Chapter 12

Using Visual Basic for Applications (VBA)

Introduction

Visual Basic for Applications (VBA) is a Microsoft Visual Basic® programming system application. VBA is an industry standard and a powerful programming environment. It is the fastest and easiest way to create and customize Microsoft Windows applications. M-Graphics is shipped with Microsoft VBA Release 5.0. VBA allows users to customize M-Graphics to suit their specific requirements. It also offers high level application programmability and features cross platform support for ActiveX technology Windows 98 and Windows NT operating systems. It is identical with VBA in Microsoft Office applications and other third-party products.

This chapter describes how to:

- create and edit VBA scripts
- add event handlers
- display the *ThisDisplay* module code
- create a macro
- edit a macro
- step a macro
- run a macro
- clean VBA unused modules

Key Concepts

VBA Features

VBA allows you to:

- create, debug, and run custom scripts or macros
- write Visual Basic code for M-Graphics events
- modify M-Graphics native objects
- connect ActiveX objects to each other and to M-Graphics native objects

VBA allows both Configure and Runtime operations.

Windows

A window is a rectangular region with its own boundaries. Examples of windows include: a Windows Explorer window, a document window within Microsoft Word, on M-Graphics displays, a dialog box, text box, message box, command button, etc. A container is a window that contains one or more other windows, buttons, controls, etc. The Microsoft Windows operating system manages all the windows by assigning a unique ID to each window.

Events

Events are actions associated with a window. Events can occur through user actions such as mouse clicks, key presses, or as a result of another window's actions. When an event occurs, a message is sent to the operating system that broadcasts the message to other windows. Each window can take appropriate action based on its own instructions. Examples of event handling are: repainting of a window by itself when uncovered by another window, closing, minimizing or maximizing a window by clicking on the appropriate control, etc.

Many of the standard events or messages are handled automatically by VBA. Other events are exposed to the user as Event procedures, and the user can write powerful code to deal with Event procedures without having to deal with unnecessary details.

Object Oriented Programming

Visual Basic is an object oriented programming language. Contrary to procedural languages like C or BASIC, VB uses objects to create applications. Examples of objects are: forms, controls, M-Graphics displays, databases, etc. Create your own objects from a set of rules called classes. Objects and classes simplify your coding and increase code reuse.

Classes

Classes are sets of rules that define objects. Objects in Visual Basic are created from classes. An object is an instance of a class. The class defines an object's interfaces, default methods, and properties. Descriptions of classes are stored in type libraries and can be viewed with object browsers.

Objects

Objects are encapsulated, which means they contain both their code and their data. This makes them easier to maintain than traditional code. Visual Basic objects have properties, methods, and events.

Methods and Properties

Properties are data that describe an object. Methods are object actions. Events are actions the object performs.

Development Using Visual Basic for Applications

VBA uses an event driven model approach for development. Visual Basic interprets the code, as you write it. Write, compile, and test code during the development.

VBA Editor

VBA Editor is an Integrated Development Environment (IDE) and is integrated into M-Graphics. It can be opened from the M-Graphics menu bar.

Create, edit, debug, and run Visual Basic code using the Visual Basic Editor. The custom code is stored in modules, class modules, and forms.

The VBA Editor supports project management. Create projects using the Editor. Projects can contain M-Graphics objects, VB modules, forms, windows, etc. Projects can be opened or closed from the View menu.

VBA Editor Menu Bar

The menu bar displays the commands used with VBA. It consists of several menus: File, Edit, View, Window, Help, Project, Format, and Debug.

VBA Editor Context Menus

The context menus can be invoked by right-clicking an object. The context menus contain shortcuts to frequently performed actions.

VBA Editor Toolbars

The toolbars provide quick access to commands in the programming environment. By default, the Standard toolbar is displayed. Additional toolbars for editing, form design, and debugging toggles on or off the Toolbars command on the View menu.

VBA Editor Toolbox

The toolbox provides a set of tools used at design time to place controls on a form. In addition to the default toolbox layout, create custom layouts by selecting Add Tab from the context menu and adding controls to the resulting tab.

VBA Editor Project Explorer Window

The Project Explorer Window lists the forms and modules in the current project. As you create, add, or remove files from a project, Visual Basic reflects the changes in the Project Explorer window, which contains a current list of the files in the project.

Project

A project is the collection of files used to build an application. In the VBA project, there is a collection of modules; all modules are stored, along with M-Graphics objects, in the same file (.gdf).

However, it is possible to export the modules to .bas files, the class modules to .cls files, and forms to .frm files. A project consists of:

- one project file that keeps track of all the components (.vbp).
- one file for each form (.frm).
- one binary data file for each form containing data for properties of controls on the form (.frx). These files are not editable and are automatically generated for any .frm file that contains binary properties such as Picture or Icon.
- optionally, one file for each class module (.cls).
- optionally, one file for each standard module (.bas).
- optionally, one or more files containing ActiveX controls (.ocx).
- optionally, a single resource file (.res).

Project File

The project file is a list of all the files and objects associated with the project as well as information on the environment options. This information is updated every time you save the project. All of the files and objects can be shared by other projects.

Properties Window

The Properties window lists the property settings for the selected form or control. A property is a characteristic of an object, such as size, caption, or color.

Object Browser

The Object Browser lists objects available for use in a project and provides quick navigation through your code. Use the Object Browser to explore objects in Visual Basic and other applications, view methods and properties are available for those objects, and paste code procedures into the application.

Form Designer

The form designer serves as a window that you customize to design the interface of your application. Add controls, graphics, and pictures to a form to create the look you want. Each form in an application has its own form designer window.

Code Editor Window

The code editor window serves as an editor for entering application code. A separate code editor window is created for each form or code module in an application.

Form Layout Window

The form layout window positions the forms in an application using a small, graphical representation of the screen.

Immediate, Locals, and Watch Windows

These additional windows are provided for use in debugging an application. They are only available when running an application within the editor.

Forms and Controls

Forms are user interfaces, which are the visual part with which the user interacts. Forms and controls are the basic building blocks used to create the interface.

Forms are objects that expose properties defining their appearance, methods defining their behavior, and events defining their interaction with the user. By setting the properties of the form and writing Visual Basic code to respond to its events, you customize the object to meet your requirements.

Controls are objects contained within form objects. Each type of control has its own set of properties, methods, and events that make it suitable for a particular purpose. Examples of controls are fields for entering or displaying text. Controls can also be used for accessing other applications and process data as if the remote application was part of your code.

ActiveX

ActiveX is a set of integration technologies that enables software components to interoperate in a networked environment using any language. ActiveX is based on Microsoft Object Linking and Embedding (OLE) and the Component Object Model (COM).

ActiveX Control

ActiveX is a type of control and is an extension to the Visual Basic Toolbox. When adding an ActiveX control to a program, it becomes part of the development and Runtime environment and provides new functionality for your application.

ActiveX Used with M-Graphics

ActiveX control is used to embed documents from other applications into M-Graphics displays. Applications supporting ActiveX include Iconics TWXView32 or other Windows applications.

Conversely you could open and run M-Graphics displays from other applications such as Microsoft Internet Explorer.

Modules

Code in VBA is stored in modules. There are three kinds of modules: form, standard, and class. By default, the VBA modules are stored in .gdf files. They can be exported to files and imported back.

Standard Module

Usually the code associated with a form resides in that form module. If you have forms or other modules that could use a common code, create a separate module containing a procedure that implements the common code. This separate module should be a standard module.

Each standard module can contain declarations such as, type, variable and procedures such as Function (functions) or Sub (subroutines). The standard module file has an extension .bas.

Form Modules

Form modules are the foundation of most Visual Basic applications. They can contain procedures that handle events, general procedures, and form level declarations of variables, constants, types, and external procedures. The code in a form module is specific to the particular application to which the form belongs; it might also reference other forms or objects within that application. The form module files have an extension .frm.

Class Modules

Class modules (.cls file name extension) are the foundation of object-oriented programming in Visual Basic. Write code in class modules to create new objects. These new objects can include customized properties and methods. Forms are just class modules that can have controls placed on them and can display form windows.

ActiveX Modules

Various ActiveX modules include ActiveX Documents, ActiveX Designers, and User Controls. From the standpoint of writing code, these modules should be considered the same as form modules.

M-Graphics VBA Project

The M-Graphics VBA project is loaded whenever you launch the VBA Editor from M-Graphics. This project contains groups of modules by default such as:

- M-Graphics Native Objects
- Modules - ThisDisplay module and GwxTools module
- Forms

Each module can contain VBA code – functions, subroutines, event handlers, and global declarations.

GwxTools *Module*

GwxTools is a custom module with common subroutines used in the VBA Wizard described in the next chapter. The GwxTools module is not available to the user until a VBA Wizard is used.

ThisDisplay *Module*

ThisDisplay represents the current M-Graphics display.

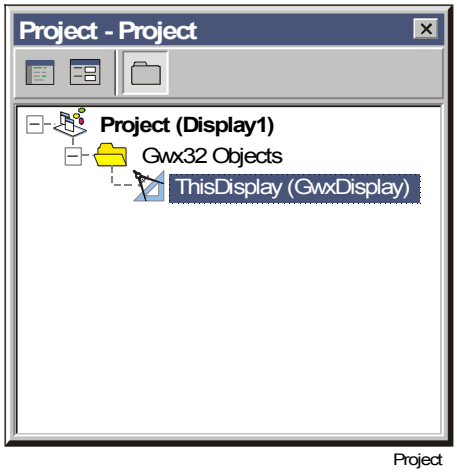


Figure 12-1: Project Window with *ThisDisplay* Module

The VBA programmer has full control over the properties and native objects of the current display and can control the rich animation interface of M-Graphics.

The Automation properties and methods are accessible through the *ThisDisplay* module. *ThisDisplay* is a representation of a current M-Graphics display and contains all Automation properties and methods.

VBA can display Automation methods and properties only if the Automation object is checked in the References dialog box. Note M-Graphics is checked automatically for every new display. Refer to Figure 12-2.

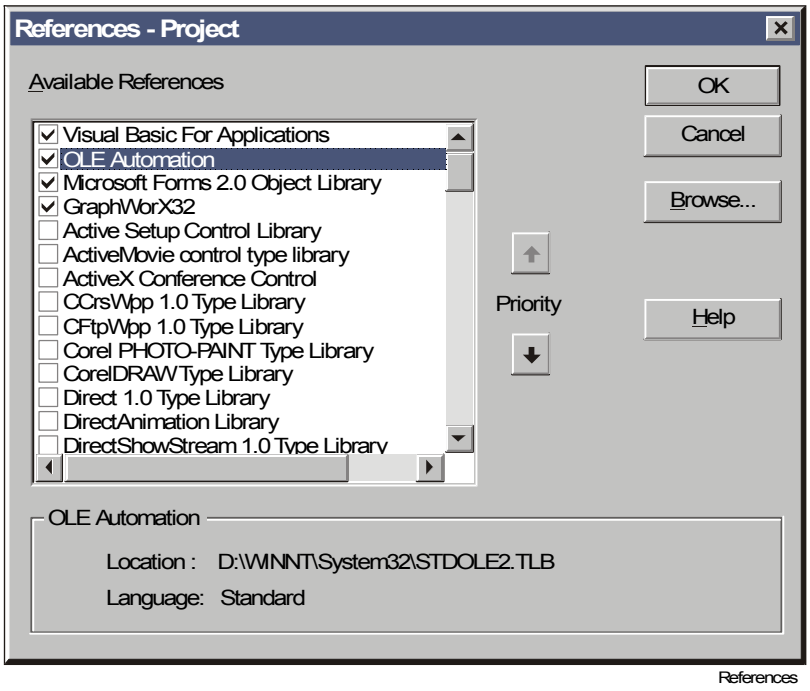


Figure 12-2: References Dialog Box

M-Graphics Native Display Objects

M-Graphics native objects like ellipses, rectangles, symbols, and dynamic actions are not exposed to VBA by default. However they can be referenced from VBA if they have an *Object Name* assigned through Property Inspector dialog box.

A reference (actually a dispatch pointer) to a named object can then be retrieved by one of following methods of the *ThisDisplay*, *Symbol,* and *Visible* objects (Table 12-1).

Refer to the *OLE Automation Ref*erence section of this manual for more information on these functions.

Table 12-1: ThisDisplay, Symbol, and Visible Methods

Method	Description	Example
ThisDisplay.Get VisibleObjectFrom Name("Name").	This function takes the name of a visual object in the display and reports back the object associated with that name. Upon storing the result of this function in a variable, can change the properties of the object in the display by changing the fields of the variable. The following example rotates a rectangle named "Square" 45 degrees from horizontal.	Dim obj As GwxRectangle Set obj = ThisDisplay.GetVisible ObjectFromName("Square") obj.Angle = 45
ThisDisplay.Get DynamicObjectFrom Name("Name").	This function takes the name of a dynamic object in the display and reports back the dynamic object associated with that name. A dynamic object is not a physical, observable object but an abstraction for the operation performed on a visible object (i.e., Hide, Rotation, Size). Once the dynamic object is stored in a variable, you can change its properties. The following example changes the data source of an M-Graphics Hide object to zero hiding the visible object associated with the dynamic object.	Dim obj As GwxHide Set obj= ThisDisplay.GetDynamic ObjectFromName("hd") obj.dataSource = 0
ThisDisplay.Get VisibleObjectFrom Index(Long Value).	This function selects an object based on the order in which visible objects on the screen were created. (The first visible object put on the screen has an index of 0.) This function is useful for iterating through all the objects in a display. The following code turns the first object created to green and the second to red.	Dim obj1 As Object Dim obj2 As Object Set obj1 = ThisDisplay.GetVisibleObject FromIndex(0) Set obj2 = ThisDisplay.GetVisibleObject FromIndex(1) obj1.fillColor = RGB(0, 255, 0) obj2.fillColor = RGB(255, 0, 0)
SymbolName.Get VisibleObjectFromIn dex(Long value)	Each M-Graphics symbol has its own index that keeps track of the objects within it. The GetVisibleObjectFromIndex method, when appended to the name of a symbol, finds the visible object within the symbol with the specified index. This function is useful for iterating through all objects in a symbol. The following example turns the third visible object in a symbol named "sym" to green.	Dim sym1 As GwxSymbol Dim obj As Object Set sym1 = GetVisibleObjectFromName ("sym") Set obj = sym1.GetVisibleObject FromIndex(2) obj.fillColor = RGB(0,255,0) (Note that the third object has index of 2. First has index 0.)
SymbolName .GetVisibleObject FromName("Name")	Finds a visible object within a symbol by the object name specified in M-Graphics.	
VisibleObjectName. GetDynamicObject FromIndex(Long value)	Every time a dynamic object is assigned to a unique visible object, it is assigned an index. The first dynamic object assigned is given an index of zero. Since one visible object can be associated with many dynamic objects, this function provides a useful way of manipulating dynamic objects. The following code takes the second dynamic object associated with a rectangle named "rect" and changes its low range to 10.	Dim o_Vis As GwxRectangle Dim o_Dyn As Object Set o_Vis = ThisDisplay.GetVisible ObjectFromName("rect") Set o_Dyn = o_Vis.GetDynamic ObjectFromIndex(1) o_dyn.lowRange = 10
VisibleObjectName. **GetDynamicObject FromName("*Name*")**	This function takes the name of a dynamic object associated with a visible object and allows you to represent the dynamic object with a variable.	

Table 12-2: Additional VBA Events for Layering

Event	Description
PreAnimateLayer(BSTR layerName)	Fired before data for the layer is requested.
PostAnimateLayer(BSTR layerName)	Fired after data for the layer has been requested.
PreDeanimateLayer(BSTR layerName)	Fired before data for the layer is released.
PostDeanimateLayer(BSTR layerName)	Fired after data for the layer has been requested.

Note: For more information on Layers, refer to the *Arranging Objects* chapter.

Unique Object Names

Sometimes it is desired to access the same objects in all duplicates (clones) of the desired symbol in the same way. This technique is useful for VBA Wizards that have a macro behind symbol. This macro is shared by all duplicates of the symbol and can be run on any of these duplicates.

To allow this feature, M-Graphics supports partial names for objects in symbols. The partial name is a name that ends with an underscore (e.g., MyEllipse_). The duplicates of MyEllipse_ are then MyEllipse_1, MyEllipse_2, ... What is important is the symbol method GetVisibleObjectFromSymbol(partialName) which accepts this partial name and returns the first occurrence of the specified object in specified symbol.

Additional Information

For VBA programming, working with the modules and forms, and customizing of the VBA Editor, refer to *Microsoft Visual Basic 5.0 Programmer's Guide* or documentation on VBA and the help file that comes with VBA. You can open it from the VBA Editor from the menu bar. You can also refer to any VB and VBA document containing useful information.

Note that the VBA in M-Graphics is the same as the one in Microsoft Office applications (Word, PowerPoint®, Access, Excel) and other products. Once you master VBA in M-Graphics, you are able to program in all other applications.

There are many examples included with M-Graphics that are good sources of information and VBA programming tips and tricks.

It is also possible and helpful to open two or more instances of M-Graphics; in each instance open the VBA Editor, and copy and paste the VBA code between the instances.

VBA Wizards

VBA Wizards are M-Graphics objects with Visual Basic code behind them. The code is run either in Design mode to help to configure M-Graphics object or in Runtime mode to execute a specific task.

Normally, the VBA code is stored in the current document, saved from the VBA Editor when the display is saved, and re-loaded to VBA Editor when a display is open in M-Graphics.

However, if the rules described below are followed, the VBA code can be bound to an M-Graphics object. When such an object is pasted/dropped to another instance of M-Graphics, to Symbol Library, or to scrap (desktop), this code goes with it.

The VBA Wizard can be run to perform a specific task either in Design or Runtime mode.

Design Mode

M-Graphics can launch the VBA Wizard macro in Design mode by double-clicking the VBA Wizard.

By default when an object is double-clicked in Design mode, a Property Inspector is launched. However, if there is a special keyword in the first line of the Custom Data field of Property Inspector, a macro can be run.

The format of the key word for a macro *MacroName* is:

```
OnDoubleClick=<GwxMacroName_Main.MacroName>, Parameters=<>
```

There must exist a macro *MacroName* in the module *GwxMacroName_Main* in VBA to successfully run the macro. The user is allowed to put any string between angle brackets of a *Parameters*=<> section. Any data worth sending to the macro would be put between the angle brackets. These data are then available when the macro runs.

The name of a macro cannot contain spaces.

Runtime Mode

M-Graphics can launch the VBA Wizard macro in Runtime mode by clicking an M-Graphics button or a pick action configured to run a macro. The Action field must be configured to Run VBA Macro. The Macro Name field must contain a macro name in its format:

```
GwxMacroName_Main.MacroName
```

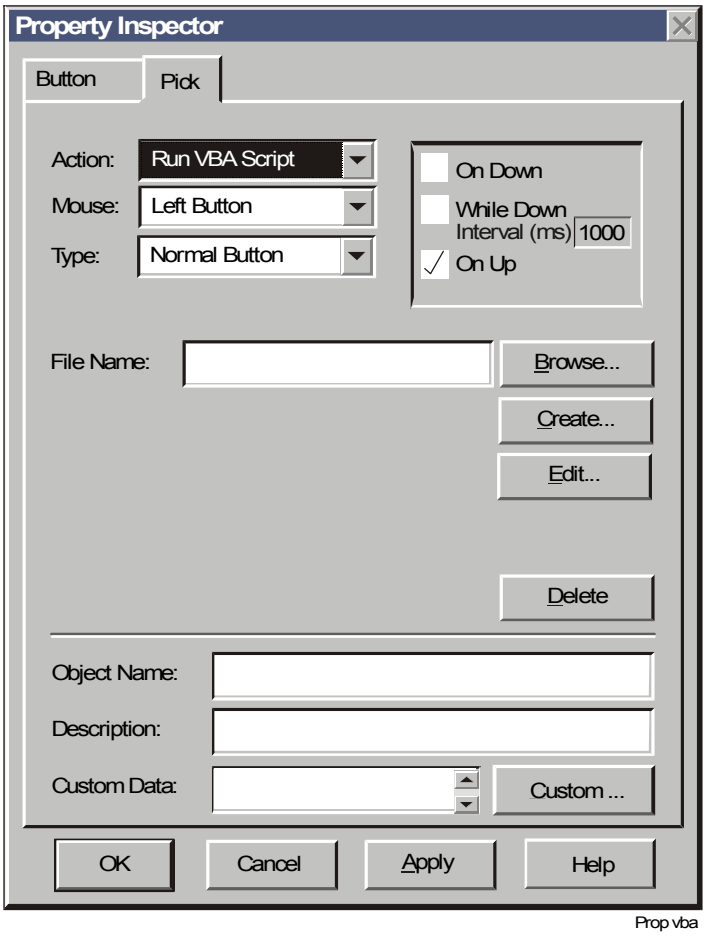


Figure 12-3: VBA Wizard in Button

Custom Data can contain any string keeping custom data as desired between angle brackets.

VBA Wizard Rules

M-Graphics takes care of code behind the VBA Wizard. Due to the special naming convention of all modules belonging to a specific VBA Wizard, the code in these modules can be properly moved with the symbol.

If the macro name of the VBA Wizard is *MacroName*, then all code modules must start with the string *GwxMacroName_*. This technique allows more code and form modules to be used for one VBA Wizard object and facilitates moving all this code with the object when necessary.

VBA Wizard Creation Tool - Macro

Because creating VBA Wizard objects would be a tedious task, M-Graphics offers a useful creation tool: a macro that converts the object to VBA Wizard and generates the VBA template code. The code can then be easily enhanced and modified by the user.

When a VBA Wizard is deleted or moved out of the current display, the VBA code is not deleted automatically. However, you can use Clean Unused VBA Modules to remove all modules from the VBA Editor which start with the *Gwx* string and are not referenced from the currently displayed VBA Wizards (either Design or Runtime mode based).

If you want to create a new Runtime based VBA Wizard, add a pick dynamic with the Run VBA Script. Refer to the *Adding Dynamics* chapter of this manual.

Procedure Overview

Table 12-3: Using VBA Scripts

To Do This	Follow These Steps:
Create and Edit VBA Scripts	On the Tools menu, select Macros > Visual Basic Editor. Add code, forms, etc., to accomplish desired task.
Add Event Handlers	Open a display. On the Tools menu, select Macros > Visual Basic Editor. Open the *ThisDisplay* module from the VBA Editor. Select the *GwxDisplay* module in left combo box of *ThisDisplay* module from the VBA Editor. Select the desired event from the list in the right combo box. On the File menu, select Save Display 1. On the File menu, select Close and Return to M-Graphics.
Display the *ThisDisplay* Module Code	On the View menu of the VBA Editor, select Project Explorer. Expand the M-Graphics Objects group and double-click the *ThisDisplay (GwxDisplay)* item.
Create a Macro	Create a symbol. On the Tools menu, select Macros > Create Macros. Fill in the parameters. Click OK. Type code. Return to M-Graphics and double-click on symbol.
Edit a Macro	Select a macro object in the display. On the Tools menu, select Macros > Edit Macro.
Step a Macro	Select a macro object in the display. On the Tools menu, select Macros > Step Macro.
Run a Macro	Select a macro object in the display. On the Tools menu, select Macros > Run Macro.
Clean VBA Unused Modules	On the Tools menu, select Macros > Clean VBA Unused Modules.

Detailed Procedures

Creating and Editing VBA Scripts

To create and edit VBA Scripts:

1. On the Tools menu, select Macros > Visual Basic Editor. The Visual Basic Editor appears (Figure 12-4).

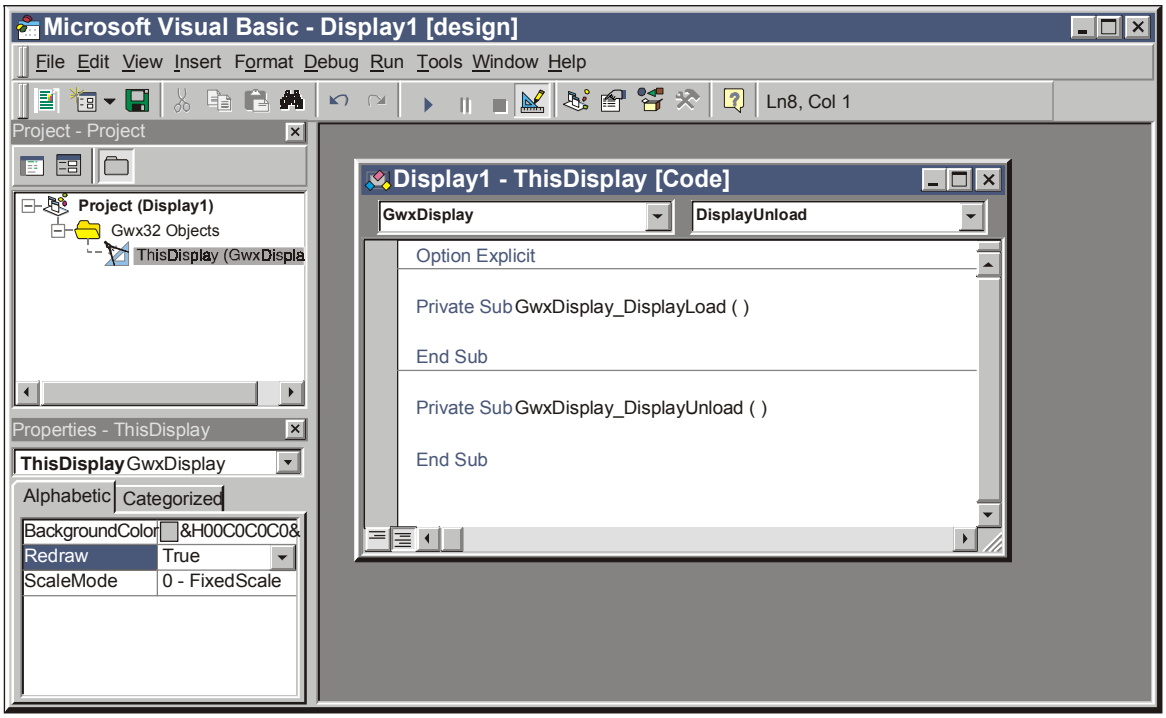


Figure 12-4: Visual Basic Editor

2. Add code, forms, etc., to accomplish desired task.

Adding Event Handlers

To add event handlers:

1. Open a display.
2. On the Tools menu, select Macros > Visual Basic Editor. The M-Graphics VBA Project loads by default.
3. Open the *ThisDisplay* module from the VBA Editor.
4. Select the *GwxDisplay* module in left combo box of *ThisDisplay* module.
5. Select the desired event from the list in the right combo box.

Note: The subroutine header is automatically inserted to current module. Insert your VBA code to the body of the subroutine.

6. On the File menu, select Save Display 1.
7. On the File menu, select Close and Return to M-Graphics.

Displaying the *ThisDisplay* Module Code

To display the *ThisDisplay* module code:

1. On the View menu of the VBA Editor, select Project Explorer.
2. Expand the M-Graphics Objects group and double-click the *ThisDisplay (GwxDisplay)* item. A *ThisDisplay* code window appears (Figure 12-5).

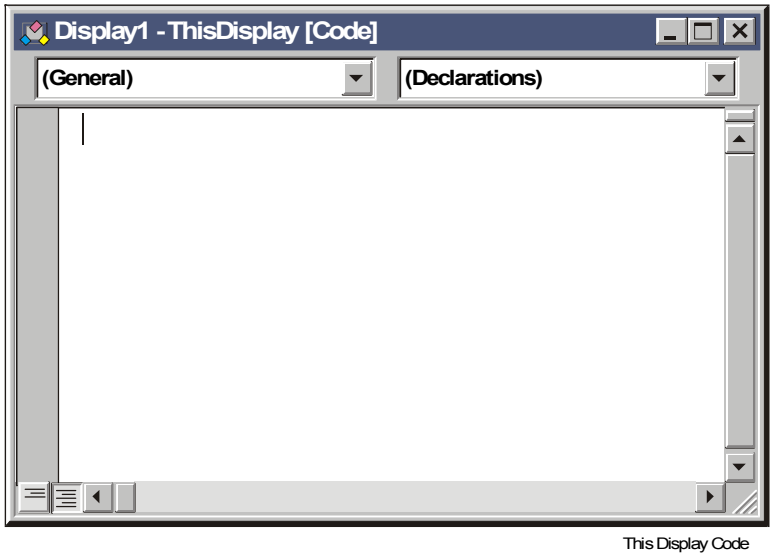


Figure 12-5: *ThisDisplay* Code Window

ThisDisplay contains two combo boxes at the top.

The top left combo box allows select items like:

(General)

GwxDisplay

The top right combo box shows events for the left combo selection. For example, the *GwxDisplay* item has events like *DisplayLoad*, *DisplayUnload*, and others.

Creating a Macro

To create a macro:

1. Create a symbol.
2. On the Tools menu, select Macros > Create Macros. The VBA Script Wizard dialog box appears (Figure 12-6).

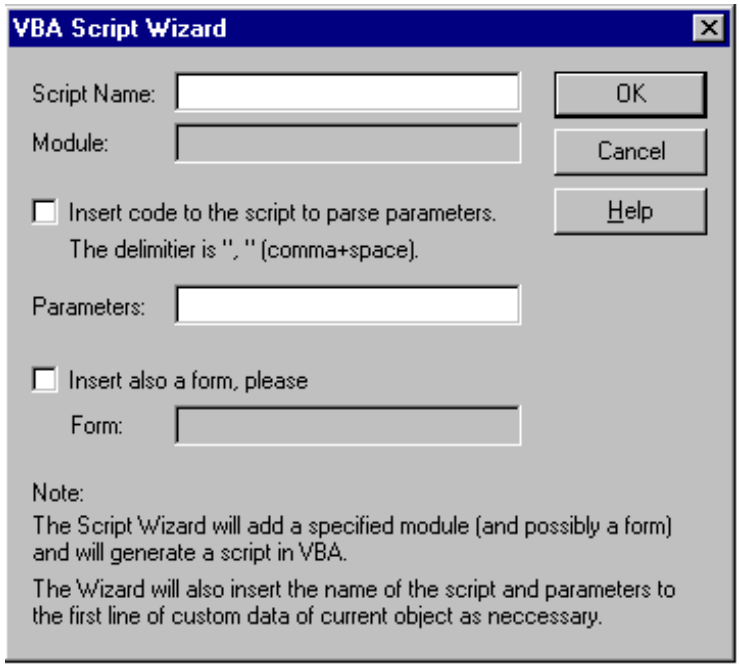


Figure 12-6: VBA Script Wizard Dialog Box

3. Fill in the parameters using Table 12-4.

Table 12-4: VBA Script Wizard Dialog Box Parameters

Parameter	Description
Script Name	Type the name of your script. The script name should begin with a character and should contain alphanumeric characters only. If there is already a macro or a module of that name, choose another name.
Module	Field Module is always grayed-out, because the module name is generated automatically based on macro name.
Insert code to the script to parse parameters.	Check this box to generate extra code in the body of the script subroutine. It helps in retrieving and storing parameters from the VBA Wizard object. This code uses the *GwxTools* module to convert parameters to a string named StrPar which is local to your script subroutine.
Parameters	Type any string you like. You can obtain this string when the macro runs. This field is designed to allow custom data specific to a VBA Wizard instance. Different instances of the same objects can keep different data. Using this field is optional. This feature simplifies your code by allowing you to pass values into a macro.
Insert also a form, please	Check this checkbox if you need a VBA form to be launched from the macro. You are allowed to create any number of forms for the VBA Wizard assuming you follow the naming convention. (If you don't follow the convention, the VBA code is not moved with the object when necessary.)
Form	Field Form is always grayed-out and generated automatically based on macro name.

4. Click OK. A VBA Editor starts, and the cursor appears in the body of *Test* subroutine in a module *GwxTest_Main*.
5. Type the code which is run when you double-click on the symbol in Design mode. Try the example code as shown in Figure 12-7.

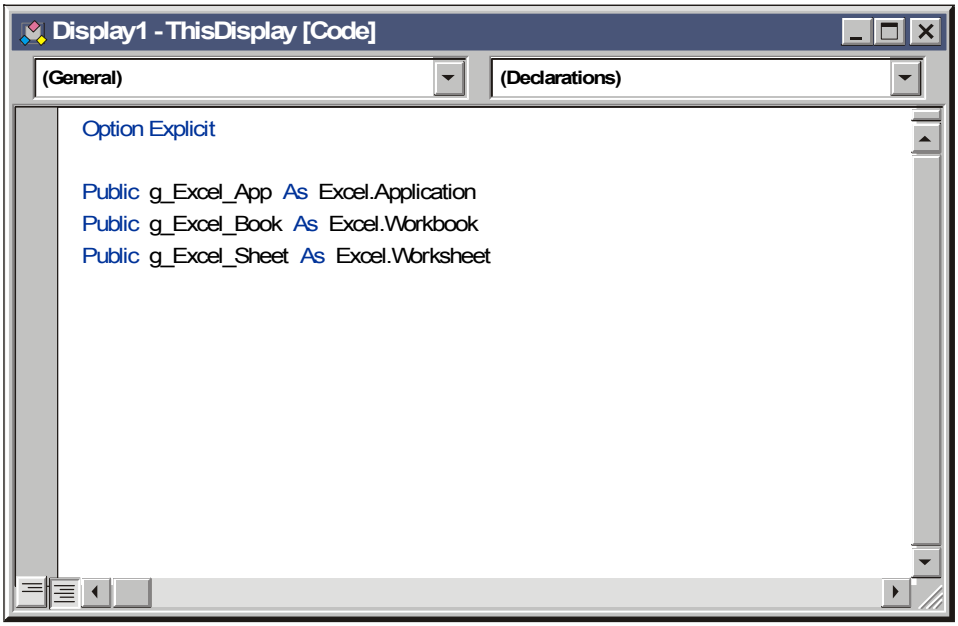


Figure 12-7: Test Subroutine

6. Return to M-Graphics, and double-click the symbol. A beep sounds, and a message appears.

Editing a Macro

To edit a macro:

1. Select a macro object in the display.
2. On the Tools menu, select Macros > Edit Macro. The VBA Editor appears with the cursor in the body of the macro.

Stepping a Macro

To step a macro:

1. Select a macro object in the display.
2. On the Tools menu, select Macros > Step Macro. The VBA Editor appears with the cursor on the first line of the macro.

Running a Macro

To run a macro:

1. Select a macro object in the display.
2. On the Tools menu, select Macros > Run Macro.

Cleaning VBA Unused Modules

To clean VBA unused modules:

On the Tools menu, select Macros > Clean VBA Unused Modules.

Examples

The following section provides several examples for reference.

Adding Event Handlers Example

Write code to pop-up a message box that displays the message `Runtime started` when you Runtime your display.

1. Open a display.
2. On the Tools menu, select Macros > Visual Basic Editor. The M-Graphics VBA Project loads by default.
3. Open the *GwxDisplay* module from the VBA Editor.
4. Select the *PostRuntimeStart* event from the list in the right combo box.
5. Insert VBA code as shown in the figure below to the body of the subroutine in the subroutine header (Figure 12-8).

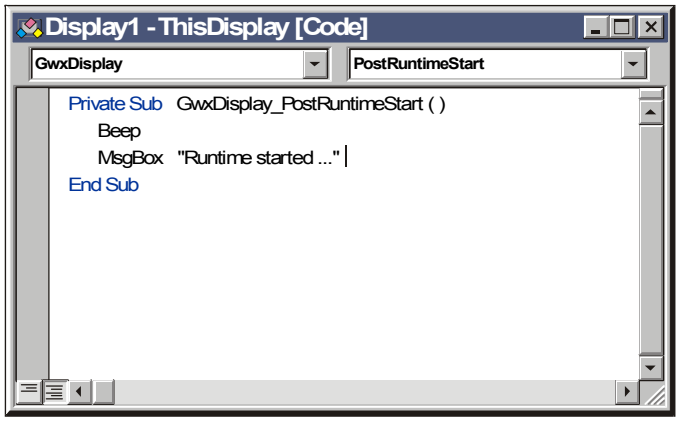


Figure 12-8: PostRuntimeStart Event Example

6. Switch to M-Graphics by closing the VBA Editor.
7. Test the event by clicking on Runtime on the menu bar.
8. You hear a beep, and a message box with a message `Runtime started` appears.

M-Graphics Native Display Objects Example

Write code for changing the color of ellipse M-Graphics native object during Runtime:

1. Load an existing display, or create a new M-Graphics display.
2. Draw an ellipse by picking Ellipse from the Draw toolbar.
3. Name the Ellipse Object for this display:
   a. Double-click on the Ellipse. The Property Inspector box opens.
   b. Type the name MyEllipse in the object name field and then click on OK**.**
4. Open VBA Editor.
5. Select the *ThisDisplay* module.
6. Select the *GwxDisplay* module in left combo box of *ThisDisplay* module.
7. Select the *PostRuntimeStart* event from the list in the right combo box.

8. The subroutine header is automatically inserted into the current module. Insert the VBA code as in the diagram below to the body of the subroutine.

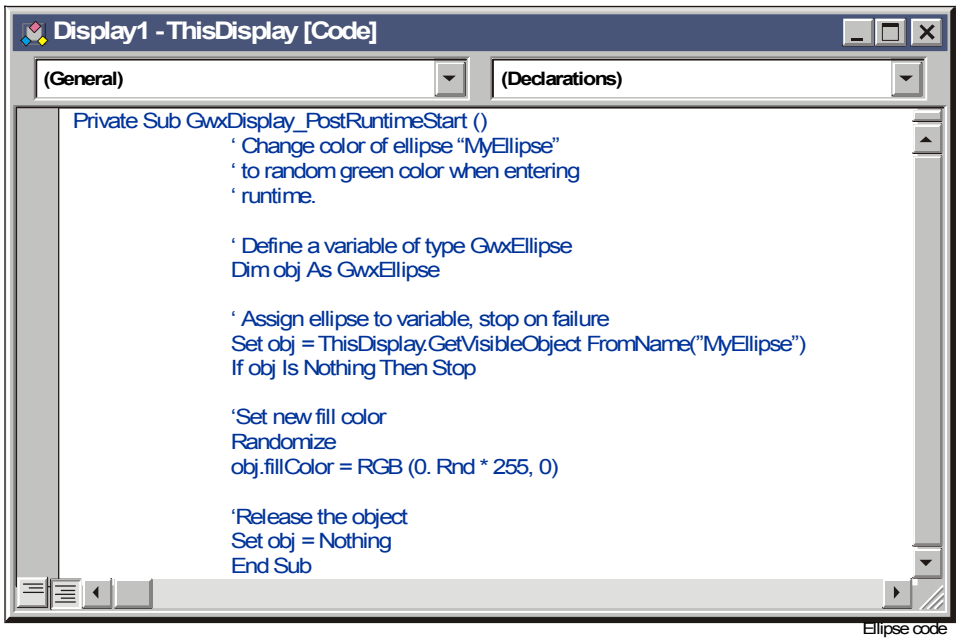


Figure 12-9: M-Graphics Ellipse Modified From VBA

9. Save the M-Graphics Project by selecting File - Save Display 1.
10. Test the example by switching to the display and going into Runtime. The color of the ellipse changes to green.

Unique Object Names Example

Assume you have a symbol which consists of a rectangle and an ellipse. Write code that modifies these objects in any copy of the symbol. (You must assign partial names to both objects, which name them MyRect_ and MyEll_.) Here is the code that shows how to access these objects in a specific symbol:

```
Dim sym As GwxSymbol
Set sym = FindSomehowDesiredSymbol()
' user method to choose the symbol
Dim ell As GwxEllipse, rect As GwxRectangle
Set ell= sym.GetVisibleObjectFromName("MyRect_")
Set rect= sym.GetVisibleObjectFromName("MyEll_")
' do something with these objects
' release references
Set ell = nothing
Set rect = nothing
Set sym = nothing
```

The following list demonstrates the unique name technique:

1. Create one rectangle, and assign object name Rect_1.
2. Duplicate this to create one more rectangle.
3. Group Rect_1 and Rect_2.
4. Duplicate this group using the Duplicate button.
5. Ungroup the duplicate object.
6. Read the Object name by using the Property Inspector dialog box. Notice that the Object names of the two rectangles (the third and the fourth) are Rect_3 and Rect_4.

Using VBA to Connect with Other Applications Example

Design a display and a spreadsheet, each with two data points, and have them communicate to each other through VBA.

1. Open a new M-Graphics project.
2. On the Tools menu, select Macros > Visual Basic Editor.
3. On the Tools menu, select References. This opens a list of available references to applications. Check the box next to Microsoft Excel Release 8.0 Object Library.

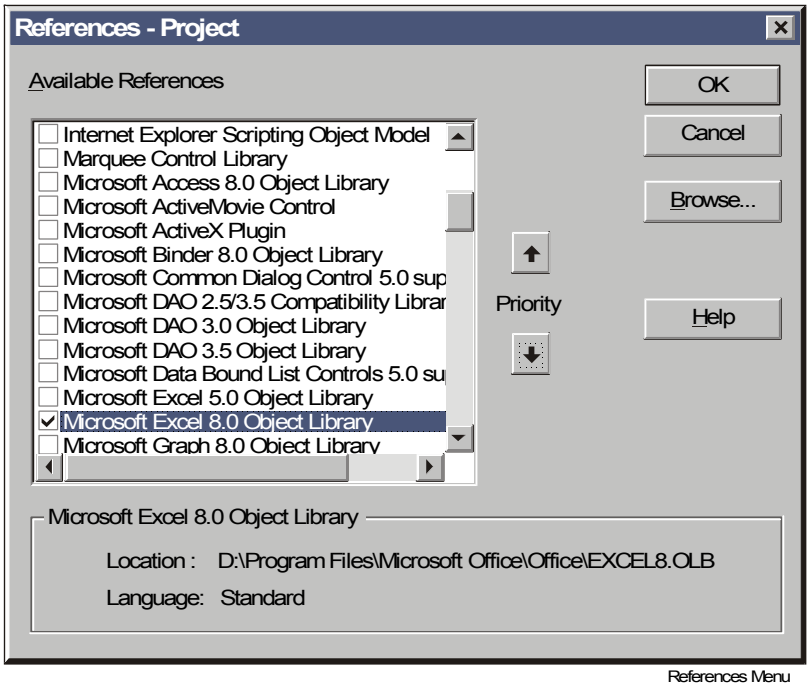
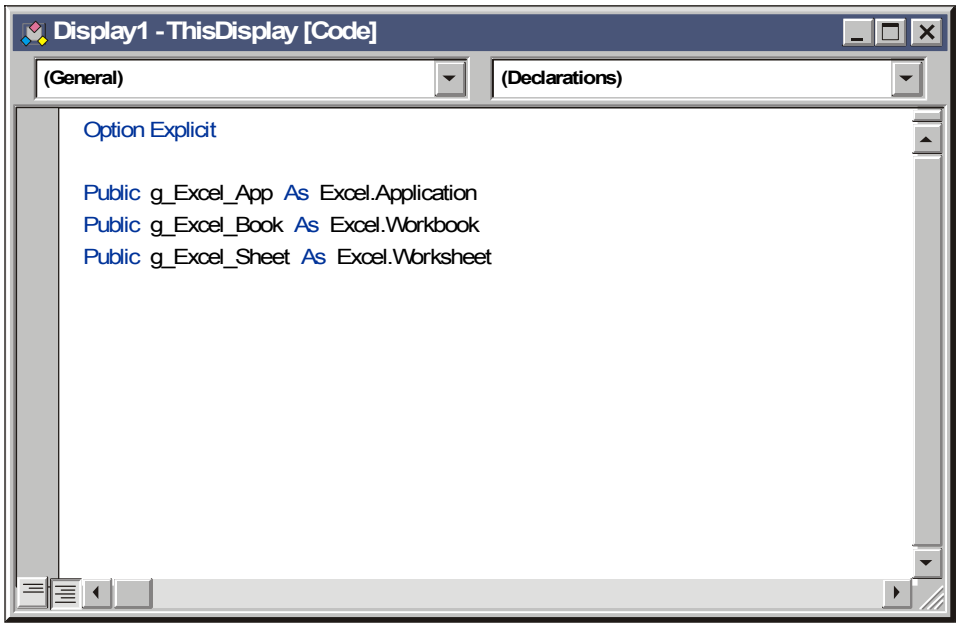


Figure 12-10: References Menu

4. In the Project viewer window, double-click on the *ThisDisplay* Module. You should see a code window with the words `Option Explicit`. In that window, you want to make global declarations that will be used later:

```
Option Explicit
Public g_Excel_App As Excel.Application
Public g_Excel_Book As Excel.Workbook
Public g_Excel_Sheet As Excel.Worksheet
```



This Display Code1

Figure 12-11: ThisDisplay Code Window with Global Declarations

5. In the left combo box at the top of the code window, select GwxDisplay. In the right combo box, select PreRuntimeStart. By entering the following code, you cause Excel to be launched before M-Graphics goes into Runtime.

```
Private Sub GwxDisplay_PreRuntimeStart()
  ' Open up Excel and make it visible
  Set g_Excel_App =
CreateObject("Excel.application")
  g_Excel_App.Visible = True
  ' Open up a sheet
  Set g_Excel_Book = g_Excel_App.Workbooks.Add
  Set g_Excel_Sheet = g_Excel_Book.Worksheets(1)
  ' Initialize the two cells you will be using
  g_Excel_Sheet.Range("a1") = 0
  g_Excel_Sheet.Range("a2") = 0
End Sub
```

6. Create two process points in your display. Make sure that the Data Entry checkbox is checked for both process points. Connect one point to a local variable ~~a1~~ and the other to a local variable named ~~a2~~. This double tilde is a standard notation for local variables in M-Graphics.

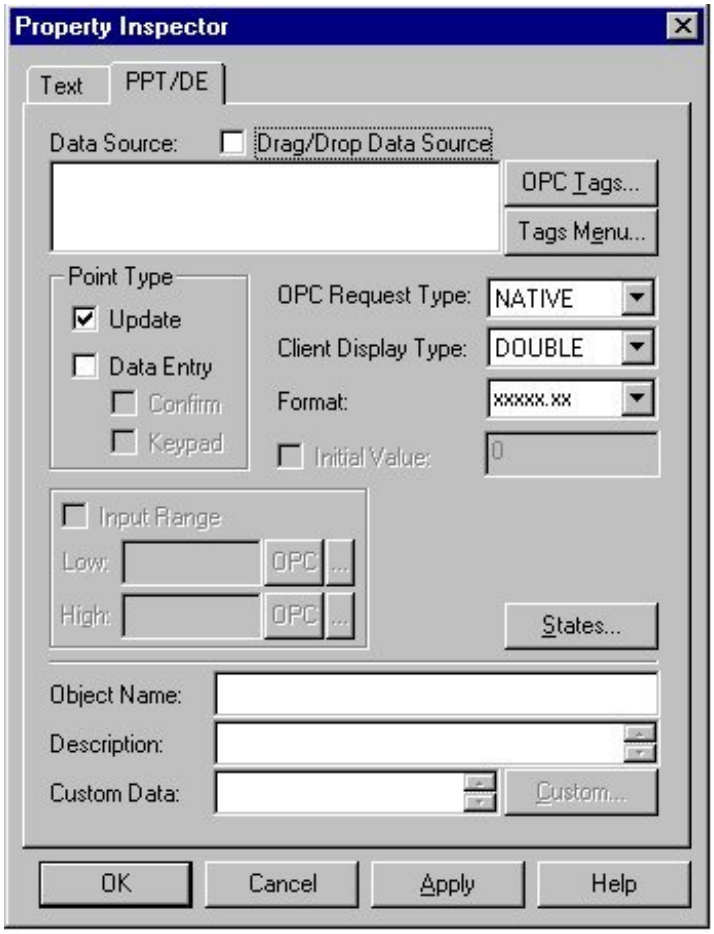


Figure 12-12: Configuration of Process Point

7. Now go back to the Visual Basic editor.
8. Make a new procedure designed to read from and write to Excel. Go to Insert-Procedure in the menu bar.

9. Name the procedure Read_Write, and enter in the following code:

```
Public Sub Read_Write(ByVal Co As Integer)
' This procedure reads from or writes to excel
' based on the condition, Co, passed as a
' parameter

  Dim Point As GwxPoint
  Dim St As String  'Used to store cell name
  Dim St2 As String 'Used to store variable name
  Dim X As Integer
  For X = 1 To 2
    ' add "a" to the value X converted to string
    St = "a" + Mid(Str(X), 2, 1)
    St2 = "~~" + St + "~~"
    Set Point=ThisDisplay.GetPointObjectFromName(St2)
    If Co = 1 Then  'Write to Excel
      ThisDisplay.g_Excel_Sheet.Range(St)=Point.Value
    Else            'Read From Excel
      Point.Value=ThisDisplay.g_Excel_Sheet.Range(St)
    End If
  Next X
End Sub
```

10. Return to the display, and create two radio buttons. Name one of them Read from Excel and the other Write to Excel. Configure both of them to run a VBA Script. Keep the execute.

11. Create a new macro for each of the two radio buttons. Call one macro Rd and call the other Wr. Both of these two macros will call the Read_Write procedure. The Code for each macro should look like this:

```
Sub Wr(o As GwxPick)
  Call ThisDisplay.Read_Write(1)
End Sub

Sub Rd(o As GwxPick)
  Call ThisDisplay.Read_Write(0)
End Sub
```

12. Go back to the display, and go into Runtime. Notice Excel starts up automatically with the value 0 in the A1 and A2 cells. If you click on the Write to Excel button and change the value of one of the process points in M-Graphics, you will notice the value in one of the cells in Excel changes. If you click on the Read from Excel button and change one of the two values in Excel, the process point will then update to match the value in Excel.

Troubleshooting

Table 12-5: Troubleshooting

Problem	Solution
An `Error Parsing Data Source` message appears after left or right mouse-click on a point data display which runs Visual Basic for Applications (VBA) in a standard point object or in a point dynamic not properly mapped during alias definition.	The graphic contains a corrupted point object. Find this object and delete it. To delete the corrupted object from the display: 1. Select Edit >Find and set the Type to Text Label. 2. For example, if <<N1.CSData.S>>.Present_Value is the corrupted information, in the Find What field, paste: <<N1.CSData.S>>.Present_Value. 3. Click Apply. 4. Select the one object that appears in the displayed tree diagram. 5. Click Show Selection. In the top left corner, the blue handles of the corrupted object appear. 6. Click Close. 7. Click Delete. 8. Click Save As to save the display under a different name. 9. Exit M-Graphics. 10. Restart M-Graphics. 11. Load the display in Runtime mode.
Multiple VBA Option Explicit statements appear in graphic files with VBA or duplicate VBA codes appear. The graphic shows a compile error in VBA when running. This occurs after templates have been edited and updated in the graphics.	After editing or updating templates, delete any multiple Option Explicit statements using the VBA Editor. This problem is fixed at M-Web Release 2.0, M3 Workstation Release 2.0, or M5 Workstation Release 1.1 with M-Graphics Release 3.0. Earlier files with existing problems need to be edited to remove redundant code lines after upgrading to M-Graphics Release 3.0. Files produced at M-Graphics Release 3.0 do not have this problem.