ' Main function for creating the stackup motion graphics
```

```

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.ActiveView.MotionGfxGetTag()
112
113      ' Generate the motion gfx.
114      Dim totalThicknessReal
115      Call GenerateStackupGraphics(Nothing, xReal, yReal,
116                                  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.ActiveView.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 so we'll consider this in a little more detail.

On Lines 131 and 132 we instantiate and configure 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)

```

On Line 135 we start a transaction, which will allow us to undo any changes made by this function with a single **Undo** command.

```

134      ' start the transaction
135      pcbDocObj.TransactionStart

```

On Line 137 we define a variable to record the total thickness of the board and we initialize this value to 0.0. In the not-so-distant future, we are going to use this value as part of our header information. (Observe that we declared a similar parameter on Line 114 in the motion graphics function, but we didn't bother initializing it there because we decided to not bother generating a header as part of the motion graphics.)

```

137      Dim totalThicknessReal : totalThicknessReal = 0.0

```

On Lines 141 and 142 we call a function to generate the stackup graphics on the user layer specified by the first parameter. We also pass in the X and Y values associated with the current

position of the mouse cursor. And we also pass in our 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 returned from this function is the Y-axis value associated with the highest (topmost) graphic. Also, the value of the *totalThicknessReal* variable will be updated to reflect the total thickness of the board; this is because the *totalThicknessReal* parameter to the function is an In-Out parameter (in fact, *all* of the parameters are In-Out parameters).

On Line 145 we increase the Y-axis value to add in a constant value for the header we are about to create. When combined with the X-axis value (which hasn't changed), the new Y-axis value will define the location of the bottom left-hand corner of the header.

```
144      ' step the Y position
145      curYReal = curYReal + STEPTITLEY
```

On Line 148 we call 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)
```

On Line 151 we end the transaction we initiated on Line 135.

```
150      ' end the transaction
151      pcbDocObj.TransactionEnd
```

On Line 154 we clear 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 is where we declare some utility routines. These routines are similar in context to our generic "helper" routines (see the next topic), except that they are specific to this script and may not be used elsewhere without some "tweaking".

GenerateHeaderOnUserLayer() Function

Lines 161 through 203 declare the function that is used to generate 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 we declare a local variable to hold a copy of the Y value (we will be modifying 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 we create 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

```

On lines 173 through 177 we take the string we 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)

```

On Line 180 we modify the Y-axis value.

```

179      ' step the Y position
180      curYReal = curYReal + STEPTITLEY

```

On Line 183 we create a string to describe document name.

```

182      ' now we need to add the header text
183      textStr = "Design: " & pcbDocObj.Name

```

On lines 185 through 190 we take the string we just created and put 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)

```

On Line 193 we modify the Y-axis value.

```

192      ' step the Y position
193      curYReal = curYReal + STEPTITLEY

```

On Line 196 we create a string to describe a title for all of the information we're presenting to the user.

```

195      ' now we need to add the header text
196      textStr = "Layer Stackup"

```

On lines 198 through 203 we take the string we 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,
203             epcbJustifyBottom, Nothing, unitsEnum)
203 End Function

```

GenerateStackupGraphics() Function

Lines 205 through 258 declare the function that constructs the stackup graphics. This same function is used to create both the motion graphics and the graphics that are placed on the user layer.

The first parameter to this function defines the user layer on which the graphics are to be placed. If this parameter has been assigned a value of Nothing, then motion graphics will be created. Meanwhile, the *xReal* and *yReal* parameters define the location of the bottom left-hand corner of the graphics. The last parameter will be used to hold the total thickness of the board (this value will eventually be used as part of the header information).

On Line 214 we declare a local variable to hold a copy of the Y value (we will be modifying the value of 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,
214                                         totalThicknessReal)
214      Dim curYReal: curYReal = yReal

```

On Line 216 we declare one variable to hold a *LayerObject* object and another to hold a *LayerObject* collection. On Line 218 we get the *LayerStack*, which is a collection of *LayerObject* objects (the *True* parameter is used to inform the system that we want to include insulating/dielectric layers).

```

216      Dim lyrObjColl, lyrObjObj
217      ' get the layer objects collection
218      Set lyrObjColl = pcbDocObj.LayerStack(True)

```

On line 222 we declare an integer variable. On Line 223 we use this variable to iterate through our collection of layers in reverse order. The reason for iterating in reverse order is that we are building our stackup graphics from "bottom" to "top". (Ideally we would prefer to use a *For Each ... In ... Next* loop rather than a *For ... To ... Next* loop, but we cannot use the former to iterate through a collection in reverse order.)

```

220      ' since we are starting at the bottom and working our way
221      ' up, loop through the collection backward in a for loop
222      Dim i
223      For i = lyrObjColl.Count To 1 Step -1

```

On Line 226 we obtain the next layer object from our collection.

```

225          ' get the layer object from the collection
226          Set lyrObjObj = lyrObjColl.Item(i)

```

On Lines 229 and 230 we update the variable we're using 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, we should note that a board may comprise a variety of different layers: *dielectric*, *conductor* (of which there are two types – *signal* and *plane*), *user*, and *fabrication* (of which there are lots of types).

On Line 233 we check to see if this is a dielectric (insulating) layer. If so, on Line 235 we call a function to draw graphic and text elements to represent the layer, then on Line 238 we modify the local Y-axis value to reflect the addition of these elements in our stackup. Observe that, thus far, no checks have been performed to determine whether the graphics are to be created on a user layer or as motion graphics; as we shall see, these checks will be performed 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,
236                                      xReal, curYReal, unitsEnum)
237          ' step the Y position
238          curYReal = curYReal + STEPCONDY
```

If the layer was not a dielectric layer, then on Line 241 we check to see if it's a conducting layer and – if it is – then on Line 242 we check to see if it's a signal layer. If this is a signal layer, on Line 244 we call 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 on Line 248 we call a function to draw graphic and text elements to represent *this* layer. On Line 253 we modify the local Y-axis value to reflect the addition of these elements in our stackup.

```
240      ' if a conductive layer then draw its graphic and text
241      ElseIf lyrObjObj.Type = epcbLayerTypeConductor Then
242          If lyrObjObj.ConductorType =
243              epcbConductorTypeSignal Then
244                  ' Draw the signal layer.
245                  Call DrawSignalLayer(lyrObjObj, usrLyrObj,
246                                      xReal, curYReal, unitsEnum)
247
248          ElseIf lyrObjObj.ConductorType =
249              epcbConductorTypePlane Then
250              ' Draw the plane layer.
251              Call DrawPlaneLayer(lyrObjObj, usrLyrObj,
252                                  xReal, curYReal, unitsEnum)
253
254      End If
255
256      ' step the Y position
257      curYReal = curYReal + STEPDIEY
258  End If
259 Next
```

In reality, the test on Line 246 to see if we are dealing with a plane layer is redundant and is included in this script only for the purposes of clarity. Finally, on Line 257, we return 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 (we'll 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 is assigned a value of *Nothing*, the result will be to create 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 be used.

On Line 269 we call our *DrawRectangle()* helper function to draw a rectangle associated with this layer. Observe the constants used to define the rectangle's width and height (we declared these constants at the beginning of the script). Also observe the *False* parameter, which is used to inform the helper function that we do not want to fill this rectangle.

On Line 271 we call our *DrawText()* helper function to draw the text associated with this layer. In particular, observe that we use our *GetLayerDescription()* helper function to retrieve the layer description from our *LayerObject* object, and then we pass 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,  
273  '                                     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 very similar to the *DrawDielectricLayer()* function presented above. The main differences are (a) we draw two small rectangles side-by-side [see Figure 14.2 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,
286                               TRACEBOXY, True, unitsEnum)
286   Call DrawRectangle(userLayerObj, originXReal + _
287                      TRACEBOX2OFFSETX, originYReal, TRACEBOXX,
288                               TRACEBOXY, True, unitsEnum)
288   Call DrawText(userLayerObj, _
289                 GetLayerDescription(layerObjObj), originXReal +
290                 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 very similar to the *DrawDielectricLayer()* function presented above. The main difference is that the rectangle has a fill pattern as defined by the *True* parameter on Lines 302. Once again, observe that no checks have been performed thus far to determine whether the graphics are to be created on a user layer or as motion graphics; as we shall see, these checks will be performed at a lower level.

```

292 ' Function for drawing plane layers. Works for user layer
293 ' grpahics 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 +
305                 TEXTOFFSETX, originYReal + TEXTOFFSETYCOND, unitsEnum)
305 End Function

```

Helper Functions and Subroutines

This is where we declare some generic "helper" routines. These routines are not focused on this script per se; they can easily be used in other scripts.

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 *then* returns it. Also, the function makes sure that this user layer is being displayed.

Observe the parameter being passed into the function on Line 314. This is a string that defines the name of the user layer in which we are interested (this string is defined by the constant we declared way back in the mists of time 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)

```

On Line 316 we attempt to find a user layer by this name.

```

315      ' See if our user layer already exists.
316      Dim usrLyrObj
317      Set usrLyrObj = pcbDocObj.FindUserLayer(userLayerNameStr)

```

If no such layer exists (as determined by the test on Line 319), then we create it on line 321. Alternatively, if this layer does exist, then on Lines 324 and 325 we 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

```

On Line 329, we ensure that the user layer is turned on in *DisplayControl*.

```

328      ' ensure the user layer is turned on
329      pcbDocObj.ActiveView.DisplayControl.
                    UserLayer(userLayerNameStr) = True

```

On Line 333 we access the global display control object. On Line 334 we access 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.ActiveView.DisplayControl.Global
334      Set colorPatternObj =
                globalDisplayControl.UserLayerColor(userLayerNameStr)

```

On Line 337 we set the color pattern object index to 13, which equates to the diagonal stripes used to fill our signal and plane layers (see Figure 14-2 for an example). On Line 341 we assign our new color pattern object back to the user layer.

```

336      ' set it to the pattern we want
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, we return the modified *UserLayer* object and exit 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 is used to access the layer's description. The input parameter to this function is a *LayerObject* object. On Lines 350 and 351 we build (and return) 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 =
351          layerObjObj.LayerProperties.Thickness & "th, " & _
352              layerObjObj.LayerProperties.Description
353  End Function
```

DrawText() Function

Lines 354 through 375 declare a function that creates text either as motion graphics or on a specified user layer. The first parameter to the function is used to specify the user layer; if this parameter has been assigned a value of *Nothing*, the function will display the text as motion graphics. The second parameter is the text string to be displayed. 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 be used.

```
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,
360                      originYReal, unitsEnum)
```

On Line 361 we test to see if the user layer has been assigned a value of *Nothing*. If this is the case, then on Lines 363 through 367 we use the *MotionGfxPutText* method to generate this text as a motion graphic. Observe that this is the point – in this lowest-level routine (and also in the *DrawRectangle()* function as discussed in the following topic) – where we check to see whether the text is to be placed on a user layer or as motion graphics. By deferring our decision to this point, we simplify the higher level functions and allow them to have the same form 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.ActiveView.MotionGfxPutText(textStr, _
363              originXReal, originYReal, epcbMotionGfxPointMoveXY, _
364              TEXTHEIGHT, 0, 5, "VeriBest Gerber 0", _
365              epcbTextAttrIsBold, epcbJustifyLeft, _
366              epcbJustifyVCenter, tagMotionInt, Nothing, _
367              True, unitsEnum, epcbAngleUnitDegrees)
```

In this case, the first parameter is the text string to be displayed. 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 will actually move (a value of *epcbMotionGfxPointMoveXY* will cause the graphics to follow the cursor in both the X and Y planes; by comparison, values of *epcbMotionGfxPointMoveX* or *epcbMotionGfxPointMoveY* will 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 we declared on Line 33 and assigned on Line 111 (this is the identifier we will use to delete this motion graphic later). The thirteenth parameter defines the color of the text (we pass a value of *Nothing*, which will default to white). The fourteenth parameter is a Boolean that can be set to *True* or *False* to specify whether or not we want to display a boundary box, respectively. 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.

Alternatively, Lines 368 through 375 are used to place the text on the specified user layer. In this case (and based on the discussions above), the parameters are self-explanatory.

```
368     Else
369         Call pcbDocObj.PutUserLayerText(textStr, _
370             originXReal, originYReal, userLayerObj, _
371             TEXTTHEIGHT, 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 on a specified user layer. The first parameter to the function is used to specify the user layer; if this parameter has been assigned a value of *Nothing*, the function will display 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 is used to specify if the rectangle is to be filled or not (this will only be used if the rectangle is to be drawn on a user layer – rectangles in motion graphics do not support fills). The seventh parameter specifies the units to be used.

```
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 we test to see if the user layer has been assigned a value of *Nothing*. If this is the case, then on Lines 385 through 387 we use the *MotionGfxPutRect* method to generate this rectangle as a motion graphic. Once again, observe that this is the point – in this lowest-level routine (and also in the *DrawText()* function as discussed in the preceding topic) – where we check to see whether the rectangle is to be placed on a user layer or as motion graphics. By deferring our decision to this point, we simplify the higher level functions and allow them to have the same form 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.ActiveView.MotionGfxPutRect(originXReal,
386          originYReal, epcbMotionGfxPointMoveXY, _
387          originXReal + width, originYReal + height,
```

387

```
epcbMotionGfxPointMoveXY, _  
tagMotionInt, 1, Nothing, unitsEnum)
```

In this case, 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 this lower left-hand corner will actually move (a value of *epcbMotionGfxPointMoveXY* will cause it to follow the cursor in both the X and Y planes; by comparison, values of *epcbMotionGfxPointMoveX* or *epcbMotionGfxPointMoveY* will cause 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 *this* corner will move.

The seventh parameter is the "tag" identifier we declared on Line 33 and assigned on Line 111 (this is the identifier we will use to delete this motion graphic later). The eighth parameter defines the width of the line. The ninth parameter defines the color of the line (we pass a value of *Nothing*, which will default to white). The tenth parameter specifies the linear units.

Alternatively, Lines 388 through 398 are used to place the text on the specified user layer. In this case we first use Lines 391 and 392 to build a points array, and then we use the *PutUserLayerGfx* method on Lines 395 and 396 to actually draw the rectangle on the user layer. (For more details on this, refer to the *Put User Layer Graphics* topic in *Chapter 12: Basic Building-Block Examples*.)

```
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,  
397             unitsEnum)  
397     End If  
398 End Function
```

Miscellaneous Functions and Subroutines

This is where we declare any miscellaneous routines (the reason we have these as a special category is that they cannot easily be classed as "Utility" or "Helper" routines).

ClearMotionGraphics() Function

Lines 401 through 408 declare a function that will remove any motion graphics. In this case, the *MotionGfxDeleteByTag* method is used in conjunction with the "tag" identifier we declared on Line 33 and assigned on Line 111 (observe that we reset this tag on line 407 after the motion graphics have been deleted).

```
401 ' Miscellaneous Functions  
402  
403 ' Deletes all motion graphics.  
404 Function ClearMotionGraphics()  
405     ' Remove motion graphics and reset the tag.  
406     pcbDocObj.ActiveView.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 above (including the license server function which is not shown above) and save it out as *LayerStackup.vbs*.
- 2) Close any instantiations of Expedition PCB that are currently running, and then launch a new instantiation of this application.
- 3) Open the *Candy.pcb* design we've been using throughout these building block examples.
- 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 motion graphics are replaced with "real" graphics (including a header) on the user layer.

Enhancing the Script

There are a number of ways in which this script could be enhanced. Some suggestions are as follows (it would be a good idea for you to practice your scripting skills by implementing these examples):

- Set this script to run as a startup script (see *Chapter 10: Running Startup Scripts*).
- 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.

Chapter 15: Performing Assembly DRC

Introduction

- Overview:** This script performs online assembly DRC checking whenever a component is placed to ensure that there isn't 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 Add-in

Before We Start

The main script described in this chapter is going to perform online DRC checking whenever a component is placed to ensure that there isn't a violation. In order to facilitate this, we've already created a "Gripper Graphics" user layer as part of the *Candy.pcb* design, so the first thing you need to do is to turn the display of this layer on. Observe, for example, the red gripper graphics shown on either side of components **M2** and **U3** as illustrated in Figure 15-1.

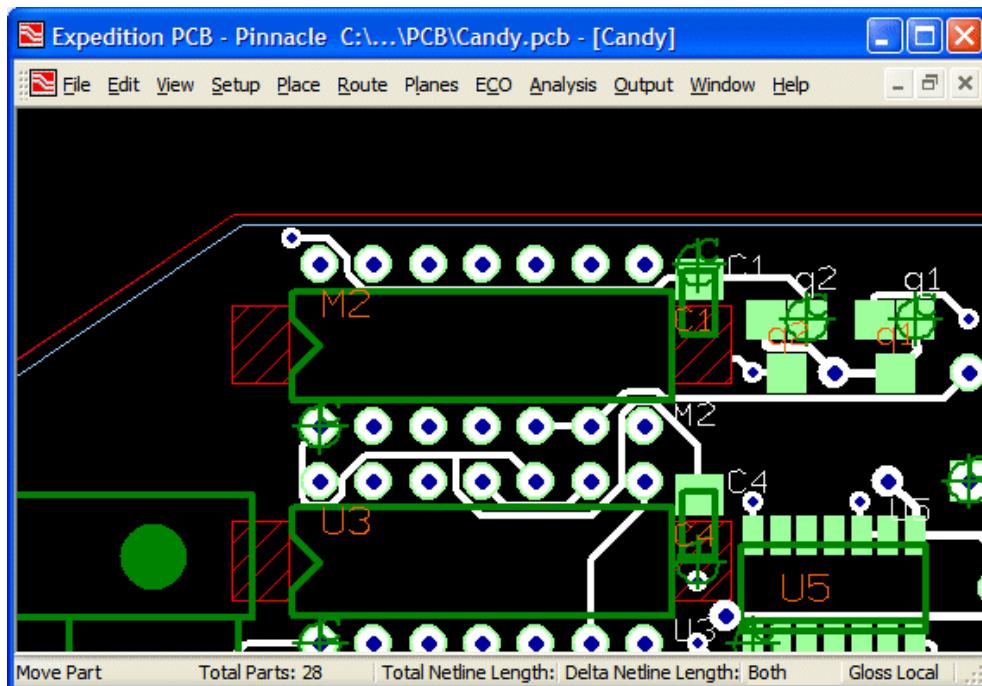


Figure 15-1. The Candy.pcb design with the "Gripper Graphics" being displayed.

In the real world, some mechanism would be used to establish the component placement order; the actual mechanism used will depend on each company's internal process. For example, the placement order may be defined in a side-file and a special script may be used to access the contents of that side-file and assign the placement order to the components on the board.

For the purposes of this test case (and for the sake of simplicity) we are going to create a special script called *AssignPlacementOrder.vbs* that will assign a numeric (integer) placement order based on an alpha-numeric sort of component names. Also, we will be creating a script called *AssemblyDRC.vbs*, which is the script that will actually perform the DRC.

Just to give ourselves a feel for how things are going to work, let's assume that we've already created the two scripts described above; that we've just opened our *Candy.pcb* layout design document, and that we've enabled the display of the "Gripper Graphics" user layer as discussed above.

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 us about a DRC violation as illustrated in Figure 15-2.

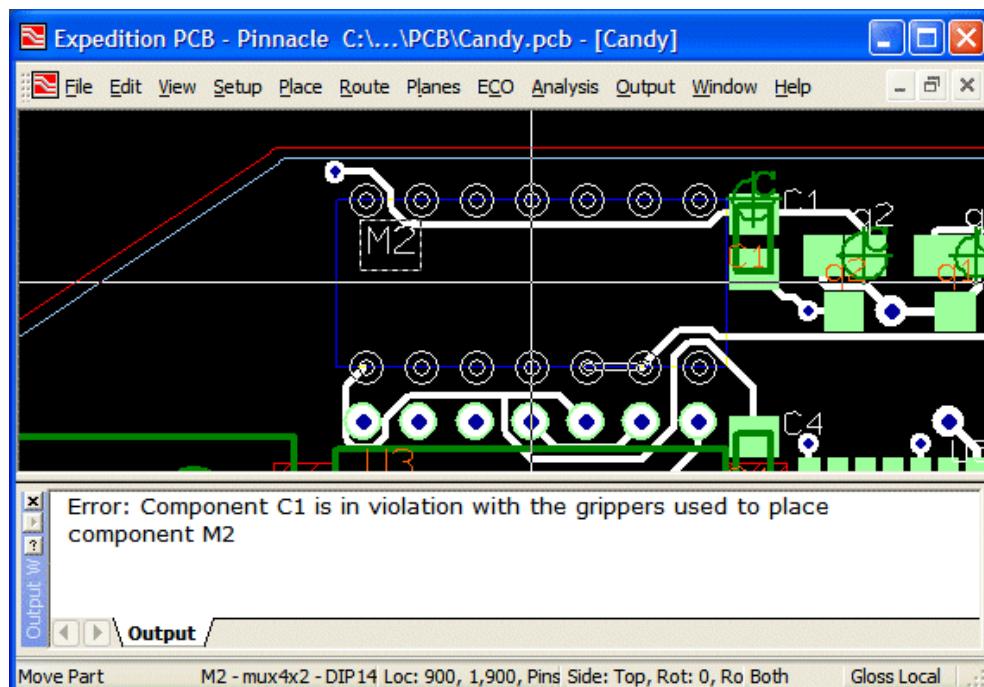


Figure 15-2. The script has detected a DRC violation.

Also observe that when the script detects a DRC violation, it won't place the selected component, but will instead keep this component as motion graphics attached to the cursor (see the discussions in *Chapter 14* 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 15-3. Also observe, however, that the original DRC error message remains on display in the **Output** window. The reason for this is simple – our script did not clear this message from the window (see also the *Enhancing the Script* discussions at the end of this chapter).

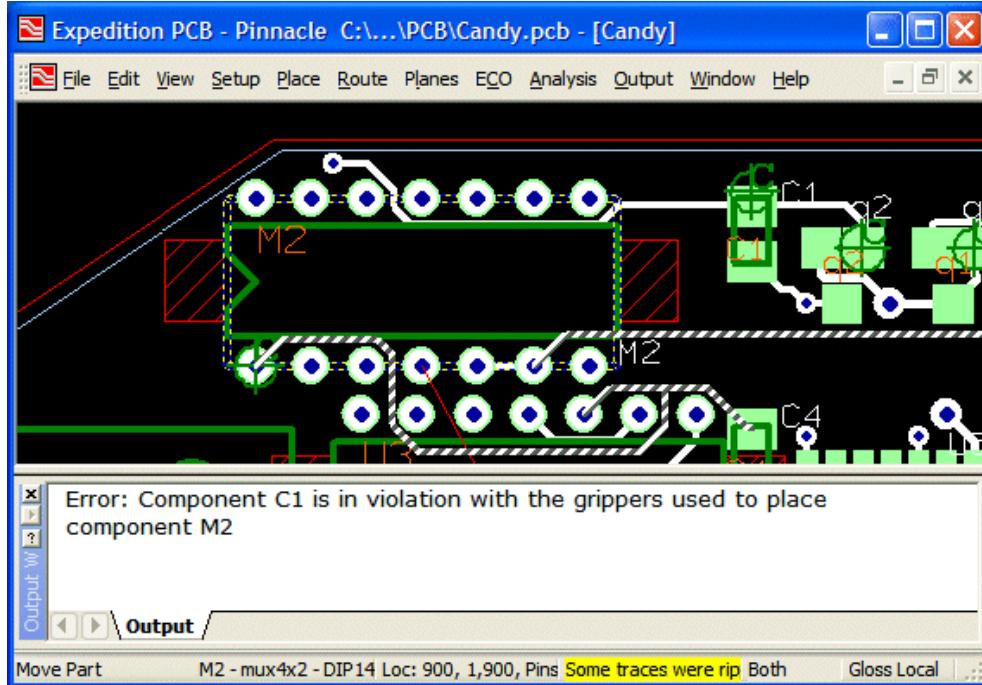


Figure 15-3. The component is successfully placed.

The Assign Placement Order Script

This is the simple script that will be used to assign a placement order to the components on the board (this script must be run before the main script). The key sections forming this script are as follows:

- **Constant Definitions** Define the constants to be used by the script.
- **Initialization/Setup** Get the *Application* and *Document* objects, license the document, and add the type libraries. Also perform any additional setup associated with this script.
- **Assign Placement Order Function** This simple function assign a numeric (integer) placement order based on an alpha-numeric sort of component names.
- **Validate Server** The standard function used to license the document and validate the server. (Note that we won't go through this function in this chapter – for more details on this function see *Chapter 3: Running Your First Script*.)

Constant Definitions

The only constant declaration used by this simple script 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 we add the type library required by this script. Lines 13 and 17 are used to get the *Application* object, while Lines 14 and 20 are used to get the *Document* object. On Line 23 we license the document by calling the standard *ValidateServer()* function; on Line 26 we call our simple *AssignPlacementOrder()* function.

```
9  ' add any type libraries to be used.
10 Scripting.AddTypeLibrary( "MGCPBCB.ExpeditionpcbAppObjlication" )
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()
```

Assign Placement Order Function

On Lines 32 through 47 we declare our simple *AssignPlacementOrder()* function. On Lines 34 and 35 we obtain a collection of all of the components on the board. On Line 38 we use the *Sort* method to perform an alphanumerical sort on our collection. On Lines 43 to 45 we loop around assigning 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
```

In particular, observe the use of the *PutProperty* method on the *Component* object in Line 44. This allows us to assign a "Property Pair" to the component, where a property pair comprises a name (string) and a value. In this case, the name string is the "PLACEMENT_ORDER" string that we assigned to our *PLACEMENT_ORDER* constant on Line 7; while the value is a second string that is generated by concatenating the iterating integer with a null string (this concatenation is performed so as to convert the integer into its string equivalent).

The Main Script

The key sections forming this script are as follows:

- **Constant Definitions** Define the constants to be used by the script.
- **Initialization/Setup** Get the *Application* and *Document* objects, license the document, and add the type libraries. Also perform any additional setup associated with this script.
- **Event Handler** This is a pre-component-place event handler that will be triggered just before a component is actually placed.
- **Assembly Violation Checking Function** This is the main function that checks to see if there has been an assembly violation.
- **Utility Functions and Subroutines** Special "workhorse" routines that are focused on this script.
 - IsCmp2CmpAssemblyViolation()
 - GetNearbyComponents()
 - GetPlacementNumber()
- **Helper Functions and Subroutines** Generic "helper" routines that can be used in other scripts.
 - Output()
 - LoadAddin()
 - FindAddin()
 - RemoveTrailingNewLine()
- **Validate Server** The standard function used to license the document and validate the server. (Note that we won't go through this function in this chapter – for more details on this function see *Chapter 3: Running Your First Script*.)

A high-level (pseudo-code) representation of the actions to be performed by the script as shown below. Observe in particular the way in which we loop around looking at each component that is "near" to the component being placed – the way in which we check 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

Our first task at the beginning of the script is to declare any constants we want to use. On Line 10 we declare the name of the user layer containing the gripper graphics (we are hard-coding this name/layer for the purposes of this script).

On Line 12 we declare the name of the property pair in which we are interested. From our discussions earlier in this chapter, you will recall that a property pair comprising a name

(string) and a value can be attached to a component (multiple different property pairs can be attached to the same component). The *AssignPlacementOrder()* script we presented in the previous topic assigned a property pair to every component in the design. The name of these property pairs was "PLACEMENT_ORDER", while the value associated with each property pair was another string representing the component's placement order.

On Line 14 we define a hard-coded distance value. This will be used to quickly locate any 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
```

At some point in our script we are going to use the **Output** window add-in. For reasons that are beyond the scope of this document, we cannot pull in the type library containing enumerates associated with this add-in. In order to get around this issue, on Lines 18 through 21 we declare any enumerates we are going to require as constants.

In fact, the **Output** window can be docked/located at the Top, Bottom, Left, or Right of the main window. We are going to dock it at the bottom, so we only really need the constant declaration on Line 21 (the others are shown here only for completeness).

Initialization/Setup

Lines 24 and 25 are used to add the type libraries required by this script. On Lines 28 and 29 we declare variables in which to hold our *Document* and *Application* objects, respectively.

On Line 30 we define the units to be used throughout the script (amongst other things, these units apply to our pick offset constant on Line 14). On Line 33 we get the *Application* object; on Line 36 we get the *Document* object; on Line 24 we license the document by calling the standard *ValidateServer()* function; and on Line 42 we ensure that the units associated with the document match the units used in our script.

```
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 our 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

```

On Line 45 we formally attach any events to our *Document* object. On Line 48 we set the *DontExit* property of the *Scripting* object to the Boolean value of *True*. This means this script will continue 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 we declare our pre-component-placement event handler. This handler, which is defined as part of the Expedition PCB automation interface, is fired (triggered) when the 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 remaining parameters are self-explanatory. Note that the values assigned to these parameters are automatically populated by the automation interface when this event is triggered.

```

53  ' Pre component placement handler (defined in MGCPBCB 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 we declare a variable to be used to hold a string describing any violation. This variable is passed as an In-Out parameter to our *IsAssemblyViolation()* function, which is called on Line 65 (remember that *all* parameters in VBScript are In-Out parameters).

If there is an assembly violation, then on Line 67 we set the return value from this function to be *False*; this prevents the component from being placed and keeps it as motion graphics. Alternatively, if no violation is detected, we set the return value from this function to be *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 we test to see whether or not our violation string is empty; if it contains something then there was a violation and we pass the violation string to the **Output** window. Note that we may have a violation string *without* actually having an assembly violation; as we shall see, this is because the violation string may contain information such as the fact 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
```

Assembly Violation Checking Function

Lines 88 through 144 are where we declare the main function used to determine if there has been an assembly violation. The first parameter to this function is the name of the component being placed; the second is a string that will end up containing descriptions of any violations that are detected by this function.

```
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 we declare a local variable to hold a string containing a list of all of the violations detected by this function and we initialize it to be an empty string.

```
90      ' String to hold the violation description  
91  Dim violationDescStr : violationDescStr = ""
```

Initially we will assume that no violations are going to be detected, so on Line 94 we set this function's return value to *False* (we will modify this if necessary as detailed below).

```
93      ' Initialize the return value  
94  IsAssemblyViolation = False
```

On Line 97 we declare a variable used to hold the placement order number associated with the component being placed; also, we call our utility function *GetPlacementNumber()* to determine this placement order number, which we assign to our variable.

```
96      ' Get the placement number of cmpObj  
97  Dim cmpPlaceInt : cmpPlaceInt = GetPlacementNumber(cmpObj)
```

On Line 100 we check to ensure that the placement order number is valid. If not (which could occur if this component was never assigned a number), then on Lines 101 and 102 we add an appropriate warning to our local violation message string. Observe the fact that we terminate this warning with a VbCrLf (carriage return and line feed), because this is something we will need to consider later.

```
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
```

Alternatively, if the placement order number is valid, then on Line 105 we declare a component collection and on Line 106 we use our *GetNearbyComponents()* utility function to retrieve a collection of any nearby components.

```
103      Else
104          ' Get the nerby components
105          Dim nearCmpColl
106          Set nearCmpColl = GetNearbyComponents(PICK_OFFSET,
                                         cmpObj)
```

On Line 109 we declare a variable in which we intend to hold an individual near component object, and on Line 110 we start to iterate through our collection of nearby components.

```
108          ' Iterate to find components in violation
109          Dim nearCmpObj
110          For Each nearCmpObj In nearCmpColl
```

On Line 112 we declare a variable used to hold the placement order number associated with the nearby component; also, we call our utility function *GetPlacementNumber()* to determine this placement order number, which we assign to our variable.

```
111          ' Get the placement ordering of nearCmpObj
112          Dim nearCmpPlaceInt : nearCmpPlaceInt =
                           GetPlacementNumber(nearCmpObj)
```

On Line 115 we check to ensure that the placement order number is valid. If not, then on Lines 116 and 117 we add an appropriate warning to our local violation message string. Once again, observe the fact that we terminate this warning with a VbCrLf (carriage return and line feed), because this is something we will need to consider later.

```
114          ' Ensure the placement number is valid
115          If nearCmpPlaceInt = -1 Then
116              violationDescStr = violationDescStr & "Warning:
                           Component " & nearCmpObj.Name &
                           " has no placement order value." & vbCrLf
117
```

On Line 120 we check to see if the placement order number associated with the nearby component is *lower* than the component we are currently evaluating, which means that the nearby component is to be placed *earlier*. If this is the case, then on line 122 we call our *IsCmp2CmpAssemblyViolation()* utility function and we pass it the nearby component object as the first parameter and the current component object as the second parameter.

Alternatively, if the placement order number associated with the nearby component is *higher* than the component we are currently evaluating, the nearby component is to be placed *later*. In this case, on line 131 we call our *IsCmp2CmpAssemblyViolation()* utility function and we pass it the current component object as the first parameter and the nearby component object as the second parameter.

In both cases, if our *IsCmp2CmpAssemblyViolation()* utility function returns a value of *True* (thereby indicating a violation), we add an appropriate warning to our local violation message string. As usual, observe the fact that we terminate these warnings with a VbCrLf (carriage return and line feed) – we will consider this in a moment. Also observe that in both cases we set the return value of this *IsAssemblyViolation()* function to be *True*, which will inform the outside world (the calling script) that an assembly violation was detected.

```
119          ' Determine which is placed first
120          ElseIf nearCmpPlaceInt < cmpPlaceInt Then
```

```

121      ' Guessed correctly. Check for a violation
122      If IsCmp2CmpAssemblyViolation(nearCmpObj,
123          cmpObj) Then
124          ' There is a violation
125          violationDescStr = violationDescStr & _
126          "Error: Component " & nearCmpObj.Name & _
127          "is in violation with the grippers " &
128          "used to place component " &
129          cmpObj.Name & vbCrLf
130          IsAssemblyViolation = True
131      End If
132  Else
133      ' Check with the different order.
134      If IsCmp2CmpAssemblyViolation(cmpObj,
135          nearCmpObj) Then
136          ' There is a violation
137          violationDescStr = violationDescStr & _
138          "Error: Component " & cmpObj.Name & _
139          " is in violation with the grippers used" &
140          " to place component " &
141          nearCmpObj.Name & vbCrLf
142          IsAssemblyViolation = True
143      End If
144  End If
145  Next
146 End If

```

Finally, we take the local violation string we've been building; we pass it as a parameter to a helper function named *RemoveTrailingNewLine()*, which removes any training carriage return and line feed characters; and we assign the result to the main violation string parameter associated with this function.

```

141      ' Set description string for the calling function to use
142      descStr = RemoveTrailingNewLine(violationDescStr)
143  End Function

```

Utility Functions and Subroutines

This is where we declare some utility routines. These routines are similar in context to our generic "helper" routines (see the next topic), except that they are specific to this script and may not be used elsewhere without some "tweaking".

IsCmp2CmpAssemblyViolation() Function

Lines 154 through 200 declare the routine that checks for component-to-component assembly violations. 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(placeCmpObj,
155          unplacedCmpObj)

```

Now things start to get a little tricky, but this is all rather cool stuff. The Mask Engine is a very powerful tool that allows us to specify any number of geometric shapes and to then perform operations on them such as AND, OR (which acts like an ADD), XOR, and SUBTRACT. In

the case of this routine, we are going to use the Mask Engine to determine specific gripper-to-component violations.

As we shall see, the way this works is that we pass a number of points arrays corresponding to different geometric shapes into the Mask Engine, instruct it as to which operation we want it to perform on these shapes, and the Mask Engine will generate a new points array containing the geometric result of this operation.

As a simple example, consider what would happen if we were to use the Mask Engine to perform an OR operation on two overlapping shapes as illustrated in Figure 15-4(a). Similarly, consider the result of performing an AND operation on the same two shapes as illustrated in Figure 15-4(b).

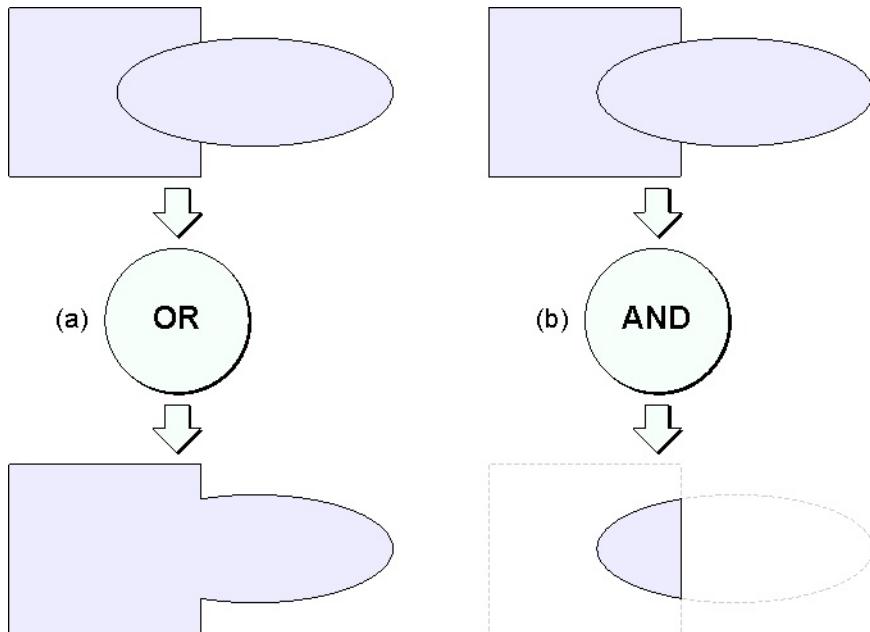


Figure 15.4. The results from using the Mask Engine to perform OR and AND operations on two overlapping geometric shapes.

Keeping this in mind, on Lines 156 and 157 we create the Mask Engine, which is actually a separate server.

```
155      ' Create the mask engine server
156      Dim maskEngineObj
157      Set maskEngineObj =
CreateObject( "MGCPCBEngines.MaskEngine" )
```

On Line 160 we set the units for the Mask Engine.

```
159      ' Set the current unit
160      maskEngineObj.CurrentUnit = unitsEnum
```

On Lines 164 and 165 we acquire the points array associated with the placed component.

```
162      ' Create mask to represent the placement
163      ' outline of placed component
164      Dim ptsArr
165      ptsArr =
```

```
placedCmpObj.PlacementOutlines.Item(1).Geometry.PointsArray
```

On Lines 168 through 170 we create a mask that corresponds to the points array for the placed component. On Line 168 we instantiate a variable to hold a *Mask* object; on Line 169 we create a new/empty *Mask* object; and on Line 170 we define our 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)
```

The next step is to create a mask to represent the gripper. We start by creating an empty *Mask* object on Lines 173 and 174.

```
172      ' Create a mask to represent the gripper
173      Dim maskUnplacedObj
174      Set maskUnplacedObj = maskEngineObj.Masks.Add()
```

On Lines 177 and 178 we 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)
```

On Lines 183 through 186 we iterate through the gripper graphics and – for each such graphic – we add a points array to our 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
```

On Line 190 we AND our two *Mask* objects together by calling the appropriate Boolean operation on the placed *Mask* object and passing in the unplaced *Mask* object.

```
188      ' AND the masks together to determine if there
189      ' is any overlap
190      Call maskPlacedObj.BooleanOp(emBooleanOpAND,
maskUnplacedObj)
```

On Line 193 we check the result, which resides in our placed *Mask* object. If there are 0 (zero) shapes then this means that there were no overlaps, in which case on Line 195 we set the return value from this function to be *False*, thereby indicating that there was no violation. Otherwise, on Line 198 we set the return value to *True* to indicate that we did detect 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 routine that checks to see if there are any components in close proximity to a specified component. This routine 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)

```

On Lines 207 and 208 we determine the extrema of the placement outline for the component of interest. This returns four 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

```

On Lines 212 through 215 we use the *PickComponents* method to locate and return all components in the specified area. This area is defined using the first four parameters to the method by "growing" the extreme values by a specified offset amount (you will recall that the offset was passed into this function as a parameter). The fifth parameter specifies that we are interested in all components; the sixth parameter specifies that we are interested in all cell types; the seventh parameter specifies the layer of interest (a value of 0 equates to "all layers"); and the eighth parameter is set to *True*, which indicates that we are only interested in components that are currently being displayed.

```

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, -
          epcbCelltypeAll, 0, True)
215

```

In addition to any nearby components, our collection also contains the original component itself, but we don't want this in this case, so on Lines 219 to 224 we iterate through the collection looking for the original component and – when we find it – remove it from the collection.

```

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, we return our component collection and exit the function.

```
226      ' Return the component collection
227      Set GetNearbyComponents = cmpColl
228  End Function
```

GetPlacementNumber() Function

Lines 232 through 247 declare a simple routine that returns the placement number associated with a specific *Component* object, which is passed in as a parameter.

```
230  ' Returns the placement number of this component as an integer
231  ' cmpObj - Component object
232  Function GetPlacementNumber(cmpObj)
```

On Line 235 we use the *FindProperty* method to acquire the *PropertyPair* object for the *PLACEMENT_ORDER* property.

```
233      ' Get the property
234      Dim propObj
235      Set propObj = cmpObj.FindProperty(PLACEMENT_ORDER)
```

On Line 239 we check to make sure that we found the property, in which case on Line 240 we get this value; otherwise on Line 242 we set the 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
```

On Line 246 we convert the string value to an integer and return it.

```
245      ' Convert the value to an int and return
246      GetPlacementNumber = CInt(placeNumStr)
247  End Function
```

Helper Functions and Subroutines

This is where we declare some generic "helper" routines. These routines are not focused on this script per se; they can easily be used in other scripts.

Output() Function

Lines 255 through 267 declare a simple routine 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 we call our *LoadAddin()*, helper function, which we can use to load any required add-in components. In this case, we're instructing it to load the Output window and to place it at the bottom of the Expedition PCB interface (if the specified add-in has already been loaded then it is simply returned). The first parameter to this function is the *ProgID* of the required

add-in (the Output window in the case of this script); the second parameter is an arbitrary name that will be associated with the add-in; the third parameter specifies a script to be associated with the add-in (we ignore this in this example); and the fourth parameter specifies the placement location of the add-in.

```
256      ' Get or load the outputwindow add-in
257      Dim outputAddinObj
258      Set outputAddinObj =
259          LoadAddin("MGCSSDOutputLogControl.Addin", _
                      "Output Window", "", PlacementLocationBottom)
```

On Line 263 we acquire the control from the add-in.

```
261      ' Get the output control from the add-in
262      Dim outputCtrlObj
263      Set outputCtrlObj = outputAddinObj.Control
```

On Line 266 we call the *AddTab* method to which we append the text string we want to display (note that this method will either add a new tab or return an existing 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 routine that can be used to add any required components into Expedition PCB. The first parameter to this function is the *ProgID* of the required add-in; the second parameter is an arbitrary name that will be associated with the add-in; the third parameter specifies a script to be associated with the add-in; and the fourth parameter specifies the placement location of the add-in.

```
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
273  '                   in later identification)
274  ' scriptStr - String (script to associate with add-in.
275  '                   Unused in this script)
276  ' placementEnum - String (docking location of add-in)
277  Function LoadAddin(progIdStr, localNameStr, scriptStr,
                           placementEnum)
```

On Line 279 we call our *FindAddin()* helper function, which will return the add-in if it's already been loaded; otherwise it will return *Nothing*.

```
277      ' Check to see if addin is already loaded.
278      Dim addinObj
279      Set addinObj = FindAddin(localNameStr)
```

On Line 282 we check to see if the add-in has already been loaded. If not, then on Lines 284 and 285 we call the *Add* method on the *Addins* collection to load the add-in. The first parameter to this method is the *ProgID* of the required add-in; the second parameter is an arbitrary name that will be associated with the add-in; the third parameter specifies a script to be associated with the add-in; and the fourth parameter specifies the placement location of the add-in.

```
281      ' Load it if not already loaded.
```

```

282     If addinObj Is Nothing Then
283         ' Use the add method on the Addins collection
284         Set addinObj = pcbAppObj.Addins.Add(progIdStr, _
285             localNameStr, scriptStr, placementEnum, "")
286     End If

```

Finally, on Lines 289 through 295 we make sure that our add-in is visible, set the add-in as this function's return value, and exit the function.

```

280
281
282     ' Ensure the addin is visible
283     If Not addinObj Is Nothing Then
284         addinObj.Visible = True
285     End If
286
287
288     ' Return the addin
289     Set LoadAddin = addinObj
290
291     End Function
292
293
294
295

```

FindAddin() Function

Lines 300 through 313 declare a simple routine that checks to see if an add-in has already been loaded and – if so – it returns that add-in; otherwise it returns *Nothing*. It accepts a single string parameter that is used to identify the add-in.

```

297     ' Returns the add-in if already loaded
298     ' Returns nothing otherwise
299     ' localNameStr - String (Name for identifying add-in)
300     Function FindAddin(localNameStr)

```

On Line 302 we initialize the return value with a default value of *Nothing*.

```

301         ' Initialize the return value
302         Set FindAddin = Nothing

```

On Lines 306 through 312 we iterate through all of the add-ins comparing each add-in's name to our identifying string. If we find the add-in, we set the return value for this function to that add-in and exit the loop.

```

303
304     ' Iterate through the add-ins.
305     Dim addinObj
306     For Each addinObj In pcbAppObj.Addins
307         ' If we find the add-in return it.
308         If addinObj.Name = localNameStr Then
309             Set FindAddin = addinObj
310             Exit For
311         End If
312     Next
313     End Function

```

RemoveTrailingNewLine() Function

This helper routine accepts a text string and looks to see if it has a trailing new-line character; if so it removes this character and returns the string; otherwise it simply returns the string.

```

315     ' Removes the trailing vbCrLf if it exists
316     ' textStr - String

```

```

317 Function RemoveTrailingNewLine(textStr)
318     If Right(textStr, 2) = vbCrLf Then
319         textStr = Left(textStr, Len(textStr) - 2)
320     End If
321     RemoveTrailingNewLine = textStr
322 End Function

```

Creating and Testing the Script

- 1) Use the scripting editor of your choice to enter the scripts described above (including the license server function which is not described in this chapter) and save them out as *AssignPlacementOrder.vbs* and *AssemblyDRC.vbs*.
- 2) Close any instantiations of Expedition PCB that are currently running, and then launch a new instantiation of this application.
- 3) Open the *Candy.pcb* design we've been using throughout these building block examples and turn on the display of the "Gripper Graphics" user layer.
- 4) Run the *AssignPlacementOrder.vbs* script.
- 5) Run the *AssemblyDRC.vbs* script.
- 6) Now, 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 15-2.
- 7) Next, 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 15-3.

Enhancing the Script

There are a number of ways in which this script could be enhanced. Some suggestions are as follows (it would be a good idea for you to practice your scripting skills by implementing these examples):

- As described in this chapter, our main script only checks for a DRC violation when you attempt to place a component. One very useful addition to the script would be to cause it to automatically check all of the existing placed components for violations when it is first run.
- This script does not distinguish between component placement outlines and the sides of the board, so it may report non-existent errors. The script could be enhanced so as to handle this sort of thing.
- In the *Before We Start* discussions at the beginning of this chapter (refer back to Figure 15-3), we noted that once the script has detected an error, that message remains in the **Output** window even after the offending component has been successfully placed. Thus, another very useful addition to the script would be to remove any offending error message once the cause of that message has been addressed and resolved. (Use the Help system and an Object Browser to obtain the information you require in order to achieve this task – see also *Appendix D: General VBScript References and Tools* for more information on using Object Browsers).

Chapter 16: Removing Unconnected Pads from Pins and Vias

Introduction

Overview:	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 to let the user select which pins and vias they would like 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:	<ul style="list-style-type: none">– Using the IDE to create a form– Modifying padstacks– Working with connectivity
Items Used:	<ul style="list-style-type: none">– Connected Object Property– Dictionary Object– Padstack Object– Objects Collection

Before We Start

The functionality provided by this script is very similar to the functionality provided by the Padstack Processor build into Expedition PCB (you can access this processor by means of the **Edit > Modify > Padstack Processor** command.) However, when it comes to deleting pads, the Padstack Processor has certain limitations, such as not being able to delete top and bottom pads. The script we are about to create 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 we start, there are a few points regarding padstacks with which we should be familiar. We start with a *Padstack* reference, which defines the base configuration of a padstack. *Padstack* objects – which may be associated with pins and vias – are instantiations of *Padstack* references.

When we are creating a new board design, we commence with a set of original *Padstack* references that are pulled from a library. Note that we cannot make any changes to these originals, even if they are not being used. However, we can use the automation interface to create clones of these originals and we can make changes to these clones. Also note that we cannot make changes to a clone once it's being used; instead we have to take another clone of the original (or a clone of a clone) and then modify this new clone.

Just to get a feel for the way in which our script is going to work, let's assume that we've launched Expedition PCB, opened our *Candy.pcb* layout design document, and selected pin 11 on component **U3**. If we now access the **Padstack Properties** dialog as illustrated in Figure 16-1, we see 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.

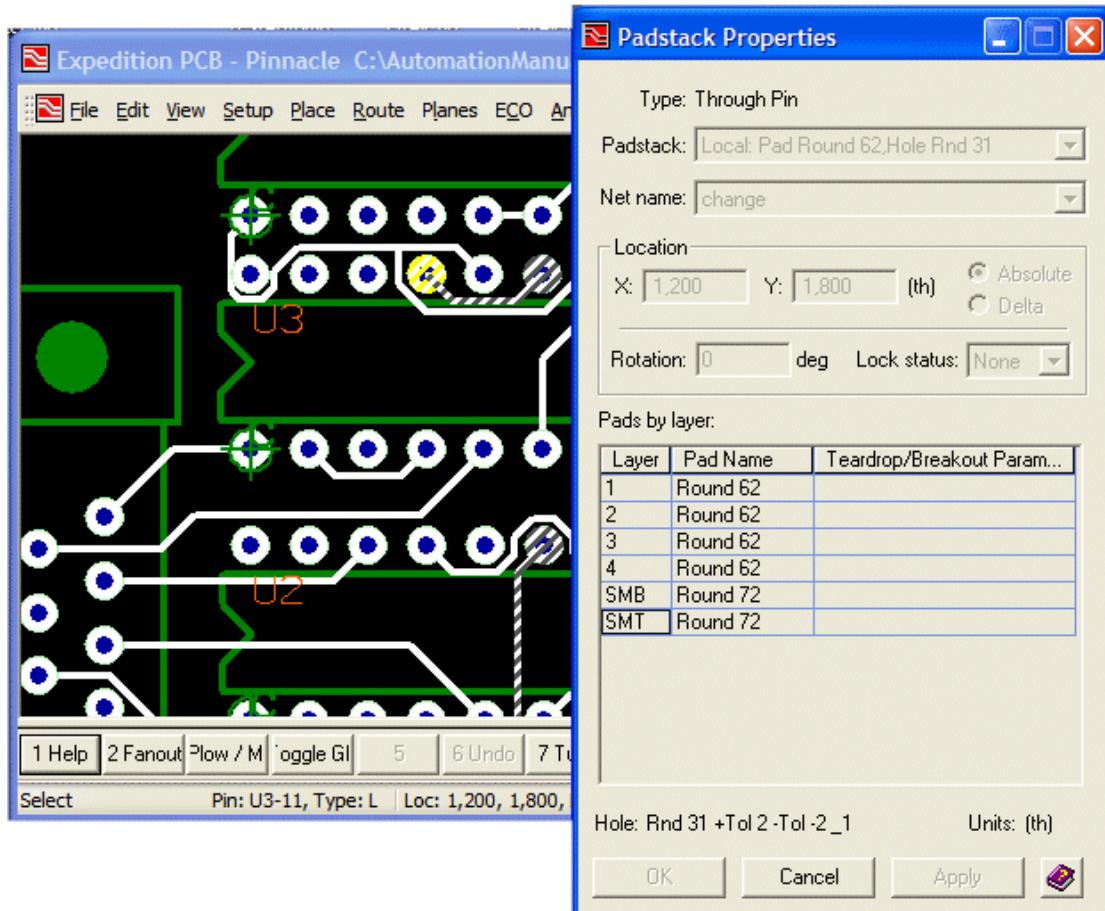


Figure 16-1. The padstack on U3.11 before running our script.

Let's assume that we've already created our form/script called *RemoveUnconnectedPads.efm*. Further assume that we launch this script (which is presented as a form/dialog) by dragging and dropping it onto our design.

Observe the two list areas called **Excluded Padstacks** (on the left) and **Included Padstacks** (on the right) as illustrated in Figure 16-2. Initially, these lists will be empty. If we now click the **Vias** checkbox we will see two via types (**VIA20** and **VIA31**) appear in the **Excluded Padstacks** list. Similarly, if we click the **Pads** checkbox we will see two pad types (**Pad Round 62, Hole Rnd 31** and **Pad Square 62, Hole Rnd 31**) appear in the **Excluded Padstacks** list.

If we now select the **Pad Round 62, Hole Rnd 31** item in the **Excluded Padstacks** list and then click on the button with the single right-pointing arrow, this item will disappear from the **Excluded Padstacks** list and reappear in the **Included Padstacks** list. The result will be as illustrated in Figure 16-2.

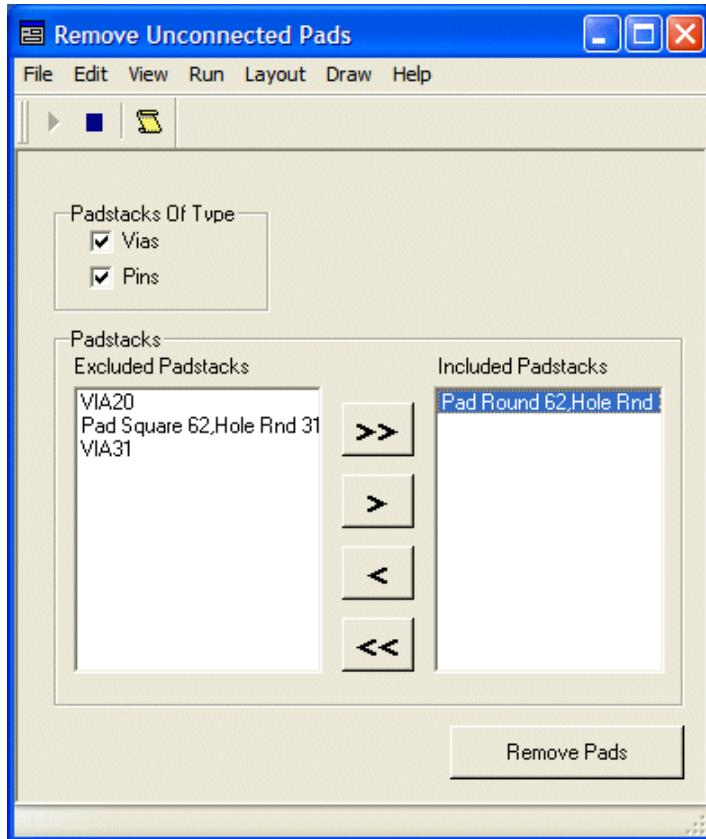


Figure 16-2. The Remove Unconnected Pads form/dialog we are going to create.

Now let's assume that we click the **Remove Pads** button. If we once again select pin 11 on component **U3**, and re-access the **Padstack Properties** dialog, the result will be as illustrated in Figure 16-3.

Observe that a new padstack has been 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 have been removed). Also observe that any unconnected pads on other pins on layer 1 (**U3.12**, **U3.13**, and **U3.14**, for example) have been removed leaving only the green solder mask pads.

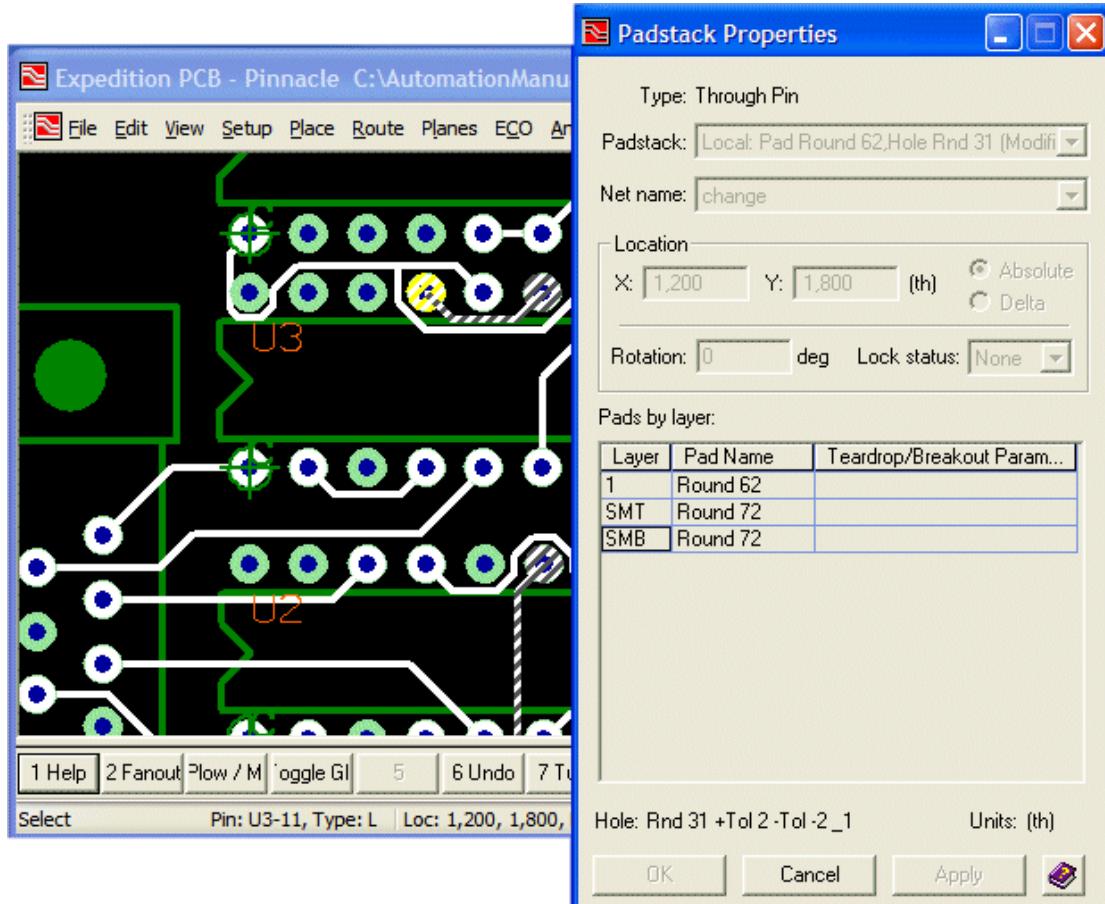


Figure 16-3. The padstack on U3.11 after running our script.

The Form Itself

First of all we are going to create the form, and then we are going to populate the various elements on the form with their portions of the script. The key scripting sections "behind" the form are as follows (these are shown in the order in which we will be creating them):

- **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 *Chapter 4: Introducing the IDE* and then create the form as illustrated in Figure 16-4. Use the **File > Save As** command to save the form in whatever folder you're using to store your scripts; save the form with the name *RemoveUnconnectedPads.efm* (as we discussed in *Chapter 4*, the *.efm* extension denotes a form, as compared to the *.vbs* extension used to denote a pure VBScript). As we start to add the code behind the various elements on the form, it's a good idea to keep on saving our work (trust us on this, if nothing else).

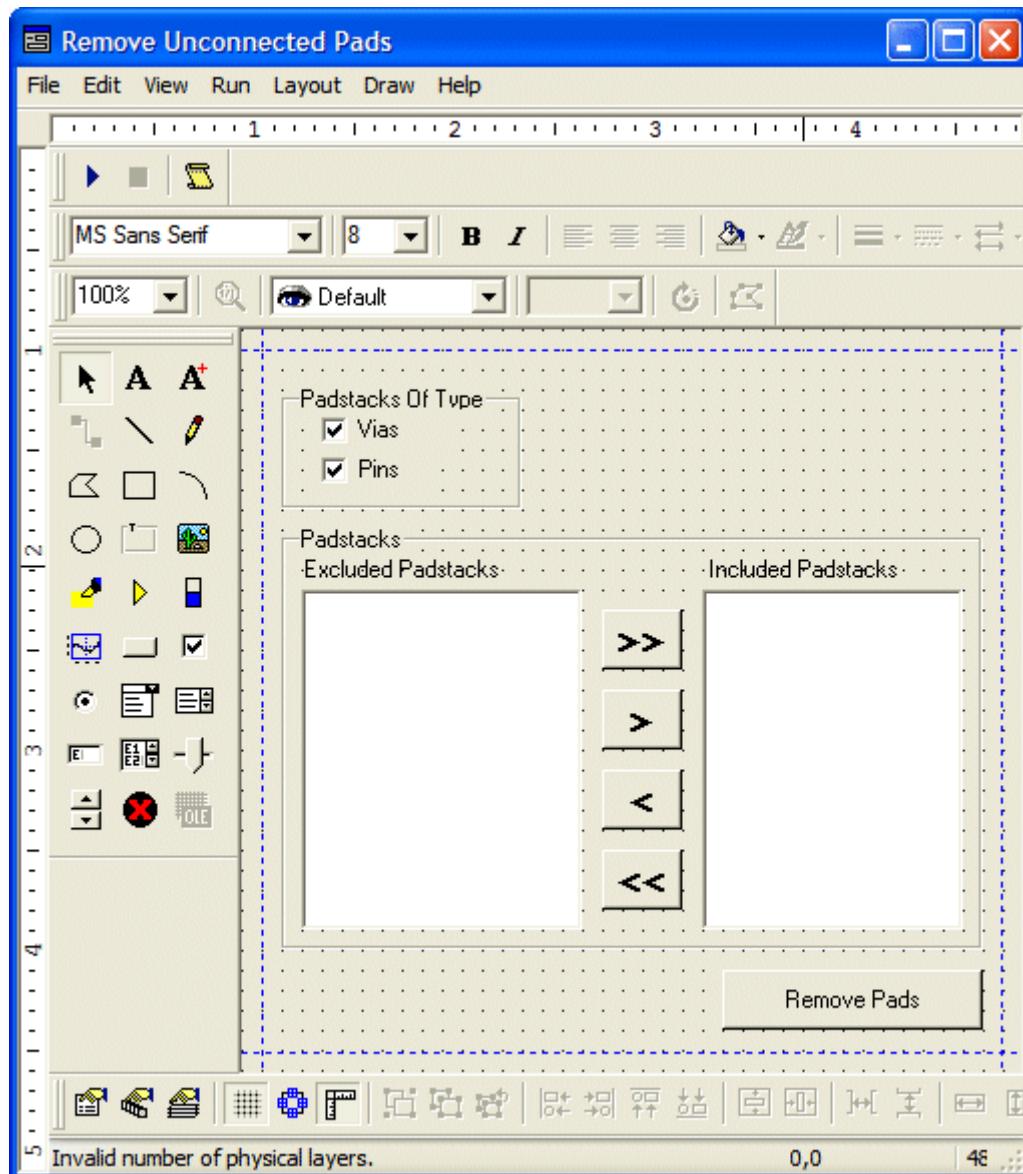


Figure 16-4. Creating the form/dialog.

Observe that there are two *Frames* on this form – one is labeled **Padstacks of Type** and the other is labeled **Padstacks**. Inside the **Padstacks of Type** frame we have two *Checkboxes* labeled **Vias** and **Pins**.

Meanwhile, inside the **Padstacks** frame we have two *List* boxes; the left- and right-hand boxes are annotated with free text saying **Excluded Padstacks** and **Included Padstacks**, respectively.

Also inside the **Padstacks** frame we have four buttons with text annotations of **>>**, **>**, **<**, and **<<**. Last but not least, we have the button at the bottom of the form carrying the **Remove Pads** annotation.

Naming Elements on the Form

Now, this bit is very important. We need to name all of the objects on the form with which we are going to associate scripting code. Right-click on the **Vias** checkbox control and select the **Object Properties** item from the resulting pop-up menu. Now change the assignment to the **(Object Code)** item from its default value to the name *viasChkObj* as illustrated in Figure 16-5.

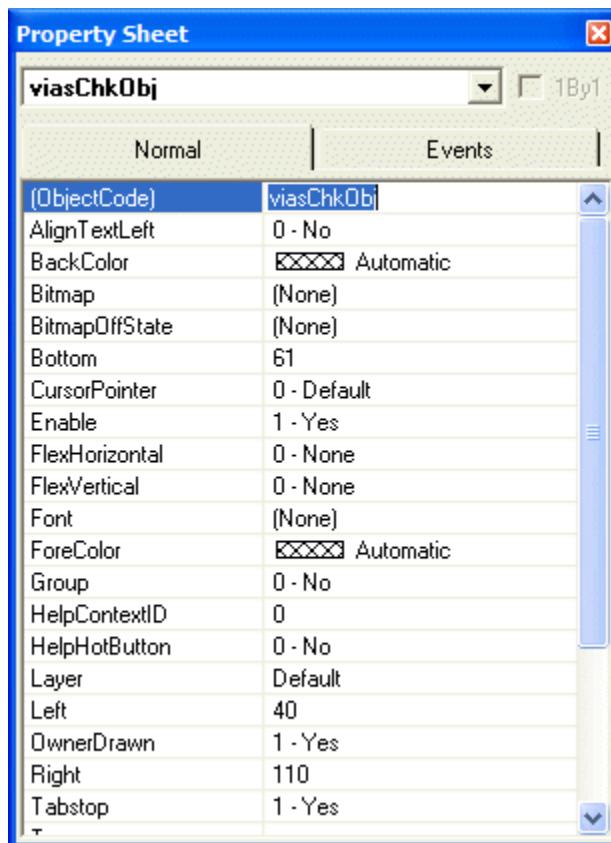


Figure 16-5. The Property Sheet dialog.

Observe that this name automatically appears in the field at the top of the form. This is the name that will be used by our script to access this control (the control is accessed in the same way we have been accessing other objects in the interface). Once you've made this modification, click the red 'X' button located in the upper right-hand corner of the form to dismiss this dialog.

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 one with the >> annotation) to *includeAllBtnObj*.
- Change the name of the **Include Selected** button (the one with the > annotation) to *includeBtnObj*.
- Change the name of the **Exclude Selected** button (the one with the < annotation) to *excludeBtnObj*.
- Change the name of the **Exclude All** button (the one with the >> annotation) to *excludeAllBtnObj*.
- Change the name of the **Remove Pads** button to *removePadsBtnObj*.

Preparing to Add Code "Behind" the Elements on the Form

Now that we've designed our form, it's time to add the code (sub-scripts) "behind" the various elements on the form. There are a number of mechanisms by which this may be achieved but, for our purposes here, simply click the **Script** icon and then make sure that the **Event View** icon is active as illustrated in Figure 16-6.

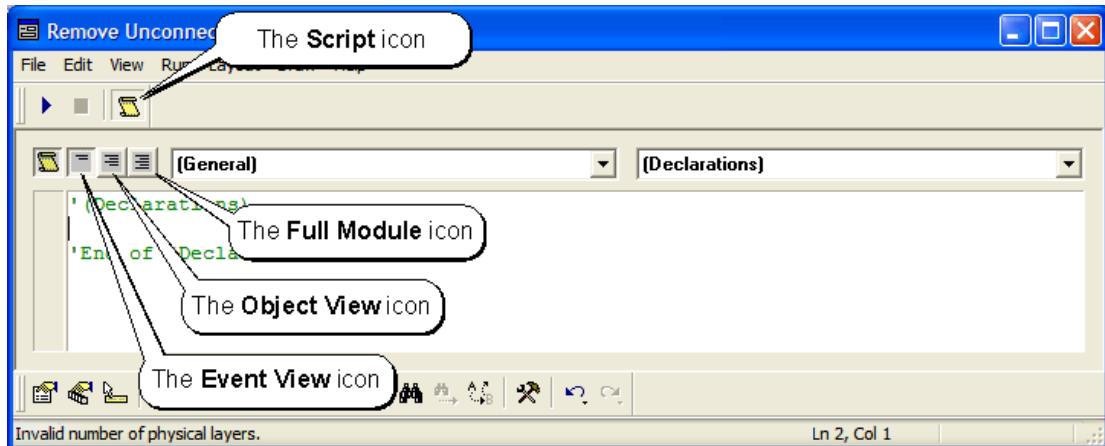


Figure 16-6. Preparing to add code "behind" the elements on the form.

Initialization and Setup

Ensure that **(General)** is selected in the left-hand (Object) list at the top of the form. This will cause the right-hand list to be automatically populated with the **(Declarations)** item as illustrated in Figure 16-7.

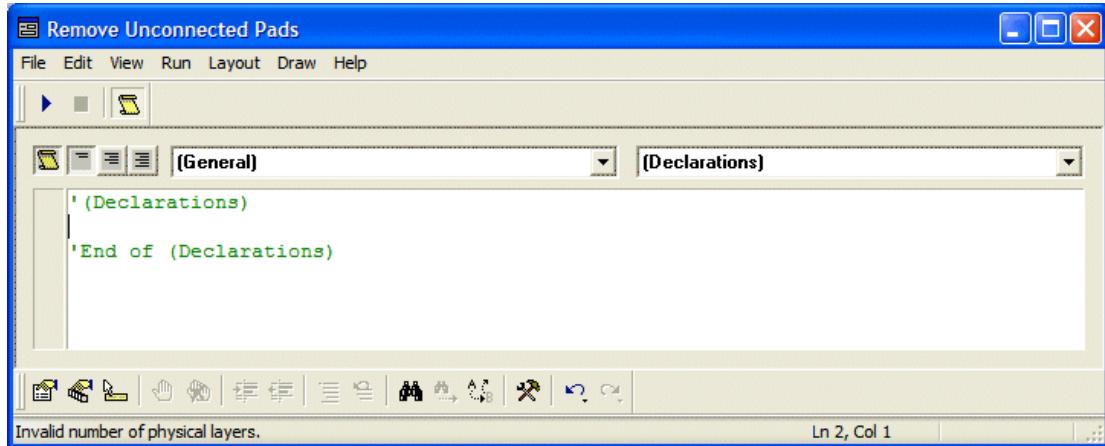


Figure 16-7. Preparing to add the initialization and setup code.

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 our standard *ValidateServer()* function.



Note: We are not going to discuss the *Initialization and Setup* code shown below in any detail, because this is absolutely standard and has been described in detail in previous chapters. By comparison, in the case of the remaining code sections associated with the various form functions, we will first show the relevant portion in its entirety for clarity, and then provide a line-by-line breakdown.



VERY IMPORTANT: Do NOT enter the line numbers shown in any of the code associated with this chapter. You don't need line numbers when populating a form. We have included them here only as an aid for our discussions.

```

1
2 Option Explicit
3
4 ' Add any type libraries to be used.
5 Scripting.AddTypeLibrary( "MGCPBC.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 .....
21 ' Miscelaneous Functions
22
23 ' Validate server function
24 Function ValidateServer(doc)
```

```

25
26     Dim key, licenseServer, licenseToken
27
28     ' Ask Expedition's document for the key
29     key = doc.Validate(0)
30
31     ' Get license server
32     Set licenseServer =
33         CreateObject("MGCPBCAutomationLicensing.Application")
34
35     ' Ask the license server for the license token
36     licenseToken = licenseServer.GetToken(key)
37
38     ' Release license server
39     Set licenseServer = nothing
40
41     ' Turn off error messages. Validate may fail if the token
42     ' is incorrect
43     On Error Resume Next
44     Err.Clear
45
46     ' Ask the document to validate the license token
47     doc.Validate(licenseToken)
48     If Err Then
49         ValidateServer = 0
50     Else
51         ValidateServer = 1
52     End If
53
54 End Function

```

Exclude List Initialization

As we shall see, this function gets 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 will be as illustrated in Figure 16-8.

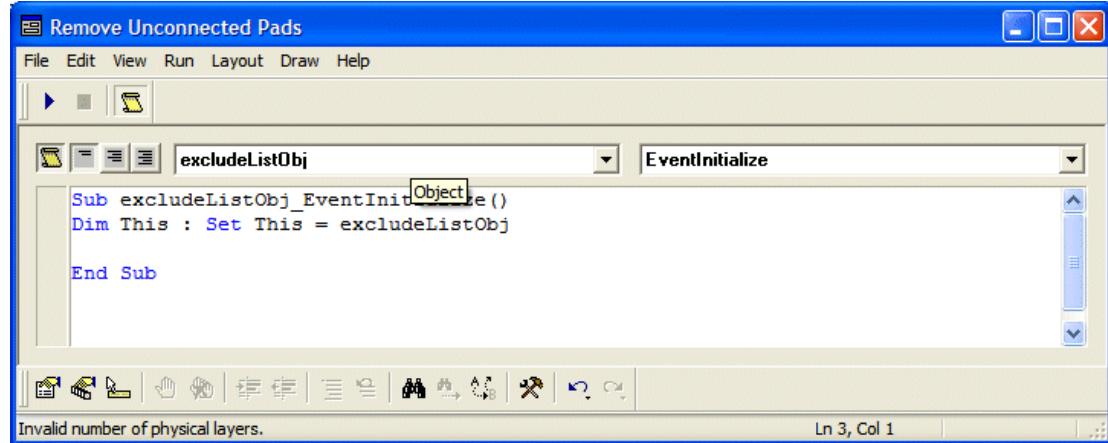


Figure 16-8. Preparing to add the Exclude List initialization code.

Place your cursor on the blank line before the *End Sub* statement and enter the following code from Lines 3 to 31 (without the line numbers, of course):

```
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 has been 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
23             pinsChkObj.Value = 1 And _
24             Not pstkObj.IsSMD) Or _
25             (pstkObj.Type = epcbPadstackObjectVia And
26             viasChkObj.Value = 1)) And _
27             Not dictObj.Exists(pstkObj.Name) Then
28                 ' Add it to the list
29                 Call This.AddString(pstkObj.Name)
30                 ' Add it to the dictionary so we know not to
31                 ' add it twice
32                 Call dictObj.Add(pstkObj.Name, True)
33             End If
34     Next
35 End Sub
```

Let's walk through this code line-by-line. The first thing we do on Line 4 is to clear any existing entries in the list.

```
4  Call This.ResetContent()
```

Now, the way we're going to make this form work is to present a single list entry for each excluded *Padstack* reference and all of its clones. In order to facilitate this, on Lines 12 and 13 we create a *Dictionary* object. The way a *Dictionary* works is that once we've added one or more names to it, we can easily and quickly check to see if any particular name has already been added.

```
12     Dim dictObj
13     Set dictObj = Createobject("Scripting.Dictionary")
```

On Lines 16 and 17 we 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
```

On Line 19 we instantiate a variable in which to hold a *Padstack* object. On Lines 20 through 31 we enter a loop that iterates through each *Padstack* object in our collection. For each Padstack object we check to see if (a) it's a *Pin Padstack* AND the **Pins** checkbox is active AND it's not a surface-mount device (SMD) or (b) it's a *Via Padstack* AND the **Vias** checkbox is active.

(Remember that when the form/dialog is first run, neither the **Pins** nor **Vias** checkboxes are selected/active, so nothing will be added to the list, which will therefore be empty.)

If either of these cases is true, on Line 25 we check to make sure that this entry doesn't already exist in our dictionary. Assuming it doesn't already exist, on Line 27 we add it to our list and on Line 29 we add it to our *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
23              pinsChkObj.Value = 1 And _
24              Not pstkObj.IsSMD) Or _
25              (pstkObj.Type = epcbPadstackObjectVia And
26              viasChkObj.Value = 1)) And _
27              Not dictObj.Exists(pstkObj.Name) Then
28                  ' Add it to the list
29                  Call This.AddString(pstkObj.Name)
30                  ' Add it to the dictionary so we know not to
31                  ' add it twice
32                  Call dictObj.Add(pstkObj.Name, True)
33      Next
34 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 will be as illustrated in Figure 16-9.

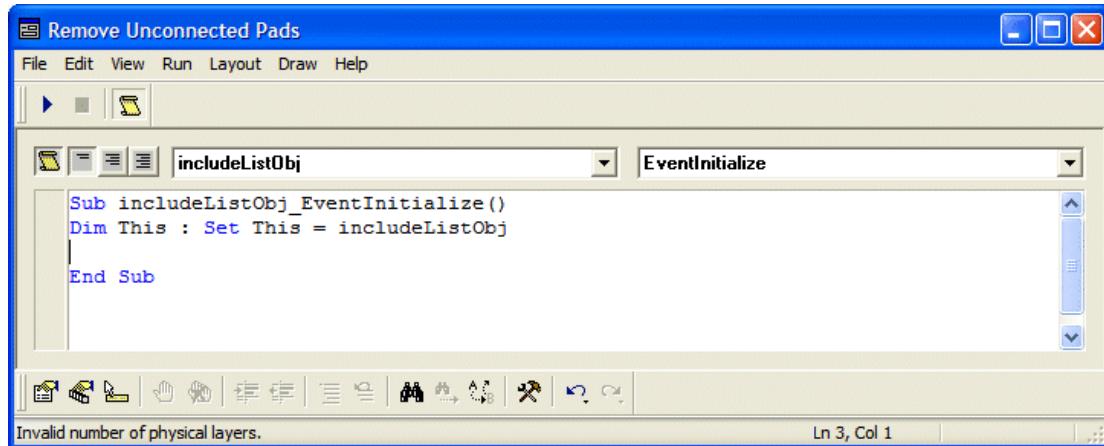


Figure 16-9. Preparing to add the Include List initialization code.

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
```

As we can see, this function is really easy, because all we do on Line 4 is clear 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 is changed; in the case of a checkbox, it is fired whenever that box is checked or unchecked). The result will be as illustrated in Figure 16-10.

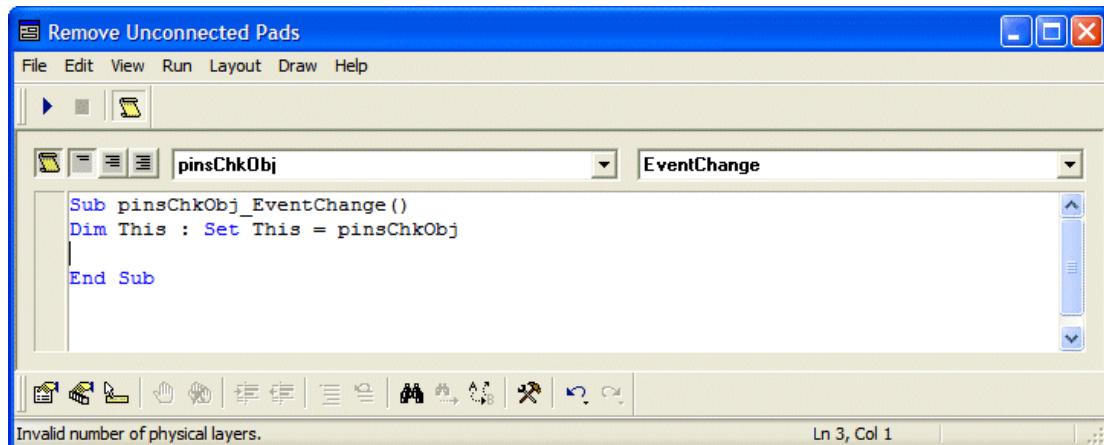


Figure 16-10. Preparing to add the Pins checkbox Event Change Handler code.

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
```

All this says is that whenever the state of the **Pins** checkbox changes (that is, irrespective as to whether it is selected or de-selected), on Lines 4 and 5 we 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 will be as illustrated in Figure 16-11.

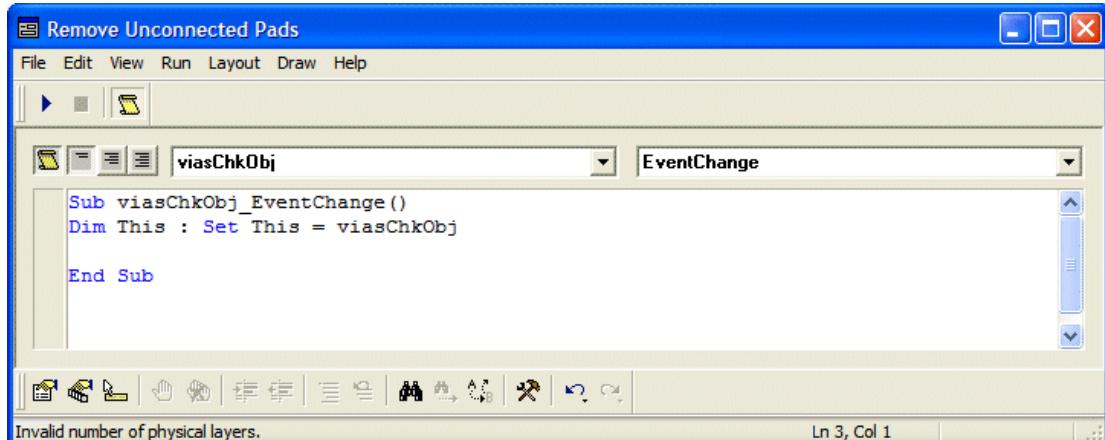


Figure 16-11. Preparing to add the Vias checkbox Event Change Handler code.

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

```

All this says is that whenever the state of the **Vias** checkbox changes (that is, irrespective as to whether it is selected or de-selected), on Lines 4 and 5 we 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 will be as illustrated in Figure 16-12.

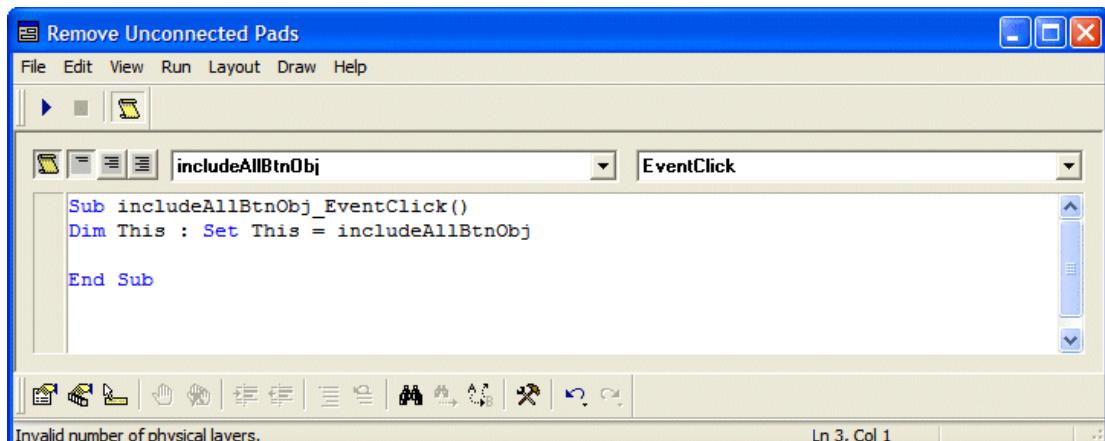


Figure 16-12. Preparing to add the Include All button code.

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
8       includeListObj.AddString(excludeListObj.GetText(entryInt))
9         ' Remove the entry from the exclude list
10        call excludeListObj.DeleteString(entryInt)
11      End If
12    Next
13 End Sub

```

As we see, this is a very simple piece of code. On lines 5 through 11 we iterate our way through any items that are currently in the **Exclude List**; for each of these items we add them to the **Include List** and then delete them from the **Exclude List**.



Note: This button – and also the other **Include/Exclude** buttons – are implemented in such a way that they could work for any list contents and not just padstack names.



Note: It is possible to copy a control from one form to another. When this is done, all of the code associated with that control will also be copied over.

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 will be as illustrated in Figure 16-13.

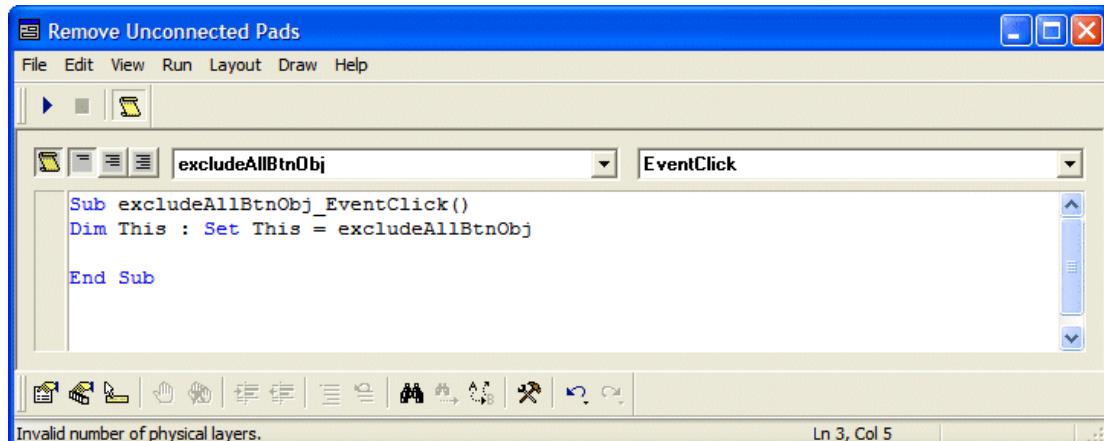


Figure 16-13. Preparing to add the Include All button code.

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

```

```

9         excludeListObj.AddString(includeListObj.GetText(entryInt))
10            ' Remove the entry from the include list
11            Call includeListObj.DeleteString(entryInt)
12        End If
12    Next
12 End Sub

```

Once again, this is a very simple piece of code. On lines 5 through 11 we iterate our way through any items that are currently in the **Include List**; for each of these items we add them to the **Exclude List** and then delete them 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 will be as illustrated in Figure 16-14.

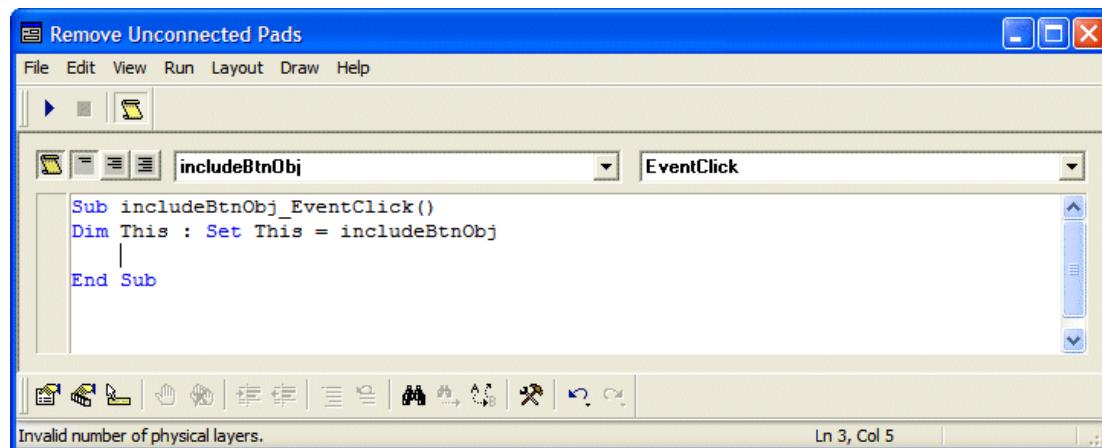


Figure 16-14. Preparing to add the Include Selected button code.

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
9             includeListObj.AddString(excludeListObj.GetText(entryInt))
10            ' Remove it from the exclude list
11            Call excludeListObj.DeleteString(entryInt)
11        End If
12 End Sub

```

As we see, this is similar to the code for the **Include All** button – the main difference is that there is no loop. Instead, on Line 5 we acquire the currently selected entry in the **Exclude List**; on Line 8 we add this entry to the **Include List**; and on Line 10 we delete the entry 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 will be as illustrated in Figure 16-15.

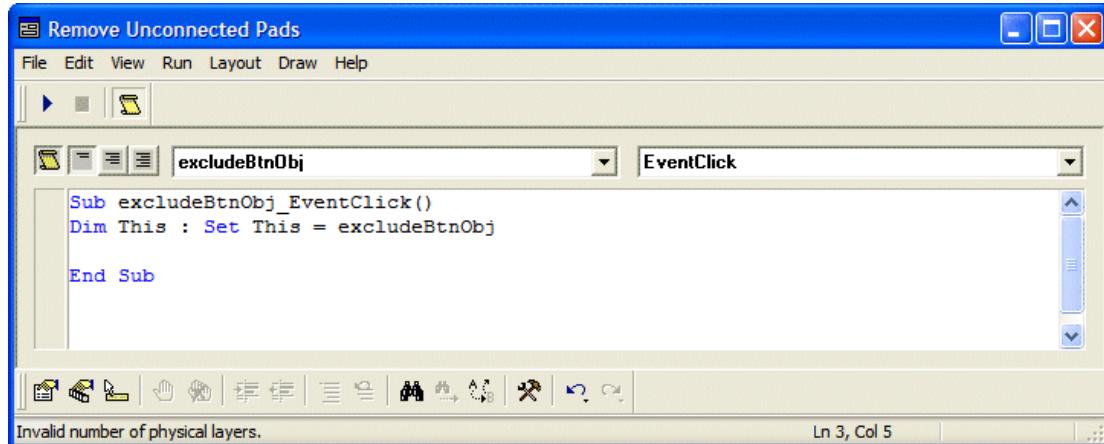


Figure 16-15. Preparing to add the Exclude Selected button code.

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
9              excludeListObj.AddString(includeListObj.GetText(entryInt))
10             ' Remove it from the exclude list
11             Call includeListObj.DeleteString(entryInt)
12     End If
13 End Sub
```

Not surprisingly, this is similar to the code for the **Exclude All** button – the main difference is that there is no loop. Instead, on Line 5 we acquire the currently selected entry in the **Include List**; on Line 8 we add this entry to the **Exclude List**; and on Line 10 we delete the entry 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 will be as illustrated in Figure 16-16.

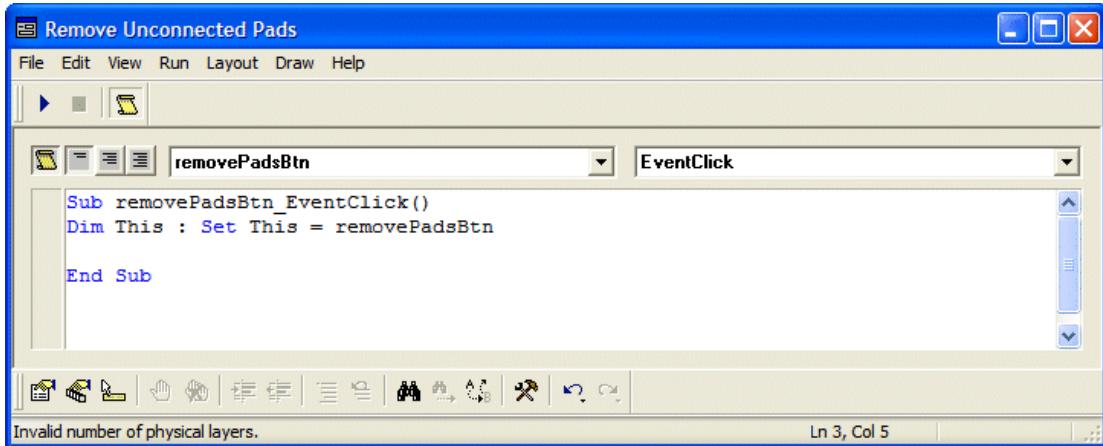


Figure 16-16. Preparing to add the Remove Pads button code.

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 =
27                             pstkNameStr Then
28                 ' Add it to the collection to be
29                 ' processed
30                 pinViaColl.Add(pinObj)
31                 ' And remove it so we don't process
32                 ' it next time
33                 pinColl.Remove(i)
34             End If
35         Next
36         For i = viaColl.Count To 1 Step -1
37             Set viaObj = viaColl.Item(i)
38             If viaObj.OriginalPadstack.Name =
39                             pstkNameStr Then
40                 ' Add it to the collection to be
41                 ' processed
42                 pinViaColl.Add(viaObj)
```

```

38                               ' And remove it so we don't process
39                               ' it next time
40                               viaColl.Remove(i)
41                           End If
42                       Next
43                   End If
44               Next
45           pcbDocObj.TransactionStart
46           ' Remove the unconnected pads.
47           Call RemoveUnconnectPads(pinViaColl)
48           pcbDocObj.TransactionEnd
49
50 End Sub

```

The idea behind the **Remove Unconnected Pads** button is that we want to build up a simple collection that includes both pins and vias that use the *PadstackObject* objects specified in the **Include List**. Once we have this collection, we want to remove any unconnected pads from the pins and vias in the collection.

On Lines 6 and 7 we use the *NewObjects* property of the *Utility* object to create a heterogeneous object collection that can contain objects of different types.

```

5      ' Create an empty generic collection
6      Dim pinViaColl
7      Set pinViaColl = pcbAppObj.Utility.NewObjects()

```

On Lines 10 and 11 we 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

```

On Lines 14 and 15 we 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 we iterate through each item (*Padstack* name) in our **Include List**. On Line 21 we get the name for the current entry and on Line 22 we check to make sure that this 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

```

Assuming that it's not empty, then on Lines 24 through 32 we iterate through our *Pin* collection in reverse order. On Line 26 we check to see if this *Pin* object has the same *Padstack* name as the current *Padstack* we're looking at. If it does, then on Line 28 we add it to our heterogeneous *Pin/Via* collection and on Line 30 we remove this *Pin* object from the *Pin* collection (we perform this removal only as a performance enhancement because it will save us from having to check this *Pin* the next time round).

```

23
24      Dim i
25      For i = pinColl.Count To 1 Step -1
26          Set pinObj = pinColl.Item(i)
              If pinObj.OriginalPadstack.Name =
                  pstkNameStr Then

```

```

27                               ' Add it to the collection to be
28                               processed
29                               pinViaColl.Add(pinObj)
30                               ' And remove it so we don't process
31                               it next time
32                               pinColl.Remove(i)
31                           End If
32                           Next

```

Similarly, on Lines 33 through 41 we iterate through our *Via* collection in reverse order. On Line 35 we check to see if this *Via* object has the same *Padstack* name as the current *Padstack* we're looking at. If it does, then on Line 37 we add it to our heterogeneous *Pin/Via* collection and on Line 39 we remove this *Via* object from the *Via* collection (again, we perform this removal only as a performance enhancement because it will save us from having to check this *Via* the next time round).

```

33                           For i = viaColl.Count To 1 Step -1
34                               Set viaObj = viaColl.Item(i)
35                               If viaObj.OriginalPadstack.Name =
36                                   pstkNameStr Then
36                                   ' Add it to the collection to be
37                                   processed
37                                   pinViaColl.Add(viaObj)
38                                   ' And remove it so we don't process
39                                   it next time
39                                   viaColl.Remove(i)
40                               End If
41                           Next
42                       End If
43                   Next

```

On Line 45 we start a transaction, which will allow us to undo all of the changes made by this routine with a single click of the **Undo** button. On Line 47 we call our *RemoveUnconnectedPads()* helper function and pass it the collection they just built. On Line 48 we end 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 will cause the right-hand list to be automatically populated with the **(Declarations)** item as illustrated in Figure 16-17.

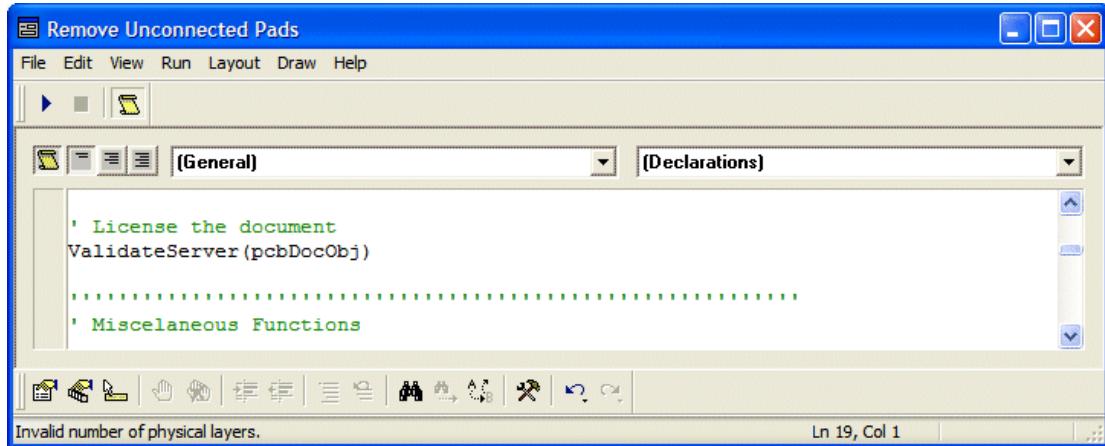


Figure 16-17. Preparing to add the Exclude List initialization code.

Scroll down 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()* and *GetConnectedLayers()*:

```

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         ' Initialize 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
27             ' is a pad on the layer delete it
28             If (Not layerArr(i)) And
29                 (Not newPstkObj.Pads(i).Count = 0 )Then
30                 lyrObject.ConductorLayer = i
31                 newPstkObj.DeletePad(lyrObject)
32                 modifiedBool = True
33             End If
34         Next
35
36         ' Apply cloned padstack back to the padstack object

```

```

36             If modifiedBool Then
37                 pstkObjObj.CurrentPadstack = newPstkObj
38             End If
39         Next
40     End Sub
41
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)
47         ' Initialize 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)
52         Dim i
53         For i = 1 to UBound(layerArr)
54             layerArr(i) = False
55         Next
56
57         ' Determine which layers are missing connections
58         Dim connectedObjColl
59         Set connectedObjColl = pstkObjObj.ConnectedObjects
60
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
69
70         ' Return the array
71         GetConnectedLayers = layerArr
72     End Function

```

RemoveUnconnectPads() Routine

This routine accepts as input a heterogeneous 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 we use a *For ... Next* loop to iterate through this collection. On Line 14 we call our helper function *GetConnectedLayers()*, which returns an array of the layers that specifies whether or not each layer has a connection.

```

10            ' Initialize 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)

```

Now, the only way we can modify a *PadstackObject* object is to first clone it, which we do on Lines 17 and 18.

```

16          ' Padstacks must be cloned to be modified.
17          Dim newPstkObj
18          Set newPstkObj = pstkObjObj.CurrentPadstack.Clone()

```

On Lines 21 and 22 we create a new *Layer* object (as we shall see, this is required by the *DeletePad* method). On Line 23 we create a Boolean flag that we will use to keep track of whether or not we make a modification and we initialize this flag to *False*.

```

20          ' Remove all the pads
21          Dim lyrObject
22          Set lyrObject = pcbAppObj.Utility.NewLayerObject()
23          Dim modifiedBool : modifiedBool = False

```

On Lines 25 through 33 we iterate through our connected layers array. On Line 28 we check to see if the entry in our unconnected layer array is *False* (indicating no connection) and – if so – we check to see if there is a *Pad* on this layer. If so, on Line 29 we set our *Layer* object to the current layer; on Line 30 we delete the *Pad*; and on Line 31 we set our 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
29                  (Not newPstkObj.Pads(i).Count = 0) Then
30                  lyrObject.ConductorLayer = i
31                  newPstkObj.DeletePad(lyrObject)
32                  modifiedBool = True
33          End If
            Next

```

On Line 36 we check to see if we did make a modification and – if so – we assign our cloned *PadstackObject* back to its original *PadstackObject* object (*Pin* or *Via*).

```

35          ' Apply cloned padstack back to the padstack object
36          If modifiedBool Then
37              pstkObjObj.CurrentPadstack = newPstkObj
38          End If
39      Next
40  End Sub

```

GetConnectedLayers() Routine

This routine 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 we instantiate a *Layer* array and on Line 51 we initialize it to contain the number of layers on the board. Now, remembering that arrays start counting from 0, we could create an array from 0 to (*LayerCount* – 1), but we decided that it was easier to create an array containing from 0 to *LayerCount* and then only work with array items 1 to *Layercount* (this is easier because the board's layers are numbered from 1 upwards).

```

47      ' Initialize 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)

```

On Lines 53 through 55 we iterate through our 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

```

On Lines 58 and 59 we acquire the heterogeneous collection of *Connected* objects for this *PadstackObject* object.

```

57  ' Determine which layers are missing connections
58  Dim connectedObjColl
59  Set connectedObjColl = pstkObjObj.ConnectedObjects

```

On Lines 62 through 68 we iterate through this collection; whenever we find a connected object we set the corresponding Boolean flag in our array to be *True* (that is, the *Layer* property of each *Connected* object in our collection is used to index into our 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, we set the return value to be our array of Boolean values and we exit the routine.

```

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 as discussed above.

-
- 2) Close any instantiations of Expedition PCB that are currently running, and then launch a new instantiation of this application.
 - 3) Open the *Candy.pcb* design we've been using throughout these examples and save it out 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 could be augmented as discussed in the following topic.

Enhancing the Script

There are a number of ways in which this script could be enhanced. Some suggestions are as follows (it would be a good idea for you to practice your scripting skills by implementing these examples):

- It would be useful to add a **Cancel** button to the form so that it can be dismissed without performing any actions.
- When the **Remove Pads** button is clicked on the form/dialog, it would be useful if the script generated a report and/or presented the user with some feedback as to what modifications the script had made to the design.
- It would be useful to add a check box allowing/disabling the removal of pads on the top and bottom routing layers.
- It would be useful to select/highlight and pins/vias that are modified by the script.
- With regard to the previous point, it would be useful to be able to click on one of the selected/highlighted pins/vias and be presented with a *MsgBox()* providing "Was-Is" type information for this pin/via.

Chapter 17: Integration with Excel

Introduction

Overview:	This script creates a new Excel® spreadsheet and then automatically loads the spreadsheet with component information that it extracts from the layout design document. The script also provides a cross-probing capability from the layout design document 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:	<ul style="list-style-type: none">– Driving two automation servers (Expedition PCB and Excel) from the same script.
Items Used:	<ul style="list-style-type: none">– Excel Server– OnSelectionChange Expedition PCB Event– VBScript Err Object
Special Notes:	This script will work only on the Windows® operating system and Excel must be installed. See also <i>Chapter 18</i> for details on a cross-platform equivalent to this script.

Before We Start

It is often advantageous to be able to take data from a layout design document and to present this data in different ways; for example, in an Excel spreadsheet. It is even more beneficial if interaction can be provided between the two applications.

This script generates an Excel document, populates it with component information, and then waits for selections to be made in the layout design document. When a selection is made in the layout design document, the script automatically locates, selects, and highlights the corresponding items in the Excel spreadsheet.

Just to get a feel for the way in which our script is going to work, let's assume that we've launched Expedition PCB, opened our *Candy.pcb* layout design document, and executed the script. The result will be something like that shown in Figure 17-1.

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 ordered as an alpha-numerical sort on their reference designators).

As noted earlier, the script also supports cross-probing between Expedition PCB and Excel. When one or more components are selected in the layout design document, the rows associated with the corresponding components in the Excel spreadsheet are located, selected, and highlighted (yellow) as illustrated in Figure 17-2.

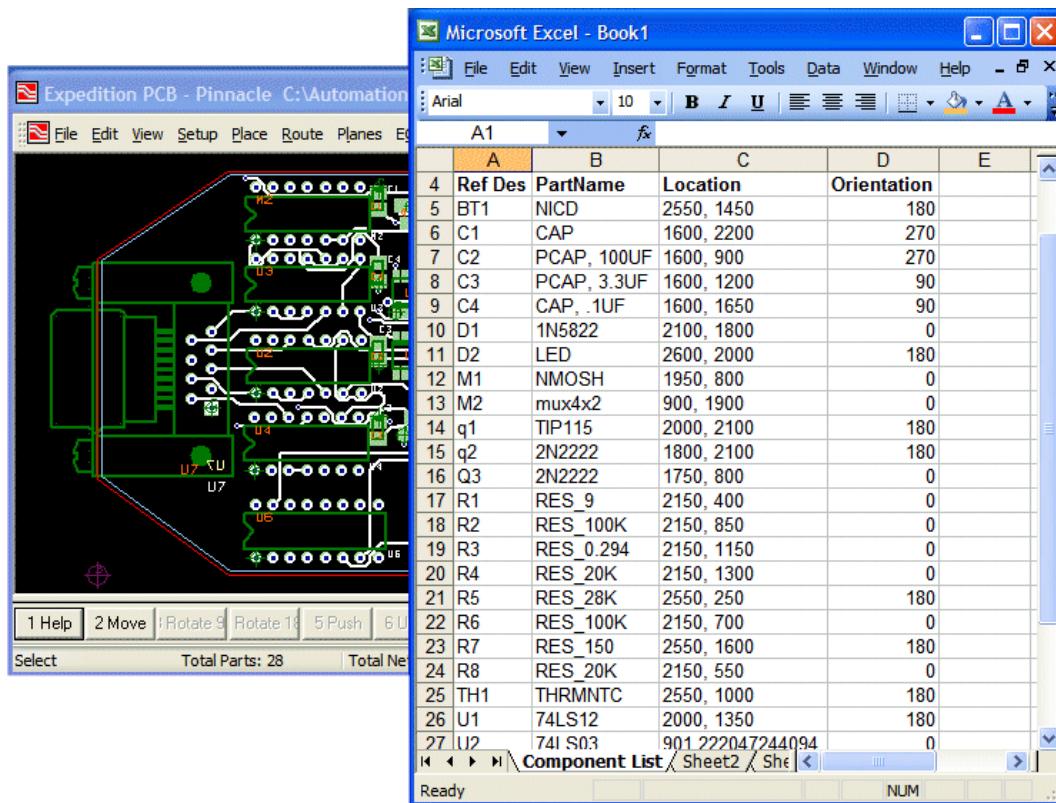


Figure 17-1. The script automatically generates an Excel spreadsheet.

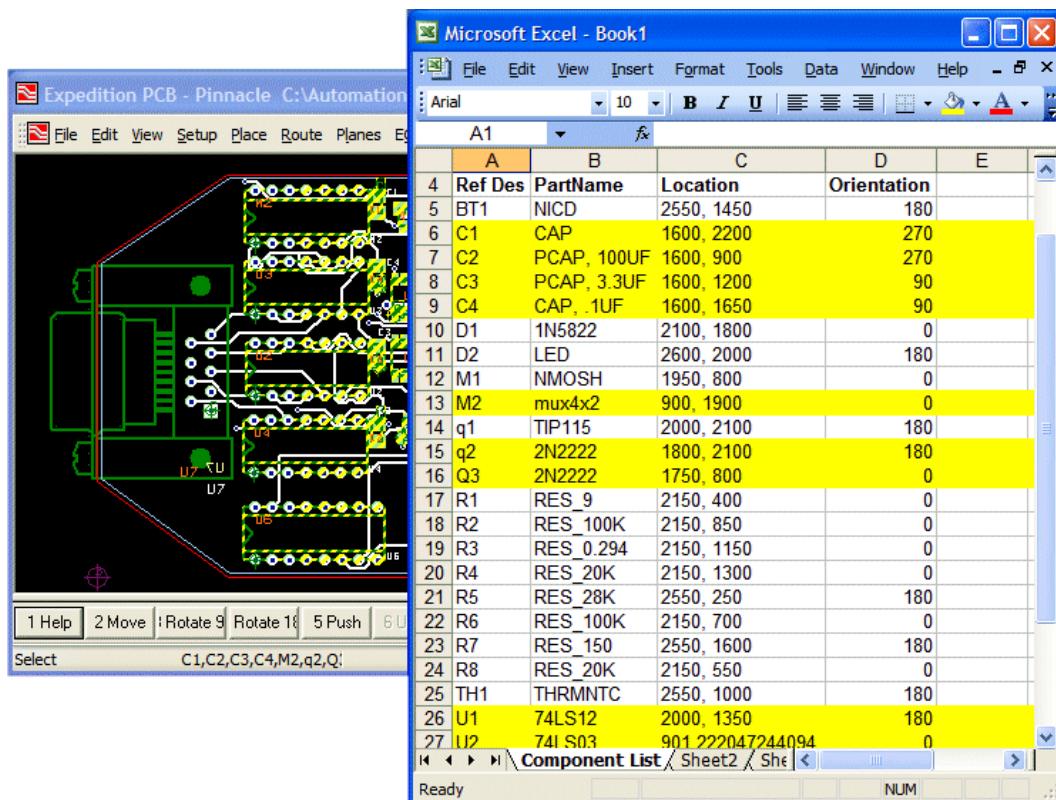


Figure 17-2. The script cross-probing between Expedition PCB and Excel.

The Script Itself

The key sections forming this script are as follows:

- **Constant Definitions** Define the constants to be used by the script.
- **Initialization/Setup** Get the *Application* and *Document* objects, license the document, and add the type libraries. Also perform any additional setup associated with this script.
- **Event Handler** When a new selection is made in the Expedition PCB layout design document, thereby triggering an event, reflect this selection in the Excel spreadsheet.
- **Load Excel Subroutine** This is the main subroutine that populates the Excel spreadsheet with the component information extracted from Expedition PCB.
- **Utility Functions and Subroutines** Special "workhorse" routines that are focused on this script.
 - SelectComponentsInExcel()
 - DefineHeaders()
 - AddComponents()
 - AddComponent()
- **Helper Functions and Subroutines** Generic "helper" routines that can be used in other scripts.
 - ExcelsRunning()
 - SelectExcelRange()
 - RemoveExcelFill()
- **Validate Server** The standard function used to license the document and validate the server. (Note that we won't go through this function in this chapter – for more details on this function see *Chapter 3: Running Your First Script*.)

Constant Definitions

Lines 10 through 23 define the constants to be used by this script. Line 10 specifies the name we are going to apply to the bottom of the Excel spreadsheet.

```
1  ' This script creates excel and loads it with component info.
2  ' The script will also support 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"
```

In Lines 12 through 15, we assign column labels in the spreadsheet to constant names in the script. In addition to making the script more readable, this makes it easy for us to rearrange the column order in the future if we so desire.

```
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 we declare constants that we will use to ensure that our header information appears on row 4 of the spreadsheet and that the information associated with the

first component will appear on row 5 (we could have started these at row 1, but 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
```

Due to the way in which Microsoft® implemented the Excel type library, it is not possible to import this library using the *AddTypeLibrary* method. In order to get around this, on Lines 22 and 23 we declare the enumerates we want to use 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
```

Actually, we are only going to use *xlwhole*; we declared the other alternative for this enumerate for clarity. You can use Excel Help or an Object Browser to discover more information on the various aspects of Excel automation (see also *Appendix D: General VBScript References and Tools*).

Initialization/Setup

Lines 25 through 55 are used to perform the initialization and setup tasks. Lines 25 through 40 involve adding type libraries, getting the *Application* and *Document* objects, and licensing the document (validating the server) as usual. The only difference to previous scripts occurs on Line 31, which is where we declare a variable to hold an Excel application object

```
25 ' Add any type libraries to be used.
26 Scripting.AddTypeLibrary("MGCPBC.ExpeditionPCBAApplication")
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 we create an Excel application. On Line 46 we call our main *LoadExcel()* subroutine, which is used to populate the spreadsheet we just created.

```
42 ' Create the Excel application
43 Set excelAppObj = CreateObject("Excel.Application")
44
45 ' Load Excel
46 Call LoadExcel()
```

Once the Excel spreadsheet has been loaded, on Line 49 we make it visible to the user. Generally speaking it is better to make applications like this spreadsheet visible after they've been loaded, organized, resized, and so forth, because watching these actions take place can be a little "messy" (the main exception to this rule is when you are debugging your script, in which case you may decide to make the application visible before you start loading it).

```

48  ' Make the excel application visible.
49  excelAppObj.Visible = True

```

On Line 52 we attach events to the Expedition PCB layout design document so as to receive notification of any selection changes.

```

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

```

On Line 55 we set the *DontExit* property of the *Scripting* object to the Boolean value of *True*. This means this script will continue running until the end of the current application session, thereby allowing the script to detect and handle events.

Event Handler

Lines 62 through 68 are used to declare an event handler subroutine that is triggered when a new selection is made in the Expedition PCB layout design document. Line 62 declares the subroutine. Observe the *typeEnum* parameter being passed into the subroutine. This parameter tells us specifics about the type of selection being made. Expedition PCB is supposed to assign values to this parameter for you. This is intended for future use, and should be employed by your scripts at that time. At the time of this writing, however, this parameter is not actually used, but it 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)

```

On Line 63 we call our *ExcelIsRunning()* helper function to check to make sure that Excel is still running (the user could have exited the Excel application, in which case our script would fail). On Lines 64 and 65 we obtain the collection of objects that are currently selected in our layout design document. On Line 66 we call our *SelectComponentsInExcel()* utility routine to locate, select, and highlight the components defined by our collection.

```

63      If ExcelIsRunning(excelAppObj) Then
64          Dim cmpsColl
65          Set cmpsColl =
66              pcbDocObj.Components(epcbSelectSelected)
67          Call SelectComponentsInExcel(cmpsColl)
68      End If
69  End Sub

```

Load Excel Subroutine

Lines 74 through 95 are where we declare the subroutine that loads the Excel spreadsheet with information about the components in the Expedition PCB layout design document.

Earlier, on Line 43, we created the Excel application (you can think of this as invoking Excel, but in "COM talk" we say that we have created an *Application* object). Now, on Lines 76 and 77, we use Excel's *Add* method on its *Workbooks* collection to add a new *Workbook* object (essentially, this is where we actually create 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 is created containing three *WorkSheet* objects. Thus, we use Lines 80 and 81 to acquire the first *Worksheet* object.

```

79      ' Get the first sheet
80      Dim cmpListSheetObj
81      Set cmpListSheetObj = workbookObj.Worksheets.Item(1)

```

On Line 84 we rename our worksheet by setting its *Name* property to the string constant we declared on Line 10.

```

83      ' Rename the worksheet.
84      cmpListSheetObj.Name = SHEETNAME_COMP_LIST

```

On Line 87 we call our *DefineHeaders()* utility routine and we pass it the current *Worksheet* object as a parameter.

```

86      ' Set the header information
87      Call DefineHeaders(cmpListSheetObj)

```

On Lines 90 and 91 we get a collection of all of the components in the Expedition PCB layout design document.

```

89      ' Get the components
90      Dim cmpColl
91      Set cmpColl = pcbDocObj.Components

```

On Line 94, we use the inbuilt *Sort* method to sort the collection. (Note that this method performs an alpha-numeric sort on the component reference designators. We could create our own *Sort()* routine if required, or we could add the components to the spreadsheet in an unsorted form and then use Excel's sort functions.)

```

93      ' Sort the component collection
94      Call cmpColl.Sort()

```

On Line 97 we call our *AddComponents()* utility routine, which we will use to add the data associated with our component collection into the Excel spreadsheet.

```

96      ' Add the collection
97      Call AddComponents(cmpListSheetObj,
                           FIRST_COMPONENT_ROW, cmpColl)
98  End Sub

```

Utility Functions and Subroutines

This is where we declare some utility routines. These routines are similar in context to our generic "helper" routines (see the next topic), except that they are specific to this script and may not be used elsewhere without some "tweaking".

SelectComponentsInExcel() Subroutine

Lines 105 through 131 declare the routine that is used to locate, select, and highlight component rows in the Excel spreadsheet. The only parameter passed into this routine is a collection containing the components we want to select.

```
103  ' Select the components in the spreadsheet.  
104  ' cmpsColl - Component Collection  
105  Sub SelectComponentsInExcel(cmpsColl)
```

On Lines 107 and 108 we get the *Worksheet* object that contains our main component list from Excel (we know this is the first sheet because that's the one we added the components to earlier).

```
106      ' Get the sheet  
107      Dim cmpListSheetObj  
108      Set cmpListSheetObj = excelAppObj.Sheets.Item(1)
```

In Excel, a *Range* object can be used to represent 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 are used to iterate through the collection of components we want to select. On Lines 119 and 120, we acquire the entire reference designator *Column* as a *Range* object. Now, we don't actually see this occur on the screen, but the effect is the same as if we'd selected this column by hand as illustrated in Figure 17-3.

Using this range, we use the *Find* method and pass it the component object name (its reference designator) as the first parameter. Observe the use of the *xlWhole* constant that we declared on Line 22. This instructs Excel that we are only interested in a complete match in the component name; that is, we aren't interested in any partial matches. (Observe the group of three commas in the middle of the parameter list to the *Find* method; this indicates that we are accepting 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)
```

A screenshot of a Microsoft Excel window titled "Microsoft Excel - Book1". The spreadsheet contains a table with columns labeled "Ref Des", "PartName", "Location", and "Orientation". The "Ref Des" column is highlighted in blue, indicating it is selected. The data includes various components like BT1, C1, C2, C3, C4, D1, D2, M1, M2, q1, q2, Q3, R1, R2, R3, R4, R5, R6, R7, R8, TH1, U1, and U2, along with their respective part names, locations, and orientations.

	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	741S03	901 222047244094	0	

Figure 17-3. Selecting the reference designator column (Column A).

On Line 122 we check to see if we found a cell containing the current component's reference designator. Let's assume that we did in fact find component C3. Once again, we don't actually see this occur on the screen, but we can visualize this as looking something like Figure 17-4 (except that we haven't actually selected this row).

Assuming we have found a cell containing the current component's reference designator, then on Line 123 we check to see if our *Range* object is empty (in which case it will equate to *Nothing*) or if it already contains something. If it is empty, then on Line 127 we load it with all of the cells forming the row associated with this component. Alternatively, if the *Range* object already contains something, then on Lines 124 and 125 we execute the *Union* method on the existing *Range* object and our new row so as to create a new *Range* object that contains all of the desired 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 =
125                  excelAppObj.Union(multiRangeObj, _
126                      cmpListSheetObj.Rows(foundCellObj.Row))
127          Else
128              Set multiRangeObj =
129                  cmpListSheetObj.Rows(foundCellObj.Row)
130          End If
131      End If
132      Next

```

The screenshot shows a Microsoft Excel window titled "Microsoft Excel - Book1". The active sheet is "Sheet2". A table is displayed with columns labeled "Ref Des", "PartName", "Location", and "Orientation". Row 8, which contains the entry "C3 PCAP, 3.3UF 1600, 1200 90", is highlighted with a blue selection bar. The rest of the rows are white. The status bar at the bottom shows "Sum=90".

A	B	C	D	E
Ref Des	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	
R6	RES_100K	2150, 700	0	
R7	RES_150	2550, 1600	180	
R8	RES_20K	2150, 550	0	
TH1	THRMNTC	2550, 1000	180	
U1	74LS12	2000, 1350	180	
U2	74LS03	901 222047244094	0	

Figure 17-4. Refining the selection with a particular row (Row 8).

Finally, after we've iterated through each of the components in our collection, on Line 133 we call our *SelectExcelRange()* helper routine to select and highlight all of the cells in our *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 is used to add header text to our *Worksheet* object, which is passed as a parameter into the routine. On Lines 139 through 142 we use the *Range* property associated with the selected *Worksheet* object to add our new header information.

The parameter to the *Range* property is a string containing the column and row information. For example, if we wanted to select the cell located at the intersection of column A and row 4, we would pass the *Range* property a string of "A4". Now observe the statement on Line 139, for example. In this case, we are concatenating two string constants REFDES_COL and

HEADER_ROW. On line 12 we assigned the string "A" to our constant REFDES_COL; similarly, on Line 17 we assigned the string "4" to our constant HEADER_ROW. Thus, on line 139 we assign 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 =
140                      "Ref Des"
140      sheetObj.Range(PARTNAME_COL & HEADER_ROW).Value =
141                      "PartName"
141      sheetObj.Range(LOCATION_COL & HEADER_ROW).Value =
142                      "Location"
142      sheetObj.Range(ORIENTATION_COL & HEADER_ROW).Value =
143                      "Orientation"
```

Finally, on Line 144 we set the *Bold* property of the *Font* property associated with the header row to *True*. This is the reason the header text in the Excel spreadsheets shown in Figures 17-1 through 17-4 is bold.

```
144      sheetObj.Rows(HEADER_ROW).Font.Bold = True
145  End Sub
```

AddComponents() Subroutine

Lines 151 through 162 declare the routine that is used to orchestrate 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 we want 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 we instantiate and initialize an integer that we're going to use to keep track of which row we're on.

```
152      Dim rowInt
153      rowInt = startRowInt
```

On Line 154 we declare a component object, then on Lines 155 through 158 we iterate through our collection of components. For each component in the collection we call our *AddComponent()* utility routine, which will actually perform the nitty-gritty task of adding the component. As parameters into this routine, we pass the *Worksheet* object, the row on which we want to add this component, and the *Component* object we want to add.

```
154      Dim cmpObj
155      For Each cmpObj In cmpColl
156          Call AddComponent(sheetObj, rowInt, cmpObj)
157          rowInt = rowInt + 1
158      Next
```

Finally, on Line 161 we use 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 is used to actually add the component information into the *Worksheet* object. The parameters passed into this routine are the *Worksheet* object, the row on which we want to add this component, and the *Component* object we want to add.

```
164  ' Add a single component to row rowInt
165  ' sheetObj - Excel Worksheet Object
166  ' rowInt - Integer
167  ' cmpObj - Component Object
168  Sub AddComponent(sheetObj, rowInt, cmpObj)
```

Lines 169 through 176 are used to take information from the Component object and assign that information to appropriate cells in the spreadsheet. The only point of interest here is the test on Line 171 to see if the component has been placed. If it has been placed, then on Lines 172 and 173 we load the appropriate cells in the spreadsheet with that component's X/Y location and orientation; otherwise, on Line 175, we simply load the "Location" cell with a string of "Unplaced".

```
169      sheetObj.Range(REFDES_COL & rowInt).Value = cmpObj.Name
170      sheetObj.Range(PARTNAME_COL & rowInt).Value =
171          cmpObj.PartName
172          If cmpObj.Placed Then
173              sheetObj.Range(LOCATION_COL & rowInt).Value =
174                  cmpObj.PositionX & ", " & cmpObj.PositionY
175              sheetObj.Range(ORIENTATION_COL & rowInt).Value =
176                  cmpObj.Orientation
177          Else
178              sheetObj.Range(LOCATION_COL & rowInt).Value =
179                  "Unplaced"
180          End If
181      End Sub
```

Helper Functions and Subroutines

This is where we declare some generic "helper" routines. These routines are not focused on this script per se; they can easily be used in other scripts.

ExcelsRunning() Function

There is a chance that Excel could be closed by the user prior to their attempting to make a selection (you know what little rascals those users can be). If we don't test for this, our script could "crash-and-burn." Generally speaking this type of thing would be addressed by listening for the *Close* event from the Excel application. However, Excel's interface does not allow events to be caught by scripts (we will discuss this in more detail later in this tutorial).

In order to get around this, we are going to use a cunning trick, which is to exercise the Excel interface and trap any errors that occur. If there aren't any errors, then we can be confident that Excel is alive and kicking; otherwise, we will assume that Excel has been closed.

Lines 184 through 208 declare a helper routine that checks to see if Excel is still running. The parameter passed into this routine is the Excel *Application* object [We could have called this routine *IsExcelRunning()*, but the name we did use works better when combined with an *If ... Then ...* statement; for example, *If ExcelIsRunning() Then ...*]

```
182  ' Returns true if Excel is still running false otherwise
183  ' appObj - Excel Application Object
184  Function ExcelIsRunning(appObj)
```

On Line 186 we start off by assuming that Excel is running, so we initialize the return value from this function to *True*.

```
185      ' Initialize return value
186      ExcelIsRunning = True
```

On Line 189 we perform a quick check to see if the variable holding the Excel *Application* object has been set to *Nothing* (see Lines 204 through 207 to discover why this might be the case). If so, On Line 190 we set the return value to *False* and on Line 191 we exit 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
```

On Line 196 we use the *On Error Resume Next* statement to prevent any errors that might occur from aborting our script (see also the *Error Handling* topic in *Chapter 2: VBS Primer*). On Line 197 we clear the *Err* object because we don't want to see 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 has been shut down.
196      On Error Resume Next
197      Err.Clear
```

On Line 200 we declare a dummy object and on Line 201 we make a simple call to the Excel interface that will cause an error if Excel happens to be closed (we chose to play with the *Sheets* property, but almost anything would have sufficed for our purposes here).

```
199      ' Make a call that would cause an exception
200      Dim sheetsObj
201      Set sheetsObj = appObj.Sheets
```

On Line 204 we check to see if an error occurred. If so, we first set the Excel *Application* object to *Nothing* and then we set the return value to *False* and exit the function. (Setting the Excel *Application* object to *Nothing* allows us to perform a "cheap-and-cheerful" test on Line 189 if this routine is called again if the user tries to select something else in the future.)

```
202      ' Check the error value.
203      If Err Then
204          Set appObj = Nothing
205          ExcelIsRunning = False
206      End If
208  End Function
```



Note: Assuming Excel is still active and there is no error, then we want our script to return to the mode where any other errors will cause it to abort. In this case, however, there is no need for us to use the *On Error GoTo 0* statement (this was detailed in the *Error Handling* topic in *Chapter 2: VBS Primer*); 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 is used to select one or more cells on an Excel spreadsheet. The cells to be selected are defined by an Excel *Range* object. The parameters to this routine are the *Worksheet* object the range is in and the *Range* object itself.

```
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)
```

The first thing we do on Line 215 is to call our *RemoveExcelFill()* helper function and pass it the *Worksheet* object. This function will set all of the cells on the specified worksheet to have a background color of white.

```
214      ' Remove the yellow
215      Call RemoveExcelFill(sheetObj)
```

On Line 216 we check to see if we have a valid Range object. If so, on Line 218 we set the *ColorIndex* property of the *Interior* property of the *Range* object to a value of 6, which equates to yellow (see also the notes at the end of this routine). Then, on Line 219 we use the *Select* method to select all of the cells defined by the *Range* object.

Alternatively, if there was no valid range, then on Line 222 we cause an un-selection (that is, we de-select any 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
```



Note: In the routine above, we set the *ColorIndex* property to 6, which equates to yellow. Now, the color values in Excel are undoubtedly documented somewhere on the planet, but the authors of this tutorial couldn't track down this documentation anywhere. The way we got around this was to use the Excel Graphical User Interface (GUI) to open a Worksheet and to set a cell to the desired color (yellow). We then created a small script to access the cell with the desired color and to display the value of its *ColorIndex* property in a *MsgBox()*. It would be very useful for you to perform this task for a range of colors and to then create your own set of enumerates for use in your future scripts. (Tricks like this can also be used with other automation interfaces.)

RemoveExcelFill() Subroutine

We're almost done (hurray)! Lines 228 through 231 declare the routine that is used to clear any cells that currently colored yellow and return them to a pristine white background. The

only point of interest is the -4142 (negative 4142) which equates to white. Do not ask us how this value came to be; we obtained it using the technique discussed in the note above.

```
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 above (including the license server function which is not shown above) and save it out as *ExcelCompList.vbs*.
- 2) Close any instantiations of Expedition PCB that are currently running, and then launch a new instantiation of this application.
- 3) Open the *Candy.pcb* design we've been using throughout these building block examples.
- 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 doesn't fail or crash because of the test we included in the event handler on Line 63.

Enhancing the Script

There are a number of ways in which this script could be enhanced. Some suggestions are as follows (it would be a good idea for you to practice your scripting skills by implementing these examples):

- Depending on the size of your 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 user has no indication as to what is going on during this time. Thus, one useful enhancement would be to display a "Progress Meter" in Expedition PCB's status bar to reflect the status of the Excel load.
- Our script has the highlight color associated with components selected in the Excel spreadsheet hard-coded to be yellow. You might provide the user with the ability to choose the highlight color from a palette presented as a pop-up form/dialog launched from an accelerator key.
- With regard to the previous point, you may want to allow the user to associate different highlight colors with different components (resistors = red, capacitors = blue, chips = green, and so forth).
- Observe the bold header text in the Excel spreadsheets illustrated in Figures 17-1 through 17-4. You could experiment by making this text a slightly bigger font and/or changing its color.
- The *ExcelIsRunning()* helper routine could be modified to inform the user that there is a problem in the case that Excel has been closed when the user attempts to make a selection in the layout design document.
- Our script only supports cross-probing from Expedition PCB to Excel. You could modify the script to perform cross-probing from Excel back to Expedition PCB. This functionality

would have to be implemented as an Excel macro (a large portion of the code shown here would work in an Excel macro). Use of 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.

Chapter 18: Generating a Component CSV File

Introduction

- Overview:** This script reads the component information from an Expedition PCB layout design document and writes this information to a text file in Comma Separated Value (CSV) format. Depending on the platform (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 document 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 We Start

In *Chapter 17* we created a script that passed data directly from the layout design document into a running Excel document. That script is suitable for running only under the Windows® operating system. In order to create a similar script that will work across multiple platforms (Windows, Unix, Linux, etc.) we are going to first output information to a text file in Comma Separated Value (CSV) format. This text file can subsequently be used by any application that supports CSV files on any platform.

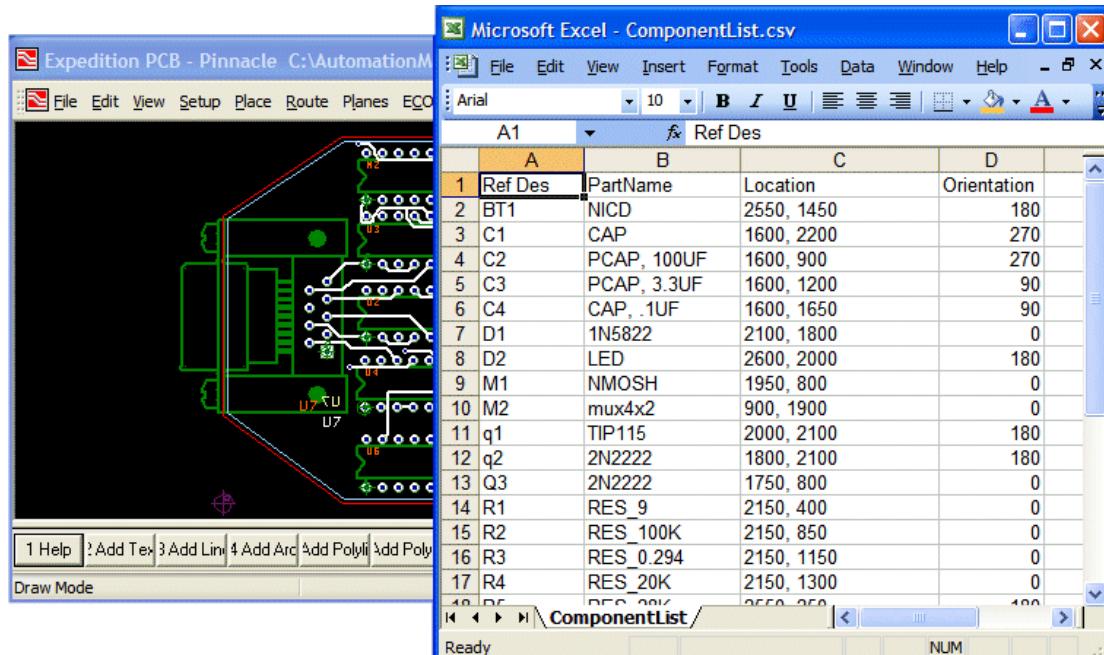


Figure 18-1. Using an appropriate viewer to display the contents of the CSV file.

Once the script has generated the CSV file, it checks to determine the current platform (operating system), launches an appropriate spreadsheet viewer for that platform, and uses this viewer to display the contents of the CSV file.

In order to get a feel for the way in which our script is going to work, let's assume that we've launched Expedition PCB, opened our *Candy.pcb* layout design document, and executed the script. The result will be something like that shown in Figure 18-1.

The Script Itself

The key sections forming this script are as follows:

- **Constant Definitions** Define the constants to be used by the script.
- **Initialization/Setup** Get the *Application* and *Document* objects, license the document, and add the type libraries. Also perform any additional setup associated with this script.
- **Create CSV Subroutine** This is the main subroutine that creates a text file and loads it with component information in CSV format.
- **Utility Functions and Subroutines** Special "workhorse" routines that are focused on this script.
 - WriteHeaders()
 - WriteComponents()
 - WriteComponent()
- **Helper Functions and Subroutines** Generic "helper" routines that can be used in other scripts.
 - ViewCSV()
- **Validate Server** The standard function used to license the document and validate the server. (Note that we won't go through this function in this chapter – for more details on this function see *Chapter 3: Running Your First Script*.)

Constant Definitions

The only constant used by this script occurs on Line 8, where we hard-code the name of the CSV file we want to create and use. As we shall see, this file will be stored in the Log Files folder associated with the layout design document on which the script is run.

```
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  ' Constants
8  Const FILE_NAME = "ComponentList.csv"
```

Initialization/Setup

Lines 11 through 34 are used to perform the initialization and setup tasks. Lines 11 through 24 are used to add the type libraries required by the script, to get the *Application* and *Document* objects, and to license the document by calling the standard *ValidateServer()* function.

```
10 ' Add any type libraries to be used.
11 Scripting.AddTypeLibrary( "MGCPBC.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 we retrieve the name/path of the design document and append it with the name of the *LogFiles* folder and our *ComponentList.csv* file name (which we declared as a constant on Line 8).

```

26  ' Create the csv file name
27  Dim fileNameStr
28  fileNameStr = pcbDocObj.Path & "LogFiles\" & FILE_NAME

```

On Line 31 we call our main *CreateCSV()* subroutine, which will create and populate the CSV file (we pass the file name we just created into this function). Then, on Line 34, we call our *ViewCSV()* helper function, which will display 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 are where we declare the function that will create and populate our text file in CSV format. On Lines 43 and 44 we create a *FileSystemObject* object (see the *File Input/Output (I/O)* topic in *Chapter 12: Basic Building-Block Examples* 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")

```

On Lines 47 through 49 we use the *CreateTextFile* method to create a new text file called *ComponentList.csv*. As part of this process we obtain a *TextStream* object to which we will output our 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)

```

On Line 52 we call our utility subroutine *WriteHeaders()*, which will write our column headers to the output file.

```

51      ' Set the header information
52      Call WriteHeaders(txtStreamObj)

```

On Lines 55 and 56 we acquire a collection containing all of the components in the layout design document. On Line 59, we use the inbuilt *Sort* method to sort the collection. (Note that

this method performs an alpha-numeric sort on the component reference designators. We could create our own *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()
```

On Line 62 we call our utility subroutine *WriteComponents()* to write the component information into our text file. On Lines 65 and 66 we 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 is where we declare some utility routines. These routines are similar in context to our generic "helper" routines (see the next topic), except that they are specific to this script and may not be used elsewhere without some "tweaking".

WriteHeaders() Subroutine

This simple routine accepts a *TextStream* object as a parameter and writes a string to our text file. The string consists of four comma-delimited header titles, each of which will appear at the top of a column of component information. Furthermore, the *Writeline* statement automatically appends a new line character (or characters, depending on the current platform/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. It then cycles through every component in the collection and – for each component – it calls our *WriteComponent()* utility routine.

```
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 we consider this routine, we need to first understand a little something about CSV files. For the purposes of these discussions, let's assume that we want to generate a CSV file that will ultimately be used to populate a spreadsheet containing four columns, and that each of these columns is going to contain the name of a single color. Let's suppose, for example, that our CSV file looked like the following:

Red, Green, Blue, Yellow
Red, Orange, Violet, Blue
Blue, Red, Yellow, Gold
Cyan, White, Red, Black

Note that there are invisible "new line" characters at the end of each line (these are generated when the person creating this file presses the <Enter> key). If we were to load this CSV file into a spreadsheet, it would appear as something like the screenshot in Figure 18-2.

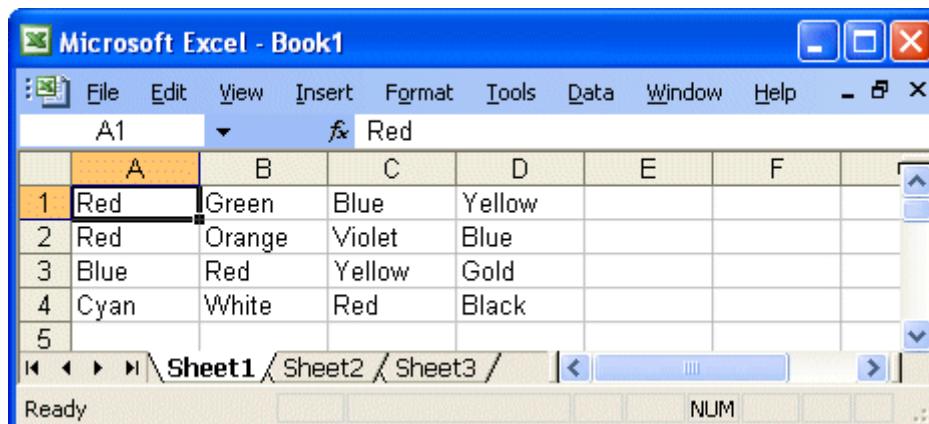


Figure 18-2. A spreadsheet with cells containing individual values.

Now, suppose that we actually wanted to combine the first two columns such that each cell in the first column contains a pair of colors separated by commas. For example, suppose that we wanted our spreadsheet to appear as illustrated in Figure 18-3.

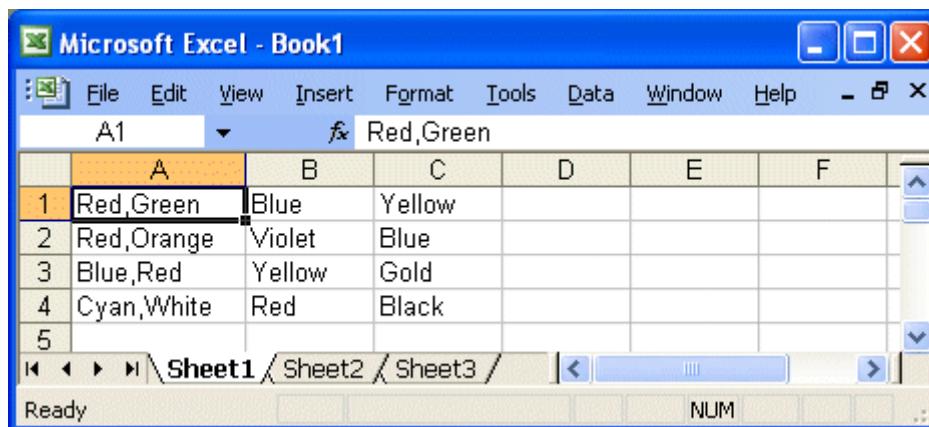


Figure 18-3. A spreadsheet with cells containing comma-separated values.

The question is: how do we present the information in the CSV file such that it can be understood by other applications (like a spreadsheet) to represent the situation illustrated in Figure 18-3. The answer is that, in the case of items containing comma-separated values, we encapsulate these values using double-quote ("") characters. Thus, we would create our CSV file as follows:

"Red, Green", Blue, Yellow
"Red, Orange", Violet, Blue
"Blue, Red", Yellow, Gold
"Cyan, White", Red, Black

The reason this is of interest is that the items forming the first three columns of the CSV file we are generating using our script can each comprise comma-separated values. For example, let's consider the **C2** component in the spreadsheet associated with our *Candy.pcb* design as shown in Figure 18-4.

A	B	C	D	
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
9	M1	NMOSH	1950, 800	0
10	M2	mux4x2	900, 1900	0
11	q1	TIP115	2000, 2100	180
12	q2	2N2222	1800, 2100	180

Figure 18-4. Some cells in our component spreadsheet can contain multiple values.

As fate would have it, the **Ref Des**, **PartName**, and **Location** columns can each contain comma-separated values (only the **PartName**, and **Location** columns do so in this particular example). This means that we need to generate the CSV file that we will use to populate our spreadsheet as illustrated in Figure 18-5.

```
Ref Des,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
```

Figure 18-5. The CSV file format we need to generate.

Keeping this in mind, let's consider our *WriteComponent()* routine (note that we could have embedded this functionality in the *WriteComponents()* routine presented in the previous topic, but breaking things out this way makes everything easier to read and understand).

As we see, this routine accepts a *TextStream* object and a *Component* object as parameters. On Line 91 we output the component's name (its reference designator); on Line 92 we output its part name. On Line 93 we perform a test to see if the component has been placed. If the component has been placed, then on Lines 94 and 95 we output its X/Y location and on Line 96 we output its orientation. Alternatively, if the component has not been placed, then on Line 98 we simply output the string "Unplaced". Finally, on Line 100 we use a *WriteLine* statement with an empty string to append a new line character (or characters, depending on the current platform/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

```

In particular, observe the groups of three and four double-quotes (""" and "") that occur on lines 91, 92, 94, and 95. What do these mean? Well, first consider the case of four double-quotes as illustrated in Figure 18-6(a).

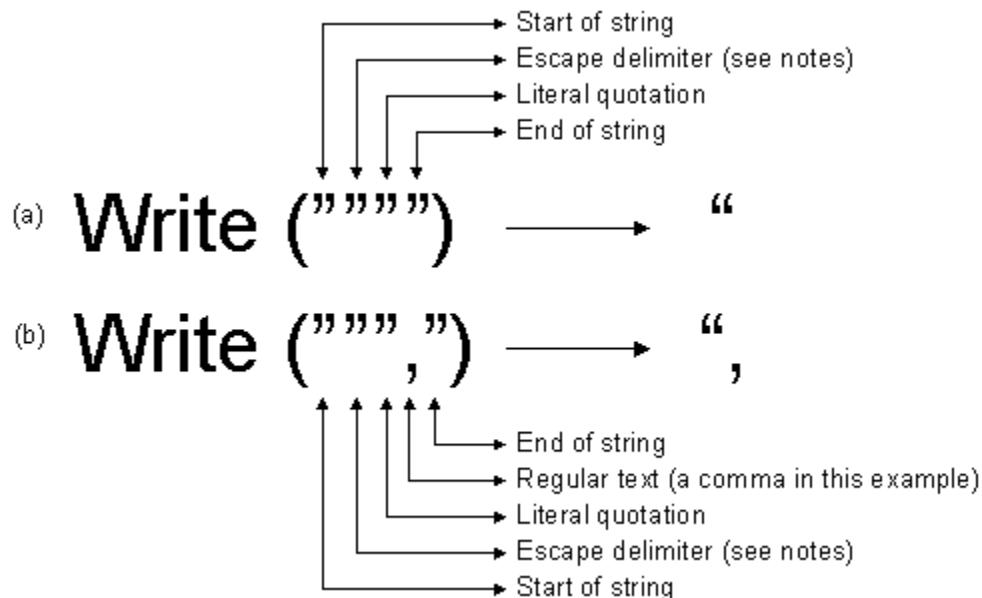


Figure 18-6. Understanding triple and quadruple double-quotes.

As we see, the first quote indicates the start of the string; the second acts as an escape delimiter, which informs the system that the following character is not intended to close the string, but should instead be treated as a literal character that we want to output; the third is the quotation character we want to output; and the fourth closes the string.

Similarly, in the case of Figure 18-6(b), the first quote indicates the start of the string; the second acts as an escape delimiter; and the third is a quotation character we want to output.

We could then include any regular text (in this example we choose to include only a comma character), and finally we have one more double-quote to close the string.

If you now carefully ponder the statements comprising our *WriteComponent()* routine and compare them to the CSV file we generate as illustrated in Figure 18-5, you'll quickly see how this all comes together.

Helper Functions and Subroutines

This is where we declare a generic "helper" routine. This routine is not focused on this script per se; it can easily be used in other scripts.

ViewCSV() Function

Lines 108 through 120 declare the routine that takes the contents of the CSV file we just generated and displays them using whichever spreadsheet application is appropriate for the current platform (operating system).

On Lines 110 and 111 we create a *ViewLogic.Exec* object, which allows us to run system commands (see also the *Running Executables* topic in *Chapter 12: Basic Building-Block Examples*).

```
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, we perform a test to see what platform we are running on. The *Scripting.IsUnix* method will return *True* if the script is running on Unix or Linux. If this is the case, then on Line 115 we use the *Run* method to launch the OpenOffice Calc spreadsheet application (this executable is called *oocalc*) and pass it the name of our CSV file. Alternatively, we assume that we are running under Windows, in which case on Line 118 we use the *Run* method to launch the Excel application (this executable is called *Excel.exe*) and pass it the name of our 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
```



Note: On Windows it is optional whether or not you specify the .exe extension (there is no .exe extension on UNIX or Linux).



Note: This script assumes that the appropriate spreadsheet application executable is loaded on the machine and can be found in the *PATH* variable.

Creating and Testing the Script

- 1) Use the scripting editor of your choice to enter the script described above (including the license server function which is not shown above) and save it out as *ComponentCVS.vbs*.

-
- 2) Close any instantiations of Expedition PCB that are currently running, and then launch a new instantiation of this application.
 - 3) Open the *Candy.pcb* design we've been using throughout these building block examples.
 - 4) Run the *ComponentCVS.vbs* script and observe the spreadsheet appear as illustrated in Figure 18-1.
 - 5) Use a text editor of your choice to display the contents of the *ComponentList.csv* file generated by our script and confirm that it looks like the file shown in Figure 18-5.

Enhancing the Script

There are a number of ways in which this script could be enhanced. Some suggestions are as follows (it would be a good idea for you to practice your scripting skills by implementing these examples):

- Modify the data and format to create a bill of materials (BOM) output in CSV format.
- Add error checking to determine if the desired application wasn't found and – if so – notify the user accordingly.

Chapter 19: Finalizing a Design

Introduction

- Overview:** This script finalizes a layout design document 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 We Start

Once a board design has been completed – prior to sending it to manufacturing – there may be some common documentation items that need to be added. This script demonstrates how to add the board's horizontal and vertical dimensions and a company logo. Consider our familiar *Candy.pcb* design as illustrated in Figure 19-1.

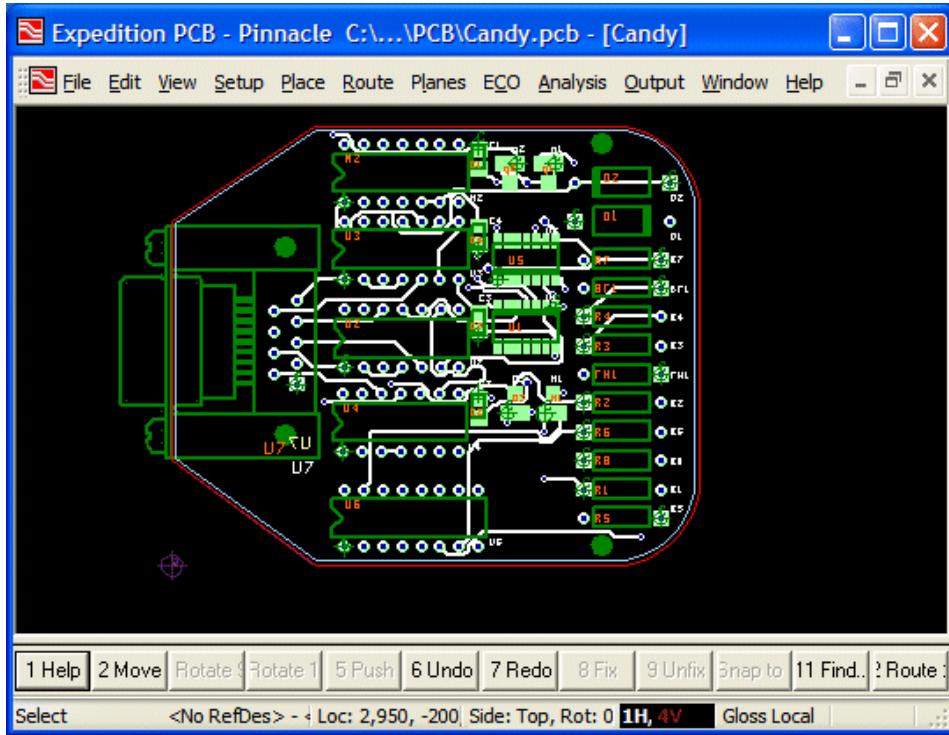


Figure 19-1. Our standard Candy.pcb design.

As a starting point, let's assume that we do not have a script available to us. In this case, we could use the **Place > Dimension Between Two Elements** command to add the horizontal and vertical board dimensions by hand as illustrated in Figure 19-2.

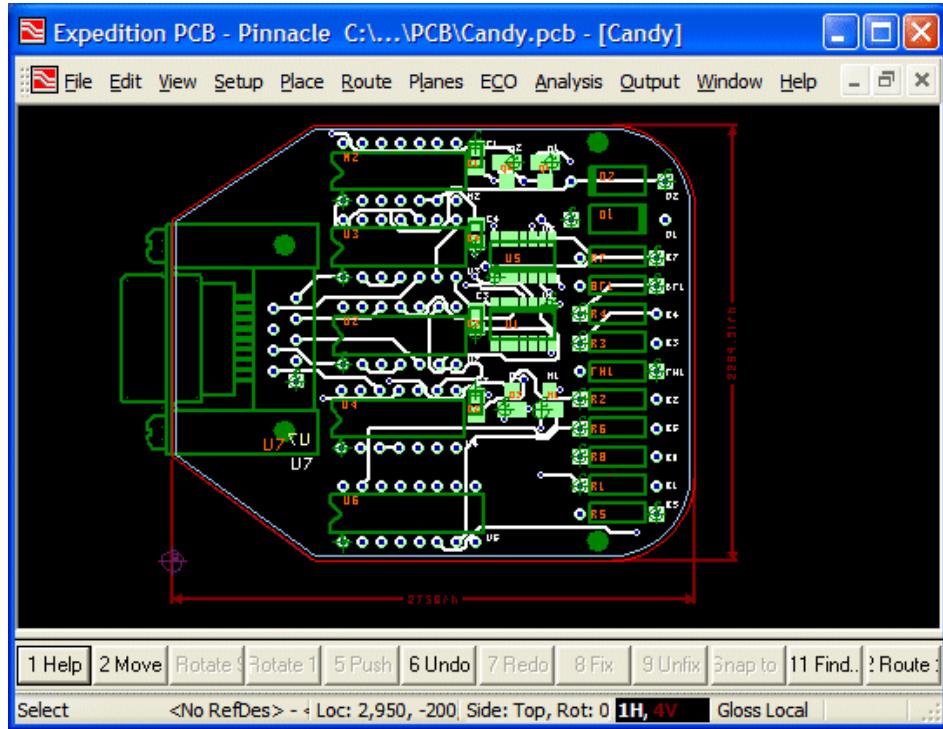


Figure 19-2. Adding the vertical and horizontal board dimensions.

Next, we could use the **Place > Place Parts and Cells** command to select and place a cell containing the company logo as illustrated in Figure 19-3 (note that we have included such a cell – called *Logo* – in the *Candy.pcb* layout design document).

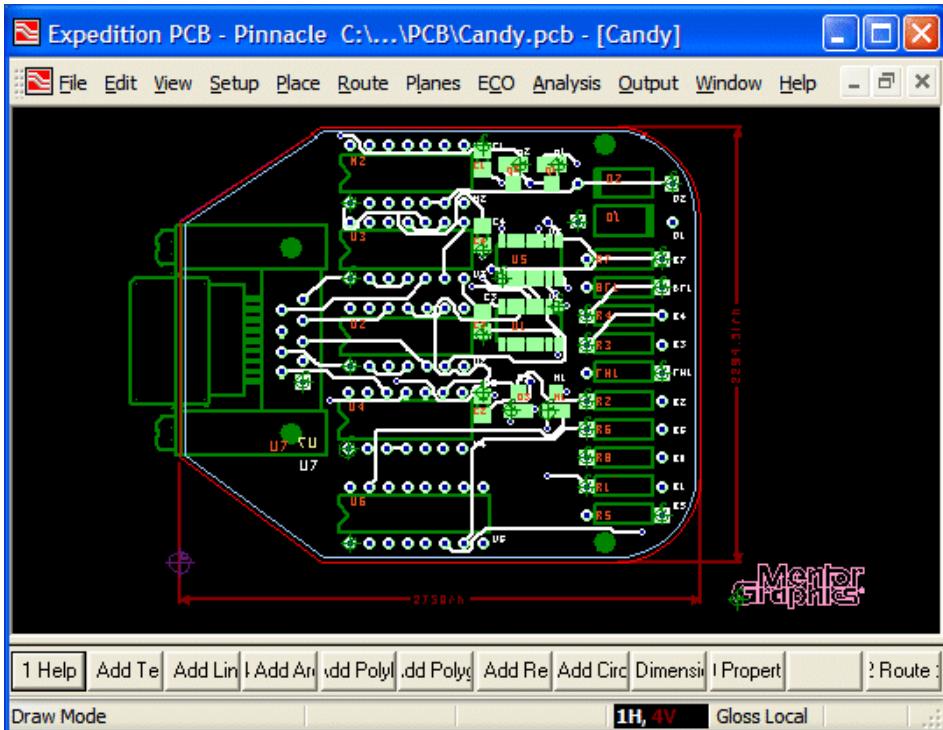


Figure 19-3. Adding the company logo.

Now, as opposed to adding this information by hand, we are going to create a script to do it for us. The main reasons for creating and using such a script are to (a) make sure that all required finishing tasks are executed on every design and (b) to ensure that such tasks are performed consistently on all designs.

The Script Itself

The key sections forming this script are as follows:

- **Constant Definitions** Define the constants to be used by the script.
- **Initialization/Setup** Get the *Application* and *Document* objects, license the document, and add the type libraries. Also perform any additional setup associated with this script.
- **Main Functions and Subroutines** The main functions and subroutines used in the script.
 - AddBoardDimensions()
 - AddCompanyLogo()
- **Utility Functions and Subroutines** Special "workhorse" routines that are focused on this script.
 - AddVertDimExtrema()
 - AddHorizDimExtrema()
 - FindExtremePoint()
 - BetterMatch()
 - BetterValueMatch()
- **Helper Functions and Structures** Generic "helper" routines that can be used in other scripts.
 - CreateUserLayer()
 - Class Point
- **Validate Server** The standard function used to license the document and validate the server. (Note that we won't go through this function in this chapter – for more details on this function see *Chapter 3: Running Your First Script*.)

Constant Definitions

The constant definitions on Lines 11 through 14 are used to instruct the various utility functions as to where to look for things and how to calculate placement locations; for example, should the company logo be placed to the top-and-right of the board, or the bottom-and-left, or ... ?

```
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 '
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
```

(Author: Toby Rimes)

The definitions on Lines 17 and 18 are used in the script for indexing into our points arrays; the purpose of these constants is simply to make the script more readable.

```
16  ' PointsArray index constants
17 Const X_NDX = 0
18 Const Y_NDX = 1
```

On Line 21 we define the name of the user layer upon which we want to add our dimensional graphics and text.

```
20  ' Dimension user layer
21 Const DIMENSION_LAYER_NAME = "Documentation"
```

On Lines 24 through 26 we specify the amounts by which we want to offset the dimensions and logo from the board outline (we could split the single dimension offset on Line 24 into separate vertical and horizontal offsets if we wanted to do so).

```
23  ' Dimension and logo offset value
24 Const DIMENSION_OFFSET = 200
25 Const LOGO_VERT_OFFSET = 200
26 Const LOGO_HORIZ_OFFSET = 400
```

Finally, on Lines 29 and 30, we hard-code the name of the logo cell and the name of the user layer on which we want to display this cell (these names are not obliged to be identical, that's just the way we did it in this script).

```
28  ' Logo cell and userlayer name
29 Const LOGO_CELL_NAME = "Logo"
30 Const LOGO_LAYER_NAME = "Logo"
```

Initialization/Setup

On Lines 33 through 47 we perform the standard initialization tasks of adding type libraries, defining the units to be used throughout the script, acquiring the *Application* and *Document* objects, and licensing the document (validating the server).

```
32  ' Add any type libraries to be 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 to be 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)
```

On Line 50 we set the units associated with the document to match those in our script.

```
49  ' Set the unitsEnum to match our hard coded values.  
50  pcbDocObj.CurrentUnit = unitsEnum
```

On Line 54 we start a transaction, which means that any of the actions we perform from this point to the end 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
```

On Line 57 we call our main *AddBoardDimensions()* subroutine and pass it two parameters: the first specifies where we want to add the horizontal dimensional information (at the bottom of the board in this example); the second specifies where we want to add our vertical dimensional information (to the right of the board in this example).

```
56  ' Add dimensions  
57  Call AddBoardDimensions(LOC_BOTTOM, LOC_RIGHT)
```

Similarly, on Line 60 we call our main *AddCompanyLogo()* subroutine and pass it two parameters that specify where we want the company logo to appear (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, on Line 63 we terminate the transaction we initiated on Line 54.

```
62  ' End the transaction  
63  pcbDocObj.TransactionEnd
```

Add Board Dimensions Subroutine

Lines 71 through 79 are where we declare the main subroutine that orchestrates the adding of the dimensional information to the layout design document. The first parameter to this routine specifies the board edge on which we want to display the horizontal dimension (the permitted options are the top or the bottom). The second parameter specifies the board side on which we want to display the vertical dimension (the permitted options are the left or the right).

On Lines 73 and 74 we get the *BoardOutline* object. On Lines 77 and 78 we call our *AddVertDimExtrema()* and *AddHorizDimExtrema()* subroutines to add the vertical and horizontal dimension graphics and text, respectively.

```
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 are where we declare the main subroutine that orchestrates the adding of the logo to the layout design document. The options for this script (as specified by the two parameters to this function) 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 we want to display the horizontal dimension (the permitted options are the top or the bottom). The second parameter specifies the board side on which we want to display the vertical dimension (the permitted options are the left or the right).

```
84  Sub AddCompanyLogo(vertLocConst, horizLocConst)
```

On Lines 88 and 89 we acquire the *Extrema* object from the Expedition PCB automation interface.

```
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. In the case of our Candy.pcb board, for example, these values are as illustrated in Figure 19-4.

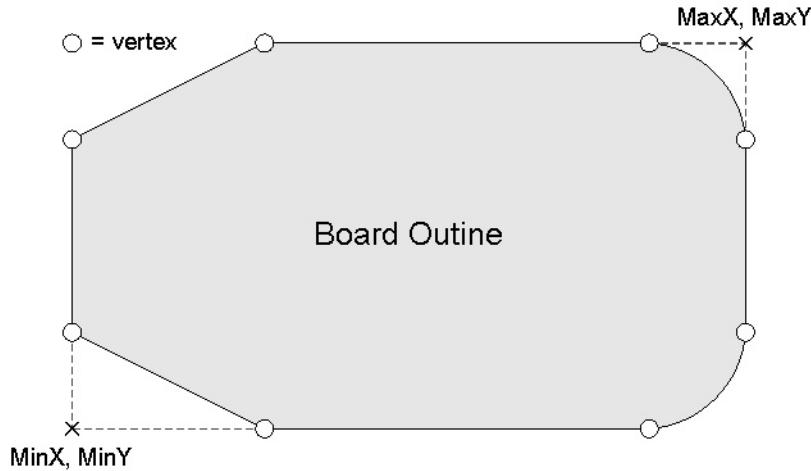


Figure 19-4. The Extrema object's MinX, MaxX, MinY, and MaxY properties.

In Lines 92 through 103 we calculate the horizontal and vertical position for the logo. The calculation of the horizontal value depends on whether the logo is to be placed to the left or right of the board outline; similarly, the calculation of the vertical value depends on whether the logo is to be placed to the top or bottom of the board. The resulting *xPosReal* and *yPosReal* values will be used to define the placement of the cell's origin that contains the logo graphics (the cell's origin has been set to the center of the logo in our 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 in the same way as components, so on Line 106 we specify that the logo's rotation should be zero and on Line 109 we use the *PutComponent* method to place the drawing cell containing our logo. The parameters to the *PutComponent* method are the name of the cell, whether or not we want to pull it from a central library (we say *False*, the *True* option is not currently supported), the X and Y coordinates, and the rotation (we're accepting the default values for any additional parameters not shown here).

```

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, on Line 112 we turn on the display of the appropriate user layer.

```

110
111     ' Ensure the logo user layer is visible.
112     pcbDocObj.ActiveView.DisplayControl.UserLayer(LOGO_LAYER_NAME)
                           = True
113 End Sub

```

Utility Functions and Subroutines

This is where we declare some utility routines. These routines are similar in context to our generic "helper" routines (see the next topic), except that they are specific to this script and may not be used elsewhere without some "tweaking".

AddVertDimExtrema() Subroutine

Lines 122 through 150 declare the routine that is used to add the vertical dimensional information. The first parameter to this routine can be any graphics object (we're passing in the board outline in this script); the second parameter defines whether the vertical dimension graphics and text are to be positioned to the left or right of the board.

```

118     ' Adds dimension marking the vertical extrema 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, we use Line 126 to locate the vertex at the very top of the board and Line 127 to locate the vertex at the very bottom of the board. (Although we 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.)

In order to do this, we use our *FindExtremePoint()* utility function. The first parameter to this function is the graphics object of interest (the board outline in this case); the second parameter specifies the *primary* search condition – whether we are looking for the vertex at the very top of the board or the very bottom of the board; the third parameter specifies the *secondary* search condition – whether we are interested in the left- or right-hand sides of the board.

```
123      ' Find the max and min points in the
124      ' gfxObj points array
125      Dim maxPointCls, minPointCls
126      Set maxPointCls = FindExtremaPoint(gfxObj,
127                                         LOC_TOP, locConst)
127      Set minPointCls = FindExtremaPoint(gfxObj,
128                                         LOC_BOTTOM, locConst)
```

Why do we need to use these two search conditions? Well, consider the board outline illustrated in Figure 19-4. In this case there are two vertices on the top of the board and two on the bottom. If our primary search criterion is to locate the vertex on the top of the board, then – in the case that there are multiple vertices on the top of the board – our secondary search criterion will act as a "tie-breaker".

Note that the *FindExtremePoint()* utility function is also used by the *AddHorizDimExtrema()* subroutine as discussed below. 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; by comparison, if the primary search parameter is left or right, then the secondary search parameter must be top or bottom.

On Line 131 we again acquire the *Extrema* object from the Expedition PCB automation interface.

```
129      ' Get the extrema of the board to guide the
130      ' placement of the dimension
131      Dim extremaObj: Set extremaObj = gfxObj.Extrema
```

On Lines 134 through 139 we use the *MinX* and *MaxX* properties associated with the *Extreme* object – in conjunction with the horizontal value of the board outline vertex in which we are interested – to determine the desired offset for the vertical dimensional information. This calculation varies depending on whether we intend to place the dimensional information to the left or the right of the board outline. If we assume the right-hand side, then the result will be as illustrated in Figure 19-5.

```
128
132
133      ' Calculate the desired offset
134      Dim offsetReal
135      If locConst = LOC_RIGHT Then
136          offsetReal = extremaObj.MaxX - maxPointCls.X +
137                           DIMENSION_OFFSET
137      Else
138          offsetReal = extremaObj.MinX - maxPointCls.X -
139                           DIMENSION_OFFSET
139      End If
```

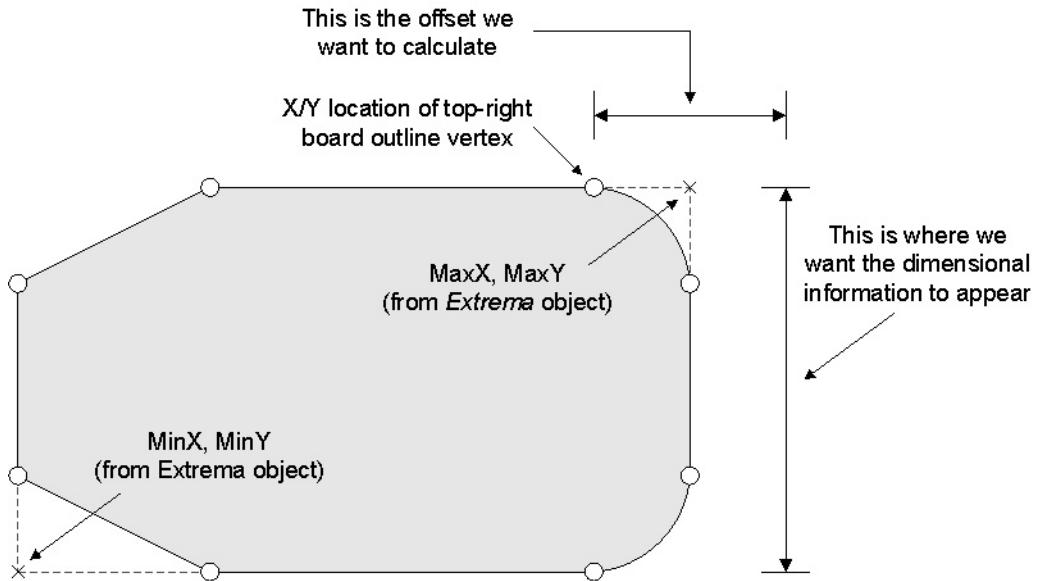


Figure 19-5. Calculating the horizontal offset for the vertical dimension (assuming placement on the right-hand side of the board).

On Lines 142 and 143 we create a new Layer object and assign our dimension user layer to it. Then, on line 144 we call our *CreateUserLayer()* helper function; if the required user layer does not yet exist this function will create it, otherwise (if the layer does exist) this function will simply return 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, on Lines 147 through 150, we use the *PutPointToPointDimension* method to add the vertical dimension to our layout design document. This method places a point to point dimension between two objects (in this case we're 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 we want to apply a horizontal or vertical dimension, the layer object upon which we want the dimension to be placed, and the scheme to be used (this defines the line style, arrow style, format of the font, etc. – we pass *Nothing* to indicate that we want to use 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 is used to add the horizontal dimensional information to the layout design object. The way in which this routine works is the counterpoint to the *AddVertDimExtrema()* routine we just discussed, so understanding the nuances of this routine will be left as an exercise for the reader.

```

153  ' Adds dimension marking the horizontal extrema of gfxObj and
154  ' places the dimension on the locConst side of the object.
155  ' gfxObj - ConductorLayerGfx, UserLayerGfx, or
156  ' FabricationLayerGfx Object
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,
162                                         LOC_RIGHT, locConst)
162     Set minPointCls = FindExtremaPoint(gfxObj,
163                                         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 +
172                         DIMENSION_OFFSET) * -1
172     Else
173         offsetReal = (extremaObj.MinY - maxPointCls.Y -
174                         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 routine that is used to find the extreme vertex in a graphical object's points array. The three parameters to this routine are the graphical object in question (this is the board outline in our example), the primary search criterion (the vertex we're looking for to be located on the top, bottom, left, or right edges of the board), and the secondary search criterion (the vertex we're looking for to be located on the top, bottom, left, or right edges of the board).

```

188  ' Returns extrema point of a vertex in gfxObj's points array.
189  ' The extrema direction is defined by primaryLocConst and
190  ' secondaryLocConst
191  ' gfxObj - ConductorLayerGfx, UserLayerGfx, or
192  ' FabricationLayerGfx Object
192  ' primaryLocConst - Constant (LOC_TOP, LOC_BOTTOM,
193  ' LOC_LEFT, LOC_RIGHT)

```

```

193   ' secondaryLocConst - Constant (LOC_TOP, LOC_BOTTOM,
194     LOC_LEFT, LOC_RIGHT)
194 Function FindExtremaPoint(gfxObj, primaryLocConst,
                           secondaryLocConst)

```

As we previously discussed, the reason for having both primary and secondary search criteria is that there may be multiple vertices on the various edges forming the board. Thus, if our primary search criterion were to look for the vertex on the top of the board and several vertices had the same maximum Y value, then the secondary search criterion would be used as a "tie-breaker" to look for the right- or left-most candidate. This means that if the primary search criterion is top or bottom, then the secondary search criterion must be left or right; by comparison, if the primary search criterion is left or right, then the secondary search criterion must be top or bottom.

On Line 196 we acquire 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

```

In all of our previous scripts, we've used pre-defined objects associated with various automation interfaces. In turn, these objects have had pre-defined properties and methods associated with them. In fact, VBScript allows us to create what may be considered to be our own custom-built objects, each of which can contain their own properties and methods. This is achieved by means of a *Class* structure.

Later in this script, we define a really simple *Class* structure called *Point*, where "Point" is an arbitrary name we decided to use (see the discussions associated with Lines 318 through 321). All we need to know at this stage is that our *Point* class has two properties associated with it: *X* and *Y*.

On Line 200, we define two variables: *bestPointCls* (which will be used to hold the current best candidate vertex) and *testPointCls* (which will be used to hold the next vertex to be compared-with / tested-against the current best candidate).

```

198      ' Create a point class and initialize it
199      ' to the first point in the array.
200      Dim bestPointCls, testPointCls

```

On Line 201 we use a *New* statement to create a new *Point* object and we assign this object to our *bestPointCls* variable. On Lines 202 and 203 we initialize the *X* and *Y* properties of the object contained in our variable to point to the first vertex in our board outline points array (*X_NDX* and *Y_NDX* are the constants we 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)

```

On Lines 205 through 216 we iterate through all of the vertices in the points array starting with the second vertex (we're assuming that the first vertex is currently our best match).

```

204      Dim i
205      For i = 1 To UBound(pntsArr, 2)

```

On Line 207 we use a *New* statement to create a new *Point* object and we assign this object to our *testPointCls* variable. On Lines 208 and 209 we initialize the X and Y properties of the object contained in our variable to point to the second vertex in our 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 we call our *BetterMatch()* utility function to determine if the new (test) vertex is a better match than our current best candidate. This determination is based on the primary and secondary search criteria that were passed into the main function. If this function returns *True* – thereby indicating that the test vertex is a better match than the current best candidate – then on Line 214 we copy the test *Point* object into our current best candidate *Point* object.

```
211          ' See 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, on Line 218, we return the best *Point* object that we found.

```
210
211
212
213
214
215
216
217
218      Set FindExtremaPoint = bestPointCls
219
220  End Function
```

BetterMatch() Function

Lines 229 through 272 declare the routine that is used to determine whether the test point (the second parameter passed into the routine) is a better match than the current best candidate (the first parameter). The routine bases this determination on the primary and secondary search criteria, which form 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,
228  '                               LOC_LEFT, LOC_RIGHT)
228  ' secondaryLocConst - Constant (LOC_TOP, LOC_BOTTOM,
229  '                               LOC_LEFT, LOC_RIGHT)
229  Function BetterMatch(bestPointCls, testPointCls, _
230                      primaryLocConst, secondaryLocConst)
```

On Line 232 we initialize this function's return value to *False*, thereby saying that we are assuming that the new value won't be a better match.

```
231      ' Initialize return value
232      BetterMatch = False
```

On Lines 234 through 271 we perform a *CASE* statement based on the primary search criterion. For each variant of the primary search criterion (top, bottom, left, right) we first check to see if the test point is "better" than the current best candidate (in each case the definition of "better" depends on the primary search criterion).

If the test point is "better", we simply set this function's return value to *True*. Otherwise, we perform a second test to determine if the test point is "equal" to the current best candidate in terms of the primary search criterion. If this is the case, we call our *BetterValueMatch()* utility function to act as a "tie-breaker".

```
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 =
242                     BetterValueMatch(bestPointCls.X, _
243                                     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 =
251                     BetterValueMatch(bestPointCls.X, _
252                                     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 =
260                     BetterValueMatch(bestPointCls.Y, _
261                                     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 =
269                     BetterValueMatch(bestPointCls.Y, _
270                                     testPointCls.Y, secondaryLocConst)
270             End If
271     End Select
272 End Function
```

BetterValueMatch() Function

Lines 279 through 292 declare the routine that is used to act as a "tie-breaker" for the previous function. In this case, it accepts as parameters the current best candidate, the test candidate, and the secondary search criterion.

```
274  ' Returns true if bestValReal is a better match than
275  ' testValReal according to the locConst criteria. Returns
276  ' false otherwise.
276  ' bestValReal - Real
277  ' testValReal - Real
278  ' locConst - Constant (LOC_TOP, LOC_BOTTOM, LOC_LEFT,
278           LOC_RIGHT)
279  Function BetterValueMatch(bestValReal, testValReal, locConst)
```

On Line 281 we initialize this function's return value to *False*, thereby saying that we are assuming that the new value won't be a better match.

```
280      ' Initialize return value to false
281      BetterValueMatch = False
```

Lines 283 through 285 say that if we're interested in either the top edge or the right side of the board – and if the test value is *greater* than the current best candidate value – then the test is better than the best and we want to return *True*. Alternatively, Lines 287 through 289 say that if we're interested in either the bottom edge or the left side of the board – and if the test value is *less* than the current best candidate value – then the test is better than the best and we want to return *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 is where we declare some generic "helper" routines. These routines are not focused on this script per se; they can easily be used in other scripts.

CreateUserLayer() Function

Lines 299 through 313 declare the routine that is used to create and return a new user layer if necessary, or to simply return the layer if it already exists. The routine accepts a single parameter, which specifies the name of the user layer in question.

```
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 we declare a variable called *usrLyrObj*. On Line 302 we use the *FindUserLayer* method to try to locate the user layer in which we are interested and to assign it to our variable (if the user layer does not exist, the variable will be assigned a value of *Nothing*).

```
300      ' See if our user layer already exists.  
301      Dim usrLyrObj  
302      Set usrLyrObj = pcbDocObj.FindUserLayer(userLayerNameStr)
```

On Line 304 we test to see if our variable contains *Nothing*. If this is the case, then we know the user layer does not exist, so on Line 306 we create it.

```
304      If usrLyrObj Is Nothing Then  
305          ' It doesn't exist. Create it.  
306          Set usrLyrObj =  
307              pcbDocObj.SetupParameter.PutUserLayer(userLayerNameStr)  
307      End If
```

In order to make sure that the user layer is visible, on Line 310 we turn on the display of the user layer. Finally, on Line 312 we set the return value from this function to be the user layer object in question.

```
303  
308  
309      ' ensure the user layer is turned on  
310      pcbDocObj.ActiveView.DisplayControl.  
310          UserLayer(userLayerNameStr) = True  
311  
312      Set CreateUserLayer = usrLyrObj  
313  End Function
```

Class Point Structure

As we noted earlier, VBScript allows us to create what may be considered to be our own 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 we define a really simple *Class* structure called *Point* (where "Point" is an arbitrary name). On Lines 219 and 320 we define two properties – X and Y – that will be associated with our *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 that falls between the *Class* and *End Class* statements – whether it be variables, subroutines or functions – is a member (part) of that class and can be accessed and modified by means of the *Variable-Name.Property* or *Variable-Name.Method* syntax.



Note: Any script can be written without the use of *Class* structures, but when used properly they can benefit your scripts with respect to organization, readability, and reusability.

Creating and Testing the Script

- 1) Use the scripting editor of your choice to enter the script described above (including the license server function which is not shown above) and save it out as *FinalizeDesign.vbs*.
- 2) Close any instantiations of Expedition PCB that are currently running, and then launch a new instantiation of this application.
- 3) Open the *Candy.pcb* design we've been using throughout these building block examples.
- 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 in which this script could be enhanced. Some suggestions are as follows (it would be a good idea for you to practice your scripting skills by implementing these examples):

- 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.

Chapter 20: Route by Layer

Introduction

- Overview:** This script guides the auto-router by specifying that certain nets will be routed only on certain layers.
- Approach:** To use the automation interface 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 We Start

Before we leap into the fray, it should be noted that this is a somewhat "theoretical" script, not the least that the example board we are using as an example (our *Candy.pcb* design) is so simple that our script really doesn't make things any better ... or any worse. In order to see how this script will work, let's suppose that we've launched Expedition PCB, opened our *Candy.pcb* layout design document, and used the **Edit > Delete All Traces and Vias** command. The result will be as shown in Figure 20-1.

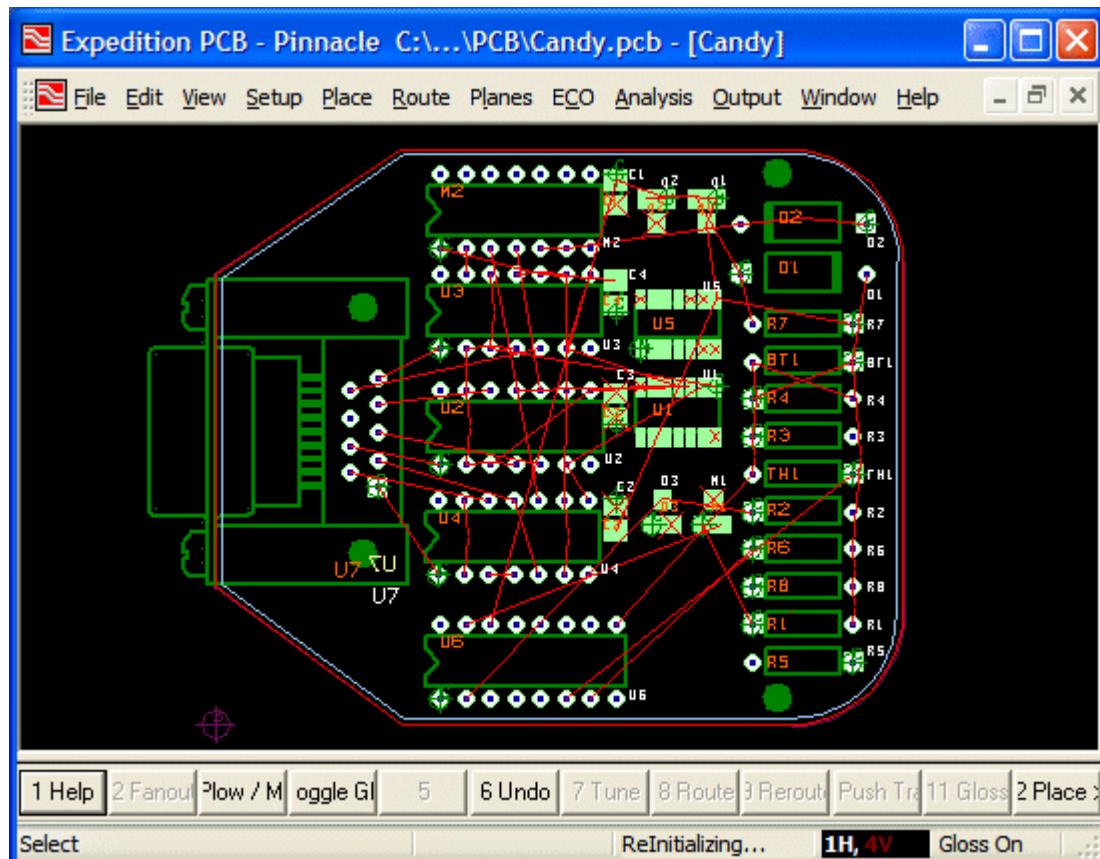


Figure 20-1. The design following an "Edit > Delete All Traces and Vias" command.

Now, although the auto-router in Expedition PCB is widely considered to be the best in the industry, there are certain cases where human observation of a board's characteristics allows certain nets to be routed in a more advantageous way based on these observations. Thus, depending on the "type" of board (as determined by human observation), we can use the automation interface to make calculations, perform evaluations, and to control the auto-router accordingly.

The script we are about to create will route the entire board. It will evaluate the From-To points of each netline and determine if a net has a strong horizontal or vertical bias. Based on these evaluations, the script will instruct the auto-router to route horizontally-biased nets on the signal layers on the top half of the board; vertically-biased nets will be routed using the signal layers on the bottom half of the board.

Assuming that we have created our script and run it on the *Candy.pcb* design, the results will be as illustrated in Figure 20-2. Observe that the signal layers on the top half of the board contain the horizontally-biased nets (white); also that vertically-based nets (red) are shown on the signal layers on the bottom half of the board.

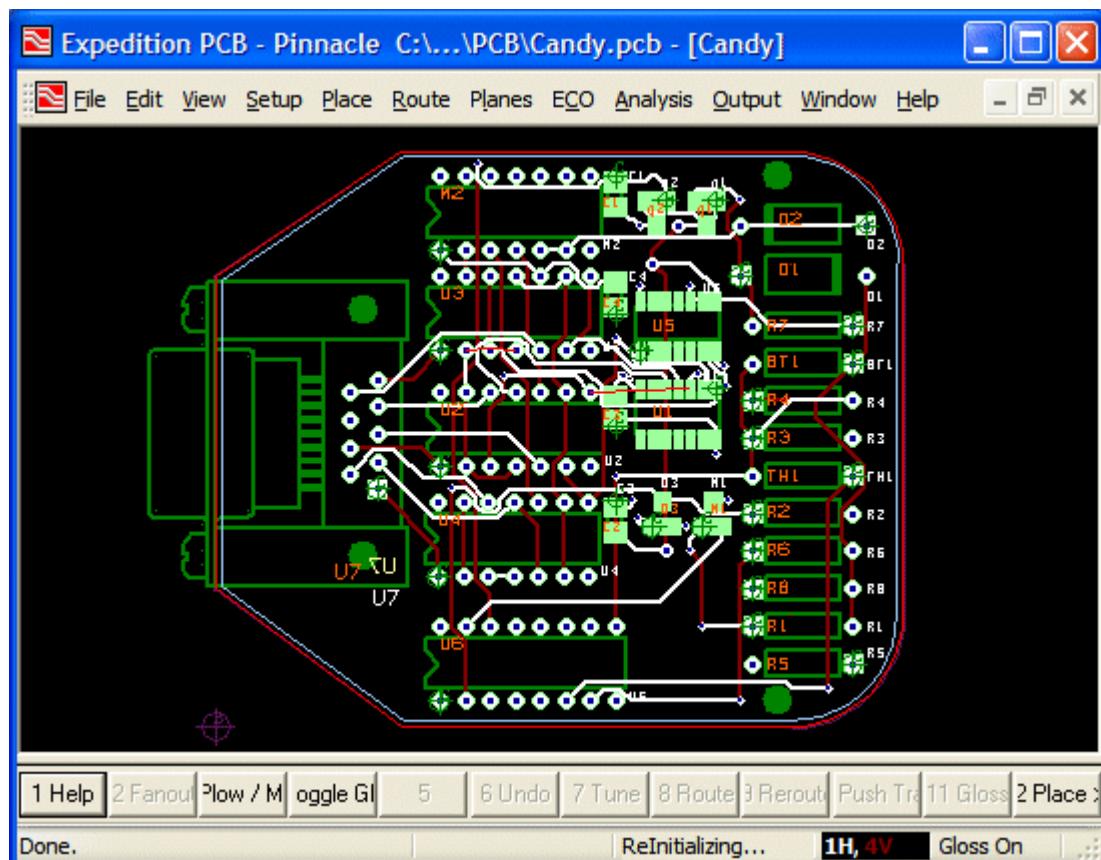


Figure 20-2. The design following the running of our script.

The Script Itself

The key sections forming this script are as follows:

- **Constant Definitions** Define the constants to be used by the script.

- **Initialization/Setup** Get the *Application* and *Document* objects, license the document, and add the type libraries. Also perform any additional setup associated with this script.
- **Main Functions and Subroutines** The main functions and subroutines used in the script.
 - RunFanoutPass()
 - RunAngleBiasRoutePass()
- **Utility Functions and Subroutines** Special "workhorse" routines that are focused on this script.
 - RunRoutePass()
 - FindNets()
- **Helper Functions and Subroutines** Generic "helper" routines that can be used in other scripts.
 - GetSignalLayer()
 - GetSignalLayerCount()
 - GetAngleOfLine()
 - RadiansToDegrees()
- **Validate Server** The standard function used to license the document and validate the server. (Note that we won't go through this function in this chapter – for more details on this function see *Chapter 3: Running Your First Script*.)

Constant Definitions

The only constant used by this script is PI. We don't know where the actual value assigned to this constant originated (we know it's the fundamental constant PI, we mean the specific number of digits specified here), but this is the one that is commonly used in this type of script.

```

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 '
6  Option Explicit
7
8  ' Constants
9  Const PI = 3.1415926535897932384626433832795

```

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( "MGCPBCB.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 we call our main *RunFanoutPass()* subroutine. Immediately afterwards, on Line 21, we run our *RunAngleBiasRoutePass()* subroutine (in fact, this routine actually performs two passes; it first looks for vertically-biased nets and routes these on the bottom signal layers, then it routes the remaining horizontally-biased nets on the top signal layers). Finally – on Line 24 – we display a message on the status bar to inform the user that our script has completed its tasks.

```
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 we are done.
34  Call pcbAppObj.Gui.StatusBarText("Done.")
```

Run Fanout Route Pass Subroutine

Lines 43 through 56 are where we declare the subroutine that performs the initial fanout route pass. On Lines 45 and 46 we use the *NewRoutePass* method to acquire a *RoutePass* object; we call our variable *fanoutPassObj* because we intend to perform a fanout pass.

```
42  ' Run a fanout route pass
43  Sub RunFanoutPass()
44    ' Get a new RoutePass object
45    Dim fanoutPassObj
46    Set fanoutPassObj = pcbDocObj.NewRoutePass
```

On Line 49 we set the *PassType* method. The first parameter is an enumerate from our type library that specifies that we want 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 we are not using a via grid or a route grid, respectively.

```
48      ' Set as fanout pass
49      Call fanoutPassObj.PassType(epcbARFanoutPass, 1, 1,
                                         False, False)
```

On Line 51 we use the *Items* method to specify that we are interested in all of the nets in the design; the first parameter is an enumerate that specifies all nets, which requires that the second parameter is set to *Nothing* (as we shall see, using other enumerates for the first parameter requires different values on the second).

```
50      ' Route all nets
51      Call fanoutPassObj.Items(epcbARAllNetsItem, Nothing)
```

On Line 53 we use the *StatusBarText* method on the *Gui* object to display a message on the status bar to inform the user as to what is happening.

```
52      ' Let the user know what is happening
53      Call pcbAppObj.Gui.StatusBarText("Running fanout pass")
```

Finally, on Line 55, we use 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 are where we declare the subroutine that performs two angle-bias-based route passes. First we need to determine the board's signal layers, so on Lines 63 and 64 we acquire the boards layer stack (the *False* parameter on Line 64 indicates that we don't want any 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)
```

On Lines 67 and 68 we create an empty *LayerObject* object collection.

```
66     ' Get an empty LayerObject collection
67     Dim signalLayerColl
68     Set signalLayerColl = pcbAppObj.Utility.NewLayerObjects
```

On Line 72 we declare a variable to control a look, and then on Lines 73 through 75 we iterate through all of the signal layers in reverse order (the reason for iterating in reverse order will be discussed later in this chapter).

Observe the use of our *GetSignalLayerCount()* helper function, which returns the number of signal layers. On Line 73 we use this function to define the scope of the iteration from the bottom signal layer (the one with the highest number) to the halfway point. Meanwhile, on Line 74 we get the *ith* signal layer and add it to our collection.

```
70     ' Add the signal layers from the bottom
71     ' half of the board.
72     Dim i
73     For i = GetSignalLayerCount(layerStackColl) To
74         Int(GetSignalLayerCount(layerStackColl) / 2) + 1 Step -1
75         signalLayerColl.Add(GetSignalLayer(layerStackColl, i))
    Next
```

On Line 78 we call our *RunRoutePass()* utility subroutine, which will cause 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 (see notes below). The third parameter is our 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 the use of 45 and 90 as parameters to specify *From* and *To* angles of 45° and 90°. We are using these values to specify that we are interested in vertically-biased nets; however, when you come to think about it, our values covers only a small range of the available possibilities as illustrated in Figure 20-3.

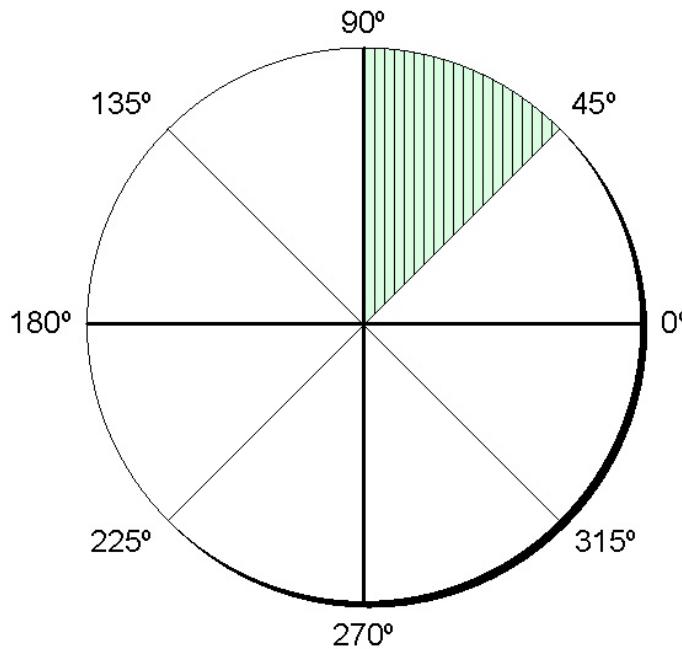


Figure 20-3. Nets theoretically defined by From/To angles of 45° and 90°.

As we shall see, however, our *RunRoutePass()* utility subroutine calls our *FindNets()* utility function, which – in turn – calls our *GetAngleOfLine()* helper function. This helper function effectively "reflects" corresponding areas in the other three quadrants back into the first area as illustrated in Figure 20-4.

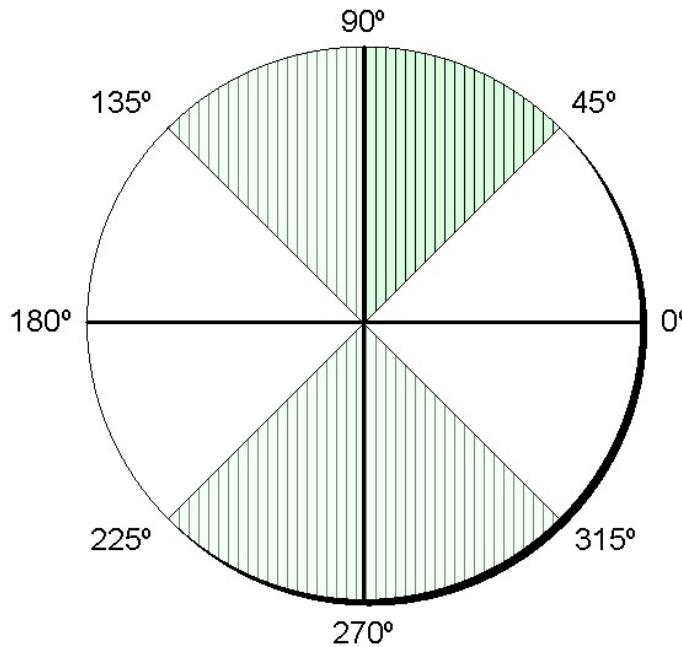


Figure 20-4. Nets actually affected by our From/To angles of 45° and 90°.

Do you remember right at the start of this chapter when we described this script as being "*somewhat theoretical*"? Well, *theoretically* we could indeed *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 we've routed the vertical nets on one set of layers, we may just as well give the auto-router the freedom to route the remaining nets using any of the signal layers. Thus, on Lines 81 through 83 we *add* the signal layers from the top half of the board *into* our existing collection.

```
80      ' Add the remaining signal layers.  
81      For i = Int(GetSignalLayerCount(layerStackColl) / 2) To  
82          1 Step -1  
83          signalLayerColl.Add(GetSignalLayer(layerStackColl, i))  
84      Next
```

Finally, on Line 87, we call our *RunRoutePass()* utility subroutine again, but this time we specify *From* and *To* angles of 0° and 90°, which actually defines an entire quadrant. Now, remembering that (as per our discussions relating to Figures 20-3 and 20-4 above) the other three quadrants will be reflected back into the first, we've effectively specified all of the nets in the design. In this case, however, this is OK, because we've already routed all of the vertically-biased nets, which means that only the horizontally-biased nets remain to be routed.

```
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 is where we declare some utility routines. These routines are similar in context to our generic "helper" routines (see the next topic), except that they are specific to this script and may not be used elsewhere without some "tweaking".

RunRoutePass() Subroutine

Lines 99 through 126 declare the routine that is used to actually run 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)
```

We commence on Lines 101 and 102 by acquiring a new *RoutePass* object.

```
100     ' Get a new RoutePass object  
101     Dim routePassObj  
102     Set routePassObj = pcbDocObj.NewRoutePass
```

On Line 106 we declare a variable in which to hold a collection of nets, then on Line 107 we call our *FindNets()* utility function, which accepts our *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)
```

On Line 110 we set the *PassType* method. As before, the first parameter is an enumerate from our type library; in this case we're using the enumerate that specifies that we want 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 we are not using a via grid or a route grid, respectively.

```
109      ' Set the pass type
110      Call routePassObj.PassType(epcbARRoutePass, 1, 1,
                                 False, False)
```

On Line 112 we use the *Items* method. In this case, the first parameter is an enumerate that specifies that we want to work with a specific collection of nets, while the second parameter is the collection of nets itself (compare this with our previous use of this method on Line 51).

```
111      ' Add the net collection
112      Call routePassObj.Items(epcbARNetsItem, netColl)
```

By default, a route pass is set to route on all layers. Thus, on Line 114 we remove all of the layers, then on Lines 117 through 121 we iterate through our collection of signal layers adding each layer using the *LayerSelect* property on Line 120 (see note below).

```
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
```



Note: This is where things get a little tricky. Observe that the *LayerSelect* property used on Line 120 accepts two parameters. The first is an enumerate, which – in this case – is being used to inform the system that we want to *add* a layer. Meanwhile, 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 will be considered to be layer 0 by the *RoutePass* object; layer 2 in the real world will be considered to be layer 1 by the *RoutePass* object; and so forth.

So here's the problem. As fate would have it, 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 us to select layer 1 in the real world. Fortunately, this doesn't impact our script due to the way in which we iterated through the signal layers in reverse order, first on Lines 73 through 75 and later on Lines 81 through 83. Also, recall that once we've routed the vertical nets on one set of layers, our script gives the auto-router the freedom to route the remaining nets using any of the signal layers. In order to understand how this works, assume that we are 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 }
```

On Lines 73 through 75 we add signal layers 6 and 5 into our collection, and then on Line 78 we call our *RunRoutePass()* utility routine for the *first* time. In this case, the loop on Lines 117 through 121 will first add layer 6 (as layer 5 from the perspective of the *RoutePass* object and its *LayerSelect* method on Line 120) and then add layer 5 (as layer 4).

Later, on Lines 81 through 83, we add signal layers 2 and 1 to the existing collection, and then on Line 87 we call our *RunRoutePass()* utility routine for the *second* time. This time, the loop on Lines 117 through 121 will first add layer 6 (as layer 5), then layer 5 (as layer 4), then layer 2 (as layer 1), and finally layer 1 (as layer 0). As we discussed above, the use of layer 0 in this final iteration will be understood by the *LayerSelect* method to mean "all layers", but that's OK because that's what we want 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 will have no effect.)

On Line 123 we use the *StatusBarText* method on the *Gui* object to display a message on the status bar to inform the 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, we use the *Go* method to start this route pass.

```
124      ' Start the route pass.
125      Call routePassObj.Go()
126  End Sub
```

FindNets() Function

Lines 132 through 164 declare the routine that is used to find the nets that fall between specified *From* and *To* angles, which are passed in as parameters.

```
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)
```

On Lines 133 and 134 we create a *Dictionary* object (a temporary mapping that will allow us to ensure that we don't add the same net more than once).

```
133      Dim netDictObj
134      Set netDictObj = Createobject("Scripting.Dictionary")
```

On Lines 137 and 138 we get a collection of *FromTos* that embraces all of the netlines on the board.

```
136      ' Get the collection of all netlines
137      Dim fromToColl
138      Set fromToColl = pcbDocObj.FromTos
```

On Lines 141 through 151 we iterate through the collection of *FromTos*. On Line 143 we call our *GetAngleOfLine()* helper function to determine the angle associated with the current *FromTo*. On Line 145 we check to see if this angle matches the specified criteria and – if so – on Line 147 we check that this net is not already in our *Dictionary* object, in which case on Line 148 we add 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,
144                                      fromToObj.FromY, fromToObj.ToX, fromToObj.ToY)
145          ' Determine if the angle is in range
146          If (angleReal >= fromAngleReal) And
147              (angleReal <= toAngleReal) Then
148                  ' Only add it if it is not present.
149                  If Not netDictObj.Exists(fromToObj.Net.Name) Then
150                      Call netDictObj.Add(fromToObj.Net.Name,
151                                         fromToObj.Net)
152                  End If
153      End If
154  Next

```

On Lines 154 and 155 we 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

```

On Lines 158 through 160 we iterate through the contents of our *Dictionary* object and we add all of the nets in this object into our net collection (see note below).

```

157      Dim keyStr
158      For Each keyStr In netDictObj.Keys
159          netColl.Add(netDictObj.Item(keyStr))
160  Next

```



Note: Remember that *Dictionary* objects use a key-value mapping structure. Thus, in order to iterate through all of the items in the dictionary, we actually iterate through the collection of *Keys*. For each of these *Keys* we use the *Item* property to acquire the value (the *Net* object in this case) associated with that *Key*.

Finally, on Line 163, we return the collection of nets.

```

162      ' Return the collection of nets
163      Set FindNets = netColl
164  End Function

```

Helper Functions and Subroutines

This is where we declare some generic "helper" routines. These routines are not focused on this script per se; they can easily be used in other scripts.

GetSignalLayer() Function

Lines 174 through 190 declare the routine that is used to take a collection of layers (this collection can include insulating, plane, and signal layers) and return the n^{th} signal layer. For example, assume that we are working with 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, while the second is the number of the signal layer (the n^{th} signal layer) in which we are interested; for example:

Value of second parameter	Which equated 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 we initialize the return value from this function to be *Nothing*.

```
175      Set GetSignalLayer = Nothing
```

On Line 176 we instantiate a variable in which to hold a *Layer* object. On Line 177 we instantiate an integer variable and initialize it with 0 (this variable will be used to keep track of how many signal layers we've found).

```
176      Dim layerObj
177      Dim numFoundInt : numFoundInt = 0
```

On Lines 181 through 189 we iterate through our layer collection. On Line 182 we first check to see if the layer type is a conductor and – if so – if the conductor type is a signal layer. If this layer is a signal layer, on Line 183 we increment the count of the number of signal layers we've found and then on Line 184 we compare this value to the one we're looking for (which was passed in as a parameter). If we do find the layer we're looking for, on Line 185 we set the function's return value to this layer and on Line 186 we exit 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
183              layerObj.ConductorType = epcbConductorTypeSignal Then
184              numFoundInt = numFoundInt + 1
185              If numFoundInt = nthLayerInt Then
186                  Set GetSignalLayer = layerObj
187                  Exit Function
188              End If
189      Next
190  End Function
```

GetSignalLayerCount() Function

Lines 194 through 207 declare the routine that is used to count the number of signal layers in whatever layer collection is passed in as a parameter.

```
192  ' Returns the number of signal layers in layerColl
193  ' layerColl - LayerObjects Collection
194  Function GetSignalLayerCount(layerColl)
```

On Line 195 we instantiate an integer variable and initialize it with 0 (this variable will be used to keep track of how many signal layers we've found).

```
195      Dim numInt: numInt = 0
```

On Lines 199 through 203 we iterate through our layer collection. On Line 200 we first check to see if the layer type is a conductor and – if so – if the conductor type is a signal layer. If this layer is a signal layer, then on Line 201 we increment the count of the number of signal layers we've 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
```

On Line 206 we set the function's return value to the number of signal layers we've found and then we exit the function.

```
196
204
205      ' Return the number of layers
206      GetSignalLayerCount = numInt
207  End Function
```

GetAngleOfLine() Function

Lines 214 through 229 declare the routine that is used to determine the angle of a line in degrees from 0° to 90°. This function accepts four parameters which 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)
```

On line 215 we declare a variable in which to store the value of the angle of the line.

```
215      Dim angleReal
```

Now, the equation we use on Line 224 doesn't handle the extreme 0° and 90° cases very well (in one of these cases we end up with a "divide-by-zero", for example, which is not a good thing to happen). Thus, on Line 218 we check to see if the Y coordinates of the line's two end-points are identical; if so, we're dealing with a horizontal line, so on Line 219 we set the angle to 0 radians (which corresponds to 0°).

Alternatively, if we aren't dealing with a horizontal line, then on Line 220 we check to see if the X coordinates of the line's two end-points are identical; if so, we're dealing with a vertical line, so on Line 221 we set the angle to PI/2 radians (which corresponds to 90°).

If the line is neither horizontal nor vertical, then we use the equation on Line 224 to calculate its actual angle in radians. Observe the use of the VBScript *Atn* (arctan) and *Abs* (absolute value) functions. In particular, it's the use of 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 Figures 20-3 and 20-4.

```
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) /  
225                                (x2Real - x1Real) ))  
225      End If
```

Finally, on Line 228 we call our *RadiansToDegrees()* helper function to convert the angle of the line from its value in radians to a corresponding value in degrees, and this is the value we return from the function.

```
216  
217  
218      ' Convert to degrees.  
219      GetAngleOfLine = RadiansToDegrees(angleReal)  
220  End Function
```

RadiansToDegrees() Function

Lines 233 through 235 declare a simple routine that is used to convert 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 described above (including the license server function which is not shown above) and save it out as *RouteByLayer.vbs*.
- 2) Close any instantiations of Expedition PCB that are currently running, and then launch a new instantiation of this application.
- 3) Open the *Candy.pcb* design we've been using throughout these building block examples.
- 4) Use the **Edit > Delete All Traces and Vias** command to remove any existing traces and vias.

-
- 5) Run the *RouteByLayer.vbs* script. Observe that the signal layers on the top half of the board contain the horizontally-biased nets (white); also that vertically-based nets (red) are shown on the signal layers on the bottom half of the board.

Enhancing the Script

There are a number of ways in which this script could be enhanced or modified. Some suggestions are as follows (it would be a good idea for you to practice your scripting skills by implementing these examples):

- Take a board containing a substantial amount of surface-mount devices and create a new version of the script that will direct 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.
- Take 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 20-5.

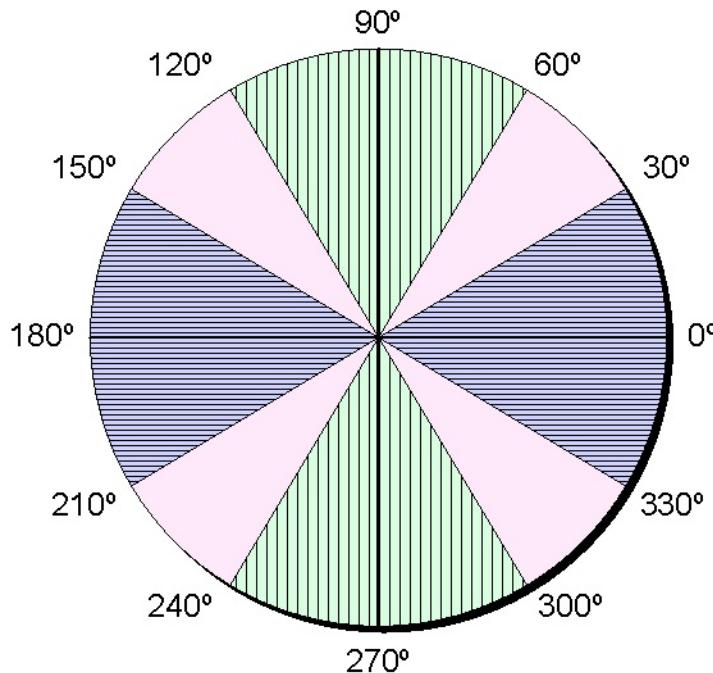


Figure 20-5. Categorizing (and routing) nets as horizontally-biased, vertically-biased, and "in-between".

Chapter 21: Show Maximum Length/Delay

Introduction

- Overview:** This script acquires the maximum length or Time-of-Flight (TOF) delay constraint information associated with each physical net from the Constraint Editor System (CES). The script then displays this information in Expedition PCB when the user selects a net.
- Approach:** To open CES one time and to read all of the maximum length or TOF delay values into a *Dictionary* object so that they can be easily accessed and displayed for the currently selected (individual) net.
- Points of Interest:**
- Connecting to CES
 - Accessing data from CES
 - Displaying a progress bar
 - Adding a menu entry
 - Implementing a time-out
- Items Used:**
- General CES Objects
 - ProgressBar and ProgressBarInitialize Methods
 - Dictionary Object
 - VBScript Timer Method

Before We Start

For the purposes of this chapter, let's assume we are working with a layout design that has an associated Constraint Editor System (CES) database as illustrated in Figure 21-1.

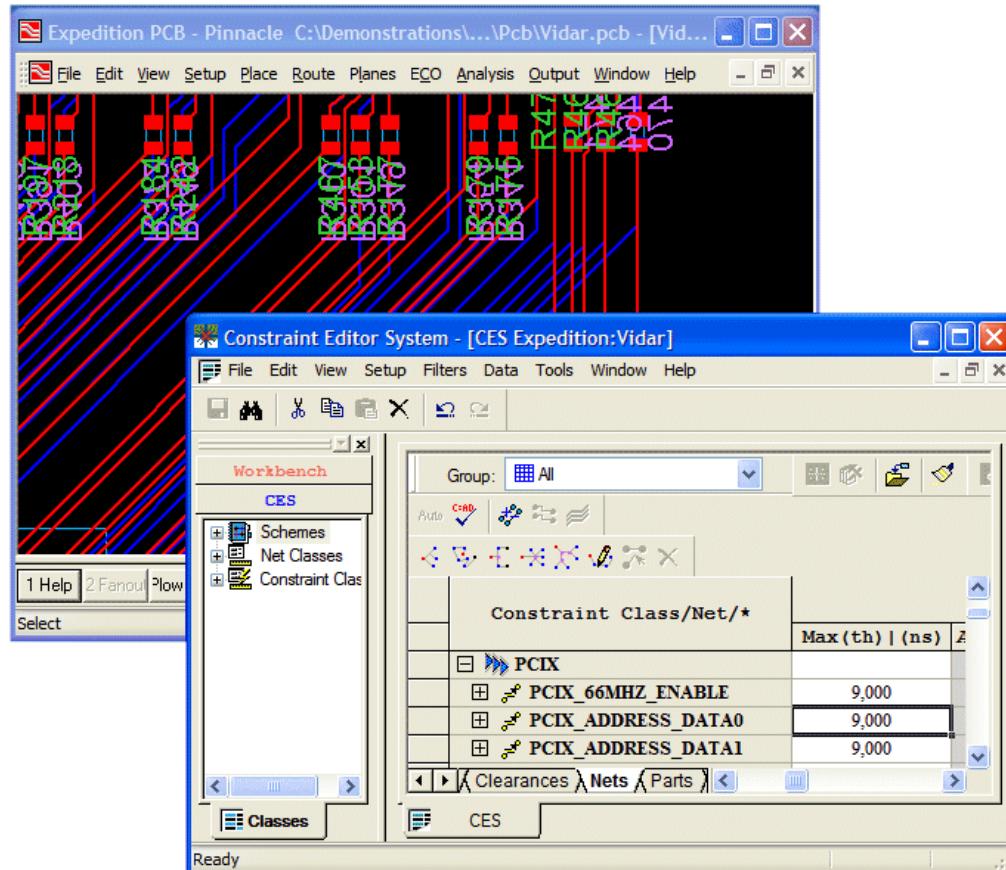


Figure 21-1. A layout design with an associated CES database.

As we see in Figure 21-1, each net can be provided with a constraint in the form of a maximum length (shown here in thousandths of an inch) or TOF delay (shown here in nanoseconds). The creator of the database decides whether to specify this information as length or TOF delay on a net-by-net basis.

Now, suppose we want to be able to determine the maximum length or delay constraint associated with a particular net. One technique would be to use the **Setup > Constraints** command to invoke CES, and to then search the database by hand to locate the required value as illustrated in Figure 21-1. However, this can be slow and time-consuming if we want to access the values associated with a number of different nets.

The alternative would be to create the script described in this chapter. When this script is run, one of the first tasks it will perform is to add a new **Display Max Delay** item to the **Route** pull-down menu as illustrated in Figure 21-2 (this command might be better named **Display Max Length/Delay** – see also the *Enhancing the Script* topic later in this chapter).

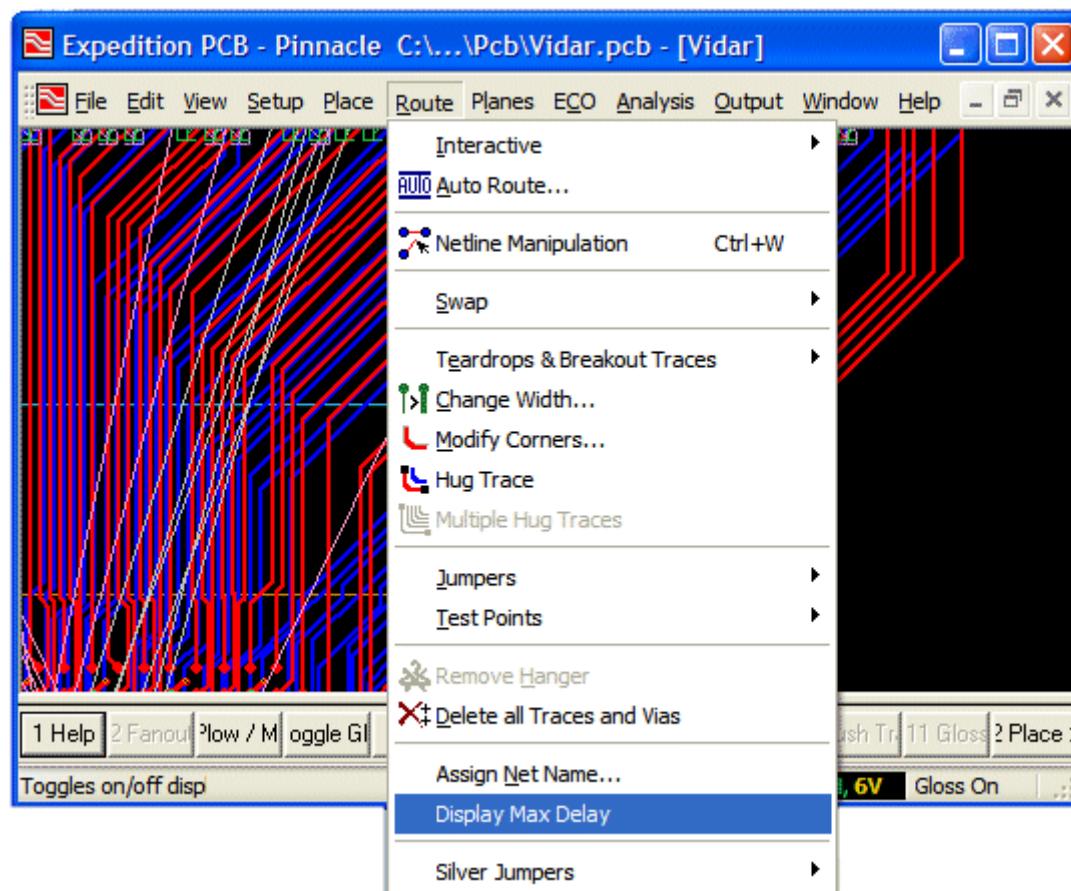


Figure 21-2. The script creates a new "Route > Display Max Delay" command.

This command is a "toggle" command that will turn our display capability *On* or *Off*. By default, the script starts with this command in its inactive state, which means that if we select a net we will *not* be presented with its maximum length / TOF delay information.

When the **Route > Display Max Delay** command is toggled *On*, the script will check to see if we've previously run this command; if not, it will open CES and read all of its maximum length / TOF delay information into a dictionary object. While this is taking place, a *Progress Bar* will be displayed as illustrated in Figure 21-3.

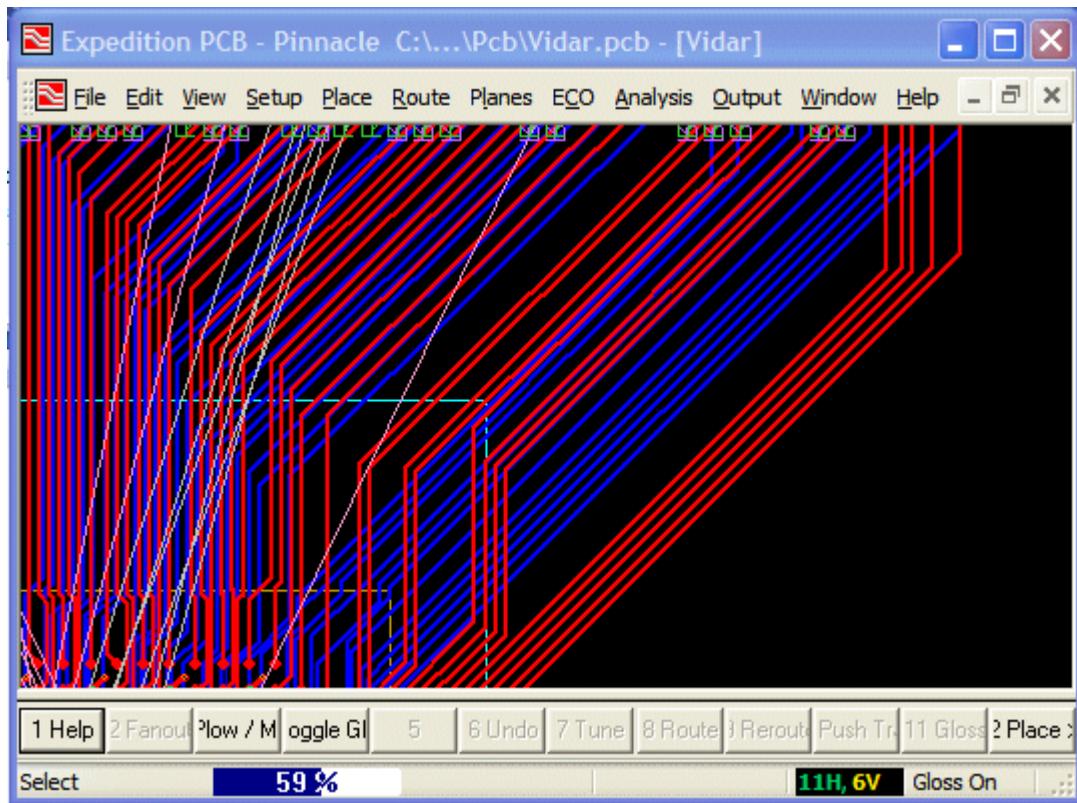


Figure 21-3. A progress bar reflects the progress of loading data from CES.

Once the data from CES has been loaded into the dictionary object, selecting a net will result in its maximum length / TOF delay being displayed in the status bar as illustrated in Figure 21-4.

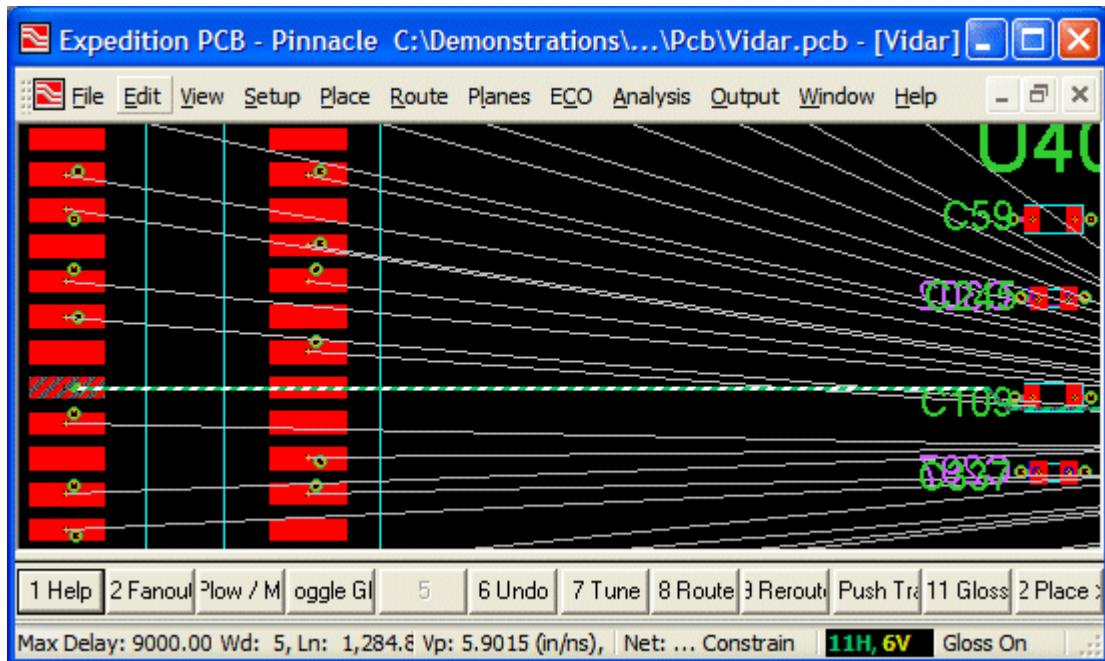


Figure 21-4. Maximum length / TOF delay data is displayed in the status bar.

If the user now toggles the **Route > Display Max Delay** command *Off*, maximum length / TOF delay information will no longer be presented in the status bar. Furthermore, if the **Route > Display Max Delay** command is toggled *On* again, the script will not bother to re-launch CES and re-load its data, because the original data is still available to the script in the form of the *Dictionary* object.

The Script Itself

The key sections forming this script are as follows:

- **Constant Definitions** Define the constants to be used by the script.
- **Initialization/Setup** Get the *Application* and *Document* objects, license the document, and add the type libraries. Also perform any additional setup associated with this script.
- **Event handlers** Any event handlers used by the script.
 - *OnDisplayMaxDelay()*
 - *OnSelectionChange()*
- **Utility Functions and Subroutines** Special "workhorse" routines that are focused on this script.
 - *LoadMaxDelayDictionary()*
- **Helper Functions and Subroutines** Generic "helper" routines that can be used in other scripts.
 - *GetCESApplication()*
 - *AddMenuAfter()*
 - *GetMenuNumber()*
- **Validate Server** The standard function used to license the document and validate the server. (Note that we won't go through this function in this chapter – for more details on this function see *Chapter 3: Running Your First Script*.)

Constant Definitions

On Line 11 we declare a string constant that we will use to access the maximum length / TOF delay items stored in CES. On Line 12 we specify a time-out value (in seconds) – see also the discussion regarding invoking/opening CES.

```
1  ' This script demonstrates how to connect to CES from
2  ' the context of Expedition. This script creates
3  ' a structure to hold the Max delay for all nets
4  ' and displays the information in the status bar
5  ' when the net is selected.
6 '
7
8 Option Explicit
9
10 ' Constants
11 Const CES_MAX_DELAY = "MAX_DELAY"
12 Const TIME_OUT = 60   ' In seconds
```

Initialization/Setup

On Line 14 we add the type library required by this script.

Lines 16 and 17 are used to get the *Application* object; Lines 20 and 21 are used to get the *Document* object; on Line 24 we license the document by calling the standard *ValidateServer()* function; and Lines 27 and 28 are used to add the type libraries required by this script.

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

On Lines 18 and 19 we instantiate variables that we will use to hold our *Application* and *Document* objects. On Line 20 we declare a Boolean variable that we will use to determine if the menu item we are about to create is in its *On* (*active*) or *Off* (*inactive*) state; then, on Line 21, we assign a value of *False* to this variable, which means the item is *Off* by default. On Line 22 we instantiate a variable to hold a *Dictionary* object and on Line 23 we assign a value of *Nothing* to this variable.

```
17  ' Global variables  
18  Dim pcbAppObj           'Application object  
19  Dim pcbDocObj           'Document object  
20  Dim enabledBool          'Flag to determine enabled.  
21  enabledBool = False  
22  Dim maxDelayDictObj     'Dictionary object  
23  Set maxDelayDictObj = Nothing
```

On Lines 26 through 32 we acquire our *Application* and *Document* objects and license (validate) the document as usual.

```
25  ' Get the application object.  
26  Set pcbAppObj = Application  
27  
28  ' Get the active document  
29  Set pcbDocObj = pcbAppObj.ActiveDocument  
30  
31  ' License the document  
32  ValidateServer(pcbDocObj)
```

As we discussed in the *Before We Start* topic, our script is going to add a new **Display Max Delay** item to the **Route** pull-down menu. On Lines 35 through 48 we add and configure this button. (For more details on fundamental concepts pertaining to adding menu commands and buttons, see also *Chapter 6: Adding a Menu and/or Menu Button/Item*. Furthermore, see *Chapter 13: Display a Single Routing Layer* for more details on the *AddMenuAfter()* and *GetMenuNumber()* helper functions.)

```
34  ' Get the document menu bar.  
35  Dim docMenuBarObj  
36  Set docMenuBarObj =  
            pcbAppObj.Gui.CommandBars( "Document Menu Bar" )  
37  
38  ' Get the Route menu  
39  Dim routeMenuObj  
40  Set routeMenuObj = docMenuBarObj.Controls.Item( "&Route" )  
41  
42  ' Setup the menu entry  
43  Dim showMaxBtnObj  
44  Set showMaxBtnObj =  
            AddMenuAfter( "Assign Net Name...", routeMenuObj )  
45  showMaxBtnObj.Caption = "Display Max Delay"  
46  showMaxBtnObj.DescriptionText =  
            "Toggles on/off display of max delay in status bar."
```

```
47 showMaxBtnObj.Target = ScriptEngine
48 showMaxBtnObj.ExecuteMethod = "OnDisplayMaxDelay"
```

On Line 51 we attach any events to our *Document* object.

```
50 ' Attach events.
51 Call Scripting.AttachEvents(pcbDocObj, "pcbDocObj")
```

On Line 54 we set the *DontExit* property of the *Scripting* object to the Boolean value of *True*. This means this script will continue running until the end of the current application session, thereby allowing the script to detect and handle events.

```
53 ' Hang around to catch events.
54 Scripting.DontExit = True
```

Event Handlers

This is where we declare any event handlers that will be used by the script.

OnDisplayMaxDelay() Function

Lines 60 through 67 declare the routine that is used to handle the user toggling the **Route > Display Max Delay** command *On* or *Off*. Note that the name of this event handler was defined on Line 48 as part of our adding a new menu button/item.

On Line 61 we simply invert the state of the Boolean variable we use to determine if our command is currently active or not. On Line 64 we test to see if our menu command is currently active and – if so – we also test to see if our *Dictionary* object is empty (*Nothing*). If both of these tests are true, then on Line 65 we call our *LoadMaxDelayDictionary()* utility subroutine (we pass this routine our *Dictionary* object as a parameter).

```
59 ' Menu handler
60 Function OnDisplayMaxDelay(id)
61     enabledBool = Not enabledBool
62
63     ' Initialize the dictionary if necessary.
64     If enabledBool And maxDelayDictObj Is Nothing Then
65         Call LoadMaxDelayDictionary(maxDelayDictObj)
66     End If
67 End Function
```

OnSelectionChange() Subroutine

Lines 70 through 88 declare the routine that is used to handle the case where there is a selection change. Remember that our script can only display the maximum length / TOF Delay associated with a single net, so we have to test for this as we shall see. Also remember that the name of this event handler subroutine is predefined as part of the Expedition PCB automation interface.

```
69 ' Fired during selection change.
70 Sub pcbDocObj_OnSelectionChange(unused)
```

On Line 72 we test to see if our menu command is currently active and – if so – we also test to see if we are in the layout editor's *Route* mode. If both of these tests are true, then on Line 74 we acquire a collection of any nets that have been selected.

```

71      ' If the functionality is enabled.
72      If enabledBool And pcbAppObj.Gui.ActiveMode =
73          epcbModeRoute Then
74          ' Get the collection of nets
75          Dim netColl: Set netColl =
76              pcbDocObj.Nets(epcbSelectSelected)

```

On Line 77 we check to see if our collections contains a single net (which means the user selected only one net). If there is only a single net in the collection, then on Line 80 we check to see if this net is in our dictionary and – if it is – in Line 82 we retrieve its maximum length / TOF delay value and display it in the status bar.

```

76          ' Show max delay for single net selection
77          If netColl.Count = 1 Then
78              Dim netObj
79              Set netObj = netColl.Item(1)
80              If maxDelayDictObj.Exists(netObj.Name) Then
81                  ' Set the status field
82                  Call pcbAppObj.Gui.StatusBarText("Max
83                      Delay: " & maxDelayDictObj.Item(netObj.Name),
84                      epcbStatusFieldPrompt)

```

Alternatively, if multiple nets have been selected, then on Line 84 we display an appropriate message in the status bar.

```

83          Else
84              Call pcbAppObj.Gui.StatusBarText("Max
85                  Delay: NA", epcbStatusFieldPrompt)
86          End If
87      End If
88 End Sub

```

Utility Functions and Subroutines

This is where we declare a utility routine. This routine is similar in context to our generic "helper" routines (see the next topic), except that it is specific to this script and may not be used elsewhere without some "tweaking".

LoadMaxDelayDictionary() Subroutine

Lines 96 through 158 declare the main routine that is used to load our dictionary object with a collection of nets and their associated maximum length / TOF delay values. This routine accepts a single parameter, which is our *Dictionary* object (remember that all VBScript parameters are of the In-Out persuasion, which means that modifications made to this parameter inside the routine will be visible to the outside world (the rest of the script) when the routine eventually terminates.

```

93  ' Initializes the dictionary with a mapping of nets to
94  ' corresponding max delay values.
95  ' dictObj - Dictionary Object (in/out)
96  Sub LoadMaxDelayDictionary(dictObj)

```

The first thing we do on Line 98 is initialize (create) a new dictionary and assign it to our *Dictionary* object.

```
97      ' Initialize the dictionary object.  
98      Set dictObj = CreateObject("Scripting.Dictionary")
```

In a moment we are going to attempt to invoke CES. But before we do that, on Line 102, we use the *ProgressBarInitialize* method to initialize a progress bar that will indicate to the user how well we are doing with respect to invoking CES.

The first parameter to this method is set to *True*, which means we want to display the progress bar; the second parameter is a string that will be displayed beside the progress bar; the third parameter is the maximum possible progress value; and the fourth parameter is the minimum possible progress value.

Now, since we can't measure the progress very accurately with regard to invoking CES, on Line 103 we instruct the progress bar to start displaying a value of 50, which equates to 50% based on the values of the third and fourth parameters specified on Line 102. (We'll consider a more "interesting" use of a progress bar shortly.)

```
100      ' Can't measure this precisely so set it  
101      ' at 50% to start out  
102      Call pcbAppObj.Gui.ProgressBarInitialize(True,  
                                         "Invoking CES", 100, 0)  
103      Call pcbAppObj.Gui.ProgressBar(50)
```

On Line 106 we instantiate two variables: one to hold a *CES Tool Application* object and one to hold a *CES Workbench Application* object (see the CES documentation for more information on these object types and how they are used). On Line 107 we instantiate a Boolean variable, which we will use to tell us whether we successfully invoked CES ... or not.

```
105      ' Get the ces application object.  
106      Dim cesAppObj, workbenchAppObj  
107      Dim createdBool
```

On Line 108 we call our *GetCESApplication()* helper function and pass it two parameters – the variables we just declared to hold our *CES Tool Application* and *CES Workbench Application* objects. The return value from this function is a Boolean that tells us if we invoked CES (the alternative is that CES was already running, in which case we simply attached to that running instance).

```
108      createdBool = GetCESApplication(cesAppObj, workbenchAppObj)
```

Irrespective of whether we succeeded or failed in invoking CES, on Line 111 we set our progress bar to display a value of 100 (which equates to 100% in this case) and then on Line 112 we disable the display of the progress bar. (In reality, turning off the display of the progress bar on Line 112 means that the statement on Line 111 is essentially meaningless, but it doesn't hurt anything and it makes our intentions clear to anyone reading this code in the future.)

```
110      ' Done. Remove status bar.  
111      Call pcbAppObj.Gui.ProgressBar(100)  
112      Call pcbAppObj.Gui.ProgressBarInitialize(False)
```

On Lines 115 through 119 we check to see if we were successful in invoking CES and in acquiring our *CES Tool Application* object. If not, we use a *MsgBox()* to inform the user, we re-initialize our *Dictionary* object to *Nothing*, and we exit this routine.

```
114      ' If we failed let the user know.  
115      If cesAppObj Is Nothing Then
```

```

116             MsgBox "Error: Unable to invoke CES. Try to
                     re-enable after CES invokes."
117             Set dictObj = Nothing
118             Exit Sub
119         End If

```

Assuming we did successfully invoke CES, then on Lines 122 and 123 we acquire the *CES Design* object.

```

121     ' Get the CES design object
122     Dim cesDesignObj
123     Set cesDesignObj = cesAppObj.ObjectDefs.Design

```

On Lines 126 and 127 we acquire a collection of physical nets from the *CES Design* object.

```

125     ' Get the collection of physical nets from CES.
126     Dim cesPhysicalNetColl
127     Set cesPhysicalNetColl = cesDesignObj.PhysNets

```

On Line 130 we declare an integer variable that we will use to keep track of the number of the current nets with which we're working. On Line 131, we initialize a progress bar that will indicate to the user how well we are doing with respect to processing our collection of nets. Once again, the first parameter to this method is set to *True*, which means we want to display the progress bar; the second parameter is a string that will be displayed beside the progress bar; the third parameter is the maximum possible progress value (which is set to the total number of physical nets in this case); and the fourth parameter is the minimum possible progress value.

```

129     ' Track progress by iteration of nets
130     Dim progressInt: progressInt = 0
131     Call pcbAppObj.Gui.ProgressBarInitialize(True,
                                         "Loading Max Delay", cesPhysicalNetColl.Count, 0)

```

On Lines 135 through 148 we iterate through our collection of CES physical net objects. On Line 138 we acquire the maximum length / TOF delay value for the current net.

```

133     ' Iterate through the physical nets.
134     Dim cesPhysicalNetObj
135     For Each cesPhysicalNetObj In cesPhysicalNetColl
136         ' Get the max delay constraint.
137         Dim maxDelayStr
138         maxDelayStr =
                  cesPhysicalNetObj.Object.ConstraintEx(CES_MAX_DELAY)

```

On Line 141 we check to make sure that this net does indeed have a maximum length / TOF delay value associated with it. If so, then on Line 142 we add this net name and its maximum length / TOF delay value to our script's *Dictionary* object.

```

140             ' If there is a value add it to the dictionary.
141             If Not maxDelayStr = "" Then
142                 Call dictObj.Add(cesPhysicalNetObj.Name,
                               maxDelayStr)
143             End If

```

In Line 146 we increment the integer we're using to keep track of which net we're currently working on, and then on Line 147 we present this new value to our progress bar.

```

145      ' Update the progress meter.
146      progressInt = progressInt + 1
147      Call pcbAppObj.Gui.ProgressBar(progressInt)
148      Next

```

On Line 151 we check again to see if we invoked CES versus attaching to it and – if so – on Line 152 we exit (quit) the *CES Workbench Application* object (the idea here is that if we didn't invoke it, then we should leave it to whoever did to close it).

```

150      ' If we created CES close it
151      If createdBool Then
152          workbenchAppObj.Quit
153      End If

```

Finally, on Line 156 we disable the display of the progress bar and then on Line 158 we exit this routine.

```

155      ' Hide the progress bar.
156      pcbAppObj.Gui.ProgressBarInitialize(False)
157
158  End Sub

```

Helper Functions and Subroutines

This is where we declare some generic "helper" routines. These routines are not focused on this script per se; they can easily be used in other scripts.

GetCESApplication() Function

Lines 167 through 200 declare the routine that is actually used to acquire a CES application. If a CES application is currently running, then this function will attach us to it; otherwise, we will attempt to invoke a new CES application. The two (In-Out) parameters to this function are variables that are intended to hold *CES Tool Application* and *CES Workbench Application* objects, respectively.

```

163  ' Attaches to a running CES application.  If no
164  ' CES application is running a new one is invoked.
165  ' cesAppObj - CES "Application" Object (in/out)
166  ' workbenchAppObj - Workbench Application Object (in/out)
167  Function GetCESApplication(cesAppObj, workbenchAppObj)

```

On Line 169 we initialize a return value to *False*; on Line 170 we initialize our *CES Workbench Application* object variable to *Nothing*; and on Line 171 we initialize our *CES Tool Application* object variable to *Nothing*.

```

169      Dim retBool: retBool = False
170      Set workBenchAppObj = Nothing
171      Set cesAppObj = Nothing

```

Now, before we do anything else, we need to check to see if CES is already running. In order to do this, on Line 174 we disable automatic error handling; on Line 175 we attempt to acquire the *CES Workbench Application* object; and on Line 176 we re-enable automatic error checking.

```

173      ' First see if it is already running
174      On Error Resume Next : Err.Clear

```

```
175      Set workBenchAppObj = GetObject( , "Workbench.Application" )
176      On Error GoTo 0
```

On Line 178, we check to see if we successfully acquired a *CES Workbench Application* object. If so, then on Line 179 we attempt to acquire a *CES Tool Application* object.

```
178      If Not workBenchAppObj Is Nothing Then
179          Set cesAppObj = workBenchAppObj.Tools("CES")
180      End If
```

On 183 we check to see if the *CES Tool Application* object is *Nothing*, which means that CES is not already running. If this is the case, then on Line 184 we use the *ProcessCommand* method to invoke CES by means of the **Setup > Constraints** command.

```
182      ' If it wasn't running try to invoke it
183      If cesAppObj Is Nothing Then
184          pcbAppObj.Gui.ProcessCommand("Setup->Constraints...")
```

Now, it may take a little while for CES to be launched. In some cases, CES may (for reasons outside of our control) fail to launch at all. The point is that we don't want our script to hang around forever waiting for CES to launch if it's not going to do so. In order to address this, on Line 185 we instantiate a variable to hold the initial time value and then on Line 186 we load this variable the current time.

```
185          Dim startTime
186          startTime = Timer
```

On Lines 187 through 195 we loop around looking to see if we've successfully acquired a *CES Tool Application* object. This loop will terminate if we do successfully invoke CES or if a time-out occurs (as determined by the *TIME_OUT* constant value we declared on Line 12).

```
187          While ((Timer - startTime) < TIME_OUT) And
188              (cesAppObj Is Nothing)
189              On Error Resume Next : Err.Clear
190              Set workBenchAppObj =
191                  GetObject( , "Workbench.Application" )
192              On Error GoTo 0
193              If Not workBenchAppObj Is Nothing Then
194                  Set cesAppObj =
195                      workBenchAppObj.Tools("CES")
196              End If
197              Call Scripting.Sleep(1000)
198          Wend
```

Finally, on Line 196 we set our return value to True; on Line 199 we assign this value to be the return value from our function; and on Line 200 we exit the function.

```
196          retBool = True
197      End If
198
199      GetCESApplication = retBool
200  End Function
```

AddMenuAfter() Function

Lines 208 through 215 declare the *AddMenuAfter()* helper function, which can be used to add a new menu button/item after an existing menu button/item (this function was previously used

in *Chapter 13*). The return value from the *AddMenuAfter()* function – as defined on Line 213 – specifies the location where the new menu button/item is to be added into the menu. Observe that, on Line 211, this function calls the *GetMenuNumber()* helper function, which we will consider in a moment.

```
202  ' Creates a new menu entry, menuToAdd, on menuBar after the
203  ' afterMenuEntry entry. If the entry is not found the menu is
204  ' added at the end.
205  ' menuToAddStr - String
206  ' afterMenuEntryStr - String
207  ' menuBarObj - CommandBarSrv menu object
208  Function AddMenuAfter(afterMenuEntryStr, menuBarObj)
209      Dim entryNumInt
210      entryNumInt = GetMenuNumber(afterMenuEntryStr,
211                                     menuBarObj) + 1
212      If entryNumInt > menuBarObj.Controls.Count Then
213          entryNumInt = -1
214      End If
215      Set AddMenuAfter =
216          menuBarObj.Controls.Add(cmdControlButton,,,entryNumInt)
217  End Function
```

GetMenuNumber() Function

Lines 221 through 238 declare the *GetMenuNumber()* function, which returns the menu index number associated with a given menu button/item string (this function was previously used in *Chapter 13*). As we previously noted, the *GetMenuNumber()* function may be considered to be a "helper's helper" (sort of like a "gentleman's gentleman"). It performs its task by cycling through all of the buttons/items and comparing their captions with the one it's looking for. The value returned by this function – as defined on Line 232 – is the required menu index number.

```
217  ' Function that returns a menu index for a given
218  ' menu entry string.
219  ' menuToFindStr - String
220  ' menuBarObj - CommandBarSrv menu object
221  Function GetMenuNumber(menuToFindStr, menuBarObj)
222      Dim ctrlColl : Set ctrlColl = menuBarObj.Controls
223      Dim ctrlObj
224
225      Dim menuCntInt : menuCntInt = 1
226      GetMenuNumber = -1
227
228      For Each ctrlObj In ctrlColl
229          Dim captStr: captStr = ctrlObj.Caption
230          captStr = Replace(captStr, "&", "")
231          If captStr = menuToFindStr Then
232              GetMenuNumber = menuCntInt
233              Exit For
234          End If
235          menuCntInt = menuCntInt + 1
236      Next
237
238  End Function
```

Creating and Testing the Script

- 1) Use the scripting editor of your choice to enter the script described above (including the license server function which is not shown above) and save it out as *ShowMaxDelay.vbs*.
- 2) Close any instantiations of Expedition PCB that are currently running, and then launch a new instantiation of this application.
- 3) Open a layout design that has an associated CES database (where this database contains maximum length / TOF delay information).
- 4) Run the *ShowMaxDelay.vbs* script.
- 5) Execute the new **Route > Display Max Delay** command created by the script and observe the progress bar as the maximum length / TOF delay information is loaded.
- 6) Select a net and observe its delay information being displayed in the status bar; repeat for other nets.
- 7) Toggle the **Route > Display Max Delay** command, select a net, and observe that its delay information is *not* displayed in the status bar.
- 8) Toggle the **Route > Display Max Delay** command again and observe that this time the script does not bother opening and accessing CES because we already have the required information.
- 9) Select a net and observe that its delay information is once again displayed in the status bar.

Enhancing the Script

There are a number of ways in which this script could be enhanced. Some suggestions are as follows (it would be a good idea for you to practice your scripting skills by implementing these examples):

- CES can contain a wide variety of constraint data; you could modify this script to access and display the constraints that are important to you.
- The item we added to the **Route** pull-down was named **Display Max Delay**. It may be better to name this item **Display Max Length/Delay** (see also the following point).
- This script simply accesses the values associated with the **Max Length/Delay** column in CES and – when requested to do so (by the user selecting a net) displays it without appending any units or appropriate description. Thus, one modification when performing the original load from CES would be to access the **Type** column to ascertain whether the maximum value for each net was specified in terms of length or TOF delay. This information could be stored along with the length/delay value, and then used to augment the information being presented to the user.

SECTION 3: DETAILED EXAMPLES – FABLINK XE

Chapter 22: Introducing FabLink XE

Introducing FabLink XE

FabLink XE 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.

FabLink XE may be regarded as being "the next step" from Expedition PCB because it helps verify your data and prepare files for manufacture. One way to think about this is that FabLink XE provides a "Bridge" between Expedition PCB and the board manufacturing environment. Another way to contrast the two applications is that Expedition PCB is used to create a single board design, while FabLink XE can be used to create a panel containing multiple boards with the sheet size of fabrication materials used in the PCB manufacturing process as illustrated in Figure 22-1.

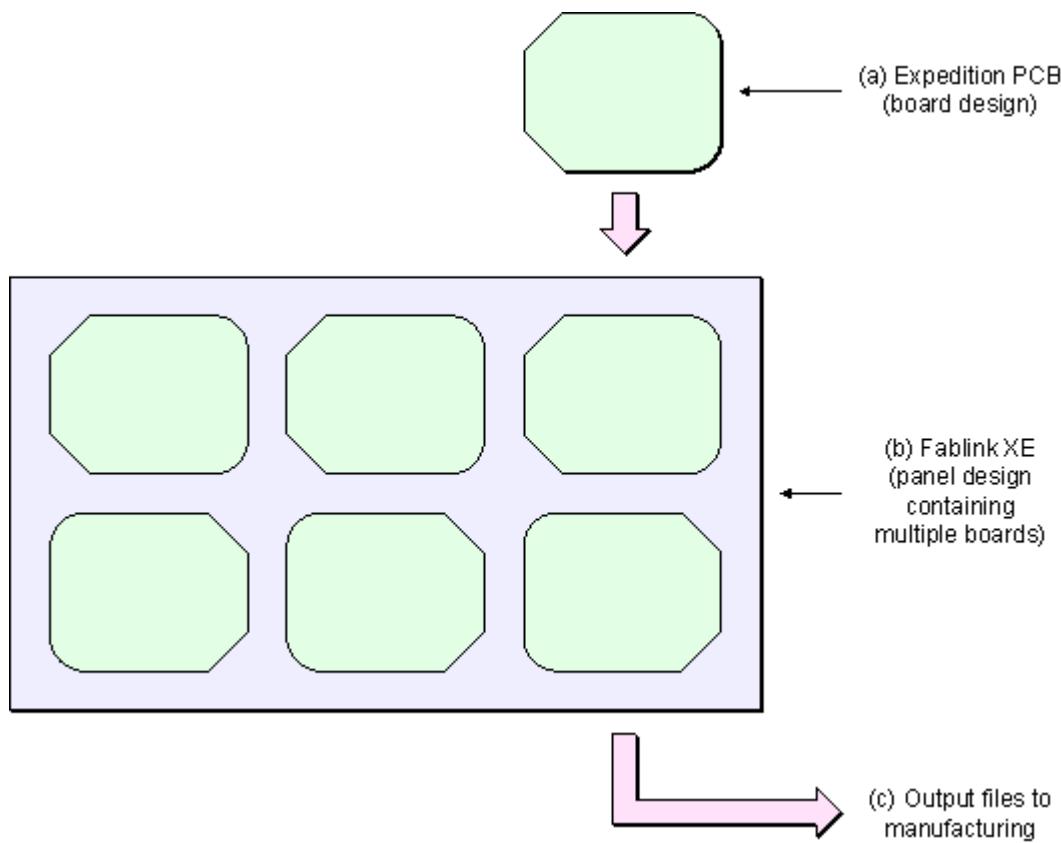


Figure 22-1. A high-level view of FabLink XE versus Expedition PCB.

FabLink XE From an Automation Perspective

The automation interface used for accessing data in FabLink XE is exactly the same as that used in Expedition PCB. Having said this, there is certain functionality in FabLink XE that is not applicable to (and therefore not available in) Expedition PCB, and vice versa.

Furthermore, FabLink XE has additional data structures, such as 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 product being accessed.

Consider, for example, the screenshot of FabLink XE in Figure 22-2, in which we see a panel called *CandyPanel* containing six instances of the *Candy.pcb* design we experimented with in Sections 1 and 2 of this tutorial.

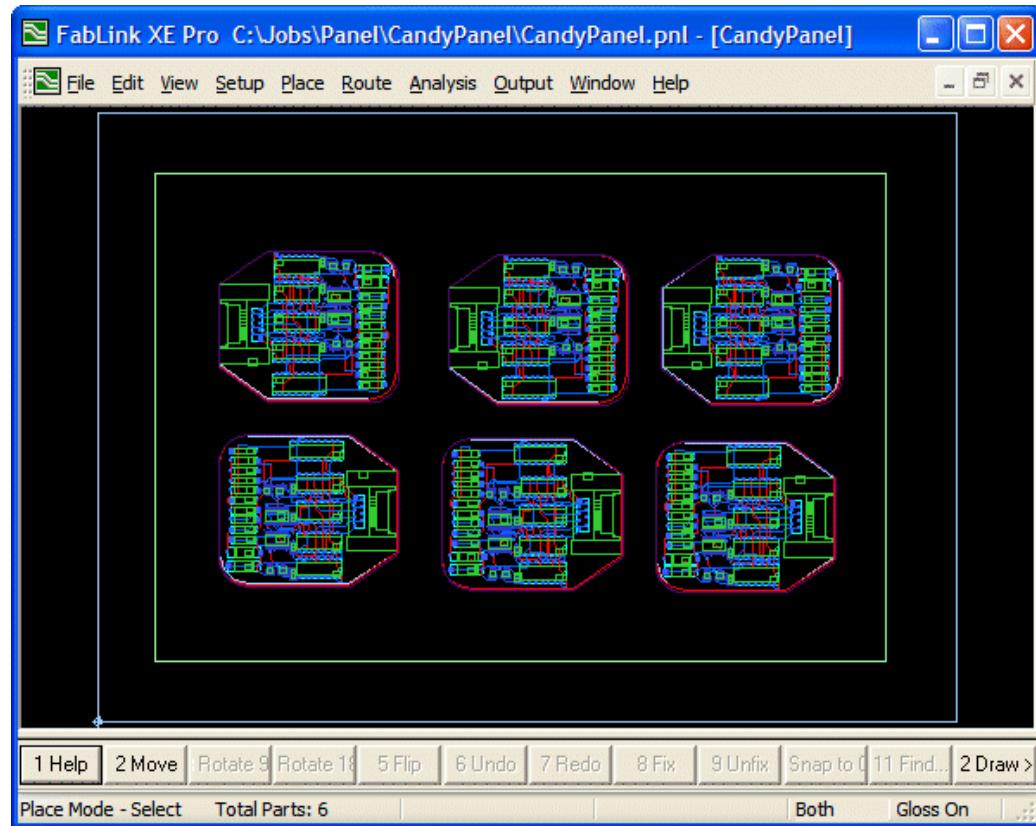


Figure 22-2. A screenshot of FabLink XE showing a panel containing six boards.

Observe that FabLink XE's Graphical User Interface (GUI) visually depicts the various items on each board (components, traces, pins, vias, etc.). However, the only items that are accessible via FabLink XE (either 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 FabLink XE.

The point here is that FabLink XE is not intended for designing boards, so it is not possible to access items that are part of a board design from within FabLink XE.

Chapter 23: Generating a BOM

Introduction

Overview:	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.
Approach:	To access board information from FabLink XE, component information from Expedition PCB, and part information from the Parts Database (Parts DB).
Points of Interest:	<ul style="list-style-type: none">– Using a FabLink XE script to access data in Expedition PCB.– Using a FabLink XE script to access data in the Parts DB.– Using dictionary-based tree structures to store and manipulate component and part data.
Items Used:	<ul style="list-style-type: none">– Expedition PCB Server– Parts Editor Server– Dictionary Object

Before We Start

This script uses the Expedition PCB, FabLink XE, and Parts Editor automation engines. The script is intended to run from the command line; this means that there is no need to have Expedition PCB, FabLink XE, or the Parts Editor up and active prior to running the script, because the script will automatically launch these applications as required.



Note: It doesn't matter if any of the Expedition PCB, FabLink XE, or the Parts Editor applications are already up and active when the script is run; the script will simply invoke new instantiations of these applications as required and then close these new instantiations when it's finished with them.

As has already been discussed, in the case of the script described in this chapter we are going to use the Parts Editor automation server. We should also note that there is a Padstack Editor that can be used in the same way as the Parts Editor. The *Prog IDs* for these engines are as follows:

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

Moving on, some panels are homogenous (all of the boards on the panel are of the same type); others may be heterogeneous (the boards on the panel may be of two or more different types). For the purposes of these discussions, we will be working with a homogeneous panel called *CandyPanel.pnl* that contains six identical copies of our *Candy.pcb* board design as illustrated in Figure 23-1.

If we knew that all of the panels to be processed by this script were to be homogeneous, the script would only need to access the components on one of the boards and then multiply the number of each type of component by the number of boards on the panel. However, we are going to create the script in such a way that it gathers enough information to support both homogeneous and heterogeneous panels; we are also going to create the script in such a way that it can be easily extended to support any required output format.

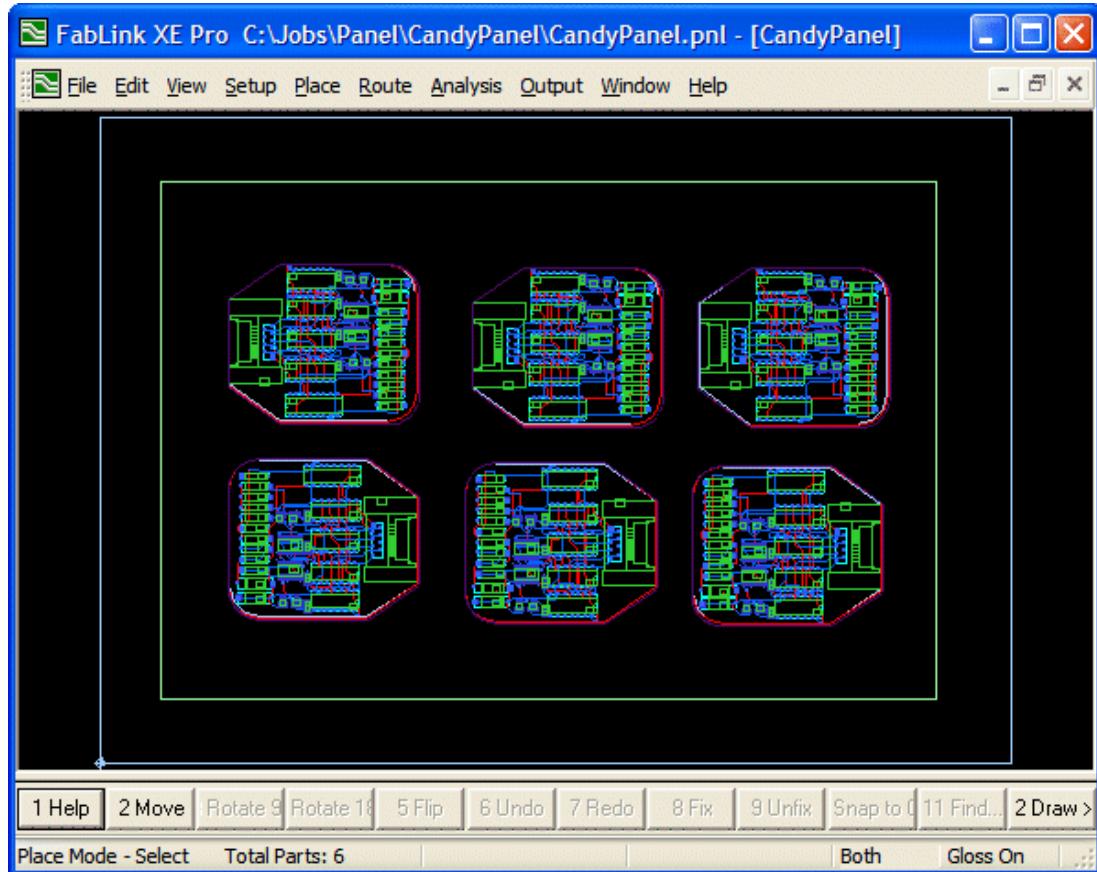


Figure 23-1. A screenshot of FabLink XE showing a panel containing six boards.

Now, assume that we've already created a script called `GeneratePanelBOM.vbs` and that this script resides in your `C:\Temp` directory/folder. Launch a console (terminal) window and set its context to `C:\Temp` (see *Chapter 8: Running Scripts from the Command Line* for more details on using console windows). Prepare to use the `mgcscript` engine to run the `GeneratePanelBOM.vbs` script. As the only argument to this script, specify the name of the panel file you want to process (`c:\jobs\Panel\CandyPanel\CandyPanel.pnl` in this example) as illustrated in Figure 23-2.

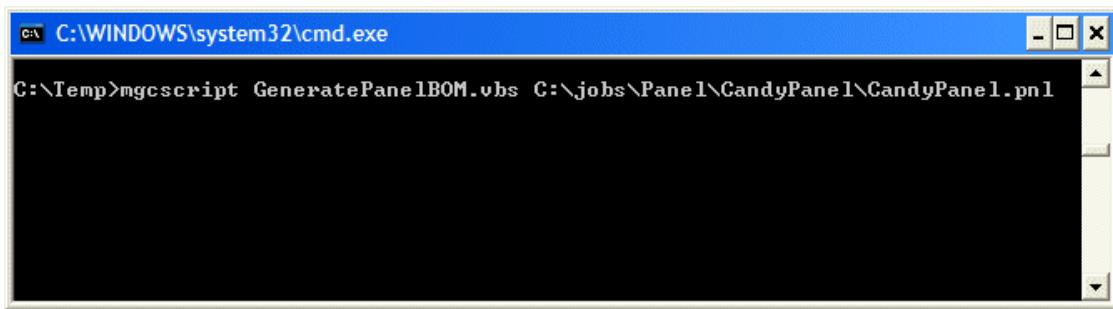


Figure 23-2. Preparing to run the `GeneratePanelBOM.vbs` script.

Once you've keyed-in this command, press the <Enter Key>. Observe that, after a few seconds during which the required information is being gathered and the output file is being generated, the console window prompt reappears, thereby indicating that the script has completed as illustrated in Figure 23-3.

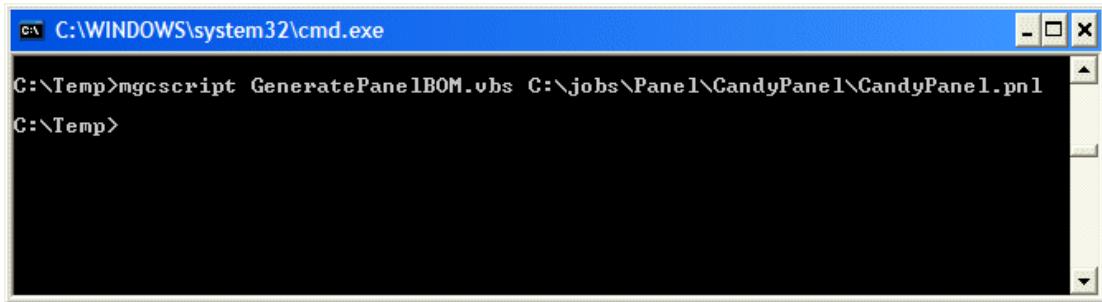


Figure 23-3. The console prompt reappears, thereby indicating that the script has run.

The script generates a text file in Comma Separated Value (CSV) format called *BOM.csv*, which is stored in the *Output* directory/folder of the *CandyPanel* design (see Chapter 18: *Generating a Component CSV file* for more discussions on this format). Assuming a Windows® operating system, the full path to this file is as follows:

C:\jobs\Panel\CandyPanel\Output\BOM.csv

If you were to launch Excel®, use the **File > Open** command, set the **Files of type** field on the resulting **Open** dialog/form to be **Text Files (*.prn; *.txt; *.csv)**, and then open our *BOM.csv* file, the result would be as illustrated in Figure 23.4.

	A	B	C	D	E	F	G	H	I
1	Board References								
2									
3	Instance Name	Board Reference	Location	Full Name					
4	Candy_0	0 <1850,4850>		C:\Jobs\Candy\PCB\Candy.pcb					
5	Candy_1	1 <5350,4800>		C:\Jobs\Candy\PCB\Candy.pcb					
6	Candy_2	2 <4600,4350>		C:\Jobs\Candy\PCB\Candy.pcb					
7	Candy_3	3 <8000,4300>		C:\Jobs\Candy\PCB\Candy.pcb					
8	Candy_4	4 <11250,4250>		C:\Jobs\Candy\PCB\Candy.pcb					
9	Candy_5	5 <8750,5100>		C:\Jobs\Candy\PCB\Candy.pcb					
10									
11									
12									
13	BOM								
14									
15	Part Number	Count	Top Cell	Bottom Cell	Description	References			
16	74LS03	18	DIP14H	None	14 PIN CHIP	0_U4, 0_U2, 0_U3, 1_U4, 1_U2, 1_U3, 2_U5, 1_R7, 2_R7, 3_R7, 4_R7, 5_R7			
17	mux4x2	6	DIP14H	None	4 by 2 mux	0_M2, 1_M2, 2_M2, 3_M2, 4_M2, 5_M2			
18	MAX2003	6	DIP16H	None	Fast-Charge Controller	0_U6, 1_U6, 2_U6, 3_U6, 4_U6, 5_U4			
19	SCONN9	6	CN9DR	None		0_U7, 1_U7, 2_U7, 3_U7, 4_U7, 5_U7			
20	LT1129-5	6	SOL14H	None	voltage regulator, 5-term	0_U5, 1_U5, 2_U5, 3_U5, 4_U5, 5_U5			
21	74LS12	6	SOL14H	None		0_U1, 1_U1, 2_U1, 3_U1, 4_U1, 5_U6			
22	THRMNTC	6	RESH	None	NTC Thermistor	0_TH1, 1_TH1, 2_TH1, 3_TH1, 4_TH1, 5_R7, 1_R7, 2_R7, 3_R7, 4_R7, 5_R7			
23	RES_20K	12	RESH	None	resistor	0_R8, 0_R4, 1_R8, 1_R4, 2_R8, 2_R4, 3_R7, 1_R7, 2_R7, 3_R7, 4_R7, 5_R7			
24	RES_150	6	RESH	None	resistor	0_R6, 0_R2, 1_R6, 1_R2, 2_R6, 2_R2, 3_R7, 1_R7, 2_R7, 3_R7, 4_R7, 5_R7			
25	RES_100K	12	RESH	None	resistor	0_R5, 1_R5, 2_R5, 3_R5, 4_R5, 5_R5			
26	RES_28K	6	RESH	None	resistor				

Figure 23-4. Using Excel to view the BOM.csv file generated by the script.

Observe that this file has two main sections. At the top we have the *Board References* portion, which lists the *Instance Names* associated with each board along with their corresponding *Board References* (these are integer values generated by our script), *Locations* (origins), and *Full Name* (the full path name to the board).

Next, we have the *BOM* portion of the file, which lists the *Part Number* for each type of component, the total *Count* for that component, and other data as shown in Figure 23-4. In

particular, observe the *References* column and note that our script prefixes each component's reference designator with a *Board Reference* number (from the first portion of the file) corresponding to the board on which that component resides.

The Script Itself

The key sections forming this script are as follows:

- **Constant Definitions** Define the constants to be used by the script.
- **Initialization/Setup** Parse the command line and check for the required number of arguments (just one in this case). Create the FabLink XE application and then open the panel design document specified by the argument to the script and license/validate this document.
- **Main Functions and Subroutines** The main functions and subroutines used in the script.
 - BuildBOMDataTree()
 - GenerateBOMFile()
- **Utility Functions and Subroutines** Special "workhorse" routines that are focused on this script.
 - GetCellReferenceName()
 - GetPCBDocObj()
 - GetPartsEditor()
 - ShutdownApps()
- **Helper Functions and Subroutines** Generic "helper" routines that can be used in other scripts.
 - InitTree()
- **Validate Server** The standard function used to license the document and validate the server. (Note that we won't go through this function in this chapter – for more details on this function see *Chapter 3: Running Your First Script*.)

Before we consider the various sections forming the script in more detail, we should remind ourselves that this script requires us to pull information from three different sources (COM servers), which can be complicated. Thus, in order to simplify things, the script is divided into two main routines: first, the *BuildBOMDataTree()* function is used to fetch and store the data in a form that facilitates our working with it; second, the *GenerateBOMFile()* subroutine is used to format the output.

This scheme is advantageous in that it allows us to easily modify the output format. Furthermore, it would be possible to create several output routines – each of which generated a different format – while only having to fetch the original data a single time (fetching the data is the time-consuming part of the process).

Next, we need a logical, hierarchical, and easily accessible way to store all of the data. In order to achieve this, our script is going to use a tree structure, which is created and populated by the *BuildBOMDataTree()* function and then used by the *GenerateBOMFile()* subroutine.

With regard to the tree structure itself, we will implement this using multiple *Dictionary* objects. As we have previously discussed in earlier scripts, anything can be stored in the value field of a *Dictionary* object, including other *Dictionary* objects. In the case of this script, we are going to create a tree that has the following overall form.

```

<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.

```

As we see, the root (top level) of the tree contains two sub-trees: the first contains the board reference information while the second contains the BOM data (these two sub-trees correspond to the two main sections in our output file as illustrated in Figure 23-4. 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 will be used to reference data in the tree structure we are going to create.

```
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 accessing 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 are arbitrary and – since they are never printed by our script – we could have used anything. For the purposes of this script, for example, it would be just as effective to assign numbers to these constants instead of strings. However, if it became 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 would be beneficial to use a descriptive string as is the case here.

Initialization/Setup

On Lines 26, 27, and 28 we add the type libraries for Expedition PCB, FabLink XE, and the Parts Editor.



Note: It is not really necessary to add type libraries for both Expedition PCB and FabLink XE because – as discussed in *Chapter 22: Introducing FabLink XE* – their automation interfaces are the same. Having said this, the reason we did 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 we declare some global variables and establish the units to be used by the script. Observe that we no longer use the name *pcbAppObj* (for a board); instead we use *pnlAppObj* (for a panel). Similarly, instead of using *pcbDocObj* we now use *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 are used to parse the command line arguments; if there are an incorrect number of arguments a *MsgBox()* is used on Line 44 to inform the user as to the correct usage model for this script (see *Chapter 8: Running Scripts from the Command Line* 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
45          <panel_design_name>")
        End If

```

On Line 47 we check to see that we have a valid design name (or at least, that we have something other than an empty string).

```
47  If Not designNameStr = "" Then
```

Assuming that we do have a valid design, then on Line 50 we create a FabLink XE server and on Line 51 we suppress any annoying dialogs because we want our script to run as a batch process in a non-interactive way.

```

49      ' Get the application object.
50      Set pnlAppObj = CreateObject("MGCPBC.FablinkXEApplication")
51      pnlAppObj.Gui.SuppressTrivialDialogs = True

```

On Line 54 we open the panel document.

```

53      ' Open the document
54      Set pnlDocObj = pnlAppObj.OpenDocument(designNameStr)

```

On Line 57 we call our *ValidateServer()* miscellaneous function to validate the panel document.

```

56      ' License the document
57      ValidateServer(pnlDocObj)

```

On Line 60 we set the units.

```

59      ' Set the unitsEnum to match our hard coded values.
60      pnlDocObj.CurrentUnit = unitsEnum

```

On Line 64 we call our main *BuildBOMDataTree()* routine, which opens any relevant Expedition PCB board design documents, reads any required data, and builds our tree structure.

```

62      ' Collect the data
63      Dim bomDataObj
64      Set bomDataObj = BuildBOMDataTree()

```

On Line 67 we call our main *GenerateBOMFile()* routine, which generates the BOM file in the format we require.

```
66      ' Generate the bom file.
67      Call GenerateBOMFile(bomDataObj, pnlDocObj.Path &
                           "Output/BOM.csv")
```

On Line 70 we close the FabLink XE document and on Line 71 we set our *Panel* object to *Nothing*.

```
69      ' Done with Fablink close it
70      Call pnlAppObj.Quit()
71      Set pnlAppObj = Nothing
72
73  End If
```

Build Bom Data Tree Function

Lines 80 through 187 are where we declare the main routine that opens any relevant Expedition PCB board documents, reads any required data, and builds our tree structure.

On Lines 82 and 83 we create the root of our 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")
```

On Lines 86 through 88 we call our *InitTree()* helper function to create the two main sub-trees: one for the board references map and one for the BOM data. (As we shall see, the *InitTree()* helper 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)
```

On Lines 91 and 92 we acquire the collection of boards from the panel.

```
90      ' Get the collection of boards
91      Dim boardColl
92      Set boardColl = pnlDocObj.Bords
```

On Line 95 we use the *Sort* method to sort our collection by performing a standard alpha-numerical sort on their instance names. The reason we 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()
```

On Lines 97 through 104 we instantiate and initialize a bunch of variables.

```

97      Dim boardRef : boardRef = 0
98      Dim fullNameStr : fullNameStr = ""
99      Dim boardObj, pcbDocObj, pcbAppObj, partsEditorObj,
100                                     partsDBObj
101      Set pcbDocObj = Nothing
102      Set pcbDocObj = Nothing
103      Set partsEditorObj = Nothing
104      Set partsDBObj = Nothing
105      Set pcbAppObj = Nothing

```

On Lines 105 through 173 we iterate through our collection of boards. For each board, on Lines 108 through 113 we create a sub-tree corresponding to this board instance using its instance name as the key. Within this sub-tree we add the board's location, its full path name, and we assign 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,
110                                         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

```



Note: Observe the statement on Line 110, for example:

```
110      boardTreeObj(XLOC_KEY) = boardObj.OriginX
```

On previous occasions when we've been working with a *Dictionary* object, we've explicitly used the *Item* property to perform this task. If we had done so here, our statement would have appeared as follows:

```
110      boardTreeObj.Item(XLOC_KEY) = boardObj.OriginX
```

However, since the *Item* property is the default property of a *Dictionary* object, we can omit it if we want, and we did do here because it makes our script easier to read and understand.



Note: Usually in the case of *Collections* and *Collection-like* objects (where *Dictionary* objects may be considered to fall in the category of *Collection-like* objects), the *Item* property is the default and may be omitted.

Similarly, in the case of singular objects, *Name* is the default property and can be omitted. Having said this, although this is generally the standard it cannot be assumed to be true for all cases.

It takes time to open a board document. With some boards, this may account for the vast amount of time consumed by a script that is working on the board. Thus, as a performance enhancement, on Line 115 we check to see if the board document is already open from a previous iteration and – if it is – we don't bother opening it again. Otherwise, on Lines 116 and 117 we instantiate and initialize the Expedition PCB and Parts Editor servers.

```

114
115      If Not fullNameStr = boardObj.FullName Then
116          Call GetPCBDocObj(pcbAppObj, pcbDocObj,
117                               boardObj.FullName)

```

```
117             Call GetPartsEditor(partsEditorObj, partsDBObj,  
118                                         boardObj.FullName)  
119         End If
```

On Lines 122 and 123 we 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
```



Note: There is only a single partition for databases associated with a board; this explains why we just take the first item in the *Partitions* collection on Line 123.

On Lines 126 and 127 we acquire the *Parts* collection from the Expedition PCB *Document* object.

```
125             ' Get all the parts in the pcb design  
126             Dim pcbPartColl  
127             Set pcbPartColl = pcbDocObj.Parts
```



Note: As opposed to the actual parts that are to be found in the Parts DB, the *Parts* collection in the Expedition PCB 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 we are accessing them in this script).

On Lines 130 through 166 we iterate through all of the Expedition PCB *Part* objects. On Line 132, we find the Parts DB *Part* object that corresponds to the current Expedition PCB *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)
```

On Line 135 we call our *InitTree()* helper 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)
```

On Line 138 we add a description to our new sub-tree using the appropriate key.

```
137             ' Set the description  
138             partTreeObj(DESCRIPTION_KEY) =  
                               pdbPartObj.Description
```

On Line 141 we create a new cell sub-tree under the current part sub-tree. On Line 142 we call our *GetCellReferenceName()* utility routine to acquire the top cell name for the current part and then add it to our cell sub tree. Similarly, on Line 143 we acquire the bottom cell name for the current part and add it to our cell sub tree.

```

140          ' Initialize and/or get cell tree for this part
141          Set cellTreeObj = InitTree(partTreeObj,
142                                      CELL_KEY)
142          cellTreeObj(TOP_KEY) =
143              GetCellReferenceName(pdbPartObj,
143                                     epdbCellRefTop)
143          cellTreeObj(BOTTOM_KEY) =
144              GetCellReferenceName(pdbPartObj,
144                                     epdbCellRefBottom)

```

On Lines 146 and 147 we create a new part references sub-tree under the current part sub-tree (this sub-tree will be used to hold 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,
147                                         PARTREFS_KEY)

```

On Lines 150 and 151 we acquire the collection of *Component* objects that reference the current part.

```

150          Dim compColl
151          Set compColl = pcbPartObj.Components
152          Dim compObj, compObjTree

```

On Lines 153 through 165 we iterate through our collection of components. On Line 155 we create 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 (you may refer back to the discussions on Figure 23-4 to better visualize what we are doing here).

```

149          ' Get all instances of this part.
153          For Each compObj In compColl
154              ' Initialize tree for this component
155              Set compObjTree = InitTree(refDesTreeObj,
155                                         boardRef & "_" & compObj.Name)

```

On Lines 158 and 159 we 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) =
159              boardObj.TransformPointX(compObj.PositionX,
159                                         compObj.PositionY)
159          compObjTree(YLOC_KEY) =
159              boardObj.TransformPointY(compObj.PositionX,
159                                         compObj.PositionY)

```



Note: Observe the use of the *TransformPointX* and *TransformPointY* methods on Lines 158 and 159, respectively. These methods are used to transform coordinates from their location on the original board to their corresponding coordinates on the panel. (See also the discussions on the related *TransformPointsArray* method in *Chapter 24* for a more detailed explanation of this "transform" concept.)

On Lines 160 through 164 we check to see if this component is on the top of the board and – if so – we add the string "Top" to the reference designator sub-tree; otherwise we add 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
```

On Line 170 we increment our 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
```

On Line 175 we call our *ShutdownApps()* utility routine, which will close the Expedition PCB and Parts Editor servers for us.

```
174          ' Close Expedition and the Parts Editor.
175      Call ShutdownApps(pcbAppObj, partsEditorObj)
```

On Lines 178 through 181 we explicitly release any references to the Expedition PCB 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, on Line 184 we set 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
```

Generate BOM Output Subroutine

Lines 191 through 250 are where we declare the main routine that accepts our tree structure and a string containing the name of the CSV file we want 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)
```

On Lines 193 through 196 we create the output file (see the *File I/O* topic in *Chapter 12: Basic Building-Block Examples* 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)
```

On Line 199 we write a header for the board references section of the output file; on Line 200 we use the *WriteBlankLines* method to output a blank line; and on Line 201 we write 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")
```

On Lines 204 and 205 we acquire the board references sub-tree.

```
203      ' Get the board refs sub tree.  
204      Dim boardRefsTreeObj  
205      Set boardRefsTreeObj = dataTreeObj(BOARDREFMAP_KEY)
```

On Lines 209 through 213 we iterate through all of the keys in the board references sub-tree (each key corresponds to a board instance name). On Lines 210 and 211 we 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)
```

On Line 212 we write a single line to our output file; this line contains all of the comma-separated data values in which we are interested (see *Chapter 13* and *Chapter 18* for detailed discussions on formatting output strings in general; also, see *Chapter 18* for discussions on generating comma-separated value output strings in particular).

```
212          fileObj.WriteLine(keyStr & ", " &  
                           boardTreeObj(BOARDREF_KEY) & ", ""<" &  
                           boardTreeObj(XLOC_KEY) & ", " &  
                           boardTreeObj(YLOC_KEY) & "> "", " &  
                           boardTreeObj(FULLNAME_KEY))  
213      Next
```

On Line 215 we use the *WriteBlankLines* method to output three blank lines.

```
215      Call fileObj.WriteBlankLines(3)
```

On Line 218 we write a header for the BOM section of the output file; on Line 219 we use the *WriteBlankLines* method to output a blank line; and on Line 220 we write 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")
```

On Lines 233 and 234 we acquire the BOM data sub-tree.

```
222     ' Get the BOM data sub tree.
223     Dim bomTreeObj
224     Set bomTreeObj = dataTreeObj(BOMDATA_KEY)
```

On Lines 227 through 246 we iterate through all of the keys in the BOM data sub-tree (each key corresponds to a part name). On Lines 228 and 229 we 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)
```

In the case of the BOM format we decided to use in this script, we chose to ignore specific information relating to individual component instances; instead, we simply count and list the component reference designators. Note, however, that our data tree does contain instance-specific data, so it would be relatively easy to modify this routine – or create a new routine – that outputs a format containing this data.

On Lines 235 through 242 we iterate through all of the keys in the part references sub-sub-tree associated with the current part sub-tree. On Line 236 we count 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
```

On Lines 237 through 241 we add the reference designator name to our reference designator string. If this is the first entry in the reference designator string, then on Line 238 we simply set the string to contain this entry; otherwise, on Line 240 we append 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
```

On Line 245 we write the BOM line for the current part to our output file; this line contains all of the comma-separated data in which we are interested (once again, see *Chapter 13* and *Chapter 18* for detailed discussions on formatting output strings in general; also, see *Chapter 18* for discussions on generating comma-separated value output strings in particular).

```
244             ' Write the parts line.
245             fileObj.WriteLine(""" "" & keyStr & """ , " &
```

```

        refDesCntInt & ", "" &
        partTreeObj(CELL_KEY)(TOP_KEY) & " , " &
        partTreeObj(CELL_KEY)(BOTTOM_KEY) & " , " &
        partTreeObj(DESCRIPTION_KEY) & " , " &
        refDesStr & " ")

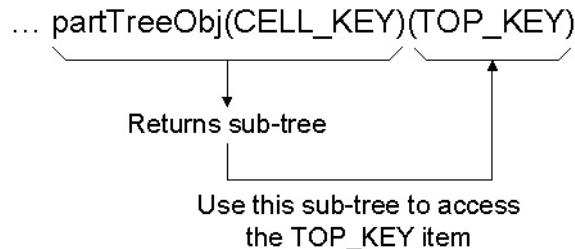
```

246

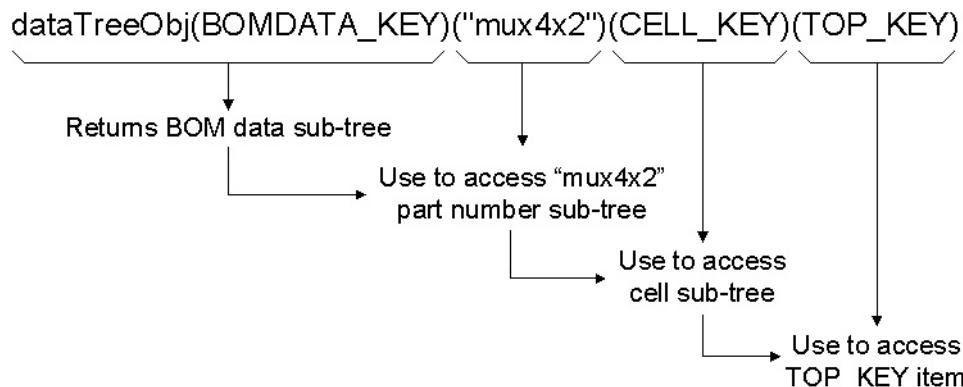
[Next](#)



Note: Observe how nested trees and items in trees can be accessed as illustrated on Line 245 above; for example, consider the following:



Actually, the above example starts in the middle of the tree. As an alternative scenario, we can access any item from the top of the tree using the list of keys that lead to that item. For example, if we wanted to get the top cell value for part *mux4x2*, we could do so using the following statement.



We should also note that values can be assigned to locations in the tree using this syntax.

On Line 249 we close the output file.

```

248      ' Close the output file.
249      Call fileObj.Close()
250  End Sub
  
```

Utility Functions and Subroutines

This is where we declare some utility routines. These routines are similar in context to our generic "helper" routines (see the next topic), except that they are specific to this script and may not be used elsewhere without some "tweaking".

GetCellReferenceName() Function

This routine accepts two parameters: a Parts Editor *Part* object and a cell reference type enumerate that specifies "top" or "bottom". On Line 263 we acquire 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)
```

On Line 265 we check to see if there are any cells in our collection and – if there are – on Line 266 we return the name of the first item; otherwise, on Line 268 we return a string containing the word "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 Expedition PCB *Application* object, an Expedition PCB *Document* object, and an Expedition PCB design document path/filename (in this case we are using the first two parameters – the *Application* and *Document* objects – as 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)
```

On Lines 279 through 281 we check to see if a document is already open and – if so – we close it.

```
278      ' Close the document if one is open.
279      If Not pcbDocObj Is Nothing Then
280          pcbDocObj.Close()
281      End If
```

On Lines 284 through 288 we check to see if the application has already been created and – if not – we create it (observe that on Line 286 we suppress any annoying messages coming from this server).

```
283      ' Create the application if not already created.
284      If pcbAppObj Is Nothing Then
285          Set pcbAppObj =
              CreateObject("MGCPBC.ExpeditionPCBApplication")
```

```
286     pcbAppObj.Gui.SuppressTrivialDialogs = True  
287 End If
```

On Line 290 we open the document and on Line 291 we validate it.

```
289     ' Open and license the document  
290     Set pcbDocObj = pcbAppObj.OpenDocument(docNameStr)  
291     ValidateServer(pcbDocObj)
```

On Line 294 we set the units associated with this document and then on Line 296 we exit 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 Expedition PCB 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)
```

On Lines 304 through 306 we check to see if there is a *PartsEditorDlg* object and – if not – we create one.

```
303     ' Open the parts editor  
304     If partsEditorObj Is Nothing Then  
305         Set partsEditorObj =  
306             CreateObject("MGCPCLibraries.PartsEditorDlg")  
307     End If
```

On Line 308 we open the required Parts Editor database.

```
308     Set partsDBObj =  
309         partsEditorObj.OpenDatabase(docNameStr, True)  
310     End Sub
```

ShutdownApps() Subroutine

This routine accepts two parameters: an Expedition PCB *Application* object and a Parts Editor *PartsEditorDlg* object. On Lines 315 and 316 we 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 is where we declare some generic "helper" routines. These routines are not focused on this script per se; they can easily be used in other scripts.

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 parentTreeObj under keyStr
323  ' parentTreeObj - Dictionary Object
324  ' keyStr - String
325  Function InitTree(parentTreeObj, keyStr)
```

On Lines 327 through 329 we check to see if a sub-tree already exists under this key and – if not – we create it.

```
326      ' Create the child tree if it doesn't exist
327      If Not parentTreeObj.Exists(keyStr) Then
328          Set parentTreeObj(keyStr) =
329              CreateObject( "Scripting.Dictionary" )
            End If
```

On Line 332 we set 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 above and save it out as *GeneratePanelBOM.vbs*.
- 2) Close any instantiations of Expedition PCB, FabLink XE, and the Parts Editor that are currently running.
- 3) Launch a console (terminal) window and set its context to C:\Temp. 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 *Chapter 8: Running Scripts from the Command Line* 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) Look in the *Output* directory/folder in the FabLink XE *CandyPanel* design and observe that it now contains a *BOM.csv* file. Launch Excel®, use it to view the contents of this file, and check that everything is as expected.

Enhancing the Script

There are a number of ways in which this script could be enhanced. Some suggestions are as follows (it would be a good idea for you to practice your scripting skills by implementing these examples):

- For each application (Expedition PCB, FabLink XE, and the Parts Editor), modify the script such that it checks to see if the application is currently running and – if it is – make the script attach to the existing process.
- As opposed to simply generating a CSV file, the script could be modified to automatically launch and populate an appropriate spreadsheet editor, such as Excel® on Windows®; this would allow you to control the formatting of the output, such as varying font sizes, using bold text, and selecting different colors. (See also *Chapter 17: Integration with Excel*.)

Chapter 24: Generating a Panel Side View

Introduction

- Overview:** This script creates a graphic representation depicting the side view of a panel.
- Approach:** To access board information from FabLink XE, to access component outline and height information from Expedition PCB, and to create the resulting side view on a user layer in FabLink XE.
- Points of Interest:**
- Using a FabLink XE script to access data in Expedition PCB
 - Transforming component positional data (in the form of points arrays) from the board in Expedition PCB to their corresponding locations on the panel in FabLink XE.
- Items Used:**
- Expedition PCB Server
 - TransformPointsArray Method (FabLink XE)

Before We Start

In order to get a feel for the way in which this script is going to perform its magic, launch FabLink XE and open our *CandyPanel* panel design as illustrated in Figure 24-1.

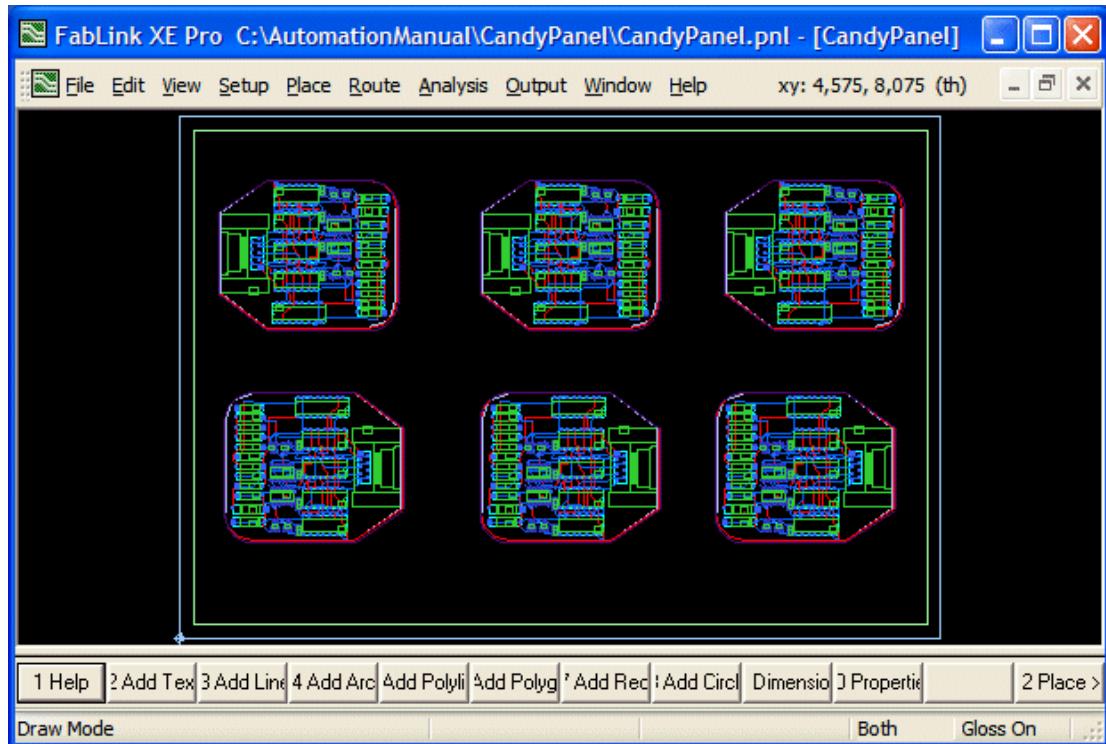


Figure 24-1. The panel in FabLink XE prior to running the script.

Now, assume that we've already created a script called *PanelSideView.vbs*, and that we drag-and-drop this script into the middle of the FabLink XE design area. The result will be to generate a vertical profile of the panel, and to present this profile to the right of the panel (the term "vertical" is used here in the context of the way in which the panel is presented in the FabLink XE GUI).

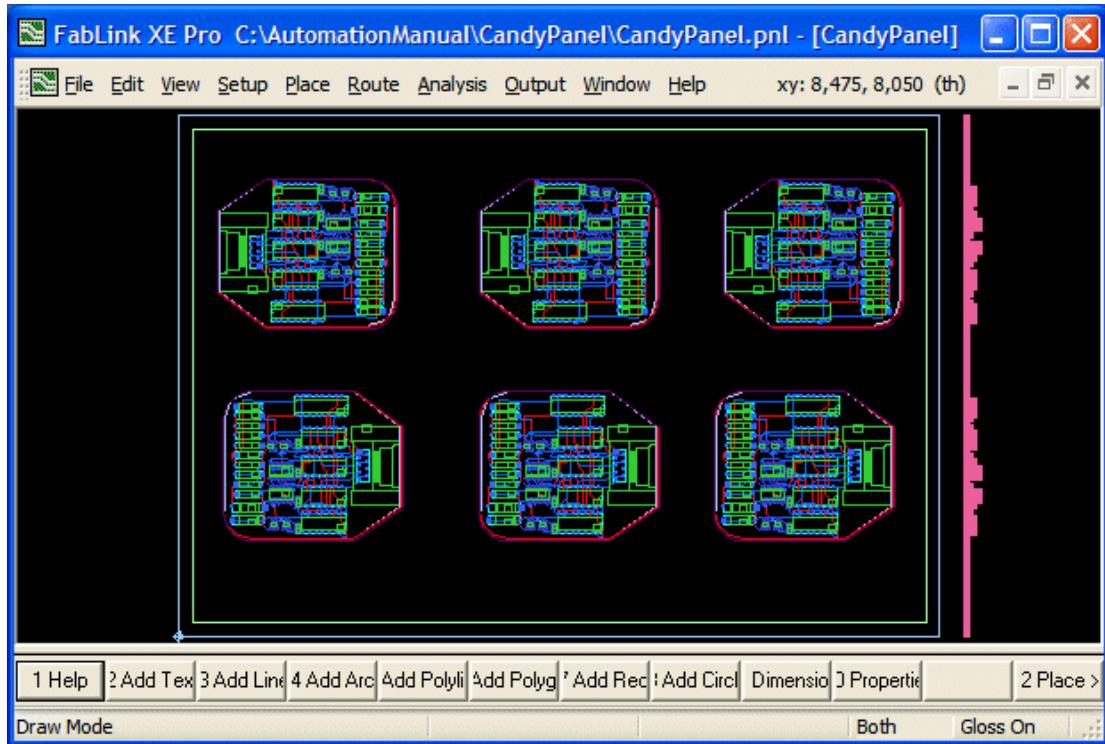


Figure 24-2. The panel in FabLink XE after the script has been run.

The tall thin 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, our script is going to acquire this data from the board design in Expedition PCB.

Now assume that we zoom in on the right-hand-side of the panel as illustrated in Figure 24-3. Observe that, due to the way in which we are going to create our script, the components on the top of the panel are presented to the right of the panel.

In fact, the *Candy.pcb* boards comprising our *CandyPanel.pnl* panel have components only on their top sides. However, we are going to create our script in such a way that it will be capable of handling components on both the top and bottom sides (any bottom-side components would, of course, appear to the left of the vertical panel graphical object).

Furthermore, the way in which we are going to create our script means that it will be capable of handling both homogeneous panels (comprising multiple copies of the same board design) and heterogeneous panels (comprising a mixture of different boards).

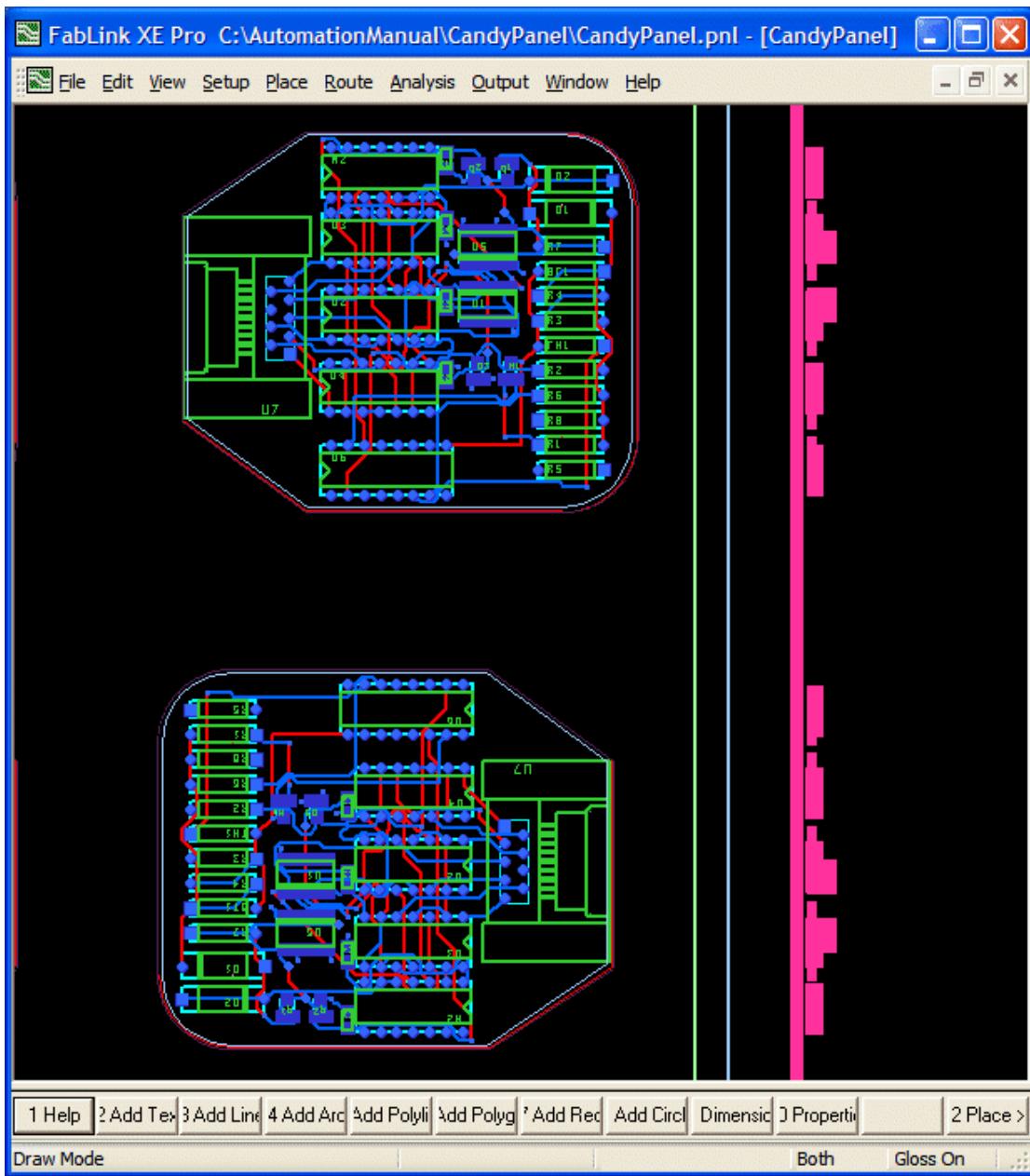


Figure 24-3. Zooming in on the right-hand-side of the panel.

The Script Itself

The key sections forming this script are as follows:

- **Constant Definitions** Define the constants to be used by the script.
 - **Initialization/Setup** Add type libraries, declare global variables, and perform any additional setup associated with this script.
 - **Main Subroutine** The main routine used in the script.
 - CreatePanelSideview()

- **Utility Functions and Subroutines** Special "workhorse" routines that are focused on this script.
 – AddPanel()
 – AddComponents()
 – AddComponent()
 – GetPCBDocObj()
- **Helper Functions and Subroutines** Generic "helper" routines that can be used in other scripts.
 – GetBoardThickness()
 – ConfigureUserLayer()
 – GetYExtreme()
- **Validate Server** The standard function used to license the document and validate the server. (Note that we won't go through this function in this chapter – for more details on this function see *Chapter 3: Running Your First Script*.)

Constant Definitions

Lines 9 through 13 declare the constants we will use throughout 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

On Lines 16 and 17 we add the type libraries for Expedition PCB and FabLink XE.



Note: As we discussed in *Chapter 23*, it is not really necessary to add type libraries for both Expedition PCB and FabLink XE because their automation interfaces are the same. The reason we do add both type libraries here is that this is good programming practice.

```

15  ' Add any type libraries to be used.
16  Scripting.AddTypeLibrary( "MGCPBC.FablinkXEApplication" )
17  Scripting.AddTypeLibrary( "MGCPBC.ExpeditionPCBApplication" )

```

On Lines 20 through 23 we instantiate and initialize our 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

```

On Lines 26 through 29 we acquire the FabLink XE *Application* and *Document* objects.

```

25  ' Get the application object.
26  Set pnlAppObj = Application
27
28  ' Get the active document

```

```
29 Set pnlDocObj = pnlAppObj.ActiveDocument
```

On Line 32 we validate the document.

```
31 ' License the document
32 ValidateServer(pnlDocObj)
```

On Line 35 we start a transaction to group all of the changes we are about to make into a single **Undo** level.

```
34 ' Start a transaction to group changes for undo
35 Call pnlDocObj.TransactionStart()
```

On Line 38 we call our main *CreatePanelSideview()* routine.

```
37 ' Create the side view.
38 Call CreatePanelSideView()
```

On line 41 we end our transaction.

```
40 ' End the transaction
41 Call pnlDocObj.TransactionEnd()
```

Finally, on Line 44 we display a message in the status bar to inform the user as to the task we've just performed.

```
43 ' Let the user know we are done
44 pnlAppObj.Gui.StatusBarText("Side view created")
```

Generate Panel Side View Subroutine

Lines 50 through 90 are where we declare the main routine that is used to generate the panel side view. On Lines 52 and 53 we 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.Borads
```

On Line 56 we use the *Sort* method to sort our collection by performing a standard alpha-numerical sort on their instance names. The reason we 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()
```

On Lines 60 through 65 we instantiate and initialize some 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
```

On Lines 66 through 81 we iterate through the collection of boards. On Line 68 we check to see if the current board is already open in Expedition PCB; if not, on Line 69 we call our *GetPCBDocObj()* utility routine to create (launch) Expedition PCB 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,
70                                  boardObj.FullName)
71          fullNameStr = boardObj.FullName
71      End If
```

On Lines 74 through 76 – if we haven't already done so – we call our *AddPanel()* utility routine to add the tall thin graphic element representing the panel to our user layer. The way we determine whether or not we have already added this element is to check the *topSurfaceXReal* variable that we initialized to 0 on Line 60; if this is non-zero, then this means that we've previously called the *AddPanel()* utility routine, 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,
76                            pcbDocObj, topSurfaceXReal, bottomSurfaceXReal)
76      End If
```

Once we've added the panel, we know the offset corresponding to the thickness of the panel that we need to apply when adding the graphics corresponding to the components. On line 79 we call our *AddComponents()* utility routine 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,
80                               topSurfaceXReal, bottomSurfaceXReal)
81      Next
```



Note: The way in which we implemented the loop defined by Lines 66 through 81 is not as performance-efficient as might desire. There is a small modification we could make to dramatically increase the performance of the script. Can you work out what this modification might be? (**Hint:** This relates to the *In-Process Client* and *Out-of-Process Client* discussions in *Chapter 1*; further musings on this topic – including a somewhat related solution – are provided in *Chapter 25*).

On Line 84 we use the *Quit* method on Expedition PCB.

```
83          ' Close Expedition
84          pcbAppObj.Quit()
```

On Lines 87 and 88 we release all references to the Expedition PCB *Application* and *Document* objects., and then on Line 90 we exit the routine.

```
86          ' Explicitly release all references
87          Set pcbDocObj = Nothing
88          Set pcbAppObj = Nothing
89
90      End Sub
```

Utility Functions and Subroutines

This is where we declare some utility routines. These routines are similar in context to our generic "helper" routines (see the next topic), except that they are specific to this script and may not be used elsewhere without some "tweaking".

AddPanel() Subroutine

Lines 100 through 121 declare the routine that is used to add the side view graphic element corresponding to the panel to a user layer in FabLink XE. There are four parameters to this routine: a FabLink XE *PanelOutline* object, an Expedition PCB *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 will be set by this routine).

```
95  ' Adds the panel profile to the side view.
96  ' pnlOutlineObj - PanelOutline Object
97  ' pcbDocObj - Document Object (Expedition PCB)
98  ' topSurfaceXReal - Real (In/Out)
99  ' bottomSurfaceXReal - Real (In/Out)
100 Sub AddPanel(pnlOutlineObj, pcbDocObj, topSurfaceXReal,
                 bottomSurfaceXReal)
```

On Lines 102 and 103 we 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
```

On Line 107 we set 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 we declared on line 9).

```
105      ' Determine the bottom surface of the
106      ' board for the side view.
107      bottomSurfaceXReal = extremaObj.MaxX + SIDE_VIEW_OFFSET
```

On Line 110 we calculate 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)
```

On Lines 113 and 114 we generate a points array to represent the panel side view graphic using the *CreateRectXYR* method (see the *Creating Objects with Geometries* topic in *Chapter 12: Basic Building-Block Examples* for more details on creating and using points arrays; see *Chapter 14: Documenting the Layer Stackup* for more details on using the *CreateRectXYR* method).

```
112      ' Create a rectangle to represent the board.
113      Dim ptsArr
114      ptsArr =
                  pnlAppObj.Utility.CreateRectXYR(bottomSurfaceXReal,
                                                 extremaObj.MinY, topSurfaceXReal, extremaObj.MaxY)
```

On Line 118 we call our *ConfigureUserLayer()* helper routine 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)

```

On Line 119 we use the *PutUserLayerGfx()* 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 is used to add the side view graphic elements corresponding to the components on a board to our user layer in FabLink XE. There are four parameters to this routine: a FabLink XE *Board* instance, the Expedition PCB *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 (Expedition PCB)
126  ' topSurfaceXReal - Real (In/Out)
127  ' bottomSurfaceXReal - Real (In/Out)
128  Sub AddComponents(boardObj, pcbDocObj, topSurfaceXReal,
                           bottomSurfaceXReal)

```

On Lines 131 through 133 we iterate through each component in the Expedition PCB *Document* object; for each component we call our *AddComponent()* helper routine, which will actually create 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 is used to actually create and display the side view graphic element(s) corresponding to a particular component on our user layer in FabLink XE. There are four parameters to this routine: a FabLink XE *Board* (panel) object, an Expedition PCB *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 (Expedition PCB)
139  ' topSurfaceXReal - Real (In/Out)
140  ' bottomSurfaceXReal - Real (In/Out)
141  Sub AddComponent(boardObj, compObj, topSurfaceXReal,
                           bottomSurfaceXReal)

```

On Lines 144 through 146 we check to see whether or not this component has been placed and – if so – we exit 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, on Lines 148 and 149 we instantiate three local variables.

```

148      Dim heightReal, underSideSpaceReal
149      Dim placeOutlineObj

```

On Lines 150 through 183 we 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

```

On Line 152 we use 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

```

On Line 153 we use the *UndersideSpace* property of the *PlacementOutline* object to determine the space (gap) that we need to have under the component.

```

153          underSideSpaceReal = placeOutlineObj.UndersideSpace

```

Now, before we proceed, there's something we need to discuss. Consider a component on a board in Expedition PCB; the X/Y values specifying the location of the component on the board will be relative to the board's origin as illustrated in Figure 24-4(a).

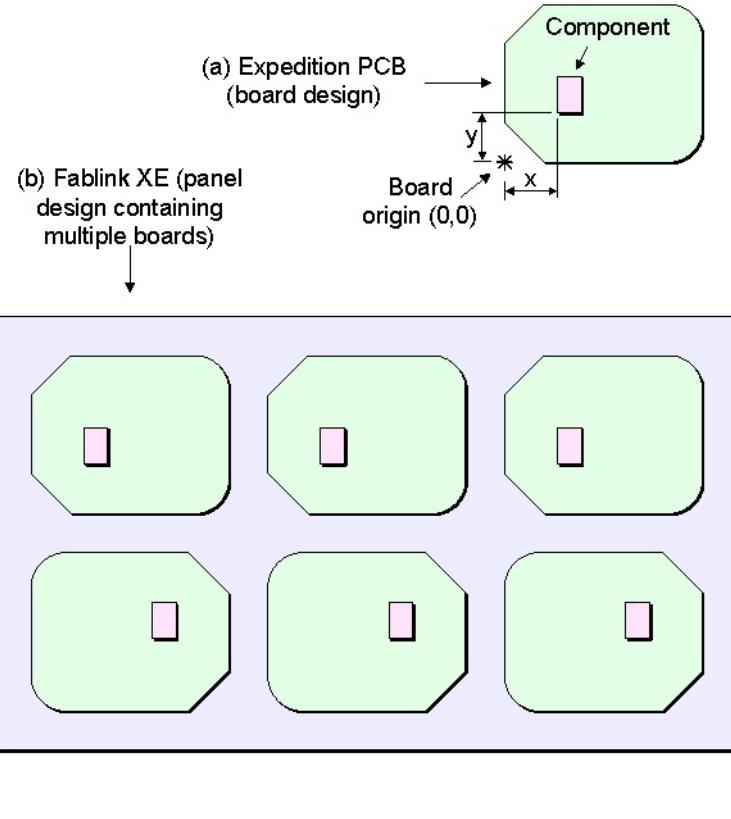


Figure 24-4. Transforming values from the board domain to the panel domain.

However, in order to construct the panel side view graphic associated with a particular component, we want to transform 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 – we need to account for any rotation of the board on the panel (observe that the bottom three boards on the panel shown in Figure 24-4(b) have been rotated by 180 degrees). Fortunately, the FabLink XE automation interface provides a *TransformPointsArray* method for just this purpose.

On Line 160 we acquire 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
```

On Line 163 we use the *TransformPointsArray* method associated with the FabLink XE *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)
```

On Lines 166 and 167 we use our *GetYExtrema()* helper function 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)
```

On Lines 170 through 176, we create the graphic elements to represent this component. On Line 170 we check to see if the component is on layer 1 (the top of the board/panel) and – if so – on Line 172 we create a points array corresponding to the component being on top of the board/panel; otherwise, on Line 175, we create 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
```

On Lines 179 through 181 we acquire the user layer upon which we want to display our side view and we add this graphic element to that layer.

```

178      ' Add the component side view to the panel.
179      Dim userLayerObj
180      Set userLayerObj =
181          ConfigureUserLayer(SIDE_VIEW_USER_LAYER)
181      Call pnlDocObj.PutUserLayerGfx(userLayerObj, 0,
181                                     UBound(sideViewPntsArr, 2)+1, sideViewPntsArr,
181                                     True, Nothing)
182      End If
183      Next
184  End Sub

```

GetPCBDocObj() Subroutine

Lines 191 through 210 declare the routine that is used to acquire an Expedition PCB Document object. There are three parameters to this routine: an Expedition PCB Application object, an Expedition PCB 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)

```

On Lines 193 through 195 we check to see if a document is already open and – if so – we close it.

```

192      ' Close the document if one is open.
193      If Not pcbDocObj Is Nothing Then
194          pcbDocObj.Close()
195      End If

```

On Lines 198 through 201 we check to see if the application has already been created and – if not – we create it (observe that on Line 200 we suppress any annoying messages coming from this server).

```

197      ' Create the application if not already created.
198      If pcbAppObj Is Nothing Then
199          Set pcbAppObj =
200              CreateObject("MGCPCB.ExpeditionPCBAplication")
200          pcbAppObj.Gui.SuppressTrivialDialogs = True
201      End If

```

On Line 204 we open the document and on Line 205 we validate it.

```

203      ' Open and license the document
204      Set pcbDocObj = pcbAppObj.OpenDocument(docNameStr)
205      ValidateServer(pcbDocObj)

```

On Line 208 we set the units associated with this document and then on Line 210 we exit this routine.

```

207      ' Set the units.
208      pcbDocObj.CurrentUnit = unitsEnum
209
210  End Sub

```

Helper Functions and Subroutines

This is where we declare some generic "helper" routines. These routines are not focused on this script per se; they can easily be used in other scripts.

GetBoardThickness() Function

Lines 217 through 230 declare the routine that is used to acquire the thickness of the board/panel. The only parameter to this routine is an Expedition PCB *Document* object.

```
215  ' Returns the total thickness of the board.
216  ' pcbDocObj - Document Object (Expedition PCB)
217  Function GetBoardThickness(pcbDocObj)
```

On Line 220 we initialize the return value to 0.

```
219      ' Initialize the return value.
220      GetBoardThickness = 0
```

On lines 225 through 228 we iterate through the *Layer* objects summing the thickness of all of the layers and we return 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 +
228              lyrObjObj.LayerProperties.Thickness
229
230  End Function
```

ConfigureUserLayer() Function

Lines 235 through 253 declare the routine that is used to create and configure 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)
```

On Line 238 we acquire the user layer if it exists.

```
236      ' See if our user layer already exists.
237      Dim usrLyrObj
238      Set usrLyrObj = pnlDocObj.FindUserLayer(userLayerNameStr)
```

On Lines 240 through 246 we create the user layer if it doesn't exist; otherwise we clear any graphics from the layer if it already exists.

```
240      If usrLyrObj Is Nothing Then
241          ' It doesn't exist, create it.
242          Set usrLyrObj =
243              pnlDocObj.SetupParameter.PutUserLayer(userLayerNameStr)
244      Else
245          ' It does exist, remove any gfx on the user layer
246          pnlDocObj.UserLayerGfxs(, userLayerNameStr).Delete
247      End If
```

On Line 249 we ensure that display of the user layer is turned on.

```
248      ' ensure the user layer is turned on
249      pnlDocObj.ActiveView.DisplayControl.UserLayer
              (userLayerNameStr) = True
```

On Line 252 we set 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 is used to determine 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 will be 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)
```

On Lines 261 and 262 we 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)
```

On Lines 265 through 273 we 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* our current minimum Y, then we reset our minimum Y to this new value; otherwise, if the new value is *greater than* our current maximum Y, then we reset our 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 above (including the license server function which is not shown above) and save it out as *PanelSideView.vbs*.
- 2) Close any instantiations of Expedition PCB and FabLink XE that are currently running, and then launch a new instantiation of FabLink XE.
- 3) Open the *CandyPanel.pnl* design.

-
- 4) Drag-and-drop the *PanelSideView.vbs* script into the middle of the FabLink XE 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 would be 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 FabLink XE GUI. One modification might be to generate a horizontal profile, and to display this profile below the panel in FabLink XE.
- Another modification would be 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 *Chapter 14: 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, looking something like crosshairs on a gun sight (see *Chapter 14: 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 would correspond to cross-sectional "slices" associated with the current positions of the vertical and horizontal white (motion graphics) lines. Moving the mouse would cause the vertical and horizontal cross-sectional profiles to be updated on-the-fly. What do you think? Are you ready for the challenge?

Chapter 25: Creating a Via Mask

Introduction

- Overview:** This script creates a mask for the top pads of specific vias on a FabLink XE panel.
- Approach:** To access board information from FabLink XE, to access via information from Expedition PCB, and to use this information to create a mask on a user layer in FabLink XE.
- Points of Interest:**
- Using a FabLink XE script to access data in Expedition PCB
 - Transforming via pad data (in the form of points arrays) from the board in Expedition PCB to their corresponding locations on the panel in FabLink XE.
 - Using the Mask Engine to generate shapes
- Items Used:**
- Expedition PCB Server
 - Mask Engine Server
 - TransformPointsArray Method (FabLink XE)
 - AbsorbHoles Method (Mask Engine)

Before We Start

In order to get a feel for the way in which this script is going to perform its magic, launch FabLink XE and open our *CandyPanel* panel design as illustrated in Figure 25-1.

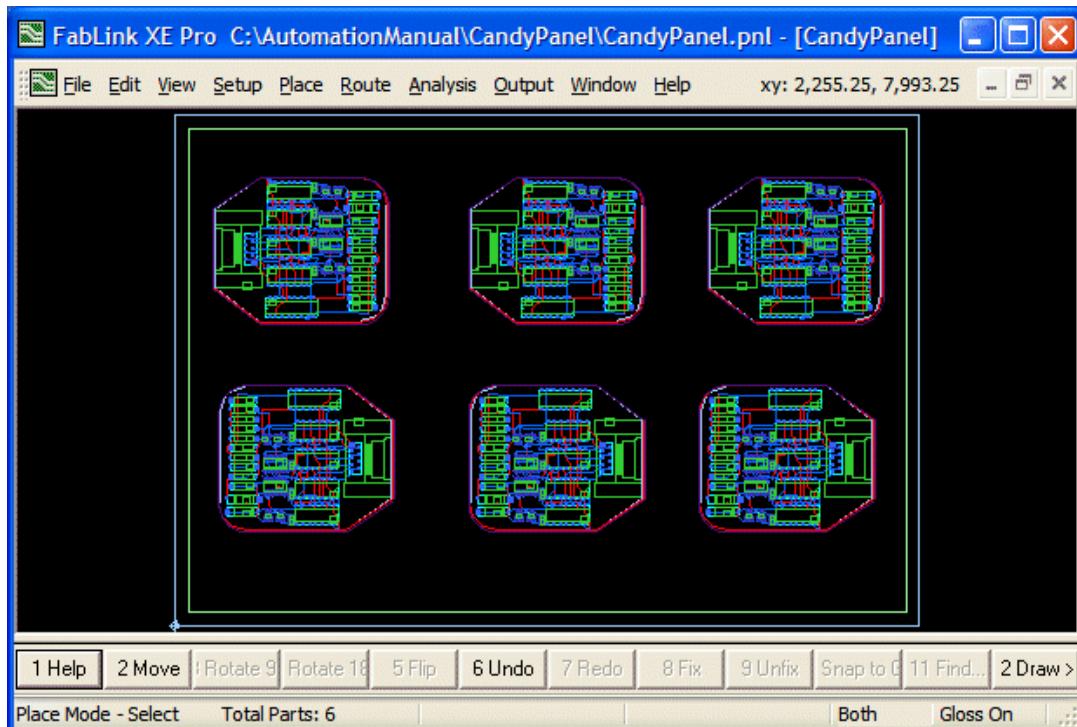


Figure 25-1. The panel in FabLink XE prior to running the script.

Let's suppose that we were to zoom-in on one of the boards on the panel; the board contains a variety of different vias as illustrated in Figure 25-2. In particular, observe the via located toward the bottom left-hand corner of component U5. We (the authors) happen to know that this via uses the padstack of type "VIA31"; furthermore, we also know that there are four such vias on each *Candy* board instance on the panel.

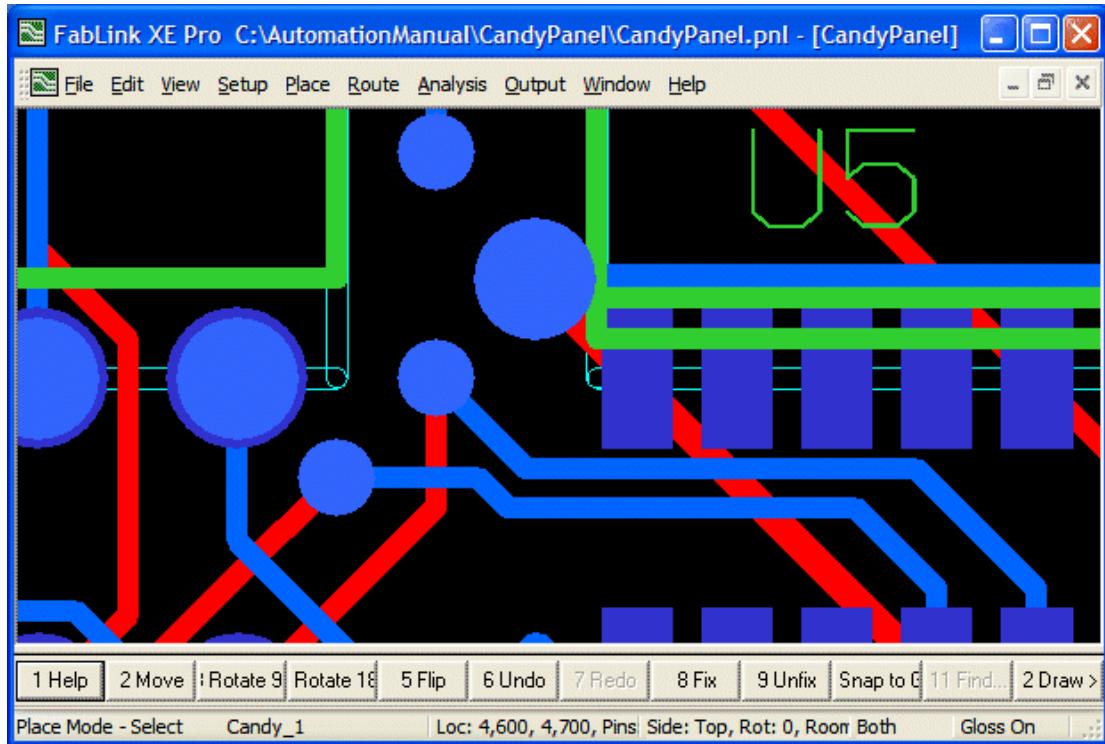


Figure 25-2. Zooming-in prior to running the script.

Now, assume that we've already created a script called *CreateViaMask.vbs*, and that we drag-and-drop this script into the middle of the FabLink XE design area (Figure 25-3).

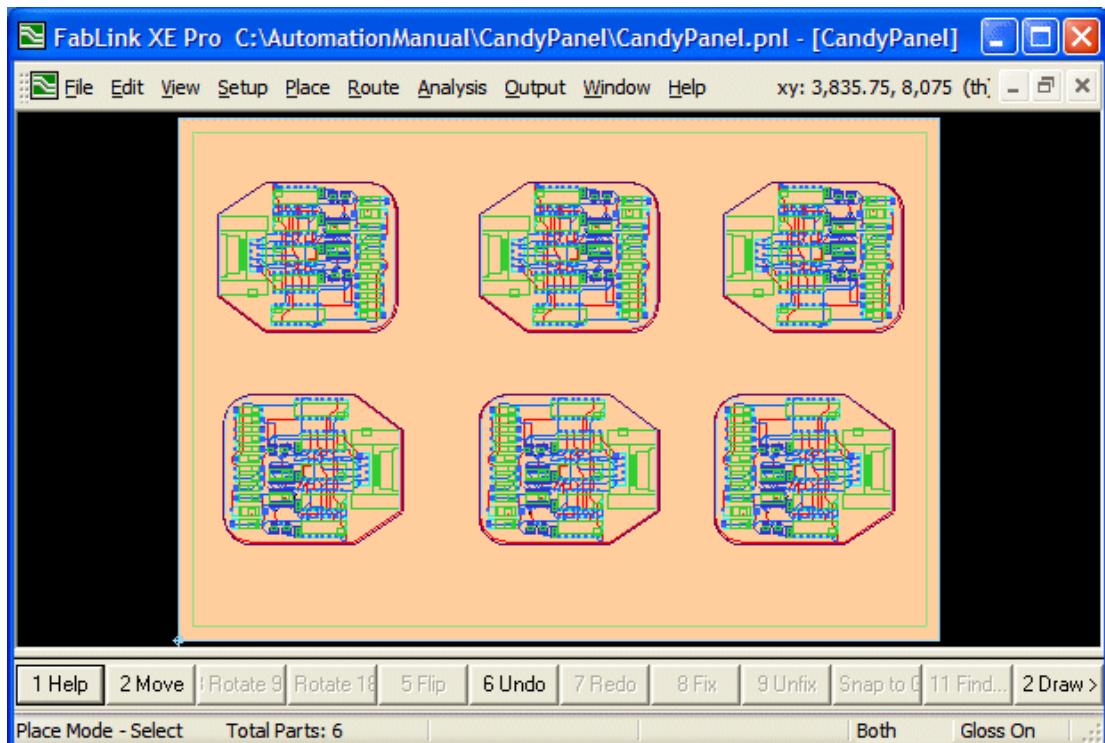


Figure 25-3. The panel in FabLink XE after the script has been run.

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). Suppose we zoom-in again (Figure 25-4).

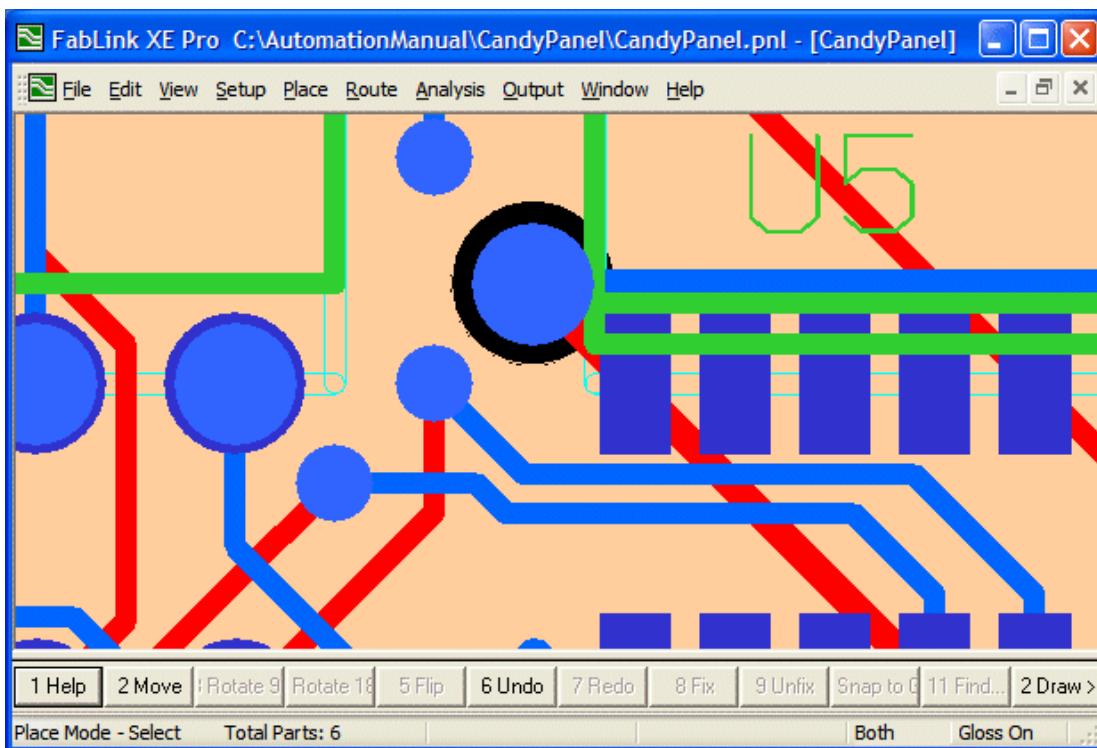


Figure 25-4. Zooming-in after running the script.

Now consider what we would see if we disabled the display of everything except the mask and zoomed-in on a single board instance on the panel. The result would be as illustrated in Figure 25-5 (as we noted above, there are four vias that use padstack "VIA31" on each board instance).

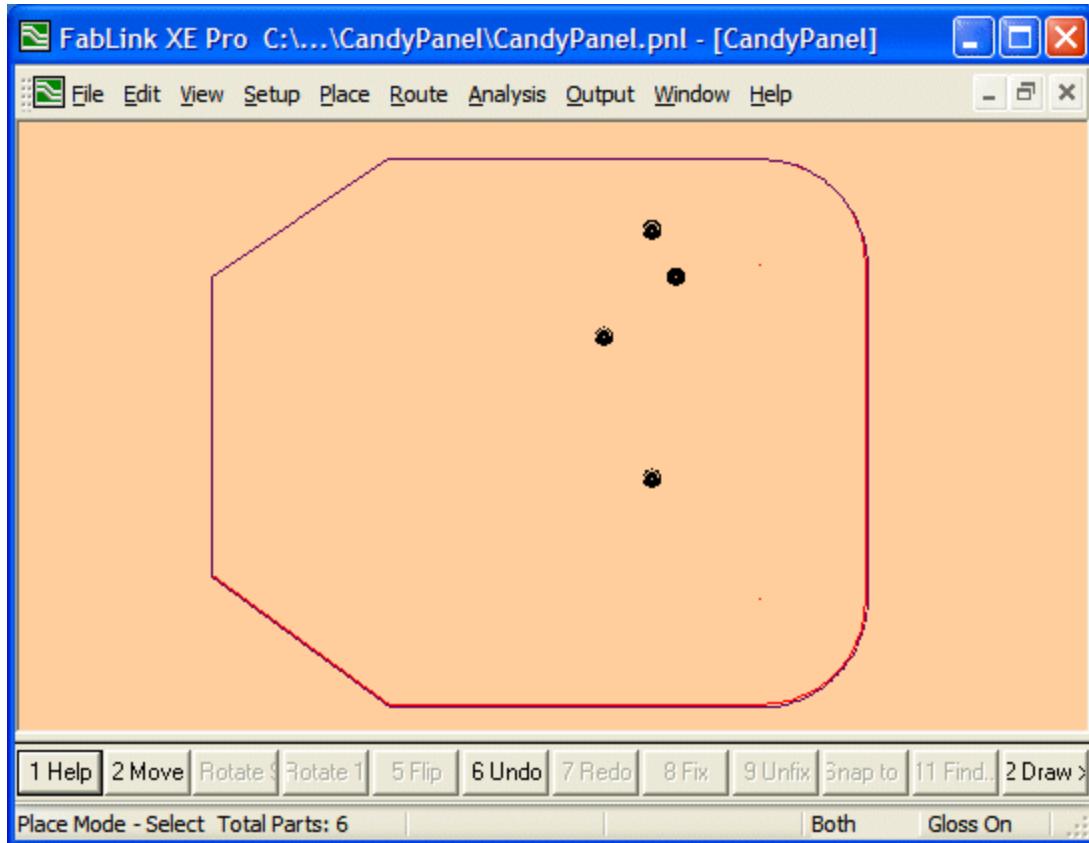


Figure 25-5. Turning everything off except the mask.

The Script Itself

The key sections forming this script are as follows:

- **Constant Definitions** Define the constants to be used by the script.
- **Initialization/Setup** Add type libraries, declare global variables, and perform any additional setup associated with this script.
- **Main Function** The main functions and subroutines used in the script.
 - CreateViaMask()
- **Utility Functions and Subroutines** Special "workhorse" routines that are focused on this script.
 - AddViaMask()
 - GetPCBDocObj()
- **Helper Functions and Subroutines** Generic "helper" routines that can be used in other scripts.
 - ConfigureUserLayer()
 - AddMaskToUserLayer()
- **Validate Server** The standard function used to license the document and validate the server. (Note that we won't go through this function in this chapter – for more details on this function see *Chapter 3: Running Your First Script*.)

Constant Definitions

Lines 9 through 13 declare the constants we will use throughout 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

On Lines 16 through 18 we add the type libraries for FabLink XE, Expedition PCB, and the Mask Engine.



Note: As we discussed in *Chapter 23*, it is not really necessary to add type libraries for both Expedition PCB and FabLink XE because their automation interfaces are the same. The reason we do add both type libraries here is that this is good programming practice.

```
15  ' Add any type libraries to be used.
16  Scripting.AddTypeLibrary("MGCPBCB.FablinkXEApplication")
17  Scripting.AddTypeLibrary("MGCPBCB.ExpeditionPCBAplication")
18  Scripting.AddTypeLibrary("MGCPBCBEngines.MaskEngine")
```

On Lines 21 through 30 we instantiate and initialize our global variables and we acquire the FabLink XE *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
```

On Line 33 we validate the document.

```
32  ' License the document
33  ValidateServer(pnlDocObj)
```

On Line 36 we start a transaction to group all of the changes we are about to make into a single **Undo** level.

```
35  ' Start a transaction to group changes for undo
36  Call pnlDocObj.TransactionStart()
```

On Line 39 we call our main *CreateViaMask()* routine.

```
38  ' Create the mask for top pads of vias with specified padstack
39  Call CreateViaMask(VIA_PADSTACK, TOP)
```

On line 41 we end our transaction.

```
41  ' End the transaction
42  Call pnlDocObj.TransactionEnd()
```

Finally, on Line 45 we display a message in the status bar to inform the user as to the task we've just performed.

```
44  ' Let the user know we are done
45  pnlAppObj.Gui.StatusBarText("Via mask created")
```

Create Via Mask Function

Lines 53 through 115 are where we declare the main routine that is used to generate the via mask. This routine accepts two parameters: a string defining the padstack name of interest and an integer – 1 or 2 – that is used to specify whether we are interested in the top or bottom of the board, respectively.

```
50  ' Creates the side view on the panel.
51  ' padstackNameStr - String
52  ' boardSideConst - Constant (TOP,BOTTOM)
53  Sub CreateViaMask(padstackNameStr, boardSideConst)
```

On Lines 55 and 56 we create the Mask Engine. On Line 57 we specify the units to be used with the Mask Engine.

```
54      ' Create the mask engine
55      Dim maskEngObj
56      Set maskEngObj = CreateObject("MGCPCBEngines.MaskEngine")
57      maskEngObj.CurrentUnit = unitsEnum
```



Note: Each server has its own library of enumerates. Earlier in this script (on Line 24) we assigned the *epcbUnitMils* enumerate from the Expedition PCB type library to our *unitsEnum* variable. Now, on line 57, we are associating the value of this variable with the Mask Engine.

Generally speaking, it is not recommended to use enumerates defined in one type library with a different server. Having said this, 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 us to use a single variable to specify the units to be used across multiple servers.

On Lines 60 and 61 we 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()
```

On Lines 64 and 65 we create a mask to hold the outlines of all of the vias in which we are interested.

```
63      ' Create a mask to hold the via outlines.
64      Dim viaMaskObj
65      Set viaMaskObj = maskEngObj.Masks.Add()
```

On Lines 52 and 53 we acquire the collection of boards from the panel.

```

67      ' Get the collection of boards
68      Dim boardColl
69      Set boardColl = pn1DocObj.Borads

```

On Line 56 we use the *Sort* method to sort our collection by performing a standard alphabetical sort on their instance names. The reason we 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()

```

On Lines 75 through 78 we instantiate and initialize some 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

```

On Lines 79 through 89 we iterate through the collection of boards. On Line 81 we check to see if the current board is already open in Expedition PCB; if not, on Line 82 we call our *GetPCBDocObj()* utility routine to create (launch) Expedition PCB and to open the board design document and on Line 83 we reset our 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,
83                               boardObj.FullName)
83                  fullNameStr = boardObj.FullName

```

On Line 87 we call our *AddViasMask()* utility routine to add the via pad outlines associated with the current board instance to our vias mask.

```

85          ' Add the via shapes to this mask
86          ' for all boards of this type
87          Call AddViasMask(boardColl, pcbDocObj,
88                            padstackNameStr, boardSideConst, viaMaskObj)
88      End If
89  Next

```



Note: The loop between lines 79 and 89 (as discussed above) is very similar to the somewhat related loop we used in the script in *Chapter 24*, but there is a significant difference. First, consider the loop from *Chapter 24* as follows:

```

' Loop from Chapter 24
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
71      :
77
78      ' Add the components for this board instance
79      Call AddComponents(boardObj, pcbDocObj, topSurfaceXReal,
80                           bottomSurfaceXReal)

```

```
81 Next
```

In this case, on Line 79 the *AddComponents()* utility routine is called for every board instance (that is, for each iteration around the loop) and the first parameter to this routine is the FabLink XE *Board* instance object.

By comparison, consider the corresponding loop in the current script:

```
' Loop from Chapter 25
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,
83                             boardObj.FullName)
84          fullNameStr = boardObj.FullName
85          ' Add the via shapes to this mask
86          ' for all boards of this type
87          Call AddViasMask(boardColl, pcbDocObj,
88                            padstackNameStr, boardSideConst, viaMaskObj)
89      End If
89 Next
```

In this case, on Line 87, the *AddViasMask()* utility routine is only called the first time the board is opened; furthermore, as opposed to the single *Board* object used in the previous script, the first parameter to this routine is the entire collection of *Board* instances on the FabLink XE panel.

As we shall see, the *AddViasMask()* utility routine is written in such a way as to read the data from Expedition PCB a single time for all boards of the same type. The end result is that this new script architecture has significantly better performance to the script in *Chapter 25*, because communicating with – and accessing data from – an *Out-of-Process Client* is much more time-consuming than working with an *In-Process Client* (see *Chapter 1* for more discussions on this topic).

On Line 84 we use the *Quit* method on Expedition PCB.

```
91     ' Close Expedition
92     pcbAppObj.Quit()
```

On Lines 95 and 96 we release all references to the Expedition PCB *Application* and *Document* objects.

```
94     ' Explicitly release all references
95     Set pcbDocObj = Nothing
96     Set pcbAppObj = Nothing
```

On Line 100 we acquire the points array for the panel outline. On Line 101 we add this shape to our 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,
102                                              2) + 1, pntsArr)
```

On Line 104 we use the *Oversize* method of the *ViaMask* object to increase the diameter of all of the via shapes in our via mask (this is to add the clearance we require).

```
103    ' Grow the via mask
104    Call viaMaskObj.Oversize(OVERSIZE)
```

On Lines 107 and 108 we create a new mask to hold the result from the operation we are about to perform.

```
106      ' Create a mask to store the result it.  
107      Dim resultMaskObj  
108      Set resultMaskObj = maskEngObj.Masks.Add()
```

On Line 111 we use the *BooleanOp* method of the *MaskEngine* object to subtract our via mask from our panel mask.

```
110      ' Subtract the via mask from the panel outline mask.  
111      Set resultMaskObj =  
             maskEngObj.BooleanOp(emeBooleanOpSubtract,  
             panelMaskObj, viaMaskObj)
```

On Line 114 we call our *AddMaskToUserLayer()* helper routine to place the result mask graphics on the desired 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 is where we declare some utility routines. These routines are similar in context to our generic "helper" routines (see the next topic), except that they are specific to this script and may not be used elsewhere without some "tweaking".

AddViaMask() Subroutine

Lines 127 through 169 declare the routine that is used to add the vias to the via mask. This routine accepts five parameters: the collection of *Board* instances on the FabLink XE panel, the Expedition PCB *Document* object for which we want to process the vias, the padstack name (string) associated with the pads for which we want to create masks, the side of the board in which we are interested (1 = top, 2 = bottom), and the via mask itself (the via mask is an *In-Out* parameter).

```
121  ' Adds the vias to the via mask.  
122  ' boardColl - Boards Collection  
123  ' pcbDocObj - Document Object (Expedition PCB)  
124  ' padstackNameStr - String  
125  ' boardSideConst - Constant (TOP,BOTTOM)  
126  ' viaMaskObj - Mask Object (In/Out)  
127  Sub AddViasMask(boardColl, pcbDocObj, padstackNameStr,  
                     boardSideConst, viaMaskObj)
```

On Lines 131 through 136 we resolve the side of the board in which we are interested (top or bottom) to a particular layer 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
```



Note: Before we dive into the following deeply nested control statement, you may wish to refer back to *Chapter 16: Removing Unconnected Pads from Pins and Vias* to review our earlier discussions on the differences/relationships between *Padstack* references and *PadstackObject* objects and the way in which they are associated with pins and vias.

On Lines 140 through 168 we 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
```

On Line 141, we check to see if this *Padstack* reference is the one we are looking for (remember 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 one we are looking for ... then on lines 144 through 166 we 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
```

On Lines 145 and 146 we get the collection of pads for the current via on the desired layer (top or bottom).

```
145      Dim padColl
146      Set padColl = viaObj.Pads(layerInt)
```

On Line 149 we check to see if there are any pads in this collection and – if so – on Lines 151 through 164 we 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
```

On Lines 152 and 153 we acquire the points array associated with the current geometry

```
152      Dim pntsArr
153      pntsArr = geomObj.PointsArray
```

On Lines 156 through 163 we iterate through all of the *Board* instances in the FabLink XE *Board* collection.

```
154      ' Transform points array for board of each type
155      Dim boardObj
156      For Each boardObj In boardColl
```

On Line 157 we make sure that the current *Board* instance is associated with the Expedition PCB *Document* object that was passed into this routine.

```
157      If boardObj.FullName = pcbDocObj.FullName Then
```

For each *Board* instance of the current type, on Line 159 we use the *TransformPointsArray* method associated with the FabLink XE *Board* instance object to modify the points array (see Chapter 23 for more discussions on this method).

```
158          Dim trasformPntsArr
159          trasformPntsArr =
            boardObj.TransformPointsArray(UBound(pntsArr,
2) + 1, pntsArr)
```

On Line 161 we add the transformed points array to our via mask.

```
160          ' Add the pad shape to the mask.
161          Call viaMaskObj.Shapes.AddByPointsArray(
            UBound(trasformPntsArr, 2) + 1,
            trasformPntsArr)
```

On Lines 162 through 168 we terminate our various control statements and then on Line 169 we exit 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 is used to acquire an Expedition PCB *Document* object. There are three parameters to this routine: an Expedition PCB *Application* object, an Expedition PCB *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)
```

On Lines 178 through 180 we check to see if a document is already open and – if so – we close it.

```
177  ' Close the document if one is open.
178  If Not pcbDocObj Is Nothing Then
179      pcbDocObj.Close()
180  End If
```

On Lines 183 through 186 we check to see if the application has already been created and – if not – we create it (observe that on Line 185 we suppress any annoying messages coming from this server).

```
182  ' Create the application if not already created.
183  If pcbAppObj Is Nothing Then
184      Set pcbAppObj =
            CreateObject("MGCPCB.ExpeditionPCBAplication")
185      pcbAppObj.Gui.SuppressTrivialDialogs = True
186  End If
```

On Line 189 we open the document and on Line 190 we validate it.

```
188      ' Open and license the document
189      Set pcbDocObj = pcbAppObj.OpenDocument(docNameStr)
190      ValidateServer(pcbDocObj)
```

On Line 193 we set the units associated with this document and then on Line 195 we exit this routine.

```
192      ' Set the units.
193      pcbDocObj.CurrentUnit = unitsEnum
194
195 End Sub
```

Helper Functions and Subroutines

This is where we declare some generic "helper" routines. These routines are not focused on this script per se; they can easily be used in other scripts.

ConfigureUserLayer() Function

Lines 203 through 221 declare the routine that is used to create and configure 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.

```
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)
```

On Lines 205 and 206 we acquire the user layer if it exists.

```
204      ' See if our user layer already exists.
205      Dim usrLyrObj
206      Set usrLyrObj = pnlDocObj.FindUserLayer(userLayerNameStr)
```

On Lines 208 through 214 we create the user layer if it doesn't exist; otherwise we clear 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 =
211          pnlDocObj.SetupParameter.PutUserLayer(userLayerNameStr)
212      Else
213          ' It does exist, remove any gfx on the user layer
214          pnlDocObj.UserLayerGfxs(, userLayerNameStr).Delete
214      End If
```

On Line 217 we ensure that display of the user layer is turned on.

```
216      ' ensure the user layer is turned on
217      pnlDocObj.ActiveView.DisplayControl.UserLayer
              (userLayerNameStr) = True
```

On Line 220 we set 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 is used to add the contents of a specified mask to a specified user layer. This routine accepts three parameters: a *Document* object (this can be an Expedition PCB document or a FabLink XE document), the name (string) of the user layer, and the *Mask* object we want to add to the user layer.

```
223  ' Adds the contents of a mask to a user layer.  
224  ' docObj - Document Object (Fablink XE or Expedition PCB)  
225  ' userLayerNameStr - String  
226  ' maskObj - Mask Object  
227 Sub AddMaskToUserLayer(docObj, userLayerNameStr, maskObj)
```

On Line 231 we call our *ConfigureUserLayer()* helper routine to either create a new user layer (or return an existing user layer) and to ensure that the display of this user layer is turned on.

```
229      ' Get/Create user layer  
230      Dim userLayerObj  
231      Set userLayerObj = ConfigureUserLayer(userLayerNameStr)
```

On Line 234 we use the *AbsorbHoles* method on the *Mask* object (see the note at the end of this routine for more discussions on this method).

```
233      ' Make mentorgons  
234      Call maskObj.AbsorbHoles()
```

On Lines 239 through 243 we iterate through all of the shapes in the mask and add them to the user layer (in this example we will have only a single shape – see the note at the end of this routine for more discussions on the *AbsorbHoles* method and the difference between shapes and holes).

On Line 241 we acquire the points array associated with the current shape; on Line 242 we use the *PutUserLayerGfx* method to create the graphics corresponding to this points array on our 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, on line 245 we exit this routine.

```
245 End Sub
```



Note: Generally speaking, masks differentiate between shapes (outlines) and holes. For example, consider Figure 25-6(a) which comprises a single shape (a rectangular outline) and a single hole. In this case, the shape comprises five points, each of which will be represented by X/Y values in a points array (remember that the final point will have the same X/Y location as the first point). By comparison, the hole comprises three points defining the start, center, and end (once again, the hole's end point will have the same X/Y location as its start point).

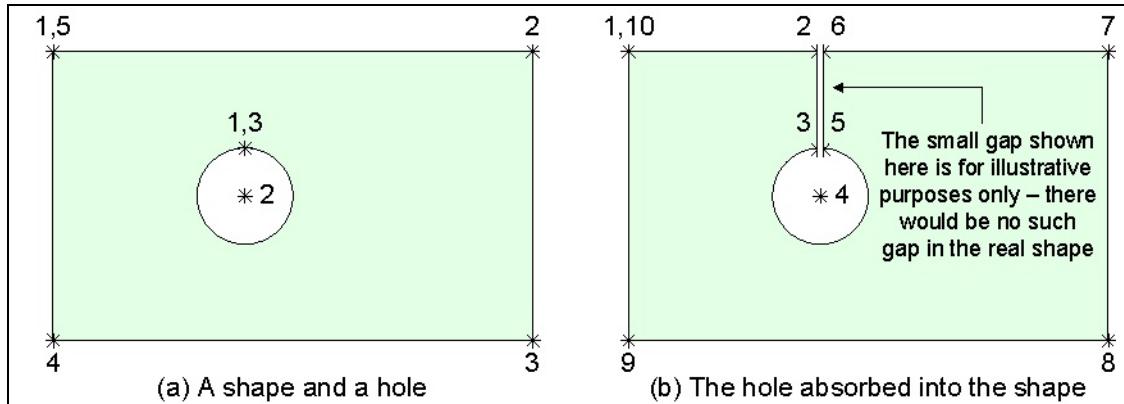


Figure 25-6. The process of absorbing holes into shapes.

The problem is that we now have two points arrays that have to be manipulated if we want to perform a task like generating the graphics associated with this shape-hole combo on a user layer. And, of course, the problem becomes much greater when we are 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 purview of that shape as illustrated in Figure 25-6(b). The result is a single points array that encompasses both the shape and any holes (the narrow gap shown in this figure is for illustrative purposes only; there would be no such gap in the real shape).

We should also note that a mask can contain multiple shapes, each of which may have holes associated with it. In this case, the *AbsorbHoles* method will absorb any holes into their corresponding shapes.

Creating and Testing the Script

- 1) Use the scripting editor of your choice to enter the script described above (including the license server function which is not shown above) and save it out as *CreateViaMask.vbs*.
- 2) Close any instantiations of Expedition PCB and FabLink XE that are currently running, and then launch a new instantiation of FabLink XE.
- 3) Open the *CandyPanel.pnl* design.
- 4) Drag-and-drop the *CreateViaMask.vbs* script into the middle of the FabLink XE design area.
- 5) Observe the via mask appear in front of the panel. Zoom on this mask and observe the way the holes surrounding vias using padstack "VIA31" are displayed.

Enhancing the Script

There are a number of ways in which this script could be enhanced. One suggestion is as follows (it would be 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.

Chapter 26: Generating Manufacturing Output Files

Introduction

- Overview:** This script uses two different output engine automation interfaces; it configures their settings and instructs them 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 We Start

After completing a panel (or a board, because this particular script will also work with Expedition PCB), a suite of manufacturing files need to be generated to be passed to the board manufacturer. FabLink XE provides many tools and a great deal of flexibility with regard to generating these files. For example, suppose we use the **Output > Gerber** command to access the *Gerber* engine (this is just one of the engines we might want to use). This returns a dialog that allows us to set up an entire suite of files (Figure 26-1).

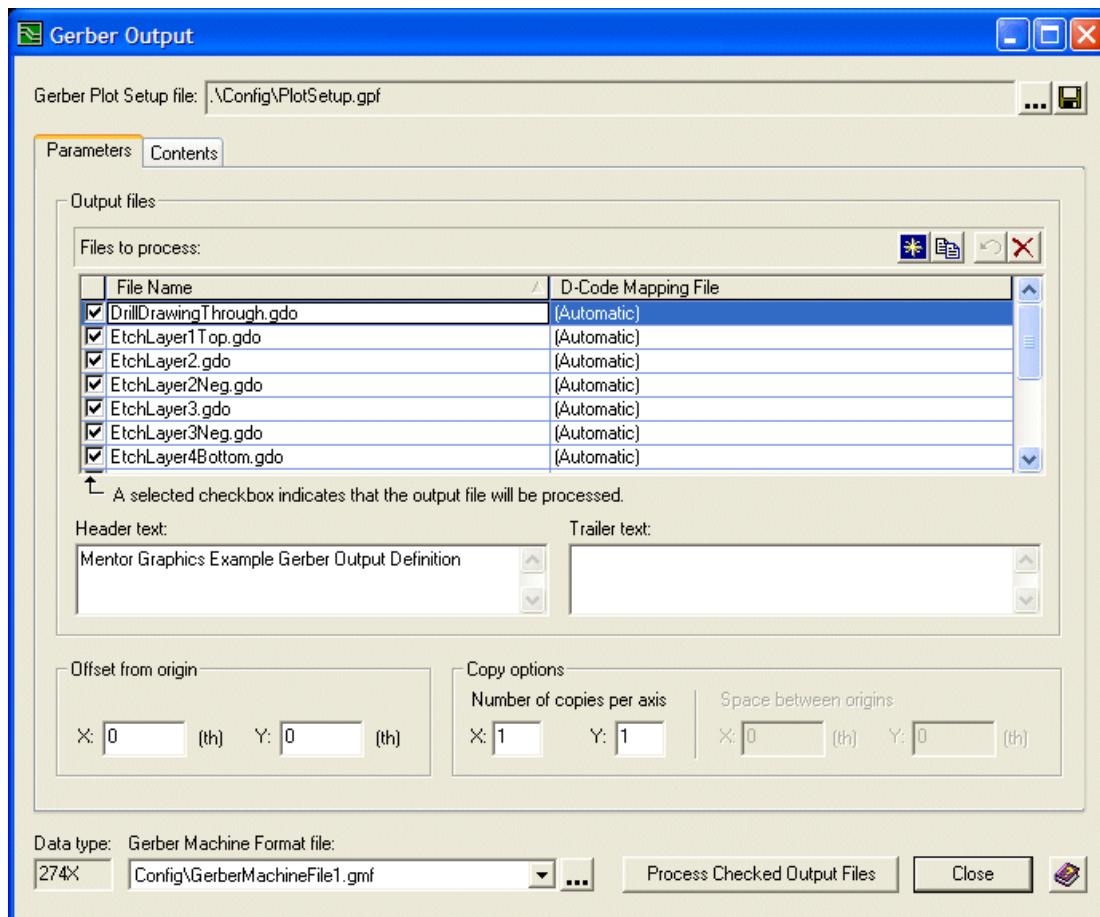


Figure 26-1. The Parameters tab on the Gerber Output dialog.

Each of these files has a set of options associated with it; for example, consider the options associated with the *EtchLayer1Top.gdo* file as illustrated in Figure 26-2 (in order to access these options, you would first select the *EtchLayer1Top.gdo* item in the **File name** list under the **Parameters** tab as shown in Figure 21, and then select the **Contents** tab to reveal the associated options).

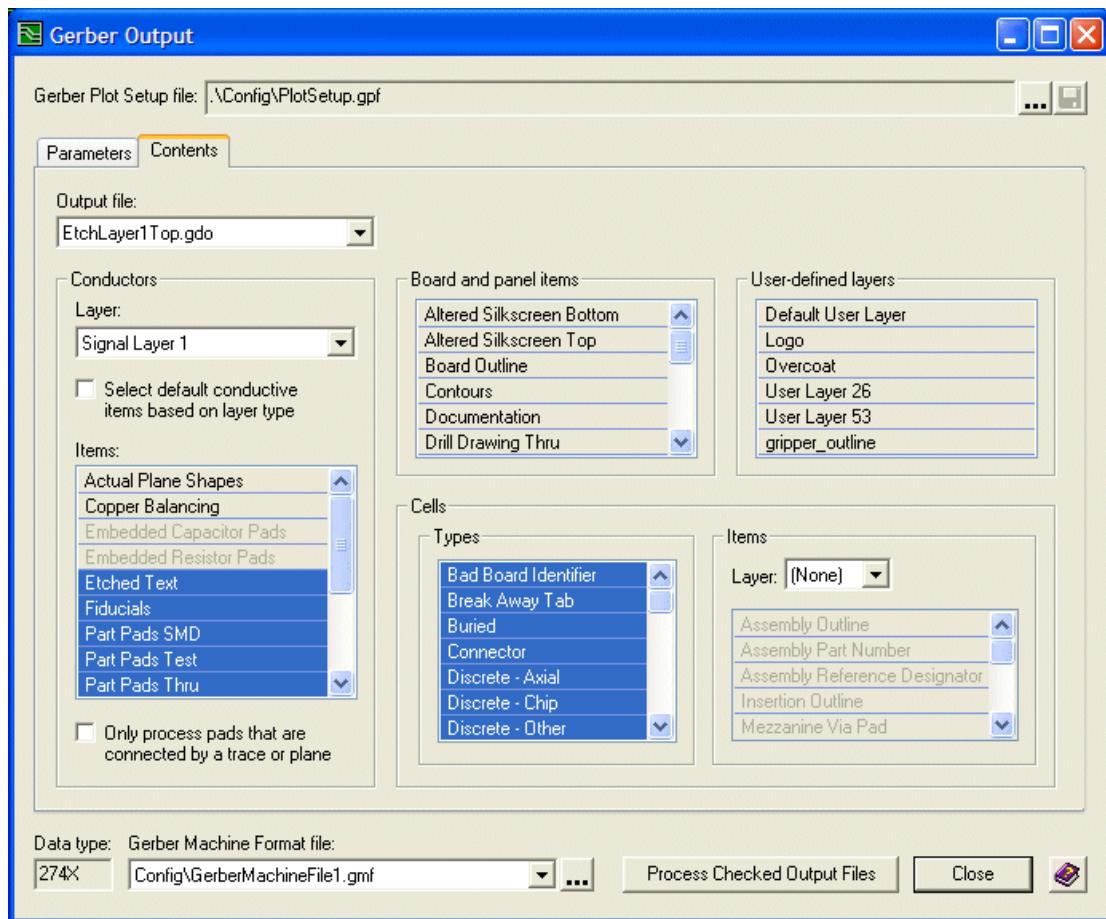


Figure 26-2. Options associated with the EtchLayer1Top.gdo output file.

The problem with the flexibility provided by FabLink XE and its output engines is that the possibility (probability) of human error becomes a major factor; for example, it's easy to incorrectly select/deselect one or more of these options. The problem is compounded by the fact that the generation of manufacturing output files occurs at the end of the board design process, by which time 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. In the case of the script described in this chapter, we are going to use the Gerber and NC Drill output engines; the *Prog IDs* for these – and other – output engines are as follows:

```
MGCPCBEngines.NCDrill
MGCPCBEngines.PDFOutput
MGCPCBEngines.DFFDRC
MGCPCBEngines.DRC
MGCPCBEngines.Gerber
MGCPCBEngines.ManufacturingOutputValidation
```



Note: For the purposes of the script described in this chapter, we are going to run the Gerber and NC Drill automation engines as batch processes. This means that we do not actually require FabLink XE to be running (it doesn't matter if FabLink XE is running).

In order to see how our script is going to work, look at the *Output* directory/folder in the FabLink XE *CandyPanel* design and observe that it is empty as illustrated in Figure 26-3.

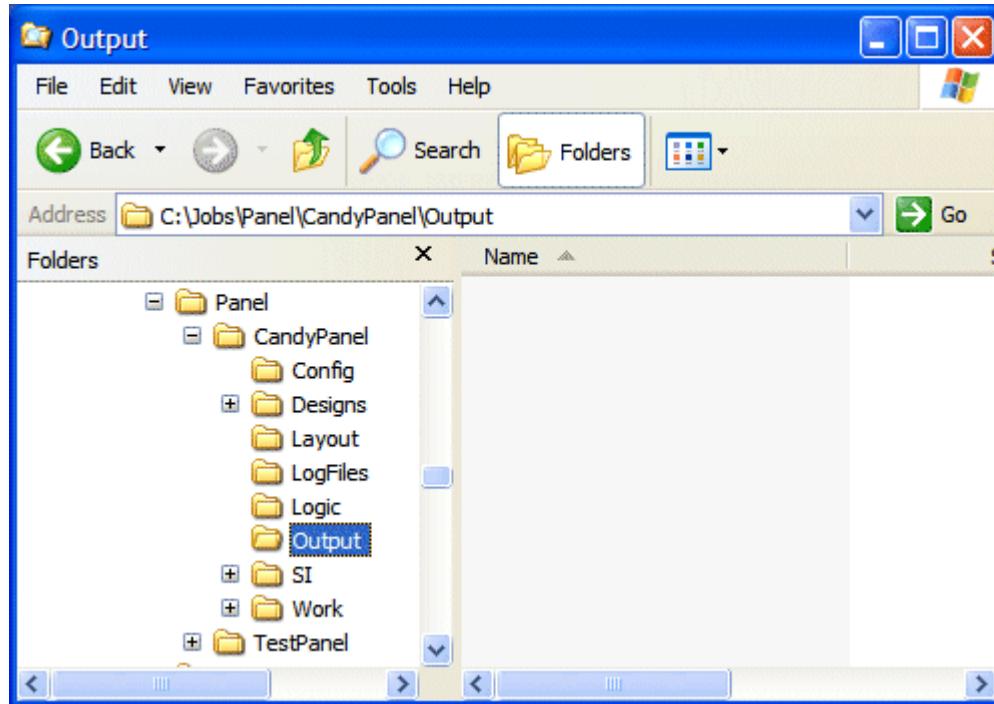


Figure 26-3. The Output directory/folder in the CandyPanel design is empty.

Now, assume that we've created a script called *CreatePanelOutputs.vbs* and that this script resides in your C:\Temp directory/folder. Launch a console (terminal) window and set its context to C:\Temp (see Chapter 8: *Running Scripts from the Command Line* for more details on using console windows). Prepare to use the *mgcscript* engine to run the *CreatePanelOutputs.vbs* script. As the only argument to this script, specify the name of the panel file you want to process (*c:\jobs\Panel\CandyPanel\CandyPanel.pnl* in this example) as illustrated in Figure 26-4. The script is written such that you must specify the full path.



Figure 26-4. Preparing to run the *CreatePanelOutputs.vbs* script.

Once you've keyed-in this command, press the <Enter Key>. 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 as illustrated in Figure 26-5.

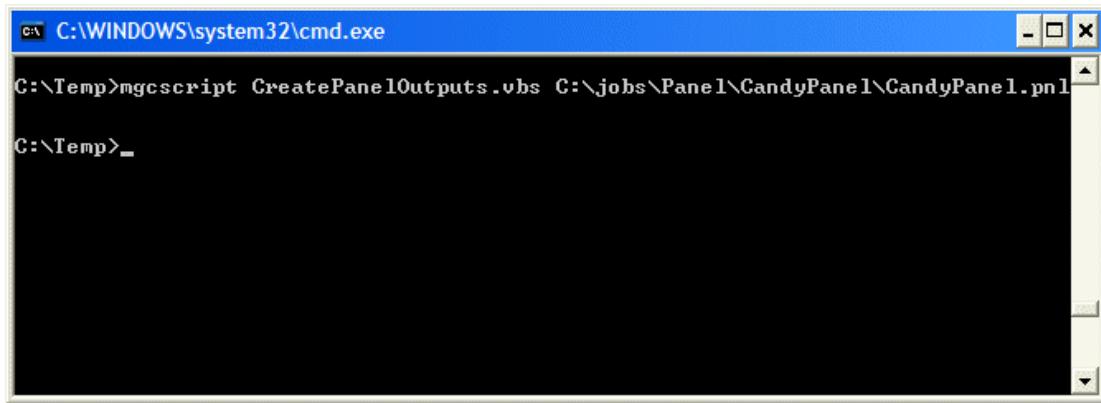


Figure 26-5. The console prompt reappears, thereby indicating that the script has run.

If you were to now look in the *Output* directory/folder in the FabLink XE *CandyPanel* design, you would observe that it now contains *Gerber* and *NCDrill* sub-folders, each of which contains appropriate manufacturing output files as indicated in Figure 26-6.

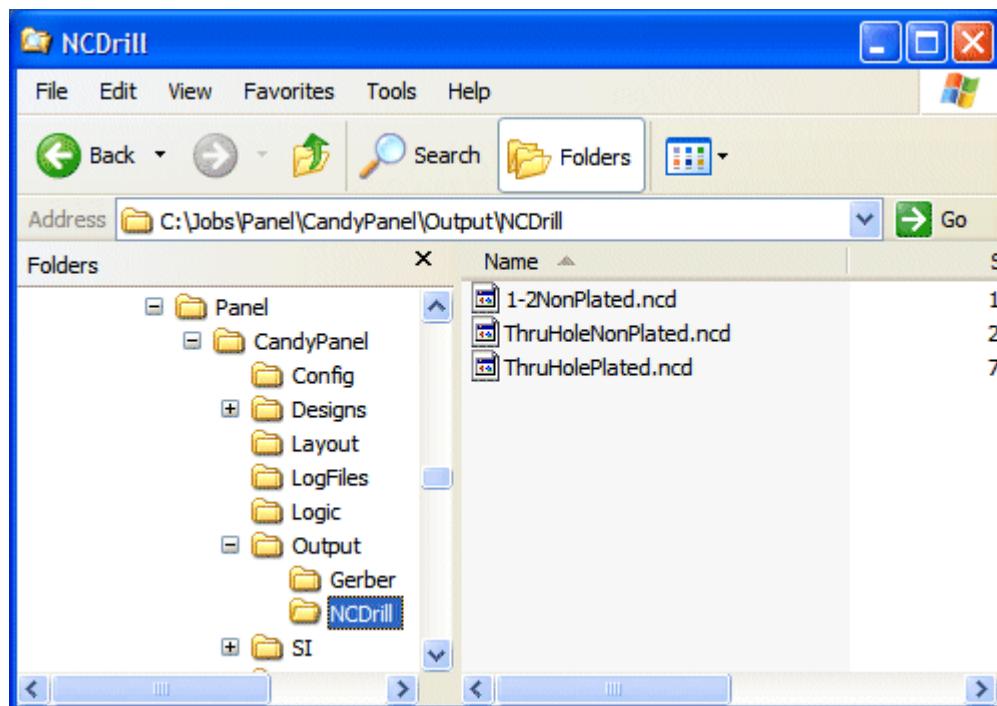


Figure 26-6. The Output directory/folder in the CandyPanel design has been populated.

The Script Itself

The key sections forming this script are as follows:

- **Initialization/Setup** Add type libraries, declare global variables, and parse command line arguments.

- **Main Functions and Subroutines** The main functions and subroutines used in the script.
– RunGerber()
– RunNCDRill()
- **Utility Functions and Subroutines** Special "workhorse" routines that are focused on this script.
– SetupGerberMachineFormat()
– SetupGerberOutputFiles()
– SetupGerberSpecificOutputFile()
– SetupGerberAllOutputFile()
– SetupNCDRillMachineFormat()
– SetupNCDRillParameters()
– SetupNCDRillChart()
- **Helper Functions and Subroutines** Generic "helper" routines that can be used in other scripts.
– StripFile()
– CorrectSlashes()

Initialization/Setup

On Lines 10 and 11 we 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( "MGCPCEngines.Gerber" )
11 Scripting.AddTypeLibrary( "MGCPCEngines.NCDRill" )

```

On line 15 we instantiate a local variable to hold the name of the design (the entire path including the file name); on Line 17 we instantiate 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 are used to parse the command line arguments; if there are an incorrect number of arguments a *MsgBox()* is used on Line 25 to inform the user as to the correct usage model for this script (see *Chapter 8: Running Scripts from the Command Line* 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
26          <design_name>" )
27  End If

```

On Line 30 we call our *StripFile()* helper function to acquire directory/folder path containing the panel file. On Line 31, we use this information to create a path to the appropriate configuration directory/folder.

```
28  ' Find the config directory
29  Dim pnlPathStr
30  pnlPathStr = StripFile(designNameStr)
31  designConfigPath = pnlPathStr + "/Config/"
```

Finally, on Lines 34 and 35 we call our main *RunGerber()* and *RunNCDrill()* routines, respectively.

```
33  ' Call main subroutines
34  Call RunGerber()
35  Call RunNCDrill()
```



Note: This script does NOT require the standard license server function. If you look at the code for the license server function (see the *Creating and Running the Via Count Script* topic in *Chapter 3: Running Your First Script*), you will notice that the validation is performed on the *Document* object. In the case of the script presented in this chapter, external engines are used and the *Document* object does not come into play; thus, since there is no *Document* object to validate there is no reason to perform the validation.

Having said this, we should also note that 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).

Main Run Gerber Engine Subroutine

Lines 41 through 77 are where we declare the main *RunGerber()* subroutine. On Lines 44 and 45 we 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("MGCPCEngines.Gerber")
```

On Line 48 we set the *DesignFileName* property of the *Gerber* object to be the name of the design we want to process (this is the command line argument we acquired on Line 23).

```
47      ' Set the design file name
48      gerberEngineObj.DesignFileName = designNameStr
```

On Lines 51 and 52 we call our *SetupGerberMachineFormat()* and *SetupGerberOutputFiles()* utility functions, respectively.

```
50      ' Setup files for Gerber output
51      Call SetupGerberMachineFormat(gerberEngineObj)
52      Call SetupGerberOutputFiles(gerberEngineObj)
```

Due to the fact that there is a strong possibility for errors, on Line 56 we use the *On Error Resume Next* statement and on Line 57 we clear any existing errors.

```
54      ' There are many possible errors when running the engine.
55      ' We will catch the case and display them on our own
56      On Error Resume Next
```

```
57 Err.Clear()
```

On Line 60 we use the *Go* method to run the *Gerber* engine.

```
59     ' Run the Gerber Engine
60     gerberEngineObj.Go()
```

Since our script is performing a lot of actions, there is the possibility we will see a lot of errors. Each output engine supports its own set of errors, and each engine allows us to acquire its collection of errors after processing has been completed. On line 62 we use the implicit *Err* object to determine if there was an error. Unlike our previous scripts, however, we don't actually query the *Err* object to obtain the error description; instead, on Line 65 we use 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
```

On Line 67 we instantiate a variable to hold a string and initialize it to contain an empty string. On Lines 68 through 70 we iterate through all of the errors in our collection and add each error to our string variable (we separate each error with a new line 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
```

On Line 72 we check to see if the string variable is empty or not; if not then we have some errors, in which case on Line 73 we use 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 our executing an *On Error Goto 0* statement before the end of the routine). Furthermore, the system will automatically return to the default error handling mode within any *called* routines; in this case, however, the system will return to the *On Error Resume Next* mode when the called routine returns control to the calling routine.

To put this in a nutshell, the effects of *On Error Resume Next* apply only within the routine in which this statement is found; any other routines (called or calling) will use the default error mode unless otherwise specified within those routines themselves.

Main Run NC Drill Engine Subroutine

Lines 80 through 115 are where we declare the main *RunNCDrill()* subroutine. On Lines 83 and 84 we 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" )

```

On Line 87 we set the *DesignFileName* property of the *NCDrill* object to be the name of the design we want to process (this is the command line argument we acquired on Line 23).

```

86      ' Set the design file name
87      ncDrillEngineObj.DesignFileName = designNameStr

```

On Lines 89, 90, and 91 we call our *SetupNCDrillMachineFormat()*, *SetupNCDrillParameters()*, and *SetupNCDrillChart()* utility functions, respectively.

```

89      Call SetupNCDrillMachineFormat(ncDrillEngineObj)
90      Call SetupNCDrillParameters(ncDrillEngineObj)
91      Call SetupNCDrillChart(ncDrillEngineObj)

```

As we noted during our discussions on the previous function, there is a strong possibility for errors to occur whilst generating these manufacturing output files. Thus, on Line 95 we use the *On Error Resmet Next* statement and on Line 96 we clear any existing errors.

```

93      ' There are many possible errors when running the engine.
94      ' We will catch the case and display them on our own
95      On Error Resume Next
96      Err.Clear

```

On Line 99 we use the *Go* method to run the *NCDrill* engine.

```

98      ' Run the NCDrill Engine
99      ncDrillEngineObj.Go

```

Once again, each output engine supports its own set of errors, and each engine allows us to acquire its collection of errors after processing has been completed. On Line 104 we use 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

```

On Line 106 we instantiate a variable to hold a string and initialize it to contain an empty string. On Lines 107 through 109 we iterate through all of the errors in our collection and add each error to our string variable (we separate each error with a new line 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

```

On Line 111 we check to see if the string variable is empty or not; if not then we have some errors, in which case on Line 112 we use a *MsgBox()* to display all of the errors.

```

111      If Not errDisplayStr = "" Then
112          MsgBox("Error(s) running NC Drill engine:" &
113              vbCrLf & errDisplayStr)
114      End If
115  End Sub

```

Utility Functions and Subroutines

This is where we declare some utility routines. These routines are similar in context to our generic "helper" routines (see the next topic), except that they are specific to this script and may not be used elsewhere without some "tweaking".

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)

```

On Lines 124 and 125 we use the *MachineFormat* property of the *Gerber* object to acquire the *GerberMachineFormat* object.

```

124      Dim gerberMachineFormatObj
125      Set gerberMachineFormatObj = gerberEngineObj.MachineFormat

```

On Line 127 we specify the file where all of the settings will be written to, and then on Lines 128 through 141 we specify the required settings.

```

127      gerberMachineFormatObj.FileName = designConfigPath &
128          "UserGerberMachineFile.gmf"
129      gerberMachineFormatObj.DataType = eengGerber274X
130      gerberMachineFormatObj.DataMode = TRUE
131      gerberMachineFormatObj.StepMode = eengStepIncremental
132      gerberMachineFormatObj.DataFormatLeadingDigits = 3
133      gerberMachineFormatObj.DataFormatTrailingDigits = 2
134      gerberMachineFormatObj.ZeroTruncation =
135          eengZeroTruncationNone
136      gerberMachineFormatObj.CharacterSet = eengCharacterSetASCII
137      gerberMachineFormatObj.ArcStyle = eengArcStyleQuadrant
138      gerberMachineFormatObj.Delimiter = eengDataDelimiterStar
139      gerberMachineFormatObj.Comments = TRUE
140      gerberMachineFormatObj.SequenceNumbering = TRUE
141      gerberMachineFormatObj.Unit = eengUnitInch
142      gerberMachineFormatObj.PolygonFillMethod =
143          eengPolygonFillDraw
144      gerberMachineFormatObj.RecordLength = 0
145  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)

```

On Line 150 we acquire the collection of output files (note that this collection will be empty at the moment).

```
149      Dim gerberoutputFileColl  
150      Set gerberoutputFileColl = gerberEngineObj.OutputFiles
```

On Lines 152 we call our *SetupGerberSpecificOutputFile()* utility routine; on Line 153 we call our *SetupGerberAllOutputFile()* utility routine.

```
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 we acquired in the *SetupGerberOutputFiles()* routine (remember that the collection is empty at this point – also 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)
```

On Line 162 we use the *Add* method on our 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")
```

On Lines 163 through 166 we specify the attributes to be associated with our 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
```

On Line 170 we acquire the *BoardItems* collection associated with this output file.

```
168  ' Add board items individually  
169  Dim boardItemsObj  
170  Set boardItemsObj = outFileAddEachObj.BoardItems
```

On Lines 171 through 180 we add specific board items we want to output to our 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)
```

On Line 184 we acquire the *CellTypes* collection associated with this output file.

```
182      ' Add all cell types individually
183      Dim cellTypesObj
184      Set cellTypesObj = outFileAddEachObj.CellTypes
```

On Lines 185 through 208 we add specific cell types we want to output to our 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)
```

On Line 210 we specify that we want to output cells on the top side.

```
210      outFileAddEachObj.CellItemsSide = eengCellSideTop
```

On Line 214 we acquire the *CellItems* collection associated with this output file.

```
212      ' Add cell items individually
213      Dim cellItemsObj
214      Set cellItemsObj = outFileAddEachObj.CellItems
```

On Lines 215 through 222 we add specific cell items we want to output to our 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)
```

On line 225 we specify the conductive layer we want to output.

```
224      ' Add conductive items individually
225      outFileAddEachObj.ConductiveLayer = 1
```

On Line 227 we acquire the *ConductiveItems* collection associated with this output file.

```
226      Dim conductiveItemsObj
```

```
227     Set conductiveItemsObj = outFileAddEachObj.ConductiveItems
```

On Lines 228 through 239 we add specific conductive items we want to output to our 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 we acquired in the *SetupGerberOutputFiles()* routine and previously modified in the *SetupGerberSpecificOutputFile()* routine.

```
243 ' Sets up a Gerber output containing all data
244 ' outputFileColl - GerberOutputFile Collection
245 Sub SetupGerberAllOutputFile(outputFileColl)
```

we use the *Add* method on our 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")
```

On Lines 250 through 253 specify the attributes to be associated with our 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
```

On Lines 255 through 260 we specify that we want to add everything.

```
255     outFileAddAllObj.BoardItems.Add(eengBoardItemAll)
256     outFileAddAllObj.CellTypes.Add(eengCellTypeAll)
257     outFileAddAllObj.CellItemssSide = 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)

```

On Lines 267 and 268 we use the *MachineFormat* property of the *NCDrill* object to acquire the *NCDrillMachineFormat* object.

```

267      Dim ncDrillMFObj
268      Set ncDrillMFObj = ncDrillEngineObj.MachineFormat

```

On Lines 271 through 284 we 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

```

On Line 287 we specify a file to which the settings we just setup can be saved (this is optional, if this file is not specified the *NCDrill* engine will use these settings but it won't 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)

```

On Line 295 we use the *Parameters* property of the *NCDrill* object to acquire the *NCDrillParameters* object.

```

294      Dim ncDrillParametersObj
295      Set ncDrillParametersObj = ncDrillEngineObj.Parameters

```

On Lines 299 through 306 we setup the required NC Drill parameters.

```

293
294
295      ' Same as the options in "Drill Options"
296      ' tab of the "NC Drill Generation" dialog
297      ncDrillParametersObj.SweepAxis = eengSweepaxisHorizontal
298      ncDrillParametersObj.Bandwidth(eengUnitInch) = 0.1
299      ncDrillParametersObj.PreDrillHolesLargerThan(eengUnitInch) =
300                                         1
301      ncDrillParametersObj.OutputFileExtension = ".ncd"
302

```

```

303      Call ncDrillParametersObj.ClearFileHeader()
304      Call ncDrillParametersObj.ClearFileNotes()
305      Call ncDrillParametersObj.AddFileHeader("Drill Setup")
306      Call ncDrillParametersObj.AddFileNotes ("This file was
307          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)

```

On Line 315 we use the *Chart* property of the *NCDrill* object to acquire the *NCDrillChart* object.

```

313      Dim ncDrillChartObj
314      Set ncDrillChartObj = ncDrillEngineObj.Chart

```

On Lines 318 through 329 we 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"

```

On Line 332 we acquire the *Columns* collection associated with the *NCDrillChart* object

```

331      Dim ncDrillChartObjColumns
332      Set ncDrillChartObjColumns = ncDrillChartObj.Columns

```

On Lines 334 through 340 we 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 is where we declare some generic "helper" routines. These routines are not focused on this script per se; they can easily be used in other scripts.



Note: It would be a good idea if both of the helper routines shown here were collected into a library along with any related routines (see also *Appendix C: Creating and Using a Library*).

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 observe that this routine calls the *CorrectSlashes()* helper routine.)

```
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 above and save it out as *CreatePanelOutputs.vbs*.
- 2) Close any instantiations of FabLink XE 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 *Chapter 8: Running Scripts from the Command Line* 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) Look in the *Output* directory/folder in the FabLink XE *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 would be a good idea for you to practice your scripting skills by implementing this example):

- The existing script uses a *MsgBox()* to display any errors. The script could be modified to display error messages directly in the console (terminal) window (look in the *Using mgcscript* section of the **MGCPCB Automation Help** for more details on how this may be achieved).
- It was noted in the introduction that this script could also be run with Expedition PCB 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 will direct you to these options. Commenting out these lines will allow the script to work with a *.pcb file.

Chapter 27: Automated Board Placement on a Panel

Introduction

- Overview:** This script attempts to place the optimal (maximum) number of copies of a specified board in Expedition PCB onto an existing panel in FabLink XE.
- Approach:** To access the board extreme in Expedition PCB; to calculate the maximum number of boards that can fit within the panel border (try both 0° and 90° board rotations); and to then automatically place the boards on the panel.
- Points of Interest:**
- Using a FabLink XE script to access data in Expedition PCB.
 - Launching a dialog/form from within a script.
 - Implementing script-to-script communication.
- Items Used:**
- Expedition PCB Server
 - ProcessScript Method
 - PutBoard Method
 - Scripting.Globals Property

Before We Start

Although arbitrary panel sizes are sometimes used, generally speaking panels tend to come in standard sizes, so it's common to try to fit that maximum number of boards into a panel. As we shall see, this script allows this user to specify a board design in Expedition PCB 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 FabLink XE (it tries both 0° and 90° board rotations). Finally, the script automatically places the boards on the panel).

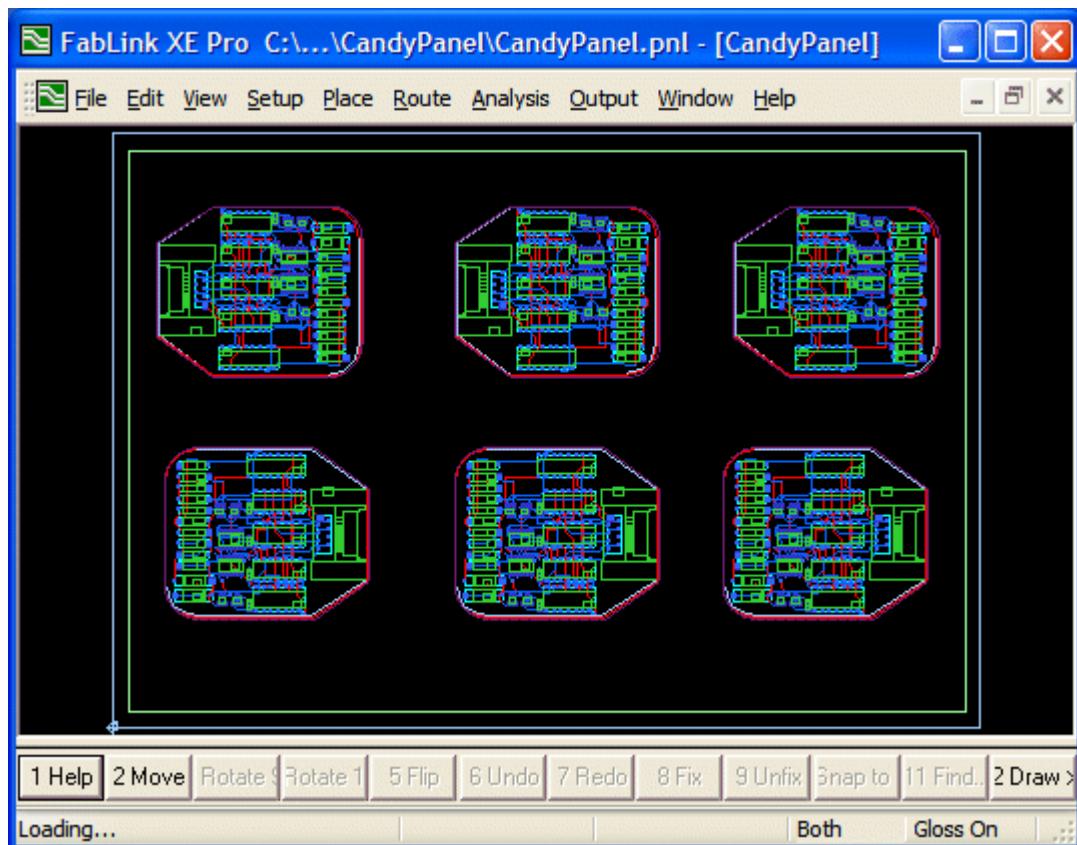


Figure 27-1. Our pre-defined CandyPanel panel.

In order to get a feel for how this script performs its magic, let's assume that we've already created the script (*PlaceBoards.vbs*) and the dialog/form (*PlaceBoardInput.efm*) presented in this chapter. If we were to launch FabLink XE and open our existing *CandyPanel* panel design, it would appear as illustrated in Figure 27-1. Observe that only six boards are presented in this design, because this is the way in which we (the authors) originally placed the boards by hand.

If we now selected and deleted the six existing boards, this would leave the bare panel as illustrated in Figure 27-2.

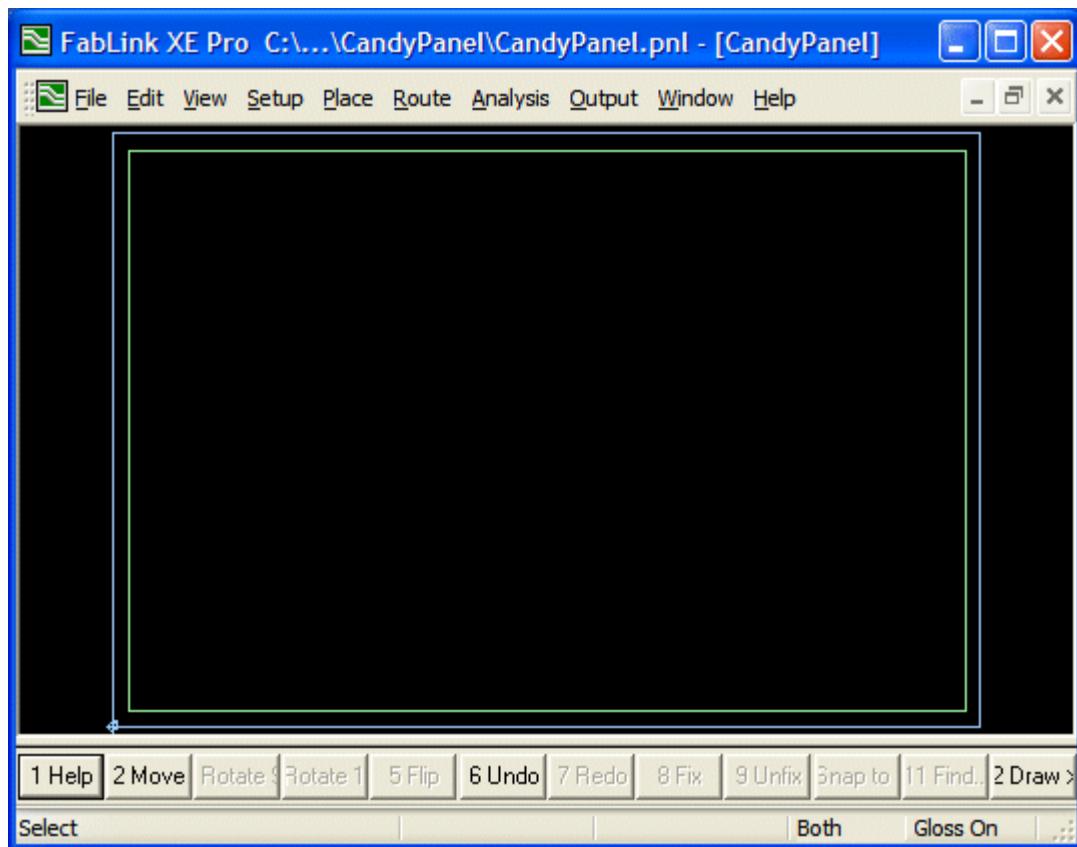


Figure 27-2. The CandyPanel panel after its boards have been deleted.

Now assume that we drag-and-drop our *PlaceBoards.vbs* script into the middle of the FabLink XE design area. This script would automatically launch the **Placement Parameters** dialog/form illustrated in Figure 27-3 (we will discuss the creation of this dialog/form later in this chapter).



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.

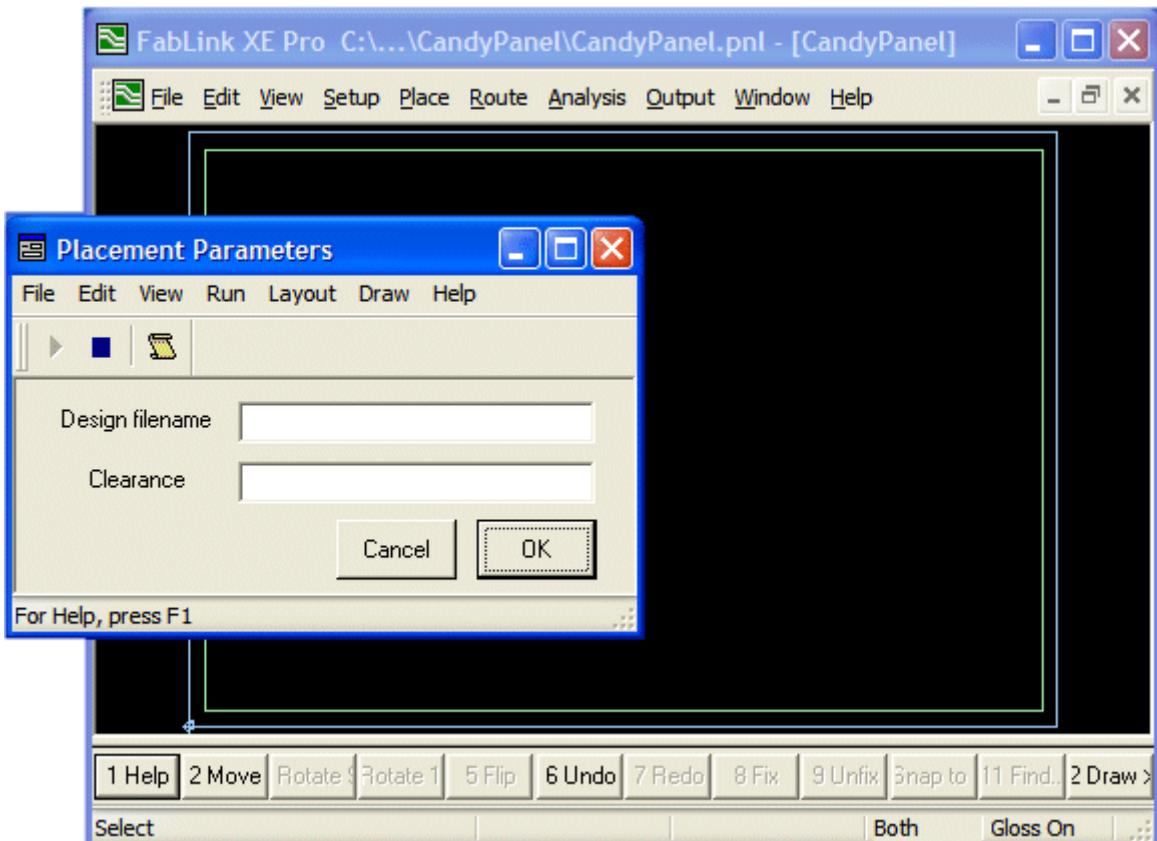


Figure 27-3. Running the script invokes the Placement Parameters dialog/form.

First, enter the path and filename to the *Candy.pcb* on your system in the **Design filename** field (this path is *C:\jobs\Candy\PCB\Candy.pcb* on our system, but it may be different on your system). Next, enter a clearance value of 100 in the **Clearance** field (our script will assume that these are the same units as in the FabLink XE panel design).



Note: When we create the **Placement Properties** dialog/form, in addition to clicking the **Cancel** or **OK** buttons, we also have to accommodate the possibility of the user clicking the red 'X' (**Close**) button in the upper right-hand corner of the dialog.

Now, assume that we've entered the values discussed above and we click the **OK** button. Our script will first evaluate the maximum number of rows and columns of this board design using a 0° board rotation. It will then evaluate the maximum number of rows and columns of this board using a 90° board rotation. Based on the result, the script will then automatically place the boards – three rows by three columns at 0° rotation, in this example – as illustrated in Figure 27-4.

Actually, we should point out that when the boards first appear in the panel, they are presented only as outlines. After our script had run, we used the **Place > Upload All Design Data** command to populate the boards with graphics so as to make them more visually appealing.

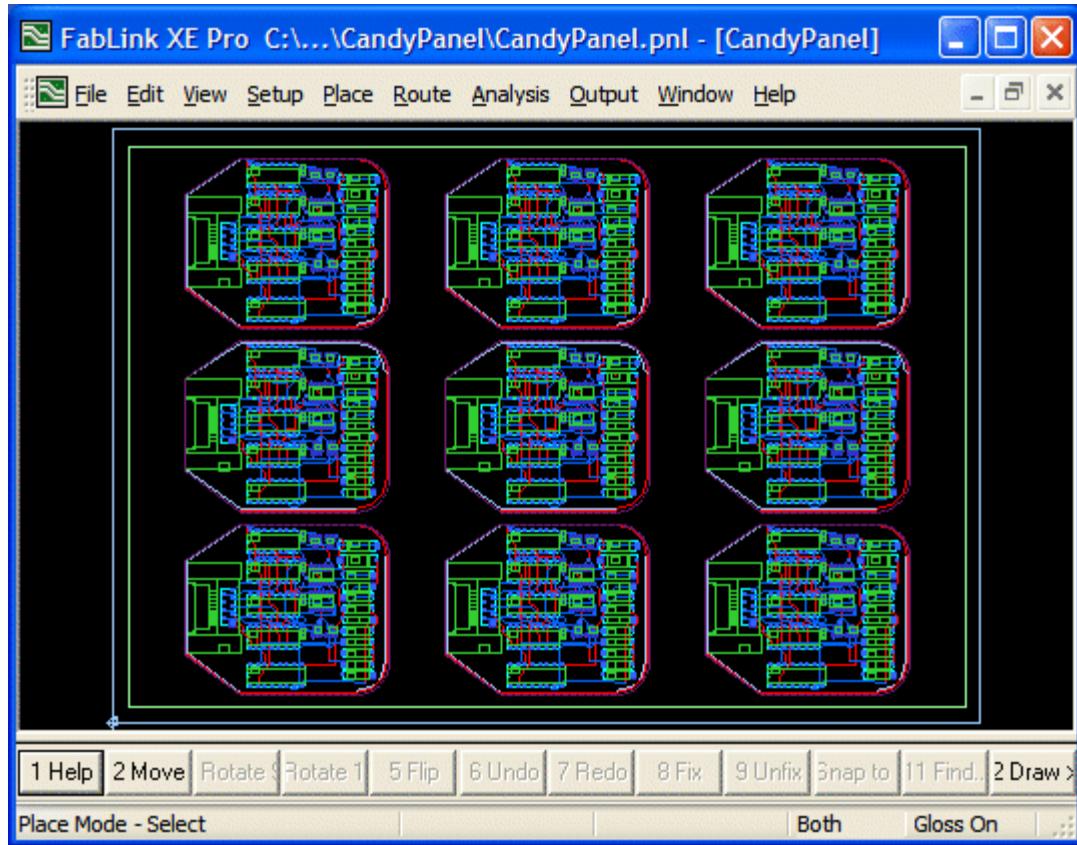


Figure 27-4. The script places the maximum number of boards on the panel.

The Script Itself

The key sections forming this script are as follows:

- **Constant Definitions** Define the constants to be used by the script.
- **Initialization/Setup** Get the FabLink XE *Application* and *Document* objects, license the document, add the type libraries, and perform any additional setup associated with this script.
- **Main Functions and Subroutines** The main functions and subroutines used in the script.
 - GetParameters()
 - PlaceBoards()
- **Utility Functions and Subroutines** Special "workhorse" routines that are focused on this script.
 - FindBestMatrix()
- **Helper Functions and Subroutines** Generic "helper" routines that can be used in other scripts.
 - GetBoardDimensions()
 - EqualizeClearance()
 - FindMaxFit()
 - FindInWDIR()
- **Validate Server** The standard function used to license the document and validate the server. (Note that we won't go through this function in this chapter – for more details on this function see *Chapter 3: Running Your First Script*.)

Constant Definitions

Lines 10 through 14 define the constants we will be using to implement script-to-script communications (we will discuss this in more detail shortly). Line 16 is used to define the name of the dialog/form that we are going 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 16 and 17 are used to get the *Application* object; Lines 20 and 21 are used to get the *Document* object; on Line 24 we license the document by calling the standard *ValidateServer()* function; and Lines 27 and 28 are used to add the type libraries required by this script.

On Lines 19 and 20 we add the type libraries for Expedition PCB and FabLink XE.



Note: As we discussed in *Chapter 23*, it is not really necessary to add type libraries for both Expedition PCB and FabLink XE because their automation interfaces are the same. The reason we do add both type libraries here is that this is good programming practice.

```
18  ' Add any type libraries to be used.
19  Scripting.AddTypeLibrary("MGCPBCB.FablinkXEApplication")
20  Scripting.AddTypeLibrary("MGCPBCB.ExpeditionPCBAplication")
```

On Lines 23 and 24 we instantiate global variables to hold our FabLink XE *Application* and *Document* objects.

```
22  ' Global variables
23  Dim pnlAppObj           ' Application object
24  Dim pnlDocObj           ' Document object
```

On Line 27 we acquire the FabLink XE *Application* object; on Line 30 we acquire the FabLink XE *Document* (panel) object; and on Line 32 we license the document (validate 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)
```

On Line 35 we instantiate a variable to hold the units we want to use, and then we assign the current design units associated with the panel to this variable.

```
34  ' Units to be used throughout scripts
35  Dim unitsEnum : unitsEnum = pnlDocObj.CurrentUnit
```



Note: In the past, we've explicitly defined the units we want to use prior to calling the *ValidateServer()* function. In this case, however, we want use the current design units associated with the panel.

On Line 38 we call our main *GetParameters()* function as part of an *If ... Then* statement. This is the function that will invoke our dialog/form, wait for the user to input the required values, and then access these values. If this function returns *True* (meaning we successfully acquired the path/name of the board in Expedition PCB and the desired clearance), then on Line 39 we call our main *PlaceBoards()* subroutine.

```
37  Dim designNameStr, clearanceReal
38  If GetParameters(designNameStr, clearanceReal) Then
39      Call PlaceBoards(designNameStr, clearanceReal)
40  End If
```

Main Get Parameters Function

Lines 50 through 72 are where we declare the function that will invoke our dialog/form, wait for the user to input the required values, and then access these values. This function accepts two parameters (both of these are *In-Out* parameters whose values will be set within the function): the path/name of the Expedition PCB 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  ' clearanceReal - Real (In/Out)
50  Function GetParameters(designNameStr, clearanceReal)
```

On Line 52 we initialize the *ANSWER* key of the *Scripting.Globals* object to be an empty string (remember that the *ANSWER* key was one of the constants we declared at the beginning of the script). As we shall see, we will be using this as a "flag" that lets us know when the user exits our 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 accessed by any in-process script/client/form. This is the mechanism by which one script/client/form can communicate with another script/client/form.

On Line 55 we use the *ProcessScript* method to invoke our dialog/form (remember that *INPUT_FORM* was one of the constants we declared at the beginning of the script, at which time we assigned it the name of the dialog/form we are going to create shortly). Observe that before we use the *ProcessScript* method to invoke our dialog/form, we use our *FindInWDIR()* helper routine to acquire the full path to the form).

```
54      ' Display the form
55      Call pnlAppObj.ProcessScript(FindInWDIR(INPUT_FORM), False)
```

On Line 59 through 61 we use a *While ... Wend* control loop to wait for the user to enter the data and/or exit the form (the *Scripting.Sleep* statement on Line 60 is used to prevent 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
```

On Line 65 we use the *Scripting.Globals* object check to see whether the dialog/form set the **ANSWER** key to **OK** or **Cancel**, thereby telling us whether the user clicked **OK** or **Cancel** buttons, respectively.

If the user clicked the **OK** button, then on Line 66 we use the *DESIGNNAME* key to retrieve the path/name of the board design in Expedition PCB; on Line 67 we use the *CLEARANCE* key to retrieve the clearance value requested by the user; and on Line 68 we set the return value for this function to be *True*. Alternatively, if the user clicked the **Cancel** button (or the 'X' **Close** button in the upper-right-hand corner of the form), then on Line 70 we set 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
```

Main Place Boards Subroutine

Lines 79 through 126 are where we declare the subroutine that (a) determines the optimal (maximum) number of boards that will fit on the panel and (b) places the boards on the panel. This routine accepts two parameters: the path/name of the Expedition PCB board document and the required clearance value (these are identical to the *GetParameters()* routine, 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)
```

On Lines 81 and 82 we call our *GetBoardDimensions()* helper routine to determine the width and height of the board in Expedition PCB.

```
80      ' Get the board dimensions
81      Dim boardXDimReal, boardYDimReal
82      Call GetBoardDimensions(designNameStr, boardXDimReal,
                                boardYDimReal)
```

On Lines 85 and 86 we acquire the extrema of the panel border in FabLink XE.

```
84      ' Get the extrema of the panel border
85      Dim borderExtremaObj
86      Set borderExtremaObj = pnlDocObj.PanelBorder.Extrema
```

On Lines 90 and 91, we 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 -
91          borderExtremaObj.MinX
91      borderYDimReal = borderExtremaObj.MaxY -
92          borderExtremaObj.MinY
```

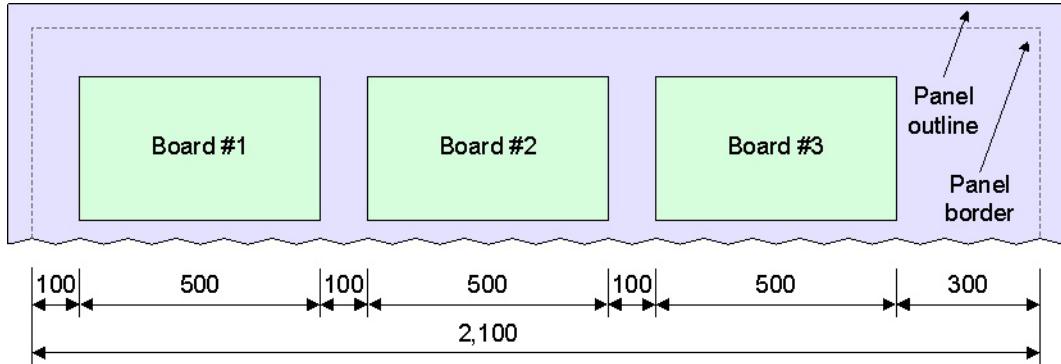
On Lines 95 and 96 we call our *FindBestMatrix()* utility routine to determine the optimal (maximum) number of boards that will fit on our panel. This function *returns* three parameters: the number of boards that will fit in the horizontal plane; the number of boards that will fit in the vertical plane; and the orientation/rotation (0 or 90, corresponding to 0° or 90°) 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,
96                          clearanceReal, borderXDimReal, borderYDimReal,
96                          numColInt, numRowInt, orientationInt)
```

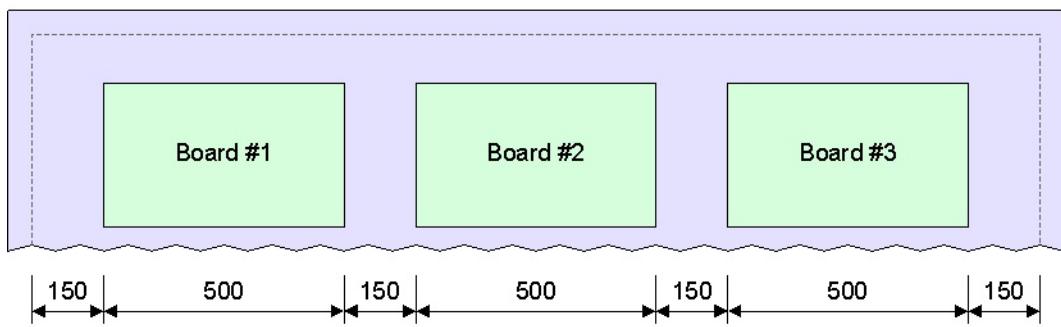
Now, before we proceed further, there are a couple of points we need to consider. The user originally specified some minimum required clearance value, which is used to fit the boards in both the horizontal and vertical directions. Based on the combination of this clearance value and the size of the boards, our *FindBestMatrix()* helper function has calculated the maximum number of boards that will fit in the horizontal direction and the maximum that will fit in the vertical direction.

It is extremely unlikely, however, that the size of the boards and the clearance values will exactly fit the space inside the panel border. Purely for the sake of an example, let's assume that the original clearance specified by the user is 100 units, that our boards are 500 units wide, and that the width of the panel (border to border) is 2,100 units. In this case, if we started with a clearance of 100 on the left of the panel, we would end up with an uneven distribution with a clearance of 300 on the right of the panel as illustrated in Figure 27-5(a).

For the purposes of this script, we decided that it would be better to equalize the clearances so as to provide a more uniform distribution as illustrated in Figure 27-5(b). Of course, we shall also want to apply a similar equalization with regard to the height of the panel, and it's unlikely that our new vertical clearances will be the same as the horizontal clearances. Thus, although we started with a single clearance value, we now need to generate new equalized clearance values in the X and Y dimensions (the variables used to hold these values in our script are called *xClearanceReal* and *yClearanceReal*, respectively).



(a) Using the original clearance "as-is" results in an uneven distribution



(b) Equalizing the clearance results in an even distribution

Figure 27-5. Equalizing the clearance.

One last point before we proceed ... in the following lines of code we will instantiate and use variables called `xOffset` and `yOffset`. In order to understand how these are used, consider the illustration in Figure 27-6. In this case, our new horizontal clearance value (`xClearanceReal`) is used to determine 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 our 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). (Of course, a similar treatment will be performed in the vertical direction using our new vertical clearance value).

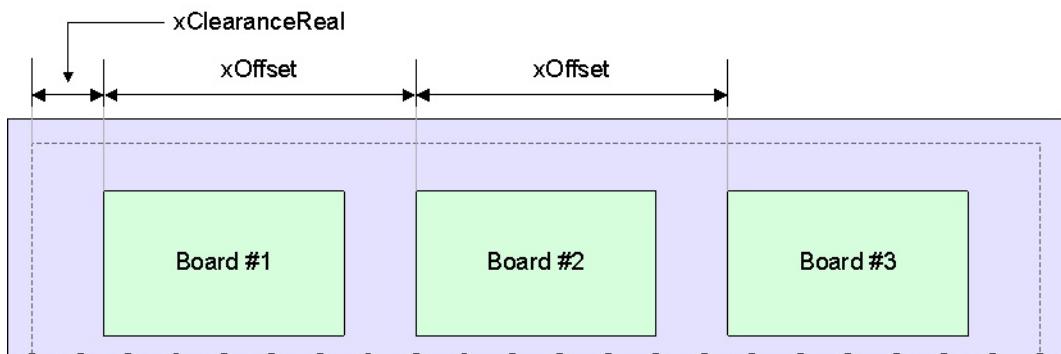


Figure 27-6. Understanding the concept of "offsets" as used by this script.

On Lines 100 and 101 we instantiate the variables we are going to use to hold our 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
```

On Line 102 we check to see if the orientation of the boards (as returned by our *FindBestMatrix()* utility routine) is 0 or 90, corresponding to 0° or 90°, respectively. If the orientation is 0, then on Lines 103 through 106 we calculate our new X and Y clearance values and X and Y offsets accordingly.

Observe that on Line 103 we call our *EqualizeClearance()* helper routine to calculate our new X clearance. The parameters we pass 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 our *FindBestMatrix()* utility routine.

Similarly, on Line 104 we call the same *EqualizeClearance()* helper routine to calculate our new Y clearance. In this case, the parameters we pass 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 our *FindBestMatrix()* utility routine.

On Line 105 we calculate the X offset as the sum of the X dimension (the width) of the board and our new X clearance. On Line 106 we calculate the Y offset as the sum of the Y dimension (the height) of the board and our new Y clearance.

```
102    If orientationInt = 0 Then
103        xClearanceReal = EqualizeClearance(boardXDimReal,
104                                         borderXDimReal, numColInt)
104        yClearanceReal = EqualizeClearance(boardYDimReal,
105                                         borderYDimReal, numRowInt)
105        xOffSet = boardXDimReal + xClearanceReal
106        yOffSet = boardYDimReal + yClearanceReal
```

By comparison, if the orientation of the boards is 90, then on Lines 108 through 111 we calculate our new X and Y clearance values and X and Y offsets accordingly.

In this case, we know that the boards are going to be rotated by 90°. Thus, when we call the *EqualizeClearance()* helper routine to calculate our new X clearance, we actually pass in the Y dimension (the height) of the board. Similarly, when we call the *EqualizeClearance()* helper routine to calculate our new Y clearance, we actually pass in the X dimension (the width) of the board. Furthermore, on Line 110 we calculate the X offset as the sum of the Y dimension (the height) of the board and our new X clearance; on Line 111 we calculate the Y offset as the sum of the X dimension (the width) of the board and our new Y clearance

```
107    Else
108        xClearanceReal = EqualizeClearance(boardYDimReal,
109                                         borderXDimReal, numColInt)
109        yClearanceReal = EqualizeClearance(boardXDimReal,
110                                         borderYDimReal, numRowInt)
110        xOffSet = boardYDimReal + xClearanceReal
111        yOffSet = boardXDimReal + yClearanceReal
112    End If
```

Finally, on Lines 116 through 125, we use the new X and Y clearance values and X and Y offsets we've just calculated in conjunction with two nested loops and the *PutBoard* method to actually add the boards into the panel.

```

113      ' Add the board matrix according to the
114      ' specified parameters.
115      Dim i, j, currXReal, currYReal
116      currYReal = borderExtremaObj.MinY + yClearanceReal
117      For i = 1 To numRowInt
118          currXReal = borderExtremaObj.MinX + xClearanceReal
119          For j = 1 To numColInt
120              Call pnlDocObj.PutBoard(designNameStr, currXReal,
121                                      currYReal, orientationInt, False,
122                                      epcbBoardAttachPointLowerLeft)
123          currXReal = currXReal + xOffset
124      Next
125      currYReal = currYReal + yOffset
126  Next
127 End Sub

```

Utility Functions and Subroutines

This is where we declare some utility routines. These routines are similar in context to our generic "helper" routines (see the next topic), except that they are specific to this script and may not be used elsewhere without some "tweaking".

FindBestMatrix() Subroutine

Lines 143 through 162 declare the routine that is used to determine the optimal (maximum) number of boards that can fit into a panel. Although we decided to present this as 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 we say this is that this routine doesn't actually understand anything about "boards" and "panels" per se. Instead, we provide it with the X and Y dimensions associated with some "item" – along with a required clearance value – and the X and Y dimensions of some "border". The routine then determines how many "items" (including clearance values) will fit within the confines of the "border". As part of this task, the routine performs its evaluations with the "items" presented at both 0° and 90° rotations to see which returns the best result (if the results are equal the routine defaults to the 0° 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 (these last three parameters are *In-Out* parameters whose values will be 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)

```

On Line 145 we commence by assuming that a rotation of 0° will give the best fit.

```
144      ' Try orientation of 0 degrees first.  
145      orientationInt = 0
```

Based on this assumption, on Line 146 we call our *FindMaxFit()* helper routine to determine the maximum number of columns; that is, the number of "items" (plus clearances) that will fit in the "border" in the horizontal direction. Due to the fact that we're currently working with a rotation of 0°, the first parameter to the *FindMaxFit()* helper routine is the X dimension (the width) of the "item".

Similarly, on Line 147 we call our *FindMaxFit()* helper routine to determine the maximum number of rows; that is, the number of "items" (plus clearances) that will fit in the "border" in the vertical direction. In this case, due to the fact that we're currently working with a rotation of 0°, the first parameter to the *FindMaxFit()* helper routine is the Y dimension (the height) of the "item".

```
146      numColInt = FindMaxFit(itemXDimReal, clearanceReal,  
147                                borderXDimReal)  
147      numRowInt = FindMaxFit(itemYDimReal, clearanceReal,  
                                borderYDimReal)
```

Next, we want to determine the result of rotating the items through 90°. In order to do this, on Line 152 we first instantiate 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
```

On Line 153 we call our *FindMaxFit()* helper routine to determine the maximum number of columns. In this case, due to the fact that we're assuming the "items" have been rotated by 90°, the first parameter to the *FindMaxFit()* helper routine is the Y dimension (the height) of the "item".

Similarly, on Line 154 we call our *FindMaxFit()* helper routine to determine the maximum number of rows. In this case, due to the fact that we're assuming the "items" have been rotated by 90°, the first parameter to the *FindMaxFit()* helper routine is the X dimension (the width) of the "item".

```
153      tempNumColInt = FindMaxFit(itemYDimReal, clearanceReal,  
                                      borderXDimReal)  
154      tempNumRowInt = FindMaxFit(itemXDimReal, clearanceReal,  
                                      borderYDimReal)
```

On Line 156 we check to see if the product of the number of rows and columns for a 90° rotation is greater than the number of rows and columns for a 0° rotation. If this is the case, then on Lines 157 through 159 we set the values of the column, row, and rotation parameters that will be returned by this routine to be the values associated with a 90° 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 is where we declare some generic "helper" routines. These routines are not focused on this script per se; they can easily be used in other scripts.

GetBoardDimensions() Subroutine

Lines 172 through 202 declare the routine that is used to determine the X and Y dimensions of the board in Expedition PCB. There are three parameters to this routine: the path/filename string for the board in Expedition PCB and the X/Y dimensions of the board (these last two parameters are In/Out parameters whose values will be assigned by this routine).

```
167  ' Returns the dimensions 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)
```

On Lines 173 and 174 we instantiate variables to hold our Expedition PCB *Application* and *Document* objects.

```
173      Dim pcbAppObj
174      Dim pcbDocObj
```

On Line 177 we create the Expedition PCB server. On Line 178 we prevent and annoying messages from being displayed.

```
176      ' Create Expedition.
177      Set pcbAppObj =
                  CreateObject("MGCPCB.ExpeditionPCBAplication")
178      pcbAppObj.Gui.SuppressTrivialDialogs = True
```

On Lines 181 and 182 we open the Expedition PCB *Document* object and license it (validate the server).

```
180      ' Open and license the document
181      Set pcbDocObj = pcbAppObj.OpenDocument(docNameStr)
182      ValidateServer(pcbDocObj)
```

On Line 185 we set the units associated with this Expedition PCB board to be the same as the units associated with the FabLink XE panel (see also the previous discussions associated with Line 35 in this script).

```
184      ' Set the units.
185      pcbDocObj.CurrentUnit = unitsEnum
```

On Lines 189 and 190 we acquire the manufacturing outline extrema associated with the board in Expedition PCB.

```
187      ' Use the manufacturing outline to calculate the
188      ' dimensions of the board.
189      Dim extremaObj
190      Set extremaObj = pcbDocObj.ManufacturingOutline.Extrema
```

On Lines 192 and 193 we 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
```

On Line 196 we use the *Quit* method to exit Expedition PCB.

```
195      ' Exit Expedetion
196      Call pcbAppObj.Quit()
```

On Lines 199 and 200 we explicitly release the Expedition PCB *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 we use to recalculate (equalize) the clearances. There are three parameters to this function: the size of a generic "item" (in the case of this script, this will be the width or height of the board, but it could be anything); the total size available (in the case of this script, this will be the border-to-border width or height of the panel, but it could be anything); and the number of "items" we want 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)
```

On Line 211 we calculate our new (equalized) clearance value.

```
211      EqualizeClearance = ( totalSizeReal - (itemSizeReal *
                                         numItemsInt) ) / ( numItemsInt + 1 )
212
213 End Function
```

Now, the formula used to calculate the new clearance value is very simple, but it can be tricky to understand when you first see it. Thus, may be useful to consider the illustration presented in Figure 27-7.

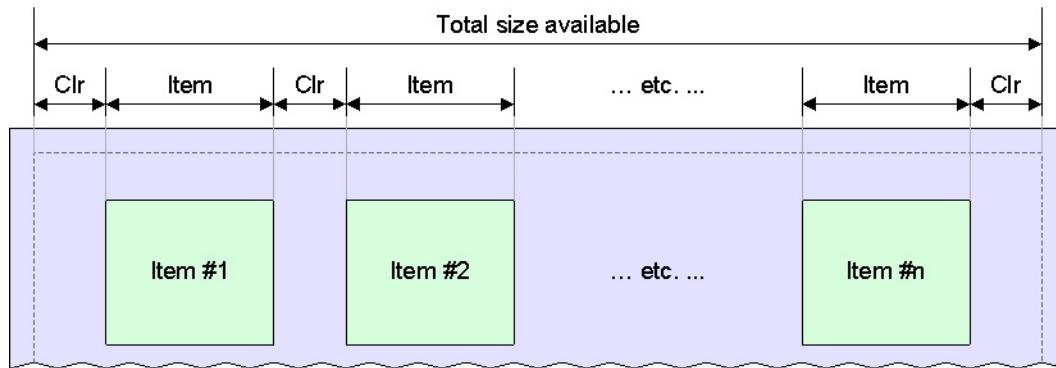


Figure 27-7. The basis for the "equalize clearance" algorithm.

Here's the way it works. We know the value of the total size available and we also know the size and number ('n') of the items. Thus, the space available for any clearances is calculated as follows:

$$\text{Remaining space available} = \text{Total size available} - (\text{size of items} \times n)$$

Furthermore, if we have 'n' items, then we are going to require '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). Thus, the size of the new clearance is calculated as follows:

$$\text{Clearance} = \text{Remaining space available} / (n + 1)$$

These two equations are captured in a single formula in Line 211 above.

FindMaxFit() Function

Lines 220 through 224 declare the routine that is used to determine that maximum number of items that will fit in a given distance with a specified clearance. There are three parameters to this function: the size of a generic "item" (in the case of this script, this will be 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 the case of this script, this will be 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)
```

On Line 222 we calculate the maximum number of "items" that will fit within the specified dimension.

```
221
222      FindMaxFit = Int((totalSizeReal - clearanceReal) /
                           (clearanceReal + itemSizeReal))
223
224  End Function
```

As with the previous routine, this is very simple, but it can be tricky to understand when you first see it. Thus, may be useful to consider the illustration presented in Figure 27-8.

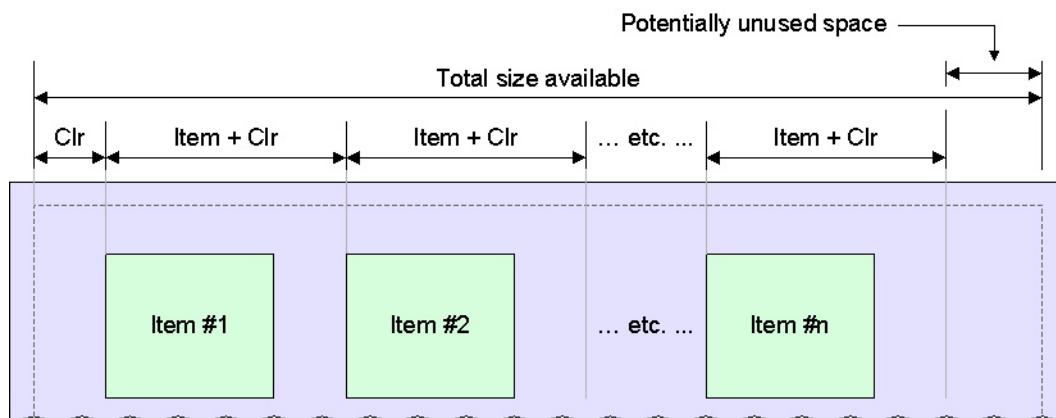


Figure 27-8. The basis for the "maximum number" algorithm.

Here's the way it works. We know the value of the total size available, the size of the "item", and the minimum required clearance. Purely for the sake of these discussions (and in the context of the diagram shown in Figure 27-8), let's assume that we are starting on the left-hand side of the total size available. As a starting point, we know that we must have a clearance on the left hand side, which means that the remaining space available is calculated as follows:

$$\text{Remaining space available} = \text{Total size available} - \text{Clearance}$$

Next, we know that for every "item" we must have an associated clearance on the right-hand side (again, we're using terms like "right-hand side" in the context of the diagram shown in Figure 27-8). Thus, we can calculate the number of items ("Real n") that will fit in the remaining space available as follows:

$$\text{Real n} = \text{Remaining space available} / (\text{Size of item} + \text{Clearance})$$

Of course, this will return some real number like 3.12 (hence our use of the term 'Real n'), where the fractional 0.12 equates to the "potentially unused space" shown in Figure 27.8. In order to determine the whole (integer) number of boards ('n'), we have to use the *Int()* function as follows:

$$n = \text{Int}(\text{Real n})$$

These three equations are captured in a single formula in Line 222 above.

FindInWDIR() Function

Our *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, because determining the path to the file is this routine's role in life).

```
226  ' Finds specified file in WDIR directory. Returns full path.  
227  ' filenameStr - String  
228  Function FindInWDIR(filenameStr)
```

On Line 229 we create a *FileSystemObject* object.

```
229      Dim fileSysObj: Set fileSysObj =  
              CreateObject("Scripting.FileSystemObject")
```

On Line 231 we specify the separator character (a semicolon) used in environment variables when running on the Windows® operating system.

```
231      Dim separatorStr: separatorStr = ";"
```

Just to refresh our minds, consider a simple WDIR value comprising two values (paths) separated by a semicolon as illustrated in Figure 27-9.

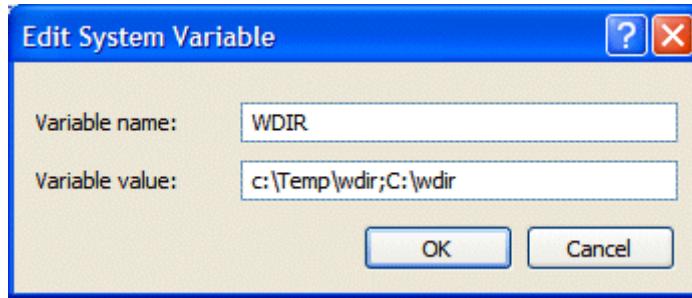


Figure 27-9. A simple WDIR example.

The problem is that the separator character employed by the UNIX or Linux operating systems is a colon. Thus, on Lines 232 through 234 we use the *Scripting.IsUnix* method to determine if we are running on UNIX or Linix and – if so – to change the separator character to be a colon (the *Scripting.IsUnix* method was introduced in *Chapter 18*).

```
232     If Scripting.IsUnix Then
233         separatorStr = ":" 
234     End If
```

On Line 235 we use our 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"), separatorStr)
```

On Lines 237 through 243 we iterate through all of these directory/folder paths checking to see if the required library file exists in any of them. As soon as we find the file we return its complete path/name and exit 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
```

Last but not least, if we don't find the specified library file in any of the WDIR directory/folder paths, then we assume that the *filenameStr* parameter to this function defined an absolute path to the library file, in which case this is the name we return as specified on Line 244.

```
244     FindInWDIR = filenameStr
245 End Function
```

The Dialog/Form Itself

With regard to the pop-up dialog/form that is launched by the main script, we are going to first create the dialog/form and then populate the various elements on the form with their portions of the script. The key scripting sections "behind" the form are as follows (these are shown in the order in which we will be creating them):

- **Initialization and Setup**
- **The "OK" Button Event/Click**
- **The "Cancel" Button Event/Click**

- The "Close" (Terminate) Event/Click

Adding Elements to the Form

Launch the IDE in form editor mode as discussed in *Chapter 4: Introducing the IDE* and then create the form as illustrated in Figure 27-10. Use the **File > Save As** command to save the form in whatever folder you're using to store your scripts; save the form with the name *PlaceBoardInput.efm* (as we discussed in *Chapter 4*, the *.efm* extension denotes a form, as compared to the *.vbs* extension used to denote a pure VBScript). As we start to add the code behind the various elements on the form, it's a good idea to keep on saving our work (trust us on this).

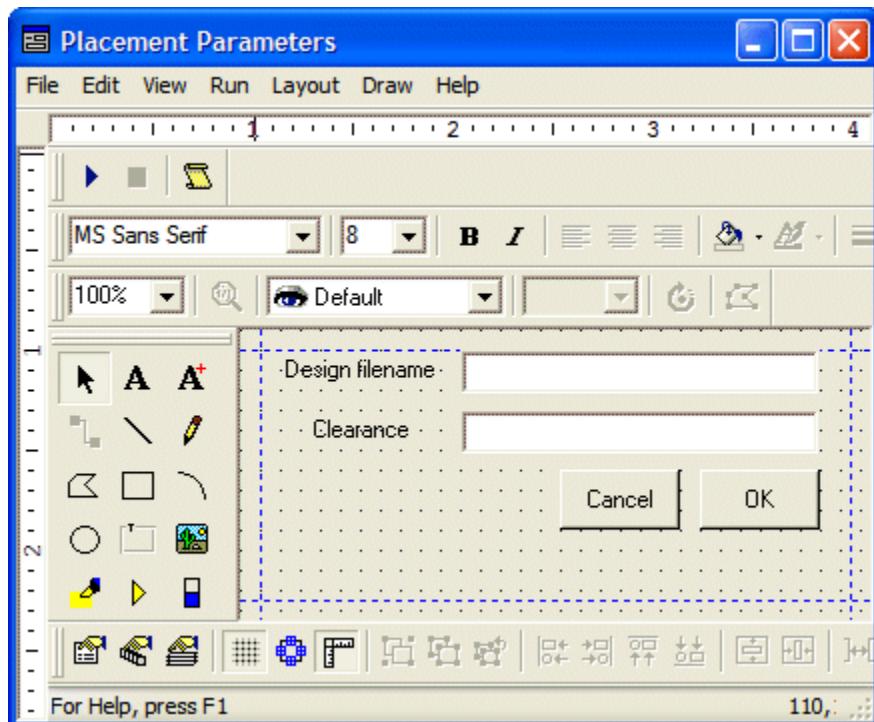


Figure 27-10. Creating the form/dialog.

Observe that there are two text edit/input boxes on this form – each has a free text item placed on its left-hand 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

Now, this bit is very important. We need to name any of the objects on the form with which we are going to associate scripting code. Right-click on the upper text edit/input boxes and select the **Object Properties** item from the resulting pop-up menu. Now change the assignment to the **(Object Code)** item from its default value to the name *fileNameEditObj* as illustrated in Figure 27-11.

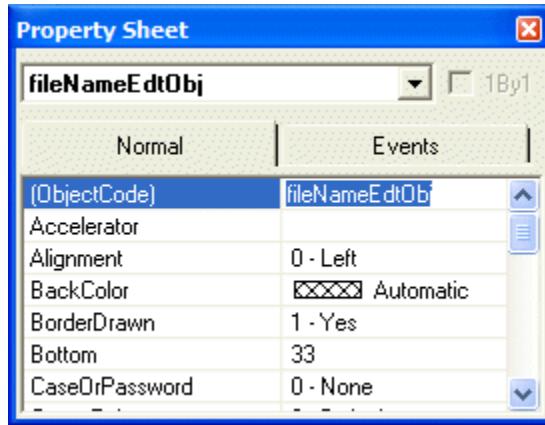


Figure 27-11. The Property Sheet dialog.

Observe that this name automatically appears in the field at the top of the form. This is the name that will be used by our script to access this form element. Once you've made this modification, click the red 'X' button located in the upper right-hand corner of the form to dismiss this dialog.

Repeat this process for the other items on the form as follows:

- Change the name of the lower text edit/input boxes control to *clearanceEditObj*.
- Change the name of the **Cancel** button to *cancelBtnObj*.
- Change the name of the **OK** button to *okBtnObj*.

Naming the Form Itself

Right-click on any blank area on the form and select the **Object Properties** item from the resulting pop-up menu. Now change the assignment to the **Title** item from its default value to *Placement Parameters* as illustrated in Figure 27-12.

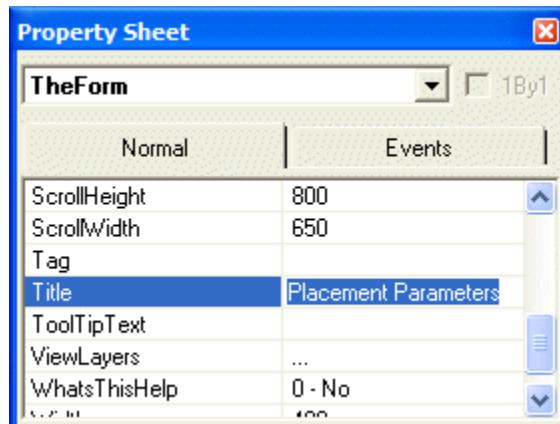


Figure 27-12. The Property Sheet dialog.

Once you've made this modification, click the red 'X' button located in the upper right-hand corner of the form to dismiss this dialog.

Preparing to Add Code "Behind" the Elements on the Form

Now that we've designed our form, it's time to add the code (sub-scripts) "behind" the various elements on the form. There are a number of mechanisms by which this may be achieved but, for our purposes here, simply click the **Script** icon and then make sure that the **Event View** icon is active as illustrated in Figure 27-13.

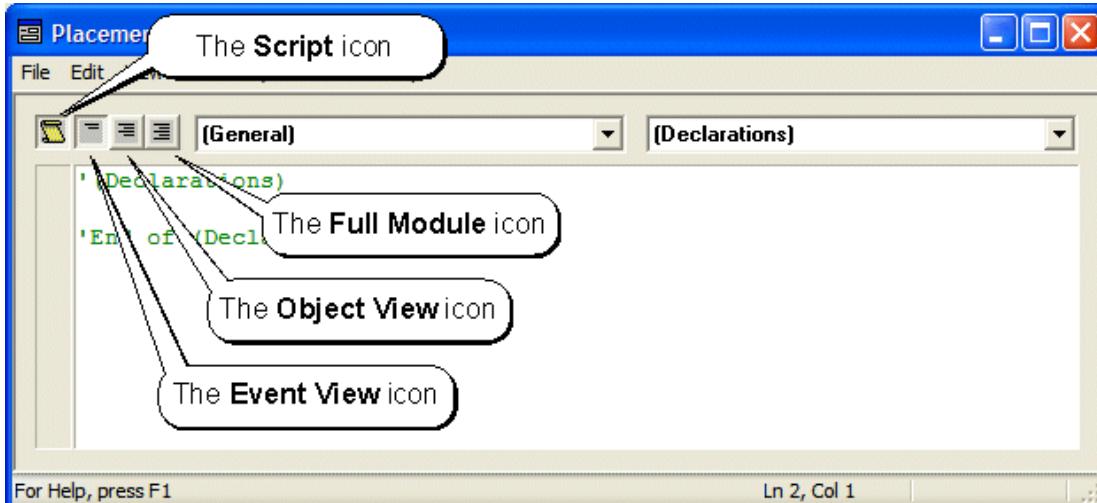


Figure 27-13. Preparing to add code "behind" the elements on the form.

Initialization and Setup

Ensure that **(General)** is selected in the left-hand (Object) list at the top of the form. This will cause the right-hand list to be automatically populated with the **(Declarations)** item as illustrated in Figure 27-14.

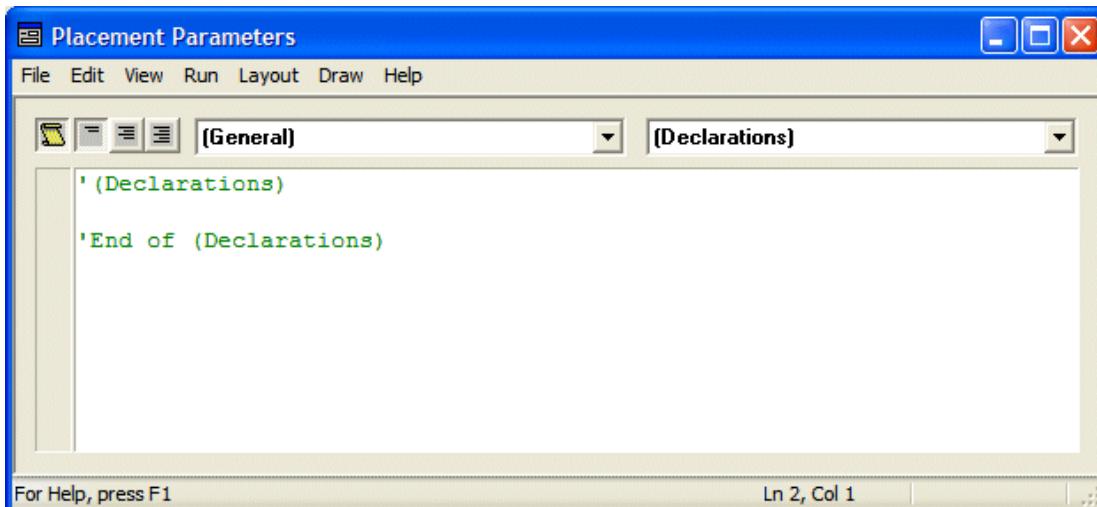
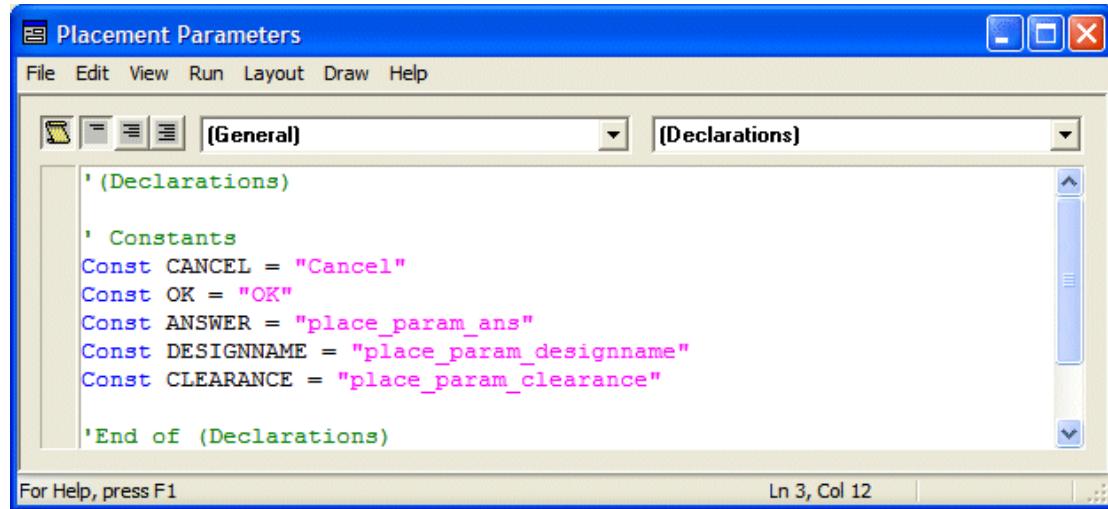


Figure 27-14. Preparing to add the initialization and setup code.

Place the cursor between the *'(Declarations)'* and *'End of (Declarations)'* comments and then enter the code shown in Figure 27-15.



The screenshot shows the 'Placement Parameters' dialog box. The title bar says 'Placement Parameters'. The menu bar includes File, Edit, View, Run, Layout, Draw, and Help. The left toolbar has icons for file operations. The top right shows standard window controls. The main area has two tabs: '(General)' and '(Declarations)'. The '(General)' tab is selected, displaying the following VBA code:

```

' (Declarations)

' Constants
Const CANCEL = "Cancel"
Const OK = "OK"
Const ANSWER = "place_param_ans"
Const DESIGNNAME = "place_param_designname"
Const CLEARANCE = "place_param_clearance"

'End of (Declarations)

```

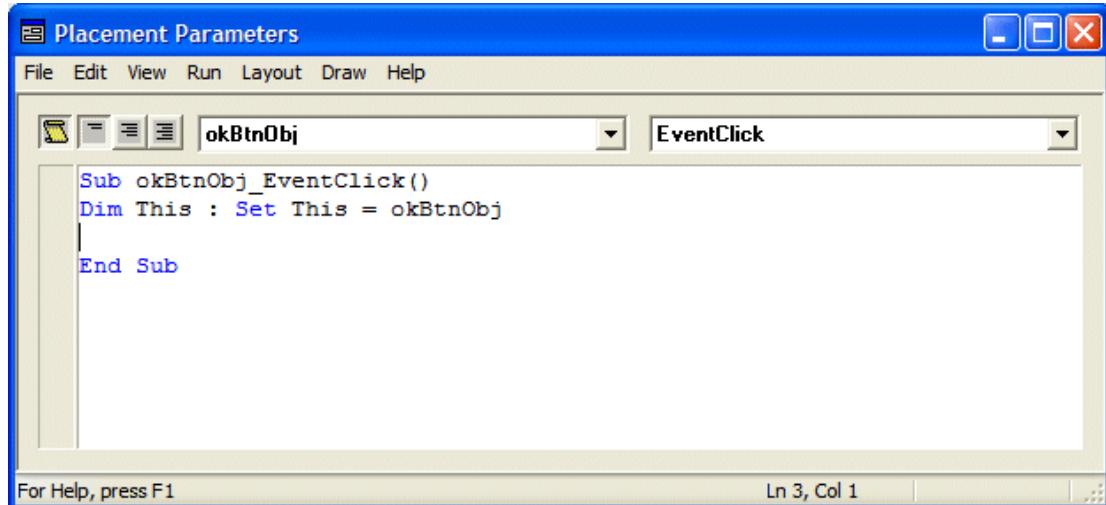
At the bottom, status bars say 'For Help, press F1' and 'Ln 3, Col 12'.

Figure 27-15. The initialization and setup code.

Observe that these are exactly the same constant values that we declared in the main script (remember that these are the keys we are using to implement script/form-to-script/form communications by means of the *Scripting.Globals Dictionary* object).

The OK Button Event/Click

Ensure 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 will be as illustrated in Figure 27-16.



The screenshot shows the 'Placement Parameters' dialog box. The title bar says 'Placement Parameters'. The menu bar includes File, Edit, View, Run, Layout, Draw, and Help. The left toolbar has icons for file operations. The top right shows standard window controls. The main area has two tabs: '(General)' and '(Declarations)'. The '(General)' tab is selected, displaying the following VBA code:

```

Sub okBtnObj_EventClick()
Dim This : Set This = okBtnObj

End Sub

```

At the bottom, status bars say 'For Help, press F1' and 'Ln 3, Col 1'.

Figure 27-16. Preparing to add the OK button code.

Place your cursor on the blank line between the *Dim* and *End Sub* statements and then enter the code shown in Figure 27-17.

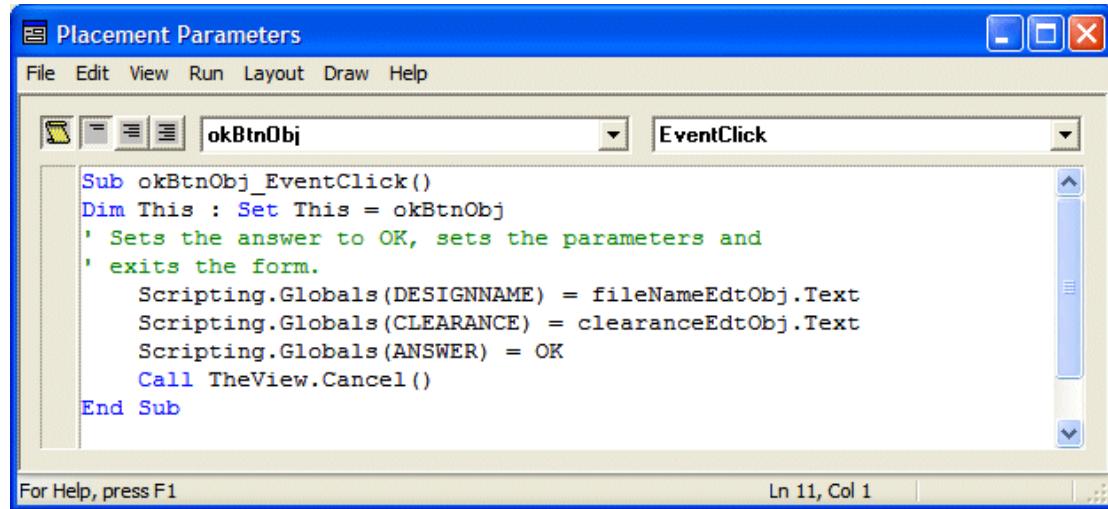


Figure 27-17. The OK button code.

Observe that we first use the *Scripting.Globals Dictionary* object to associate the *DESIGNNAME* key with the contents of whatever text string the user enters into the upper text edit/input box; next we associate the *CLEARANCE* key with the contents of whatever text string the user enters into the lower text edit/input box; next we associate the *ANSWER* key with a text string value of "OK"; and finally we use the *Cancel* method associated with *TheView* to exit the form (where *TheView* refers to the form itself; see also the notes associated with the *Close/Terminate* button below).

The Cancel Button Event/Click

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 will be as illustrated in Figure 27-18.

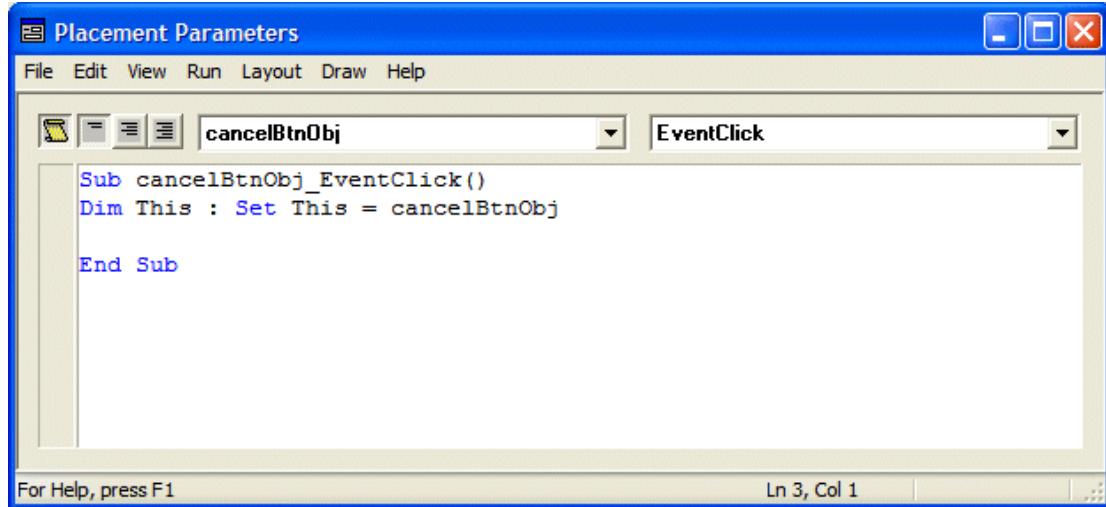
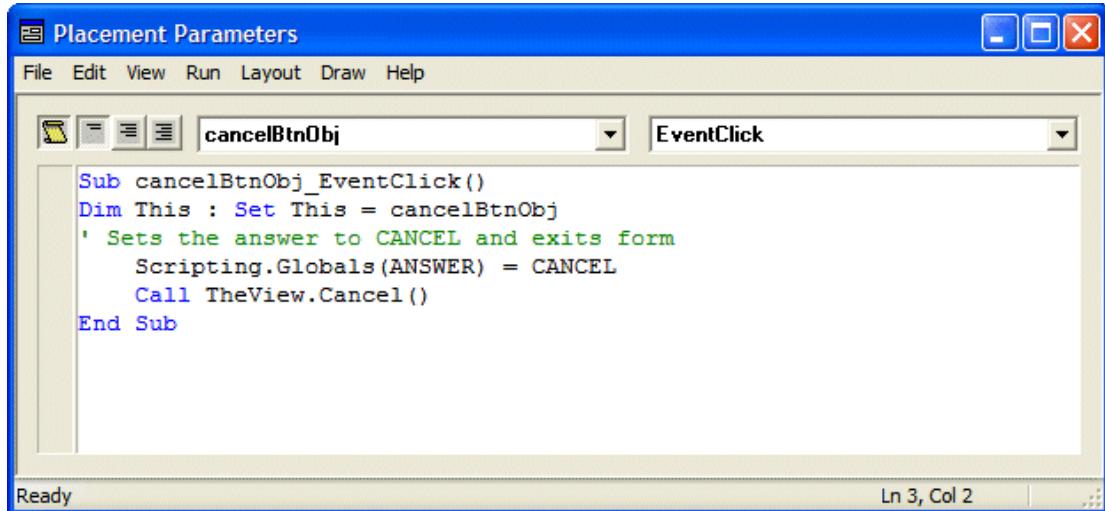


Figure 27-18. Preparing to add the Cancel button code.

Place your cursor on the blank line between the *Dim* and *End Sub* statements and then enter the code shown in Figure 27-19.



The screenshot shows the 'Placement Parameters' dialog box. The left-hand list contains 'cancelBtnObj'. The right-hand list is set to 'EventClick'. The code editor displays the following VBA code:

```

Sub cancelBtnObj_EventClick()
Dim This : Set This = cancelBtnObj
' Sets the answer to CANCEL and exits form
    Scripting.Globals(ANSWER) = CANCEL
    Call TheView.Cancel()
End Sub

```

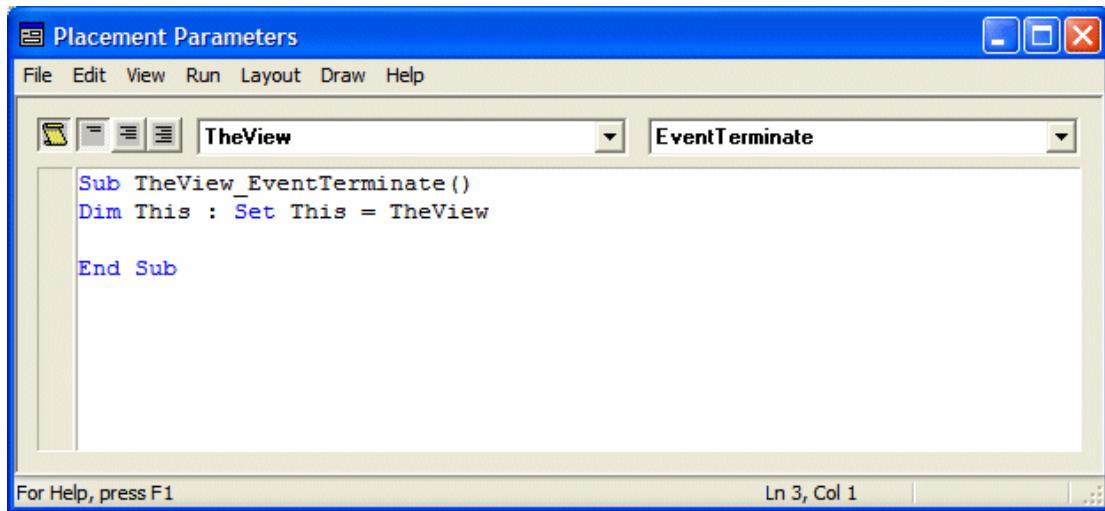
The status bar at the bottom indicates 'Ready' and 'Ln 3, Col 2'.

Figure 27-19. The OK button code.

In this case, we first use the *Scripting.Globals Dictionary* object to associate the *ANSWER* key with a text string value of "Cancel". Next, we use the *Cancel* method associated with *TheView* to exit the form (where *TheView* refers to the form itself; see also the notes associated with the *Close/Terminate* button below).

The Close (Terminate) Button Event Click

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 will be as illustrated in Figure 27-20.



The screenshot shows the 'Placement Parameters' dialog box. The left-hand list contains 'TheView'. The right-hand list is set to 'EventTerminate'. The code editor displays the following VBA code:

```

Sub TheView_EventTerminate()
Dim This : Set This = TheView

End Sub

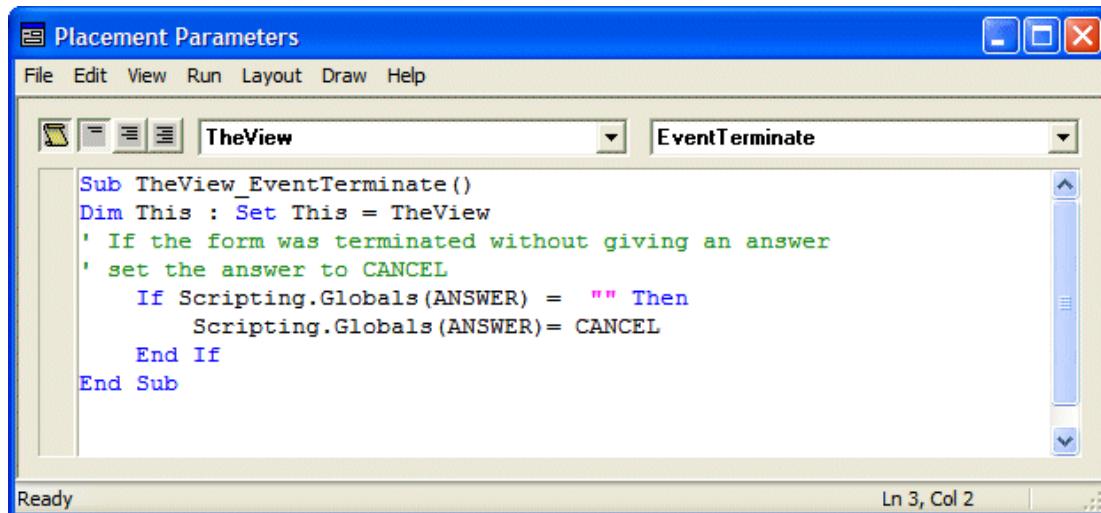
```

The status bar at the bottom indicates 'For Help, press F1' and 'Ln 3, Col 1'.

Figure 27-20. Preparing to add the Close (Terminate) button code.

The *EventTerminate* event associated with *TheView* is called whenever we exit the form, irrespective of the mechanism (in our case, this means that this event will be called whenever the 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).

Place your cursor on the blank line between the *Dim* and *End Sub* statements and then enter the code shown in Figure 27-21.



The screenshot shows a Windows application window titled "Placement Parameters". The menu bar includes File, Edit, View, Run, Layout, Draw, and Help. The title bar has tabs for "TheView" and "EventTerminate". The main area contains the following VBA code:

```
Sub TheView_EventTerminate()
Dim This : Set This = TheView
' If the form was terminated without giving an answer
' set the answer to CANCEL
If Scripting.Globals(ANSWER) = "" Then
    Scripting.Globals(ANSWER)= CANCEL
End If
End Sub
```

The status bar at the bottom right shows "Ln 3, Col 2".

Figure 27-21. the Close (Terminate) button code.

Observe that we first use the *Scripting.Globals Dictionary* object to check to see if the *ANSWER* key is associated with an empty string. If so, this means that the user must have clicked the **Close** (Terminate) in the upper-right-hand corner of the form, in which case we associate the *ANSWER* key with whatever string we assigned to our *CANCEL* constant ("Cancel" in this script and form).

Creating and Testing the Script

- 1) Use the scripting editor of your choice to enter the script described above (including the license server function which is not shown above) and save it out as *PlaceBoards.vbs*.
- 2) Use the IDE to create the dialog/form described above and save it out 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 Expedition PCB and FabLink XE that are currently running, and then launch a new instantiation of FabLink XE.
- 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 middle of the FabLink XE 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 would be a good idea for you to practice your scripting skills by implementing these examples):

- In the case of the pop-up dialog/form, you could add some error checking to ensure that the user enters appropriate values (that the value entered into the **Clearance** text edit/input box is a number, for example).
- In the case of the pop-up dialog/form, you have to specify the full path to the Expedition PCB board design. It would be useful to add a **Browse** button to this dialog/form that allowed you to easily locate and select a board design file (this is relatively easy on Windows® because there's already an automation server **Browse** dialog available; implementing this functionality on UNIX or Linux will be a little more challenging – enjoy!)
- Our script presents all of the boards in the same orientation; that is, either 0° or 90°. However, there could be a process requirement to have different boards in different rotations. Consider our default *CandyPanel* panel, for example, in which the top row of boards are presented with 0° rotation and the bottom row of boards are presented with 180° rotation as illustrated in Figure 27-22.

Thus, 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 0° rotation, for example, then it could flip the boards on odd rows to have a rotation of 180°; or it could flip the boards on odd columns to have a rotation of 180°; or it could create a checkerboard pattern of boards at 0° and 180°.

Alternatively, if the script initially decided that the optimal fit was N-rows by M-columns with a 90° rotation, then it could flip the boards on odd rows to have a rotation of 270° (or -90°) or it could flip the boards on odd columns to have a rotation of 270° (or -90°); or it could create a checkerboard pattern of boards at 90° and 270°.

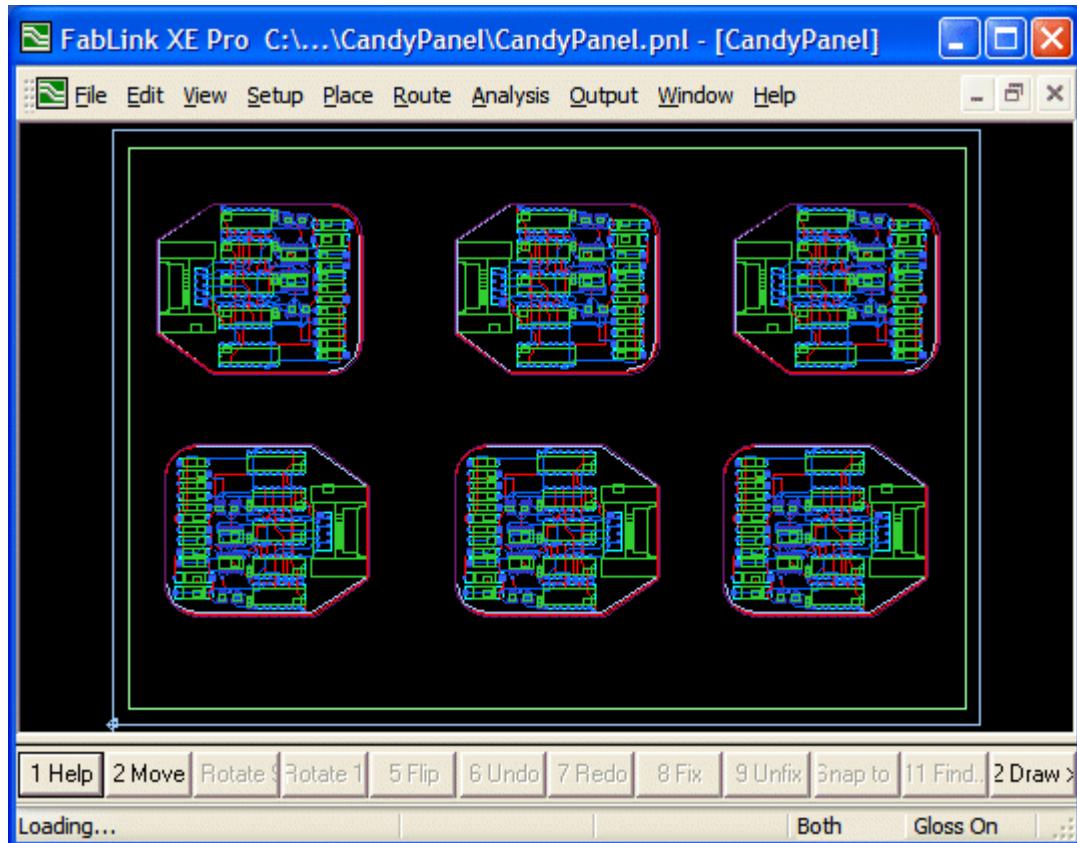


Figure 27-22. Our pre-defined CandyPanel panel.

As another possibility, 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.

- Last but not least, 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 0° board rotation while others used a 90° board rotation (it would be best to implement this using a loop that simply tried boards with differing rotations as opposed to a formula as is the case in the existing script).

Appendix A: Good Scripting Practices

Introduction

The concept of "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 insert or omit a space between a method name and the parenthesis surrounding any parameters to be used with that method).

In such a case, it really doesn't matter which option you choose, the important thing is to be consistent. The recommended practices in this appendix have been 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 created by others (assuming, of course, that they also opt to use these practices).

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

VBScript is *not* a strongly typed language. For example, you may declare a variable and initially assign an integer value to it. Some time later, you may assign a real number to the same variable, or even a string. Having said this, when you declare a variable, you typically have a certain type of assignment in mind. Thus, 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 by this tutorial and used in our scripts:

Bool	Boolean (e.g. 0 / 1 or True / False)	' Primitive type
Int	Integer (e.g. 10)	' Primitive type
Real	Real number (e.g. 3.142)	' Primitive type
Str	String (e.g. "Hello World!")	' Primitive type
Enum	Enumerate	' Special primitive (named integer)
Obj	Object	' Object type
Coll	Collection (of objects)	' Object type
Cls	Class (user-defined object/structure)	' Object type
Const	A variable assigned a constant value	' Object type

For example, a *Trace* object could be named something like *traceObj*, while a collection of *Trace* objects might be called *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. Due to the fact that they are used so often, 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, if we want to create a *Trace* object that will be used as part of a function to find the longest trace in a design, we might call it something like:

```
longestTraceObj
```

Last but not least, if the variable is intended to represent an array, then the string *Arr* should be appended to the end; for example, *nameStr* would indicate a scalar variable to which we might assign a single string, while *nameStrArr* implies 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. Having the first letter uppercase makes it easy to differentiate between variable and function/subroutine names at a glance. For example, assume that we've already declared a variable called *longestTraceObj*. Also assume that we've declared a trace collection called *traceColl* and that we've already assigned a collection of traces to this variable. Now consider the following statement:

```
Set longestTraceObj = FindLongestTrace(traceColl)
```

In this case, the name of our function (whose contents would be described elsewhere) is *FindLongestTrace()*; observe the use of the initial uppercase letter.

Method and Property Names

Observe the following example lines of code (assume *pcbDocObj* has previously been declared and had a layout document assigned to it).

```
Dim viasColl  
Set viasColl = pcbDocOBJ.Vias
```

In particular, observe the *Vias* property. The first letter of *Vias* is in uppercase, because *Vias* is a property that is defined in the interface. It is recommended that the first letter of any method or property is capitalized because they are capitalized in the interfaces and in the documentation.

Event-Handler Function Names

Assume that we have already instantiated a variable called *pcbDocObj* and that we have assigned a *Document* object to this variable.

Now, assume that we are interested in the events that are associated with the *Document* object we assigned to the *pcbDocObj* variable. In this case, we would use the *AttachEvents* method associated with the *Scripting* object as follows:

```
Call Scripting.AttachEvents(pcbDocObj, "pcbDocObj")
```

This method instructs the script engine that we are interested in any events associated with the object that is specified by its first parameter. In this case, we're using the *Document* object variable *pcbDocObj* as this parameter.

Observe that the second parameter to the *AttachEvents* method is a string. This string will eventually be used as a common prefix for any of the event-handler functions we declare in the future that are associated with events originating from the object assigned to the *pcbDocObj* variable. The string itself is arbitrary – you could call it "Fred" if you wanted – however, it is recommended programming practice to set this string to be identical to the name of the first parameter. Thus, as the first parameter in our example is the *pcbDocObj* variable, we will use "pcbDocObj" as our string.

Finally, assume that we want to declare an event-handler function to process the event that occurs just prior to a document closing. This is known as the *OnPreClose* event. Consider the dummy event-handler function shown below:

```
Function pcbDocObj_OnPreClose()
    ' Some statements go here
End Function
```

The name of the event-handler function – *pcbDocObj_OnPreClose()* in this example – comprises three parts as follows:

- The string we previously specified as the second parameter to the *AttachEvents* method (this was the "pcbDocObj" string).
- An underscore '_' character.
- The name of the event as defined in the Expedition Automation Help documentation, which is *OnPreClose* in this example.

Object Names in IDE Forms

The names of objects used in IDE forms are similar to the names of variables. When in the IDE's form editor mode, you can add controls such as buttons, radio buttons, combo boxes, check boxes, and list boxes to your form. For example, in the case of a *Button* object whose purpose was to count all of the vias in a design, such a button may be named *countAllViasButObj* (standing for "Count All Vias Button Object").

The following abbreviations are recommended for objects used in forms:

- *BtnObj* = "Button object"
- *RadObj* = "Radio button object"
- *CmbObj* = "Combo-box object"
- *ChkObj* = "Check-box object"
- *LstObj* = "List box object"

File Names for Scripts and Forms/Dialogs

File names for scripts written in VBScript have the extension **.vbs*. File names for forms/dialogs have the extension **.efm*. The main body of the file name should follow the same rules as for functions and subroutines; that is, the first letter of the file name should be uppercase and each new sub-word in the variable name should be uppercase; for example:

HelloWorld.vbs

ViaCountForm.efm

Furthermore, in the case of a library (see Appendix C: Creating and Using a Library), it is recommended to use the *Lib* qualifier as part of the file name, because this informs us that this is going to act as a library rather than a standalone script; for example:

MenuLib.vbs

Option Explicit

By default, VBScript does not require the use of *Dim* (dimension) statements. Thus, in the case of the following two statements, for example, the *Dim* statement is optional:

```
Dim floppyInt  
floppyInt = 10
```

The problem with this scheme is that it's easy for misspelled variable names to creep into a script. For example, suppose we want to assign a new value of 20 to our *floppyInt* value as follows:

```
flooopyInt = 20
```

Do you notice anything funny here? In fact, we've misspelled the variable name (it used to be *floppyInt* and now it's *flooopyInt*). The point is that, by default, VBScript will accept this new statement and simply create a new variable.

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  
  
Dim floppyInt  
floppyInt = 10  
:  
etc.
```

Using this option means that you are forced to use *Dim* statements to declare your variables. In turn, this means that if you use a misspelled variable name, the script host will report an error and guide 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 the ' character be followed by a space character.

Parentheses

In the case of any function, method, or property that requires parameters, VBScript allows you to enclose these parameters in parentheses or to omit the parentheses if you require. For example, both of the following statements will be accepted by VBScript:

```
MsgBox "Hello World"      ' This is not recommended  
MsgBox("Hello World")    ' This IS recommended
```

For the sake of consistency you should always use one of these techniques throughout your script(s); we recommend the use of parentheses.

Space Characters

Space characters are not allowed in variable names. However, VBScript allows the optional use of a space between the name of a function, method, or property and the opening parenthesis enclosing any parameters. For example, both of the following statements will be accepted by VBScript:

```
MsgBox ("Hello World")      ' This is not recommended  
MsgBox("Hello World")      ' This IS recommended
```

For the sake of consistency you should always use one of these techniques throughout your script(s); we recommend not using a space character.

Dim versus Dim()

Consider the three statements shown below:

```
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) would appear to be a scalar (single) variable; the second statement (b) instantiates a two-dimensional static array whose size cannot be changed; the third statement (c) instantiates a dynamic array whose size can subsequently be changed using a *ReDim* statement.

The problem here is that the first and third statements are functionally identical; that is, it's possible to subsequently 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 will assume that *tempAStr* is a scalar variable. Instead, if you do want to use a dynamic array, you should use parentheses when dimensioning the variable with the *Dim* statement [as shown with the *tempCStr()* example above] so as to make it obvious to anyone reading the script that this is an array whose size will be specified in the future.

Appendix B: Recommended Script Architecting Process

Introduction

When you come to create a new script, there are a number of tricks and techniques you can use to architect it in such a way as to make it easier to understand, use, re-use, and maintain.

Creating a Template

One of the most efficacious things you can do is to create a standard template that you use as the starting point for most of your scripts. A simple example would be as shown below:

```
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( "MGCPBC.ExpeditionPCBAApplication" )
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
25 ' Include File helper functions go here (see notes)
26
27 ' Server validation function goes here
28 :
29     Etc.
```

Observe the comment on Line 41. This is where the "Include File" helper functions will go (these functions are described in *Appendix C: Creating and Using a Library*).

Re-using Functions and Subroutines

The idea here is to keep from constantly "re-inventing the wheel". This can best be achieved by creating libraries of commonly used functions and subroutines and then calling these routines from your scripts (see also *Appendix C: Creating and Using a Library*). The two main advantages to this approach are as follows:

- Modifying a single routine allows you to fix bugs across multiple scripts.
- Using well-defined and well-known library routines across multiple scripts can make those scripts easier to read and maintain.

Internal versus External Scripts

As a "rule-of-thumb" you should always try to use internal scripts because these have the best performance. Note, however, that it is possible to connect to Expedition PCB using an external script and to then execute an internal script to do all of the 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, ad infinitum. This technique allows you to quickly and easily pull multiple scripts and forms together so as 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

```
1 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 above, the parameter ("MyGlobalName" in this example) is an arbitrary string (or string variable or string constant) defined by you. Not surprisingly, you need to choose a unique name, because this can be used by anyone.

Delivering Scripts

The most flexible way to add functionality to an application like Expedition PCB is via the use of one or more startup scripts. There are several approaches here; for example:

- You can create one startup script that adds a group of menu buttons/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 button/item).
- You can create a number of startup scripts, each of which adds an individual menu button/item or defines an accelerator key, and each of which defines the functionality associated with that particular entity.

The authors of this tutorial prefer the second approach (keeping the menu or key-binding together with its functionality), but you must make your own decision as to 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 customization, this might be the best approach.

The most flexible place to put these scripts is in a WDIR directory/folder. Furthermore, 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 will automatically be picked up by all of the users (see *Chapters 9 and 10* for more details on using the WDIR environment variable).

Appendix C: Creating and Using a Library

Introduction

Way back in the mists of time we used to know as *Chapter 13*, we created two rather useful helper functions called *AddMenuAfter()* and *GetMenuNumber()*. These routines greatly eased the task of adding a new button/item to a pull-down menu.

Later, in *Chapter 21*, we reused these helper functions. The way in which we (the authors of this tutorial) did this was to return to the original script from *Chapter 13*, take a copy of these routines, and paste them into the new script associated with *Chapter 21*.

There are two main problems with this approach as follows:

- As you create more and more helper-type functions, it becomes 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've copied into multiple scripts, it can be extremely boring and time-consuming to fix the same bug over and over again. It can also be difficult to remember in which scripts the routine was used, and it is easy to lose consistency (that is, to "fix" the bug in different ways in different scripts) – this can really come back to haunt you later on.

The solution is to create a library of helper functions and subroutines that you can then call from multiple scripts. In this case, if you detect a bug, a single modification to the library routine will fix all of the scripts calling that routine. Another advantage of this approach is that using well-defined and well-known library routines across multiple scripts can make those scripts easier to read and maintain.

Creating a Library

One problem is that VBScript does not inherently support the concept of libraries per se, but we won't let this slow us down. Before you do anything else, create a directory/folder called *\Lib* (for library) somewhere on your system or on a server. The main point is that this folder should be in a path that is defined by your WDIR environment variable (see *Chapters 9* and *10* for more details on using the WDIR environment variable).

Next, consider the *AddMenuAfter()* and *GetMenuNumber()* helper functions mentioned in the previous topic (see *Chapters 13* and *21* for more details on how these routines perform their magic).

We are going to gather these functions into a single file as shown below and then save this file with the name *MenuLib.vbs* (which stands for "Menu Library") in our *\Lib* directory/folder. This file is what we are going to regard as being a library (using the *Lib* qualifier as part of the file name is recommended because it informs us that this is going to act as 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,
```

```

15      If entryNumInt > menuBarObj.Controls.Count Then
16          entryNumInt = -1
17      End If
18
19      Set AddMenuAfter =
20          menuBarObj.Controls.Add(cmdControlButton,,,entryNumInt)
21  End Function
22
23  ' Function that returns a menu index for a given
24  ' menu entry string.
25  ' menuToFindStr - String
26  ' menuBarObj - CommandBarSrv menu object
27  Function GetMenuNumber(menuToFindStr, menuBarObj)
28      Dim ctrlColl : Set ctrlColl = menuBarObj.Controls
29      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 particular case, we know that these routines will always be used together (not the least that the *AddMenuAfter()* routine calls the *GetMenuNumber()* routine). However, even in the case of routines that don't call each other, it may make sense to gather them all into a single file if they serve a common or related purpose. If, for example, we created 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 might be a function to determine if a menu button/item has already been added).

Calling Routines from a Library

Assuming we've created a library as discussed in the previous topic, we now want to call these library routines from a script. As a simple example, consider the following script. All this script does is to add a new **Hello** button/item to the **View** pull-down menu. When the user subsequently executes this new **View > Hello** command, a *MsgBox()* will appear on the screen saying "Hello".

Obviously this isn't a very realistic example, but it will serve to demonstrate the process of calling routines from our library. First, we'll show this script in its entirety (apart from the standard *ValidateServer()* function), and then we'll focus on items of interest.

```

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 =
22                         pcbAppObj.Gui.CommandBars("Document Menu Bar")
23
24 ' Get the view menu
25 Dim viewMenuObj
26 Set viewMenuObj = docMenuBarObj.Controls.Item("&View")
27
28 ' Call the library function
29 Dim helloBtnObj
30 Set helloBtnObj = AddMenuAfter("Mouse Mapping", viewMenuObj)
31
32 helloBtnObj.Caption = "Hello"
33 helloBtnObj.Target = ScriptEngine
34 helloBtnObj.ExecuteMethod = "OnHello"
35
36 Scripting.DontExit = True
37
38 Function OnHello(id)
39     MsgBox("Hello")
40 End Function
41
42 '-----'
43 'Helper functions
44
45 ' Includes the contents of WDIR file.
46 ' fileNameStr - String
47 Sub Include(fileNameStr)
48     Dim fileSysObj: Set fileSysObj =
49                         CreateObject("Scripting.FileSystemObject")
50     Dim fileObj: Set fileObj =
51         fileSysObj.OpenTextFile(FindInWDIR(fileNameStr, fileSysObj), 1)
52     Dim scriptStr: scriptStr = fileObj.ReadAll
53     Call ExecuteGlobal(scriptStr)
54 End Sub
55
56 ' Finds specified file in WDIR directory. Returns full path.
57 ' filenameStr - String
58 ' fileSysObj - FileSystemObject Object
59 Function FindInWDIR(filenameStr, fileSysObj)
60     Dim separatorStr: separatorStr = ";"
61     If Scripting.IsUnix Then
62         separatorStr = ":"
63     End If
64     Dim wdirArr:
65     wdirArr = Split(Scripting.GetEnvVariable("WDIR"),
66                           separatorStr)
67     Dim i
68     For i = 0 To UBound(wdirArr)
69         Dim pathStr: pathStr = wdirArr(i) & "/" & filenameStr
70         If fileSysObj.FileExists(pathStr) Then
71             FindInWDIR = pathStr
72             Exit Function
73         End If

```

```

70      Next
71      FindInWDIR = filenameStr
72  End Function
73
74  ' Server validation function
75  :
76  etc.

```

Now, we aren't going to discuss the way in which we add and configure new menu entries here (see *Chapters 6, 13, and 21* for more details). All we need note is that we call our *AddMenuAfter()* library helper function on Line 29.

```

27  ' Call the library function
28  Dim helloBtnObj
29  Set helloBtnObj = AddMenuAfter("Mouse Mapping", viewMenuObj)

```

So, how did we manage to do this? Well, first consider Line 3 where we call a local *Include()* helper function and pass it the name of the library file we want to open. In this case we are passing it the string "Lib\MenuLib.vbs", which says we are interested in the *MenuLib.vbs* file in a *\Lib* directory/folder (we're assuming that this *\Lib* folder is in a path that is defined by your WDIR environment variable). As we shall see, we could also specify an absolute path/filename if we wanted to do so.

```

1  Option Explicit
2
3  Include("Lib\MenuLib.vbs")

```

Just in case you were wondering, the reason we've created our own *Include()* helper function is that VBScript doesn't support the "include" concept.

In a moment we're going to consider the two local helper functions: *Include()* [Lines 44 through 81] and *FindInWDIR()* [Lines 56 through 72]. Before we do so, we should note that it is strongly recommended that you incorporate these routines in all of your scripts (the best way to do this is to add them into your "Template Script" as discussed in *Appendix B*).

Include() Function

This function accepts a single parameter, which is the name of the library file we want to open. As we shall see, the way in which we've implemented these helper routines means that 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 we create a *FileSystemObject* object.

```

47      Dim fileSysObj: Set fileSysObj =
                  CreateObject("Scripting.FileSystemObject")

```

On Line 48 we call our *FindInWDIR()* local helper function to locate the required file and return its path, which is then used by the *OpenTextFile* method (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)

```

On Line 49 we read the entire contents of this file into a string variable we've called *scriptStr*.

```
49      Dim scriptStr: scriptStr = fileObj.ReadAll
```

On Line 50 we use the built-in VBScript *ExecuteGlobal()* routine to execute the code from our library file.

```
50      Call ExecuteGlobal(scriptStr)
51  End Sub
```

FindInWDIR() Function

Our *FindInWDIR()* helper function accepts two parameters. The first is typically a relative path to a library file in a directory/folder specified by the WDIR environment variable; as we shall see, however, this first parameter may also be an absolute path to a library file if required. The second parameter is the *FileSystemObject* object we instantiated in our *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 we specify the separator character (a semicolon) used in environment variables when running on the Windows® operating system.

```
57  Dim separatorStr: separatorStr = ";"
```

Just to refresh our minds, consider a simple WDIR value comprising two values (paths) separated by a semicolon as illustrated in Figure C-1.

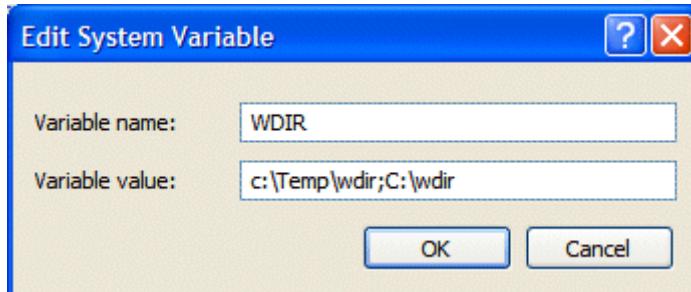


Figure C-1. A simple WDIR example.

The problem is that the separator character employed by the UNIX or Linux operating systems is a colon. Thus, on Lines 58 through 60 we use the *Scripting.IsUnix* method to determine if we are running on UNIX or Linix and – if so – to change the separator character to be a colon (the *Scripting.IsUnix* method was introduced in *Chapter 18*).

```
58  If Scripting.IsUnix Then
59      separatorStr = ":""
60  End If
```

On Lines 61 and 62 we use our separator character to split the WDIR environment variable and to create an array of directory/folder paths.

```
61  Dim wdirArr:
62  wdirArr = Split(Scripting.GetEnvVariable("WDIR"),
                  separatorStr)
```

On Lines 64 through 70 we iterate through all of these directory/folder paths checking to see if the required library file exists in any of them. As soon as we find the file we return its complete path/name and exit 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
```

Last but not least, if we don't find the specified library file in any of the WDIR directory/folder paths, then we assume that the *filenameStr* parameter to this function defined an absolute path to the library file, in which case this is the name we return as specified on Line 71.

```
71      FindInWDIR = filenameStr
72  End Function
```



Note: It is difficult to debug code that is pulled in from a library. This is because the error messages don't give 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 new helper routines are created and used, they should be embedded in the main script for the purposes of testing and debugging. It is only after you have determined that your routines are correct that they should be moved to a library file to be included by other scripts.

Appendix D: General VBScript References and Tools

General References

- For general scripting information, go to:
<http://www.msdn.microsoft.com/library/default.asp?url=/library/en-us/dnanchor/html/scriptinga.asp>
- For a VBScript language reference, go to:
<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 that we care to mention (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, go to:
<http://www.activestate.com>

Text/Scripting Editors

There are a number of text editors with automatic syntax that can be of use in developing scripts; 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). A debug environment can also be used with this product (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) Run the **Tools > Macro > Visual Basic Editor** command.
- 2) In the **Visual Basic Editor** run the **Tools > References** command.
- 3) In the **Available References** list check the box associated with the **MGCPCB (AutoActive Series) Type Library** item.

-
- 4) Click the **OK** button in the **Available References** dialog.
 - 5) In the **Visual Basic Editor** run the **View > Object Browser** command.
 - 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.

Appendix E: Accessing Example Scripts and Designs

Introduction

The scripts and designs referenced in this manual can be found at:

<http://supportnet.mentor.com/reference/tutorials/10393.cfm>

Index

^ (Exponentiation)	Ch 2
* (Multiplication)	Ch 2
& (String concatenation)	Ch 2
/ (Division)	Ch 2
\ (Integer Division)	Ch 2
+ (Addition and string concatenation)	Ch 2
- (Subtraction and unary negation)	Ch 2
= (Equality)	Ch 2
<> (Inequality)	Ch 2
< (Less than)	Ch 2
> (Greater than)	Ch 2
<= (Less than or equal to)	Ch 2
>= (Greater than or equal to)	Ch 2

A

AbsorbHoles Method	Ch 25
Add-in, Output Window	Ch 15
AddMaskToUserLayer() Helper Subroutine	Ch 25
AddMenuAfter() Helper Function	Ch 13, 21
Accelerator Key Binding	Ch 1
Associating an Accelerator (Shortcut) Key with a Script	Ch 5
Accessing Data from CES	Ch 21
Adding Dimensions	Ch 19
And (Logical operator)	Ch 2
API (Application Programming Interface)	Ch 1
Application Programming Interface (API)	Ch 1
Applications (Attaching or Creating)	Ch 8
Arguments (Passing into scripts from the command line)	Ch 8
Arithmetic Operators (^, -, *, /, \, Mod, +, -)	Ch 2
Assembly DRC	Ch 15
Attaching to Applications	Ch 8
Automation (Data-, Function, and Flow-Centric)	Ch 1
Output Engines	Ch 26

B

BoardOutline Object	Ch 19, 22
BOM (Build-of-Materials), Generating	Ch 23
Building Block Examples	Ch 12
Build-of-Materials (BOM), Generating	Ch 23

C

C	Ch 2
C++	Ch 2
Call [Calling a method or function]	Ch 2
CDbl() Function	Ch 2
CES (Constraint Editor System)	Ch 21
CES Objects	Ch 21
Checking Shape-to-Shape Overlaps	Ch 15

CInt() Function	Ch 2
Class	
Objects	Ch 19
Point Structure	Ch 19
ClearMotionGraphics() Miscellaneous Function	Ch 14
Client/Server	Ch 1
In-Process Client	Ch 1
Out-of-Process Client	Ch 1
Collection	Ch 1, 16
Filters	Ch 12
COM (Component Object Model) automation	Ch 1
Command Bar Server	Ch 6, 7
Command	
Line, Running Scripts from	Ch 8
Object	Ch 12, 14
Command Bar Server	Ch 6, 7, 13
Commands, Interactive	Ch 12
Comma Separated Value (CSV) File	Ch 18
Comparison Operators (=, <>, <, >, <=, >=, Is)	Ch 2
Component Object Model (COM) automation	Ch 1
Component	
CSV File	Ch 18
Positional Data, Transforming	Ch 24
Property	Ch 12
Components, Traversing	Ch 12
ConfigureUserLayer() Helper Function	Ch 14, 24, 25
Connected Object Property	Ch 16
Connecting to CES	Ch 21
Constraint Editor System (CES)	Ch 21
Controlling the Auto-Router	Ch 20
Control Statements	Ch 2
Conversion and Formatting Functions	Ch 2
Conversion Functions [CInt(), CDbl(), CStr(), ...]	Ch 2
String Manipulation Functions [Len(), Left(), Right(), Replace(), ...]	Ch 2
CorrectSlashes() Helper Function	Ch 26
CreateUserLayer() Helper Function	Ch 19
Creating	
Applications	Ch 8
Cross-Platform Scripts	Ch 18
Form/Dialog (using the IDE)	Ch 4, 16, 27
New Top-Level Menu Item	Ch 6
Objects with Geometries	Ch 12
Script (using the IDE)	Ch 4
Via Mask	Ch 25
CreateTextFile Method	Ch 12, 18
Cross-Platform Scripts, Creating	Ch 18
Cscript.exe	Ch 8
CStr() Function	Ch 2
CSV (Comma Separated Value) File	Ch 18

D

Database (Parts DB)	Ch 23
Data-Centric Automation	Ch 1
Data Hierarchy	Ch 3
Date() Function	Ch 2

Delay, Maximum	Ch 21
Design	
Finalizing	Ch 19
for Fabrication (DFF)	Ch 22
Rule Checking (DRC)	
Assembly	Ch 15
Fabrication	Ch 22
Manufacturing	Ch 22
DFF (Design for Fabrication)	Ch 22
Dictionary-Based Tree Structures	Ch 23
Dictionary Object	Ch 16, 20, 21, 23
Dim [Dimension a variable]	Ch 2
Dim (x,y,x...) [Dimension an array of variables]	Ch 2
Dimensions, Adding	Ch 19
Display Control	Ch 12
DisplayControl Object	Ch 12, 13
Displaying a Progress Bar	Ch 21
Documents, Opening	Ch 8
Drawing Cells, Placing	Ch 19
DrawRectangle() Helper Function	Ch 14
DrawText() Helper Function	Ch 14
DRC	
Assembly	Ch 15
Fabrication	Ch 22
Manufacturing	Ch 22
Dynamic (Run-Time) Code Generation	Ch 13

E

Enumerate	Ch 1
EqualizeClearance() Helper Function	Ch 27
Eqv (Logical operator)	Ch 2
Error Handling	Ch 2
Err.Clear	Ch 2
Err.Description	Ch 2
Err.Number	Ch 2
IDE Error Handling Capability	Ch 4
On Error Goto 0	Ch 2
On Error Resume Next	Ch 2
Event(s)	Ch 1, 11
OnPreComponentPlace Event	Ch 15
OnSelectionChange Event	Ch 17
Excel	
Integration with	Ch 17
Server	Ch 17
ExcelsRunning() Helper Function	Ch 17
Exec Object	Ch 12
Executing Menu Commands (from within scripts)	Ch 12
Expedition PCB	Ch 1
Server	Ch 23, 24, 25, 27
Extrema Object	Ch 19

F

FabLink XE	
Automated Board Placement on a Panel	Ch 27

Creating a Via Mask	Ch 25
Introducing	Ch 22
Generating	
BOM	Ch 23
Panel Side View	Ch 24
Manufacturing Output Files	Ch 26
Fabrication DRC	Ch 22
FileExists Method	Ch 12
File Input/Output (I/O)	Ch 12
FileSystemObject Object	Ch 12, 18
File System Object Server	Ch 18
Filters, Collection	Ch 12
Finalizing a Design	Ch 19
FindAddin() Helper Function	Ch 15
FindInWDIR() Helper Function	Ch 27
FindMaxFit() Helper Function	Ch 27
Flow-Centric Automation	Ch 1
For Each ... In ... Next	Ch 2
For ... To ... Next	Ch 2
FormatDateTime() Function	Ch 2
FormatNumber() Function	Ch 2
Function-Centric Automation	Ch 1
Functions	Ch 2

G

Geometric Calculations, Performing	Ch 20
Gerber	
Interface	Ch 26
Object	Ch 26
Output Engine	Ch 26
GetAngleOfLine() Helper Function	Ch 20
GetBoardDimensions() Helper Subroutine	Ch 27
GetBoardThickness() Helper Function	Ch 24
GetCESApplication() Helper Function	Ch 21
GetConnectedLayers() Helper Function	Ch 16
GetLayerDescription() Helper Function	Ch 14
GetMenuNumber() Helper Function	Ch 13, 21
GetSignalLayer() Helper Function	Ch 20
GetYExtrema() Helper Subroutine	Ch 24
GlobalDisplayControl Object	Ch 12
Glue Languages (JScript, Perl, Python, Tcl, VBScript)	Ch 2
Gui Property	Ch 5, 6, 7, 12

H

Hello	
World	Ch 3
Sailor	Ch 8
Helper Functions/Subroutines	
AddMaskToUserLayer() Subroutine	Ch 25
AddMenuAfter() Function	Ch 13, 21
ConfigureUserLayer() Function	Ch 14, 24, 25
CorrectSlashes() Function	Ch 26
CreateUserLayer() Function	Ch 19
DrawRectangle() Function	Ch 14

DrawText() Function	Ch 14
EqualizeClearance() Function	Ch 27
ExcellsRunning() Function	Ch 17
FindAddin() Function	Ch 15
FindInWDIR() Function	Ch 27
FindMaxFit() Function	Ch 27
GetAngleOfLine() Function	Ch 20
GetBoardDimensions() Subroutine	Ch 27
GetBoardThickness() Function	Ch 24
GetCESApplication() Function	Ch 21
GetConnectedLayers() Function	Ch 16
GetLayerDescription() Function	Ch 14
GetMenuNumber() Function	Ch 13, 21
GetSignalLayer() Function	Ch 20
GetYExtrema() Subroutine	Ch 24
InitTree() Function	Ch 23
LoadAddin() Function	Ch 15
Output() Function	Ch 15
RadiansToDegrees() Function	Ch 20
RemoveExcelFill() Subroutine	Ch 17
RemoveTrailingNewLine() Function	Ch 15
RemoveUnconnectedPads() Subroutine	Ch 16
SelectExcelRange() Subroutine	Ch 17
StripFile() Function	Ch 26
ViewCSV() Function	Ch 18
Help System, Using	Ch 3
Hierarchy, Data	Ch 3

IDE (Integrated Development Environment)	Ch 4
Creating a Form/Dialog	Ch 4, 16, 27
Creating a Script	Ch 4
Error Handling Capability	Ch 4
If ... Then ... Else ... End If	Ch 2
If ... Then ... End If	Ch 2
Imp (Logical operator)	Ch 2
InitTree() Helper Function	Ch 23
In-Process	
Client	Ch 1
Scripts	Ch 11
InputBox() Function	Ch 2
Input/Output (I/O)	
File I/O	Ch 12
Functions	Ch 2
InputBox()	Ch 2
MsgBox()	Ch 2
Integrated Development Environment (IDE)	Ch 4
Creating a Form/Dialog	Ch 4, 16, 27
Creating a Script	Ch 4
Error Handling Capability	Ch 4
Integration with Excel	Ch 17
Interactive Commands	Ch 12
Is (object comparison operator)	Ch 2

J

Java	Ch 2
JScript	Ch 2

K

Key Bind Server	Ch 5, 13
-----------------------	----------

L

LayerObject Object	Ch 13, 20
Left() Function	Ch 2
Len() Function	Ch 2
Length, Maximum	Ch 21
LoadAddin() Helper Function	Ch 15
Logical Operators (Not, And, Or, Xor, Eqv, Imp)	Ch 2

M

Manufacturing

DRC	Ch 22
Output Files, Generating	Ch 26
Mask Engine Server	Ch 15, 25
Maximum Length/Delay	Ch 21
Menu	
Adding a Menu Entry	Ch 21
Commands (Executing from within scripts)	Ch 12
Button/Item	Ch 1
Associating with Script	Ch 6
Creating New Top-Level Menu Item	Ch 6
Popup	Ch 1
Method(s)	
AbsorbHoles Method	Ch 25
CreateTextFile Method	Ch 12, 18
FileExists Method	Ch 12
MotionGfx Method	Ch 14
PickComponents Method	Ch 15
ProgressBarInitialize Method	Ch 21
ProgressBar Method	Ch 21
ProcessCommand Method	Ch 12
ProcessScript Method	Ch 27
PutBoard Method	Ch 27
PutComponent Method	Ch 19
PutPointToPointDimension Method	Ch 19
PutTrace Method	Ch 12
PutUserLayerGfx Method	Ch 12
Scripting.IsUnix Method	Ch 18
TransformPointsArray Method	Ch 24, 25
VBScript Timer Method	Ch 21
MGCscript.exe	Ch 8, 11
Mod (Arithmetic operator)	Ch 2
Modifying PadStacks	Ch 16
MotionGfx Method	Ch 14
Motion Graphics	Ch 14

MotionGfx Method	Ch 14
MsgBox() Function	Ch 2

N

Name-Value Property Pairs	Ch 15
NC Drill	
Interface	Ch 26
Object	Ch 26
Output Engine	Ch 26
Nets, Traversing	Ch 12
Not (Logical operator)	Ch 2
NotePad Editor	Ch 12
Now() Function	Ch 2

O

Object(s)	Ch 1
BoardOutline Object	Ch 19, 22
CES Objects	Ch 21
Class Objects	Ch 19
Collection	Ch 16
Command Object	Ch 12, 14
Creating Objects with Geometries	Ch 12
Dictionary Object	Ch 16, 20, 21, 23
DisplayControl Object	Ch 12, 13
Exec Object	Ch 12
Extreme Object	Ch 19
FileSystemObject Object	Ch 12, 18
Gerber Object	Ch 26
GlobalDisplayControl Object	Ch 12
LayerObject Object	Ch 13, 20
NCDrill Object	Ch 26
PadStackObject Object	Ch 12, 16
PanelOutline Object	Ch 22
RoutePass Object	Ch 20
Scripting Object	Ch 11
TextStream Object	Ch 12
Traversing Objects	Ch 12
Utility Object	Ch 12
VBScript Err Object	Ch 17
Viewlogic.Exec Object	Ch 18
On Error Resume Next	Ch 2
OnPreComponentPlace Event	Ch 15
OnSelectionChange Event	Ch 17
Opening Documents	Ch 8
Operators (Arithmetic, Comparison, Logical, String)	Ch 2
Option Explicit	Ch 2
Or (Logical operator)	Ch 2
Out-of-Process	
Client	Ch 1
Scripts	Ch 11
Output	
Engines, Automation	Ch 26
Files, Generating Manufacturing	Ch 26
Window Add-in	Ch 15

Output() Helper Function	Ch 15
--------------------------------	-------

P

PadStackObject	
Object	Ch 12, 16
Property	Ch 12
PadStacks, Modifying	Ch 16
Pairs (Name-Value Property Pairs)	Ch 15
Panel	Ch 22
Side View, Generating	Ch 24
PanelOutline Object	Ch 22
Parts	
Database (Parts DB)	Ch 23
Editor Server	Ch 23
Performing Geometric Calculations	Ch 20
Perl	Ch 2
PickComponents Method	Ch 15
Pins, Traversing	Ch 12
Placing Drawing Cells	Ch 19
Points Array(s)	Ch 12, 19
ProcessCommand Method	Ch 12
Prog ID	Ch 1
Progress Bar, Displaying	Ch 21
ProgressBarInitialize Method	Ch 21
ProgressBar Method	Ch 21
Property/Properties	Ch 1
Component Property	Ch 12
Connected Object Property	Ch 16
Gui Property	Ch 12
PadStackObject Property	Ch 12
Pairs (Name-Value Property Pairs)	Ch 15
Scripting.Globals Property	Ch 27
Traces Property	Ch 12
ProcessScript Method	Ch 27
PutBoard Method	Ch 27
PutComponent Method	Ch 19
PutPointToPointDimension Method	Ch 19
PutTrace Method	Ch 12
PutUserLayerGfx Method	Ch 12
Python	Ch 2

R

RadiansToDegrees() Helper Function	Ch 20
ReDim (x,y,z,...) [Re-dimension an array of variables]	Ch 2
RemoveExcelFill() Helper Subroutine	Ch 17
RemoveTrailingNewLine() Helper Function	Ch 15
RemoveUnconnectedPads() Helper Subroutine	Ch 16
Replace() Function	Ch 2
Right() Function	Ch 2
Route by Layer	Ch 20
RoutePass Object	Ch 20
Running	
Auto-Router	Ch 20
Executables	Ch 12

Scripts

From the Command Line (with/without Arguments)	Ch 8
From within an Application	Ch 9
Startup Scripts	Ch 10
Run-Time (Dynamic) Code Generation	Ch 13

S

Script	Ch 1
Host	Ch 1
Scripting	
Language	Ch 1, 2
Object	Ch 11
Scripting.Globals Property	Ch 27
Scripting.IsUnix Method	Ch 18
scripts.ini Files	Ch 10
Select Case ... End Select	Ch 2
SelectExcelRange() Helper Subroutine	Ch 17
Server(s)	Ch 1
Command Bar Server	Ch 6, 7, 13
Excel Server	Ch 17
Expedition PCB Server	Ch 23, 24, 25, 27
File System Object Server	Ch 18
Key Bind Server	Ch 5, 13
Mask Engine Server	Ch 15, 25
Parts Editor Server	Ch 23
Viewlogic COM Server	Ch 12
Set [Setting/assigning an object to a variable]	Ch 2
Shape-to-Shape Overlaps, Checking	Ch 15
Side View of Panel, Generating	Ch 24
Special Characters	Ch 2
Startup Scripts	Ch 10
Status Bar	Ch 12
String	
Manipulation Functions [Len(), Left(), Right(), Replace(), ...]	Ch 2
Operators (&, +)	Ch 2
StripFile() Helper Function	Ch 26
Structure, Class Point	Ch 19
Subroutines	Ch 2
System	
Integration Languages (JScript, Perl, Python, Tcl, VBScript)	Ch 2
Programming Languages (C, C++, Java)	Ch 2

T

Tcl	Ch 2
TextStream Object	Ch 12
Time() Function	Ch 2
Time-of-Flight (TOF)	Ch 21
Time-Out, Implementing	Ch 21
TOF (Time-of-Flight)	Ch 21
Tool Bar Icon, Adding	Ch 7
Traces Property	Ch 12
Transforming	
Component Positional Data	Ch 24
Via Pad Data	Ch 25

TransformPointsArray Method	Ch 24, 25
Traversing	
Components	Ch 12
Nets	Ch 12
Objects	Ch 12
Pins	Ch 12
Tree Structures, Dictionary-Based	Ch 23
Type Library	Ch 1

U

UBound(...) [Return the upper-bound(s) of an array]	Ch 2
Using	
Help System	Ch 3
Layer Objects	Ch 20
Utility Object	Ch 12

V

VBScript	Ch 2
VBScript Err Object	Ch 17
VBScript Timer Method	Ch 21
Via	
Mask, Creating	Ch 25
Pad Data, Transforming	Ch 25
ViewCSV() Helper Function	Ch 18
Viewlogic	
COM Server	Ch 12
Exec Object	Ch 18

W

WDIR Environment Variable	Ch 9
While ... Wend	Ch 2
Wscript.exe	Ch 8

X

Xor (Logical operator)	Ch 2
------------------------------	------

End-User License Agreement

The latest version of the End-User License Agreement is available on-line at:

www.mentor.com/terms_conditions/enduser.cfm

IMPORTANT INFORMATION

USE OF THIS SOFTWARE IS SUBJECT TO LICENSE RESTRICTIONS. CAREFULLY READ THIS LICENSE AGREEMENT BEFORE USING THE SOFTWARE. USE OF SOFTWARE INDICATES YOUR COMPLETE AND UNCONDITIONAL ACCEPTANCE OF THE TERMS AND CONDITIONS SET FORTH IN THIS AGREEMENT. ANY ADDITIONAL OR DIFFERENT PURCHASE ORDER TERMS AND CONDITIONS SHALL NOT APPLY.

END-USER LICENSE AGREEMENT ("Agreement")

This is a legal agreement concerning the use of Software between you, the end user, as an authorized representative of the company acquiring the license, and Mentor Graphics Corporation and Mentor Graphics (Ireland) Limited acting directly or through their subsidiaries (collectively "Mentor Graphics"). Except for license agreements related to the subject matter of this license agreement which are physically signed by you and an authorized representative of Mentor Graphics, this Agreement and the applicable quotation contain the parties' entire understanding relating to the subject matter and supersede all prior or contemporaneous agreements. If you do not agree to these terms and conditions, promptly return or, if received electronically, certify destruction of Software and all accompanying items within five days after receipt of Software and receive a full refund of any license fee paid.

1. **GRANT OF LICENSE.** The software programs, including any updates, modifications, revisions, copies, documentation and design data ("Software"), are copyrighted, trade secret and confidential information of Mentor Graphics or its licensors who maintain exclusive title to all Software and retain all rights not expressly granted by this Agreement. Mentor Graphics grants to you, subject to payment of appropriate license fees, a nontransferable, nonexclusive license to use Software solely: (a) in machine-readable, object-code form; (b) for your internal business purposes; (c) for the license term; and (d) on the computer hardware and at the site authorized by Mentor Graphics. A site is restricted to a one-half mile (800 meter) radius. Mentor Graphics' standard policies and programs, which vary depending on Software, license fees paid or services purchased, apply to the following: (a) relocation of Software; (b) use of Software, which may be limited, for example, to execution of a single session by a single user on the authorized hardware or for a restricted period of time (such limitations may be technically implemented through the use of authorization codes or similar devices); and (c) support services provided, including eligibility to receive telephone support, updates, modifications, and revisions.
2. **EMBEDDED SOFTWARE.** If you purchased a license to use embedded software development ("ESD") Software, if applicable, Mentor Graphics grants to you a nontransferable, nonexclusive license to reproduce and distribute executable files created using ESD compilers, including the ESD run-time libraries distributed with ESD C and C++ compiler Software that are linked into a composite program as an integral part of your compiled computer program, provided that you distribute these files only in conjunction with your compiled computer program. Mentor Graphics does NOT grant you any right to duplicate, incorporate or embed copies of Mentor Graphics' real-time operating systems or other embedded software products into your products or applications without first signing or otherwise agreeing to a separate agreement with Mentor Graphics for such purpose.
3. **BETA CODE.** Software may contain code for experimental testing and evaluation ("Beta Code"), which may not be used without Mentor Graphics' explicit authorization. Upon Mentor Graphics' authorization, Mentor Graphics grants to you a temporary, nontransferable, nonexclusive license for experimental use to test and evaluate the Beta Code without charge for a limited period of time specified by Mentor Graphics. This grant and your use of the Beta Code shall not be construed as marketing or offering to sell a license to the Beta Code, which Mentor Graphics may choose not to release commercially in any form. If Mentor Graphics authorizes you to use the Beta Code, you agree to evaluate and test the Beta Code under normal conditions as directed by Mentor Graphics. You will contact Mentor Graphics periodically during your use of the Beta Code to discuss any malfunctions or suggested improvements. Upon completion of your evaluation and testing, you will send to

Mentor Graphics a written evaluation of the Beta Code, including its strengths, weaknesses and recommended improvements. You agree that any written evaluations and all inventions, product improvements, modifications or developments that Mentor Graphics conceived or made during or subsequent to this Agreement, including those based partly or wholly on your feedback, will be the exclusive property of Mentor Graphics. Mentor Graphics will have exclusive rights, title and interest in all such property. The provisions of this section 3 shall survive the termination or expiration of this Agreement.

4. **RESTRICTIONS ON USE.** You may copy Software only as reasonably necessary to support the authorized use. Each copy must include all notices and legends embedded in Software and affixed to its medium and container as received from Mentor Graphics. All copies shall remain the property of Mentor Graphics or its licensors. You shall maintain a record of the number and primary location of all copies of Software, including copies merged with other software, and shall make those records available to Mentor Graphics upon request. You shall not make Software available in any form to any person other than employees and on-site contractors, excluding Mentor Graphics' competitors, whose job performance requires access and who are under obligations of confidentiality. You shall take appropriate action to protect the confidentiality of Software and ensure that any person permitted access to Software does not disclose it or use it except as permitted by this Agreement. Except as otherwise permitted for purposes of interoperability as specified by applicable and mandatory local law, you shall not reverse-assemble, reverse-compile, reverse-engineer or in any way derive from Software any source code. You may not sublicense, assign or otherwise transfer Software, this Agreement or the rights under it, whether by operation of law or otherwise ("attempted transfer"), without Mentor Graphics' prior written consent and payment of Mentor Graphics' then-current applicable transfer charges. Any attempted transfer without Mentor Graphics' prior written consent shall be a material breach of this Agreement and may, at Mentor Graphics' option, result in the immediate termination of the Agreement and licenses granted under this Agreement. The terms of this Agreement, including without limitation, the licensing and assignment provisions shall be binding upon your successors in interest and assigns. The provisions of this section 4 shall survive the termination or expiration of this Agreement.

5. LIMITED WARRANTY.

- 5.1. Mentor Graphics warrants that during the warranty period Software, when properly installed, will substantially conform to the functional specifications set forth in the applicable user manual. Mentor Graphics does not warrant that Software will meet your requirements or that operation of Software will be uninterrupted or error free. The warranty period is 90 days starting on the 15th day after delivery or upon installation, whichever first occurs. You must notify Mentor Graphics in writing of any nonconformity within the warranty period. This warranty shall not be valid if Software has been subject to misuse, unauthorized modification or improper installation. MENTOR GRAPHICS' ENTIRE LIABILITY AND YOUR EXCLUSIVE REMEDY SHALL BE, AT MENTOR GRAPHICS' OPTION, EITHER (A) REFUND OF THE PRICE PAID UPON RETURN OF SOFTWARE TO MENTOR GRAPHICS OR (B) MODIFICATION OR REPLACEMENT OF SOFTWARE THAT DOES NOT MEET THIS LIMITED WARRANTY, PROVIDED YOU HAVE OTHERWISE COMPLIED WITH THIS AGREEMENT. MENTOR GRAPHICS MAKES NO WARRANTIES WITH RESPECT TO: (A) SERVICES; (B) SOFTWARE WHICH IS LICENSED TO YOU FOR A LIMITED TERM OR LICENSED AT NO COST; OR (C) EXPERIMENTAL BETA CODE; ALL OF WHICH ARE PROVIDED "AS IS."
- 5.2. THE WARRANTIES SET FORTH IN THIS SECTION 5 ARE EXCLUSIVE. NEITHER MENTOR GRAPHICS NOR ITS LICENSORS MAKE ANY OTHER WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, WITH RESPECT TO SOFTWARE OR OTHER MATERIAL PROVIDED UNDER THIS AGREEMENT. MENTOR GRAPHICS AND ITS LICENSORS SPECIFICALLY DISCLAIM ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT OF INTELLECTUAL PROPERTY.
6. **LIMITATION OF LIABILITY.** EXCEPT WHERE THIS EXCLUSION OR RESTRICTION OF LIABILITY WOULD BE VOID OR INEFFECTIVE UNDER APPLICABLE LAW, IN NO EVENT SHALL MENTOR GRAPHICS OR ITS LICENSORS BE LIABLE FOR INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES (INCLUDING LOST PROFITS OR SAVINGS) WHETHER BASED ON CONTRACT, TORT OR ANY OTHER LEGAL THEORY, EVEN IF MENTOR GRAPHICS OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. IN NO EVENT SHALL MENTOR GRAPHICS' OR ITS LICENSORS' LIABILITY UNDER THIS AGREEMENT EXCEED THE AMOUNT PAID BY YOU FOR THE SOFTWARE OR SERVICE GIVING RISE TO THE CLAIM. IN THE CASE WHERE NO AMOUNT WAS PAID, MENTOR GRAPHICS AND ITS LICENSORS SHALL

HAVE NO LIABILITY FOR ANY DAMAGES WHATSOEVER. THE PROVISIONS OF THIS SECTION 6 SHALL SURVIVE THE EXPIRATION OR TERMINATION OF THIS AGREEMENT.

7. **LIFE ENDANGERING ACTIVITIES.** NEITHER MENTOR GRAPHICS NOR ITS LICENSORS SHALL BE LIABLE FOR ANY DAMAGES RESULTING FROM OR IN CONNECTION WITH THE USE OF SOFTWARE IN ANY APPLICATION WHERE THE FAILURE OR INACCURACY OF THE SOFTWARE MIGHT RESULT IN DEATH OR PERSONAL INJURY. THE PROVISIONS OF THIS SECTION 7 SHALL SURVIVE THE EXPIRATION OR TERMINATION OF THIS AGREEMENT.
8. **INDEMNIFICATION.** YOU AGREE TO INDEMNIFY AND HOLD HARMLESS MENTOR GRAPHICS AND ITS LICENSORS FROM ANY CLAIMS, LOSS, COST, DAMAGE, EXPENSE, OR LIABILITY, INCLUDING ATTORNEYS' FEES, ARISING OUT OF OR IN CONNECTION WITH YOUR USE OF SOFTWARE AS DESCRIBED IN SECTION 7. THE PROVISIONS OF THIS SECTION 8 SHALL SURVIVE THE EXPIRATION OR TERMINATION OF THIS AGREEMENT.
9. **INFRINGEMENT.**
 - 9.1. Mentor Graphics will defend or settle, at its option and expense, any action brought against you alleging that Software infringes a patent or copyright or misappropriates a trade secret in the United States, Canada, Japan, or member state of the European Patent Office. Mentor Graphics will pay any costs and damages finally awarded against you that are attributable to the infringement action. You understand and agree that as conditions to Mentor Graphics' obligations under this section you must: (a) notify Mentor Graphics promptly in writing of the action; (b) provide Mentor Graphics all reasonable information and assistance to defend or settle the action; and (c) grant Mentor Graphics sole authority and control of the defense or settlement of the action.
 - 9.2. If an infringement claim is made, Mentor Graphics may, at its option and expense: (a) replace or modify Software so that it becomes noninfringing; (b) procure for you the right to continue using Software; or (c) require the return of Software and refund to you any license fee paid, less a reasonable allowance for use.
 - 9.3. Mentor Graphics has no liability to you if infringement is based upon: (a) the combination of Software with any product not furnished by Mentor Graphics; (b) the modification of Software other than by Mentor Graphics; (c) the use of other than a current unaltered release of Software; (d) the use of Software as part of an infringing process; (e) a product that you make, use or sell; (f) any Beta Code contained in Software; (g) any Software provided by Mentor Graphics' licensors who do not provide such indemnification to Mentor Graphics' customers; or (h) infringement by you that is deemed willful. In the case of (h) you shall reimburse Mentor Graphics for its attorney fees and other costs related to the action upon a final judgment.
 - 9.4. THIS SECTION IS SUBJECT TO SECTION 6 ABOVE AND STATES THE ENTIRE LIABILITY OF MENTOR GRAPHICS AND ITS LICENSORS AND YOUR SOLE AND EXCLUSIVE REMEDY WITH RESPECT TO ANY ALLEGED PATENT OR COPYRIGHT INFRINGEMENT OR TRADE SECRET MISAPPROPRIATION BY ANY SOFTWARE LICENSED UNDER THIS AGREEMENT.
10. **TERM.** This Agreement remains effective until expiration or termination. This Agreement will immediately terminate upon notice if you exceed the scope of license granted or otherwise fail to comply with the provisions of Sections 1, 2, or 4. For any other material breach under this Agreement, Mentor Graphics may terminate this Agreement upon 30 days written notice if you are in material breach and fail to cure such breach within the 30 day notice period. If Software was provided for limited term use, this Agreement will automatically expire at the end of the authorized term. Upon any termination or expiration, you agree to cease all use of Software and return it to Mentor Graphics or certify deletion and destruction of Software, including all copies, to Mentor Graphics' reasonable satisfaction.
11. **EXPORT.** Software is subject to regulation by local laws and United States government agencies, which prohibit export or diversion of certain products, information about the products, and direct products of the products to certain countries and certain persons. You agree that you will not export any Software or direct product of Software in any manner without first obtaining all necessary approval from appropriate local and United States government agencies.

12. **RESTRICTED RIGHTS NOTICE.** Software was developed entirely at private expense and is commercial computer software provided with RESTRICTED RIGHTS. Use, duplication or disclosure by the U.S. Government or a U.S. Government subcontractor is subject to the restrictions set forth in the license agreement under which Software was obtained pursuant to DFARS 227.7202-3(a) or as set forth in subparagraphs (c)(1) and (2) of the Commercial Computer Software - Restricted Rights clause at FAR 52.227-19, as applicable. Contractor/manufacturer is Mentor Graphics Corporation, 8005 SW Boeckman Road, Wilsonville, Oregon 97070-7777 USA.
13. **THIRD PARTY BENEFICIARY.** For any Software under this Agreement licensed by Mentor Graphics from Microsoft or other licensors, Microsoft or the applicable licensor is a third party beneficiary of this Agreement with the right to enforce the obligations set forth herein.
14. **AUDIT RIGHTS.** You will monitor access to, location and use of Software. With reasonable prior notice and during your normal business hours, Mentor Graphics shall have the right to review your software monitoring system and reasonably relevant records to confirm your compliance with the terms of this Agreement, an addendum to this Agreement or U.S. or other local export laws. Such review may include FLEXIm or FLEXnet report log files that you shall capture and provide at Mentor Graphics' request. Mentor Graphics shall treat as confidential information all of your information gained as a result of any request or review and shall only use or disclose such information as required by law or to enforce its rights under this Agreement or addendum to this Agreement. The provisions of this section 14 shall survive the expiration or termination of this Agreement.
15. **CONTROLLING LAW, JURISDICTION AND DISPUTE RESOLUTION.** THIS AGREEMENT SHALL BE GOVERNED BY AND CONSTRUED UNDER THE LAWS OF THE STATE OF OREGON, USA, IF YOU ARE LOCATED IN NORTH OR SOUTH AMERICA, AND THE LAWS OF IRELAND IF YOU ARE LOCATED OUTSIDE OF NORTH OR SOUTH AMERICA. All disputes arising out of or in relation to this Agreement shall be submitted to the exclusive jurisdiction of Portland, Oregon when the laws of Oregon apply, or Dublin, Ireland when the laws of Ireland apply. Notwithstanding the foregoing, all disputes in Asia (except for Japan) arising out of or in relation to this Agreement shall be resolved by arbitration in Singapore before a single arbitrator to be appointed by the Chairman of the Singapore International Arbitration Centre ("SIAC") to be conducted in the English language, in accordance with the Arbitration Rules of the SIAC in effect at the time of the dispute, which rules are deemed to be incorporated by reference in this section 15. This section shall not restrict Mentor Graphics' right to bring an action against you in the jurisdiction where your place of business is located. The United Nations Convention on Contracts for the International Sale of Goods does not apply to this Agreement.
16. **SEVERABILITY.** If any provision of this Agreement is held by a court of competent jurisdiction to be void, invalid, unenforceable or illegal, such provision shall be severed from this Agreement and the remaining provisions will remain in full force and effect.
17. **PAYMENT TERMS AND MISCELLANEOUS.** You will pay amounts invoiced, in the currency specified on the applicable invoice, within 30 days from the date of such invoice. Any past due invoices will be subject to the imposition of interest charges in the amount of one and one-half percent per month or the applicable legal rate currently in effect, whichever is lower. Some Software may contain code distributed under a third party license agreement that may provide additional rights to you. Please see the applicable Software documentation for details. This Agreement may only be modified in writing by authorized representatives of the parties. Waiver of terms or excuse of breach must be in writing and shall not constitute subsequent consent, waiver or excuse.

Rev. 060210, Part No. 227900