WINTASK

Develop efficient and reliable automation scripts

Version 3.8a

AUTOMATION FOR WINDOWS XP, Vista, Windows 7, 2003 and 2008 Server

Published by: TaskWare
18, allee des Lilas
92150 SURESNES France

© Copyright 1997-2012 TaskWare January 2012

WinTask™ is a trademark of TaskWare.

CONTENTS

Introduction **5**
Why WinTask? 5
WinTask Components 5
TASKEDIT 5
TASKCOMP 5
TASKEXEC 6
SPY 6
TASKWIZ 6
DIALOG BOX EDITOR 6
Recording Mode **7**
Exercise 1 7
Capture/Replay Tool Myths 8
Exercise 2 8
A Simple Automation Script 9
Exercise 3 11
WinTask IDE **13**
Exercise 4 13
Terminating a Script 14
Synchronization **15**
Synchronization Methods 15
Exercise 5 17
Exercise 6 21
WinTask Scripting Language **24**
Functions 24
Variables 26
System Variables 26
Integers 27
Strings 27
Reals 27
Arrays 27
Operators 27
Window Management Functions **29**
#IgnoreErrors 29
#ActionTimeout 29
#UseExact 30
ExistW() 30
Focus$() 31
Top$() 31
Exercise 7 32
Iteration **34**
Iteration 34
Exercise 8 34
File Functions **36**
Exist() 36
Kill() 36
Read() 37

Eof() 38
Write() 38
Exercise 9 39
ReadExcel and WriteExcel 40
Exercise 10 41
Subroutines and Functions 42
Sub...ExitSub...EndSub 42
Converting a Script into a Subroutine 42
Exercise 11 44
Function...ExitFunction...EndFunction 45
Exercise 12 46
Putting it all Together 47
Exercise 13 48
Debugging 50
Compilation errors 50
Execution errors 50
Exercise 14 50
Exercise 15 51
Exercise 16 56
Conclusion 57
APPENDIX A 58
WinTask Toolbar 58
APPENDIX B 60
WinTask Floating Toolbar 60
APPENDIX C 62
Exercise Solutions 62
Exercise 7, Script07a.src 62
Exercise 7, Script07b.src 63
Exercise 7, Script07c.src 64
Exercise 7, Script07d.src 65
Exercise 8, Script08a.src 66
Exercise 8, Script08b.src 67
Exercise 9, Script09.src 68
Exercise 10, Script10.src 69
Exercise 11, Script11.src 70
Exercise 12, Script12.src 72
Exercise 13, Script13.src 74
Glossary 77
Index 79

Introduction

This manual has been written to give WinTask users a comprehensive resource to create efficient and reliable automation scripts. The manual covers the WinTask tool-set, the WinTask script language and the WinTask Application Programming Interface (API). This manual covers pure Windows applications automation but does not cover Web automation – this part is covered in another manual that can be downloaded from www.wintask.com/manuals.php.

The manual also provides exercises that help to illustrate the capabilities of WinTask and how to avoid common pitfalls. The exercises in this manual use the readily available Microsoft Windows Notepad™ and WordPad™ applications, and the www.wintask.com Web site as automation targets. Once the reader masters the exercises, it should be an easy transition to apply that knowledge to the applications that they wish to automate.

Why WinTask?

WinTask is a complete automation tool for Windows. It can automate any Windows application or Web page. WinTask is more than just another macro recorder. The automation script can be extended to include standard programming paradigms well beyond a simple repetition of steps. It provides too Capture tools for extracting data from the application under automation, and it includes OCR techniques for even capturing text embedded in an image. To execute your automation scripts unattended, you can schedule them using Windows built-in Task Scheduler.

WinTask Components

WinTask is composed of several components. Each of the components is fully explained later in the manual. Accompanying exercises illustrate how to use each component.

TASKEDIT

This is the WinTask Integrated Development Editor (IDE) Windows application. With the Editor, the user can record keyboard entries and mouse actions to generate an automation script file. The WinTask Script Language syntax is similar to Microsoft Visual Basic and provides a powerful set of library functions. Script files can be modified through the Editor, compiled, and executed to automate any application. WinTask Script Language files are stored in ASCII format (or Unicode) and have a *.SRC* extension. The Editor is a multi-pane window, main one showing the script code, the one on the right displays all the functions available in the language (the Language window), the one at the bottom displays results when the script is executed (the Output window). At any time, help on a function can be accessed by double-clicking its name in the Language window.

TASKCOMP

This WinTask component is a command line program that compiles WinTask Script Language files into a file that can be interpreted by the Executor. It can be invoked too through the Editor. The compiler flags any violations of the WinTask Script Language syntax in the script file and reports them to the user (Compilation errors are

listed in the Output window). The compiler generates two files with *.ROB* and *.LST* extensions for each script file compiled. The executable script files are created with the *.ROB* extension and are binary in nature. The *.LST* files are where the compiler logs errors and warnings along with detailed program statistics. This file is an ASCII file.

TASKEXEC

The WinTask Executor executable interprets compiled WinTask Script Language files and generates the keystrokes and mouse actions that automate a Windows application. Compiled WinTask Script Language files have a *.ROB* extension. A separate instance of the Executor executes each compiled script file.

SPY

The WinTask Spy component is used to peer into the internal data structures of Windows to extract the names that Windows uses to reference the different application windows and controls. It can also used to extract information from web pages such as control names and HTML tag information. This information is vital in order to allow the WinTask Executor to access the application to be automated. The Window names are passed to various WinTask Script Language library functions in the script files.

TASKWIZ

The WinTask Editor provides a recording mode that provides access to a set of Synchronization Wizards. The wizards aid in automation script development to compensate for external events that may not always occur at predictable intervals during script execution. The Synchronization Wizards are explained in the chapter titled **Synchronization**.

DIALOG BOX EDITOR

The WinTask Editor provides a dialog box editor that allows the user to develop dialog boxes for display during the execution of automation scripts. Dialog boxes provide the ability to collect information during automation script execution to customize the actions of the script from run to run. Development and invocation of dialog boxes is explained in the chapter titled **Dialog Boxes**.

Recording Mode

The quickest way to start automating an application or process is to record your keystrokes and mouse actions into an automation script. The WinTask Editor has a recording mode that performs this function. The automation script can then be saved and replayed at a later date.

Exercise 1

This exercise demonstrates how to record a user session using Notepad as the application to be automated.

NOTE: The 32-bit versions of notepad and wordpad, not the 64-bit versions, are used in all the exercises of this book (Under a Windows 64-bit, wordpad is located in c:\Program Files (x86)\Windows NT\Accessories).

1. Launch the WinTask Editor by clicking the Windows **Start** button, then click **All Programs**, select **WinTask** in the list of Programs and double click **WinTask**. If Your First Script Wizard screen comes up, click **Cancel** button.

2. When the WinTask Editor window comes up, click the **Record** button Record on the WinTask Toolbar to start Recording Mode.

3. When the **Starting Recording Mode** dialog box appears, click the **OK** button.

4. The **Launching a Program** dialog box will appear. Enter **Notepad** into the **Program** field and click the **OK** button.

5. The Notepad window will open and WinTask will transform itself into a floating toolbar titled **WinTask Toolbar**. There will also be a flashing icon 7:33 PM added to the system tray that indicates that Recording Mode is active. Click on the Notepad text area and type **Hello**.

6. From Notepad, select menu item **File/Exit**.

7. Select **Don't save** button (or **No** button under XP or Windows 2003) when Notepad presents the **Save Changes** dialog.

8. Click the **Stop Recording** button on the WinTask Floating Toolbar to stop Recording Mode.

9. The WinTask Editor window is restored to its normal size and the recorded automation script is displayed in the text area.

10. Click the **Play** button Play on the WinTask toolbar to run the previously recorded script. You are prompted for saving the file, give the name **script01**, and click **Save** button, you see all your actions replayed.

11. From the WinTask Editor, select menu item **File/Exit**.

Capture/Replay Tool Myths

At first glance, it may appear that any Capture/Replay tool is sufficient to automate a task. It soon becomes apparent that unless the tool is flexible enough to handle a dynamic PC environment, the replay aspects will not meet expectations.

Exercise 2

In this exercise WinTask will be used in a purely Capture/Replay style. It will be shown that generated automation scripts may not always execute successfully.

1. Launch the WinTask Editor. If Your First Script Wizard screen comes up, click **Cancel** button. The title bar should display **WinTask – [Untitled1]**. If not click **New** icon New on the WinTask toolbar.

2. Click the **Record** button on the WinTask Toolbar to start Recording Mode.

3. When the **Starting Recording Mode** dialog box appears, click the **OK** button.

4. The **Launching a Program** dialog box will appear. Enter **Notepad** into the **Program** field and click the **OK** button.

5. The Notepad window will open and WinTask will transform itself into a floating toolbar titled **WinTask Toolbar**. Click on the Notepad text area and type **Hello** followed by the **Enter** key.

6. From Notepad, select menu item **Edit/Time/Date**.

7. From Notepad, select menu item **File/Exit**.

8. Select **Save** button (or **Yes** button under XP or 2003) when Notepad presents the **Save Changes** dialog.

9. Save the file as **test02.txt** in the current folder.

10. Click the **Stop Recording** button on the WinTask Floating Toolbar to stop Recording Mode.

11. The WinTask Editor window is restored to its normal size and the recorded automation script is displayed in the text area.

12. Click the **Play** button Play on the WinTask Toolbar to replay the actions listed in the script. WinTask will now compile the script before running it. You will be prompted to save the script. Enter the name **script02.src** when the **Save As** dialog is displayed. The **Compilation** results are displayed in the Output window of the Editor and the compiled script starts its execution. Please refrain from moving the mouse or entering any keystrokes as the script is executed.

13. As you watch the WinTask Executor execute the compiled script you will see a repetition of Steps 5 through 8. You will notice that Notepad doesn't save the file and the remains open on the **Confirm Save As** dialog box.

14. Click the **No** button on the message box presented by the **Confirm Save As** dialog box, select **Cancel** on the **Save As** dialog box and close Notepad. Click **Don't save** button (or **No** button under XP/2003) when Notepad presents the **Save Change** dialog.

15. Questions for Discussion: What are the causes of the script's failure to run to completion? How might the script be modified to allow it to run to completion?

When your actions were recorded in the script, file test02.txt did not exist. When the script is replayed, the Notepad Save As dialog detects that a file by the same name exists and prompts the WinTask Executor to confirm that the file is to be overwritten. The script has not been created to handle this situation and simply terminates assuming that the file was saved and Notepad had been closed.

Without manual modifications to an automation script, the script will only complete successfully if the PC environment is IN THE SAME STATE as at the time when the script was recorded. Using Exercise 2 as an example, file test02.txt must be deleted prior to the execution of the compiled script02.src. This can be accomplished by manually deleting the file, defeating the purpose of an automation script, or modifying the script to delete the file if it exists.

A Simple Automation Script

So far we have ignored the contents of the automation script file created during Exercise 2. The following section examines the individual lines stored in script02.src. The lines from the script file are listed in *Italic* followed by a short explanation of what action each line performs. Based upon your actions during the recording session, your version of the file may contain additional lines not documented here.

The user can view the script file in the WinTask Editor. Placing the cursor on the name of the function call and pressing the F1 key will bring up the WinTask Help System for the function. Descriptions of many WinTask Application Programming Interface functions can be found throughout this manual.

Shell("Notepad",1)

Shell starts the specified application as a new process. The second parameter controls whether the application starts minimized, maximized, or normal. This line launches Notepad with the window size it had when it last closed.

UseWindow("NOTEPAD.EXE|Edit|Untitled - Notepad|1",1)

UseWindow defines the target window for all WinTask generated keystrokes and mouse actions. The function will wait for the specified window to be displayed up to a timeout value before displaying an error message. The default timeout value is 30 seconds. This line will wait for Notepad to fully load and display on the Windows desktop. Focus is placed in the Notepad text area.

SendKeys("Hello<Enter>")

SendKeys sends the specified string to the window specified by the previous *UseWindow* call. The non-printing and special keyboard keys are specified inside angle brackets < >. This line will enter **Hello** followed by the **Enter** key into the text area of Notepad.

UseWindow("NOTEPAD.EXE|Notepad|Untitled - Notepad",1)

This *UseWindow* function call changes the target for keystrokes and mouse actions to the Notepad menu bar.

ChooseMenu(Normal,"&Edit|Time/&Date   F5")

ChooseMenu selects the specified menu item for the window specified by the previous *UseWindow* call. This line will select the Notepad sub-menu item **Time/Date** below the **Edit** menu item. Note that the second parameter to *ChooseMenu* has an ampersand in the **&Edit** and **Time/&Date** menu items in addition to **F5**. They represent the windows shortcut keys <ALT>+E for the **Edit** pull-down menu and <ALT>+D and F5 for the **Time/Date** menu item.

The ampersand is used internally by Windows to designate the next character as the Windows menu shortcut key. The F5 key provides hot key access to the menu item. The *ChooseMenu* function may not select the desired menu item if the shortcut keys are not defined.

ChooseMenu("&File|E&xit")

This line will select the Notepad sub-menu item **Edit** below the **File** menu item. This line will close Notepad.

UseWindow("NOTEPAD.EXE|CtrlNotifySink|Notepad|7",1)

This line changes the target for keystrokes and mouse actions to the Notepad message box that asks the user if they would like to save the entered text.
Under XP or 2003, the line is: *UseWindow("NOTEPAD.EXE|#32770|Notepad",1)*

Click(Button,"&Save")

The *Click* function will click the specified button for the window specified by the previous *UseWindow* call. This line will click the **Save** button on the Notepad message box.
Under XP or 2003, the line is: *Click(Button,"&Yes")*

UseWindow("NOTEPAD.EXE|FloatNotifySink|Save As|1",1)

This line selects the Filename edit control on the Save As dialog to receive keyboard and mouse input.
Under XP or 2003, the line is: *UseWindow("NOTEPAD.EXE|Edit|Save As|1",1)*

WriteCombo("1","test02.txt")

This line will type **test02.txt** into the Filename edit control on the Save As dialog.
Under XP or 2003, the line is: *SendKeys("test02.txt")*

UseWindow("NOTEPAD.EXE|#32770|Save As",1)

This line selects the Save As dialog to receive keyboard and mouse input.

Click(Button,"&Save")

This line will click the **Save** button on the Notepad Save As dialog.

Exercise 3

This exercise will demonstrate that other factors may cause an automation script to fail even after compensating for obvious failure modes.

1. Use Explorer to delete the **test02.txt** file that was saved during Exercise 2.

2. Launch Notepad using the Windows Start menu.

3. Click on the Notepad text area and type **Hello** followed by the **Enter** key.

4. From Notepad, select menu item **File/Exit**.

5. Select **Save** button (or **Yes** button under XP or 2003) when Notepad presents the **Save Changes** dialog.

6. Browse to another folder and save the file as **test02.txt**. This is the same name as used in Exercise 2.

7. Launch the WinTask Editor and click on the **Open** button Open on the WinTask Toolbar. On the **File Open** Dialog, browse to the folder containing **script02.src** and open it.

8. Click the **Play** button Play on the WinTask Toolbar. As you watch the compiled script execute, you will notice that Notepad still doesn't save the file.

9. Questions for Discussion: Why did the script fail even though the **test02.txt** file was deleted by Step 1?

Notepad like many other applications remembers the folder where the last File Open or File Save took place. Step 6 of Exercise 2 simply moved the default folder to another location. Since the script relies on Notepad's default folder; Notepad found the second **test02.txt** in the new folder causing the script to fail again.

Attempting to browse to the appropriate folder is also problematic since the PC folder hierarchy may change from one automation run to the next. Therefore, in order to create a truly reliable automation script, the full pathname of the file to be opened or saved must be typed into the File name field while recording.

In conclusion, Recording Mode is incapable of creating all of the lines in the automation script to make it reliable. Always remember that the PC must be in the same state as when the script was created.

WinTask IDE

The WinTask Editor provides full IDE capabilities to support the development of automation scripts. The user can record, edit, compile and execute scripts through the user interface of the WinTask Editor. Context sensitive Help on the WinTask Script Language and WinTask Application Programming Interface is also available when editing a script.

Exercise 4

This exercise demonstrates integration of the WinTask Editor with the WinTask Compiler, the WinTask Executor and the WinTask context sensitive Help.

1. Launch the WinTask Editor. If Your First Script Wizard screen comes up, click **Cancel** button. The title bar should display **WinTask – [Untitled1]**. If WinTask opens a previously saved script (Ex. **script02.src**), close the window by selecting menu item **File/Close** and click the **New** button on the WinTask Toolbar.

2. Click the **Record** button to start Recording Mode. Click the **OK** button on the **Starting Recording Mode** dialog box.

3. The **Launching a Program** dialog box will appear. Enter **notepad** into the **Program** field and click the **OK** button.

4. Click on the Notepad text area and type **Hello**.

5. Highlight the word **Hello**.

6. From Notepad, select menu item **Format/Font** and change the **Font Size** to **20**. Click the **OK** button to close the **Font** Dialog.

7. Click after the word **Hello** and press **Enter**.

8. On the second line, type **WinTask** and press **Enter**.

9. Highlight the word **WinTask**.

10. From Notepad, select menu item **Format/Font** and change the **Font Style** to **Bold**. Click the **OK** button to close the **Font** Dialog.

11. Click the **Stop Recording** button on the WinTask Floating Toolbar to stop Recording Mode.

12. Close notepad without saving.

13. The WinTask Editor window is restored to its normal size and the recorded automation script is displayed in the text area.

14. Review the lines of the generated script file. The script should contain two lines with the function **ChooseItem** on it. Place the cursor on the function name closest to the end of the script and press **F1** to invoke the context sensitive help for **ChooseItem**. Review the page and then close the help window.

15. Place the cursor at the beginning of the last line of the script.

16. Type the following on the blank line: **MsgBox(Typed Hello WinTask")** followed by the **ENTER** key. (Type exactly as listed.)

17. Place the cursor at the beginning of the first line top of the script. Select the **Edit/Replace** menu item. Use the **Replace** dialog box to replace the font size of **20** with **24** in the entire script. (Ex. **ChooseItem(Combo, "3", "20"** and **ChooseItem(Combo, "3", "24"** respectively).

18. Click the **Play** button to replay the script. Save the script as **script04.src** when prompted to save.

19. The WinTask Compiler will detect the error and writes several error messages to the Output window of the Editor. Review the errors and double click the first error. The cursor will be moved to the point of the error in the Edit window containing **script04.src**. Note that the WinTask Status Bar displays the line and column numbers of the cursor location at the bottom-right of the window.

20. Delete the line that causes the error.

21. Click the **Play** button to replay the script. The script should compile this time and execution starts. Close Notepad without saving.

Terminating a Script

There are times when you may want to terminate an actively running automation script. This can be accomplished by simultaneously pressing the **Ctrl+Shift+Pause** keys.

Synchronization

When the WinTask Executor replays an automation script, it often needs to wait for an event to occur after executing a script action before it can execute the next action. In the previous exercises you had to wait for Notepad to start before you could enter text into the text area. When closing Notepad, you had to wait for the Save As dialog to display before being able to save the file. Both cases illustrate synchronization between application and user. Task automation cannot be realized without synchronization between actions.

WinTask accomplishes synchronization through functions that automatically wait for events and automation script actions inserted by the user. The UseWindow function provides automatic synchronization by waiting for a window to open or become available when the next action must be done in that window. WinTask will report an error after reaching an internal timeout value if the window is unavailable.

At times the user must manually add synchronization to an automation script. For example, a CPU intensive operation may cause a timeout error under some circumstances. Another situation may be while waiting for a rather large program to load. To avoid intermittent problems, synchronization functions and script language constructs need to be added to the automation script. WinTask provides synchronization wizards that are available from the WinTask Floating Toolbar in Recording Mode and from the WinTask Editor toolbar and menus.

Synchronization Methods

WinTask supports synchronization on six system events and three user events. The system events wait for the operating system to complete a task. The user events, or Wait Triggers, wait for the user to provide input to an application. A wizard is available to generate and insert automation script snippets to handle each synchronization event. When invoking a synchronization wizard from the WinTask Editor, the generated script is inserted into the active script window at the current cursor position. If invoked during Recording Mode, the synchronization script is automatically included at that point in the recording process.

The WinTask Help System provides additional details on how to use each wizard. See Appendix A for screen shots of the WinTask Toolbar buttons that invoke each wizard. See Appendix B for screen shots of the WinTask Floating Toolbar and the buttons that invoke each synchronization wizard during Recording Mode.

- **Text Synchronization**

  This synchronization waits until a pre-defined text string appears somewhere on the desktop. This type of synchronization can be useful when working with command line applications. The Text Synchronization wizard can be accessed through the WinTask Editor **Insert/Synchronization/On Text** menu item.

- **OCR Text Synchronization**

  This synchronization waits until a pre-defined text string recognized by an OCR engine (OCR : Optical Character Recognition) somewhere on the desktop. This type of synchronization can be useful when Text Synchronization is not able to recognize text when the text is embedded in an image. The OCR Text Synchronization wizard can be accessed through the WinTask Editor **Insert/Synchronization/On OCR Text** menu item. Two OCR engines are supported, an internal WinTask one, and the one from Microsoft Office 2003 or 2007.

- **Bitmap Synchronization**

  This synchronization waits until a pre-defined image appears somewhere on the desktop. One area where this type of synchronization may be useful is waiting for an icon to load. The Image Synchronization wizard can be accessed through the WinTask Editor **Insert/Synchronization/On Image** menu item.

- **Window Synchronization**

  This synchronization waits until a new window is displayed or becomes active. A user may use this type of synchronization to access a window or dialog that is opened in response to the selection of a menu item. The Window Synchronization wizard can be accessed through the WinTask Editor **Insert/Synchronization/On Window** menu item.

- **Time Synchronization**

  This synchronization waits until a specified amount of time has expired. The time interval can be set from a tenth of a second to several days. Time synchronization can be used to wait for a program to load. The Time Delay Synchronization wizard can be accessed through the WinTask Editor **Insert/Synchronization/On Time Delay** menu item.

- **Date/Hour Synchronization**

  This synchronization waits until a specified date and time. A user may wish to use this type of synchronization to perform daily back-ups. The Date/Hour Synchronization wizard can be accessed through the WinTask Editor **Insert/Synchronization/On Date/Hour** menu item.

- **Keyboard Wait**

  This synchronization waits until the user types the specified key. This type of synchronization can be used to check the intermediate progress of an automated task, or to wait for the user to change a removable storage media. The Wait for Keyboard Action Synchronization wizard can be accessed through the WinTask Editor **Insert/Wait for/Key** menu item.

- **Menu Wait**

  This synchronization waits until the user selects a specified menu item. This type of synchronization can be used to make some manual adjustments before selecting the menu item that continues the automated task. The Wait for Menu Synchronization wizard can be accessed through the WinTask Editor **Insert/Wait for/Menu** menu item.

- **Mouse Wait**

  This synchronization waits until the user clicks or double clicks the mouse. This type of synchronization can be used to check the intermediate progress of an

automated task, or to wait for the user to change a removable storage media. The Wait for Mouse Action Synchronization wizard can be accessed through the WinTask Editor **Insert/Wait for/Mouse** menu item.

Exercise 5

This exercise demonstrates how to invoke and use the OCR Text Synchronization wizard while recording an automation script.

1. Launch the WinTask Editor. If Your First Script Wizard screen comes up, click **Cancel** button. The title bar should display **WinTask – [Untitled1]**. If WinTask opens a previously saved script, close the window by selecting menu item **File/Close** and click the **New** button on the WinTask Toolbar.

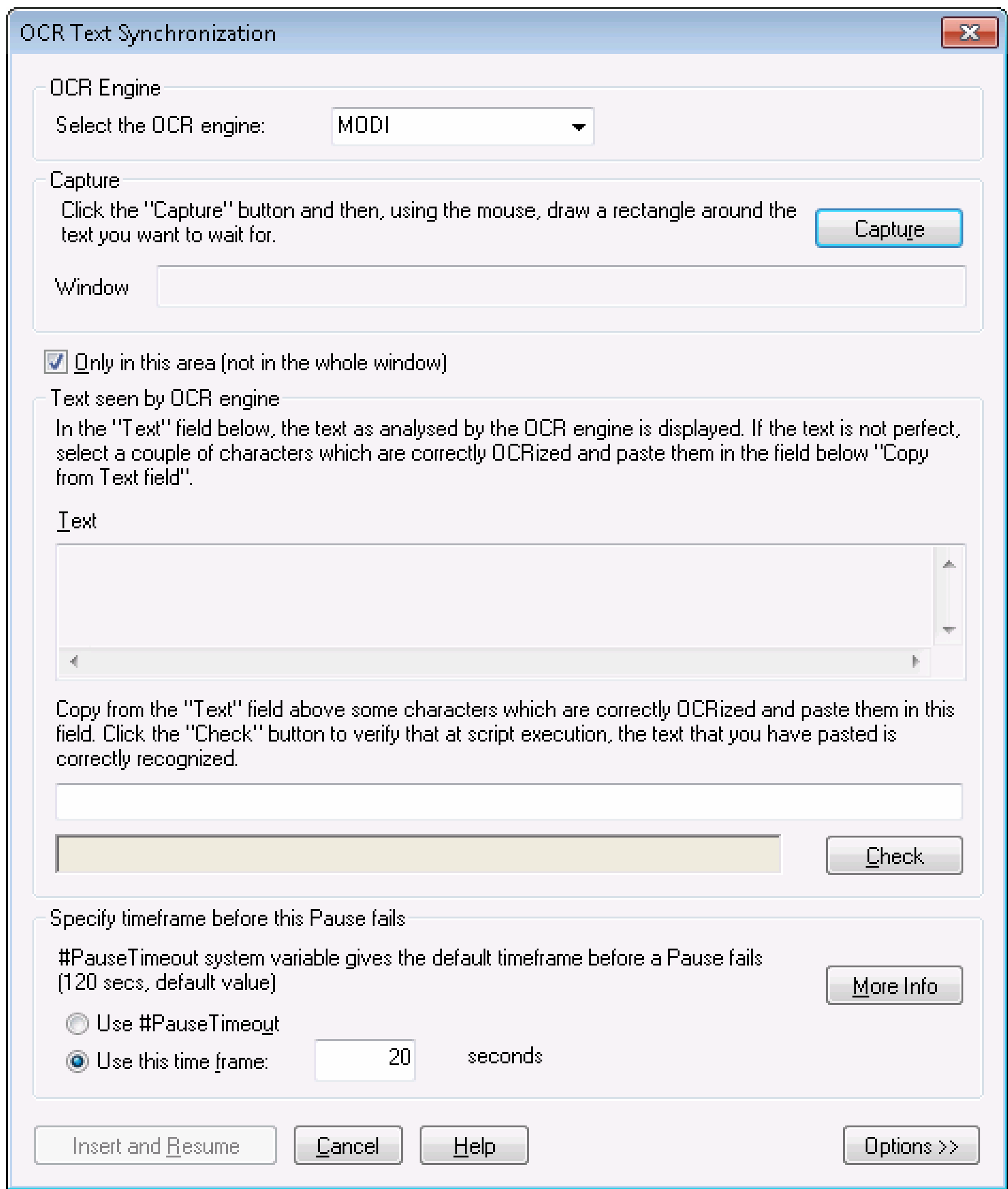
2. Click the **Record** button to start Recording Mode. Check the **Internet Explorer** button on the **Starting Recording Mode** dialog box. Enter **www.wintask.com/manuals.php** into the **Web address** field and click the **OK** button.

3. The page titled **WinTask Manuals** is loaded, click **Tutorial** link.

4. The manual takes some time to load, insert an OCR Text Synchronization to check that the manual is loaded.

5. Invoke the OCR Text Synchronization wizard by clicking the OCR icon on the WinTask toolbar and select OCR Text Synchronization .

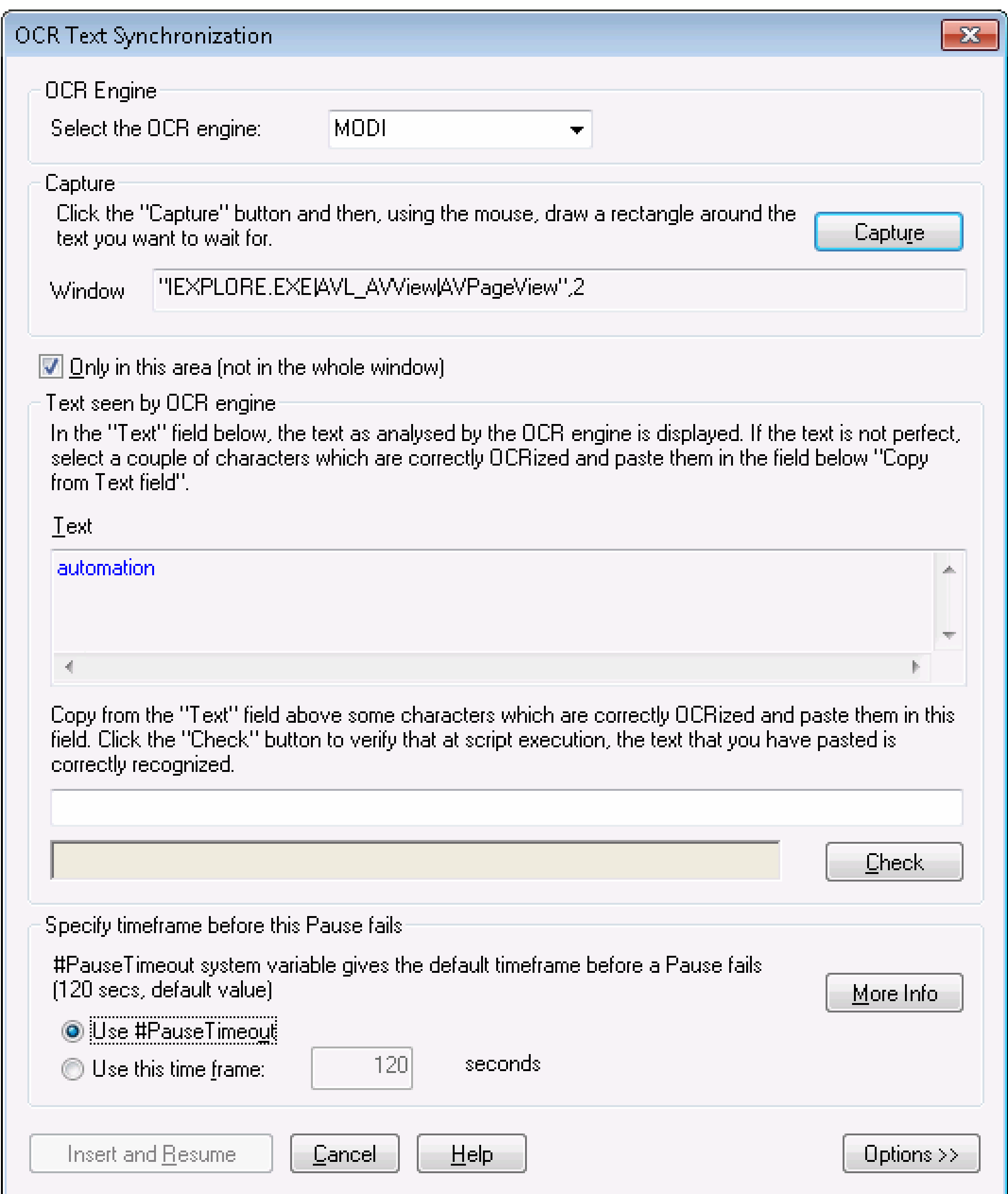
6. The OCR Text Synchronization wizard is displayed.



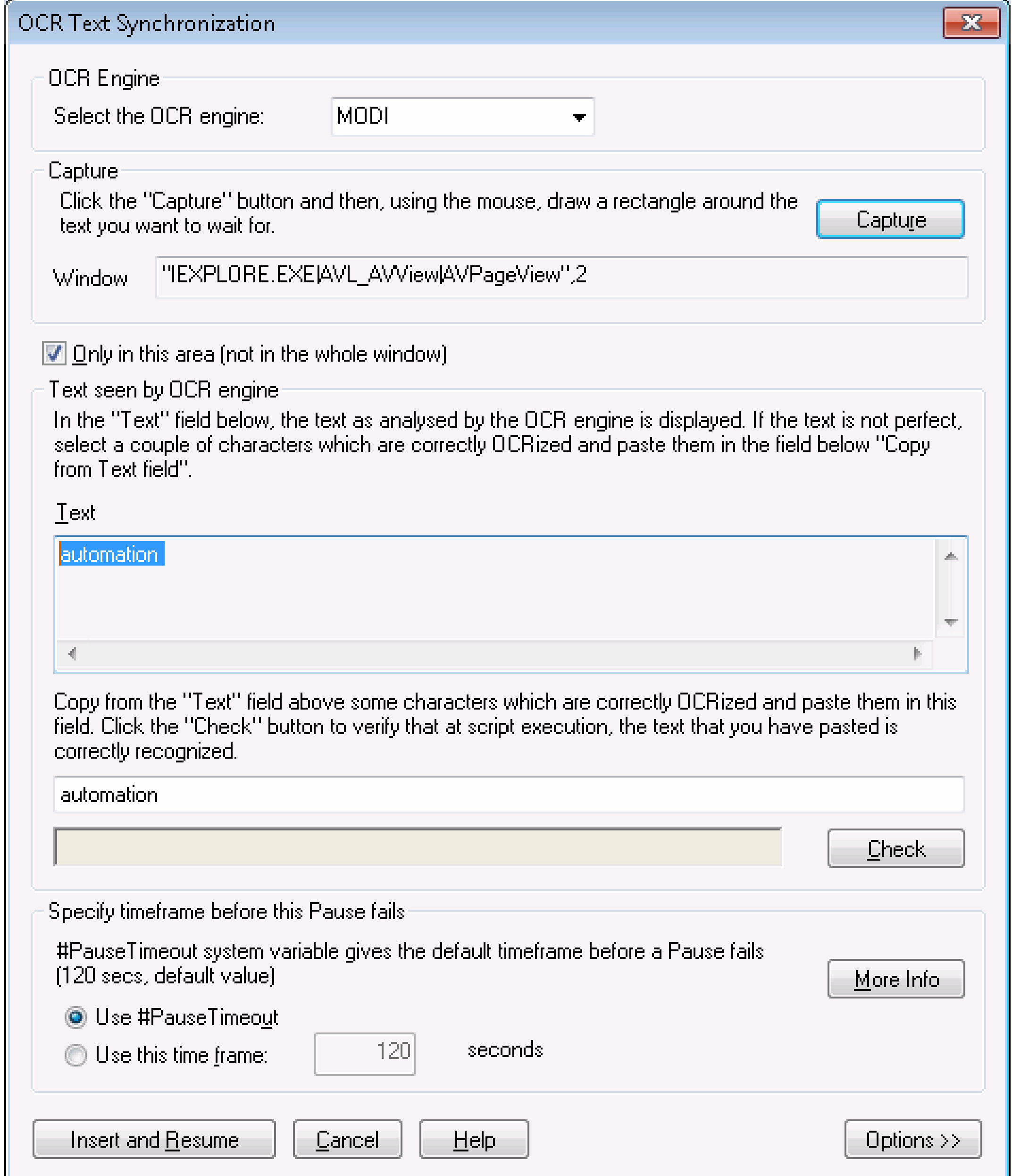
7. Click the **Capture** button on the OCR Text Synchronization wizard. The wizard will be hidden and the cursor will change to a crosshair.

8. Use a drag box to capture the text **automation** in the pdf window.

9. The OCR Text Synchronization screen comes back to focus with automation word filling the Text field.



10. Copy the automation word and paste it in the field below the Text field:



11. Click **Check** button to check that the OCR engine recognizes correctly the word.

12. Click **Insert and Resume** button to resume Recording mode.

13. Click the **Stop Recording** button on the WinTask Floating Toolbar to stop Recording Mode.

The OCR Text Synchronization wizard inserted the following lines of script into the recorded automation script:

```
Pause Until
    TextOCR("automation")
    InWindow("IEXPLORE.EXE|AVL_AVView|AVPageView",2)
    InArea(330,509,45,130)
PauseFalse
    MsgBox("'Wait for' at line " + #ErrorLine$ + " has failed!")
    End
EndPause
```

IMPORTANT: under IE9, change the 2 number in line
InWindow("IEXPLORE.EXE|AVL_AVView|AVPageView",2)
to a 1 as:
InWindow("IEXPLORE.EXE|AVL_AVView|AVPageView",1)

The preceding lines of script instruct WinTask to pause execution of the automation script until the text string *automation* appears in the pdf window. Once the text appears, execution of the script continues following the *EndPause* statement. WinTask will wait up to 120 seconds for the text to appear in the pdf window before executing the error handling script between the *PauseFalse* and *EndPause* statements. The user may wish to override the default wait value of 120 seconds to 10 seconds by modifying the first line of the script slightly as shown below. Also note that the error message presented to the user has been changed from a generic automation script error to a system specific error.

```
Pause 10 sec Until
    TextOCR("automation")
    InWindow("IEXPLORE.EXE|AVL_AVView|AVPageView",1)
    InArea(330,509,45,130)
PauseFalse
    MsgBox("Manual has not been loaded!")
    End
EndPause
```

14. Add a new line after EndPause line and type:

    CloseBrowser()

15. Close manually the Internet Explorer window and then Play the script. Give the name script06 when prompted.

If no text can be used for synchronization, you should switch to Image Synchronization as outlined in the next exercise.

Exercise 6

This exercise demonstrates how to invoke and use the Image and Time Synchronization wizards while recording an automation script.

1. Launch the WinTask Editor. If Your First Script Wizard screen comes up, click **Cancel** button. The title bar should display **WinTask – [Untitled1]**. If WinTask opens a previously saved script, close the window by selecting menu item **File/Close** and click the **New** button New on the WinTask Toolbar..
2. Click the **Record** button to start Recording Mode. Check the **Internet Explorer** button on the **Starting Recording Mode** dialog box. Enter

www.wintask.com/manuals.php into the **Web address** field and click the **OK** button.

3. The page titled **WinTask Manuals** is loaded, click **Tutorial** link.

4. Invoke the Image Synchronization wizard by clicking the **Image Synchronization** button on the WinTask Floating Toolbar. Recording Mode will be paused.

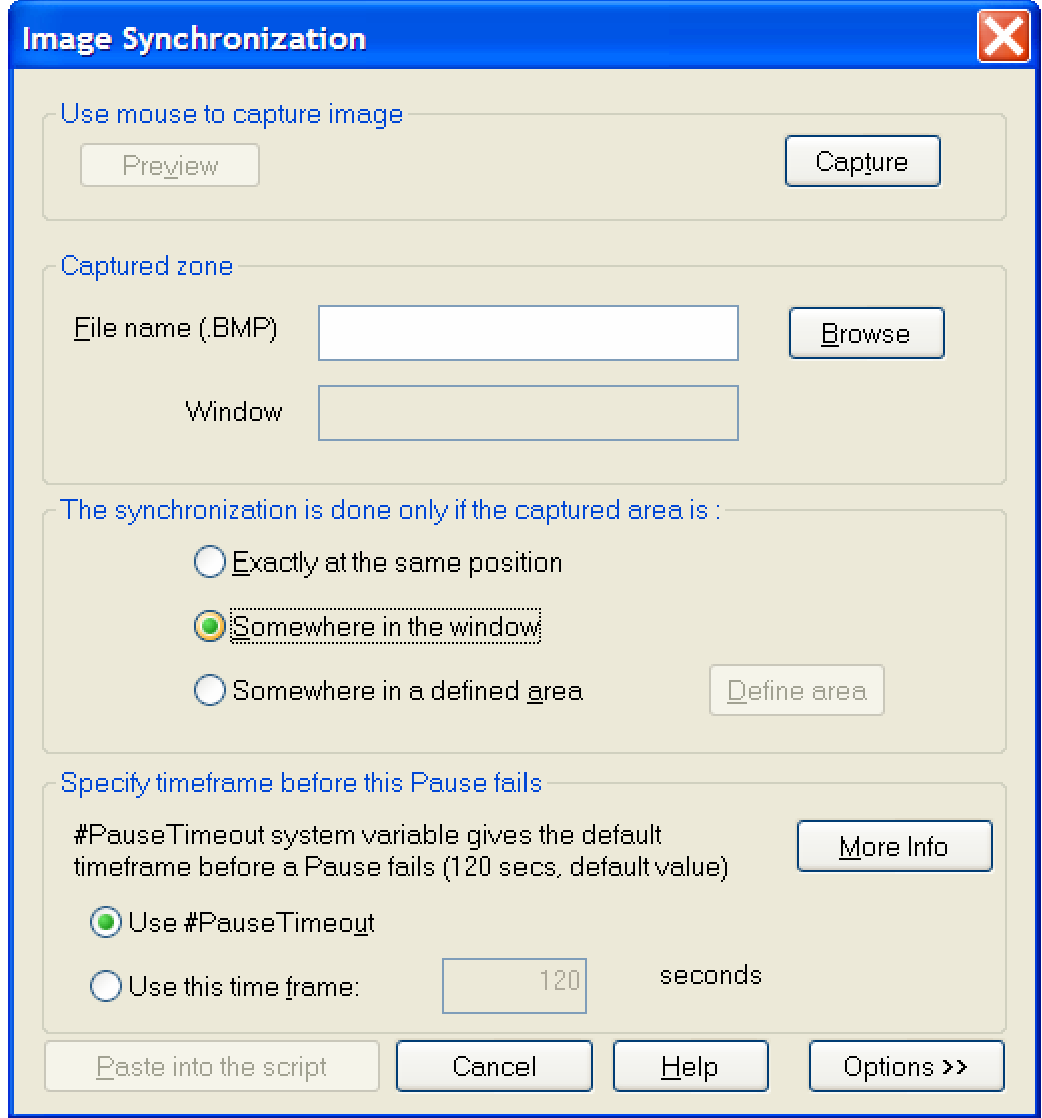


Figure 2. Image Synchronization Wizard.

4. Click the **Somewhere in the Window** radio button on the Image Synchronization wizard.

5. Use a drag box to capture the text **automation** in the pdf window. A preview window will appear displaying the captured image. Close the preview window.

6. The Image Synchronization wizard will reappear. Click on the **Browse** button and use the **Save As** dialog to save the captured bitmap as **Image06.bmp**.

7. Click the **Insert and Resume** button on the Image Synchronization wizard. Recording Mode will become active again.

8. Close the pdf window.

9. Click the **Stop Recording** button on the WinTask Floating Toolbar to stop Recording Mode.

10. Save the script as **script06.src** when the WinTask Editor window is restored.

The Image Synchronization wizard inserted the following lines of script into the recorded automation script:

```
Pause Until
    Bitmap("C:\Program Files(x86)\WinTask\Scripts\Image06.bmp")
    InWindow("IEXPLORE.EXE|AVL_AVView|AVPageView",2)
    InArea( 334, 516, 40, 127 )
PauseFalse
    MsgBox("'Wait for' at line " + #ErrorLine$ + " has failed!",0,"Runtime error")
    End
EndPause
```

Under IE9, change the 2 value in 1 as:

```
InWindow("IEXPLORE.EXE|AVL_AVView|AVPageView",1)
```

Click Play icon to run the script.

This section has described the WinTask Synchronization wizards and presented how to use two of them. The remaining wizards are similar in use and should prove to be intuitive to the user familiar with the other three wizards.

WinTask Scripting Language

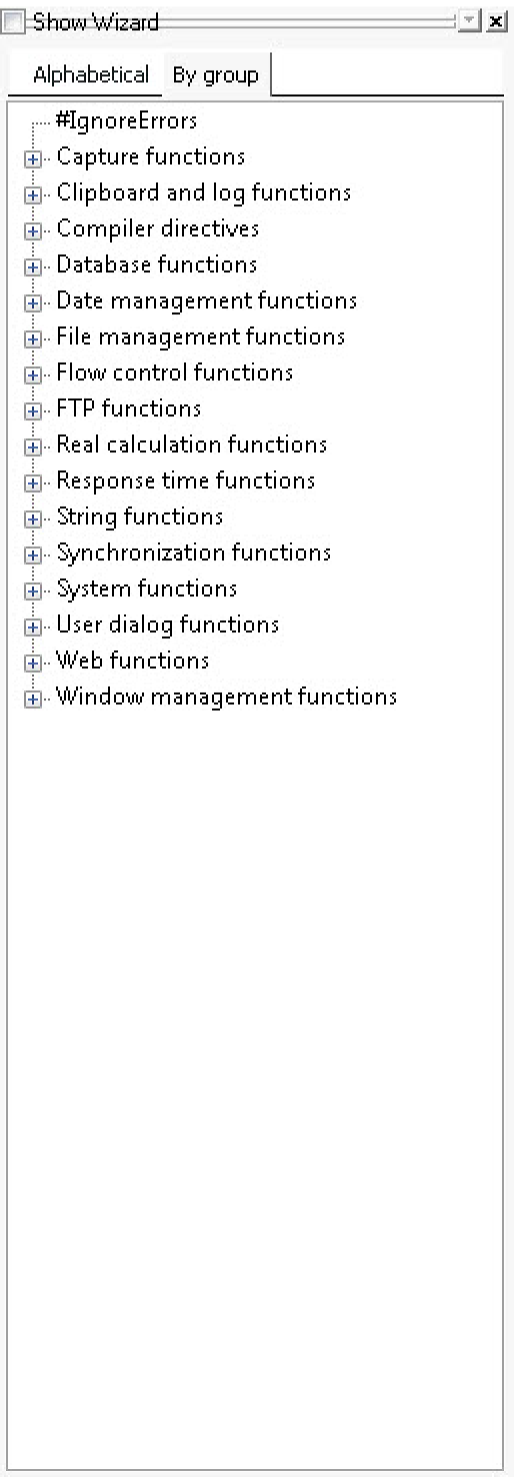
The WinTask Scripting Language has a program structure similar to Microsoft Visual Basic™ and other scripting languages. WinTask automation scripts must be structured correctly to avoid compiler errors. WinTask scripts are composed of three distinct sections. The first section is reserved for data arrays, the second for user-defined functions and subroutines, and the final section for the main program. Other features of the script language include variables, integers and strings.

Automation scripts generated during Recording Mode fall into the category of main program script. The user can wrap repetitive blocks of script in functions or subroutines that can then be called from other function/subroutines or the main program.

Functions

Functions support one or more input parameters and a single return value. Both are optional. If a function has input parameters, they follow the function name as a comma separated list enclosed in parenthesis. Constant strings passed to functions must be enclosed in double quotes (Ex: "qwerty"). Functions that return a string must have a function name which ends with **$**. Functions that return an integer CANNOT have a function name which ends with **$**.

The Insert Statement dialog can be invoked by clicking the **Language List** button Language on the WinTask Toolbar. The Insert Statement dialog presents all of the predefined functions and system variables available in the WinTask Scripting Language in a tree structure. Clicking an item in the list labeled with a plus sign will expand that item. Expanded items; designated by a minus sign, can be collapsed by clicking the item. Double clicking any other item will invoke the context sensitive help for the selected function or system variable.



The Insert Statement dialog provides an easy way to drill down into the help system to locate the function(s) that can fine-tune your script. Please take a few minutes to search the WinTask Help System to learn more about the available functions.

Variables

The WinTask Scripting Language supports variables that allow commonly used values to be stored in memory for later use by the compiled script as it executes. Using variables is as easy as specifying the name of the variable and assigning a value. Except for arrays, variables do not need to be declared before use. Consider the following line of script:

Read("file.txt", line$, crlf)

The *Read* function will open the file *file.txt* and reads the first line of the file into a string variable named *line$*. The script can then extract the desired information from the string contained in the *line$* variable. As with Functions that return a string, string variable names must end with the **$** character. It should be noted that this line of script is self-sufficient and is not dependent on any other line of script.

Another useful feature of string variables is to reduce the amount of typing required to develop an automation script. Using variables to store a long string also reduces the possibility of run-time errors caused by mistyping or confusing similar values. Please review the following lines of script.

```
compta$ = Chr$(34) + "C:\Program Files (x86)\visicompt\visicompt.exe" +
Chr$(34)
Shell(compta$)

Shell(Chr$(34) + "C:\Program Files (x86)\visicompt\visicompt.exe" + Chr$(34))
```

The first two lines are equivalent to the third line of script. At first it seems easier to just use the third line of script instead of the first two lines. However if the script needs to make numerous references to the *visicompt.exe* program, the first two lines are the better choice. Succeeding references to *visicompt.exe* can be replaced with the string variable *compta$*.

System Variables

WinTask uses a set of system variables that define the default behavior of the Executor when executing an automation script. If a particular automation task requires a longer time-out value, the script can compensate by modifying the appropriate system variable to modify the default behavior to wait for a longer period of time before reporting an error.

All system variables start with the **#** character. See the WinTask Help System for further details on the available system variables and the appropriate values to modify the default behavior of the Executor. The following is an example of a system variable:

```
#IgnoreErrors = 1
```

Integers

Integer values in the WinTask Scripting Language range from –2,147,483,648 to +2,147,483,647 inclusive. Integer variable names cannot end with the **$** character.

Strings

Strings can contain any number of characters. Empty strings (zero length) are also valid, an empty string is "". A string containing a space is " ". String variable names must end with the **$** character. The following is an example of a string being assigned to a string variable:

Greeting$ = "Hello, how are you?"

Reals

Real values are not supported in WinTask. They have to be stored as strings and **Add$**, **Divide$**, **Multiply$**, **Subtract$** functions allow operations on reals. Here is an example:

```
Real1$ = "135.46"
Real2$ = "-10.46"
Result$=Add$(Real1$,Real2$)
Msgbox(Result$)
```

Arrays

One dimension arrays are supported in WinTask. They have to be declared at the very beginning of the script using the **Dim** function. The maximum size of an array is 65535. The first element of an array starts at index 0. Here is an example declaring an integer array of 1001 elements:

Dim Data(1000)

Here is an example declaring a string array of 1001 elements:

Dim Data$(1000)

Operators

The WinTask Scripting Language supports assignment, arithmetic operators, logical operators and a string concatenation operator. Operators take an argument on either side and can be combined into longer combinations. Logical operators are used in conjunction with conditional statements such as **If**, **While** and **Repeat**. Most of the operators are listed below with sample usage. See the WinTask Help System for a comprehensive list.

= Assignment: Assign one value to a variable

count = 38
name$ = "John Q. Public"

+ Arithmetic Addition

rows = lines + 7

- Arithmetic Subtraction

Profit = Gross - Expenses

* Arithmetic Multiplication

Fingers = Hands * 5

/ Arithmetic Division

Hours = Days / 24

= Logical Equal To

If (employees = 15) Then … Endif

<> Logical Not Equal To

If (employees <> 15) Then … Endif

< Logical Less Than

While (employees < 15) … Wend

<= Logical Less Than or Equal To

If (employees <= 15) Then … Endif

> Logical Greater Than

Repeat … Until (employees > 15)

>= Logical Greater Than or Equal To

If (employees >= 15) Then … Endif

AND Logical AND

If (employees = 15) AND (desks < 10) Then … Endif

OR Logical OR

If (employees <> desks) OR (employees <> computers) Then … Endif

+ String Concatenation

FullName$ = FirstName$ + LastName$

Title$ = "President and CEO " + FullName$

Window Management Functions

The WinTask Script Language contains several mechanisms to control the behavior of the WinTask Executor when attempting to locate and identify windows on the desktop at run-time. Several of the more important system variables and functions are described in this chapter. Additional window management functions can be found in the WinTask Help System.

#IgnoreErrors

The **#IgnoreErrors** system variable controls how run-time errors are handled during the execution of a function in an automation script. By default the variable is set to 0 which terminates the script and displays an error message whenever the WinTask Executor encounters an error. Setting the variable equal to 1 allows execution to continue if the function fails. In this circumstance, the function returns an error code that can be tested by the script and the appropriate action taken.

The following block of code illustrates how the **#IgnoreErrors** system variable may be used:

```
#IgnoreErrors = 1

ret = Shell("notepad.exe")

If (ret = 0) Then
    MsgBox("Notepad has been launched successfully")
Else
    MsgBox("Could not start Notepad!")
Endif
```

The above code will display a message after attempting launch to **Notepad**. The message that is displayed depends upon the value returned by the **Shell** function.

#ActionTimeout

The **#ActionTimeout** system variable controls how long the WinTask Executor will wait before reporting a run-time error when selecting a window, control, or menu item. The variable can be used to shorten or lengthen the amount of time waited before reporting an error. The default value is 30 seconds.

The following block of code illustrates how the **#ActionTimeout** system variable may be used:

```
#ActionTimeout = 5
#IgnoreErrors = 1

ret = Shell("notepad.exe")

If (ret = 0) Then
    MsgBox("Notepad has been launched successfully")
Else
    MsgBox("Buy a faster computer!")
Endif
```

The above code will display a message after attempting launch to **Notepad**. The message that is displayed depends upon the value returned by the **Shell** function. The second message will be displayed if **Notepad** requires more than 5 seconds to launch.

#UseExact

The **#UseExact** system variable plays an important role in helping WinTask select the correct window when the window title changes, or multiple instances of the same application or window are on the Windows desktop. Review the following script statements:

```
UseWindow("NOTEPAD.EXE|Notepad|Untitled - Notepad")

UseWindow("NOTEPAD.EXE|Notepad|test.txt - Notepad")
```

Both statements for search an instance of **Notepad**. The first statement represents the window title when a new document is being created. The second statement represents the window title after saving the file as **test.txt**.

By default the WinTask Executor looks for an exact match of the window title. If an exact match is not found, WinTask attempts to match an approximation of the specified title. If both attempts fail to find a match, an error will be reported.

By setting the **#UseExact** system variable equal to zero, the default behavior described above will be in effect. Setting **#UseExact** equal to one will change the behavior of WinTask to only accept an exact match of the specified window title.

TIP: WinTask will attempt to exactly match the window title for about 3 seconds (1/9 of the **#ActionTimeout** system variable default of 30 seconds). If the title of a window changes during the execution of the script, numerous delays may be encountered until a match is accomplished. To avoid unnecessary delays, shorten the specified window title to increase the probability of an exact match.

TIP: When numerous windows with similar titles are present on the desktop, WinTask may potentially synchronize on the same window each time leading to unpredictable results. The **#UseExact** system should be set to 1 in this case and full window titles should be specified.

ExistW()

The **ExistW** function allows the script developer to determine whether a particular window is present on the desktop at run-time. The function returns a value that can

be used in conjunction with conditional statements. Use this function to avoid run-time errors due to windows that can't be located.

The following block of code illustrates how the **ExistW** function may be used:

```
ret = ExistW("NOTEPAD.EXE|#32770|Save As",1)

If (ret = 1) Then
    UseWindow("NOTEPAD.EXE|#32770|Save As",1)
        Click(Button,"&Save")
Else
    MsgBox("Notepad Save As dialog not present!")
Endif
```

Focus$()

The **Focus$** function allows the script developer to determine the title of the window which has focus on the desktop at run-time. In the case of an application with several child windows, **Focus$** returns the title of the child window which has focus.

The following block of code illustrates how the **Focus$** function may be used:

```
Window_title$ = Focus$()

If (Window_title$ = "NOTEPAD.EXE|Edit|Untitled – Notepad|1") Then
    UseWindow("NOTEPAD.EXE|#32770|Save As",1)
        Click(Button,"&Save")
Else
    MsgBox("Notepad Save As dialog does not have focus!")
Endif
```

Top$()

The **Top$** function allows the script developer to determine the title of the active window on the desktop at run-time. This function is similar to the **Focus$** function with the exception that it returns the title of the parent window if one of its child windows has focus.

The following block of code illustrates how the **Top$** function may be used:

```
Window_title$ = Top$()

If (Window_title$ = "NOTEPAD.EXE|Notepad|Untitled – Notepad") Then
    UseWindow("NOTEPAD.EXE|#32770|Save As",1)
        Click(Button,"&Save")
Else
    MsgBox("Notepad is not the active application!")
Endif
```

The following block of code illustrates how the **Top$** function may be used to distinguish between two instances of **Notepad**:

```
Shell("notepad.exe file01.txt")
Shell("notepad.exe file02.txt")

Window_title$ = Top$()

If (Window_title$ = "NOTEPAD.EXE|Notepad|file02.txt – Notepad") Then
    UseWindow("NOTEPAD.EXE|Edit|Save As|1",1)
        SendKeys("<Backspace>Updated file02.txt <Enter>")
    UseWindow("NOTEPAD.EXE|#32770|Save As",1)
        Click(Button,"&Save")
Else
    UseWindow("NOTEPAD.EXE|Edit|Save As|1",1)
        SendKeys("<Backspace>Updated file01.txt <Enter>")
    UseWindow("NOTEPAD.EXE|#32770|Save As",1)
        Click(Button,"&Save")
Endif
```

Exercise 7

This programming exercise demonstrates how a script can be modified to test for the existence of a window and how that information may be put to use. You will be asked to modify the script based upon the information presented in this chapter and the chapter titled **WinTask Scripting Language**. See the WinTask Help System for additional information to complete this exercise. Appendix C contains the solution for this exercise.

1. Launch the WinTask Editor. The title bar should display **WinTask – [Untitled1]**. If WinTask opens a previously saved script (Ex. **script02.src**), close the window and click the **New** button on the WinTask Toolbar.

2. Click the **Record** button to start Recording Mode. Click the **OK** button on the **Starting Recording Mode** dialog box.

3. The **Launching a Program** dialog box will appear. Enter **Notepad** into the **Program** field and click the **OK** button.

4. Click on the Notepad text area and type **Hello** followed by the **Enter** key.

5. From Notepad, select menu item **Edit/Time/Date**.

6. From Notepad, select menu item **File/Exit** to close Notepad.

7. Select **Save** button (or **Yes** button under XP or 2003) when Notepad presents the **Save Changes** dialog.

8. Save the file as **test07.txt** in the current folder

9. Click the **Stop Recording** button on the WinTask Floating Toolbar to stop Recording Mode.

10. In the WinTask Editor, modify the script to replace the save file name ("*test07.txt*") with a string variable named *filename$*. Define the variable at the beginning of the script and assign the string "*test07.txt*" to the variable.

11. Click the **Play** button on the WinTask Toolbar to replay the actions listed in the script. Enter the name **script07a.src** when the **Save As** dialog is displayed. Correct any compilation errors if the Output window displays errors and run again.

12. The script will fail to run to completion since the target file name already exists.

13. Modify the script to test for the **Confirm Save As** message box (or **Save As Overwrite** message box under XP or 2003). If the window exists, add code to click the **Yes** button. See the WinTask Help System for usage of the *ExistW* function and the *If* statement.

14. Recompile and test the modified script. Save the updated script as **script07b.src**.

15. Now remove the call for *ExistW*, set the *#IgnoreErrors* system variable to *1*, and test the return value of the *UseWindow* call to test for the existence of the **Confirm Save As** message box (or **Save As Overwrite** message box under XP or 2003). The *If* construct will need some slight modification. See the WinTask Help System for usage of the *UseWindow* function and the *#IgnoreErrors* system variable.

16. Recompile and test the modified script. Save the updated script as **script07c.src**.

17. Now change the name of the text file to"*test07d.txt*".

18. Recompile and test the modified script. Save the updated script as **script07d.src**.

19. Questions for Discussion: What are the strengths and weaknesses of the techniques presented in Steps 13 and 15 to test for the existence of a window? How is the behavior different when the file exists (Steps 13, 15) compared to when the file does not exist (Step 17)?

This exercise illustrates that there is usually more than one way to solve a runtime problem. Any task that is to be automated will present its own unique challenges. With some experience, the correct solution will become readily apparent.

Iteration

The WinTask Scripting Language provides two mechanisms to repetitively execute the same block of script. The While/Wend construct is used to execute the block of script zero or more times. The Repeat/Until construct executes the block one or more times.

Iteration

The following sample script illustrates the syntax for the **While/Wend** and the **Repeat/Until** statements.

```
Start = 0
End = 10
Index = Start
While (Index < End)
    Functions and/or other code to be executed repetitively
    Index = Index + 1
Wend
```

```
Start = 0
End = 10
Index = Start
Repeat
    Functions and/or other code to be executed repetitively
    Index = Index + 1
Until (Index >= End)
```

In both cases, the start and end points for the loops are set to zero and ten respectively. The code in both constructs will be executed ten times. Notice that the Boolean expressions are different in the two constructs yet they achieve the same result.

Exercise 8

This programming exercise demonstrates how a script can be modified to repeat a block of script several times. You will be asked to modify the script based upon the information presented in this chapter. See the WinTask Help System for additional information to complete this exercise. Appendix C contains the solution for this exercise.

1. Launch the WinTask Editor. The title bar should display **WinTask – [Untitled1]**. If WinTask opens a previously saved script (Ex. **script07c.src**), close the window and click the **New** button on the WinTask Toolbar.

2. Click the **Record** button to start Recording Mode. Click the **OK** button on the **Starting Recording Mode** dialog box.

3. The **Launching a Program** dialog box will appear. Enter **Notepad** into the **Program** field and click the **OK** button.

4. Click on the Notepad text area and type **Hello 1** followed by the **Enter** key.

5. Type **Hello 2** followed by the **Enter** key on the next line.

6. Click the **Stop Recording** button on the WinTask Floating Toolbar to stop Recording Mode.

7. Close Notepad without saving.

8. In the WinTask Editor, modify the script to use the *While/Wend* construct to enter following 10 lines of text into the Notepad text area:

   Hello 1
   Hello 2
   ...
   Hello 10

TIP: Use the string concatenation operator (+) to construct the string to be written into the Notepad text area.

9. The script should use a string variable for the word "*Hello* " and an integer variable to count the number of lines entered.

TIP: Use the *Str$* function to convert the line number into a string. Concatenate the converted number with the string variable.

10. Change the string variable to type "*Notepad Text Line*" instead of "*Hello*". The first line of text written into the Notepad text area should read "*Notepad Text Line 1*".

11. Click the **Play** button on the WinTask Toolbar to replay the actions listed in the script. Enter the name **script08a.src** when the **Save As** dialog is displayed. Correct any compilation errors if the Output window displays errors and run again.

12. Modify the script to use the *Repeat/Until* construct instead of the *While/Wend* construct.

13. Recompile and test the modified script. Save the updated script as **script08b.src**.

After completing this exercise, you will be able update a script to execute repetitive tasks without copying the same block of code numerous times.

File Functions

The WinTask Script Language provides several functions that perform basic file input/output at run-time. Several of the more important file functions are described in this chapter. Additional file functions and specific details on the functions described in this chapter can be found in the WinTask Help System.

Exist()

The **Exist** function is used to determine whether a particular file exists on the computer at run-time. The function accepts a full pathname, a UNC (Universal Naming Convention), or a file from the current working directory. The function returns a value that can be used in conjunction with conditional statements. Use this function to avoid run-time errors due to files that can't be located.

The following block of code illustrates how the **Exist** function may be used:

```
FileName$ = "C:\Program Files (x86)\WinTask\Help\WinTask.chm"
ret = Exist(FileName$)
If (ret = 1) Then
    Shell("hh.exe " + Chr$(34) + FileName$ + Chr$(34))
Else
    MsgBox("Cannot find file: " + FileName$)
Endif
```

TIP: The filename passed to the **Exist** function should not be surrounded by double quotes (*Chr$(34))* as required by the Shell function. The quotes should not be supplied even if the filename contains embedded spaces.

Kill()

The **Kill** function is used to permanently remove a file from the computer at run-time. The function accepts a full pathname, a UNC, or a file from the current working directory. The function returns a value that can be used in conjunction with conditional statements.

The following block of code illustrates the **Kill** function:

```
FileName$ = "C:\Temp\SomeUselessFile.txt"
ret = Kill(FileName$)
If (ret = 0) Then
    MsgBox("File: " + FileName$ + " has been deleted")
Else
    MsgBox("Cannot delete file: " + FileName$)
Endif
```

TIP: The filename passed to the **Kill** function should not be surrounded by double quotes (*Chr$(34)*) as required by the Shell function. The quotes should not be supplied even if the filename contains embedded spaces.

Read()

The **Read** function is used to read the contents of a file from the computer's file system. The function accepts a full pathname, a UNC, or a file from the current working directory. The function optionally returns a value that can be used in conjunction with conditional statements.

The **Read** function will typically be called several times in order to read the entire contents of the target file. The first call to the function will read up to 32k bytes from the file and copy them into the specified buffer. Succeeding calls to the function will continue from where the last read left off, and reads the next block of data until the file contents are exhausted. The **Eof** function (described later in this chapter) should be called between calls of the **Read** function to see if the file has been read completely.

The **Read** function has four basic forms:

ret = Read(FileName$, DataBuffer$)

This form of the function will read up to 32K bytes of data from the file ***FileName$*** and copies them into the string variable ***DataBuffer$***. If the file is smaller than 32K, then ***DataBuffer$*** will contain the entire file contents.

ret = Read(FileName$, DataBuffer$, BytesToRead)

This form of the function will read up to the number of bytes of data from the file ***FileName$*** as specified by the ***BytesToRead*** parameter and copies them into the string variable ***DataBuffer$***. If the file is smaller than the specified number of bytes, then ***DataBuffer$*** will contain the entire file contents. The number of bytes to read must be less than or equal to 32K bytes.

ret = Read(FileName$, DataBuffer$, CRLF)

This form of the function reads one line of data from the file ***FileName$*** and copies it into the string variable ***DataBuffer$***. A line of data is terminated by a CR-LF (Carriage Return-Line Feed) pair. If the CR-LF pair is not found, then ***DataBuffer$*** will contain up to 32K bytes, or the entire file contents if less than 32K.

ret = Read(FileName$, DataBuffer$, Separator$)

This form of the function reads data from the file ***FileName$*** and copies it into the string variable ***DataBuffer$*** until it matches the string specified by the ***Separator$*** parameter. If the specified separator string is not found, then ***DataBuffer$*** will contain up to 32K bytes, or the entire file contents if less than 32K.

TIP: Advanced users can use the **SetReadPos** function to reposition the file read pointer to take advantage of the files with a known structure or format. See the WinTask Help System for additional information.

Eof()

The **Eof** function is used in conjunction with the **Read** function to determine if the entire contents of the target file have been read. The function accepts a full pathname, a UNC, or a file from the current working directory. Use this function to avoid run-time errors due to attempts to read beyond the end of a file.

The following block of code illustrates the **Eof** function:

```
FileName$ = "C:\Temp\SomeUselessFile.txt"

Read(FileName$, DataBuffer$)

ret = Eof(FileName$)

If (ret = 1) Then
    MsgBox("Entire contents of file have been read.  Processing data.")
Else
    MsgBox("File larger than 32K bytes!  Need to call Read() again!")
Endif
```

TIP: Test the return value of the **Eof** function as part of a While/Wend or Repeat/Until construct with the **Read** function inside the loop to safely read the contents of an entire file without run-time errors.

Write()

The **Write** function is used to write data to a file on the computer's file system. The function accepts a full pathname, a UNC, or a file from the current working directory. The function optionally returns a value that can be used in conjunction with conditional statements.

The **Write** function can be called as many times as necessary to populate the contents of the target file. If the file does not exist when the function is called, it will be created and the specified data written to the file. If the file does exist, the specified data will be appended to the file.

The **Write** function has four basic forms:

ret = Write(FileName$, DataBuffer$)

This form of the function will write up to 32K bytes of data from the string variable ***DataBuffer$*** into the file specified by ***FileName$***. The entire contents of the string variable ***DataBuffer$*** is written to the file if it's length is less than 32K bytes.

ret = Write(FileName$, DataBuffer$, BytesToWrite)

This form of the function will write up to ***BytesToWrite*** bytes from the string variable ***DataBuffer$*** to the specified file by ***FileName$***. If the string variable ***DataBuffer$*** is less than ***BytesToWrite*** bytes in length, the contents of the string variable is written to the file followed by Space characters to total ***BytesToWrite*** bytes.

ret = Write(FileName$, DataBuffer$, CRLF)

This form of the function writes one line of data to the file ***FileName$***. Up to 32K bytes of data from the string variable ***DataBuffer$*** are written to the file followed by

a CR-LF (Carriage Return-Line Feed) pair. The entire contents of the string variable ***DataBuffer$*** is written to the file if it's length is less than 32K bytes.

ret = Write(FileName$, DataBuffer$, Separator$)

This form of the function writes to 32K bytes of data from the string variable ***DataBuffer$*** to the specified file by ***FileName$***. The contents of the string variable ***DataBuffer$*** is followed by the string specified by the ***Separator$*** parameter.

Exercise 9

This programming exercise demonstrates how a script can be used to read the contents of a file and use the information to perform some other task. This exercise assumes that Exercise 8 has been completed. If not, please refer to the chapter titled **Iteration**. See the WinTask Help System for additional information to complete this exercise. Appendix C contains the solution for this exercise.

1. Launch the WinTask Editor. If the **script08b.src** script file from the previous exercise does not open, click the **Open** button on the WinTask Toolbar and open either the **script08a.src** or **script08b.src** script file.

2. Click the **Play** button on the WinTask Toolbar to replay the actions listed in the script. Click the **Run** button to execute the compiled script.

3. When the script completes, close Notepad and save the file as **test09.txt** in the folder c:\program files (x86)\wintask\scripts.

4. Click the **New** button on the WinTask Toolbar.

5. Click the **Record** button to start Recording Mode. Click the **OK** button on the **Starting Recording Mode** dialog box.

6. The **Launching a Program** dialog box will appear. Enter **WordPad** into the **Program** field and click the **OK** button. If you use a x64 version of Windows, click **Browse** button and browse for C:\Program Files (x86)\Windows NT\Accessories\wordpad.exe file.

7. Click on the WordPad text area and type **Hello WordPad** followed by the **Enter** key.

8. Click the **Stop Recording** button on the WinTask Floating Toolbar to stop Recording Mode.

9. Close WordPad without saving.

10. Modify the script just recorded to use a loop to read the contents of the **test09.txt** file one line at a time. Use a string variable to hold the line read from the file. Place the loop after WordPad is invoked and before the text **Hello WordPad** is entered into the WordPad text area.

TIP: Use the *Eof* function to detect when the **test09.txt** file has been completely read.

11. Copy the line that enters text into WordPad inside the loop. Replace the text **Hello WordPad** with the string variable that contains the line read from the **test09.txt** file.

12. Click the **Play** button on the WinTask Toolbar to replay the actions listed in the script. Enter the name **script09.src** when the **Save As** dialog is displayed. Correct any compilation errors if the Output window displays errors and run again.

This exercise demonstrated how to read data from a file for use elsewhere in the script. The line of data read from the file could very easily be written into another file making an exact duplicate of the original file.

ReadExcel and WriteExcel

The WinTask Script Language provides two functions that interface to Microsoft Excel™ allowing the script developer to read and write Excel formatted files (.XLS). The function accepts a full pathname, a UNC, or a file from the current working directory. Microsoft Excel must be installed for these functions to execute properly.

The **ReadExcel** function is used to read a block of consecutive cells from an Excel spreadsheet file. The cells read from the spreadsheet must be either in the same row or the same column. The spreadsheet can be open in Excel when reading from the Excel file.

The **WriteExcel** function is used to write a block of consecutive cells to an Excel spreadsheet file. The cells written to the spreadsheet must be either in the same row or the same column. The spreadsheet cannot be open in Excel when writing to the Excel file. The function cannot create an Excel spreadsheet file and as such is restricted to modifying existing files.

The two functions have the following syntax:

ret = ReadExcel(ExcelFileName$, CellRangeDesc$, CellStrArray$())

ret = WriteExcel(ExcelFileName$, CellRangeDesc$, CellStrArray$())

The ***ExcelFileName$*** parameter specifies the name of the Excel spreadsheet file.

The ***CellRangeDesc$*** parameter specifies which cells are to be read or written. The string passed to the function is composed of two parts: a sheet descriptor and a cell descriptor separated by an exclamation (Ex. *"Sheet1!A9:F9"*). The sheet descriptor can be omitted if the spreadsheet only contains one sheet. The string *"Expenses!B3:B5"* will read the cells in column B, rows 3 through 5 on the sheet labeled Expenses. The string *"D8:H8"* will read the cells in row 8, columns D through H on the first sheet regardless of it's name.

The ***CellStrArray$()*** parameter specifies a string array which is to receive or supply the cell data. The cell data is stored in consecutive locations in the array starting at location 0. The string array must be declared at the beginning of the script using the *DIM* statement.

<u>Exercise 10</u>

This programming exercise demonstrates how a script can be used to update the contents of an Excel spreadsheet. This exercise assumes that Exercise 9 has been completed. If not, please refer to this chapter. See the WinTask Help System for additional information to complete this exercise. Appendix C contains the solution for this exercise.

1. Open Excel. From Excel, select menu item **File/Save As**.

2. Save the file as **exercise10.xlsx** in the c:\program files (x86)\wintask\scripts folder. Close Excel. If you use an Excel version prior Excel 2007, call the file **exercise10.xls**.

3. Launch the WinTask Editor. The title bar should display **WinTask – [Untitled1]**. If WinTask opens a previously saved script (Ex. **script09.src**), close the window and click the **New** button on the WinTask Toolbar.

4. Use the DIM statement to declare an array of 25 strings at the beginning of the script.

5. Add a loop to read the contents of the **test09.txt** file one line at a time. Use the string array to hold the lines read from the file. Use the *Eof* function to determine when to break out of the loop.

6. Add an integer variable to count the number of lines read from the file and to index into the string array. Take into account that indexing into the string array is zero-based.

7. Use the *MsgBox* function to print the number of entries in the string array. Use the *Str$* function to convert an integer variable to printable text.

8. Use the *WriteExcel* function to write the contents of the string array into the spreadsheet. Modify the file **exercise10.xlsx** that was created in step 2. Use a cell range descriptor that updates row 1, cells A1 through J1. It is assumed that file **test09.txt** contains 10 lines.

9. Use another *WriteExcel* function to write the contents of the string array into the spreadsheet a second time. Use a cell range descriptor that updates column C, cells C3 through C12.

10. Click the **Play** button on the WinTask Toolbar to replay the actions listed in the script. Enter the name **script10.src** when the **Save As** dialog is displayed. Correct any compilation errors if the Output window displays errors and run again.

11. Open file **exercise10.xlsx** in Excel. Verify that the appropriate cells have the correct information in them.

This exercise demonstrated how to modify the contents of an Excel spreadsheet. As can be seen from the exercise, access of an Excel spreadsheet file is essentially identical to reading or writing from any other file.

Subroutines and Functions

The WinTask Script Language provides many pre-defined functions that make the job of automating a task much easier. As one becomes more proficient with the language, they will undoubtedly recognize that similar functionality is repeated over and over when scripts are being written. The WinTask Script Language supports Subroutines and Functions that allow the script developer to consolidate common functionality. Coding Best Practices recommend the use of Subroutines and Functions to promote code reuse and to make automation script development more efficient.

Sub...ExitSub...EndSub

The syntax for the **Sub**, **ExitSub** and **EndSub** statements is as follows:

> ***Sub*** *<sub_name>([<param1>[, <param2>] ...])*
>
> *<statements>*
>
> ***ExitSub***
>
> *<statements>*
>
> ***EndSub***

At a minimum, a Subroutine consists of a block of script statements surrounded by the **Sub** and **EndSub** statements. The **ExitSub** statement is typically used in conjunction with a conditional statement to leave the Subroutine before reaching the **EndSub** statement. The name of the Subroutine is specified by *<sub_name>* and should be both descriptive and unique. Subroutines can also accept optional parameters, as designated by *<param1>*, which the Subroutine can perform operations on. The name of each parameter is a valid variable until the **EndSub** statement. Strings passed to the Subroutine should end with the **$** character.

TIP: Make sure the parameter names are different from any variables used elsewhere in the script. Since variables are global in scope, defining a parameter with the same name as a variable can lead to confusion as to which one (variable or parameter) is being referenced in the Subroutine.

TIP: Make sure the Subroutine appears in the script file before attempting to invoke it.

Converting a Script into a Subroutine

Any collection of script statements can be wrapped into a Subroutine that can be used several times during the automation of an application or task. The actual effort requires no more than a little planning to decide the scope of the Subroutine, it's input parameters, adding the **Sub** and **EndSub** statements, and finally invoking the Subroutine with the appropriate parameters.

What follows is the automation script that was created during Exercise 2.

```
Shell("notepad",1)
UseWindow("NOTEPAD.EXE|Edit|Untitled - Notepad|1",1)
    SendKeys("Hello<Enter>")
UseWindow("NOTEPAD.EXE|Notepad|Untitled - Notepad",1)
    ChooseMenu(Normal, "&Edit|Time/&Date      F5")
    ChooseMenu(Normal,"&File|E&xit")

UseWindow("NOTEPAD.EXE|CtrlNotifySink|Notepad|7",1)

    Click(Button,"&Save")

UseWindow("NOTEPAD.EXE|FloatNotifySink|Save As|1",1)

    WriteCombo("1","test02.txt")

UseWindow("NOTEPAD.EXE|#32770|Save As",1)

    Click(Button,"&Save")
```

The script presents two opportunities to generalize its behavior. The text written into the Notepad text area, *"Hello<Enter>"*, and the text file that Notepad creates, *"test02.txt"*, can both be replaced by input parameters. The above script statements can be wrapped into a Subroutine. The Subroutine and its invocation follows:

```
Sub WriteTextToFile(Text$, FileName$)

    Shell("notepad",1)
    UseWindow("NOTEPAD.EXE|Edit|Untitled - Notepad|1",1)
        SendKeys(Text$ + "<Enter>")
    UseWindow("NOTEPAD.EXE|Notepad|Untitled - Notepad",1)
        ChooseMenu(Normal, "&Edit|Time/&Date  F5")
    CloseWindow("NOTEPAD.EXE|Notepad|Untitled - Notepad",1)
    UseWindow("NOTEPAD.EXE|CtrlNotifySink|Notepad|7",1)
        Click(Button, "&Save")
    UseWindow("NOTEPAD.EXE|FloatNotifySink|Save As|1",1)
        WriteCombo("1",FileName$)
    UseWindow("NOTEPAD.EXE|#32770|Save As",1)
        Click(Button, "&Save")

EndSub



' Main Program

Text1$ = "The sky is blue, except when it rains!"
FileName1$ = "blue_sky.txt"
WriteTextToFile(Text1$, FileName1$)

Text2$ = "Balloons come in many shapes and sizes."
FileName2$ = "balloons.txt"
WriteTextToFile(Text2$, FileName2$)
```

The new Subroutine is more versatile than the original script due to the two input parameters. The Main Program shows just two instances of how the Subroutine can be invoked.

Exercise 11

This programming exercise demonstrates how existing functionality can be repackaged into a Subroutine to provide code reuse. This exercise assumes that Exercises 7 and 8 have been completed. If not, please refer to the chapters titled **Window Management Functions** and **Iteration**. See the WinTask Help System for additional information to complete this exercise. Appendix C contains the solution for this exercise.

1. Launch the WinTask Editor. The title bar should display **WinTask – [Untitled1]**. If WinTask opens a previously saved script (Ex. **script10.src**), close the window and click the **New** button on the WinTask Toolbar. This window will be used to create the new script.

2. Click the **Open** button on the WinTask Toolbar and open the **script08a.src** script file.

3. Select the entire contents of the **script08a.src** script file and copy them to the clipboard (**CTRL+C** or menu item **Edit/Copy**). The window for **script08a.src** can be closed. Do not save if prompted.

4. From WinTask Editor window, switch to **Untitled1**. Paste the contents of the clipboard into this window (**CTRL+V** or menu item **Edit/Paste**).

5. Wrap the functionality of **script08a.src** into a Subroutine named **Fill_Notepad** with three parameters, **FileName$**, **Text$** and **Count**. The **FileName$** parameter specifies the name of the saved Notepad file. The **Text$** parameter specifies the line of text written into the Notepad text area. **Count** specifies how many times to write **Text$** into the Notepad text area.

TIP: Replace variables from **script08a.src** with the calling parameters.

TIP: The readability of the script can be improved by indenting the statements between the **Sub** and **EndSub** statements. Also leave several blank lines between Dim statements, Subroutines, Functions, and the Main Program to help delineate them.

6. Click the **Open** button on the WinTask Toolbar and open the **script07b.src** script file.

7. Search for the **ChooseMenu(Normal,"&File|E&xit")** statement. Select all statements from the **ChooseMenu** statement to the end of the script and copy them to the clipboard. Add above the line **ChooseMenu(Normal,"&File|E&xit")** the **UseWindow("NOTEPAD.EXE|Notepad|Untitled - Notepad",1)** line. The window for **script07b.src** can be closed. Do not save if prompted.

8. From WinTask, select menu item **Window/Untitled1**. Paste the contents of the clipboard into the window at the end of the Subroutine but before the **EndSub** statement.

9. Modify the script copied from **script07b.src** to create a text file as specified by the **FileNames$** parameter.

10. Add a Main Program section after the **EndSub** statement. Invoke the **Fill_Notepad** Subroutine with the required parameters. Specify **"text11.txt"** as the filename, **"Subroutine Exercise"** as the text, and generate **12** lines of text in the file.

11. Click the **Play** button on the WinTask Toolbar to replay the actions listed in the script. Enter the name **script11.src** when the **Save As** dialog is displayed. Correct any compilation errors if the Output window displays errors and run again.

12. Questions for Discussion: What happens if the script statements in the Main Program section appear before the Subroutine in the script file?

This exercise demonstrated how to wrap a group of script statements into a Subroutine. It was also shown how "hard-coded" values could be replaced by parameters adding to the utility of the Subroutine.

Function...ExitFunction...EndFunction

The syntax for the **Function**, **ExitFunction** and **EndFunction** statements is as follows:

> ***Function*** *<function_name>([<param1>[, <param2>] ...])*
>
> *<statements>*
>
> ***<function_name> = <value>***
>
> ***ExitFunction***
>
> *<statements>*
>
> ***EndFunction***

Functions are similar to Subroutines with the exception that Functions return a result to the calling script statement. A Function consists of a block of script statements surrounded by the **Function** and **EndFunction** statements and the assignment of a return value to the Function's name. The name of the Function is specified by *<function_name>* and should be both descriptive and unique. By default a Function returns an integer value. A Function can return a string by appending the **$** character to the end of the Function name.

The **ExitFunction** statement is typically used in conjunction with a conditional statement to leave the Function before reaching the **EndFunction** statement. If the script developer decides to use the **ExitFunction** statement, an assignment to *<function_name>* must precede the **ExitFunction** statement. Functions can also accept optional parameters, as designated by *<param1>*, which the Function can perform operations on. The name of each parameter is a valid variable until the **EndFunction** statement. Strings passed to the Function should end with the **$** character.

TIP: Make sure the parameter names are different from any variables used elsewhere in the script. Since variables are global in scope, defining a parameter with the same name as a variable can lead to confusion as to which one (variable or parameter) is being referenced in the Function.

TIP: Make sure the Function appears in the script file before attempting to invoke it.

TIP: Failure to set the Function name equal to a value returns 0 to the calling script statement. An empty string is returned if the Function returns a string but a value was not assigned to the Function name.

An example of a Function and its invocation follows:

```
Function Absolute_diff(Integer1, Integer2)
    If (Integer1 = Integer2) Then
        Absolute_diff = 0
        ExitFunction
    Endif
    If (Integer1 > Integer2) Then
        Absolute_diff = integer1 - integer2
    Else
        Absolute_diff = Integer2 - Integer1
    Endif
EndFunction


' Main Program
TestValue = 89
Result = Absolute_diff(TestValue, 123)
```

Exercise 12

This programming exercise demonstrates how existing functionality can be repackaged into a Function to provide code reuse. This exercise assumes that Exercises 9 and 11 have been completed. If not, please refer to this chapter and the chapter titled **File Functions**. See the WinTask Help System for additional information to complete this exercise. Appendix C contains the solution for this exercise.

1. Launch the WinTask Editor. The title bar should display **WinTask – [Untitled1]**. If WinTask opens a previously saved script (Ex. **script11.src**), close the window. If **script11.src** is opened, you can skip Step 2.
2. Click the **Open** button on the WinTask Toolbar and open the **script11.src** script file.
3. From WinTask, select menu item **File/Save As**. Enter the name **script12.src** when the **Save As** dialog is displayed. This step prepares for this exercise by changing the script name and keeps the **script11.src** available as a reference.
4. Click the **Open** button on the WinTask Toolbar and open the **script09.src** script file.

5. Select the entire contents of the **script09.src** script file and copy them to the clipboard. The window for **script09.src** can be closed. Do not save if prompted.

6. From WinTask Editor, switch to **script12**. Paste the contents of the clipboard into the window after the **EndSub** statement and before the Main Program section.

7. Wrap the functionality of **script09.src** into a Function named **Fill_Wordpad** with a single parameter, **FileName$**. The **FileName$** parameter specifies the name of a saved text file. The Function should return the number of lines written into the WordPad text area.

8. Modify the calling parameters to **Fill_Notepad** in the Main Program section to pass **"c:\program files\wintask\scripts\text12.txt"** as the filename, **"Subroutine and Function Exercise"** as the text, and generate **8** lines of text in the file. Invoke the **Fill_Wordpad** Function with the required parameter after the call to **Fill_Notepad**. Specify the same filename that is passed to **Fill_Notepad**. Assign the value returned by **Fill_Wordpad** to a variable. Use the *MsgBox* Function to display the number of lines written into WordPad.

9. Click the **Play** button on the WinTask Toolbar to replay the actions listed in the script. Correct any compile errors if the Output window displays errors and run again.

10. Dismiss the Message Box and close WordPad without saving. The Message Box should report that 9 lines were written into the WordPad text area.

11. Questions for Discussion: What happens if the script statements in the Main Program section appear before the Function in the script file? How many lines are reported as being written into WordPad if the line count is not assigned to Fill_WordPad in the Function?

This exercise demonstrated how to wrap a group of script statements into a Function. It was also shown how "hard-coded" values could be replaced by parameters adding to the utility of the Function. And finally, it was shown how the invoking script statement could use the return value from a Function.

Putting it all Together

The exercises in this chapter have drawn upon the information presented in this document to create a structured automation script file that is capable of performing real world tasks. The WinTask Script Language contains additional statements such as *Include*, *Run$*, *Local*, and a host of other functions which you are encouraged to explore in the WinTask Help System. A few hours of general browsing may help you save days when creating your automation scripts. We leave this chapter with one final exercise.

Exercise 13

One piece that hasn't been integrated into the previous exercises is an array. This exercise builds on Exercises 10 and 12 to add an array. If they haven't been completed, please refer to this chapter and the chapter titled **File Functions**. See the WinTask Help System for additional information to complete this exercise. Appendix C contains the solution for this exercise.

1. Open Excel. From Excel, select menu item **File/Save As**.

2. Save the file as **exercise13.xlsx** in the c:\program files (x86)\wintask\scripts folder. If you use an Excel version prior Excel 2007, the file is saved with the extension xls instead of xlsx. Close Excel.

3. Launch the WinTask Editor. The title bar should display **WinTask – [Untitled1]**. If WinTask opens a previously saved script (Ex. **script12.src**), close the window. If **script12.src** is opened, you can skip Step 4.

4. Click the **Open** button on the WinTask Toolbar and open the **script12.src** script file.

5. From WinTask, select menu item **File/Save As**. Enter the name **script13.src** when the **Save As** dialog is displayed. This step prepares for this exercise by changing the script name and keeps the **script12.src** available as a reference.

6. Click the **Open** button on the WinTask Toolbar and open the **script10.src** script file.

7. Copy the array definition (**DIM**) statement from the top of the **script10.src** window to the clipboard. Switch to the **script13.src** window and paste it into the window at the beginning of the script.

TIP: Note that an array cannot be passed as a parameter. They must be declared at a global level.

8. Switch to the **script10.src** window and copy the remainder of the script to the clipboard. Switch to the **script13.src** window and paste it into the window after the **EndFunction** statement and before the Main Program section.

9. Wrap the functionality of **script10.src** into a Subroutine named **Fill_Excel** with two parameters, **FileName$** and **ExcelFileName$**. The **FileName$** parameter specifies the name of a saved text file. The **ExcelFileName$** parameter specifies the name of an empty spreadsheet file to update.

TIP: Note that the calls to **Read** in **Fill_Wordpad** move the file position to the end of the file. When **Fill_Excel** attempts to read the same file, the file position will still be at the end of file since they are both executed in the same script. The result is that nothing is read from the file. Add a call to **SetReadPos** before attempting to read from the file in **Fill_Excel** to correct this situation. See the WinTask Help System for more information.

10. Delete the **MsgBox** call from **script10.src**.

11. Add logic to the **While/Wend** block of statements so that the code does not attempt to access elements beyond the array boundaries.

12. This exercise will generate a text file with only 5 lines. Replace the **"Sheet1!A1:J1"** and **"Sheet1!C3:C12"** strings used to designate the spreadsheet cells with **"Sheet1!A1:E1"** and **"Sheet1!C3:C7"** respectively.

13. Optional: Replace the **"Sheet1!A1:E1"** and **"Sheet1!C3:C7"** strings used to designate the spreadsheet cells with string variables created on the fly based upon the number of elements in the array. Use A1 as the first cell when updating a row in the spreadsheet. Use C3 as the first cell when updating a column in the spreadsheet.

TIP: Use the **Asc**, **Chr$**, **Str$** and **Val** functions to calculate the name of the last cell in the row or column to be updated. Use string concatenation to build the final string variables. See the WinTask Help System for more information.

14. Modify the calling parameters to **Fill_Notepad** in the Main Program section to pass **"text13.txt"** as the filename, **"Sub, Func, Excel Exercise"** as the text, and generate **5** lines of text in the file. Invoke the **Fill_Excel** Subroutine with the required parameters after the call to **Fill_Wordpad**. Specify the same filename that is passed to **Fill_Notepad** and **Fill_Wordpad** as the first parameter. Specify **"exercise13.xls"** as the second parameter (or **exercise13.xlsx** if you use Excel 2007). Move the **MsgBox** function to the last line of the script.

15. Click the **Play** button on the WinTask Toolbar to replay the actions listed in the script. Correct any compilation errors if the Output window displays errors and run again.

16. Dismiss the Message Box and close WordPad without saving. The Message Box should report that 6 lines were written into the WordPad text area. Open the Excel spreadsheet file to verify that it contains the correct information.

This exercise demonstrated how to add and utilize an array in the context of a structured script file. At this point you should be able to write relatively complex scripts to automate your application or process.

Compilation errors

When you run a script (SRC file), it is first compiled. In case of compilation errors, those are listed in the Output window of the Editor (if the Output window is not displayed, select menu item **View/Output window** to display it).

Double clicking an error line puts the cursor in the main window of the Editor on the line which is in error for an easy correction.

Remember the order of statements within a script:
Dim statements first,
Sub and Functions second, and a call to a function within another one needs the declaration of the called function before,
Main program last.

Included scripts must respect this structure

Include statement: Include "tools.src"

Execution errors

Trace using MsgBox or MsgFrame

Exercise 14

1. Launch the WinTask Editor. The title bar should display **WinTask – [Untitled1]**. If WinTask opens a previously saved script, close the window and click the **New** button on the WinTask toolbar. This window will be used to create the new script.

2. Click the **Record** button to start Recording Mode. Click the **OK** button on the **Starting Recording Mode** dialog box.

3. The **Launching a Program** dialog box will appear. Enter **notepad** into the **Program** field and click the **OK** button.

4. Click on the notepad text area and type **Hello** followed by the **Enter** key.

5. Close notepad by selecting menu item **File/Exit** without saving.

6. Click the **Stop Recording** button on the WinTask Floating Toolbar to stop Recording Mode.

7. Edit the script for inserting a loop with two indexes, i and j, which type Helloi and WinTaskj in notepad. Start index i at value 0 and make it increase, start index j at value 10 and make it decrease. Repeat the loop until i=8.

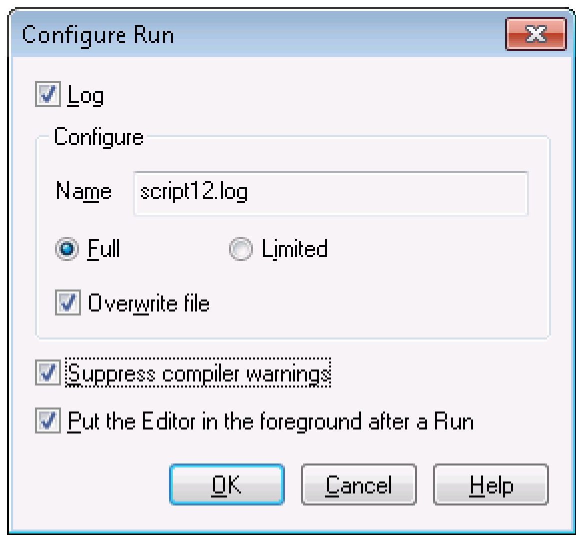
8. Click the **Play** button on the WinTask toolbar to replay the actions listed in the script. Enter the name **script14.src** when prompted for saving. Correct any compilation errors it the Output window displays errors and run again.

Let's suppose now that we have trouble with which value has j index. Insert in the script first a Msgbox function for displaying j index for each iteration. Play.
Then change Msgbox function by a MsgFrame function (see MsgFrame help). Play.

TIP: Note that Msgbox displays the value and prompts for an OK while MsgFrame just displays the value. So Msgbox takes the focus and stops script execution.

Log file

It is possible to request that each line during execution is logged in a log file. This option is in menu item **Configure/Run**:



Full log generates a log file that each script line logged. Limited log writes in the log only the Comment function lines encountered within the script.

<u>Exercise 15</u>

1. Launch the WinTask Editor. If the **script14.src** script file from the previous exercise does not open, click the **Open** button on the WinTask Toolbar and open the **script14.src** script file.

2. Configure Run for a full log (select menu item **Configure/Run**) and **Play** the script.

3. Open the generated log file (select menu item **File/Open Log**) – see the generated lines. Close the log.

4. Configure now Run for a limited log.

5. Add above the until line the line Comment ("value for i is:"+str$(i)).

6. Add the same type of Comment for j index.

7. Click the **Play** button on the WinTask toolbar to replay the actions listed in the script. Enter the name **script15.src** when prompted for saving. Correct any compilation errors it the Output window displays errors and run again.

8. Open the log.

Creation of your own log procedure:

You can write yourself your own log Sub; here is an example:

```
sub log(msg_log$)
	local buffer$
	buffer$=Date$()+", "+hour$()+":"+min$()+":"+sec$()+" --> "+msg_log$
	write(file_log$,buffer$,CRLF)
endsub
```

Using the script done in exercise 15, integrate this Sub in the script, and replace the two COMMENT lines by a call to this proc. Save the script as script15b and Play.
Open the log.

Object not found (window, button, menu option, ...)

It's the most common execution error.

To understand why, write down the line number of the error, click OK on the error message and look at the application under automation exactly at the state where the error occurs.

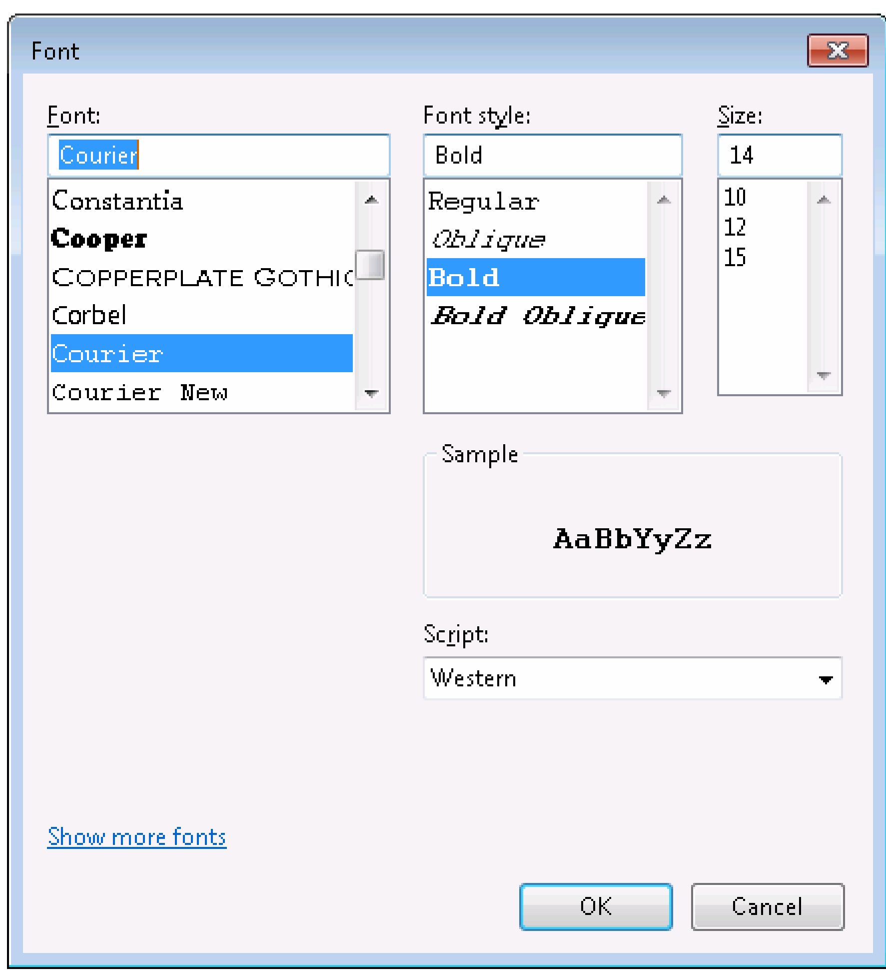
- Is the object there or not; if not, a possible is a synchro problem, add a quick Pause x secs in the script to validate this opinion. If it's that, you can then enhance the synchro by a nicer synchro than an absolute one.
- The object seems to be there. Use Spy to validate that its name at replay is really the one that is in the script.

The script is long, you don't understand what's happening at all. Still stay on the window where the problem occurs, and extract in a new script the couple of lines in your original script where the error occurs. Play like that, it will demonstrate more easily where the exact problem is.

Incorrect filling (WriteEdit or WriteCombo or ChooseItem versus SendKeys) {XE "ChooseItem"}

When a couple of fields have to be filled, WinTask identifies them by an index. Sometimes, at replay, the index could have changed and so it does not fill correctly.

Example on Notepad (under a Windows 64-bit, launch notepad from c:\windows\SysWoW64), Font dialog box:



Using Recording mode, on that window, we record the selection of Courier and stop Recording. Here are the two lines:

```
UseWindow("NOTEPAD.EXE|#32770|Font",1)
    ChooseItem(Combo, "1", "Courier", single, left )
```

The "1" parameter in ChooseItem means the Combo 1, and so on real applications, this number can make an execution error if the combo or editboxes have been renumbered. In that case, you can replace (manually) by :

```
UseWindow("NOTEPAD.EXE|#32770|Font",1)
    SendKeys("Courier",NoActivate)
    pause 5 ticks
    SendKeys("<Tab>",Noactivate)
    pause 5 ticks
    SendKeys("Bold",NoActivate)
```

NoActivate keyword means that Sendkeys must send the specified keys where the cursor is, whatever window. It's a good practise to add a small pause when NoActivate keyword is used to avoid a too fast typing.

Item in a list not correctly selected

Again let's take the Font dialog box in notepad and we record the selection of Bold in the list of Font style. Here are the two lines:

```
UseWindow("NOTEPAD.EXE|#32770|Font",1)
   ChooseItem(Combo, "2", "Bold", single, left )
```

The generated code is correct because the proposed list is a standard Windows Combo. If the application under automation has been developed with specific development tools, such a list could be misunderstood by WinTask. Then the way to select an item is to use Down keys. So in our example:

```
UseWindow("NOTEPAD.EXE|#32770|Font",1)
   SendKeys("<Down>",Noactivate)
```

Menu option not correctly selected

In that case, it's usually possible to use directly the shortcut for selecting options.

Example on notepad, File/Save option:

```
UseWindow("NOTEPAD.EXE|Notepad|Untitled - Notepad")
      SendKeys("<Alt f>",Noactivate)
      pause 5 ticks
      SendKeys("a",NoActivate)
```

The window name must be the one for the Menu window (use Spy), and again the small pause for letting the File menu open.

A click not correctly executed

This happens on software where options are not in the menus but in icons. WinTask does not access by object icons.

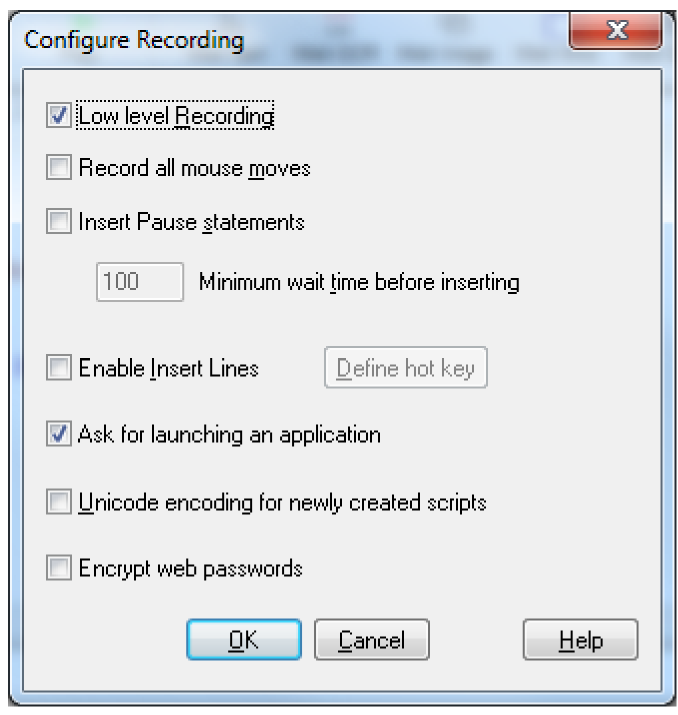
Two functions for addressing this problem:
ClickOnText
ClickOnBitmap
Use their wizard to automatically generate the correct lines in the script.

Low level Recording mode when nothing works

With this other way to record, the actions are recorded as mouse clicks not any more as objects.

Use only for a short time just when needed.

Configure by selecting menu item **Configure/Recording**.



Example, menu File/Open recorded in normal way versus recording in Low level:

```
Shell("notepad",1)

UseWindow("NOTEPAD.EXE|Notepad|Untitled - Notepad",1)
    ChooseMenu(Normal,"&File|&Open...    Ctrl+O")

UseWindow("NOTEPAD.EXE|#32770|Open",1)
    Click(Button,"Cancel")

CloseWindow("NOTEPAD.EXE|Notepad|Untitled - Notepad",1)
```

In Low level:

```
Shell("notepad",1)

UseWindow("NOTEPAD.EXE|Notepad|Untitled - Notepad",1)
    ClickMouse(Left,Down,30,51)
    ClickMouse(Left,Up,30,51)
    ClickMouse(Left,Down,22,93)
    ClickMouse(Left,Up,22,93)

UseWindow("NOTEPAD.EXE|Button|Cancel",1, NoActivate)
    ClickMouse(Left,Down,72,19)
    ClickMouse(Left,Up,72,19)

UseWindow("NOTEPAD.EXE|Notepad|Untitled - Notepad",1)
    ClickMouse(Left,Down,530,16)
    ClickMouse(Left,Up,530,16)
```

Slow execution

Exercise 16

1. Launch notepad (under a Windows 64-bit, launch notepad from c:\windows\SysWoW64), and type Hello in notepad.

2. Then launch the WinTask Editor. The title bar should display **WinTask – [Untitled1]**. If WinTask opens a previously saved script, close the window and click the **New** button on the WinTask toolbar. This window will be used to create the new script.

3. Click the **Record** button to start Recording Mode. Check the **Nothing** button on the **Starting Recording Mode** dialog box and click **OK** button.

4. Record the **File/Print** menu item and **Cancel**.

5. Click the **Stop Recording** button on the WinTask Floating Toolbar to stop Recording mode.

6. Close notepad and save the document as **Document1**.

7. Open manually the document Document1.

8. Play the script, when prompted save it as **script16.src**.

Why it waits before replaying the File/Print option?
How to avoid the delay?

The delay is due to this line:
UseWindow("NOTEPAD.EXE|Notepad|Untitled - Notepad",1)
When you run the script after having opened manually Document1, the window name is not any more *"NOTEPAD.EXE|Notepad|Untitled - Notepad"* but *"NOTEPAD.EXE|Notepad|Document1 – Notepad".*
At replay, WinTask looks first for a window with the exact same window name and after 3 secs (default value related to #ActionTimeout system variable) starts to truncate the window name by its rightmost part.
If you truncate manually the window name to accept any document name, such as: *"NOTEPAD.EXE|Notepad|"*, the window will be immediately found.

Two other system variables can be used for speeding up execution:
#ExecuteDelay
#SendKeysDelay
See help on those variables.

Conclusion

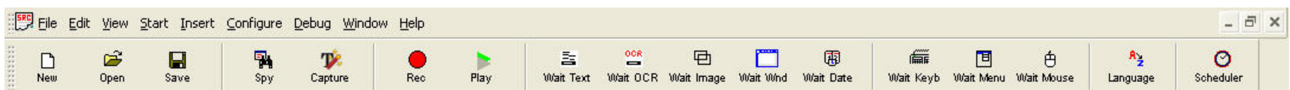
This book did not address the specific WinTask functions for Web automation. Direct Tutorials are available on our web site http://www.wintask.com/tutorials.php

Feel free to post your remarks on this book at info@wintask.com

APPENDIX A

WinTask Toolbar

The WinTask Toolbar offers the following functionality. The toolbar and button images present the large buttons.



New	**New Script Window**. Open a new automation script window.
Open	**Open Script File**. Open an existing automation script file and load into a new window.
Save	**Save Script File**. Save the contents of the active automation script window to a file.
Spy	**Start Spy Tool**. Launch the Spy tool.
Capture	**Start Capture Wizard**. Launch the Capture tool.
Rec	**Start Recording Mode**. Record the user's keyboard, menu, and mouse actions and record them in the active automation script window.
Play	**Compile and Execute**. Compile the automation script in the active window and replay.
Wait Text	**Text Synchronization**. Invokes the Text Synchronization wizard. Insert the generated script into the active automation script window.
Wait OCR	**OCR Text Synchronization**. Invokes the OCR Text Synchronization wizard. Insert the generated script into the active automation script window.
Wait Image	**Image Synchronization**. Invokes the Image Synchronization wizard. Insert the generated script into the active automation script window.
Wait Wnd	**Window Synchronization**. Invokes the Window Synchronization wizard. Insert the generated script into the active automation script window.
Wait Date	**Date/Hour Synchronization**. Invokes the Date/Hour Synchronization wizard. Insert the generated script into the active automation script window.
Wait Keyb	**Keyboard Synchronization**. Invokes the Wait for Keyboard Action Synchronization wizard. Insert the generated script into the active automation script window.

Wait Menu	**Menu Synchronization**. Invokes the Wait for Menu Action Synchronization wizard. Insert the generated script into the active automation script window.
Wait Mouse	**Mouse Synchronization**. Invokes the Wait for Mouse Click Action Synchronization wizard. Insert the generated script into the active automation script window.
Language	**Language List**. Open the Insert Language dialog to insert WinTask Script Language statements into the active automation script window.
Scheduler	**Start Scheduler Tool**. Launch the Scheduler tool to execute an automation script at a specified time (Scheduler is not available under Vista/Windows 7/2008 – Scheduler is included in WinTask x64).

APPENDIX B

WinTask Floating Toolbar

During Recording Mode, WinTask converts itself into a floating toolbar. The toolbar offers the following functionality while in Recording Mode:

	Stop Recording Mode. Restores the WinTask Editor.
	Start Spy Tool. Launches the Spy tool.
	Start Capture Wizard. Launches the Capture Wizard.
	Text Synchronization. Invokes the Text Synchronization wizard. Insert the generated script into the automation script being recorded.
	OCR Text Synchronization. Invokes the OCR Text Synchronization wizard. Insert the generated script into the automation script being recorded
	Image Synchronization. Invokes the Image Synchronization wizard. Insert the generated script into the automation script being recorded.
	Window Synchronization. Invokes the Window Synchronization wizard. Insert the generated script into the automation script being recorded.
	Time Synchronization. Invokes the Time Synchronization wizard. Insert the generated script into the automation script being recorded.
	Date/Hour Synchronization. Invokes the Date and Time Synchronization wizard. Insert the generated script into the automation script being recorded.
	Keyboard Synchronization. Invokes the Wait for Keyboard Action Synchronization wizard. Insert the generated script into the automation script being recorded.
	Menu Synchronization. Invokes the Wait for Menu Action Synchronization wizard. Insert the generated script into the automation script being recorded.
	Mouse Synchronization. Invokes the Wait for Mouse Click Action Synchronization wizard. Insert the generated script into the automation script being recorded.

	Insert Lines. Opens the Insert Lines in to Script dialog in which the user can insert lines directly into the automation script being recorded.

APPENDIX C

Exercise Solutions

The solutions to the programming exercises are provided below.

Exercise 7, Script07a.src

```
'================================================================
'   Script07a
'
'   WinTask Training Version 3.8a
'   © Copyright 1997-2012 TaskWare      January 2012
'
'   Exercise 7, Part 1
'
'   This programming exercise demonstrates how a script can
'   be modified to test for the existence of a window and how
'   that information may be put to use.  You will be asked to
'   modify the script based upon the information presented in
'   the chapters titled "WinTask Scripting Language" and "Window
'   Management Functions".  See the WinTask Help System for
'
'   The notepad interface used in this exercise is the Vista/Windows 7 one.
'   Under XP or 2003, the Save as window names and buttons are different.
'================================================================

' This string variable defines the name of the text file
' to be used in the Notepad Save As dialog
filename$ = "test07"

Shell("notepad",1)

UseWindow("NOTEPAD.EXE|Edit|Untitled - Notepad|1",1)
        SendKeys("Hello<Enter>")

UseWindow("NOTEPAD.EXE|Notepad|Untitled - Notepad",1)
        ChooseMenu(Normal,"&Edit|Time/&Date           F5")
        ChooseMenu(Normal,"&File|E&xit")

UseWindow("NOTEPAD.EXE|CtrlNotifySink|Notepad|7",1)
        Click(Button,"&Save")

' The string variable filename$ replaced "test07" in the
' following block
UseWindow("NOTEPAD.EXE|FloatNotifySink|Save As|1",1)
        WriteCombo("1",filename$)

        UseWindow("NOTEPAD.EXE|#32770|Save As",1)
        Click(Button,"&Save")
```

Exercise 7, Script07b.src

```
'=================================================================
'  Script07b
'
'  WinTask Training Version 3.8a
'  © Copyright 1997-2012 TaskWare      January 2012
'
'  Exercise 7, Part 2
'
'  This programming exercise demonstrates how a script can
'  be modified to test for the existence off a window and how
'  that information may be put to use.  You will be asked to
'  modify the script based upon the information presented in
'  the chapters titled "WinTask Scripting Language" and "Window
'  Management Functions".  See the WinTask Help System for
'  additional information to complete this exercise.
'=================================================================

' This string variable defines the name of the text file
' to be used in the Notepad Save As dialog
filename$ = "test07"

Shell("notepad",1)

UseWindow("NOTEPAD.EXE|Edit|Untitled - Notepad|1",1)
        SendKeys("Hello<Enter>")

UseWindow("NOTEPAD.EXE|Notepad|Untitled - Notepad",1)
        ChooseMenu(Normal,"&Edit|Time/&Date            F5")
        ChooseMenu(Normal,"&File|E&xit")

UseWindow("NOTEPAD.EXE|CtrlNotifySink|Notepad|7",1)
Click(Button,"&Save")

UseWindow("NOTEPAD.EXE|FloatNotifySink|Save As|1",1)
        WriteCombo("1",filename$)

UseWindow("NOTEPAD.EXE|#32770|Save As",1)

Click(Button,"&Save")

' Confirm overwrite of existing file by testing for the Notepad
' Confirm Save As dialog using ExistW
' Small pause to let time for the window to appear
Pause 50 ticks
ret = ExistW("NOTEPAD.EXE|#32770|Confirm Save As",1)
If (ret = 1) Then
        UseWindow("NOTEPAD.EXE|CtrlNotifySink|Confirm Save As|7",1)
        Click(Button,"&Yes")
Endif
```

Exercise 7, Script07c.src

```
'=================================================================
'  Script07c
'
'  WinTask Training Version 3.8a
'  © Copyright 1997-2012 TaskWare      January 2012
'
'  Exercise 7, Part 3
'
'  This programming exercise demonstrates how a script can
'  be modified to test for the existence off a window and how
'  that information may be put to use.  You will be asked to
'  modify the script based upon the information presented in
'  the chapters titled "WinTask Scripting Language" and "Window
'  Management Functions".  See the WinTask Help System for
'  additional information to complete this exercise.
'=================================================================

' This string variable defines the name of the text file
' to be used in the Notepad Save As dialog
filename$ = "test07"

Shell("notepad",1)

UseWindow("NOTEPAD.EXE|Edit|Untitled - Notepad|1",1)
	SendKeys("Hello<Enter>")

UseWindow("NOTEPAD.EXE|Notepad|Untitled - Notepad",1)
	ChooseMenu(Normal,"&Edit|Time/&Date	F5")
	ChooseMenu(Normal,"&File|E&xit")

UseWindow("NOTEPAD.EXE|CtrlNotifySink|Notepad|7",1)
Click(Button,"&Save")

UseWindow("NOTEPAD.EXE|FloatNotifySink|Save As|1",1)
	WriteCombo("1",filename$)

UseWindow("NOTEPAD.EXE|#32770|Save As",1)

Click(Button,"&Save")

' Confirm overwrite of existing file by testing for the Notepad
' Confirm Save As dialog by ignoring errors
#IgnoreErrors = 1
ret = UseWindow("NOTEPAD.EXE|CtrlNotifySink|Confirm Save As|7",1)
If (ret = 0) Then
	Click(Button,"&Yes")
Endif
```

Exercise 7, Script07d.src

```
' '==================================================================
'  Script07d
'
'  WinTask Training Version 3.8a
'  © Copyright 1997-2012 TaskWare      January 2012
'
'  Exercise 7, Part 4
'
'  This programming exercise demonstrates how a script can
'  be modified to test for the existence off a window and how
'  that information may be put to use.  You will be asked to
'  modify the script based upon the information presented in
'  the chapters titled "WinTask Scripting Language" and "Window
'  Management Functions".  See the WinTask Help System for
'  additional information to complete this exercise.
'==================================================================

' This string variable defines the name of the text file
' to be used in the Notepad Save As dialog
filename$ = "test07d"

Shell("notepad",1)

UseWindow("NOTEPAD.EXE|Edit|Untitled - Notepad|1",1)
        SendKeys("Hello<Enter>")

UseWindow("NOTEPAD.EXE|Notepad|Untitled - Notepad",1)
        ChooseMenu(Normal,"&Edit|Time/&Date          F5")
        ChooseMenu(Normal,"&File|E&xit")

‘ Note that under Vista, those 2 lines below are not the same, see Part 1.
UseWindow("NOTEPAD.EXE|#32770|Notepad",1)
        Click(Button,"&Yes")

' The string variable filename$ replaced "test07" in the
' following block
UseWindow("NOTEPAD.EXE|Edit|Save As|1",1)
        SendKeys(filename$)

UseWindow("NOTEPAD.EXE|#32770|Save As",1)
        Click(Button,"&Save")

' Confirm overwrite of existing file by testing for the Notepad
' Save As Overwrite dialog by ignoring errors
#IgnoreErrors = 1
ret = UseWindow("NOTEPAD.EXE|CtrlNotifySink|Confirm Save As|7",1)
If (ret = 0) Then
        Click(Button,"&Yes")
Endif
```

Exercise 8, Script08a.src

```
'=====================================================================
'   Script08a
'
'   WinTask Training Version 3.8a
'   © Copyright 1997-2012 TaskWare      January 2012
'
'   Exercise 8, Part 1
'
'   This programming exercise demonstrates how a script can
'   be modified to repeat a block of code several times.
'   You will be asked to modify the script based upon the
'   information presented in the chapter titled "Iteration".
'   See the WinTask Help System for additional information
'   to complete this exercise.
'=====================================================================

Shell("notepad",1)

Greeting$ = "Notepad Text Line "
LineNumber = 1

UseWindow("NOTEPAD.EXE|Edit|Untitled - Notepad|1",1)
	' This block executes 10 times
	While (LineNumber <= 10)
		SendKeys(Greeting$ + Str$(LineNumber) + "<Enter>")
		LineNumber = LineNumber + 1
	Wend
```

Exercise 8, Script08b.src

```
'=================================================================
'   Script08b
'
'   WinTask Training Version 3.8a
'   © Copyright 1997-2012 TaskWare      January 2012
'
'   Exercise 8, Part 2
'
'   This programming exercise demonstrates how a script can
'   be modified to repeat a block of code several times.
'   You will be asked to modify the script based upon the
'   information presented in the chapter titled "Iteration".
'   See the WinTask Help System for additional information
'   to complete this exercise.
'=================================================================

Shell("notepad",1)

Greeting$ = "Notepad Text Line "
LineNumber = 1

UseWindow("NOTEPAD.EXE|Edit|Untitled - Notepad|1",1)
	' This block executes 10 times
	Repeat
		SendKeys(Greeting$ + Str$(LineNumber) + "<Enter>")
		LineNumber = LineNumber + 1
	Until (LineNumber > 10)
```

Exercise 9, Script09.src

```
'==================================================================
'  Script09
'
'  WinTask Training Version 3.8a
'  © Copyright 1997-2012 TaskWare      January 2012
'
'  Exercise 9
'
'  This programming exercise demonstrates how a script can
'  be used to read the contents of a file and use the
'  information to perform some other task.  This exercise
'  assumes that Exercise 8 has been completed.  If not, please
'  refer to the chapter titled "Iteration".  See the WinTask
'  Help System for additional information to complete this
'  exercise.
'==================================================================

Shell(Chr$(34)+"C:\Program Files (x86)\Windows NT\Accessories\wordpad.exe"+Chr$(34),1)

FileName$ = "test09.txt"

UseWindow("WORDPAD.EXE|RICHEDIT50W|Document - WordPad|1",1)
	' Execute block until End-of-File is reached
	While (Eof(FileName$) = 0)
		' Read one line from file into string variable
		ret = Read(FileName$, DataBuffer$, CRLF)
		' Copy line read from file into WordPad text area
		SendKeys(DataBuffer$ + "<Enter>")
	Wend
	SendKeys("Hello Wordpad<Enter>")
```

Exercise 10, Script10.src

```
'=================================================================
'  Script10
'
'  WinTask Training Version 3.8a
'  © Copyright 1997-2012 TaskWare      January 2012
'
'  Exercise 10
'
'  This programming exercise demonstrates how a script can
'  be used to update the contents of an Excel spreadsheet.
'  This exercise assumes that Exercise 9 has been completed.
'  If not, please refer to the chapter titled "File Functions".
'  See the WinTask Help System for additional information
'  to complete this exercise.
'=================================================================

' Define an array that holds up to 26 strings (zero-based, 0-25)
Dim LineOfText$(25)

FileName$ = "test09.txt"
' In this exercise, we use an Excel 2007 version, so the extension for Excel is xlsx.
ExcelFileName$ = "exercise10.xlsx"

LineNumber = 0
' Execute block until End-of-File is reached
While (Eof(FileName$) = 0)
        ' Read one line from file into string array
        ret = Read(FileName$, LineOfText$(LineNumber), CRLF)
        LineNumber = LineNumber + 1
Wend

LineCount = LineNumber

MsgBox("Number of Lines Read = " + Str$(LineCount))

' Write string array into Excel spreadsheet cells A1 through J1
WriteExcel(ExcelFileName$, "Sheet1!A1:J1", LineOfText$())

' Write string array into Excel spreadsheet cells C3 through C12
WriteExcel(ExcelFileName$, "Sheet1!C3:C12", LineOfText$())
```

Exercise 11, Script11.src

```
'=================================================================
'  Script11
'
'  WinTask Training Version 3.8a
'  © Copyright 1997-2012 TaskWare      January 2012
'
'  Exercise 11
'
'  This programming exercise demonstrates how existing
'  functionality can be repackaged into a Subroutine to
'  provide code reuse.  This exercise assumes that Exercises
'  7 and 8 have been completed.  If not, please refer to the
'  chapters titled "Window Management Functions" and "Iteration".
'  See the WinTask Help System for additional information to
'  complete this exercise.
'=================================================================


'=================================================================
'  This subroutine writes the specified text string a number
'  of times to a file.  The line number is appended to the
'  text string written to the Notepad text area.
'
'  Input Parameters:
'      FileName$       - File name to use when writing text
'      Text$           - Text string to write
'      Count           - Number of times Text$ is to be written
'=================================================================
Sub Fill_Notepad(FileName$, Text$, Count)

        Shell("notepad",1)

        LineNumber = 1

        UseWindow("NOTEPAD.EXE|Edit|Untitled - Notepad|1",1)
                ' Write specified the text into Notepad the specified
                ' number of times
                While (LineNumber <= Count)
                        SendKeys(Text$ + Str$(LineNumber) + "<Enter>")
                        LineNumber = LineNumber + 1
                Wend

        UseWindow("NOTEPAD.EXE|Notepad|Untitled - Notepad",1)
                        ChooseMenu(Normal,"&File|E&xit")


UseWindow("NOTEPAD.EXE|CtrlNotifySink|Notepad|7",1)
                Click(Button,"&Save")

        ' Save text to the specified filename
        UseWindow("NOTEPAD.EXE|Edit|Save As|1",1)
                WriteCombo("1",filename$)

        UseWindow("NOTEPAD.EXE|#32770|Save As",1)
                Click(Button,"&Save")

        Pause 20 ticks
        ' Confirm overwrite of existing file
        ret = ExistW("NOTEPAD.EXE|#32770|Save As",1)
        If (ret = 1) Then
                UseWindow("NOTEPAD.EXE|CtrlNotifySink|Confirm Save As|7",1)
```

```
                Click(Button,"&Yes")
        Endif

EndSub


'=================================================================
' Main Program
'=================================================================

NotepadFile$ = "text11.txt"
NotepadText$ = "Subroutine Exercise"

' Call subroutine to write 12 lines of text into file "text11.txt"
Fill_Notepad(NotepadFile$, NotepadText$, 12)
```

Exercise 12, Script12.src

```
'=================================================================
'  Script12
'
'  WinTask Training Version 3.8a
'  © Copyright 1997-2012 TaskWare      January 2012
'
'  Exercise 12
'
'  This programming exercise demonstrates how existing
'  functionality can be repackaged into a Function to provide
'  code reuse.  This exercise assumes that Exercises 9 and 11
'  have been completed.  If not, please refer to the chapters
'  titled "Subroutines and Functions" and "File Functions.
'  See the WinTask Help System for additional information
'  to complete this exercise.
'=================================================================


'=================================================================
'  This subroutine writes the specified text string a number
'  of times to a file.  The line number is appended to the
'  text string written to the Notepad text area.
'
'  Input Parameters:
'      FileName$       - File name to use when writing text
'      Text$           - Text string to write
'      Count           - Number of times Text$ is to be written
'=================================================================
Sub Fill_Notepad(FileName$, Text$, Count)

        Shell("notepad",1)

        LineNumber = 1

        UseWindow("NOTEPAD.EXE|Edit|Untitled - Notepad|1",1)
                ' Write specified the text into Notepad the specified
                ' number of times
                While (LineNumber <= Count)
                        SendKeys(Text$ + Str$(LineNumber) + "<Enter>")
                        LineNumber = LineNumber + 1
                Wend

        UseWindow("NOTEPAD.EXE|Notepad|Untitled - Notepad",1)
                        ChooseMenu(Normal,"&File|E&xit")

                UseWindow("NOTEPAD.EXE|CtrlNotifySink|Notepad|7",1)
                Click(Button,"&Save")

        ' Save text to the specified filename
        UseWindow("NOTEPAD.EXE|FloatNotifySink|Save As|1",1)
                WriteCombo("1",filename$)

        UseWindow("NOTEPAD.EXE|#32770|Save As",1)
                Click(Button,"&Save")

        ' Confirm overwrite of existing file
        ret = ExistW("NOTEPAD.EXE|#32770|Save As",1)
        Pause 20 ticks
        If (ret = 1) Then
                UseWindow("NOTEPAD.EXE|CtrlNotifySink|Confirm Save As|7",1)
                Click(Button,"&Yes")
```

```
        Endif

EndSub


'=================================================================
'  This function reads the text from the specified file name
'  and writes the contents into WordPad.
'
'  NOTE:  WordPad is not closed, nor are it's contents saved.
'
'  Input Parameters:
'     FileName$       - File name of text file to read
'
'  Returns:
'     Number of lines written to WordPad text area
'=================================================================
Function Fill_Wordpad(FileName$)

        Shell("wordpad",1)

        LinesWritten = 0

        UseWindow("WORDPAD.EXE|RICHEDIT50W|Document - WordPad|1",1)
                While (Eof(FileName$) = 0)
                        ret = Read(FileName$, DataBuffer$, CRLF)
                        SendKeys(DataBuffer$ + "<Enter>")
                        LinesWritten = LinesWritten + 1
                Wend
                SendKeys("Hello Wordpad<Enter>")
                LinesWritten = LinesWritten + 1

        Fill_Wordpad = LinesWritten

EndFunction


'=================================================================
' Main Program
'=================================================================

NotepadFile$ = "text12.txt"
NotepadText$ = "Subroutine and Function Exercise"

' Call subroutine to write 8 lines of text into file "text12.txt"
Fill_Notepad(NotepadFile$, NotepadText$, 8)

' Call function to read contents of "text12.txt" and write them
' into the WordPad text area
Result = Fill_Wordpad(NotepadFile$)

MsgBox("Number of Lines Written to WordPad = " + Str$(Result))
```

Exercise 13, Script13.src

```
'============================================================
'   Script13
'
'   WinTask Training Version 3.8a
'   © Copyright 1997-2012 TaskWare      January 2012
'
'   Exercise 13
'
'   One piece that hasn't been integrated into the previous
'   exercises is an array.  This exercise builds on Exercises
'   10 and 12 to add an array.  If they haven't been completed,
'   please refer to the chapters titled titled "File Functions"
'   and "Subroutines and Functions".  See the WinTask Help
'   System for additional information to complete this exercise.
'============================================================

' Define an array that holds up to 26 strings (zero-based, 0-25)
Dim LineOfText$(25)



'============================================================
'   This subroutine writes the specified text string a number
'   of times to a file.  The line number is appended to the
'   text string written to the Notepad text area.
'
'   Input Parameters:
'      FileName$       - File name to use when writing text
'      Text$           - Text string to write
'      Count           - Number of times Text$ is to be written
'============================================================
Sub Fill_Notepad(FileName$, Text$, Count)

        Shell("notepad",1)

        LineNumber = 1

        UseWindow("NOTEPAD.EXE|Edit|Untitled - Notepad|1",1)
                While (LineNumber <= Count)
                        SendKeys(Text$ + Str$(LineNumber) + "<Enter>")
                        LineNumber = LineNumber + 1
                Wend

        UseWindow("NOTEPAD.EXE|Notepad|Untitled - Notepad",1)
                        ChooseMenu(Normal,"&File|E&xit")

                UseWindow("NOTEPAD.EXE|CtrlNotifySink|Notepad|7",1)
                Click(Button,"&Save")

        UseWindow("NOTEPAD.EXE|Edit|Save As|1",1)
                SendKeys(FileName$)

        UseWindow("NOTEPAD.EXE|#32770|Save As",1)
                Click(Button,"&Save")

        ' Confirm overwrite of existing file
        ret = ExistW("NOTEPAD.EXE|#32770|Save As",1)
        If (ret = 1) Then
                UseWindow("NOTEPAD.EXE|#32770|Save As",1)
                Click(Button,"&Yes")
        Endif
```

```
EndSub


'==================================================================
'   This function reads the text from the specified file name
'   and writes the contents into WordPad.
'
'   NOTE:  WordPad is not closed, nor are its contents saved.
'
'   Input Parameters:
'       FileName$       - File name of text file to read
'
'   Returns:
'       Number of lines written to WordPad text area
'==================================================================
Function Fill_Wordpad(FileName$)

        Shell("wordpad",1)

        LinesWritten = 0

        UseWindow("WORDPAD.EXE|RICHEDIT50W|Document - WordPad|1",1)
                While (Eof(FileName$) = 0)
                        ret = Read(FileName$, DataBuffer$, CRLF)
                        SendKeys(DataBuffer$ + "<Enter>")
                        LinesWritten = LinesWritten + 1
                Wend
                SendKeys("Hello Wordpad<Enter>")
                LinesWritten = LinesWritten + 1

        Fill_Wordpad = LinesWritten

EndFunction


'==================================================================
'   This subroutine reads the text from the specified file
'   name and writes the contents into an Excel spreadsheet.
'   The file contents into a spreadsheet row starting with
'   cell A1 and into a column starting with cell C3.
'
'   Input Parameters:
'       FileName$       - File name of text file to read
'       ExcelFileName$  - File name of Excel spreadsheet to modify
'==================================================================
Sub Fill_Excel(FileName$, ExcelFileName$)

        ' Reset file position to beginning of file
        ' Currently at end-of-file due to Read() calls in Fill_Wordpad
        SetReadPos(FileName$, 0)

        ' Read contents of file into string array
        LinesRead = 0
        While (Eof(FileName$) = 0) AND (LinesRead <= 25)
                ret = Read(FileName$, LineOfText$(LinesRead), CRLF)
                LinesRead = LinesRead + 1
        Wend
```

```
        ' Determine range of row cells based on number of elements in array
        FirstCellCol$ = "A"
        CellCol = Asc(FirstCellCol$) + LinesRead - 1
        LastCellCol$ = Chr$(CellCol)
        XL_RowRangeDesc$ = "Sheet1!" + FirstCellCol$ + "1:" + LastCellCol$ + "1"

        ' Determine range of column cells based on number of elements in array
        FirstCellRow$ = "3"
        CellRow = Val(FirstCellRow$) + LinesRead - 1
        LastCellRow$ = Str$(CellRow)
        XL_ColRangeDesc$ = "Sheet1!C" + FirstCellRow$ + ":C" + LastCellRow$

        WriteExcel(ExcelFileName$, XL_RowRangeDesc$, LineOfText$())
        WriteExcel(ExcelFileName$, XL_ColRangeDesc$, LineOfText$())

EndSub



'=================================================================
' Main Program
'=================================================================

' The following filenames should be converted to full pathnames
NotepadFile$ = "c:\program files (x86)\wintask\scripts\text13.txt"
NotepadText$ = "Sub, Func, Excel Exercise"
ExcelFile$ = "exercise13.xlsx"

' Call subroutine to write 5 lines of text into file "text13.txt"
Fill_Notepad(NotepadFile$, NotepadText$, 5)

' Call function to read contents of "text13.txt" and write them
' into the WordPad text area
Result = Fill_Wordpad(NotepadFile$)

' Call subroutine to read contents of "text13.txt" and write them
' into Excel spreadsheet
Fill_Excel(NotepadFile$, ExcelFile$)

MsgBox("Number of Lines Written to WordPad = " + Str$(Result))

‘Kill wordpad application without saving
KillApp(“wordpad.exe”,1)
```

GLOSSARY

Compiler: The WinTask program that reads a source script file and transforms it's instructions into a form that can be executed. WinTask automation scripts can be compiled by hitting the Play button or selecting the Start/Run menu item before the Executor can execute them.

Dialog Box: A window displayed by a program to gather information from the user in order to perform a task. Dialog Boxes occasionally are used to provide task status to the user.

Editor: The WinTask program that serves as the focal point for the development, compilation and execution of automation scripts. The Editor provides access to Recording mode, the Synchronization wizards and other tools which aid in the creation and maintenance of automation scripts.

Executor: The WinTask program that interprets the automation instructions in a .ROB file. Double-clicking a file with a .ROB extension will invoke the WinTask Executor and begin executing the compiled automation script.

HTML Descriptor: A unique identifier for any HTML object displayed on a Web page. It includes the HTML tag and any meaningful text.

Log: A file that documents the execution and result of script statements during the execution of an automation script. The log file can be reviewed to determine why a particular script failed to run to completion.

Macro: (See Script).

Recording Mode: An operational mode of the WinTask Editor that records a user's keyboard and mouse actions as script statements. Recording mode allows the user to rapidly develop automation scripts.

.ROB: Files with a .ROB extension are created by the WinTask Compiler when compiling an automation script file. The WinTask Executor reads and executes the instructions contained in a .ROB file.

Runtime: A term used to describe the period of time when an automation script is being executed. It is used to distinguish itself from Recording mode or script editing when the user can manually intervene to contend with unexpected circumstances.

Script: A collection of WinTask language script statements, which when compiled and executed, automate an application or process. WinTask script files use the .SRC extension.

Spy: The WinTask program tool that determines the internal Windows Name, a control name, or an HTML descriptor. This tool is needed to distinguish between the numerous elements that may populate the Windows desktop.

.SRC: Files with .SRC extension contain the script statements that provide the instructions to automate a task or process. The WinTask Editor creates and modifies .SRC files. The WinTask Compiler reads .SRC files to produce .ROB files.

Window Name: The unique identifier of a window displayed on desktop, usually the text found in the title bar. The Window Name typically consists of the program name and the name of the document is has open. WinTask uses the Window Name to distinguish between the numerous windows that may populate the desktop.

WinTask Floating Toolbar: A small toolbar displayed during Recording mode. The toolbar provides access to the WinTask Editor functions that the user may need during a recording session. The toolbar provides access to the Spy tool, the Synchronization wizards and a Stop button.

Index

#ActionTimeout....29
#IgnoreErrors....29
#UseExact....30
Arrays....27
ChooseItem....55
ChooseMenu....55
ClickOnBitmap....55
ClickOnText....55
Compilation errors....51
Compiler
  TaskComp....5
Dialog Box Editor....6
Editor
  TaskEdit....5
Eof()....38
Execution error....51
Execution is slow....57
Executor
  TaskExec....6
Exist()....36
ExistW()....31
Focus$()....31
Function/ExitFunction/EndFunction....46
Functions....24
Include....51
Integers....27
Iteration (Loops)....34
Kill()....36
Log file....52
Low level Recording mode....55
Msgbox....51
MsgFrame....51
Object not found error message....53
Operators....27
Read()....37
ReadExcel....40
Real numbers....27
Repeat/Until....34
Spy....6
Strings....27
Sub/ExitSub/EndSub....43
Synchronization - Bitmap....16
Synchronization – Date/Hour....16
Synchronization - Text....15
Synchronization - Time....16
Synchronization - Window....16
Synchronization Wizards....6
System Variables....26
Terminating a script....14
Top$()....31
Variables....26
Wait - Keyboard....16
Wait - Menu....16
Wait – Mouse....17
While/Wend....34
Write()....38
WriteCombo....54
WriteEdit....54
WriteExcel....40