CONTENTS

Programming Reference

Error Handling

INTRODUCTION

WIL (Windows Interface Language) is a powerful, yet easy-to-learn, procedural language, which provides a rich set of functions to Windows users. It can:

- Run Windows and DOS programs.
- Send keystrokes directly to Windows applications.
- Rearrange, resize, hide, and close windows.
- Run programs either concurrently or sequentially.
- Display information to the user in various formats.
- Prompt the user for any needed input.
- Present scrollable file and directory lists.
- Copy, move, delete, and rename files.
- Read and write files directly.
- Copy text to and from the Clipboard.
- Perform string and arithmetic operations.
- Make branching decisions based upon numerous factors.

And much, much more.

About This Online Reference Guide

The WIL Language for Windows is not a stand-alone product. Rather, it is an accessory to a range of Windows applications.

This user's guide is a reference for the WIL language itself, as well as a guide to creating basic WIL programs.

Because the WIL language is used by widely differing applications, it can be used in various ways. This guide covers the general functions common to applications that use WIL. Specific functions and operations are in your specific application's manual.

In all cases, your product-specific documentation supersedes the information provided in this online computer reference.

Users gain access to WIL in two main ways: through menus and by executing batch language progams. In a menu system, WIL commands are defined in one or more **menu files**, each of which is a list of different tasks, or **menu items**. On the other hand, a batch program contains an operation script in a separate **batch file**.

In this manual, we use the term **WIL program** to refer to either an individual menu item or to a complete

batch file.

We will use the term **WIL Interpreter** to refer to that part of your application which is responsible for executing WIL programs.

A **WIL compiler** is a program that turns WIL batch language scripts into programs that can be launched just like standard Windows applications.

Notational Conventions

Throughout this manual, we use the following conventions to distinguish elements of text:

ALL-CAPS

> Used for filenames.

Boldface

> Used for important points, programs, function names, and parts of syntax that must appear as shown.

Acknowledgements

WIL software developed by Morrie Wilson.

Documentation written by Richard Merit.

Online Help organized and compiled by Jim Stiles of The Stiles Group

> (206) 937-8419
>
> Compuserve: 73240,3131, Internet: 73240.3131@compuserve.com

About Online Help

Using Windows Help
There are many little known, but handy, capabilities in Windows help files. You will get a good overview of these by selecting the Help, How to use Help menu item from the WIL Language Reference main menu.

Linked References. Hypertext.

This help system includes most of the WIL manual. Most key terms are linked to examples and definitions. This is one use of hypertext--computers used for rapid cross-linking of topics. You can click on the highlighted and underlined text and get more information. Then you can quickly return to your original topic.

Changing display colors of reference topics.

On many monitors, the help crosslinks are displayed in a green that can be difficult to read. You can change the color.

To do this, you will need to edit your WIN.INI file: Add these lines:

```
[Windows Help]

Jumpcolor=0 0 255

Popupcolor=255 0 0
```

You can run the program SYSEDIT.EXE located in your Windows system directory to do this. The zeros in the lines have spaces separating them from the other numbers there.

The above setting will give you blue hot jump keywords and red popup definitions in all your help applications, not just WIL Online Reference.

Searching for topics.

The subject areas in WIL Online Reference are linked. If you want to view all the commands in one subject area, network functions is one example, you first use the "SEARCH" function on the toolbar to look for any single network function. Then, once that is on your display, use the "BROWSE" toolbar button to view related commands.

Returning to previous topics.

The "Back" feature is convenient. With it, you can safely backtrack to your initial spot after a side trip. Windows 3.1 lets you view a history of where you have been in WIL Online Reference. You get a list of the topics you have found and can easily click on them to return to specific ones.

Making annotations for single or group use.

The WIL Online Reference has an "annotation" feature that is useful on an individual, or even a company-wide basis. You can make specific comments or even add your own code samples to individual sections in the WIL Online Reference. An icon of a paperclip appears in the upper left corner of topics that you annotate. The annotations are kept in a file called WILHELP.ANN in the Windows directory. This file can be copied between computers to exchange these annotations.

Using bookmarks to save time.

The WIL Online Reference also has a "bookmark" feature that works from its main menu. You will probably find that you look some items up more frequently than others. You can set a bookmark by these and then access the bookmark location from the Bookmark main menu item.

Copying Code Samples.

The examples in the function descriptions are good sources of useful command scripts. The Windows 3.1 version of WINHELP lets you use the Edit Copy function to select specific lines for copying to the clipboard. In this way, you can copy the code without having to remove the surrounding descriptive text that you don't need.

Most of the examples include cross references to commands. The formatting does not copy into your scripts.

About DDE

Using DDE

DDE is the means Windows uses to send data between programs. Those who use the WIL language have at hand a particularly easy way to use it. With DDE, you can control programs with both the WIL language, and with the macro language in the target program. WIL is particularly good at displaying dialog boxes, arranging windows, and working with programs from different manufacturers. The macro languages found in many programs are, of course, well suited to the strengths of those applications.

Your WIL scripts can include both WIL statements, and the statements and functions of a program's macro language. For instance, you could use WIL to display a dialog asking for input. Then your script could combine that user input with your program's macro statements. This package is sent to you application via DDE.

Using the WIL language for sending macro commands has the advantage of being one source for instructions useful in many tasks. Be using WIL to feed macro instructions into your word processor, you can use these macros without needing to load a particular template. If the product you use to access WIL compiles scripts into Windows programs, your macros will be securely hidden and protected from alteration.

DDE—Step by Step

1. The first step to get DDE running is the opening of a channel of inter-program data exchange. This is much simpler from WIL than from any application. The WIL statement, AppExist, easily tests for a loaded program without your needing to reference the "class" of the program or other obscure traits.

The "FileLocate" statement can find a program that has not been loaded, and the "Run" statement can load it. If you will need to use a particular file with that applicaiton, you can specify this with the "Run" command. With WIL you can even display a list of files and choose one for loading with your application. DirChange and DirItemize statements can do this.

2. The second step is obtaining a channel number for every DDE link you want to set up. Every program is fussy about this, and they all are slightly different. You will need to specify a server name for the application, and a file name for the data file that you need to access.

3. The third step is to send to the server requests for information (using DDERequest) or macro instructions (DDEExecute).

4. Finally, when you have concluded your session, you will need to terninate the communications with a "DDETerminate" instruction.

Here is an example:

```
If AppExist("excel.exe") == @TRUE Then GoTo excelrunning
Run("e:\xl\excel.exe", "/E")
channel1 = DDEInitiate("excel", "System")
DDETimeout(20000)
If channel1 == 0 Then Goto failed
DDEEXECUTE(channel1,"[OPEN(""d:\temp\sales.xls"")]")
DDETerminate(channel1)

:excelrunning
channel2 = DDEInitiate("excel" , "sales.xls")
item = "R1C1"
```

```
output = DDERequest(channel2,item)
Message("January Results", output)

DDETerminate(channel2)
WinClose("Microsoft Excel")
If output == "" Then Goto Failed

Exit

:failed
Message("DDE operation unsuccessful", "Check your syntax")
```

Here is the explanation:

The first line loads Excel if it isn't already loaded. The /E causes Excel to load without its standard spreadsheet "sheet1.xls". Where this simple task is frustrating in other environments, the WIL language makes it easy.

The first channel is opened with the Excel server name "excel" and the generic topic of "system". In this way Excel's macros work without having a specific topic. In Excel's case, a specific topic is a file name, and, of course, you don't have one because you haven't yet loaded a spreadsheet file.

The DDEExecute command uses channel1 to run the Excel macro. Note the use of repeated double quotes. You can also string a long series of commands together. They must fit between brackets like this:

"[xxx] [xxx] [xxx] [xxx] [xxx] [xxx] [xxx] [xxx] [xxx] "

The first channel cannot be used to get your data from Excel. For this you need a specific link, the generic "system" won't do. The channel2 connection does this by using the name of an active spreadsheet, "sales.xls", as the topic. The item you want must be requested once the channel is open. It can be either a cell address, or a range name. In the example here it is simply the first cell in the spreadsheet, R1C1. Excel needs this notation for DDE to get data from specific cells or ranges of cells.

The WinClose statement closes Excel. If you wanted quicker data updating, you would delete this statement and leave Excel open.

You can discover more about DDE by clicking the browse buttons to get a tour of the WIL DDE functions. Most Windows programs do support DDE, but information on this may need to be gleaned from product support people. Sadly, reference manuals offer scanty information on this important capability.

MENU FILES

Some applications have the special capability of using WIL language commands from menus. This section applies to them. If you are using a batch file-based implementation of WIL, you can skip this section.

About Menu Files

This section of the manual shows how to create menu files with WIL. It is presented here so that you will be able to follow along with the tutorial material which follows. It is not important at this point to understand the actual commands which are shown in the menus. If you are curious, the colored and underlined items are hot.

Hot items can be used to jump to the topics they represent. Either a double mouse click or a Tab Enter keystroke sequence jumps to the topic. Clicking the Back button on the help toolbar lets you return easily. Alt B is the keystroke for this. There is much more to WIL Online Reference. See **Help** for the particulars.

Menu File Structure

WIL menus are defined in standard ASCII text files (the kind created by Notepad). See your product documentation for the name of the default menu file that it uses.

Every menu file contains one or more **menu items** which appear in drop-down menus. They may also contain top-level menu names which show up in a main menu bar (refer to your product documentation for more information).

Each menu item consists of a **title** which identifies the item, followed by one or more lines of menu **code.** The WIL Interpreter will execute these when you choose the item.

Your application probably included a pre-defined sample menu, and you should refer to it as a practical example of correct menu structure.

Here is an extremely simple menu file:

```
&Games
 &Solitaire
   Run("sol.exe", "")
```

The first line, **&Games**, begins in column 1, and therefore defines a top-level menu item. Depending on the product you are using, it may either appear on a menu bar or it may appear on the first-level drop-down menu. The ampersand (**&**) is optional; it defines an Alt-key combination for the entry (Alt-G in this example). It will appear in the menu as **Games.**

The second line, **&Solitaire**, begins in column 2, and defines the **title** for an individual menu item. Again, the ampersand (**&**) is optional, and defines an Alt-key combination of Alt-S. This item will appear in the menu as **Solitaire.**

The third line, **Run("sol.exe", "")**, is the actual code which will be executed when this menu item is selected. Like all menu code, it must be indented at least four spaces (i.e., it must begin in column 5 or higher).

To see more about the RUN function, just double click on it. Use the BACK toolbar button to return here.

This third line is really the entire **WIL program**; the two lines above it are simply titles which define the

position of the program (i.e., the menu item) in the overall menu structure.

Here's a slightly expanded version of the program:

```
&Games
 &Solitaire
    Display(1, "Game Time", "About to play Solitaire")
    Run("sol.exe", "")
```

We've simply added a line of code, changing this into a two-line program. Notice that each additional line of code is still indented the same four spaces.

Now, let's look at a menu file which contains two menu items:

```
&Games
 &Solitaire
    Run("sol.exe", "")
 &Minesweeper
    Run("winmine.exe", "")
```

We've added a new menu item, **Minesweeper**, which begins in column 2 (like **Solitaire**) and will appear under the top-level menu item **Games** (like **Solitaire**).

To add a new top-level menu item, just create a new entry beginning in column 1:

```
&Games
 &Solitaire
        Run("sol.exe", "")
 &Minesweeper
        Run("winmine.exe", "")

&Applications
 &Notepad
        Run("notepad.exe", "")
 &WinEdit
        Run("winedit.exe", "")
```

Now there are two top-level menu titles, **Games** and **Applications**, each of which contains two individual items (the blank line between **Games** and **Applications** is not necessary, but is there just for readability).

In addition to top-level menus, you can optionally define one or two levels of **submenus**. The titles for the first-level and second-level submenus must begin in columns 2, and 3, respectively, and the individual menu items they contain must be indented one additional column.

For example:

```
&Applications
 &Editors
   &Notepad
        Run("notepad.exe", "")
   &WinEdit
        Run("winedit.exe", "")

 &Excel
        Run("excel.exe", "")
```

In the above example, **Editors** is a submenu (which begins in column 2), which contains two menu items (which begin in column 3). **Excel** also begins in column 2, but since it does not have any submenus defined below it, it is a **bottom-level** (i.e., individual) menu item.

Here's an even more complex example:

```
&Applications
 &Editors
  &Notepad
        Run("notepad.exe", "")
  &WinEdit
        Run("winedit.exe", "")

 |&Spreadsheets
  &Windows-based
   &Excel
        Run("excel.exe", "")

  _&DOS-based
   &Quattro
        Run("q.exe", "")
```

We've added an additional level of submenus under **Spreadsheets**, so that the bottom-level menu items (**Excel** and **Quattro**) now begin in column 4. There are also two special symbols presented in this menu: the underscore (**_**), which causes a horizontal separator line to be drawn above the associated menu title, and the vertical bar (**|**), which causes the associated menu title to appear in a new column.

It is possible to place an individual (bottom-level) menu item in column 1:

```
&Notepad
        Run("notepad.exe", "")
```

in which case it will appear on the top-level menu, but will be executed immediately upon being selected (i.e., there will be no drop-down menu).

Modifying Menus

As stated above, menu files must be created and edited with an editor, such as **Notepad**, that is capable of saving files in pure ASCII text format.

After you have edited your menu, it must be **reloaded** into memory for the changes to take effect. You may be able to do this manually, via the application's control menu (see your product documentation for information). Or, you can have a menu item use the **Reload** function. Otherwise, the menus will be reloaded automatically the next time you execute any menu item. However, if the menus are reloaded automatically, the WIL Interpreter will not be able to determine which menu item you had just selected, and it will therefore display a message telling you that you need to re-select it.

Menu Hotkeys

In addition to the standard methods for executing a menu item (double-clicking on it, highlighting it and pressing Enter, or using Alt + the underlined letter), you may be able to define optional **hotkeys** for your menu items (depending on the implementation of WIL in the product you are using), which will cause an item to be executed immediately upon pressing the designated hot key.

Hotkeys are defined by following the menu item with a backslash (\) and then the hotkey:

```
&Accessories
 &Notepad   \   (F2)
          Run("notepad.exe", "")
 &Calculator   \   ^C
          Run("calc.exe", "")
```

In the above example, the F2 key is defined as the hotkey for Notepad, and Ctrl-C is defined as the hotkey for Calculator.

Most single keys and key combinations may be used as hotkeys, except for the F10 key, and except for Alt and Alt-Shift key combinations (although you may use AltCtrl key combinations). Refer to the **SendKey** function for a list of special keycodes which may also be used as hot keys.

If you always access a menu item by using its hotkey, you may not need or want the menu item to appear in the pull-down menus. If so, you can make it a non-displayed menu item by placing a **@** symbol in front of the title. For example:

```
&Accessories
 @Notepad   \   (F2)
          Run("notepad.exe", "")
```

In this case, Notepad would not appear in the pull-down menus, but could still be accessed by using the F2 hotkey.

Note: Hotkeys and non-displayed menu items may not work in all implementations of the WIL Interpreter.

WIL TUTORIAL

WIL Basics

What is a WIL Program?

A WIL program, like a DOS batch file, is simply a list of commands for the computer to process. Any task which will be run more than once, or which requires entering multiple commands or even a single complex command, is a good candidate for automation as a WIL program. For example, suppose you regularly enter the following commands to start Windows:

First:

```
cd\windows
```

then:

```
win
```

and then:

```
cd\
```

Here, you are changing to the Windows directory, running Windows, and then returning to the root directory. Instead of having to type these three commands every time you run Windows, you can create a DOS batch file, called WI.BAT, which contains those exact same commands:

```
cd\windows
win
cd\
```

Now, to start Windows, you merely need to type the single command **WI**, which runs the WI.BAT batch file, which executes your three commands.

WIL programs work basically the same way.

Our First WIL Program

Our first WIL program will simply run a favourite Windows application: Solitaire. If you are using a batch file-based implementation of the WIL Interpreter, you will be creating your batch files using an editor, such as **Notepad**, that is capable of saving text in pure ASCII format. Let's create a WIL program containing the following line of text:

```
Run("sol.exe", "")
```

Save the program, and run it (refer to your product documentation for information on how to execute a WIL program). Presto! It's Solitaire.

Functions and Parameters

Now, let's look more closely at the line we entered:

```
Run("sol.exe", "")
```

The first part, **Run**, is a WIL function. As you might have guessed, its purpose is to run a Windows program. There are a large number of functions and commands in WIL, and each has a certain syntax which must be used. The correct syntax for all WIL functions may be found in the **WIL Function Reference**. The entry for **Run** starts off as follows:

Syntax:

Run (program-name**,** parameters**)**

Parameters:

(s) program-name = the name of the desired.**EXE**,.**COM**,.**PIF,**.**BAT** file, or a data file.

(s) parameters = optional parameters as required by the application.

Like all WIL functions, **Run** is followed by a number of **parameters**, enclosed in parentheses. Parameters are simply additional pieces of information which are provided when a particular function is used; they may be either required or optional. Optional parameters are indicated by being enclosed in square brackets. In this case, **Run** has two required parameters: the name of the program to run, and the parameters to be passed to the program.

WIL functions use several types of parameters. Multiple parameters are separated by commas. In the example

```
Run("sol.exe", "")
```

"sol.exe" and **""** are both **string constants**. String constants can be identified by the quote marks which **delimit** (surround) them. You may use either double (**"**), single forward (**'**) or single back (`) quote marks as string **delimiters**; the examples in this manual will use double quotes.

Note: In our shorthand method for indicating syntax the **(s)** in front of a parameter indicates that it is a string parameter.

You may have noticed how we said earlier that the two parameters for the **Run** function are *required*, and yet the entry for **Run** in the WIL Function Reference describes the second parameter – **"parameters" –** as being *optional*. Which is correct? Well, from a language standpoint, the second parameter is required. That is, if you omit it, you will get a syntax error, and your WIL program will not run properly. However, the program that you are running may not need any parameters. Solitaire, for example, does not take any parameters. The way we handle this in our programs is to specify a **null string** – two quote marks with nothing in between – as the second parameter, as we have done in our example above.

To illustrate this further, let's create a WIL program containing the following line:

```
Run("notepad.exe", "")
```

This is just like our previous file, with only the name of the program having been changed. Save the file, and run it. You should now be in Notepad. Now, edit the WIL program as follows:

```
Run("notepad.exe", "c:\autoexec.bat")
```

Save the program, exit Notepad, and run the WIL program again. You should now be in Notepad, with AUTOEXEC.BAT loaded. As we've just demonstrated, Notepad is an example of a program which can be run with or without a file name parameter passed to it by WIL.

It can often be helpful to add descriptive text to your WIL programs:

```
; This is an example of the Run function in WIL
Run("notepad.exe", "c:\autoexec.bat")
```

The semicolon at the beginning of the first line signifies a **comment**, and causes that line to be ignored. You can place comment lines, and/or blank lines anywhere in your WIL programs. In addition, you can place a comment on the same line as a WIL statement by preceding the comment with a semicolon. For example:

```
Run("sol.exe", "")        ; this is a very useful function
```

Everything to the right of a semicolon is ignored. However, if a semicolon appears in a string delimited by quotes, it is treated as part of the string.

Displaying Text

Now, let's modify our original WIL program as follows:

```
; solitare.program
Display(5, "Good Luck!", "Remember... it's only a game.")
Run("sol.exe", "")
```

And run it. Notice the message box which pops up on the screen with words of encouragement:

That's done by the **Display** function in the second line above. Here's the reference for **Display**:

Syntax:

Display (seconds**,** title**,** text**)**

Parameters:

(i) seconds = seconds to display the message (1-3600).

(s) title = Title of the window to be displayed.

(s) text = Text of the window to be displayed.

Note that the **Display** function has three parameters. The first parameter – in our example, 5 – is the number of seconds which the display box will remain on the screen (you can make the box disappear before then by pressing any key or mouse button). This is a **numeric constant**, and – unlike string constants – it does not need to be enclosed in quotes (although it can be, if you wish, as WIL will automatically try to convert string variables to numeric variables when necessary, and vice versa). The second parameter is the title of the message box, and the third parameter is the actual text displayed in the box.

Note: In our shorthand method for indicating syntax the **(i)** in front of a parameter indicates that it is a string parameter.

Now, exit Solitaire (if you haven't done so already), and edit the WIL program by placing a semicolon at the beginning of the line with the Run function. This is a handy way to disable, or "comment out," lines in your WIL programs when you want to modify and test only certain segments. Your WIL program should look like this:

```
; solitare.program
Display(5, "Good Luck!", "Remember... it's only a game.")
; Run("sol.exe", "")
```

Now, experiment with modifying the parameters in the **Display** function. Try adjusting the value of the first parameter. If you look up **Display** in the WIL reference section, you will see that the acceptable values for this parameter are **13600**. If you use a value outside this range, WIL will adjust it to "make it fit"; that is, it will treat numbers less than 1 as if they were **1**, and numbers greater than 3600 as **3600**. Also, try using a non-integer value, such as 2.5, and see what happens (you should receive an error message). Play around with the text in the two string parameters; try making one, or both, null strings (**""**).

Getting Input

Now, let's look at ways of getting input from a user and making decisions based on that input. The most basic form of input is a simple Yes/No response, and, indeed, there is a WIL function called **AskYesNo**:

Syntax:

AskYesNo (title**,** question**)**

Parameters

(s) title = title of the question box.

(s) question = question to be put to the user.

Returns:

(i) **@YES** or **@NO**, depending on the button pressed.

You should be familiar with the standard syntax format by now; it shows us that **AskYesNo** has two required parameters. The **Parameters** section tells us that these parameters both take strings, and tells us what each of the parameters means.

You will notice that there is also a new section here, called **Returns**. This section shows you the possible values that may be returned by this function. *All* functions return values. We weren't concerned with the values returned by the **Run** and **Display** functions. But with **AskYesNo**, the returned value is very important, because we will need that information to decide how to proceed. We see that **AskYesNo** returns an **integer** value. An integer is a whole (non-fractional) number, such as 0, 1, or 2 (the number 1.5 is *not* an integer).

We also see that the integer value returned by **AskYesNo** is either **@YES** or **@NO**. @YES and @NO are **predefined constants** in WIL. All predefined constants begin with an **@** symbol, and we will distinguish them further by typing them in all caps. You will find a list of all predefined constants in **Appendix A** (pg. **Error! Bookmark not defined.**).

Even though the words **Yes** and **No** are strings, it is important to remember that the predefined constants **@YES** and **@NO** are *not* string variables. (Actually, @YES is equal to 1, and @NO is equal to 0. Don't worry if this is confusing; you really don't need to remember or even understand it.)

Now, let's modify our WIL program as follows:

```
AskYesNo("Really?", "Play Solitaire now?")
Run("sol.exe", "")
```

and run it. You should have gotten a nice dialog box which asked if you wanted to play Solitaire:

but no matter what you answered, it started Solitaire anyway. This is not very useful. We need a way to use the Yes/No response to determine further processing. First, we need to explore the concept and use of **variables**.

Using Variables

A **variable** is simply a placeholder for a value. The value that the variable stands for can be either a text string (**string variable**) or a number (**numeric variable**).

You may remember from Algebra 101 that if *X=3*, then *X+X=6*. *X* is simply a numeric variable, which stands here for the number *3*. If we change the value of *X* to *4* (*X=4*), then the expression *X+X* is now equal to *8*.

Okay. We know that the **AskYesNo** function returns a value of either @YES or @NO. What we need to do is create a variable to store the value that **AskYesNo** returns, so that we can use it later on in our WIL program.

First, we need to give this variable a name. In WIL, variable names must begin with a letter, may contain any combination of letters or numbers, and may be from 1 to 30 characters long. So, let's use a variable called **response**. (We will distinguish variable names in this text by printing them in all lowercase letters; we will print function and command names starting with a capital letter.

However, in WIL, the case is not significant, so you can use all lowercase, or all uppercase, or whatever combination you prefer.) We assign the value returned by **AskYesNo** to the variable **response**, as follows:

```
response = AskYesNo("Really?", "Play Solitaire now?")
```

Notice the syntax. The way that WIL processes this line is to first evaluate the result of the **AskYesNo** function. The function returns a value of either @YES or @NO. Then, WIL assigns this returned value to **response**. Therefore, **response** is now equal to either @YES or @NO, depending on what the user enters.

Now, we need a way to make a decision based upon this variable.

Making Decisions

WIL provides a way to conditionally execute a statement, and that is by using the **If**... **Then** command. Actually, there are two separate parts to this construct: **If** and **Then**. The format is:

If condition **Then** statement

(We refer to **If**... **Then** as a **command**, rather than a **function**, because functions are followed by parameters in parentheses, while commands are not. Commands are used for system control.)

The use of **If**... **Then** can easily be illustrated by going back to our WIL program and making these modifications:

```
response = AskYesNo("Really?", "Play Solitaire now?")
If response == @YES Then Run("sol.exe", "")
```

In this example, we are using **If**... **Then** to test whether the value of the variable **response** is @YES. If it *is* @YES, we start Solitaire. If it *isn't* @YES, we don't.

The rule is: if the condition following the **If** keyword is true, then the statement following the **Then** keyword is performed. If the condition following the **If** keyword is false, then anything following the **Then** keyword is ignored.

There is something extremely important that you should note about the syntax of the **If**... **Then** command: the double equal signs (**==**). In WIL, a single equal sign (**=**) is an **assignment operator** – it assigns the value on the right of the equal sign to the variable on the left of the equal sign. As in:

```
response = AskYesNo("Really?", "Play Solitaire now?")
```

This is saying, in English: "Assign the value returned by the **AskYesNo** function to the variable named **response**." But in the statement:

```
If response == @YES Then Run("sol.exe", "")
```

we do *not* want to assign a new value to **response**, we merely want to test whether it is equal to @YES. Therefore, we use the double equal signs (**==**), which is the **equality operator** in WIL. The statement above is saying, in English: "If the value of the variable named **response** is equal to @YES, then run the program SOL.EXE." If you used a single equal sign (**=**) here by mistake, you would get an error message:

Which is WIL's way of telling you to re-check your syntax.

If you've become confused, just remember that a single equal sign (**=**) is an assignment operator, used to assign a value to a variable. Double equal signs (**==**) are an equality operator, used to test whether the values on both sides of the operator are the same.

If you have a problem with one of your WIL programs, make sure to check whether you've used one of these symbols incorrectly. It's a very common mistake, which is why we emphasize it so strongly!

We've seen what happens when the condition following the **Then** keyword is true. But what happens when it is false? Remember we said that when the **If** condition is false, the **Then** statement is ignored.

There will be times, however when we want to perform an alternate action in this circumstance. For example, suppose we want to display a message if the user decides that he or she *doesn't* want to play Solitaire. We could write:

```
response = AskYesNo("Really?", "Play Solitaire now?")
If response == @YES Then Run("sol.exe", "")
If response == @NO Then Display(5, "", "Game canceled")
```

In this case there are two **If** statements being evaluated, with one and *only* one of them possibly being true. However, this is inefficient from a processing standpoint. Furthermore, what would happen if you had several functions you wanted to perform if the user answered **Yes**? You would end up with something unwieldy:

```
response = AskYesNo("Really?", "Play Solitaire now?")
If response == @YES Then Display(5, "", "On your mark...")
If response == @YES Then Display(5, "", "Get set...")
If response == @YES Then Display(5, "", "Go!")
If response == @YES Then Run("sol.exe", "")
```

Clearly, there must be a better way of handling this.

Branching

Enter the **Goto** command. **Goto**, in combination with **If**... **Then**, gives you the ability to redirect the **flow of control** in your WIL programs. **Goto** does exactly what it says – it causes the flow of control to go to another point in the WIL program.

You must specify where you want the flow of control to be transferred, and you must mark this point with a **label**. A label is simply a destination address, or marker. The form of the **Goto** command is:

Goto label

where **label** is an identifier that you specify. The same rules apply to label names as to variable names (the first character must be a letter, the label name may consist of any combination of letters and numbers, and the label name may be from 1 to 30 characters long).

In addition, the label is preceded by a colon (**:**) at the point where it is being used as a destination address. Here's an example:

```
response = AskYesNoIf response == @NO Then Goto quit
Display(5, "", "On your mark...")
Display(5, "", "Get set...")
Display(5, "", "Go!")
Run("sol.exe", "")
:quit
```

If the **If** condition is true (that is, if the user answers **No**), then the **Goto** statement is performed. The **Goto** statement is saying, in English "go to the line marked **:quit**, and continue processing from there."

Notice how the label **quit** is preceded by a colon on the last line, but *not* on the line where it follows the **Goto** keyword. This is important. Although you can have multiple lines in your WIL program which say **Goto quit**, you can have only one line marked **:quit** (just like you can have several people going to your house, but can have only one house with a particular address).

Of course, you can use many different labels in a WIL program, just as you can use many different variables, as long as each has a unique name. For example:

```
response = AskYesNo("Really?", "Play Solitaire now?")
If response == @NO Then Goto quit
Display(5, "", "On your mark...")
Display(5, "", "Get set...")
Display(5, "", "Go!")
Run("sol.exe", "")
Goto done
:quit
Display(5, "", "Game canceled")
:done
```

This is a little more complicated. It uses two labels, **quit** and **done**. If the user answers **No**, then the **If** condition is true, control passes to the line marked **:quit**, and a message is displayed.

If, on the other hand, the user answers **Yes**, then the **If** condition is false, and the **Goto quit** line is ignored. Instead, the next four lines are processed, and then the **Goto done** statement is performed.

The purpose of this line is to bypass the **Display** line which follows, by transferring control to the end of the WIL program.

There is another way to keep the processing from "falling through" to unwanted lines at the end of a program, and that is with the **Exit** command. **Exit** causes a WIL program to end immediately. So, for example, we could rewrite the above WIL program as follows:

```
response = AskYesNo("Really?", "Play Solitaire now?")
If response == @NO Then Goto quit
Display(5, "", "On your mark...")
Display(5, "", "Get set...")
Display(5, "", "Go!")
Run("sol.exe", "")
Exit
:quit
Display(5, "", "Game canceled")
```

Since the **Run** function is the last thing we want to do if the user answers **Yes**, the **Exit** command simply ends the program at that point. Note that we could put an **Exit** command at the end of the program as well, but it isn't necessary. An **Exit** is implied at the end of a WIL program.

This concludes the first part of our tutorial. You now have the building blocks you need to create useful WIL programs. In the second part, which follows, we will look in more detail at some of the WIL functions which are available for your use.

Exploring WIL

What follows is just a sample of the many functions and commands available in WIL. These should be sufficient to begin creating versatile and powerful WIL programs.

Running Programs

There are three functions which you can use to start an application, each of which shares a common syntax:

Run (program-name, parameters)

We've already seen the **Run** function. This function starts a program in a "normal" window. Windows, or the application itself, decides where to place the application's window on the screen.

Example:

```
Run("notepad.exe", "myfile.txt")
```

If the program has an EXE extension, its extension may be omitted:

```
Run("notepad", "myfile.txt")
```

Also, you can "run" data files if they have an extension in WIN.INI which is associated with an executable program. So, if TXT files are associated with Notepad:

```
Run("myfile.txt", "")
```

would start Notepad, using the file MYFILE.TXT.

When you specify a file to run, WIL looks first in the current directory, and then in the directories on your DOS path. If the file is not found, WIL will return an error. You can also specify a full path name for WIL to use, as in:

```
Run("c:\windows\apps\winedit.exe", "")
```

RunZoom (**program-name**, **parameters**)

RunZoom is like **Run**, but starts a program as a full-screen window.

Example:

RunZoom
RunIcon (**program-name**, **parameters**)

RunIcon starts a program as an icon at the bottom of the screen.

Example:

Display and Input

Here we have functions which display information to the user and prompt the user for information, plus a couple of relevant system functions.

Display **(**seconds**,** title**,** text**)**

Displays a message to the user for a specified period of time. The message will disappear after the time expires, or after any keypress or mouse click.

Example:

```
Display(2, "", "Loading Solitaire now")
```

Message **(**title**,** text**)**

This command displays a message box with a title and text you specify, which will remain on the screen until the user presses the **OK** button.

Example:

```
Message("Sorry", "That file cannot be found")
```

Pause **(**title**,** text**)**

This command is similar to Message, except an exclamation-point icon appears in the message box, and the user can press **OK** or **Cancel**. If the user presses **Cancel**, the WIL program ends (or goes to the label **:cancel**, if one is defined).

Example:

```
Pause("Delete Backups", "Last chance to stop!")
; if we got this far, the user pressed OK
FileDelete("*.bak")
```

AskYesNo **(**title**,** question**)**

Displays a dialog box with a given title, which presents the user with three buttons: **Yes**, **No**, and **Cancel.** If the user presses **Cancel**, the WIL program ends (or goes to the label **:cancel**, if one is defined). Otherwise, the function returns a value of @YES or @NO.

Example:

```
response = AskYesNo("End Session", "Really quit Windows?")
```

AskLine **(**title**,** prompt**,** default**)**

Displays a dialog box with a given title, which prompts the user for a line of input. Returns the default if the user just presses the **OK** button.

Example:

```
yourfile = AskLine("Edit File", "Filename:", "newfile.txt")
Run("notepad", yourfile)
```

If you specify a **default** value (as we have with NEWFILE.TXT), it will appear in the response box, and will be replaced with whatever the user types. If the user doesn't type anything, the default is used.

Beep

Beeps once.

```
Beep
```

And if *one* beep isn't enough for you:

```
Beep
Beep
Beep
```

Delay**(**seconds**)**

Pauses WIL program execution.

The Delay function lets you suspend processing for a fixed period of time, which can be anywhere from 1 to 3600 seconds.

Manipulating Windows

There are a large number of functions which allow you to manage the windows on your desktop. Here are some of them:

WinZoom **(**partial-windowname**)**

Maximizes an application window to full-screen.

WinIconize **(**partial-windowname**)**

Turns an application window into an icon.

WinShow **(**partial-windowname**)**

Shows a window in its "normal" state.

These three functions are used to modify the size of an already-running window. **WinZoom** is the equivalent of selecting **Maximize** from a window's control menu, **WinIconize** is like selecting **Minimize**, and **WinShow** is like selecting **Restore**.

The window on which you are performing any of these functions does not have to be the active window. If the specified window is in the background, and a **WinZoom** or **WinShow** function causes the size of the window to change, then the window will be brought to the foreground. The **WinZoom** function has no effect on a window which is already maximized; likewise, **WinShow** has no effect on a window which is already "normal."

Each of these functions accepts a **partial windowname** as a parameter. The windowname is the name which appears in the title bar at the top of the window. You can specify the full name if you wish, but it may often be advantageous not to have to do so. For example, if you are editing the file SOLITARE.WBT in a Notepad window, the windowname will be **Notepad - SOLITARE.WBT**:

You probably don't want to have to hard-code this entire name into your WIL program as:

```
WinZoom("Notepad - SOLITARE.WBT")
```

Instead, you can specify the partial windowname "Notepad":

```
WinZoom("Notepad")
```

If you have more than one Notepad window open, WIL will use the one which was most recently used or started.

Note that WIL matches the partial windowname beginning with the first character, so that while

```
WinZoom("Note")
```

would be correct,

```
WinZoom("pad")
```

would not result in a match.

Also, the case (upper or lower) of the title is significant, so

```
WinZoom("notepad")
```

would not work either.

WinActivate **(**partial-windowname**)**

Makes an application window the active window.

This function makes a currently-open window the active window. If the specified window is an icon, it will be restored to normal size; otherwise, its size will not be changed.

WinClose **(**partial-windowname**)**

Closes an application window.

This is like selecting **Close** from an application's control menu. You will still receive any closing message(s) that the application would normally give you, such as an "unsaved-file" dialog box.

WinCloseNot (partial-windowname
[, partial-windowname**]...)**

Closes all application windows except those specified.

This function lets you close all windows *except* the one(s) you want to remain open. For example:

```
WinCloseNot("Program Man")
```

would leave only the Program Manager open, and:

```
WinCloseNot("Program Man", "Solit")
```

would leave the Program Manager and Solitaire windows open.

WinWaitClose **(**partial-windowname**)**

Waits until an application window is closed.

This function causes your WIL program to pause until you have manually closed a specified window. This is a convenient way to get WIL to open several windows sequentially, thereby not having unnecessary windows all over your desktop. For example:

```
RunZoom("invoices.xls", "")                ;balance the books
WinWaitClose("Microsoft Ex")               ;wait till Excel closed
RunZoom("sol", "")                         ;you deserve a break
WinWaitClose("Solitaire")                  ;wait until Solit closed
Run("winword", "agenda.doc")               ;more paperwork
```

```
WinWaitClose("Microsoft Wor")          ;wait until W4W closed
Run("clock","")                        ;lunchtime yet?
```

WinExist **(**partial-windowname**)**

Tells if a window exists.

This function returns @TRUE or @FALSE, depending on whether a matching window can be found. This provides a way of insuring that only one copy of a given window will be open at a time.

If you've been following this tutorial faithfully from the beginning, you probably have several copies of Solitaire running at the moment. (You can check by pressing **Ctrl-Esc** and bringing up the Task Manager. You say you've got *five* Solitaire windows open? Okay, close them all.) Now, let's modify our WIL program. First, trim out the excess lines so that it looks like this:

```
Run("sol.exe", "")
```

Now, let's use the **WinExist** function to make sure that the WIL program only starts Solitaire if it isn't already running:

```
If WinExist("Solitaire") == @FALSE Then Run("sol.exe", "")
```

And this should work fine. Run the WIL program twice now, and see what happens. The first time you run it, it should start Solitaire; the second (and subsequent) time, it should not do anything.

However, it's quite likely that you want the WIL program to do *something* if Solitaire is already running – namely, bring the Solitaire window to the foreground. This can be accomplished by using the **WinActivate** function, along with a couple of **Goto** statements:

```
If WinExist("Solitaire") == @FALSE Then Goto open
WinActivate("Solitaire")
Goto loaded
:open
Run("sol.exe", "")
:loaded
```

Note that we can change this to have WinExist check for a **True** value instead, by modifying the structure of the WIL program:

```
If WinExist("Solitaire") == @TRUE Then Goto activate
Run("sol.exe", "")
Goto loaded
:activate
WinActivate("Solitaire")
:loaded
```

Either format is perfectly correct, and the choice of which to use is merely a matter of personal style. The result is exactly the same.

EndSession**( )**

Ends the current Windows session.

This does exactly what it says. It will not ask any questions (although you will receive any closing messages that your currently-open windows would normally display), so you may want to build in a little

safety net:

```
sure = AskYesNo("End Session", "Really quit Windows?")
If sure == @YES Then EndSession()
```

EndSession is an example of a WIL function which does not take any parameters, as indicated by the empty parentheses which follow it. The parentheses are still required, though.

Files and Directories

DirChange (pathname**)**

Changes the directory to the pathname specified.

Use this function when you want to run a program which must be started from its own directory. "Pathname" may optionally include a drive letter.

Example:

```
DirChange("c:\windows\winword")
Run("winword.exe", "")
```

DirGet ( )

Gets the current working directory.

This function is especially useful in conjunction with **DirChange**, to save and then return to the current directory.

Example:

```
origdir = DirGet()
DirChange("c:\windows\winword")
Run("winword.exe", "")
DirChange(origdir)
```

FileExist **(**filename**)**

Determines if a file exists.

This function will return @TRUE if the specified file exists, and @FALSE if it doesn't exist.

Example:

```
If FileExist("win.bak") == @FALSE Then FileCopy("win.ini",    "win.bak", @FALSE)
Run("notepad.exe", "win.ini")
```

FileCopy **(**from-list**,** to-file**,** warning**)**

Copies files.

If **warning** is @TRUE, WIL will pop up a dialog box warning you if you are about to overwrite an existing file, and giving you an opportunity to change your mind. If **warning** is @FALSE, it will overwrite existing files with no warning.

Example:

```
FileCopy("win.ini", "*.sav", @TRUE)
Run("notepad.exe", "win.ini")
```

The asterisk (*****) is a **wildcard** character, which matches any letter or group of letters in a file name. In this case, it will cause WIN.INI to be copied as WIN.SAV.

FileDelete **(**file-list**)**

Deletes files.

Example:

```
If FileExist("win.bak") == @TRUE Then FileDelete("win.bak")
```

FileRename **(**from-list**,** to-file**)**

Renames files to another set of names.

We can illustrate the use of these WIL program functions with a typical WIL application. Let's suppose that our word processor saves a backup copy of each document, with a BAK extension, but we want a larger safety net when editing important files. We want to keep the five most recent versions of the wonderful software manual we're writing. Here's a WIL program to accomplish this:

```
If FileExist("wil.bak") == @TRUE Then Goto backup
:edit
Run("winword.exe", "wil.doc")
Exit
:backup
FileDelete("wil.bk5")
FileRename("wil.bk4", "wil.bk5)
FileRename("wil.bk3", "wil.bk4)
FileRename("wil.bk2", "wil.bk3)
FileRename("wil.bk1", "wil.bk2)
FileRename("wil.bak", "wil.bk1)
Goto edit
```

If the file WIL.BAK exists, it means that we have made a change to WIL.DOC. So, before we start editing, we delete the oldest backup copy, and perform several **FileRename** functions, until eventually WIL.BAK becomes WIL.BK1. Notice how the flow of control moves to the line labeled **:backup**, and then back to the line labeled **:edit**, and how we terminate processing with the **Exit** command.

If we did not include the **Exit** command, the WIL program would continue in an endless loop.

However, this still isn't quite right. What would happen if the file WIL.BK5 didn't exist? In the DOS batch language, we would get an error message, and processing would continue. But in WIL, the error would cause the WIL program to terminate:

There are two ways that we can handle this. We could use an **If FileExist** test before every file operation, and test the returned value for a @TRUE before proceeding. But this is clumsy, even with such a small WIL program, and would become unwieldy with a larger one.

Handling Errors

Luckily, there is a WIL system function to help us here: **ErrorMode**. The **ErrorMode** function lets you decide what will happen if an error occurs during WIL processing. Here's the syntax:

ErrorMode **(**mode**)**

Specifies how to handle errors.

Parameters:

(i) mode = **@CANCEL**, **@NOTIFY**, or **@OFF**.

Returns:

(i) previous error setting.

Use this command to control the effects of runtime errors. The default is **@CANCEL**, meaning the execution of the WIL program will be canceled for any error.

@CANCEL: All runtime errors will cause execution to be canceled. The user will be notified which error occurred.

@NOTIFY: All runtime errors will be reported to the user, and they can choose to continue if it isn't fatal.

@OFF: Minor runtime errors will be suppressed. Moderate and fatal errors will be reported to the user. User has the option of continuing if the error is not fatal.

As you can see, the default mode is @CANCEL, and it's a good idea to leave it like this. However, it is quite reasonable to change the mode for sections of your WIL program where you anticipate errors occurring. This is just what we've done in our modified WIL program:

```
@@If FileExist("wil.bak") == @TRUE Then Goto backup
:edit
Run("winword.exe", "wil.doc")
Exit
:backup
ErrorMode(@OFF)
FileDelete("wil.bk5")
FileRename("wil.bk4", "wil.bk5)
FileRename("wil.bk3", "wil.bk4)
FileRename("wil.bk2", "wil.bk3)
FileRename("wil.bk1", "wil.bk2)
FileRename("wil.bak", "wil.bk1)
ErrorMode(@CANCEL)
Goto edit
```

Notice how we've used **ErrorMode(@OFF)** to prevent errors in the section labeled **backup:** from aborting the WIL program, and then used **ErrorMode(@CANCEL)** at the end of the that section to change back to the default error mode. This is a good practice to follow.

Selecting from Lists

So far, whenever we have needed to use a file name, we've hard-coded it into our WIL programs. For example:

```
Run("notepad.exe", "agenda.txt")
```

Naturally, there should be a way to get this information from the user "on the fly", so that we wouldn't have to write hundreds of different WIL programs. And there is a way. Three or four ways, actually. Consider, first, a function that we have already seen, **AskLine**:

```
file = AskLine("", "Enter Filename to edit?", "")
Run("notepad.exe", file)
```

This will prompt for a filename, and run Notepad on that file:

There are only three problems with this approach. First, the user might not remember the name of the file. Second, the user might enter the name incorrectly. And finally, modern software is supposed to be sophisticated and user-friendly enough to handle these things the *right* way. And WIL certainly can.

There are two new functions we need to use for an improved file selection routine: **FileItemize** and **ItemSelect**.

FileItemize (file-list**)**

Returns a space-delimited list of files.

This function compiles a **list** of filenames and separates the names with spaces. There are several variations we can use:

```
FileItemize("*.doc")
```

would give us a list of all files in the current directory with a DOC extension,

```
FileItemize("*.com *.exe")
```

would give us a list of all files in the current directory with a COM or EXE extension, and

```
FileItemize("*.*")
```

would give us a list of *all* files in the current directory.

Of course, we need to be able to use this list, and for that we have:

ItemSelect **(**title**,** list**,** delimiter**)**

Displays a listbox filled with items from a list you specify in a string. The items are separated in your string by a delimiter character.

This is the function which actually displays the list box. Remember that **FileItemize** returns a file list delimited by spaces, which would look something like this:

FILE1.DOC FILE2.DOC FILE3.DOC

When we use **ItemSelect**, we need to tell it that the delimiter is a space. We do this as follows:

```
textfiles = FileItemize("*.doc *.txt")
yourfile = ItemSelect("Select file to edit", textfiles, " ")
Run("notepad.exe", yourfile)
```

First, we use **FileItemize** to build a list of filenames with DOC and TXT extensions. We assign this list to the variable **textfiles**. Then, we use the **ItemSelect** function to build a list box, passing it the variable **textfiles** as its second parameter. The third parameter we use for **ItemSelect** is simply a space with quote marks around it; this tells **ItemSelect** that the variable **textfiles** is delimited by spaces. (Note that this is different from the null string that we've seen earlier – here, you must include a space between the quote marks.) Finally, we assign the value returned by **ItemSelect** to the variable **yourfile**, and run Notepad using that file.

In the list box, if the user presses **Enter** or clicks on the **OK** button without a file being highlighted, ItemSelect returns a null string. If you want, you can test for this condition:

```
textfiles = FileItemize("*.doc *.txt")
:retry
yourfile = ItemSelect("Select file to edit", textfiles, " ")
If yourfile == "" Then Goto retry
Run("notepad.exe", yourfile)
```

DirItemize **(**dir-list**)**

Returns a space-delimited list of directories.

This function is similar to **FileItemize**, but instead of returning a list of files, it returns a list of directories. Remember we said that **FileItemize** only lists files in the current directory. Often, we want to be able to use files in other directories as well. We can do this by first letting the user select the appropriate directory, using the **DirItemize** and **ItemSelect** combination:

```
DirChange("\")
subdirs = DirItemize("*")
targdir = ItemSelect("Select dir", subdirs, " ")        DirChange(targdir)
files = FileItemize("*.*")
file = ItemSelect("Select file", files, " ")
Run("notepad.exe", file)
```

First we change to the root directory. Then we use **DirItemize** to get a list of all the subdirectories off of the root directory. Next, we use **ItemSelect** to give us a list box of directories from which to select. Finally, we change to the selected directory, and use **FileItemize** and **ItemSelect** to pick a file.

Although this WIL program works, it needs to be polished up a bit. What happens if the file we want is in the \WINDOWS\BATCH directory? Our WIL program doesn't go more than one level deep from the root directory. We want to continue down the directory tree, but we also need a way of telling when we're at the end of a branch. As it happens, there is such a way: **DirItemize** will return a null string if there are no directories to process. Given this knowledge, we can set up a loop to test when we are at the lowest level:

```
DirChange("\")
:getdir
subdirs = DirItemize("*")
If subdirs == "" Then Goto getfile
targdir = ItemSelect("Select dir (OK = curr)", subdirs, " ")
If targdir == "" Then Goto getfile
DirChange(targdir)
Goto getdir
:getfile
files = FileItemize("*.*")
file = ItemSelect("Select file", files, " ")
If file == "" Then Goto getfile
Run("notepad.exe", file)
```

After we use the **DirItemize** function, we test the returned value for a null string. If we have a null string, then we know that the current directory has no subdirectories, and so we proceed to select the filename from the current directory (**Goto getfile**). If, however, **DirItemize** returns a non-blank list, then we know that there is, in fact, at least one subdirectory. In that case, we use **ItemSelect** to present the user with a list box of directories. Then, we test the value returned by **ItemSelect**. If the returned value is a null string, it means that the user did not select a directory from the list, and presumably wants a file in the current directory. We happily oblige (**Goto getfile**). On the other hand, a non-blank value returned by **ItemSelect** indicates that the user has selected a subdirectory from the list box. In that case, we change to the selected directory, and loop back to the beginning of the directory selection routine (**Goto getdir**). We continue this process until either (a) the user selects a directory, or (b) there are no directories left to select. Eventually, we get to the section labeled **:getfile**.

Nicer File Selection

An even more elegant way of selecting a file name is provided by the **Dialog** function, which also allows the user to select various options via check boxes and radio buttons.

Nicer Messages

Have you tried displaying long messages, and found that WIL didn't wrap the lines quite the way you wanted? Here are a couple of tricks.

Num2Char **(**integer**)**

Converts a number to its character equivalent.

We want to be able to insert a carriage return/line feed combination at the end of each line in our output, and the **Num2Char** function will let us do that. A carriage return has an ASCII value of 13, and a line feed has an ASCII value of 10 (don't worry if you don't understand what this sentence means). To be

able to use these values, we must convert them to characters, as follows:

```
cr = Num2Char(13)
lf = Num2Char(10)
```

Now, we need to be able to place the variables **cr** and **lf** in our message. For example, let's say we want to do this:

```
Message("", "This is line one This is line two")
```

If we just inserted the variables into the string, as in:

```
cr = Num2Char(13)
lf = Num2Char(10)
Message("", "This is line one cr lf This is line two")
```

we would not get the desired effect. WIL would simply treat them as ordinary text:

However, WIL does provide us with a method of performing variable substitution such as this, and that is by delimiting the variables with percentage signs (**%**). If we do this:

```
cr = Num2Char(13)
lf = Num2Char(10)
Message("", "This is line one %cr% %lf%This is line two")
```

we will get what we want:

Note that there is no space after **%lf%**; this is so that the second line will be aligned with the first line (every space within the delimiting quote marks of a string variable is significant).

Now, wouldn't it be convenient if we could combine **cr** and **lf** into a single variable? We can.

StrCat (string**[,** string**]...)**

Concatenates strings together.

The **StrCat** function lets us combine any number of string constants and/or string variables. Here's how

we combine the variables **cr** and **lf** into the single variable **crlf**:

```
crlf = StrCat(cr, lf)
```

Note that the strings to be concatenated are separated by commas, within the parentheses. Now, we can rewrite our example, as follows:

```
cr = Num2Char(13)
lf = Num2Char(10)
crlf = StrCat(cr, lf)
Message("", "This is line one %crlf%This is line two")
```

If we wanted to re-use this message a number of times, it would be quite convenient to use the **StrCat** function to make a single variable out of it:

```
cr = Num2Char(13)
lf = Num2Char(10)
crlf = StrCat(cr, lf)
line1 = "This is line one"
line2 = "This is line two"
mytext = StrCat(line1, crlf, line2)
Message("", mytext)
```

Running DOS Programs

WIL can run DOS programs, just like it runs Windows programs:

```
DirChange("c:\game")
Run("scramble.exe", "")
```

If you want to use an internal DOS command, such as **DIR** or **TYPE**, you can do so by running the DOS command interpreter, COMMAND.COM, with the **/c** program parameter, as follows:

```
Run("command.com", "/c type readme.txt")
```

Everything that you would normally type on the DOS command line goes after the **/c** in the second parameter. Here's another example:

```
Run("command.com", "/c type readme.txt | more")
```

These examples assume that COMMAND.COM is in a directory on your DOS path. If it isn't, you could specify a full path name for it:

```
Run("c:\command.com", "/c type readme.txt | more")
```

Or, better still, you could use the WIL **Environment** function.

Environment (env-variable**)**

Gets a DOS environment variable.

Since DOS always stores the full path and filename of the command processor in the DOS environment variable **COMSPEC**, it is an easy matter to retrieve this information:

```
coms = Environment("COMSPEC")
```

and use it in our WIL program:

```
coms = Environment("COMSPEC")
Run(coms, "/c type readme.txt")
```

To get a DOS window, just run COMMAND.COM with no parameters:

```
coms = Environment("COMSPEC")
Run(coms, "")
```

Sending Keystrokes to Programs

Here we come to one of the most useful and powerful features of WIL: the ability to send keystrokes to Windows programs, just as if you were typing them directly from the keyboard.

SendKey**(character-codes)**

Sends keystrokes to the active application.

This is an ideal way to make the computer automatically type the keystrokes that you enter every time you start a certain program. For example, to start up Notepad and have it prompt you for a file to open, you would use:

```
Run("notepad.exe", "")
SendKey("!fo")
```

The parameter you specify for **SendKey** is the string that you want sent to the program. This string consists of standard characters, as well as some special characters which you will find listed under the entry for **SendKey** in the **WIL Function Reference**. In the example above, the exclamation mark (**!**) stands for the **Alt** key, so **!f** is the equivalent of pressing and holding down the **Alt** key while simultaneously pressing the **F** key. The **o** in the example above is simply the letter **O**, and is the same as pressing the **O** key by itself:

Here's another example:

```
RunZoom("sol.exe", "")
SendKey("!gc(RIGHT)(SP)~")
```

This starts up Solitaire, brings up the **Game** menu (**!g**), and selects **Deck** (**c**) from that menu:

Then it moves the cursor to the next card back style on the right (**(RIGHT)**), selects that card back (**(SP)**), and then selects **OK** (**~**).

And *voilà*! A different card design every time you play!

Our Completed WIL File

Here is the final working version of the WIL program that we've slowly been building throughout this tutorial:

```
; solitare.wbt
mins = AskLine("Solitaire", "How many mins do you want to play?", "")
If WinExist("Solitaire") == @TRUE Then Goto activate
RunZoom("sol.exe", "")
Goto loaded
:activate
WinActivate("Solitaire")
WinZoom("Solitaire")
:loaded
SendKey("!gc(RIGHT)(SP)~")
goal = mins * 60
timer = 0
:moretime
remain = goal - timer
WinTitle("Solitaire", "Solitaire (%remain% seconds left)")
Delay(10)
timer = timer + 10
If WinExist("Solitaire") == @FALSE Then Exit
If timer < goal Then Goto moretime
Beep
WinClose("Solitaire")
Message("Time's up", "Get back to work!")
```

It incorporates many of the concepts that we've discussed so far, as well as using some arithmetic (*****, **-**, **+**) and relational (**<**) operators that are covered in the section on the **WIL Language** (pg. **Error! Bookmark not defined.**).

It can also be improved and customized in a number of ways, but we'll leave that up to you.

If you can understand and follow the structures and processes illustrated in this sample file, and can begin to incorporate them into your own WIL programs, you are well on your way to becoming a true WIL guru!

Advanced Techniques

This section covers some miscellaneous items, of a more advanced nature.

Recovering from Cancel

If the user presses the **Cancel** button (in any dialog which has one), the label **:CANCEL** will be searched for in the WIL program, and, if found, control will be transferred there. If no label **:CANCEL** is found, processing simply stops.

This allows the program developer to perform various bits of cleanup processing after a user presses **Cancel**.

Aborting WIL Processing

A currently-executing WIL program can be terminated immediately by pressing the **<CtrlShiftBreak>** key combination.

Default Program for Unknown Extension

The **Run** function (and related members of the **Run...** family of functions) allow you to run a data file if it is associated with a program via the [Extensions] section of the WIN.INI file. You can also (optionally) create a special default program entry in that section, as follows:

```
*=program.exe
```

where an asterisk is used instead of a file extension. Then, if you try to run a data file whose extension is not specified in [Extensions], WIL will run "program.exe." Even though the customary **^.ext** is not included in the example line above, WIL will pass the name of the data file as a command-line parameter to "program.exe."

Note: WIL does **not** use the Windows registration database to match data files with their associated programs.

Partial window names

Those WIL functions which take a partial windowname as a parameter can be directed to accept only an exact match, by ending the window name with a tilde (**~**). For example, **WinShow("Note~")** would only match a window whose title was "Note"; it would **not** match "Notepad".

Sounds

If you have Windows Multimedia extensions, and hardware capable of playing WAV waveform files, there will be sounds audible at various points in the execution of WIL programs. By default, these sounds are enabled. If you want sounds to be off by default, enter the line:

```
Sounds=Off
```

in the [Main] section of the WWWBATCH.INI file.

You can also use the **Sounds** function to turn sounds on and off from within a WIL program.

If you add to the [Sounds] section of your WIN.INI file a line such as:

```
StartProgram=CHIMES.WAV,Program Launch
```

then the WIL Interpreter will make sounds whenever a new program is launched.

Programming Reference

The WIL language consists of commands for controlling the Windows interface and the programs that run within it.

These commands are written as scripts into a plain text file. When this file is run by a user, each line is then executed.

Automating Windows tasks with WIL gives you a script that is easily edited when you need changes. It is also a common denominator language that works the same for all Windows applications.

Programming Style in WIL

WIL supports, but does not enforce, structured programming. Small scripts of 20 lines or less probably do not benefit from a structured approach. Longer scripts can be written in structured style with an opening section for declaring variable and constant values. Liberal use of comments (anything beginning with a semicolon is a comment in WIL) can make your scripts clear.

WIL includes the capability of using sequential **IFTHEN** Else conditional instructions. Nesting of conditional statements is done with **Goto** statements and labels. While the use of Goto statements goes against structured programming dogma, the scripts produced in WIL are generally short and easily documented.

Scope of variables in WIL is handled by the **Drop** function. If you drop it, it's gone. If you don't, it sticks around as long as the WIL application runs.

In the case of batch language applications, the variables will vanish when the batch file finishes running. In the case of menuing applications that use WIL, the variables keep their values until they are defined again or they are dropped.

Language Components

WIL statements are constructed from **constants**, **variables**, **operators**, **functions**, **commands**, and **comments**.

Each line in a WIL program can be up to 255 characters long.

Constants

The programming language supports both integer and string constants.

Integer Constants

Integer constants are built from the digits **0** through **9**. They can range in magnitude from negative to positive $2^{31} - 1$ (approximately two billion). Constants larger than these permissible magnitudes will produce unpredictable results.

Examples of integer constants:

1
-45
377849
-1999999999

String Constants

String constants are comprised of displayable characters bounded by quote marks. You can use double quotes (**"**), single quotes (**'**), or back quotes (`) to enclose a string constant, as long as the same type of quote is used to both start and end it. If you need to embed the delimiting quote mark inside the string constant, use the delimiting quote mark twice.

Examples of string constants:

```
"a"
`Betty Boop`
"This constant has an embedded "" mark"
'This constant also has an embedded " mark'
```

Predefined Constants

The programming language has a number of built-in integer constants that can be used for various purposes. These start with the **@**-sign, and are **case-insensitive** (although we prefer to use ALL CAPS).

The most used predefined constants:

@FALSE

@NO

@STACK

@TILE

@TRUE

@YES

More predefined constants can be found in the **Predefined Constants** List.

Identifiers

Identifiers are the names supplied for variables, functions, and commands in your program.

An identifier is a sequence of one or more letters or digits that begins with a letter. Identifiers may have up to 30 characters.

All identifiers are *case insensitive*. Upper-case and lower-case characters may be mixed at will inside variable names, commands or functions.

For example, these statements all mean the same thing:

```
AskLine(MyTitle, Prompt, Default)
ASKLINE(MYTITLE, PROMPT, DEFAULT)
aSkLiNe(MyTiTlE, pRoMpT, dEfAuLt)
```

Variables

A variable may contain an integer, a string, a list, or a string representing an integer. Automatic

conversions between integers and strings are performed as a matter of course during execution.

If a function requires a string parameter and an integer parameter is supplied, the variable will be automatically modified to include the representative string.

If a function requires an integer parameter and a string parameter is supplied, an attempt will be made to convert the string to an integer. If it does not convert successfully, an error will result.

Lists

A **list** is a string variable which itself contains one or more strings, each of which is **delimited** (separated) by a common character. For example, the **FileItemize** function returns a list of file names, delimited by spaces, and the **WinItemize** function returns a list of window names, delimited by tabs. In order to use functions which accept a list as a parameter, such as **ItemSelect**, you will need to know what character is being used to delimit your list.

Keywords

Keywords are the predefined identifiers that have special meaning to the programming language. These cannot be used as variable names.

WIL keywords consist of the **functions**, **commands**, and **predefined constants**.

Some examples of reserved keywords:

Beep

DirChange

@Yes

FileCopy

Operators

The programming language operators take one operand ("unary operators") or two operands ("binary operators").

Unary operators (integers only):

- Arithmetic Negation (Two's complement)

+ Identity (Unary plus)

~ Bitwise Not. Changes each **0** bit to **1**, and vice-versa.

! Logical Not. Produces **0** (**@FALSE**) if the operand is nonzero, else **1** (**@TRUE**) if the operand is zero.

Binary arithmetic operators (integers only):

***** Multiplication

/ Division

mod	Modulo
+	Addition
-	Subtraction
<<	Left Shift
>>	Right Shift
&	Bitwise And
|	Bitwise Or
^	Bitwise Exclusive Or (XOR)
&&	Logical And
| |	Logical Or

Binary relational operators (integers and strings):

>	Greater-than
>=	Greater-than or equal
<	Less-than
<=	Less-than or equal
==	Equality
!= or **<>**	Inequality

Assignment operator (integers and strings):

=	Assigns evaluated result of an expression to a variable

Precedence and evaluation order

The precedence of the operators affect the evaluation of operands in expressions. Operands associated with higher-precedence operators are evaluated before the lower-precedence operators.

The table below shows the precedence of the operators. Where operators have the same precedence, they are evaluated from left to right.

Operator	**Description**
( )	Parenthetical grouping
~ ! - +	Unary operators
*** / mod**	Multiplication & Division
+ -	Addition & Subtraction

<< >>	Shift operators
< <= == >= > != <>	Relational operators
& ^ |	Bit manipulation operators
&& ||	Logical operators

Comments

A comment is a sequence of characters that are ignored when processing a command. A semicolon (not otherwise part of a string constant) indicates the beginning of a comment.

All characters to the right of the semicolon are considered comments, and are ignored.

Blank lines are also ignored.

Examples of comments:

```
; This is a comment
abc = 5       ; This is also a comment
```

Statements

Assignment Statements

Assignment statements are used to set variables to specific or computed values. Variables may be set to integers or strings.

Examples:

```
a = 5
value = Average(a, 10, 15)
location = "Northern Hemisphere"
world = StrCat(location, " ", "Southern Hemisphere")
```

Control Statements

Control statements are generally used to execute system management functions and consist of a call to a command without assigning any return values.

Examples:

```
Exit
Yield
```

Substitution

The WIL language has a powerful substitution feature which inserts the contents of a string variable into a statement before the line is parsed.

To substitute the contents of a variable in the statement, simply put a percent-sign (**%**) on both sides of the variable name.

Examples:

```
mycmd = "DirChange('c:\')"          ;set mycmd to a command
%mycmd%                             ;execute the command
```

Or consider this one:

```
IniWrite("PC", "User", "Richard")
...
name = IniRead("PC", "User", "somebody")
message("", "Thank you, %name%")
```

will produce this message box:

The variable substitution feature can be used to simulate an "array" of strings. For example, if you wanted to read the lines contained in a file into an array of variables named **line1** through **line#** (where **#** is the line number of the last line in the file), and then write them to a new file in reverse order, you could do so as follows:

```
handle = FileOpen("c:\config.sys", "READ")
num = 0
:readnext
num = num + 1
line%num% = FileRead(handle)
If line%num% != "*EOF*" Then Goto readnext
FileClose(handle)
handle = FileOpen("c:\config.rev", "WRITE")
:writenext
num = num - 1
FileWrite(handle, line%num%)
If num > 1 Then Goto writenext
FileClose(handle)
Message("Processing complete", "CONFIG.REV created")
```

To put a single percent-sign (**%**) on a source line, specify a double percent sign(**%%**). This is required even inside quoted strings.

Note: The length of a line, after any substitution occurs, may not exceed 255 characters.

Function Parameters

Most of the functions and commands in the language require parameters. These come in several types:

Integer

String

List

Variable name

The interpreter performs automatic conversions between strings and integers, so in general you can use them.

Integer parameters may be any of the following:

An integer (i.e. 23)

A string representing an integer (i.e. "23")

A variable containing an integer

A variable containing a string representing an integer

String parameters may be any of the following:

A string

An integer

A variable containing a string

A variable containing a list

A variable containing an integer

Predefined Constants

WIL provides you with a number of predefined integer constants to help make your WIL programs more mnemonic:

Logical Conditions

@FALSE

@NO

@OFF

@TRUE

@YES

@ON

Window Arranging

@NORESIZE

@ABOVEICONS

@STACK

@ARRANGE

@TITLE

@ROWS

@COLUMNS

Window Status

@NORMAL

@ZOOMED

@ICON

@HIDDEN

Menu Handling

@CHECK

@UNCHECK

@DISABLE

@ENABLE

String Handling

@FWDSCAN

@BACKSCAN

Menu Handling

@ENABLE

@DISABLE

@UNCHECK

@CHECK

System Control

@MAJOR

@MINOR

Error Handling

@CANCEL

@NOTIFY

@OFF

Keyboard Status

@SHIFT

@CTRL

Debug Control

@PARSEONLY

INI File Management

@WHOLESECTION

Error Handling

There are three types of errors that can occur while processing a WIL program: **Minor**, **Moderate**, and **Fatal**. What happens when an error occurs depends on the current error mode, which is set with the **ErrorMode** function.

There are three possible modes you can specify:

@CANCEL

User is notified when any error occurs, and then the WIL program is canceled. This is the default.

@NOTIFY

User is notified when any error occurs, and has option to continue unless the error is fatal.

@OFF

User is only notified if the error is moderate or fatal. User has option to continue unless the error is fatal.

The function **LastError** returns the code of the most-recent error encountered during the currently-executing WIL program.

Minor errors are numbered from **1000** to **1999**.
Moderate errors are numbered from **2000** to **2999**.
Fatal errors are numbered from **3000** to **3999**.

Error handling is reset to **@CANCEL** at the start of each WIL program.

The Complete Functions & Statements

Inputting Information

AskLine (title, prompt, default)
Lets user enter a line of information.
AskPassword (title, prompt)
Prompts the user for a password.
AskYesNo (title, question)
Lets user choose from Yes, No, or Cancel.
ItemSelect (title, list, delimiter)
Chooses an item from a listbox.
TextBox (title, filename)
Fills a listbox with text strings from a file.
TextBoxSort (title, filename)
Fills a sorted listbox with text strings from a file.
TextSelect (title, list, delimiter)
Allows the user to choose an item from an unsorted listbox.

Displaying Information

Beep
Beeps at the user.
Dialog (dialog-name)
Displays a user-defined dialog box.
DialogBox (title, WDG file)
Pops up a Windows dialog box defined by the WDG template file.
Display (seconds, title, text)
Momentarily displays a string.
Message (title, text)
Displays text in a message box.
Pause (title, text)
Displays text in a message box.
TextBox (title, filename)
Fills a listbox with text strings from a file.
TextBoxSort (title, filename)
Fills a sorted listbox with text strings from a file.
TextSelect (title, list, delimiter)
Allows the user to choose an item from an unsorted listbox.

File Management

FileAppend (from-list, to-file)
Appends one or more files to another file.
FileAttrGet (filename)
Returns file attributes.
FileAttrSet (file-list, settings)
Sets file attributes.
FileClose (filehandle)
Closes a file.
FileCopy (from-list, to-file, warning)
Copies files.
FileDelete (file-list)
Deletes files.
FileExist (filename)
Determines if a file exists.
FileExtension (filename)
Returns extension of file.
FileItemize (file-list)

Builds a list of files.

FileLocate (filename)
Finds a file within the current DOS path.

FileMove (from-list, to-file, warning)
Moves files to another set of pathnames.

FileOpen (filename, open-type)
Opens a STANDARD ASCII (only) file for reading or writing.

FilePath (filename)
Returns path of file.

FileRead (filehandle)
Reads data from a file.

FileRename (from-list, to-file)
Renames files to another set of names.

FileRoot (filename)
Returns root of file.

FileSize (file-list)
Adds up the total size of a set of files.

FileTimeGet (filename)
Returns file date and time.

FileTimeTouch (file-list)
Sets file (s) to current time.

FileWrite (filehandle,output-data)
Writes data to a file.

IniDelete (section, keyname)
Removes a line or section from WIN.INI.

IniDeletePvt (section, keyname, filename)
Removes a line or section from a private INI file.

IniItemize (section)
Lists keywords or sections in WIN.INI.

IniItemizePvt (section, filename)
Lists keywords or sections in a private INI file.

IniRead (section, keyname, default)
Reads a string from the WIN.INI file.

IniReadPvt (section, keyname, default, filename)
Reads a string from a private INI file.

IniWrite (section, keyname, string)
Writes a string to the WIN.INI file.

IniWritePvt (section, keyname, data, filename)
Writes a string to a private INI file.

Directory Management

DirChange ([d:]path)
Changes the current directory.

DirGet ( )
Returns the current directory path.

DirHome ( )
Returns the initial directory path.

DirItemize (dir-list)
Builds a list of directories.

DirMake ([d:]path)
Creates a new directory.

DirRemove ([d:]path)
Removes an existing directory.

DirRename ([d:]oldpath, [d:]newpath)
Renames a directory.

DirWindows **(**request#**)**
Returns the name of the Windows or Windows System directory.

Disk Drive Management

DiskFree **(**drive-list**)**
Returns the amount of free space on a set of drives.
DiskScan **(**request#**)**
Returns list of drives.
LogDisk **(**drive**)**
Changes the logged disk drive.

Window Management

AppExist **(**program-name**)**
Tells if an application is running.
AppWaitClose **(**program-name**)**
Suspends WIL program execution until a specified application has been closed.
IconArrange **( )**
Rearranges icons.
WinActivate **(**partial-winname**)**
Makes an application window the active window.
WinArrange **(**style**)**
Arranges all running application windows on the screen.
WinClose **(**partial-winname**)**
Closes an application window.
WinCloseNot **(**partial-winname [**,** partial-winname...]**)**
Closes all application windows except those specified.
WinExeName **(**partial-winname**)**
Returns the name of the executable file which created a specified window.
WinExist **(**partial-winname**)**
Tells if window exists.
WinGetActive **( )**
Gets the title of the active window.
WinHide **(**partial-winname**)**
Hides an application window.
WinIconize **(**partial-winname**)**
Turns an application window into an icon.
WinItemize **( )**
Lists all the main windows currently running.
WinName **( )**
Returns the name of the window calling the WIL Interpreter.
WinPlace **(**x-ul**,** y-ul**,** x-br**,** y-br**,** partial-winname**)**
Changes the size and position of an application window on the screen.
WinPlaceGet **(**win-type**,** partial-winname**)**
Returns window coordinates.
WinPlaceSet **(**win-type**,** partial-winname**,** position-string**)**
Sets window coordinates.
WinPosition **(**partial-winname**)**
Returns window position.
WinShow **(**partial-winname**)**
Shows a currently-hidden application window.
WinState **(**partial-winname**)**
Returns the current state of a window.
WinTitle **(**partial-winname**,** new-winname**)**
Changes the title of an application window.
WinWaitClose **(**partial-winname**)**
Waits until an application window is closed.

WinZoom (partial-winname)
Maximizes an application window to full-screen.

Program Management

Run (program-name, parameters)
Runs a program as a normal window.
RunHide (program-name, parameters)
Runs a program in a hidden window.
RunHideWait (program-name, parameters)
Runs a program in a hidden window, and waits for it to close.
RunIcon (program-name, parameters)
Runs a program as an icon.
RunIconWait (program-name, parameters)
Runs a program as an icon, and waits for it to close.
RunWait (program-name, parameters)
Runs a program as a normal window, and waits for it to close.
RunZoom (program-name, parameters)
Runs a program in a maximized window.
RunZoomWait (program-name, parameters)
Runs a program in a maximized window, and waits for it to close.

String Handling

Char2Num (string)
Returns the ANSI code of a string's first character.
IsNumber (string)
Determines if a string represents a valid number.
ItemCount (list, delimiter)
Returns the number of items in a list.
ItemExtract (select, list, delimiter)
Returns the selected item from a list.
ItemInsert (item, index, list, delimiter)
Adds an item to a list.
ItemLocate (item, list, delimiter)
Returns the position of an item in a list.
ItemRemove (index, list, delimiter)
Removes an item from a list.
ItemSort (list, delimiter)
Sorts a list.
Num2Char (number)
Converts a number to the ANSI character it represents.
ParseData (string)
Parses the passed string, just like passed parameters are parsed.
StrCat (string[, string...])
Concatenates strings together.
StrCmp (string1, string2)
Compares two strings.
StrFill (string, string-length)
Builds a string from a repeated smaller string.
StrFix (base-string, padding-string, length)
Pads or truncates a string to a fixed length.
StriCmp (string1, string2)
Compares two strings, ignoring their case.
StrIndex (main-str, sub-str, start, direction)
Locates a string within a larger string.
StrLen (string)
Returns the length of a string

StrLower (string)
Converts a string to all lower-case characters.
StrReplace (string, old, new)
Replaces all occurrences of a substring with another.
StrScan (main-str, delims, start, direction)
Finds an occurrence of one or more delimiter characters in a string.
StrSub (string, start, length)
Returns a substring from within a string.
StrTrim (string)
Trims leading and trailing blanks from a string.
StrUpper (string)
Converts a string to all upper-case characters.

Arithmetic Functions

Abs (number)
Returns the absolute value of a number.
Average (num [, num...])
Returns the average of a list of numbers.
Max (num [, num...])
Determines the highest number in a list.
Min (num [, num...])
Determines the lowest number in a list.
Random (max)
Generates a positive random number.

Clipboard Handling

ClipAppend (string)
Appends a string to the end of the Clipboard.
ClipGet ( )
Returns the Clipboard contents into a string.
ClipPut (string)
Replaces the Clipboard contents with a string.

Process Control

Call (filename, parameters)
Calls a WIL batch file as a subroutine.
Debug (mode)
Turns Debug mode on or off.
Delay (seconds)
Pauses WIL program execution.
Drop (var [, var...])
Deletes variables to recover their memory.
Else statement
Continues a previous **If** statement.
EndSession ( )
Ends the current Windows session.
ErrorMode (mode)
Sets what happens in the event of an error.
Exclusive (mode)
Controls whether or not other Windows program will get any time to execute.
Execute statement
Directly executes a WIL statement.
Exit
Unconditionally ends a WIL program.
Goto label
Changes the flow of control in a WIL program.
Ifcondition **Then** statement

Conditionally performs a function.

IgnoreInput **(**mode**)**
Turns off hardware input to windows.

IsDefined **(**variable**)**
Determines if a variable is currently defined.

IsKeyDown **(**key-codes**)**
Tells about keys/mouse.

LastError **( )**
Returns the last error encountered.

Return
Returns from a Call to the calling program.

SKDebug **(**mode**)**
Controls how **SendKey** works

Terminate
Conditionally ends a WIL program.

Then statement
Continues a previous **If** statement.

WaitForKey
Waits for a specific key to be pressed.

Yield
Pauses WIL processing so other applications can process some messages.

Miscellaneous Functions

ButtonNames **(**OK-name**,** Cancel-name**)**
Changes the names of the buttons which appear in WIL dialogs.

IntControl **(**request#**,** p1**,** p2**,** p3**,** p4**)**
Internal control functions.

SendKey **(**character-codes**)**
Sends keystrokes to the active application.

SnapShot **(**request#**)**
Takes a snapshot of the screen and pastes it to the clipboard.

WallPaper **(**bmp-name**,** tile**)**
Changes the Windows wallpaper.

WinParmSet **(**request#**,** new-value**,** ini-control**)**
Sets system information.

WinHelp **(**help-file**,** function**,** keyword**)**
Calls a Windows help file.

System Information

DateTime **( )**
Returns the current date and time.

DOSVersion **(**level**)**
Returns the version numbers of the current version of DOS.

Environment **(**env-variable**)**
Returns the value of a DOS environment variable.

IsLicensed **( )**
Tells if the calling application is licensed.

MouseInfo **(**request#**)**
Returns assorted mouse information.

Version **( )**
Returns the version of the parent program currently running.

VersionDLL **( )**
Returns the version of the WIL Interpreter currently running.

WinConfig **( )**
Returns WIN3 mode flags.

WinMetrics **(**request#**)**

Returns Windows system information.

WinParmGet (request#)
Returns system information.

WinParmSet (request#, new-value, ini-control)
Sets system information.

WinResources (request#)
Returns information on available memory and resources.

WinVersion (level)
Returns the version of Windows that is currently running.

DDE Functions

DDEExecute (channel, command string)
Sends commands to a DDE server application.

DDEInitiate (app name, topic name)
Opens a DDE channel.

DDEPoke (channel, item name, item value)
Sends data to a DDE server application.

DDERequest (channel, item name)
Gets data from a DDE server application.

DDETerminate (channel)
Closes a DDE channel.

DDETimeout (value)
Sets the DDE timeout value.

Network Functions

NetAddCon (net-path, password, local-name)
Connects network resources to imaginary local disk drives or printer ports.

NetAttach (server-name)
Attaches to a network file server.

NetBrowse (request#)
Displays a network dialog box allowing the user to select a network resource.

NetCancelCon (name, force)
Breaks a network connection.

NetDetach (server-name)
Detaches from a network file server.

NetDialog ( )
Brings up the network driver's dialog box.

NetGetCaps (request#)
Returns information on network capabilities.

NetGetCon (local-name)
Returns the name of a connected network resource.

NetGetUser ( )
Returns the name of the user currently logged into the network.

NetLogin (server-name, user-name, password)
Performs a network login.

NetLogout (server-name)
Performs a network logout.

NetMapRoot (local-name, net-path)
Maps a local drive to a network resource.

NetMemberGet (server-name, group-name)
Determines whether the current user is a member of a specific group.

NetMemberSet (server-name, group-name)
Sets the current user as a member of a group.

NetMsgAll (server-name, message)
Broadcasts a message to all users on the network.

NetMsgSend (server-name, user-name, message)

Sends a message to a specific user on the network.

Multimedia Functions

PlayMedia (command-string**)**
Controls multimedia devices.

PlayMidi (filename**,** mode**)**
Plays a MID or RMI sound file.

PlayWaveForm (filename**,** mode**)**
Plays a WAV sound file.

Sounds (request#**)**
Turns sounds on or off.

Menu Management

CurrentFile ( )
Returns the selected filename.

IsMenuChecked (menuname**)**
Determines if a menu item has a checkmark next to it.

IsMenuEnabled (menuname**)**
Determines if a menu item has been enabled.

MenuChange (menuname**,** flags**)**
Checks, unchecks, enables, or disables a menu item.

Inputting Information

AskLine (title, prompt, default)
Lets user enter a line of information.

AskPassword (title, prompt)
Prompts the user for a password.

AskYesNo (title, question)
Lets user choose from Yes, No, or Cancel.

ItemSelect (title, list, delimiter)
Chooses an item from a listbox.

TextBox (title, filename)
Fills a listbox with text strings from a file.

TextBoxSort (title, filename)
Fills a sorted listbox with text strings from a file.

TextSelect (title, list, delimiter)
Allows the user to choose an item from an unsorted listbox.

Displaying Information

Beep
Beeps at the user.

Dialog (dialog-name)
Displays a user-defined dialog box.

DialogBox (title, WDG file)
Pops up a Windows dialog box defined by the WDG template file.

Display (seconds, title, text)
Momentarily displays a string.

Message (title, text)
Displays text in a message box.

Pause (title, text)
Displays text in a message box.

TextBox (title, filename)
Fills a listbox with text strings from a file.

TextBoxSort (title, filename)
Fills a sorted listbox with text strings from a file.

TextSelect (title, list, delimiter)
Allows the user to choose an item from an unsorted listbox.

File Management

FileAppend (from-list, to-file)
Appends one or more files to another file.
FileAttrGet (filename)
Returns file attributes.
FileAttrSet (file-list, settings)
Sets file attributes.
FileClose (filehandle)
Closes a file.
FileCopy (from-list, to-file, warning)
Copies files.
FileDelete (file-list)
Deletes files.
FileExist (filename)
Determines if a file exists.
FileExtension (filename)
Returns extension of file.
FileItemize (file-list)
Builds a list of files.
FileLocate (filename)
Finds a file within the current DOS path.
FileMove (from-list, to-file, warning)
Moves files to another set of pathnames.
FileOpen (filename, open-type)
Opens a STANDARD ASCII (only) file for reading or writing.
FilePath (filename)
Returns path of file.
FileRead (filehandle)
Reads data from a file.
FileRename (from-list, to-file)
Renames files to another set of names.
FileRoot (filename)
Returns root of file.
FileSize (file-list)
Adds up the total size of a set of files.
FileTimeGet (filename)
Returns file date and time.
FileTimeTouch (file-list)
Sets file (s) to current time.
FileWrite (filehandle,output-data)
Writes data to a file.
IniDelete (section, keyname)
Removes a line or section from WIN.INI.
IniDeletePvt (section, keyname, filename)
Removes a line or section from a private INI file.
IniItemize (section)
Lists keywords or sections in WIN.INI.
IniItemizePvt (section, filename)
Lists keywords or sections in a private INI file.
IniRead (section, keyname, default)
Reads a string from the WIN.INI file.
IniReadPvt (section, keyname, default, filename)
Reads a string from a private INI file.
IniWrite (section, keyname, string)

Writes a string to the WIN.INI file.

IniWritePvt (section, keyname, data, filename**)**

Writes a string to a private INI file.

Directory Management

DirChange ([d**:**]path**)**
Changes the current directory.

DirGet ( )
Returns the current directory path.

DirHome ( )
Returns the initial directory path.

DirItemize (dir-list**)**
Builds a list of directories.

DirMake ([d**:**]path**)**
Creates a new directory.

DirRemove ([d**:**]path**)**
Removes an existing directory.

DirRename ([d**:**]oldpath**,** [d:]newpath**)**
Renames a directory.

DirWindows (request**#)**
Returns the name of the Windows or Windows System directory.

Disk Drive Management

DiskFree (drive-list**)**
Returns the amount of free space on a set of drives.

DiskScan (request#**)**
Returns list of drives.

LogDisk (drive**)**
Changes the logged disk drive.

Window Management

AppExist **(**program-name**)**
Tells if an application is running.
AppWaitClose **(**program-name**)**
Suspends WIL program execution until a specified application has been closed.
IconArrange **( )**
Rearranges icons.
WinActivate **(**partial-winname**)**
Makes an application window the active window.
WinArrange **(**style**)**
Arranges all running application windows on the screen.
WinClose **(**partial-winname**)**
Closes an application window.
WinCloseNot **(**partial-winname [**,** partial-winname...]**)**
Closes all application windows except those specified.
WinExeName **(**partial-winname**)**
Returns the name of the executable file which created a specified window.
WinExist **(**partial-winname**)**
Tells if window exists.
WinGetActive **( )**
Gets the title of the active window.
WinHide **(**partial-winname**)**
Hides an application window.
WinIconize **(**partial-winname**)**
Turns an application window into an icon.
WinItemize **( )**
Lists all the main windows currently running.
WinName **( )**
Returns the name of the window calling the WIL Interpreter.
WinPlace **(**x-ul**,** y-ul**,** x-br**,** y-br**,** partial-winname**)**
Changes the size and position of an application window on the screen.
WinPlaceGet **(**win-type**,** partial-winname**)**
Returns window coordinates.
WinPlaceSet **(**win-type**,** partial-winname**,** position-string**)**
Sets window coordinates.
WinPosition **(**partial-winname**)**
Returns window position.
WinShow **(**partial-winname**)**
Shows a currently-hidden application window.
WinState **(**partial-winname**)**
Returns the current state of a window.
WinTitle **(**partial-winname**,** new-winname**)**
Changes the title of an application window.
WinWaitClose **(**partial-winname**)**
Waits until an application window is closed.
WinZoom **(**partial-winname**)**
Maximizes an application window to full-screen.

Program Management

Run **(**program-name**,** parameters**)**
Runs a program as a normal window.
RunHide **(**program-name**,** parameters**)**
Runs a program in a hidden window.
RunHideWait **(**program-name**,** parameters**)**
Runs a program in a hidden window, and waits for it to close.
RunIcon **(**program-name**,** parameters**)**
Runs a program as an icon.
RunIconWait **(**program-name**,** parameters**)**
Runs a program as an icon, and waits for it to close.
RunWait **(**program-name**,** parameters**)**
Runs a program as a normal window, and waits for it to close.
RunZoom **(**program-name**,** parameters**)**
Runs a program in a maximized window.
RunZoomWait **(**program-name**,** parameters**)**
Runs a program in a maximized window, and waits for it to close.

String Handling

Char2Num (string)
Returns the ANSI code of a string's first character.

IsNumber (string)
Determines if a string represents a valid number.

ItemCount (list, delimiter)
Returns the number of items in a list.

ItemExtract (select, list, delimiter)
Returns the selected item from a list.

ItemInsert (item, index, list, delimiter)
Adds an item to a list.

ItemLocate (item, list, delimiter)
Returns the position of an item in a list.

ItemRemove (index, list, delimiter)
Removes an item from a list.

ItemSort (list, delimiter)
Sorts a list.

Num2Char (number)
Converts a number to the ANSI character it represents.

ParseData (string)
Parses the passed string, just like passed parameters are parsed.

StrCat (string[, string...])
Concatenates strings together.

StrCmp (string1, string2)
Compares two strings.

StrFill (string, string-length)
Builds a string from a repeated smaller string.

StrFix (base-string, padding-string, length)
Pads or truncates a string to a fixed length.

StriCmp (string1, string2)
Compares two strings, ignoring their case.

StrIndex (main-str, sub-str, start, direction)
Locates a string within a larger string.

StrLen (string)
Returns the length of a string

StrLower (string)
Converts a string to all lower-case characters.

StrReplace (string, old, new)
Replaces all occurances of a substring with another.

StrScan (main-str, delims, start, direction)
Finds an occurrence of one or more delimiter characters in a string.

StrSub (string, start, length)
Returns a substring from within a string.

StrTrim (string)
Trims leading and trailing blanks from a string.

StrUpper (string)
Converts a string to all upper-case characters.

Arithmetic Functions

Abs (number**)**
Returns the absolute value of a number.

Average (num [**,** num...]**)**
Returns the average of a list of numbers.

Max (num [**,** num...]**)**
Determines the highest number in a list.

Min (num [**,** num...]**)**
Determines the lowest number in a list.

Random (max**)**
Generates a positive random number.

Clipboard Handling

ClipAppend (string**)**
Appends a string to the end of the Clipboard.

ClipGet ( )
Returns the Clipboard contents into a string.

ClipPut (string**)**
Replaces the Clipboard contents with a string.

Process Control

Call **(**filename, parameters**)**
Calls a WIL batch file as a subroutine.

Debug **(**mode**)**
Turns Debug mode on or off.

Delay **(**seconds**)**
Pauses WIL program execution.

Drop **(**var [**,** var...]**)**
Deletes variables to recover their memory.

Else statement
Continues a previous **If** statement.

EndSession **( )**
Ends the current Windows session.

ErrorMode **(**mode**)**
Sets what happens in the event of an error.

Exclusive **(**mode**)**
Controls whether or not other Windows program will get any time to execute.

Execute statement
Directly executes a WIL statement.

Exit
Unconditionally ends a WIL program.

Goto **label**
Changes the flow of control in a WIL program.

Ifcondition **Then** statement
Conditionally performs a function.

IgnoreInput **(**mode**)**
Turns off hardware input to windows.

IsDefined **(**variable**)**
Determines if a variable is currently defined.

IsKeyDown **(**key-codes**)**
Tells about keys/mouse.

LastError **( )**
Returns the last error encountered.

Return
Returns from a Call to the calling program.

SKDebug **(**mode**)**
Controls how **SendKey** works

Terminate
Conditionally ends a WIL program.

Then statement
Continues a previous **If** statement.

WaitForKey
Waits for a specific key to be pressed.

Yield
Pauses WIL processing so other applications can process some messages.

Miscellaneous Functions

ButtonNames (OK-name, Cancel-name)
Changes the names of the buttons which appear in WIL dialogs.

IntControl (request#, p1, p2, p3, p4)
Internal control functions.

SendKey (character-codes)
Sends keystrokes to the active application.

SnapShot (request#)
Takes a snapshot of the screen and pastes it to the clipboard.

WallPaper (bmp-name, tile)
Changes the Windows wallpaper.

WinParmSet (request#, new-value, ini-control)
Sets system information.

WinHelp (help-file, function, keyword)
Calls a Windows help file.

System Information

DateTime ( )
Returns the current date and time.

DOSVersion (level**)**
Returns the version numbers of the current version of DOS.

Environment (env-variable**)**
Returns the value of a DOS environment variable.

IsLicensed ( )
Tells if the calling application is licensed.

MouseInfo (request#**)**
Returns assorted mouse information.

Version ( )
Returns the version of the parent program currently running.

VersionDLL ( )
Returns the version of the WIL Interpreter currently running.

WinConfig ( )
Returns WIN3 mode flags.

WinMetrics (request#**)**
Returns Windows system information.

WinParmGet (request#**)**
Returns system information.

WinParmSet (request#**,** new-value**,** ini-control**)**
Sets system information.

WinResources (request#**)**
Returns information on available memory and resources.

WinVersion (level**)**
Returns the version of Windows that is currently running.

DDE Functions

DDEExecute **(**channel**,** command string**)**
Sends commands to a DDE server application.

DDEInitiate **(**app name**,** topic name**)**
Opens a DDE channel.

DDEPoke **(**channel**,** item name**,** item value**)**
Sends data to a DDE server application.

DDERequest **(**channel**,** item name**)**
Gets data from a DDE server application.

DDETerminate **(**channel**)**
Closes a DDE channel.

DDETimeout **(**value**)**
Sets the DDE timeout value.

Network Functions

NetAddCon **(**net-path**,** password**,** local-name**)**
Connects network resources to imaginary local disk drives or printer ports.

NetAttach **(**server-name**)**
Attaches to a network file server.

NetBrowse **(**request#**)**
Displays a network dialog box allowing the user to select a network resource.

NetCancelCon **(**name**,** force**)**
Breaks a network connection.

NetDetach **(**server-name**)**
Detaches from a network file server.

NetDialog **( )**
Brings up the network driver's dialog box.

NetGetCaps **(**request#**)**
Returns information on network capabilities.

NetGetCon **(**local-name**)**
Returns the name of a connected network resource.

NetGetUser **( )**
Returns the name of the user currently logged into the network.

NetLogin **(**server-name**,** user-name**,** password**)**
Performs a network login.

NetLogout **(**server-name**)**
Performs a network logout.

NetMapRoot **(**local-name**,** net-path**)**
Maps a local drive to a network resource.

NetMemberGet **(**server-name**,** group-name**)**
Determines whether the current user is a member of a specific group.

NetMemberSet **(**server-name**,** group-name**)**
Sets the current user as a member of a group.

NetMsgAll **(**server-name**,** message**)**
Broadcasts a message to all users on the network.

NetMsgSend **(**server-name**,** user-name**,** message**)**
Sends a message to a specific user on the network.

Multimedia Functions

PlayMedia (command-string**)**
Controls multimedia devices.

PlayMidi (filename**,** mode**)**
Plays a MID or RMI sound file.

PlayWaveForm (filename**,** mode**)**
Plays a WAV sound file.

Sounds (request**#)**
Turns sounds on or off.

WIL FUNCTION REFERENCE

Introduction

The WIL programming language consists of a large number of functions and commands, which we describe in detail in this section.

We use a shorthand notation to indicate the syntax of the functions.

Function names and other actual characters you type are in **boldface**. Optional parameters are enclosed in square brackets "**[ ]**". When a function takes a variable number of parameters, the variable parts will be followed by ellipses ("**...**").

Take, for example, string concatenation:

StrCat (string[**,** string...]**)**

This says that the **StrCat** function takes at least one string parameter. Optionally, you can specify more strings to concatenate. If you do, you must separate the strings with commas.

For each function and command, we show you the **Syntax**, describe the **Parameters** (if any), the value it **Returns** (if any), a description of the function, **Example** code, and related functions you may want to **See Also**.

Please note:

(i) indicates an integer parameter or return value.
(s) indicates a string parameter or return value.

Errors

If the current error mode is **@CANCEL** (the default), any WIL errors encountered while processing a WIL program cause the item to be canceled with an error message.

Minor Errors

Minor errors are ignored if the current error mode has been set to **@OFF**. If the error mode is **@NOTIFY** the user has the option of continuing with the WIL program or canceling it.

1006 File Copy/Move: No matching files found

1017 File Delete: No matching files found

1018 File Delete: Delete Failed

1024 File Rename: No matching files found

1025 File Rename: Rename failed

1028 LogDisk: Requested drive not online

1029 DirMake: Dir not created

1030 DirRemove: Dir not removed

1031 DirChange: Dir not found/changed

1039 WinClose: Window not found

1040 WinHide: Window not found

1041 WinIconize: Window not found

1042 WinZoom: Window not found

1043 WinShow: Window not found

1044 WinPlace: Window not found

1045 WinActivate: Window not found

1077 FileOpen: Open failed

1083 FileAttrGet: File not found

1086 FileAttrSet: File not found or access denied

1100 StrIndex/StrScan 3rd parameter out of bounds

1119 WinPosition: Window not found

1121 WinTitle: Window not found

1125 FileTimeGet: File not found

1128 FileTimeTouch: File not found

1150 DDEExec: DDE Post failed

1155 DDEReq: DDE Post failed

1163 DDEPoke: DDE Post failed

1164 DDEPoke: DDE Timeout

1165 DDEReq: DDE Timeout

1166 DDEExec: DDE Timeout

1172 WinExeName: Window not found

1173 Net: No network found

1174 Net: Security Violation

1175 Net: Function not supported

1176 Net: Out of Memory

1177 Net: Network Error

1178 Net: Windows function failed

1179 Net: Invalid type of request

1180 Net: Invalid Pointer

1181 Net: Cancelled at users request

1182 Net: Bad user / Not logged in

1183 Net: Buffer too small - Internal Error

1184 Net: Invalid Network name

1185 Net: Invalid Local Name

1186 Net: Invalid Password

1187 Net: Local Device already connected

1188 Net: Not a valid local device or network name

1189 Net: Not a redirected local device or current net name

1190 Net: Files were open with FORCE=FALSE

1191 Net: Function busy

1192 Net: Unknown network error

1193 Function not supported in this version of Windows

1194 PlaySounds: File not found

1195 PlayMedia: Unrecognized Error

1200 WinPlaceGet/Set: Window not found

1201 WinPlaceGet/Set: Function failed

1207 SnapShot: Out of memory

1208 SnapShot: Palette Creation Error

1209 SnapShot: Cannot open clipboard

1213 Cmd Extender: Minor error occurred

1216 RunWait Commands not supported in 3.0 Debug Windows

1217 WinHelp: Invalid SubCommand Requested

Moderate Errors

If the error mode is **@NOTIFY** or **@OFF**, the user has the option of continuing with the WIL program or canceling it.

2001 SendKey: Illegal Parameters

2002 File Copy/Move: 'From' file illegal

2003 File Copy/Move: 'To' file illegal

2004 File Copy/Move: Cannot put wildcards into fixed root

2005 File Copy/Move: Cannot put wildcards into fixed extension

2007 File Move: Unable to rename source file

2016 File Delete: File name illegal

2019 File Rename: 'From' file illegal

2020 File Rename: 'To' file illegal

2021 File Rename: Can't change disk drives. Use MOVE instead.

2022 File Rename: Cannot put wildcards into a fixed root

2023 File Rename: Cannot put wildcards into a fixed extension

2038 WinCloseNot Function Syntax error

2058 StrCat: Function syntax error

2060 AVERAGE function syntax error

2093 Dialog Box: Bad Filespec, using *.*

2106 SetDisplay: Type not NAME, DATE, SIZE, KIND or UNSORTED

2112 FileSize: File Not Found

2118 FileCopy/Move: Destination file same as source

2120 SetDisplay: Display type not SHORT or LONG

2122 FileAppend: Target cannot be wildcarded

2203 Dir Rename: 'From' file illegal

2204 Dir Rename: 'To' file illegal

2214 Cmd Extender: Moderate Error Occurred

Fatal Errors

Fatal errors cause the current WIL program to be canceled with an error message, regardless of the error mode in effect. (We show the error codes here for consistency, but in practice you will never be able to call **LastError** after a fatal error.)

3008 File Copy/Move: 'From' file open error

3009 SendKey: Could not open DEBUG TEXT file

3010 SendKey: Could not install hook - Already Active??

3011 File Copy/Move: 'From' file length error

3012 File Copy/Move: No room left on disk. Out of space??

3013 File Copy/Move: 'To' file open error

3014 File Copy/Move: I/O Error

3015 File Move: Unable to remove source file

3026 LogDisk: Illegal disk drive

3027 LogDisk: DOS reports no disks!! ???

3032 GoTo unable to lock memory for batch file

3033 GoTo label not found

3034 Clipboard owned by another app. Cannot open.

3035 Clipboard does not contain text for CLIPAPPEND.

3036 Clipboard cannot hold that much text (>64000 bytes)

3037 Unable to get memory for clipboard. Close some apps

3046 Internal Error 3046. Function not defined

3047 Variable name over 30 chars. Too Long

3048 Substitution %Variable% not followed by a % (Use %% for %)

3049 No variables exist??!!

3050 Undefined variable

3051 Undefined variable or function

3052 Uninitialized variable or undefined function

3053 Character string too long (>256 chars??)

3054 Unrecognizable item found on line

3055 Variable name over 30 chars. Too Long

3056 Variable could not be converted to string

3057 Variable could not be converted to integer

3059 Illegal Bounds for STRSUB function

3061 Illegal Syntax

3062 Attempt to divide by zero

3063 Internal Error 3063. Binary op not found

3064 Internal Error 3064. Unary op not found

3065 Unbalanced Parenthesis

3066 Wrong Number of Arguments in Function

3067 Function Syntax. Opening parenthesis missing.

3068 Function Syntax. Illegal delimiter found.

3069 Bad assignment statement. (Use == for equality testing)

3070 Internal error 3070. Too many arguments defined.

3071 Missing or incomplete statement

3072 THEN not found in IF statement

3073 Goto Label not specified

3074 Expression continues past expected end.

3075 Call: Parse of file/parameter line failed

3076 FileOpen: READ or WRITE not properly specified

3078 FileOpen: Too many (>5) files open

3079 FileClose: Invalid file handle

3080 FileClose: File not currently open

3081 FileRead: Invalid file handle

3082 FileRead: File not currently open

3084 FileWrite: Invalid file handle

3085 FileWrite: File not currently open

3087 FileRead: File not open for reading

3088 FileRead: Attempt to read past end of file

3089 FileWrite: File not open for writing

3090 Dialog Box: File open error

3091 Dialog Box: Box too large (20x60 max)

3092 Dialog Box: Non-text control used w/filebox.

3094 Dialog Box: Window Registration Failed

3095 Compare: Not an integer or string compare

3096 Memory allocation failure. Out of memory for strings

3097 Memory allocation failure. Out of memory for variables

3098 IntErr: NULL pointer passed to xstrxxx subroutines

3099 CallExt function disabled. Not currently available.

3101 Substituted line too long. (> 256 characters)

3102 Drop: Can only drop variables

3103 IsDefined: Attempting to test non-variable item

3104 Dialog Box: Window Creation Failed

3105 CALL and CALLEXT not supported in compiled versions

3107 Run: Filetype is not COM, EXE, PIF or BAT

3108 FileItemize: Unable to lock file info segment

3109 FileItemize: Unable to unlock file info segment

3110 FileItemize: Unable to lock file index segment

3111 FileItemize: Unable to unlock file index segment

3113 FileSize: Filelength I/O Error

3114 FileSize: Buffer Overrun Error

3115 FileDelete: Buffer Overrun Error

3116 FileRename: Buffer Overrun Error

3117 FileCopy/Move: Buffer Overrun Error

3123 WaitForKey: Invalid key codes specified

3124 WinMetrics: Invalid code

3126 FileAttrSet: Buffer Overrun Error

3127 FileTimeTouch: Buffer Overrun Error

3138 DDE: Too many DDE conversations

3139 DDEInitiate: Client window create failed

3140 DDEInitiate: GlobalAddAtom failure

3142 DDETerminate: Channel does not exist

3144 DDETerminate: Internal Error 3144

3145 DDEExec: GlobalAlloc failed

3146 DDEExec: Global Lock failed

3147 DDEExec: Bad channel number

3149 DDEExec: Internal Error 3149

3154 DDEReq: GlobalAddAtom failed

3156 DDEReq: GlobalLock failed

3160 DDEPoke: GlobalAlloc failed

3161 DDEPoke: GlobalAddAtom failed

3162 DDEPoke: GlobalLock failed

3167 DDE Recv Data: GlobalLock 1 failed

3168 DDE Recv Data: GlobalAlloc 2 failed

3169 DDE Recv Data: GlobalLock 2 failed

3170 DDEInitiate: Internal Error 3170

3171 IntControl: Invalid IntControl opcode

3196 PlayMedia: Do not use WAIT or NOTIFY in MCI string

3197 WinResources: Invalid request number

3198 WinParmGet/Set: Invalid request number

3199 WinPlaceGet/Set: Invalid window-size number

3202 WinPlaceSet: Wrong number of window co-ordinates

3205 MouseInfo: Invalid request number

3206 SnapShot: Invalid request number

3210 Cmd Extender: Out of memory to save result

3211 Call: More than 6 levels of Calls

3212 PlayMedia: Notify Window creation failed

3215 Cmd Extender: Severe error occurred

3218 Dialog: Dialog name too long (>16 chars)

3219 Dialog: Format variable missing

3220 Dialog: Format version not supported

3221 Dialog: x, y, width or height variables bad

3222 Dialog: Control definition variable missing

3223 Dialog: Bad Control type in definition variable

3224 Dialog: Bad or missing Value for Radio/Checkbox button

3225 Dialog: Too many items in definition variable

AppExist

Tells if an application is running.

Syntax:

AppExist (program-name)

Parameters:

(s) program-name name of a Windows EXE or DLL file.

Returns:

(i) **@TRUE** if the specified application is running;
@FALSE if the specified application is not running.

Use this function to determine whether a specific Windows application is currently running. Unlike **WinExist**, you can use **AppExist** without knowing the title of the application's window.

"Program-name" is the name of a Windows EXE or DLL file, including the file extension (and, optionally, a full path to the file).

Example:

```
If AppExist("clock.exe") == @FALSE Then Run("clock.exe", "")
```

See Also:

AppWaitClose, **Run**, **WinExeName**, **WinExist**

AppWaitClose

Suspends WIL program execution until a specified application has been closed.

Syntax:

AppWaitClose (program-name)

Parameters:

(s) program-name name of a Windows EXE or DLL file.

Returns:

(i) **@TRUE** if the specified application is running;
@FALSE if the specified application is not running.

Use this function to suspend the WIL program's execution until the user has finished using a given application and has manually closed it. Unlike **WinWaitClose**, you can use **AppWaitClose** without knowing the title of the application's window.

"Program-name" is the name of a Windows EXE or DLL file, including the file extension (and, optionally, a full path to the file).

Example:

```
Run("clock.exe", "")
Display(4, "Note", "Close Clock to continue")
AppWaitClose("clock.exe")
Message("Continuing...", "Clock closed")
```

See Also:

AppExist, **Delay**, **RunWait**, **WinExeName**, **Yield**

Abs

Returns the absolute value of a number.

Syntax:

Abs (integer**)**

Parameters:

integer = integer whose absolute value is desired.

Returns:

(integer) absolute value of integer.

This function returns the absolute (positive) value of the integer which is passed to it, regardless of whether that integer is positive or negative.

Example:

```
dy = Abs(y1 - y2)

Message("Years", "There are %dy% years 'twixt %y1% and %y2%")
```

See Also:

Average, **Max**, **Min**, **IsNumber**

AskLine

Prompts the user for one line of input.

Syntax:

AskLine (title**,** prompt**,** default**)**

Parameters:

"title" = title of the dialog box.

"prompt" = question to be put to the user.

"default" = default answer.

Returns:

(string) user response.

Use this function to query the user for a line of data. The entire user response will be returned if the user presses the OK button or the Enter key. If the user presses Cancel, the batch file processing is canceled.

Example:

```
name = AskLine("Game", "Please enter your name", "")

game = AskLine("Game", "Favorite game?", "Solitaire")

 message(StrCat(name,"'s favorite game is "), game)
```

The first line displays a dialog that lets the user enter a name.

The second line displays a dialog that lets the user enter a game.

The third line combines the results from the first two and displays a dialog with the information entered in the previous dialogs.

See Also:

AskPassword,**AskYesNo**, **Dialog**, **Display**, **ItemSelect**, **Message**, **Pause**, **TextBox**, **TextSelect**

AskPassword

Prompts the user for a password.

Syntax:

AskPassword (title, prompt)

Parameters:

(s) title title of the dialog box.

(s) prompt question to be put to the user.

Returns:

(s)user response.

Pops up a special dialog box to ask for passwords. An asterisk (*) is echoed for each character that the user types; the actual characters entered are not displayed.

Example:

```
pw = AskPassword("Security check", "Please enter your password")

 If StriCmp(pw, "winguy") != 0 Then Goto nogo

 Run(Environment("COMSPEC"), "")

 Exit

:nogo

 Pause("Security breach", "Invalid password entered")
```

See Also:

AskLine, **AskYesNo**, **DialogBox**

AskYesNo

Prompts the user for a YES or NO answer.

Syntax:

AskYesNo (title**,** question**)**

Parameters

"title" = title of the question box.

"question" = question to be put to the user.

Returns:

(integer) **@YES** or **@NO**, depending on the button pressed.

This function displays a message box with three pushbuttons - Yes, No, and Cancel. If the user presses Cancel**,** the current batch file is ended, so there is no return value.

Example:

```
q = AskYesNo('Testing', 'Please press "YES"')

If q == @YES Then Exit

Display(3, 'ERROR', 'I said press "YES"')
```

See Also:

AskLine, **Display**, **ItemSelect**, **Message**, **Pause**, **TextBox**

Average

Returns the average of a list of numbers.

Syntax:

Average (integer [**,** integer]...**)**

Parameters:

integer = integers to get the average of.

Returns:

(integer) average of the integers.

Use this function to compute the mean average of a series of numbers, delimited by commas. This function returns an integer value, so there can be some rounding error involved.

Example:

```
avg = Average(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12)

Message("The average is", avg)
```

See Also:
Abs, **Max**, **Min**, Random

Beep

Beeps once.

Syntax:

Beep

Use this command to produce a short beep, generally to alert the user to an error situation.

Example:

```
Beep

Pause("WARNING!!!", "You are about to destroy data!")
```

See Also:

PlayMedia, **PlayMidi**, **PlayWaveForm**, **Sounds**

ButtonNames

Changes the names of the buttons which appear in WIL dialogs.

Syntax:

ButtonNames (OK-name, Cancel-name)

Parameters:

(s) OK-name new name for the OK button.

(s) Cancel-name new name for the Cancel button.

Returns:

(i) always 1.

This function allows you to specify alternate names for the OK and/or Cancel buttons which appear in any dialogs displayed by the WIL Interpreter. Each use of the **ButtonNames** statement only affects the next WIL dialog which is displayed.

You can specify a null string (**""**) for either the OK-name or Cancel-Name parameter, to use the default name for that button (i.e., "OK" or "Cancel").

You can place an ampersand before the character which you want to be the underlined character in the dialog.

Example:

```
ButtonNames("", "&Abort")
user = AskLine("Hello", "What is your name", "")
Message("Hello", user)
```

Call

Calls a WIL batch file as a subroutine in applications that have that capability. (This function also works from batch files compiled with WIL Language compilers. The file that will be called must be compiled separately with the compiler. It can be given any MSDOS file name, not just the *.WBT names illustrated in the examples here.)

Syntax:

Call (filename.wbt, parameters**)**

Parameters:

"filename.wbt" = the WBT file you are calling. The WBT extension is required. *

"parameters" = the parameters to pass to the file, if any, in the form "p1 p2 p3... pn".

Returns:

(integer) always @FALSE.

This function is used to pass control temporarily to a secondary WBT file. The main WBT file can optionally pass parameters to the secondary WBT file. All variables are common (**global**) between the calling and the called WBT files, so that the secondary WBT file may modify or create variables. The secondary WBT file should end with a **Return** statement, to pass control back to the main WBT file.

If a string of parameters is passed to the secondary WBT file, it will automatically be parsed into individual variables with the names **param1**, **param2**, etc., (maximum of nine parameters). The variable **param0** will be a count of the total number of parameters in the string.

Example:

```
; MAIN.WBT

  name = AskLine("", "What is your name?", "")

age = AskLine("", "How old are you?", "")

valid = @NO

Call("chek-age.wbt", age)

  If valid == @NO Then Message("", "Invalid age")

; CHEK-AGE.WBT

userage = param1

  really = AskYesNo("", "%name%, are you really %userage%?")

  If really == @NO Then Return

If (userage > 0) && (userage < 150) Then valid = @YES
```

Return

See Also:

ParseData, **Return**

* Compiled files can have any extension, not just WBT.

Char2Num

Converts the first character of a string to its numeric equivalent.

Syntax:

Char2Num (string**)**

Parameters:

"string" = any text string. Only the first character will be converted.

Returns:

(integer) ANSI character code.

This function returns the 8-bit ANSI code corresponding to the first character of the string parameter.

Note: For the commonly-used characters (with codes below 128), ANSI and ASCII characters are identical.

Example:

```
; Show the hex equivalent of entered character

inpchar = AskLine("ANSI Equivalents", "Char:", "")

ansi = StrSub(inpchar, 1, 1)

ansiequiv = Char2Num(InpChar)

Message("ANSI Codes", "%ansi% => %ansiequiv%")
```

See Also:

Num2Char

ClipAppend

Appends a string to the Clipboard.

Syntax:

ClipAppend (string**)**

Parameters:

"string" = text string to add to Clipboard.

Returns:

(integer) **@TRUE** if string was appended;

@FALSE if Clipboard ran out of memory.

Use this function to append a string to the Windows Clipboard. The Clipboard must either contain text data or be empty for this function to succeed.

Example:

```
; The code below will append 2 copies of the

; Clipboard contents back to the Clipboard, resulting

; in 3 copies of the original contents with a CR/LF

; between each copy.

  a = ClipGet()

  crlf = StrCat(Num2Char(13), Num2Char(10))

ClipAppend(crlf)

ClipAppend(a)

ClipAppend(crlf)

ClipAppend(a)
```

See Also:

ClipGet, **ClipPut**

ClipGet

Returns the contents of the Clipboard.

Syntax:

ClipGet ( )

Parameters:

(none)

Returns:

(string) clipboard contents.

Use this function to copy text from the Windows Clipboard into a string variable.

Note: If the Clipboard contains an excessively large string a (fatal) out of memory error may occur.

Example:

```
; The code below will convert Clipboard contents to
; uppercase
ClipPut(StrUpper(ClipGet()))
a = ClipGet()
Message("UPPERCASE Clipboard Contents", a)
```

See Also:

ClipAppend, **ClipPut**

ClipPut

Copies a string to the clipboard.

Syntax:

ClipPut (string**)**

Parameters:

"string" = any text string.

Returns:

(integer) **@TRUE** if string was copied;

@FALSE if clipboard ran out of memory.

Use this function to copy a string to the Windows Clipboard. The previous Clipboard contents will be lost.

Example:

```
; The code below will convert Clipboard contents to

; lowercase

ClipPut(StrLower(ClipGet()))

a = ClipGet()

Message("lowercase Clipboard Contents", a)
```

See Also:

ClipAppend, **ClipGet**

CurrentFile

Returns the selected file name.

Syntax:

CurrentFile()

Returns:

(string) currently-selected file name.

When a WIL menuing program (a Windows program that makes a unique use of the WIL Language) displays the files in the current directory, one of them may be "selected." This function returns the name of that file, if any. This capability is not present in most programs that call WIL.

For these menu-based programs, a current file is different from a "highlighted" file. When a file is highlighted, it shows up in inverse video (usually white-on-black). To find the filenames that are highlighted, see **FileItemize**.

Note: This command is not part of the WIL Interpreter package, but is documented here because it has been implemented in some applications which call WIL.

Example:

```
;Ask which program to run (default = current file)

TheFile = AskLine("Run It","Program:", CurrentFile())

Run(TheFile,"")
```

See Also:

FileItemize, DirGet, DirItemize

DateTime

Provides the current Date and time.

Syntax:

DateTime ( )

Parameters:

(none)

Returns:

(string) today's date and time

This function will return the current date and time in a pre-formatted string. The format it is returned in depends on how it is set up in the international section of the WIN.INI file:

ddd mm:dd:yy hh:mm:ss XX

ddd dd:mm:yy hh:mm:ss XX

ddd yy:mm:dd hh:mm:ss XX

Where:

ddd is day of the week (e.g. Mon)

mm is the month (e.g. 10)

dd is the day of the month (e.g. 23)

yy is the year (e.g. 90)

hh is the hours

mm is the minutes

ss is the seconds

XX is the Day/Night code (e.g. AM or PM)

Note: Windows provides even more formatting options than this.

The WIN.INI file will be examined to determine which format to use. You can adjust the WIN.INI file via the **International** section of **Control Panel** if the format isn't what you prefer.

Example:

```
; assuming the current standard is U.S.

; (i.e. day   dd/mm/yy hh:mm:ss AM)
```

Message("Current Date & Time", DateTime())

would produce a dialog box with a title of "Current Date & Time" and a message of "Sat 2/29/92 2:53:18 PM". The dialog includes an OK button the user can use to cancel the dialog.

See Also:

FileTimeGet

DDEExecute

Sends commands to a DDE server application.

Syntax:

DDEExecute (channel, command string)

Parameters:

(i) channel	same integer that was returned by **DDEInitiate**.
(s) command string	one or more commands to be executed by the server application.

Returns:

(i)	@TRUE if successful; @FALSE if unsuccessful.

Use the **DDEInitiate** function to obtain a channel number.

In order to use this function successfully, you will need appropriate documentation for the server application you wish to access, which must provide information on the DDE functions that it supports and the correct syntax to use.

Example:

```
Run("wincheck.exe", "TUT")

channel = DDEInitiate("wincheck", "TUT")

If channel == 0 Then Goto failed

  result = DDEExecute(channel, '[WriteCheck:p="Shorewood Apartments",t=580.00,l="Rent"]')

  DDETerminate(channel)

  WinClose("WinCheck")

  If result == @FALSE Then Goto Failed

  Message("DDE Execute", "Operation complete")

  Exit

:failed

  Message("DDE operation unsuccessful", "Check your syntax")
```

See Also:

DDEInitiate, **DDEPoke**, **DDERequest**, **DDETerminate**, **DDETimeout**

DDEInitiate

Opens a DDE channel.

Syntax:

DDEInitiate (server name, topic name)

Parameters:

(s) server name name of the application (without the **E** extension).

(s) topic name name of the topic you wish to access.

Returns:

(i) communications channel.

This function opens a DDE communications channel with a server application. The communications channel can be subsequently used by the **DDEExecute**, **DDEPoke**, and **DDERequest** functions. You should close this channel with **DDETerminate** when you are finished using it. If the communications channel cannot be opened as requested, **DDEInitiate** returns a channel number of 0.

You can call **DDEInitiate** more than once, in order to carry on multiple DDE conversations (with multiple applications, or with multiple data sources within one application) simultaneously.

In order to use this function successfully, you will need appropriate documentation for the server application you wish to access, which must provide information on the DDE functions that it supports and the correct syntax to use. Most often, the server name is identical to the program file name, without the ".exe". Some server names are listed here:

Ami Pro	amipro
Crosstalk for Windows	xtalk
DynaComm	dynacomm
Excel	excel
Microphone II	microphone
Q + E	qe
PackRat	packrat
Superbase 4	SB4W
Toolbook	toolbook
Word for Windows	winword

Example:

```
Run("wincheck.exe", "TUT")

channel = DDEInitiate("WinCheck", "TUT")

If channel == 0 Then Goto failed

output = DDERequest(channel, "GetChecking")

DDETerminate(channel)

WinClose("WinCheck")
```

```
If output == "" Then Goto_Failed
  Message("Account balance", output)
  Exit
:failed
Message("DDE operation unsuccessful", "Check your syntax")
```

See Also:

DDEExecute, **DDEPoke**, **DDERequest**, **DDETerminate**, **DDETimeout**

DDEPoke

Sends data to a DDE server application.

Syntax:

DDEPoke (channel, item name, item value)

Parameters:

(i) channel same integer that was returned by **DDEInitiate**.

(s) item name identifies the type of data being sent.

(s) item value actual data to be sent to the server.

Returns:

(i) @TRUE if successful; @FALSE if unsuccessful.

Use the **DDEInitiate** function to obtain a channel number.

In order to use this function successfully, you will need appropriate documentation for the server application you wish to access, which must provide information on the DDE functions that it supports and the correct syntax to use.

Example:

```
Run("reminder.exe", "")
channel = DDEInitiate("Reminder", "items")
If channel == 0 Then Goto failed
result = DDEPoke(channel, "all", "11/3/92 Misc Remember to vote")
DDETerminate(channel)
WinClose("Reminder")
If result == @FALSE Then Goto Failed
Message("DDE Poke", "Operation complete")
Exit
:failed
Message("DDE operation unsuccessful", "Check your syntax")
```

See Also:

DDEExecute, **DDEInitiate**, **DDERequest**, **DDETerminate**, **DDETimeout**

DDERequest

Gets data from a DDE server application.

Syntax:

DDERequest (channel, item name)

Parameters:

(i) channel same integer that was returned by **DDEInitiate**.

(s) item name identifies the data to be returned by the server.

Returns:

(s)information returned from the server.

Use the **DDEInitiate** function to obtain a channel number.

In order to use this function successfully, you will need appropriate documentation for the server application you wish to access, which must provide information on the DDE functions that it supports and the correct syntax to use.

Example:

```
Run("wincheck.exe", "TUT")
channel = DDEInitiate("WinCheck", "TUT")
If channel == 0 Then Goto failed
output = DDERequest(channel, "GetChecking")
DDETerminate(channel)
WinClose("WinCheck")
If output == "" Then Goto Failed
Message("Account balance", output)
Exit
:failed
Message("DDE operation unsuccessful", "Check your syntax")
```

See Also:

DDEExecute, **DDEInitiate**, **DDEPoke**, **DDETerminate**, **DDETimeout**

DDETerminate

Closes a DDE channel.

Syntax:

DDETerminate (channel)

Parameters:

(i) channel same integer that was returned by **DDEInitiate**.

Returns:

(i) always 1.

This function closes a communications channel that was opened with **DDEInitiate**.

Example:

```
Run("wincheck.exe", "TUT")
channel = DDEInitiate("WinCheck", "TUT")
If channel == 0 Then Goto failed
output = DDERequest(channel, "GetChecking")
DDETerminate(channel)
WinClose("WinCheck")
If output == "" Then Goto Failed
Message("Account balance", output)
Exit
:failed
Message("DDE operation unsuccessful", "Check your syntax")
```

See Also:

DDEExecute, **DDEInitiate**, **DDEPoke**, **DDERequest**, **DDETimeout**

DDETimeout

Sets the DDE timeout value.

Syntax:

DDETimeout (value)

Parameters:

(i) value DDE timeout time.

Returns:

(i) previous timeout value.

Sets the timeout time for subsequent DDE functions to specified value in milliseconds (1/1000 second). Default is 3000 milliseconds (3 seconds). If the time elapses with no response, the WIL Interpreter will return an error. The value set with **DDETimeout** stays in effect until changed by another **DDETimeout** statement or until the WIL program ends, whichever comes first.

Example:

```
DDETimeout(5000)
  Run("wincheck.exe", "TUT")
  channel = DDEInitiate("WinCheck", "TUT")
  If channel == 0 Then Goto failed
  output = DDERequest(channel, "GetChecking")
  DDETerminate(channel)
  WinClose("WinCheck")
If output == "" Then Goto Failed
  Message("Account balance", output)
  Exit
:failed
  Message("DDE operation unsuccessful", "Check your syntax")
```

See Also:

DDEExecute, **DDEInitiate**, **DDEPoke**, **DDERequest**, **DDETerminate**

Debug

Controls the debug mode.

Syntax:

Debug (mode**)**

Parameters:

mode = **@ON** or **@OFF**

Returns:

(integer) previous debug mode

Use this function to turn the debug mode on or off. The default is **@OFF**.

When debug mode is on, the interpreter will display the statement just executed, its result (if any), any error conditions, and the next statement to execute.

The statements are displayed in a special dialog box. As you can see in the **Example** section following, the dialog box gives the user four options: Next, Run, Cancel and Show Var**.**

Next executes the next statement and remains in debug mode.

Run exits debug mode and runs the rest of the program normally.

Cancel terminates the current batch file.

Show Var displays the contents of a variable whose name the user entered in the edit box.

Example:

```
Debug(@ON)

a = 6

  q = AskYesNo("Testing Debug Mode", "Is the Pope Catholic")

Debug(@OFF)

b = a + 4
```

produces:

... then, if the user presses **Next**:

... and presses **Next** again:

... and then presses **Yes**:

etc. (If the user had pressed **No** it would have said "VALUE=>0".)

See Also:

ErrorModeErrorMode.cmd, **LastError**LastError.cmd

Delay

Pauses execution for a specified amount of time.

Syntax:

Delay (seconds**)**

Parameters:

seconds = integer seconds to delay (**2** - **15**)

Returns:

(integer) always **@TRUE**

This function causes the currently-executing batch file to be suspended for the specified period of time. **Seconds** must be an integer between **2** and **15**. Smaller or larger numbers will be adjusted accordingly.

Example:

```
Message("Wait", "About 15 seconds")

Delay(15)

Message("Hi", "I'm Baaaaaaack")
```

See Also:

YieldYield.cmd

Dialog

Displays a custom dialog box. The WIL Language dialog editor makes dialog creation automatic. This is an accessory furnished to registered users of WIL Language products. You will not need the dialog editor to create your custom dialogs, the instructions here are sufficient. Even if you are using the editor to define your dialogs, you may find here the quick instructions you need to make small changes.

Syntax:

Dialog (dialog-name)

Parameters:

(s) dialog-name name of the dialog box.

Returns:

(i) value of the pushbutton used to close the dialog box.

The text which follows describes how to define a dialog box for use by the **Dialog** function. Please refer to your product-specific documentation for any additional information which may supplement or supersede that which is described here.

Before the **Dialog** function is called, you must include a section of code in your WIL program which will define the characteristics of the dialog box to be displayed. First of all, the dialog must be declared, and a name must be assigned to it. This is done with a line of the following format:

<name>Format="WWWDLGED,4.0"

where <name> is the dialog name. This should follow the standard rules for WIL variable names, and may not exceed 17 characters in length.

Next, the format of the dialog box is defined, as follows:

<name>X=<x-origin>
<name>Y=<y-origin>
<name>Width=<box-width>
<name>Height=<box-height>
<name>NumControls=<control-count>
<name>Caption="<box-caption>"

where:

<name> is the name of the dialog box, as described above.

<x-origin> is the horizontal coordinate of the upper left corner of dialog box.

<y-origin> is the vertical coordinate of the upper left corner of the dialog box.

<box-width> is the width of the dialog box.

<box-height> is the height of the dialog box.

<control-count> is the total number of controls in the dialog box (see below).

<box-caption> is the text which will appear in the title bar of the dialog box.

Finally, you will need to define the objects, or **controls**, which will appear inside the dialog box. Each control is defined with a line of the following format:

<name>nn=`x,y,width,height,type,var,"text",value`

where:

"nn" is the ordinal position of the control in the dialog box (starting with 1).

"<name>" is the name of the dialog box, as described above.

"x" is the horizontal coordinate of the upper left corner of the control.

"y" is the vertical coordinate of the upper left corner of the control.

"width" is the width of the control.

"height" is the height of the control.

[This should be DEFAULT for all controls except filelistboxes.]

"type" is the type of control, (see below).

"var" is the name of the variable affected by the control.

"text" is the description which will be displayed with the control.

[Use a null string ("") if the control should appear blank.]

"value" is the value returned by the control.

[Use only for pushbuttons, radiobuttons, and checkboxes.]

Note: The numbers used for "x-origin", "y-origin", "box-width", "box-height", "x", "y", "width," and "height" are expressed in a unit of measure known as "Dialog Units." Basically speaking:

1 width unit = 1/4 width of system font.

1 height unit = 1/4 width of system font.

4 units wide = Average width of the system font.

8 units high = Average height of the system font.

There are seven types of controls available:

PUSHBUTTON

A button, which can be labeled and used as desired.

When the user presses a pushbutton, the **Dialog** function will exit and will return the "value" assigned to the button which was pressed. Therefore, you should assign a unique "value" to each pushbutton in a dialog.

Pushbuttons with values of 0 and 1 have special meaning. If the user presses a pushbutton which has a value of **0**, the WIL program will be terminated (or will go to the label marked ":CANCEL", if one is defined); this corresponds to the behavior of the familiar **Cancel** button. A pushbutton with a value of **1** is the default pushbutton, and will be selected if the user presses the **Enter** key; this corresponds to the behavior of the familiar **OK** button.

Note: Every dialog box must contain at least one pushbutton.

For pushbuttons, "var" should be DEFAULT.

RADIOBUTTON

One of a group of circular buttons, only one of which can be "pressed" (filled in) at any given time. You can have more than one group of radio buttons in a dialog box, but each group must use a different "var". When the **Dialog** function exits, the value of "var" will be equal to the "value" assigned to the radiobutton which is pressed. Therefore, you should assign a unique "value" to each radiobutton in a group.

Normally, when a dialog box opens, the default radiobutton in each group (i.e., the one which is pressed) is the one which has a value of 1. You can change this by assigning a different value to "var" before calling the **Dialog** function.

CHECKBOX

A square box, in which an "X" appears when selected. A check box can have a value of **0** (un-checked) or **1** (checked).

Each checkbox in a dialog should use a unique "var".

Normally, when a dialog box opens, every checkbox defaults to being un-checked. You can change this by assigning a value of 1 to "var" before calling the **Dialog** function.

Note for advanced users only:

it is possible to define a group of checkboxes which have the same "var". Each box in the group must have a unique value, which must be a power of 2 (1, 2, 4, etc.). The user can check and uncheck individual checkboxes in the group, and when the **Dialog** function exits the value of "var" will be equal to the values of all the checkboxes in the group, combined using the bitwise OR operator (**|**).

EDITBOX

A box in which text can be typed. Whatever the user types in the editbox will be assigned to the variable "var".

Normally, when a dialog box opens, editboxes are empty. You can change this by assigning a value to the string variable "var" before calling the **Dialog** function, in which case the value of "var" will be displayed in the editbox.

STATICTEXT

Descriptive text, which does not change. This can be used to display titles, instructions, etc.

For statictext controls, "var" should be DEFAULT.

VARYTEXT

Variable text. The current value of "var" is displayed. If "var" is not assigned a value in the WIL program before calling the **Dialog** function, the "text" field of the control definition will be used.

FILELISTBOX

A file selection listbox. This will allow the user to select a file from any directory or drive on the system.

The value of "var" will be set to the selected filename; if you need to know what directory the file is in, use the **DirGet** function after the **Dialog** function exits.

Normally, when a dialog box opens, filelistboxes display files matching a filemask of "*.*" (i.e., all

files). You can change this by assigning a different filemask value to the string variable"var" before calling the **Dialog** function.

In combination with the filelistbox, you can include an EDITBOX control which has the same "var" name as the filelistbox. If you do, the user can type a filemask into the editbox (eg., "*.TXT"), which will cause the filelistbox to be redrawn to display only those files which match the specified filemask.

Also in combination with the filelistbox, you can include a VARYTEXT control which has the same "var" name as the filelistbox. If you do, this control will show the name of the directory currently displayed in the filelistbox.

For filelistboxes, "text" should be DEFAULT.

Note: You can have only one filelistbox in a dialog.

Example:

```
GeorgeFormat=`WWWDLGED,4.0`

GeorgeCaption=`Sample Dialog`
GeorgeX=56
GeorgeY=72
GeorgeWidth=228
GeorgeHeight=113
GeorgeNumControls=12

George01=`20,88,64,DEFAULT,PUSHBUTTON,DEFAULT,"&Ok",1`
George02=`140,88,64,DEFAULT,PUSHBUTTON,DEFAULT,"&Cancel",0`
George03=`8,6,98,DEFAULT,STATICTEXT,DEFAULT,"Please enter your        name:"`
George04=`108,6,94,DEFAULT,EDITBOX,UserName,"<Enter your name here>"`
George05=`8,24,108,DEFAULT,STATICTEXT,DEFAULT,"Choose preferred     environment:"`
George06=`18,38,64,DEFAULT,RADIOBUTTON,Rad,"DOS",1`
George07=`18,50,64,DEFAULT,RADIOBUTTON,Rad,"Windows",2`
George08=`18,62,64,DEFAULT,RADIOBUTTON,Rad,"OS/2",3`
George09=`130,24,88,DEFAULT,STATICTEXT,DEFAULT,"Check ones you use:"`
George10=`144,38,64,DEFAULT,CHECKBOX,CHKDOS,"DOS",1`
George11=`144,50,64,DEFAULT,CHECKBOX,CHKWIN,"Windows",1`
George12=`144,62,64,DEFAULT,CHECKBOX,CHKOS2,"OS/2",1`

Dialog("George")
```

See Also:

AskLine, AskPassword, AskYesNo, DialogBox, ItemSelect

DialogBox

Pops up a Windows dialog box defined by the WBD template file. The WIL Language includes a dialog editor that automates the production of dialogs. (If a WIL Language compiler is used to produce executable batch files, and dialog boxes are produced with this DialogBox function, the WBD template file must also be compiled.)

Note: This function has been entirely supplanted by the **Dialog** function, but is retained in the WIL language and documented here for backwards compatibility.

Syntax:

DialogBox ("title", "WDG file"**)**

Parameters:

"title" = the title of the dialog box.

"WDG file" = the name of the WDG template file.

Returns:

(integer) always **0**

Each element in the template file is enclosed in square brackets, and consists of a variable name, followed by one of the following symbols:

Symbol	Meaning	Example
+	check box	[backup+1Save backup]
#	edit box	[newfile#]
****	file selection listbox	[editfile\]
^	radio button	[prog^1Note] [prog^2Write]
$	variable	[var$]

The number following the check box and radio button symbols is the value which will get assigned to the variable if its corresponding box is checked, or button is selected. Following the number is the descriptive text which will appear next to the box or button.

When used in conjunction with a file selection list box variable with the same name, two of these symbols have special meanings:

#	file mask edit box	[editfile#]
$	directory variable	[editfile$]

Anything not appearing within square brackets is displayed as text.

[editfile$]

File mask [editfile#]

[editfile\]

[editfile\]

[editfile\]

[editfile\]

[editfile\]

[backup+1Save backup of file]

[prog^1Notepad] [prog^2WinEdit]

See Also:

Dialog

DirChange

Changes the current directory. Can also log a new drive.

Syntax:

DirChange ([d:]path**)**

Parameters:

"[d:]" = an optional disk drive to log onto.

"path" = the desired path.

Returns:

(integer) **@TRUE** if directory was changed;

@FALSE if the path could not be found.

Use this function to change the current working directory to another directory, either on the same or a different disk drive.

Example:

```
DirChange("c:\")

TextBox("This is your CONFIG.SYS file", "config.sys")
```

See Also:

DirGet, **DirHome**, **LogDisk**

DirGet

Gets the current working directory.

Syntax:

DirGet ( )

Parameters:

(none)

Returns:

(string) = current working directory.

Use this function to determine which directory we are currently in. It's especially useful when changing drives or directories temporarily.

Example:

```
; Get, then restore current working directory

origdir = DirGet()

DirChange("c:\")

FileCopy("config.sys", "%origdir%xxxtemp.xyz", @FALSE)

DirChange(origdir)
```

See Also:

DirHome

DirHome

Returns the directory of the calling program.

Syntax:

DirHome ( )

Parameters:

(none)

Returns:

(string) pathname of the home directory.

Use this function to determine the location of the calling program.

Example:

```
a = DirHome()

Message("The calling program is in ", a)
```

See Also:

DirGet, **DirWindows**

DirItemize

Returns a space-delimited list of directories.

Syntax:

DirItemize (dir-list**)**

Parameters:

"dir-list" = a string containing a set of subdirectory names, which may be wildcarded.

Returns:

(string) list of directories.

This function compiles a list of subdirectories and separates the names with spaces.

This is especially useful in conjunction with the **ItemSelect** function, which enables the user to choose an item from such a space-delimited list.

DirItemize("*.*") returns all dirs

Example:

```
a = DirItemize("*")

ItemSelect("Directories", a, " ")
```

See Also:

FileItemize, **ItemSelect,TextSelect,WinItemize**,

DirMake

Creates a new directory.

Syntax:

DirMake ([d:]path**)**

Parameters:

"[d:]" = the desired disk drive.

"path" = the path to create.

Returns:

(integer) **@TRUE** if the directory was successfully created;

@FALSE if it wasn't.

Use this function to create a new directory.

Example:

```
DirMake("c:\xxxstuff")
```

See Also:

DirRemove, **DirRename**

DirRemove

Removes a directory.

Syntax:

DirRemove (dir-list**)**

Parameters:

"dir-list" = a space-delimited list of directory pathnames.

Returns:

(integer) **@TRUE** if the directory was successfully removed;

@FALSE if it wasn't.

Use this function to delete directories. You can delete one or more at a time by separating directory names with spaces. You cannot, however, use wildcards.

Example:

```
DirRemove("c:\xxxstuff")

DirRemove("tempdir1 tempdir2 tempdir3")
```

See Also:

DirMake, **DirRename**

DirRename

Renames a directory.

Syntax:

DirRename ([d:]oldpath**,** [d:]newpath**)**

Parameters:

"oldpath" = existing directory name, with optional drive.

"newpath" = new name for directory.

Returns:

(integer) **@TRUE** if the directory was successfully renamed;

@FALSE if it wasn't.

```
DirRename("c:\temp", "c:\work")
```

See Also:

DirMake, **DirRemove**

DirWindows

Returns the name of the Windows or Windows System directory.

Syntax:

DirWindows (request#)

Parameters:

(i) request# see below.

Returns:

(s)directory name.

This function returns the name of either the Windows directory or the Windows System directory, depending on the request# specified.

Req#	Return value
0	Windows directory
1	Windows System directory

Example:

```
DirChange(DirWindows(0))

ini = ItemSelect("Select file to edit", FileItemize("*.ini"), " ")

Run("notepad.exe, ini)
```

See Also:

DirGet,DirHome

DiskFree

Finds the total space available on a group of drives.

Syntax:

DiskFree (drive-list**)**

Parameters:

"drive-list" = at least one drive letter, separated by spaces.

Returns:

(integer) the number of bytes available on all the specified drives.

This function takes a string consisting of drive letters, separated by spaces. Only the first character of each non-blank group of characters is used to determine the drives, so you can use just the drive letters, or add a colon (**:**), or add a backslash (\), or even a whole pathname, and still get a perfectly valid result.

Example:

```
size = DiskFree("c d")

Message("Space Available on C: & D:", size)
```

See Also:

DiskScan,**FileSize**

DiskScan

Returns list of drives.

Syntax:

DiskScan (request#)

Parameters:

(i) request# see below.

Returns:

(s)drive list.

Scans disk drives on the system, and returns a space-delimited list of drives of the type specified by request#, in the form "A: B: C: D: ".

The request# is a bitmask, so adding the values together (except for 0) returns all drive types specified; eg., a request# of 3 returns floppy plus local hard drives.

Req#	Return value
0	List of unused disk IDs
1	List of floppy drives
2	List of local hard drives
4	List of remote (network) drives

Example:

```
hd = DiskScan(2)

Message("Hard drives on system", hd)
```

See Also:

DiskFree, **LogDisk**

Display

Displays a message to the user for a specified period of time.

Syntax:

Display (seconds**,** title**,** text**)**

Parameters:

seconds = integer seconds to display the message (**1**-**15**).

"title" = title of the window to be displayed.

"text" = text of the window to be displayed.

Returns:

(integer) always **@TRUE**.

Use this function to display a message for a few seconds, and then continue processing without user input.

Seconds must be an integer between **1** and **15**. Smaller or larger numbers will be adjusted accordingly.

The display box may be prematurely canceled by the user by clicking a mouse button, or hitting any key.

Example:

```
Display(3, "Current window is", WinGetActive())
```

See Also:

Pause, **Message**

DOSVersion

Returns the version numbers of the current version of DOS.

Syntax:

DOSVersion (level)

Parameters:

level = **@MAJOR** or **@MINOR**.

Returns:

(integer) integer or decimal part of DOS version number.

@MAJOR returns the integer part (to the left of the decimal).

@MINOR returns the decimal part (to the right of the decimal).

If the version of DOS in use is **4.0**, then:

DOSVersion(@MAJOR) == **4**

DOSVersion(@MINOR) == **0**

Example:

```
i = DOSVersion(@MAJOR)
d = DOSVersion(@MINOR)
If StrLen(d) == 1 Then d = StrCat("0", d)
Message("DOS Version", "%i%.%d%")
```

See Also:

Environment, **Version**, **WinVersion**

Drop

Removes variables from memory.

Syntax:

Drop (var**,** [var]...**)**

Parameters:

var = variable names to remove.

Returns:

(integer) always **@TRUE**.

This function removes variables from the language processor's variable list, and recovers the memory associated with the variable (and possibly related string storage).

Example:

```
a = "A variable"

b = "Another one"

Drop(a, b)                ; This removes A and B from memory
```

Else

Continues a previous **If** statement.

Syntax:

Else statement

Parameters:

(s) statement any valid WIL function or command.

This command continues the last-encountered **If** command. It allows the user to specify an alternate action to be taken if the **If** condition was false. If the previous **If** condition was **false**, the statement following the **Else** keyword is executed. If the previous **If** condition was true, the statement following the **Else** keyword is ignored.

Example:

```
windir = DirWindows(0)
inifiles = FileItemize("%windir%*.ini")
ini = ItemSelect("INI file to edit", inifiles, " ")
If ini == "" Then Exit
Else Run("notepad.exe", ini)
```

See Also:

Goto, **If** ... Then, **Then**

EndSession

Ends the Windows session.

Syntax:

EndSession ( )

Parameters:

(none)

Returns:

(integer) always **0**.

Use this command to end the Windows session. This command is equivalent to closing the **Program Manager** window.

Example:

```
sure = AskYesNo ("End Session", "You SURE you want to exit        Windows?")

If sure == @No Then Goto cancel

EndSession()

:cancel

Message("", "Exit Windows canceled")
```

See Also:

Exit, **WinClose**, **WinCloseNot**

Environment

Gets a DOS environment variable.

Syntax:

Environment (env-variable**)**

Parameters:

"env-variable" = any defined environment variable.

Returns:

(string) environment variable contents.

Use this function to query the DOS environment.

Note: It is **not** possible to change a DOS environment variable from within Windows.

Example:

```
; Display the PATH for this DOS session
currpath = Environment("PATH")
Message("Current DOS Path", currpath)
```

See Also:

IniRead, **Version**, **WinVersion**,**WinMetrics**, **WinParmGet**

ErrorMode

Specifies how to handle errors.

Syntax:

ErrorMode (mode**)**

Parameters:

mode = **@CANCEL** or **@NOTIFY** or **@OFF**.

Returns:

(integer) previous error setting.

Use this function to control the effects of runtime errors. The default is **@CANCEL**, meaning the execution of the batch file will be canceled for any error.

@CANCEL: All runtime errors will cause execution to be canceled. The user will be notified which error occurred.

@NOTIFY: All runtime errors will be reported to the user, and the user can choose to continue if it isn't fatal.

@OFF: Minor runtime errors will be suppressed. Moderate and fatal errors will be reported to the user. User has the option of continuing if the error is not fatal.

In general, we suggest the normal state of the program should be **ErrorMode(@CANCEL)**, especially if you are writing a batch file for others to use. You can always suppress errors you expect will occur and then re-enable **ErrorMode (@CANCEL)**.

Example:

```
; Delete xxxtest.xyz.  If file doesn't exist,

; continue execution; don't stop

prevmode = ErrorMode(@OFF)

  FileDelete("c:\xxxtest.xyz")

ErrorMode(prevmode)
```

See Also:

Debug, **LastError**, **Execute**, **Error List**

Exclusive

Controls whether or not other Windows programs will get any time to execute.

Syntax:

Exclusive (mode)

Parameters:

mode = **@ON** or **@OFF**.

Returns:

(integer) previous Exclusive mode.

Exclusive(@OFF) is the default mode. In this mode,the interpreter is well-behaved toward other Windows applications.

Exclusive(@ON) allows WIL files to run somewhat faster, but causes the interpreter to be "greedier" about sharing processing time with other active Windows applications. For the most part, this mode is useful only when you have a series of WIL statements which must be executed in quick succession.

Example:

```
Exclusive(@ON)
x = 0
  start = DateTime()
:add
x = x + 1
  If x < 1000 Then Goto add
stop = DateTime()
  crlf = StrCat(Num2Char(13), Num2Char(10))
  Message("Times", "Start: %start%%crlf%Stop:    %stop%")
Exclusive(@OFF)
```

Execute

Executes a statement in a protected environment. Any errors encountered are recoverable.

Syntax:

Execute statement

Parameters:

"statement" = is (hopefully) an executable statement.

Use this command to execute computed or user-entered statements. Due to the built-in error recovery associated with **Execute**, it is ideal for interactive execution of user-entered commands.

Note that the **Execute** command doesn't operate on a string, *per se*, but rather on a direct statement. If you want to put a code segment into a string variable, you must use the substitution feature of the language, as in the example below.

Example:

```
cmd = ""

  cmd = AskLine("WIL Interactive", "Command:", cmd)

Execute %cmd%
```

Exit

Terminates the batch file being interpreted.

Syntax:

Exit

Use this command to prematurely exit a batch file process. An **exit** is implied at the end of each batch file.

Example:

```
a = 100

Message("The value of a is", a)

Exit
```

See Also:

Pause

FileAppend

Appends one or more files to another file.

Syntax:

FileAppend (source-list**,** destination**)**

Parameters:

"source-list" = a string containing one or more filenames, which may be wildcarded.

"destination" = target file name.

Returns:

(integer) **@TRUE** if all files were appended successfully;

@FALSE if at least one file wasn't appended.

Use this function to append an individual file or a group of files to the end of an existing file. If "destination" does not exist, it will be created.

The file(s) specified in "source-list" will not be modified by this function.

"Source-list" may contain ***** and **?** wildcards. "Destination" may not contain wildcards of any type; it must be a single file name.

Example:

```
FileAppend("c:\config.sys", "c:\misc\config.sav")

DirChange("c:\batch")

FileDelete("allbats.fil")

FileAppend("*.bat", "allbats.fil")
```

See Also:

FileCopy, **FileDelete**, **FileExist**

FileAttrGet

Returns file attributes.

Syntax:

FileAttrGet (filename)

Parameters:

(s) filename file whose attributes you want to determine.

Returns:

(s)attribute settings.

Returns attributes for the specified file, in a string of the form "RASH". This string is composed of four individual attribute characters, as follows:

Char	Symbol	Meaning
1	R	Read-only ON
2	A	Archive ON
3	S	System ON
4	H	Hidden ON

A hyphen in any of these positions indicates that the specified attribute is OFF. For example, the string "-A-H" indicates a file which has the Archive and Hidden attributes set.

Example:

```
editfile = "c:\config.sys"

  attr = FileAttrGet(editfile)

  If StrSub(attr, 1, 1) == "R" Then Goto readonly

  Run("notepad.exe", editfile)

  Exit

:readonly

  Message("File is read-only", "Cannot edit %editfile%")
```

See Also:

FileAttrSet, **FileTimeGet**

FileAttrSet

Sets file attributes.

Syntax:

FileAttrSet (file-list, settings)

Parameters:

(s) file-list space-delimited list of files.

(s) settings new attribute settings for those file(s).

Returns:

(i) always 0.

The attribute string consists of one or more of the following characters (an upper case letter turns the specified attribute ON, a lower case letter turns it OFF):

R read only ON

A archive ON

S system ON

H hidden ON

r read only OFF

a archive OFF

s system OFF

h hidden OFF

Example:

```
FileAttrSet("win.ini system.ini", "rAsH")

FileAttrSet("c:\command.com", "R")
```

See Also:

FileAttrGet, **FileTimeTouch**

FileClose

Closes a file.

Syntax:

FileClose (filehandle)

Parameters:

filehandle = same integer that was returned by **FileOpen**.

Returns:

(integer) always 0.

Example:

```
; the hard way to copy an ASCII file
  old = FileOpen("config.sys", "READ")
new = FileOpen("sample.txt", "WRITE")
:top
  x = FileRead(old)
  If x != "*EOF*" Then FileWrite(new, x)
If x != "*EOF*" Then Goto top
FileClose(new)
FileClose(old)
```

See Also:

FileOpen, **FileRead**, **FileWrite**

FileCopy

Copies files.

Syntax:

FileCopy (source-list**,** destination**,** warning**)**

Parameters:

"source-list" = a string containing one or more filenames, which may be wildcarded.

"destination" = target file name.

warning = **@TRUE** if you want a warning before overwriting existing files;

@FALSE if no warning desired.

Returns:

(integer) **@TRUE** if all files were copied successfully;

@FALSE if at least one file wasn't copied.

Use this function to copy an individual file, a group of files using wildcards, or several groups of files by separating the names with spaces.

You can also copy files to any **COM** or **LPT** device.

"Source-list" may contain ***** and **?** wildcards. "Destination" may contain the ***** wildcard only.

Example:

```
FileCopy("c:\config.sys", "d:", @FALSE)

FileCopy("c:\*.sys", "d:devices\*.sys", @TRUE)

FileCopy("c:\config.sys", "LPT1", @FALSE)
```

See Also:

FileDelete, **FileExist**, **FileLocate**, **FileMove**, **FileRename**

FileDelete

Deletes files.

Syntax:

FileDelete (file-list**)**

Parameters:

"file-list" = a string containing one or more filenames, which may be wildcarded.

Returns:

(integer) **@TRUE** if all the files were deleted;

@FALSE if a file didn't exist or is marked with the READ-ONLY attribute.

Use this function to delete an individual file, a group of files using wildcards, or several groups of files by separating the names with spaces.

Example:

```
FileDelete("*.bak temp???.fil")
```

See Also:

FileExist, **FileLocate**, **FileMove**, **FileRename**

FileExist

Tests for the existence of files.

Syntax:

FileExist (filename**)**

Parameters:

"filename" = either a fully qualified filename with drive and path, or just a filename and extension.

Returns:

(integer) **@TRUE** if the file exists;

@FALSE if it doesn't or if the pathname is invalid.

This function is used to test whether or not a specified file exists.

If a fully-qualified file name is used, only the specified drive and directory will be checked for the desired file. If only the root and extension are specified, then first the current directory is checked for the file, and then, if the file is not found in the current directory, all directories in the DOS path are searched.

Example:

```
; check for file in current directory

  fex = FileExist(StrCat(DirGet(), "myfile.txt"))

  tex = StrSub("NOT", 1, StrLen("NOT") * fex)

  Message("MyFile.Txt"," Is %tex%in the current directory")

; check for file someplace along path

fex = FileExist("myfile.txt")

  tex = StrSub("NOT", 1, StrLen("NOT") * fex)

Message("MyFile.Txt", " Is %tex% in the DOS path")
```

See Also:

FileLocate

FileExtension

Returns extension of file.

Syntax:

FileExtension (filename)

Parameters:

"filename" = [optional path]complete file name, with extension.

Returns:

(string) file extension.

FileExtension parses the passed filename and returns the extension part of the filename.

Example:

```
; prevent the user from editing a COM or E file
  allfiles = FileItemize("*.*")
  editfile = ItemSelect("Select file to edit", allfiles, " ")
ext = FileExtension(editfile)
  If (ext == "com") || (ext == "exe") Then Goto noedit
  Run("notepad.exe", editfile)
  exit
:noedit
  Message("Sorry", "You may not edit a program file")
```

See Also:

FileRoot, **FilePath**

FileItemize

Returns a space-delimited list of files.

Syntax:

FileItemize (file-list**)**

Parameters:

"file-list" = a string containing a list of filenames, which may be wildcarded.

Returns:

(string) space-delimited list of files.

This function compiles a list of filenames and separates the names with spaces.

This is especially useful in conjunction with the **ItemSelect** function, which lets the user choose an item from such a space-delimited list.

Example:

```
FileItemize("*.bak")                    ;all BAK files

FileItemize("*.arc *.zip *.lzh")        ;compressed files

; Get which.INI file to edit

ifiles = FileItemize("c:\windows\*.ini")

ifile = ItemSelect(".INI Files", ifiles, " ")

  RunZoom("notepad", ifile)

  Drop(ifiles, ifile)
```

See Also:

DirItemize, **WinItemize**, **ItemSelect**

FileLocate

Finds file in current directory or along the DOS path.

Syntax:

FileLocate (filename**)**

Parameters:

"filename" = root name, "**.**", and extension.

Returns:

(string) fully-qualified path name.

This function is used to obtain the fully qualified path name of a file. The current directory is checked first, and if the file is not found, the DOS path is searched. The first occurrence of the file is returned.

Example:

```
; Edit WIN.INI

winini = FileLocate("win.ini")

  If winini == "" Then Goto notfound

  Run("notepad.exe", winini)

Exit

:notfound

  Message("???", "WIN.INI not found")
```

See Also:

FileExist

FileMove

Moves files.

Syntax:

FileMove (source-list**,** destination**,** warning**)**

Parameters:

"source-list" = one or more filenames separated by spaces.

"destination" = target filename.

warning = **@TRUE** if you want a warning before overwriting existing files;

@FALSE if no warning desired.

Returns:

(integer) **@TRUE** if the file was moved;

@FALSE if the source file was not found or had the READ-ONLY attribute, or the target filename is invalid.

Use this function to move an individual file, a group of files using wildcards, or several groups of files by separating the names with spaces.

You can also move files to another drive, or to any **COM** or **LPT** device.

"Source-list" may contain ***** and **?** wildcards. "Destination" may contain the ***** wildcard only.

Example:

```
FileMove("c:\config.sys", "d:", @FALSE)

FileMove("c:\*.sys", "d:*.sys", @TRUE)
```

See Also:

FileCopy, **FileDelete**, **FileExist**, **FileLocate**, **FileRename**

FileOpen

Opens a STANDARD ASCII (only) file for reading or writing.

Syntax:

FileOpen (filename, open-type)

Parameters:

"filename" = name of the file to open.

open-type = **READ** or **WRITE**.

Returns:

(special integer) filehandle

The "**filehandle**" returned by the **FileOpen** function is subsequently used by the **FileRead**, **FileWrite**, and **FileClose** functions.

Example:

```
; To open for reading:

FileOpen("stuff.txt", "READ")

; To open for writing:

FileOpen("stuff.txt", "WRITE")
```

See Also:

FileClose, **FileRead**, **FileWrite**

FilePath

Returns path of file.

Syntax:

FilePath (filename)

Parameters:

"filename" = fully qualified file name, including path.

Returns:

(string) fully qualified path name.

FilePath parses the passed filename and returns the drive and path of the file specification, if any.

Example:

```
coms = Environment("COMSPEC")
compath = FilePath(coms)
Message("", "Your command processor is located in the %compath% directory")
```

See Also:

FileRoot, **FileExtension**

FileRead

Reads data from a file.

Syntax:

FileRead (filehandle)

Parameters:

filehandle = same integer that was returned by **FileOpen**.

Returns:

(string) line of data read from file.

When the end of the file is reached, the string ***EOF*** will be returned.

Example:

```
handle = FileOpen("autoexec.bat", "READ")
:top
line = FileRead(handle)
Display(4, "AUTOEXEC DATA", line)
If line != "*EOF*" Then Goto top
FileClose(handle)
```

See Also:

FileOpen, **FileClose**, **FileWrite**

FileRename

Renames files.

Syntax:

FileRename (source-list**,** destination**)**

Parameters:

"source-list" = one or more filenames, separated by spaces.

"destination" = target filename.

Returns:

(integer) **@TRUE** if the file was renamed;

@FALSE if the source file was not found or had the READ-ONLY attribute, or the target filename is invalid.

Use this function to rename an individual file, a group of files using wildcards, or several groups of files by separating the names with spaces.

Note: Unlike **FileMove**, you cannot make a file change its resident disk drive with **FileRename**.

"Source-list" may contain ***** and **?** wildcards. "Destination" may contain the ***** wildcard only.

Example:

```
FileRename("c:\config.sys", "config.old")

FileRename("c:\*.txt", "*.bak")
```

See Also:

FileCopy, **FileExist**, **FileLocate**, **FileMove**

FileRoot

Returns root of file.

Syntax:

FileRoot (filename)

Parameters:

"filename" = [optional path]complete file name, with extension.

Returns:

(string) file root.

FileRoot parses the passed filename and returns the root part of the filename.

Example:

```
allfiles = FileItemize("*.*")
editfile = ItemSelect("Select file to edit", allfiles, " ")
root = FileRoot(editfile)
ext = FileExtension(editfile)
lowerext = StrLower(ext)
nicefile = StrCat(root, ".", lowerext)
Message("", "You are about to edit %nicefile%.")
Run("notepad.exe", editfile)
```

See Also:

FileExtension, **FilePath**

FileSize

Finds the total size of a group of files.

Syntax:

FileSize (file-list**)**

Parameters:

"file-list" = zero or more filenames, separated by spaces.

Returns:

(integer) total bytes taken up by the specified files.

This function returns the total size of the specified files. Note that it doesn't handle wildcarded filenames. You can, however, use **FileItemize** on a wildcarded filename and use the resulting string as a **FileSize** parameter.

Example:

```
size = FileSize(FileItemize("*.*"))

Message("Size of All Files in Directory", size)
```

See Also:

DiskFree

FileTimeGet

Returns file date and time.

Syntax:

FileTimeGet (filename)

Parameters:

(s) filename name of file for which you want the date and time.

Returns:

(s)file date and time.

This function will return the date and time of a file, in a pre-formatted string. The format it is returned in depends on the date format specified in the [International] section of the WIN.INI file:

ddd mm:dd:yy hh:mm:ss XX

ddd dd:mm:yy hh:mm:ss XX

ddd yy:mm:dd hh:mm:ss XX

Where:

ddd is day of the week (e.g. Mon)

mm is the month (e.g. 10)

dd is the day of the month (e.g. 23)

yy is the year (e.g. 90)

hh is the hours

mm is the minutes

ss is the seconds

XX is the Day/Night code (e.g. AM or PM)

The WIN.INI file will be examined to determine which format to use. You can adjust the WIN.INI file via the **International** section of **Control Panel** if the format isn't what you prefer.

Example:

```
oldtime = FileTimeGet("win.ini")

Run("notepad.exe", "win.ini")

WinWaitClose("Notepad - WIN.INI")
```

```
newtime = FileTimeGet("win.ini")
If StrCmp(oldtime, newtime) == 0 Then Exit
Message("", "WIN.INI has been changed")
```

See Also:

FileAttrGet, FileTimeTouch

FileTimeTouch

Sets file(s) to current time.

Syntax:

FileTimeTouch (file-list)

Parameters:

(s) file-list a space-delimited list of files

Returns:

(i) always 0

"File-list" is a space-delimited list of files, which may not contain wildcards. The path is searched if the file is not found in current directory and if the directory is not specified in "file-list".

Example:

```
FileTimeTouch("wac.c wac.rc")

Run("make.exe", "-fwac.mak")
```

See Also:

FileAttrSet, **FileTimeGet**

FileWrite

Writes data to a file.

Syntax:

FileWrite(filehandle, output-data)

Parameters:

filehandle = same integer that was returned by **FileOpen**.

"output-data" = data to write to file.

Returns:

(integer) always 0.

Example:

```
handle = FileOpen("stuff.txt", "WRITE")
FileWrite(handle, "Gobbledygook")
FileClose(handle)
```

See Also:

FileOpen, **FileClose**, **FileRead**

Goto

Changes the flow of control in a batch file.

Syntax:

Goto label

Parameters:

"label" = user-defined identifier.

Goto *label* causes an unconditional branch to the batch file line marked *:label*, where the identifier is preceded by a colon (**:**).

Example:

```
If WinExist("Solitaire") == @FALSE Then Goto open

WinActivate("Solitaire")

Goto loaded

:open

Run("sol.exe", "")

:loaded
```

See Also:

If...Then

IconArrange

Rearranges icons.

Syntax:

IconArrange ( )

Parameters:

(none)

Returns:

(i) always 0.

This function rearranges the icons at the bottom of the screen, spacing them evenly. It does not change the order in which the icons appear.

Example:

```
IconArrange ( )
```

See Also:

RunIcon, **WinIconize**, **WinPlaceSet**

If…Then

Conditionally performs a function.

Syntax:

If condition **Then** statement

Parameters:

"condition" = an expression to be evaluated.

"statement" = any valid WIL function or command.

If the condition following the **If** keyword is true, the statement following the **Then** keyword is executed. If the condition following the **If** keyword is false, the statement following the **Then** keyword is ignored.

Example:

```
sure = AskYesNo("End Session", "Really quit Windows?")

If sure == @YES Then EndSession()
```

See Also:

Goto

IgnoreInput

Turns off hardware input to windows.

Syntax:

IgnoreInput(mode)

Parameters:

mode = **@TRUE** or **@FALSE**.

Returns:

(integer) previous IgnoreInput mode.

IgnoreInput causes mouse movements, clicks and keyboard entry to be completely ignored. Good for self-running demos.

Warning: If you are not careful with the use of **IgnoreInput**, you can lock up your computer!

Example:

```
username = AskLine("Hello", "Please enter your name","")
IgnoreInput(@TRUE)
Call("demo.wbt", username)
IgnoreInput(@FALSE)
```

IniDelete

Removes a line or section from WIN.INI.

Syntax:

IniDelete (section, keyname)

Parameters:

(s) section the major heading under which the item is located.

(s) keyname the name of the item to delete.

Returns:

(i) always 0

This function will remove the specified line from the specified section in WIN.INI. You can remove an entire section, instead of just a single line, by specifying a keyword of @WHOLESECTION. Case is not significant in section or keyname.

Example:

```
IniDelete("Desktop", "Wallpaper")

IniDelete("Quicken",@WHOLESECTION)
```

See Also:

IniDeletePvt, **IniItemize**, **IniRead**, **IniWrite**

IniDeletePvt

Removes a line or section from a private INI file.

Syntax:

IniDeletePvt (section, keyname, filename)

Parameters:

(s) section the major heading under which the item is located.

(s) keyname the name of the item to delete.

(s) filename name of the INI file.

Returns:

(i) always 0.

This function will remove the specified line from the specified section in a private INI file. You can remove an entire section, instead of just a single line, by specifying a keyword of @WHOLESECTION. Case is not significant in section or keyname.

Example:**text:**

```
IniDeletePvt("Current Users", "Excel", "meter.ini")
```

See Also:

IniDelete, IniItemizePvt, IniReadPvt, IniWritePvt

IniItemize

Lists keywords or sections in WIN.INI.

Syntax:

IniItemize (section)

Parameters:

(s) section the major heading to itemize.

Returns:

(s)list of keywords or sections.

IniItemize will scan the specified section in WIN.INI, and return a space-delimited list of all keyword names contained within that section. If a null string ("") is given as the section name, **IniItemize** will return a list of all section names contained within WIN.INI. Case is not significant in section names.

Example:

```
; Returns all keywords in the [Extensions] section

keywords = IniItemize("Extensions")

; Returns all sections in the entire WIN.INI file

sections = IniItemize("")
```

See Also:

IniDelete, **IniItemizePvt**, **IniRead**, **IniWrite**

IniItemizePvt

Lists keywords or sections in a private INI file.

Syntax:

IniItemizePvt (section, filename)

Parameters:

(s) section the major heading to itemize.

(s) filename name of the INI file.

Returns:

(s)list of keywords or sections.

IniItemizePvt will scan the specified section in a private INI file, and return a space-delimited list of all keyword names contained within that section. If a null string ("") is given as the section name, **IniItemizePvt** will return a list of all section names contained within the file. Case is not significant in section names.

Example:

```
; Returns all keywords in the [Boot] section of SYSTEM.INI

keywords = IniItemizePvt("Boot", "system.ini")
```

See Also:

IniDeletePvt, IniItemize, IniReadPvt, IniWritePvt

IniRead

Reads data from the WIN.INI file.

Syntax:

IniRead (section**,** keyname**,** default**)**

Parameters:

"section" =the major heading to read the data from.

"keyname = the name of the item to read.

"default" = string to return if the desired item is not found.

Returns:

(string) data from WIN.INI file.

This function allows a program to read data from the **WIN.INI** file.

The WIN.INI file has the form:

[section**]**

keyname**=**settings

Most of the entries in WIN.INI are set from the Windows **Control Panel** program, but individual applications can also use it to store option settings in their own sections.

Example:

```
; Find the default output device

a = IniRead("windows", "device", "No Default")

Message("Default Output Device", a)
```

See Also:

IniWrite, **IniReadPvt**, **IniWritePvt**, **Environment**

IniReadPvt

Reads data from a private INI file.

Syntax:

IniReadPvt (section, keyname, default, filename)

Parameters:

"section" =the major heading to read the data from.

"keyname = the name of the item to read.

"default" = string to return if the desired item is not found.

"filename" = name of the INI file.

Returns:

(string) data from the INI file.

Looks up a value in the "filename".INI file. If the value is not found, the "default" will be returned.

Example:

IniReadPvt("Main", "Lang", "English", "WB.INI")

Given the following segment from WB.INI:

[Main]

Lang=French

The batch file line above would return:

French

See Also:

IniWritePvt, **IniRead**, **IniWrite**

IniWrite

Writes data to the **WIN.INI** file.

Syntax:

IniWrite (section**,** keyname**,** data**)**

Parameters:

"section" = major heading to write the data to.

"keyname = name of the data item to write.

"data" = string to write to the WIN.INI file.

Returns:

(integer) always **@TRUE.**

This command allows a program to write data to the **WIN.INI** file. The "section" is added to the file if it doesn't already exist.

Example:

```
; Change the list of pgms to load upon Windows

; startup

  loadprogs = IniRead("windows", "load", "")

  newprogs = AskLine("Add Pgm To LOAD= Line", "Add:", loadprogs)

IniWrite("windows", "load", newprogs)
```

See Also:

IniRead, **IniReadPvt**, **IniWritePvt**

IniWritePvt

Writes data to a private INI file.

Syntax:

IniWritePvt (section, keyname, data, filename)

Parameters:

"section" = major heading to write the data to.

"keyname = name of the data item to write.

"data" = string to write to the INI file.

"filename" = name of the INI file.

Writes a value in the "filename".INI file.

Example:

IniWritePvt("Main", "Lang", "French, "WB.INI")

This would create the following entry in WB.INI:

[Main]

Lang=French

See Also:

IniReadPvt, **IniRead**, **IniWrite**

IntControl

Internal control functions.

Syntax:

IntControl (request#, p1, p2, p3, p4)

Parameters:

(i) request# specifies which sub-function is to be performed (see below).

(s) p1 - p4 parameters which may be required by the function (see below).

Returns:

(s)varies (see below).

Short for Internal Control, a special function that permits numerous internal operations in products that use the WIL Language The first parameter of IntControl defines exactly what the function does, the other parameters are possible arguments to the function.

Warning: Many of these operations are useful only under special circumstances, and/or by technically knowledgeable users. Some could lead to adverse side effects. If it isn't clear to you what a particular function does, don't use it.

IntControl (1, p1, 0, 0, 0)

Just a test IntControl. It echoes back P1 & P2 and P3 & P4 in a pair of message boxes.

IntControl (4, p1, 0, 0, 0)

Controls whether or not a dialog box with a file listbox in it has to return a file name, or may return merely a directory name or nothing.

P1	Meaning
0	May return nothing, or just a directory name
1	Must return a file name (default)

IntControl (5, p1, 0, 0, 0)

Controls whether system & hidden files are seen and processed.

P1	Meaning
0	System & Hidden files not used (default)
1	System & Hidden files seen and used

IntControl (10, p1, 0, 0, 0)

Interrogates the Command Extender DLL status

P1	Meaning
0	Command Extender present
	0 No
	1 Yes
1	Command Extender version
	-1 No Extender present
	0 Incompatible extender present
	(other) Extender version code
2	Interpreter's Extender interface code
3	Name of Extender DLL

IntControl (18, 0, 0, 0, 0)

Suspends the program (the WIL Language interpreter program) waiting for some other process to do the equivalent of IntControl(19). **This command will hang your system if used improperly**.

IntControl (19, p1, 0, 0, 0)

Un-suspends a process stopped with IntControl(18). P1 is a window handle (not a window title). Windows handles may be derived from window titles using IntControl(21).

IntControl (21, p1, 0, 0, 0)

Returns window handle of window matching the partial window-name in p1.

IntControl (22, p1, p2, p3, p4)

Issues a Windows "SendMessage".

p1 Window handle to send to

p2 Message ID number (in decimal)

p3 wParam value

p4 assumed to be a character string. String is copied to a GMEM_LOWER buffer, and a LPSTR to the copied string is passed as lParam. The GMEM_LOWER buffer is freed immediately upon return from the SendMessage

IntControl (23, 0, 0, 0, 0)

Issues a windows PostMessage

p1 Window handle

p2 Message ID number (in decimal)

p3 wParam

p4 lParam -- assumed to be numeric

IntControl (66, 0, 0, 0, 0)

Restarts Windows, just like exiting to DOS and typing WIN again. Could be used to restart Windows after editing the SYSTEM.INI file to change video modes.

IntControl (67, 0, 0, 0, 0)

Performs a warm boot of the system, just like <Ctrl-Alt-Del>. Could be used to reboot the system after editing the AUTOEXEC.BAT or CONFIG.SYS files.

Note: IntControl(67) works only in Windows 3.1 or higher. In Windows 3.0, it behaves just like IntControl(66) and restarts Windows.

IsDefined

Determines if a variable name is currently defined.

Syntax:

IsDefined (var**)**

Parameters:

"var" = a variable name.

Returns:

(integer) **@YES** if the variable is currently defined;

@NO if it was never defined or has been dropped.

A variable is defined the first time it appears at the left of an equal sign in a statement. It stays defined until it is explicitly dropped with the **Drop** function, or until the batch file ends.

Example:

```
def = IsDefined(thisvar)

If def == @FALSE Then Message("ERROR!", "Variable not defined")
```

See Also:

Drop

IsKeyDown

Tells about keys/mouse.

Syntax:

IsKeyDown(keycodes)

Parameters:

keycodes = **@SHIFT** and/or **@CTRL**

Returns:

(integer) **@YES** if the key is down.

@NO if the key is not down.

Determines if the **Shift** key or the **Ctrl** key is currently down.

Note: The right mouse button is the same as **Shift**, and the middle mouse button is the same as **Ctrl**.

Example:

IsKeyDown(@SHIFT)

IsKeyDown(@CTRL)

IsKeyDown(@CTRL | @SHIFT)

IsKeyDown(@CTRL & @SHIFT)

See Also:

WaitForKey

IsLicensed

Tells if the WIL interpreter is licensed. This applies only to WIL scripts that are run by an interpreter. This does not apply to scripts compiled with the WIL Compiler.

Syntax:

IsLicensed()

Parameters:

(none)

Returns:

(integer) **@YES** if current version of the WIL interpreter is licensed.

@NO if current version of the WIL interpreter is not licensed.

Returns information on whether or not the current version of the WIL interpreter is a licensed copy.

IsLicensed

See Also:

Version

IsMenuChecked

Determines if a menu item has a checkmark next to it.

Note: This command is not part of the WIL Interpreter package, but is documented here because it has been implemented in many of the shell or file manager-type applications which use the WIL Interpreter.

Syntax:

IsMenuChecked (menuname)

Parameters:

(s) menuname name of the menu item to test.

Returns:

(i) **@YES** if the menu item has a checkmark;
@NO if it doesn't.

You can place a checkmark next to a menu item with the **MenuChange** command, to indicate an option has been enabled. This function lets you determine if the menu item has already been checked or not.

Example:

```
; assume we've defined a "Misc | Prompt Often" menu item
prompt = IsMenuChecked("MiscPromptOften")
ifprompt = StrSub(";", 1, (prompt == @FALSE))
Execute %ifprompt% confirm = AskYesNo("???", "REALLY do this?")
; some risky operation the user has just confirmed they want to do
Execute %ifprompt% Terminate(confirm != @YES, "", "")
```

See Also:

IsMenuEnabled, **MenuChange**

IsMenuEnabled

Note: This command is not part of the WIL Interpreter package, but is documented here because it has been implemented in many of the shell or file manager-type applications which use the WIL Interpreter.

Determines if a menu item has been enabled.

Syntax:

IsMenuEnabled (menuname)

Parameters:

(s) menuname name of the menu item to test.

Returns:

(i) **@YES** if the menu item is enabled;
@NO if it is disabled & grayed.

You can disable a menu item with the **MenuChange** command if you want to prevent the user from choosing it. It shows up on the screen as a grayed item. **IsMenuEnabled** lets you determine if the menu item is currently enabled or not.

Example:

```
; allow editing of autoexec.bat file only if choice enabled
Terminate(!IsMenuEnabled("UtilitiesEditBatFile"), "", "")
Run("notepad.exe", "c:\autoexec.bat")
```

See Also:

IsMenuChecked, **MenuChange**

IsNumber

Determines whether a variable contains a valid number.

Syntax:

IsNumber (string**)**

Parameters:

"string" = string to test to see if it represents a valid number.

Returns:

(integer) **@YES** if it contains a valid number;

@NO if it doesn't.

This function determines if a string variable contains a valid integer. Useful for checking user input prior to using it in computations.

Example:

```
a = AskLine("ISNUMBER", "Enter a number", "0")

If IsNumber(a) == @NO Then Message("", "You didn't enter a          number")
```

See Also:

Abs, **Char2Num**

ItemCount

Returns the number of items in a list.

Syntax:

ItemCount (list**,** delimiter**)**

Parameters:

"list" = a string containing a list of items to choose from.

"delimiter" = a string containing the character to act as delimiter between items in the list.

Returns:

(integer) the number of items in the list.

If you create the list with the **FileItemize** or **DirItemize** functions you will be using a space-delimited list. **WinItemize**, however, creates a tab-delimited list of window titles since titles can have embedded blanks.

Example:

```
a = FileItemize("*.*")

n = ItemCount(a, " ")

Message("Note", "There are %n% files")
```

See Also:

DirItemize, **FileItemize**, **WinItemize**, **ItemExtract**, **ItemSelect**

ItemExtract

Returns the selected item from a list.

Syntax:

ItemExtract (select**,** list**,** delimiter**)**

Parameters:

select = the position in "list" of the item to be selected.

"list" = a string containing a list of items to choose from.

"delimiter" = a string containing the character to act as delimiter between items in the list.

Returns:

(string) the selected item.

If you create the list with the **FileItemize** or **DirItemize** functions you will be using a space-delimited list. **WinItemize**, however, creates a tab-delimited list of window titles since titles can have embedded blanks.

Example:

```
bmpfiles = FileItemize("*.bmp")

bmpcount = ItemCount(bmpfiles, " ")

pos = (Random(bmpcount - 1)) + 1

paper = ItemExtract(pos, bmpfiles, " ")

Wallpaper(paper, @FALSE)
```

See Also:

DirItemize, **FileItemize**, **WinItemize**, **ItemCount**, **ItemSelect**

ItemInsert

Adds an item to a list.

Syntax:

ItemInsert (item, index, list, delimiter)

Parameters:

(s) item a new item to add to **list**.

(i) index the position in **list** after which the item will be inserted.

(s) list a string containing a list of items.

(s) delimiter a character to act as a delimiter between items in the list.

Returns:

(s)new list, with **item** inserted.

This function inserts a new item into an existing list, at the position following **index**. It returns a new list, with the specified item inserted; the original list (**list**) is unchanged. For example, specifying an index of 1 causes the new item to be inserted after the first item in the list; i.e., the new item becomes the second item in the list.

You can specify an index of **0** to add the item to the beginning of the list, and an index of **-1** to append the item to the end of the list.

If you create the list with the **FileItemize** or **DirItemize** functions you will be using a space-delimited list. **WinItemize**, however, creates a tab-delimited list of window titles since titles can have embedded blanks.

Example:

```
newlist = ItemInsert(item, index, list, delimiter)
```

See Also:

ItemCount, **ItemRemove**

ItemLocate

Returns the position of an item in a list.

Syntax:

ItemLocate (item, list, delimiter)

Parameters:

(s) item item to search for in **list**.

(s) list a string containing a list of items.

(s) delimiter a character to act as a delimiter between items in the list.

Returns:

(i) position in **list** of **item**, or **0** if no match found.

This function finds the first occurrence of **item** in the specified list, and returns the position of the item (the first item in a list has a position of 1). If the item is not found, the function will return a **0**.

If you create the list with the **FileItemize** or **DirItemize** functions you will be using a space-delimited list. **WinItemize**, however, creates a tab-delimited list of window titles since titles can have embedded blanks.

Example:

ItemLocate(item, list, delimiter)

See Also:

ItemExtract

ItemRemove

Removes an item from a list.

Syntax:

ItemRemove (index, list, delimiter)

Parameters:

(i) index the position in **list** of the item to be removed.

(s) list a string containing a list of items.

(s) delimiter a character to act as a delimiter between items in the list.

Returns:

(s)new list, with **item** removed.

This function removes the item at the position specified by **index** from a list. The delimiter following the item is removed as well. It returns a new list, with the specified item removed; the original list (**list**) is unchanged.

If you create the list with the **FileItemize** or **DirItemize** functions you will be using a space-delimited list. **WinItemize**, however, creates a tab-delimited list of window titles since titles can have embedded spaces.

Example:

```
newlist = ItemRemove(index, list, delimiter)
```

See Also:

ItemCount, ItemInsert

ItemSelect

Allows the user to choose an item from a listbox.

Syntax:

ItemSelect (title**,** list**,** delimiter**)**

Parameters:

"title" = the title of dialog box to display.

"list" = a string containing a list of items to choose from.

"delimiter" = a string containing the character to act as delimiter between items in the list.

Returns:

(string) the selected item.

This function displays a dialog box with a listbox inside. This listbox is filled with a sorted list of items taken from a string you provide to the function.

Each item in the string must be separated ("delimited") by a character, which you also pass to the function.

The user selects one of the items by either doubleclicking on it, or single-clicking and pressing OK. The item is returned as a string.

If you create the list with the **FileItemize** or **DirItemize** functions you will be using a space-delimited list. **WinItemize**, however, creates a tab-delimited list of window titles since titles can have embedded blanks.

Example:

```
DirChange("e:\word")

alldotfiles = FileItemize("*.dot")

dotfile = ItemSelect("W4W Templates", alldotfiles, " ")

Run("winword.exe", dotfile)
```

Which would produce:

See Also:

AskYesNo, **Display**, **DirItemize**, **FileItemize**, **WinItemize**, **Message**, **Pause**, **TextBox**, **ItemCount**, **ItemExtract**

ItemSort

Sorts a list.

Syntax:

ItemSort (list, delimiter)

Parameters:

(s) list a string containing a list of items.

(s) delimiter a character to act as a delimiter between items in the list.

Returns:

(s)new, sorted list.

This function sorts a list, using an ANSI sort sequence. It returns a new, sorted list; the original list is unchanged.

If you create the list with the **FileItemize** or **DirItemize** functions you will be using a space-delimited list. **WinItemize**, however, creates a tab-delimited list of window titles since titles can have embedded spaces.

Example:

newlist = ItemSort(list, delimiter)

See Also:

ItemExtract

LastError

Returns the most-recent error encountered during the current batch file.

Syntax:

LastError ( )

Parameters:

(none)

Returns:

(integer) most-recent WIL error code encountered.

WIL errors are numbered according to their severity. "Minor" errors go from 1000 through 1999. Moderate errors are 2000 through 2999. Fatal errors are numbered 3000 to 3999.

Depending on which error mode is active when an error occurs, you may not get a chance to check the error code. See **ErrorMode** for a discussion of default error handling.

Don't bother checking for "fatal" error codes. When a fatal error occurs, the batch file is canceled before the next WIL statement gets to execute (regardless of which error mode is active).

Every time the **LastError** function is called, the "last error" indicator is reset to zero.

A full listing of possible errors you can encounter in processing a batch file is in **Appendix B** (pg.).

Example:

```
ErrorMode(@OFF)

FileCopy("data.dat", "c:\backups", @FALSE)

ErrorMode(@CANCEL)

If LastError() == 1006 Then Message("Error", "Please call  Tech Support at 555-9999.")
```

See Also:

Debug, **ErrorMode**

LogDisk

Logs (activates) a disk drive.

Syntax:

LogDisk (drive-letter**)**

Parameters:

"drive-letter" = the disk drive to log into.

Returns:

(integer) **@TRUE** if the current drive was changed;

@FALSE if the drive doesn't exist.

Use this function to change the logged disk drive.

This command produces the same effect as if you typed the drive name from the DOS command prompt.

Example:

```
LogDisk("c:")
```

See Also:

DirChange

Max

Returns largest number in a list of numbers.

Syntax:

Max (integer [**,** integer]...**)**

Parameters:

integer = an integer number.

Returns:

(integer) largest parameter.

Use this function to determine the largest of a set of comma-delimited integers.

Example:

```
a = Max(5, -37, 125, 34, 2345, -32767)

Message("Largest number is", a)
```

See Also:

Abs, **Average**, **Min**

MenuChange

Checks, unchecks, enables, or disables a menu item.

Syntax:

MenuChange (menuname**,** flags**)**

Parameters:

"menuname" = menu item whose status you wish to change.

"flags" = **@CHECK**, **@UNCHECK**,

@ENABLE, or **@DISABLE**.

Returns:

(integer) always **@TRUE**.

There are currently two ways you can modify a menu item:

You can check and uncheck the item to imply that it corresponds to an option that can be turned on or off.

You can temporarily disable the item (it shows up as gray) and later re-enable it.

The two sets of flags (**@Check**/**@UnCheck** and **@Enable**/**@Disable**) can be combined in one function call by using the **|** (or) operator.

Example:

```
MenuChange (FilePrint, @Disable)

MenuChange (WPWrite, @Enable|@Check)
```

See Also:

IsMenuChecked, **IsMenuEnabled**

Message

Displays a message to the user.

Syntax:

Message (title**,** text**)**

Parameters:

"title" = title of the message box.

"text" = text to display in the message box.

Returns:

(integer) always **@TRUE**.

Use this function to display a message to the user. The user must respond by selecting the **OK** button before processing will continue.

Example:

```
Message("Current directory is", DirGet())
```

See Also:

Display, **Pause**

Min

Returns lowest number in a list of numbers.

Syntax:

Min (integer **[,** integer]...**)**

Parameters:

integer = an integer number.

Returns:

(integer) lowest parameter.

Use this function to determine the lowest of a set of comma-delimited integers.

Example:

```
a = Min( 5, -37, 125, 34, 2345, -32767)

Message("Smallest number is", a)
```

See Also:

Abs, **Average**, **Max**

MouseInfo

Returns assorted mouse information.

Syntax:

MouseInfo (request#)

Parameters:

(i) request# see below.

Returns:

(s)see below.

The information returned by **MouseInfo** depends on the value of request#.

Req#	Return value
0	Window name under mouse
1	Top level parent window name under mouse
2	Mouse coordinates, assuming a 1000x1000 virtual screen
3	Mouse coordinates in absolute numbers
4	Status of mouse buttons, as a bitmask:

Binary	Decimal	Meaning
000	0	No buttons down
001	1	Right button down
010	2	Middle button down
011	3	Right and Middle buttons down
100	4	Left button down
101	5	Left and Right buttons down
110	6	Left and Middle buttons down
111	7	Left, Middle, and Right buttons down

For example, if mouse is at the center of a 640x480 screen and above the "Clock" window, and the left button is down, the following values would be returned:

Req#	Return value
1	"Clock"

2 "500 500"

3 "320 240"

4 "4"

Example:

```
Display(1, "", "Press a mouse button to continue")
:loop
buttons = MouseInfo(4)
If buttons == 0 Then Goto loop
If buttons & 4 Then Display(1, "", "Left button was pressed")
If buttons & 1 Then Display(1, "", "Right button was pressed")
```

See Also:

WinMetrics, **WinParmGet**

NetAddCon

Connects network resources to imaginary local disk drives or printer ports.

Syntax:

NetAddCon (net-path, password, local-name)

Parameters:

(s) net-path	net resource or string returned by **x**.
(s) password	password required to access resource, or "".
(s) local-name	local drive name or printer port.

Returns:

(i) @TRUE if successful; @FALSE if unsuccessful.

You can use **NetAddCon** to connect a local drive to a network directory, in which case "local-name" will be a drive name (eg, "Z:"). You can also connect a local printer port to a network print queue, in which case "local-name" will be the name of the printer port (eg, "LPT1").

Use the **NetBrowse** function to obtain a value for "net-path".

If no password is required, use a null string ("") for the "password" parameter.

Example:

```
availdrive = DiskScan(0)
drvlen = StrLen(availdrive)
If drvlen == 0 Then Goto nomore
availdrive = StrSub(availdrive, drvlen - 2, 2)
netpath = NetBrowse(0)
pswd = AskPassword("Enter password for", netpath)
NetAddCon(netpath, pswd, availdrive)
Exit
:nomore
Message("Connect Drive to Net", "No drives avail for assignment")
```

See Also:

NetBrowse, **NetCancelCon**, **NetGetCon**

NetAttach

Attaches to a network file server.

Syntax:

NetAttach (server-name)

Parameters:

(s) server-name name of the network file server.

Returns:

(i) @TRUE if successful; @FALSE if unsuccessful.

This function may not work with all networks.

Example:

```
NetAttach("userapps")
```

See Also:

NetAddCon, **NetDetach**, **NetLogin**

NetBrowse

Displays a dialog box allowing the user to select a network resource.

Syntax:

NetBrowse (request#)

Parameters:

(i) request# see below.

Returns:

(s)see below.

Displays a dialog box allowing the user to select a network resource. Request#=0 allows selection of a print queue and Request#=1 allows selection of a network directory. This function returns a string that can be used by **NetAddCon** to add a connection.

Example:

```
availdrive = DiskScan(0)

drvlen = StrLen(availdrive)

If drvlen == 0 Then Goto nomore

availdrive = StrSub(availdrive, drvlen - 2, 2)

netpath = NetBrowse(0)

pswd = AskPassword("Enter password for", netpath)

NetAddCon(netpath, pswd, availdrive)

Exit

:nomore

Message("Connect Drive to Net", "No drives avail for assignment")
```

See Also:

NetAddCon

NetCancelCon

Breaks a network connection.

Syntax:

NetCancelCon (name, force)

Parameters:

(s) name network resource name or local name.

(i) force force flag (see below).

Returns:

(i) @TRUE if successful; @FALSE if unsuccessful.

If "force" is set to 0, **NetCancelCon** will not break the connection if any files on that connection are still open. If "force" is set to 1, the connection will be broken regardless.

Example:

```
  availdrive = DiskScan(4)

  n = ItemCount(availdrive, " ")This example in plain text:

  If n == 0 Then Exit

i = 1

dislist = ""

:loop

  drv = ItemExtract(i, availdrive, " ")

  dislist = StrCat(drv, Num2Char(9), NetGetCon(drv), "|")

i = i + 1

If i < n Then Goto loop

  availdrive = ItemSelect("Disconnect", dislist, "|")

NetCancelCon(availdrive, 0)
```

See Also:

NetAddCon, NetGetCon

NetDetach

Detaches from a network file server.

Syntax:

NetDetach (server-name)

Parameters:

(s) server-name name of the network file server.

Returns:

(i) @TRUE if successful; @FALSE if unsuccessful.

This function may not work with all networks.

Example:

```
NetDetach("userapps")
```

See Also:

NetAttach, **NetCancelCon**

NetDialog

Brings up the network driver's dialog box.

Syntax:

NetDialog ( )

Parameters:

(none)

Returns:

(i) @TRUE if successful; @FALSE if unsuccessful.

A network driver's dialog box displays copyright information, and may allow access to the network, depending on the particular network driver. The WIL program will wait until the network dialog terminates before continuing.

Example:

```
NetDialog()
```

NetGetCaps

Returns information on network capabilities.

Syntax:

NetGetCaps (request#)

Parameters:

(i) request# see below.

Returns:

(i) see below.

NetGetCaps returns 0 if no network is installed (it is the only network function you can use without having a network installed and not get an error).

Req#	Return value
1	Network driver specification number
2	Type of network installed:
	0 None
	256 MS Network
	512 Lan Manager
	768 Novell NetWare
	1024 Banyan Vines
	1280 10 Net
	(other) Other network
3	Network driver version number
4	Returns 1 if any network is installed
6	Bitmask indicating whether the network driver supports the following connect functions:
	1 AddConnection
	2 CancelConnection
	4 GetConnection
	8 AutoConnect via DOS

16 BrowseDialog

7 Bitmask indicating whether the network driver supports the following print functions:

2 Open Print Job

4 Close Print Job

16 Hold Print Job

32 Release Print Job

64 Cancel Print Job

128 Set number of copies

256 Watch Print Queue

512 Unwatch Print Queue

1024 Lock Queue Data

2048 Unlock Queue Data

4096 Driver will send QueueChanged messages to Print Manager

8192 Abort Print Job

Example:

```
caps = NetGetCaps(6)
If caps & 16 Then Message("", "Your network supports BrowseDialog")
```

See Also:

NetGetUser, **WinConfig**, **WinMetrics**, **WinParmGet**

NetGetCon

Returns the name of a connected network resource.

Syntax:

NetGetCon (local-name)

Parameters:

(s) local-name local drive name or printer port.

Returns:

(s)name of network resource.

NetGetCon returns the name of the network resource currently connected to "local-name".

Example:

```
local = AskLine("NetGetCon", "Enter local drive name", "")
If local == "" Then Exit
resource = NetGetCon(local)
Message("NetGetCon", "%local% is connected to %resource%")
```

See Also:

NetAddCon, **NetCancelCon**

NetGetUser

Returns the name of the user currently logged into the network.

Syntax:

NetGetUser ( )

Parameters:

(none)

Returns:

(s)name of current user.

Example:

IniWritePvt("Current Users", "Excel", NetGetUser(), "usagelog.ini")

Run("excel.exe", "")

See Also:

NetGetCaps

NetLogin

Performs a network login.

Syntax:

NetLogin (server-name, user-name, password)

Parameters:

(s) server-name	name of the network file server.
(s) user-name	name of the current user.
(s) password	password required to access server, or "".

Returns:

(i) @TRUE if successful; @FALSE if unsuccessful.

This function may not work with all networks.

Example:

```
pwd = AskPassword("Hello", "Enter password for network access")
NetLogin("userapps", "admin1", pwd)
```

See Also:

NetAttach, **NetLogout**

NetLogout

Performs a network logout.

Syntax:

NetLogout (server-name)

Parameters:

(s) server-name name of the network file server.

Returns:

(i) @TRUE if successful; @FALSE if unsuccessful.

This function may not work with all networks.

Example:

```
NetLogout("userapps")
```

See Also:

NetLogin

NetMapRoot

Maps a local drive to a network resource.

Syntax:

NetMapRoot (local-name, net-path)

Parameters:

(s) local-name local drive name.

(s) net-path net resource or string returned by **NetBrowse**.

Returns:

(i) @TRUE if successful; @FALSE if unsuccessful.

This function maps a local drive letter as the fake root to a network resource. This is supported by Novell NetWare, but may not work with any other networks.

Example:

```
availdrive = DiskScan(0)
drvlen = StrLen(availdrive)
If drvlen == 0 Then Goto nomore
availdrive = StrSub(availdrive, drvlen - 2, 2)
netpath = NetBrowse(0)
NetMapRoot(availdrive, netpath)
Exit
:nomore
Message("Connect Drive to Net", "No drives avail for assignment")
```

See Also:

NetAddCon, **NetBrowse**, **NetCancelCon**

NetMemberGet

Determines whether the current user is a member of a specific group.

Syntax:

NetMemberGet (server-name, group-name)

Parameters:

(s) server-name name of the network file server.

(s) group-name name of the group.

Returns:

(i) @TRUE if successful; @FALSE if unsuccessful.

This function may not work with all networks.

Example:

```
member = NetMemberGet("userapps", "sales")
If member == @YES Then Run("notepad.exe", "dailyrpt.txt")
```

See Also:

NetMemberSet

NetMemberSet

Sets the current user as a member of a group.

Syntax:

NetMemberSet (server-name, group-name)

Parameters:

(s) server-name name of the network file server.

(s) group-name name of the group.

Returns:

(i) @TRUE if successful; @FALSE if unsuccessful.

This function may not work with all networks.

Example:

```
NetMemberSet("userapps", "sales")
```

See Also:

NetMemberGet

NetMsgAll

Broadcasts a message to all users on the network.

Syntax:

NetMsgAll (server-name, message)

Parameters:

(s) server-name name of the network file server.

(s) message message to be broadcast.

Returns:

(i) @TRUE if successful; @FALSE if unsuccessful.

This function may not work with all networks.

Example:

```
NetMsgAll("userapps", "System going down in 5 minutes.")
```

See Also:

NetMsgSend

NetMsgSend

Sends a message to a specific user on the network.

Syntax:

NetMsgSend (server-name, user-name, message)

Parameters:

(s) server-name	name of the network file server.
(s) user-name	name of the user to whom the message should be sent.
(s) message	message to be sent.

Returns:

(i) @TRUE if successful; @FALSE if unsuccessful.

This function may not work with all networks.

Example:

```
NetMsgSend("userapps", "compmgr", "Are those reports ready yet?")
```

See Also:

NetMsgAll

Num2Char

Converts a number to its character equivalent.

Syntax:

Num2Char (integer**)**

Parameters:

number = any number from **0** to **255**.

Returns:

(string) one-byte string containing the character the number represents.

Use this function to convert a number to its ASCII equivalent.

Example:

```
; Build a variable containing a CRLF combo

crlf = StrCat(Num2Char(13), Num2Char(10))

Message("NUM2CHAR", StrCat("line1", crlf, "line2"))
```

See Also:

Char2Num

ParseData

Parses the passed string, just like passed parameters are parsed.

Syntax:

ParseData (string**)**

Parameters:

"string" = string to be parsed.

Returns:

(integer) number of parameters in "string".

This function breaks a string constant or string variable into new sub-string variables named **param1**, **param2**, etc. (maximum of nine parameters). Blank spaces in the original string are used as delimiters to create the new variables.

Param0 is the count of how many sub-strings are found in "string".

Example:

```
username = AskLine("Hello", "Please enter your name","")

ParseData(username)
```

If the user enters:

Joe Q. User

ParseData would create the following variables:

param1 == **Joe**

param2 == **Q.**

param3 == **User**

param0 == **3**

See Also:

ItemExtract, **StrSub**

Pause

Provides a message to user. User may cancel processing.

Syntax:

Pause (title**,** text**)**

Parameters:

"title" = title of pause box.

"text" = text of the message to be displayed.

Returns:

(integer) always **@TRUE**.

This function displays a message to the user with an exclamation point icon. The user may respond by selecting the **OK** button, or may cancel the processing by selecting **Cancel**.

The **Pause** function is similar to the **Message** function, except for the addition of the **Cancel** button and icon.

Example:

```
Pause("Change Disks", "Insert new disk into Drive A:")
```

See Also:

Display, **Message**

PlayMedia

Controls multimedia devices.

Syntax:

PlayMedia (command-string)

Parameters:

(s) command-string string to be sent to the multimedia device.

Returns:

(s) response from the device.

If the appropriate Windows multimedia extensions are present, this function can control multimedia devices. Valid command strings depend on the multimedia devices and drivers installed. The basic Windows multimedia package has a waveform device to play and record waveforms, and a sequencer device to play MID files. Refer to the appropriate documentation for information on command strings.

Many multimedia devices accept the WAIT or NOTIFY parameters as part of the command string:

WAIT Causes the system to stop processing input until the requested operation is complete. You cannot switch tasks when WAIT is specified.

NOTIFY Causes the WIL program to suspend execution until the requested operation completes. You can perform other tasks and switch between tasks when NOTIFY is specified.

WAIT NOTIFY Same as WAIT

If neither WAIT nor NOTIFY is specified, the multimedia operation is started and control returns immediately to the WIL program.

In general, if you simply want the WIL program to wait until the multimedia operation is complete, use the NOTIFY keyword. If you want the system to hang until the operation is complete, use WAIT. If you just want to start a multimedia operation and have the program continue processing, don't use either keyword.

The return value from **PlayMedia** is whatever string the driver returns. This will depend on the particular driver, as well as on the type of operation performed.

Example:

```
; Plays a music CD on a CDAudio player.  It plays whatever is in the

; drive, from start to finish

stat = PlayMedia("status cdaudio mode")

answer = 1

 If stat == "playing" Then answer = AskYesNo("CD Audio", "CD is

    Playing.  Stop?")
```

```
  If answer == 0 Then Exit
PlayMedia("open cdaudio shareable alias donna notify")
PlayMedia("set donna time format tmsf")
PlayMedia("play donna from 1")
PlayMedia("close donna")
Exit
:cancel
PlayMedia("set cdaudio door open")
```

See Also:

PlayMidi, **PlayWaveForm**

PlayMidi

Plays a MID or RMI sound file.

Syntax:

PlayMidi (filename, mode)

Parameters:

(s) filename name of the MID or RMI sound file.

(i) mode play mode (see below).

Returns:

(i) @TRUE if successful; @FALSE if unsuccessful.

If Windows multimedia sound extensions are present, and MIDI-compatible hardware is installed, this function will play a MID or RMI sound file. If "filename" is not in the current directory and a directory is not specified, the path will be searched to find the file.

If "mode" is set to 0, the WIL program will wait for the sound file to complete before continuing. If "mode" is set to 1, it will start playing the sound file and continue immediately.

Example:

```
PlayMidi("canyon.mid", 1)
```

See Also:

PlayMedia, **PlayWaveForm**

PlayWaveForm

Plays a WAV sound file.

Syntax:

PlayWaveForm (filename, mode)

Parameters:

(s) filename

(i) mode play mode (see below).

Returns:

(i) @TRUE if successful; @FALSE if unsuccessful.

If Windows multimedia sound extensions are present, and waveform-compatible hardware is installed, this function will play a WAV sound file. If "filename" is not in the current directory and a directory is not specified, the path will be searched to find the file. If "filename is not found, the WAV file associated with the "SystemDefault" keyword is played, (unless the "NoDefault" setting is on).

Instead of specifying an actual filename, you may specify a keyword name from the [Sound] section of the WIN.INI file (eg, "SystemStart"), in which case the WAV file associated with that keyword name will be played.

"Mode" is a bitmask, composed of the following bits:

Mode	Meaning
0	Wait for the sound to end before continuing.
1	Don't wait for the sound to end. Start the sound and immediately process more statements.
2	If sound file not found, do not play a default sound
9	Continue playing the sound forever, or until a **PlayWaveForm("", 0)** statement is executed
16	If another sound is already playing, do not interrupt it. Just ignore this PlayWaveForm request.

You can combine these bits using the binary OR operator.

The command **PlayWaveForm("", 0)** can be used at any time to stop sound.

Example:

```
PlayWaveForm("tada.wav", 0)

PlayWaveForm("SystemDefault", 1 | 16)
```

See Also:

PlayMedia, **PlayMidi**

Random

Computes a pseudo-random number.

Syntax:

Random (max**)**

Parameters:

max = largest desired integer number.

Returns:

(integer) unpredictable positive number.

This function will return a random integer between **0** and "max".

Example:

```
a = Random(79)

Message("Random number between 0 and 79", a)
```

Return

Used to return from a **Call** or a **CallExt** to the calling program.

Syntax:

Return

If the program was not called, then an **Exit** is assumed.

Example:

Display(2, "End of subroutine", "Returning to MAIN.WBT")

Return

See Also:

Call, **Exit**

Run

Runs a program as a normal window.

Syntax:

Run (program-name**,** parameters**)**

Parameters:

"program-name" =the name of the desired.**EXE**,.**COM**,.**PIF,**.**BAT** file, or a data file.

"parameters" = optional parameters as required by the application.

Returns:

(integer) **@TRUE** if the program was found;

@FALSE if it wasn't.

Use this command to run an application.

If the drive and path are not part of the program name, the current directory will be examined first, and then the DOS path will be searched to find the desired executable file.

If the "program-name" doesn't have an extension of.EXE,.COM,.PIF, or.BAT, it will be run in accordance with whatever is in the **[extensions]** section of the WIN.INI file. When this happens, any "parameters" you specified are ignored.

Example:

```
Run("notepad.exe", "abc.txt")

Run("clock.exe", "")

Run("paint.exe", "pict.msp")
```

See Also:

RunHide, **RunIcon**, **RunZoom**, **WinClose**, **WinWaitClose**

RunHide

Runs a program as a hidden window.

Syntax:

RunHide (program-name**,** parameters**)**

Parameters:

"program-name" =the name of the desired.**EXE**,.**COM**,.**PIF,**.**BAT** file, or a data file.

"parameters" = optional parameters as required by the application.

Returns:

(integer) **@TRUE** if the program was found;

@FALSE if it wasn't.

Use this command to run an application as a hidden window.

If the drive and path are not part of the program name, the current directory will be examined first, and then the DOS path will be searched to find the desired executable file.

If the "program-name" doesn't have an extension of.EXE,.COM,.PIF, or.BAT, it will be run in accordance with whatever is in the **[extensions]** section of the WIN.INI file. When this happens, any "parameters" you specified are ignored.

Note: When this command launches an application, it informs it that you want it to run as a hidden window. Whether or not the application honors your wish is beyond **RunHide**'s control.

Example:

```
RunHide("notepad.exe", "abc.txt")

RunHide("clock.exe", "")

RunHide("paint.exe", "pict.msp")
```

See Also:

Run, **RunIcon**, **RunZoom**, **WinHide**, **WinClose**, **WinWaitClose**

RunHideWait

Runs a program as a hidden window, and waits for it to close.

Syntax:

RunHideWait (program-name, parameters)

Parameters:

(s) program-name the name of the desired.EXE,.COM,.PIF,.BAT file, or a data file.

(s) parameters optional parameters as required by the application.

Returns:

(i) **@TRUE** if the program was found;
@FALSE if it wasn't.

Use this command to run an application as a hidden window. The WIL program will suspend processing until the application is closed.

If the drive and path are not part of the program name, the current directory will be examined first, followed by the Windows and Windows System directories, and then the DOS path will be searched to find the desired executable file.

If the "program-name" doesn't have an extension of.EXE,.COM,.PIF, or.BAT, it will be run in accordance with whatever is in the **[extensions]** section of the WIN.INI file. When this happens, any "parameters" you specified are ignored.

Note: When this command launches an application, it informs it that you want it to run as a hidden window. Whether or not the application honors your wish is beyond **RunHideWait**'s control.

Example:

```
NetAddCon("winword", "", "g:")
RunHideWait("winword.exe", "")
NetCancelCon("g:", 0)
```

See Also:

RunHide, **RunIconWait**, **RunWait**, **RunZoomWait**, **WinWaitClose**

RunIcon

Runs a program as an iconic (minimized) window.

Syntax:

RunIcon (program-name**,** parameters**)**

Parameters:

"program-name" =the name of the desired.**EXE**,.**COM**,.**PIF,**.**BAT** file, or a data file.

"parameters" = optional parameters as required by the application.

Returns:

(integer) **@TRUE** if the program was found;

@FALSE if it wasn't.

Use this command to run an application as an icon.

If the drive and path are not part of the program name, the current directory will be examined first, and then the DOS path will be searched to find the desired executable file.

If the "program-name" doesn't have an extension of.EXE,.COM,.PIF, or.BAT, it will be run in accordance with whatever is in the **[extensions]** section of the WIN.INI file. When this happens, any "parameters" you specified are ignored.

Note: When this command launches an application, it merely informs it that you want it to begin as an icon. Whether or not the application honors your wish is beyond **RunIcon**'s control.

Example:

```
RunIcon("notepad.exe", "abc.txt")

RunIcon("clock.exe", "")

RunIcon("paint.exe", "pict.msp")
```

See Also:

Run, **RunHide**, **RunZoom**, **WinIconize**, **WinClose**, **WinWaitClose**

RunIconWait

Runs a program as an iconic (minimized) window, and waits for it to close.

Syntax:

RunIconWait (program-name, parameters)

Parameters:

(s) program-name the name of the desired.EXE,.COM,.PIF,.BAT file, or a data file.

(s) parameters optional parameters as required by the application.

Returns:

(i) **@TRUE** if the program was found;
@FALSE if it wasn't.

Use this command to run an application as an icon. The WIL program will suspend processing until the application is closed.

If the drive and path are not part of the program name, the current directory will be examined first, followed by the Windows and Windows System directories, and then the DOS path will be searched to find the desired executable file.

If the "program-name" doesn't have an extension of.EXE,.COM,.PIF, or.BAT, it will be run in accordance with whatever is in the **[extensions]** section of the WIN.INI file. When this happens, any "parameters" you specified are ignored.

Note: When this command launches an application, it merely informs it that you want it to begin as an icon. Whether or not the application honors your wish is beyond **RunIconWait**'s control.

Example:

```
NetAddCon("winword", "", "g:")
RunIconWait("winword.exe", "")
NetCancelCon("g:", 0)
```

See Also:

IconArrange, **RunHideWait**, **RunIcon**, **RunWait**, **RunZoomWait**, **WinWaitClose**

RunWait

Runs a program as a normal window, and waits for it to close.

Syntax:

RunWait (program-name, parameters)

Parameters:

(s) program-name the name of the desired.EXE,.COM,.PIF,.BAT file, or a data file.

(s) parameters optional parameters as required by the application.

Returns:

(i) **@TRUE** if the program was found;
@FALSE if it wasn't.

Use this command to run an application. The WIL program will suspend processing until the application is closed.

If the drive and path are not part of the program name, the current directory will be examined first, followed by the Windows and Windows System directories, and then the DOS path will be searched to find the desired executable file.

If the "program-name" doesn't have an extension of.EXE,.COM,.PIF, or.BAT, it will be run in accordance with whatever is in the **[extensions]** section of the WIN.INI file. When this happens, any "parameters" you specified are ignored.

Example:

```
NetAddCon("winword", "", "g:")
RunWait("winword.exe", "")
NetCancelCon("g:", 0)
```

See Also:

AppWaitClose, Run, **RunHideWait**, **RunIconWait**, **RunZoomWait**, **WinWaitClose**

RunZoom

Runs a program as a full-screen (maximized) window.

Syntax:

RunZoom (program-name**,** parameters**)**

Parameters:

"program-name" =the name of the desired.**EXE**,.**COM**,.**PIF,**.**BAT** file, or a data file.

"parameters" = optional parameters as required by the application.

Returns:

(integer) **@TRUE** if the program was found;

@FALSE if it wasn't.

Use this command to run an application as a full-screen window.

If the drive and path are not part of the program name, the current directory will be examined first, and then the DOS path will be searched to find the desired executable file.

If the "program-name" doesn't have an extension of.EXE,.COM,.PIF, or.BAT, it will be run in accordance with whatever is in the **[extensions]** section of the WIN.INI file. When this happens, any "parameters" you specified are ignored.

Note: When this command launches an application, it merely informs it that you want it to be maximized to full-screen. Whether or not the application honors your wish is beyond **RunZoom**'s control.

Example:

```
RunZoom("notepad.exe", "abc.txt")

RunZoom("clock.exe", "")

RunZoom("paint.exe", "pict.msp")
```

See Also:

Run, **RunHide**, **RunIcon**, **WinZoom**, **WinClose**, **WinWaitClose**

RunZoomWait

Runs a program as a full-screen (maximized) window, and waits for it to close.

Syntax:

RunZoomWait (program-name, parameters)

Parameters:

(s) program-name the name of the desired.EXE,.COM,.PIF,.BAT file, or a data file.

(s) parameters optional parameters as required by the application.

Returns:

(i) **@TRUE** if the program was found;
@FALSE if it wasn't.

Use this command to run an application as a full-screen window. The WIL program will suspend processing until the application is closed.

If the drive and path are not part of the program name, the current directory will be examined first, followed by the Windows and Windows System directories, and then the DOS path will be searched to find the desired executable file.

If the "program-name" doesn't have an extension of.EXE,.COM,.PIF, or.BAT, it will be run in accordance with whatever is in the **[Extensions]** section of the WIN.INI file. When this happens, any "parameters" you specified are ignored.

Note: When this command launches an application, it merely informs it that you want it to be maximized to full-screen. Whether or not the application honors your wish is beyond **RunZoomWait**'s control.

Example:

```
NetAddCon("winword", "", "g:")
RunZoomWait("winword.exe", "")
NetCancelCon("g:", 0)
```

See Also:

RunHideWait, **RunIconWait**, **RunWait**, **RunZoom**, **WinWaitClose**

SendKey

Sends keystrokes to the active application.

Syntax:

SendKey (char-string)

Parameters:

"char-string" = string of regular and/or special characters.

Returns:

(integer) always **0**

This function is used to send keystrokes to the current window, just as if they had been entered from the keyboard. Any alphanumeric character, and most punctuation marks and other symbols which appear on the keyboard, may be sent simply by placing it in the "char-string." In addition, the following special characters, enclosed in "curly" braces, may be placed in "char-string" to send the corresponding special characters:

Key	SendKey equivalent
~	**(~)**
!	**(!)**
^	**(^)**
+	**(+)**
Backspace	**(BACKSPACE)** or **(BS)**
Break	**(BREAK)**
Clear	**(CLEAR)**
Delete	**(DELETE)** or **(DEL)**
Down Arrow	**(DOWN)**
End	**(END)**
Enter	**(ENTER)** or **~**
Escape	**(ESCAPE)** or **(ESC)**
F1 through F16	**(F1)** through **(F16)**
Help	**(HELP)**
Home	**(HOME)**
Insert	**(INSERT)**

Left Arrow	**(LEFT)**
Page Down	**(PGDN)**
Page Up	**(PGUP)**
Right Arrow	**(RIGHT)**
Space	**(SPACE)** or **(SP)**
Tab	**(TAB)**
Up Arrow	**(UP)**

To enter an **Alt**, **Control**, or **Shift** key combination, precede the desired character with one or more of the following symbols:

Alt	**!**
Control	**^**
Shift	**+**

To enter **Alt**-**S**:

```
SendKey("!S")
```

To enter **Ctrl**-**Shift**-**F7**:

```
SendKey("^+(F7)")
```

You may also repeat a key by enclosing it in braces, followed by a space and the total number of repetitions desired.

To type 20 asterisks:

```
SendKey("(* 20)")
```

To move the cursor down 8 lines:

```
SendKey("(DOWN 8)")
```

It is possible to use **SendKey** to send keystrokes to a DOS application, but only if you are running Windows in 386 Enhanced mode. You would then transfer the keystrokes to the DOS application via the Clipboard. It is only possible to send standard ASCII characters to DOS applications; you cannot send function key or Alt-key combinations.

Example:

```
; Start Notepad, and use *.* for filenames

Run("notepad.exe", "")

SendKey("!FO*.*~")
```

```
; run DOS batch file which starts our editor
  Run("edit.bat", "")
; wait 15 seconds for editor to load
  Delay(15)
; send string (with carriage return) to the clipboard
  crlf = StrCat(Num2Char(13), Num2Char(10))This example in plain text:
  ClipPut("Hello%crlf%")
; paste contents of clipboard to DOS window
SendKey("!(SP)EP")
```

See Also:

SKDebug

SKDebug

Controls how **SendKey** works

Syntax:

SKDebug(mode)

Parameters:

mode = **@OFF** Keystrokes sent to application. No debug file written. Default mode.

@ON Keystrokes sent to application. Debug file written.

@PARSEONLY Keystrokes not sent to application. Debug file written.

Returns:

(integer) previous SKDebug mode.

This function allows you to direct the keystrokes generated by your **SendKey** statements to a disk file in addition to, or instead of, the application window. Normally, keystrokes are sent only to the application. If you specify **SKDebug (@ON)**, keystrokes are sent to a disk file as well as to the application. If you specify **SKDebug (@PARSEONLY)**, keystrokes are sent *only* to the disk file, and not to the application. **SKDebug (@OFF)** returns to the default mode.

By default, the file which will receive the parsed keystrokes is named **C:\@@SKDBUG.TXT**. You can override this by making an entry in your **program ini** file, under the heading **[program]**:

If the program you use to call WIL is named "VXPA", you will have a VXPA.INI file. In this file you can insert a VXPA topic that tells WIL where to put a file containing the debugging information. You can specify both the name of this file and its location. If you choose a file name with an extension of .txt, this will probably already be associated by Windows with the Notepad application. If you would like to use another extension for file managing reasons, you could pick ".bug". You would then use the Windows, or any other, file management program to associate this extension with Notepad, or any other plain text editor.

[VXPA]

SKDFile=vxpa.txt

where vxpa.txt is the filename that you want to receive the keystrokes. It is good practice to include the full pathname for this file.

Example:

```
Run("notepad.exe", "")

SKDebug(@ON)

SendKey("!FO*.*~")

SKDebug(@OFF)
```

See Also:

SendKey

SnapShot

Takes a snapshot of the screen and pastes it to the clipboard.

Syntax:

SnapShot (request#)

Parameters:

(i) request# see below.

Returns:

(i) always 0.

Req#	Meaning
0	Take snapshot of entire screen
1	Take snapshot of client area of parent window of active window
2	Take snapshot of entire area of parent window of active window
3	Take snapshot of client area of active window
4	Take snapshot of entire area of active window

Example:

```
SnapShot(2)
```

See Also:

ClipPut

Sounds

Turns sounds on or off.

Syntax:

Sounds (request#)

Parameters:

(i) request# see below.

Returns:

(i) previous Sound setting.

If Windows multimedia sound extensions are present, this function turns sounds made by the WIL Interpreter on or off. Specify a request# of 0 to turn sounds off, and a request# of 1 to turn them on. By default, the WIL Interpreter makes noise.

You can override this by entering:

Sounds=0

in the [Main] section of the WWWBATCH.INI file.

Example:

Sounds(0)

See Also:

Beep, **PlayMedia**, **PlayMidi**, **PlayWaveForm**

StrCat

Concatenates two or more strings.

Syntax:

StrCat (string1**,** string2[**,** stringN]...**)**

Parameters:

"string1", etc = at least two strings you want to "string" together (so to speak).

Returns:

(string) concatenation of the entire list of input strings.

Use this command to stick character strings together, or to format display messages.

Example:

```
user = AskLine("Login", "Your Name:", "")

Message("Login", StrCat("Hi, ", user))

; note that this will do the same:

Message("Login", "Hi, %user%")
```

See Also:

StrFill, **StrFix**, **StrTrim**

StrCmp

Compares two strings.

Syntax:

StrCmp (string1**,** string2**)**

Parameters:

"string1", "string2" = strings to compare.

Returns:

(integer) -1, 0, or 1; depending on whether string1 is less than, equal to, or greater than string2, respectively.

Use this command to determine whether two strings are equal, or which precedes the other in an ANSI sorting sequence.

Note: This command has been included for semantic completeness. The relational operators **>**, **>=**, **==**, **!=**, **<=**, and **<** provide the same capability.

Example:

```
a = AskLine("STRCMP", "Enter a test line", "")

b = AskLine("STRCMP", "Enter another test line", "")

c = StrCmp(a, b)This example in plain text:

c = c + 1

d = StrSub("less than     equal to      greater          than", c * 12,     12)

; Note that above string is grouped into 12-character

; chunks.

; Desired chunk is removed with the StrSub statement.

Message("STRCMP", "%a% is %d% %b%")
```

See Also:

StriCmp, **StrIndex**, **StrLen**, **StrScan**, **StrSub**

StrFill

Creates a string filled with a series of characters.

Syntax:

StrFill (filler**,** length**)**

Parameters:

"filler" = a string to be repeated to create the return string. If the filler string is null, spaces will be used instead.

length = the length of the desired string.

Returns:

(string) character string.

Use this function to create a string consisting of multiple copies of the filler string concatenated together.

Example:

Message("My Stars", StrFill("*", 30))

which produces a dialog titled My Stars that is filled with 30 asterisks. It includes an OK button for canceling the message.

See Also:

StrCat,**StrFix**, **StrLen**, **StrTrim**

StrFix

Pads or truncates a string to a fixed length.

Syntax:

StrFix (base-string**,** pad-string**,** length**)**

Parameters:

"base-string" = string to be adjusted to a fixed length.

"pad-string" = appended to "base-string" if needed to fill out the desired length. If "pad-string" is null, spaces are used instead.

length = length of the desired string.

Returns:

(string) fixed size string.

This function "fixes" the length of a string, either by truncating it on the right, or by appending enough copies of pad-string to achieve the desired length.

Example:

```
a = StrFix("Henry", " ", 15)

b = StrFix("Betty", " ", 15)

c = StrFix("George", " ", 15)

Message("Spaced Names", StrCat(a, b, c))
```

which produces a dialog with names given 15 spaces.

See Also:

StrFill, **StrLen**, **StrTrim**

StriCmp

Compares two strings without regard to case.

Syntax:

StriCmp (string1**,** string2**)**

Parameters:

"string1", "string2" = strings to compare.

Returns:

(integer) -1, 0, or 1; depending on whether string1 is less than, equal to, or greater than string2, respectively.

Use this command to determine whether two strings are equal, or which precedes the other in an ANSI sorting sequence, when case is ignored.

Example:

```
a = AskLine("STRICMP", "Enter a test line", "")

b = AskLine("STRICMP", "Enter another test line", "")This example in plain text:

c = StriCmp(a, b)

c = c + 1

d = StrSub("less than     equal to      greater than", c * 12,      12)

; Note that above string is grouped into 12-character

; chunks.

; Desired chunk is removed with the StrSub statement.

Message("STRICMP", "%a% is %d% %b%")
```

See Also:

StrCmp, **StrIndex**, **StrLen**, **StrScan**, **StrSub**

StrIndex

Searches a string for a substring.

Syntax:

StrIndex (string**,** sub-string**,** start**,** direction**)**

Parameters:

"string" = the string to be searched for a substring.

"substring" = the string to look for within the main string.

start = the position in the main string to begin search. The first character of a string is position **1**.

direction = the search direction. **@FWDSCAN** searches forward, while **@BACKSCAN** searches backwards.

Returns:

(integer) position of "sub-string" within "string";

0 if not found.

This function searches for a substring within a "target" string. Starting at the "start" position, it goes forward or backward depending on the value of the "direction" parameter. It stops when it finds the "substring" within the "target" string, and returns its position.

A start position of **0** has special meaning depending on which direction you are scanning. For **forward** searches, zero indicates the search should start at the *beginning* of the string. For **reverse** searches, zero causes it to start at the *end* of the string.

Example:

```
instr = AskLine("STRINDEX", "Type a sentence:", "")

start = 1

end = StrIndex(instr, " ", start, @FWDSCAN)

If end == 0 Then Goto error

Message("STRINDEX", StrCat("The first word is: ", StrSub(instr, start, end - 1))

Exit

:error

Message("Sorry...", "No spaces found")
```

See Also:

StrLen, **StrScan**, **StrSub**

StrLen

Provides the length of a string.

Syntax:

StrLen (string**)**

Parameters:

"string" = any text string.

Returns:

(integer) length of string.

Use this command to determine the length of a string variable or expression.

Example:

```
myfile = AskLine("Filename", "File to process:", "")

namlen = StrLen(myfile)

If namlen > 13 Then Message("", "Filename too long!")
```

See Also:

StrFill, **StrFix**, **StrIndex**, **StrScan**, **StrTrim**

StrLower

Converts a string to lowercase.

Syntax:

StrLower (string**)**

Parameters:

"string" = any text string.

Returns:

(string) lowercase string.

Use this command to convert a text string to lower case.

Example:

```
a = AskLine("STRLOWER", "Enter text", "")
b = StrLower(a)
Message(a, b)
```

See Also:

StriCmp, **StrUpper**

StrReplace

Replaces all occurances of a substring with another.

Syntax:

StrReplace (string, old, new)

Parameters:

"string" = string in which to search.

"old" = target substring.

"new" = replacement substring.

Returns:

(string) updated "string" with "old" replaced by "new"

StrReplace scans the "string", searching for occurrences of "old" and replacing each occurrence with "new".

Example:

```
; Copy all INI files to clipboard

a = FileItemize("*.ini")

crlf = StrCat(Num2Char(13), Num2Char(10))

b = StrReplace(a, " ", crlf)

ClipPut(b)
```

StrScan

Searches string for occurrence of delimiters.

Syntax:

StrScan (string**,** delimiters**,** start**,** direction**)**

Parameters:

"string" = the string that is to be searched.

"delimiters" = a string of delimiters to search for within "string".

start = the position in the main string to begin search. The first character of a string is position **1**.

direction = the search direction. **@FWDSCAN** searches forward, while **@BACKSCAN** searches backwards.

Returns:

(integer) position of delimiter in string, or **0** if not found.

This function searches for delimiters within a target "string". Starting at the "start" position, it goes forward or backward depending on the value of the "direction" parameter. It stops when it finds any one of the characters in the "delimiters" string within the target "string".

Example:

```
thestr = "123,456.789:abc"

start = 1

end = StrScan(thestr, ",.:", start, @FWDSCAN)

  If end == 0 Then Goto error

  Message("The first parameter", StrSub(thestr, start, end -    start + 1))

  Exit

:error

Message("Sorry...", "No delimiters found")
```

See Also:

StrLen, **StrSub**

StrSub

Extracts a substring out of an existing string.

Syntax:

StrSub (string**,** start**,** length**)**

Parameters:

"string" = the string from which the substring is to be extracted.

start = character position within "string" where the sub-string starts. (The first character of the string is at position **1**).

length = length of desired substring. If you specify a length of zero it will return a null string.

Returns:

(string) substring of parameter string.

This function extracts a substring from within a "target" string. Starting at the "start" position, it copies up to "length" characters into the substring.

Example:

```
a = "My dog has fleas"

animal = StrSub(a, 4, 3)

Message("STRSUB", "My animal is a %animal%")
```

See Also:

StrLen, **StrScan**

StrTrim

Removes leading and trailing spaces from a character string.

Syntax:

StrTrim (string**)**

Parameters:

"string" = a string with unwanted spaces at the beginning and/or the end.

Returns:

(string) string devoid of leading and trailing spaces.

Use this function to remove unwanted spaces from the beginning and end of text data.

Example:

```
myfile = AskLine("STRTRIM", "Filename ('exit' cancels)", "")
tstexit = StrTrim(StrLower(myfile))
If tstexit == "exit" Then Goto cancel
; processing of myfile continues...
: cancel
Message("Canceled", "...by user request")
```

See Also:

StrFill, **StrFix**, **StrLen**

StrUpper

Converts a string to uppercase.

Syntax:

StrUpper (string**)**

Parameters:

"string" = any text string.

Returns:

(string) uppercase string.

Use this function to convert a text string to upper case.

Example:

```
a = AskLine("STRUPPER", "Enter text","")
b = StrUpper(a)
Message(a, b)
```

See Also:

StriCmp, **StrLower**

Terminate

Conditionally ends the procedure.

Syntax:

Terminate (expression, title, message)

Parameters:

"expression" = any logical expression

"title" = the title of a message box to be displayed before termination

"message" = the message in the message box

Returns:

(integer) always @TRUE

This command ends processing for the menu item or procedure if "expression" is not zero. Note that many functions return @TRUE (1) or @FALSE (0), whick you can use to decide whether to cancel a menu item or procedure.

If either "tltle" or "message" contains a string, a messame gox with a title and a message is displayed before exiting.

Example:

```
;Unconditional Termination w/o message

;Same as "Exit"

Terminate (@TRUE,"","")

;Basically a no-op

Terminate(@FALSE,"","This will never terminate")

;Exits with message if a is less than zero

Terminate(a<0,"Error","Cannot use negative num;bers!")

;Exits w/o message if answer isn't "YES"

Terminate(answer!="YES","","")
```

See Also:

Display, **Pause**, **Message**

TextBox

Displays a file in a listbox on the screen and returns selected line, if any.

Syntax:

TextBox (title**,** filename**)**

Parameters:

"title" = listbox title.

"filename" = file containing contents of listbox.

Returns:

(string) = highlighted string, if any.

This function loads a file into a Windows listbox and displays the listbox to the user. **TextBox** has two primary uses: First, it can be used to display multi-line messages to the user. In addition, because of its ability to return a selected line, it may be used as a multiple choice question box. The line highlighted by the user (if any) will be returned to the program.

If disk drive and path not are part of the filename, the current directory will be examined first, and then the DOS path will be searched to find the desired file.

Example:

```
; Display WIN.INI, choose a line from it, and display the line in a dialog.

a = TextBox("Choose a line", "c:\windows\win.ini")

Display(3, "Chosen line", a)
```

See Also:

ItemSelect

TextBoxSort

Displays a file in a sorted listbox on the screen and returns the selected line.

Syntax:

TextBoxSort (title, filename)

Parameters:

(s) title listbox title.

(s) filename file containing contents of listbox.

Returns:

(s)highlighted string, if any.

This function loads a file into a Windows listbox, which is sorted alphabetically and displayed to the user. The line highlighted by the user (if any) will be returned to the program. If the user does not make a selection, a null string (**""**) is returned.

If disk drive and path are not part of the filename, the current directory will be examined first, and then the DOS path will be searched to find the desired file.

TextBox is like **TextBoxSelect**, except that with **TextBoxSelect** the items in the displayed box are sorted and with **TextBox** they are left unsorted.

Example:

```
a = TextBoxSort("Select a phone number", "phones.txt")
Display(3, "Selected number is", a)
```

See Also:

ItemSelect, **TextBox**, **TextSelect**

TextSelect

Allows the user to choose an item from an unsorted listbox.

Syntax:

TextSelect (title, list, delimiter)

Parameters:

(s) title the title of dialog box to display.

(s) list a string containing a list of items to choose from.

(s) delimiter a string containing the character to act as delimiter between items in the list.

Returns:

(s)the selected item.

This function displays a dialog box with a listbox inside. This listbox is filled with an unsorted list of items taken from a string you provide to the function.

Each item in the string must be separated (delimited) by a character, which you also pass to the function.

The user selects one of the items by either doubleclicking on it, or single-clicking and pressing OK. The item is returned as a string.

If you create the list with the **FileItemize** or **DirItemize** functions you will be using a space-delimited list. **WinItemize**, however, creates a tab-delimited list of window titles since titles can have embedded blanks.

TextSelect is like **ItemSelect**, except that with **TextSelect** the displayed box is larger and the items in the box are not sorted alphabetically.

Example:

```
DirChange(DirWindows(0))

inifiles = FileItemize("*.ini")

ini = TextSelect("Select an INI file to edit", inifiles, " ")

If ini == "" Then Exit

RunZoom("notepad.exe", ini)
```

See Also:

AskLine, **DirItemize**, **FileItemize**, **ItemSelect**, **TextBox**, **WinItemize**

Then

Continues a previous **If** statement.

Syntax:

Then statement

Parameters:

(s) statement any valid WIL function or command.

This command continues the last-encountered **If** command. It provides a method of conditionally executing multiple statements, without having to test the condition more than once. If the previous **If** condition was true, the statement following the **Then** keyword is executed. If the previous **If** condition was false, the statement following the **Then** keyword is ignored.

Example:

```
answer = AskYesNo("Financial Management", "Run WinCheck now?")
If answer == @YES Then DirChange("c:\win\check")
Then Run("wincheck.exe", "")
Then WinWaitClose("WinCheck")
Message("Okay", "Processing complete")
```

See Also:

Else, Goto, If... Then

Version

Returns the version number of the program that calls the WIL Language.

Syntax:

Version ( )

Parameters:

(none)

Returns:

(string) = WIL version number.

Use this function to report the version of the program that uses WIL. It is most used in scripts run on different computers, perhaps with different versions of the programs that call the WIL language. The Version function enables scripts to adapt to variabilities between the programs that call the WIL language.

Example:

a = Version()

See Also:

Environment, **DOSVersion**, **WinVersion**

VersionDLL

Returns the version number of the WIL Interpreter currently running.

Syntax:

VersionDLL ( )

Parameters:

(none)

Returns:

(s)WIL Interpreter version number.

This function is use with software applications that use the WIL interpreter. Use this function to determine the version of the WIL Interpreter that is currently running. It is useful to verify that a WIL program generated with the one version of the language will operate properly on a different machine with a different WIL Interpreter installed there.

Example:

```
ver = VersionDLL()
If ver >= "1.0c" Then Goto proceed
Message("Sorry", "WIL Interpreter version 1.0c or higher required")
Exit
:proceed
NetDialog()
```

See Also:

DOSVersion, **Environment**, **Version**, **WinVersion**

WaitForKey

Waits for a specific key to be pressed.

Syntax:

WaitForKey (key1, key2, key3, key4, key5)

Parameters:

(s) key1 - key5 five keystrokes to wait for.

Returns:

(i) position of the selected keystroke (1-5).

WaitForKey requires five parameters, each of which represents a keystroke (refer to the **SendKey** function for a list of special keycodes which can be used). The WIL program will be suspended until one of the specified keys are pressed, at which time the **WaitForKey** function will return a number from 1 to 5, indicating the position of the "key" that was selected, and the program will continue. You can specify a null string ("") for one or more of the "key" parameters if you don't need to use all five.

WaitForKey will detect its keystrokes in most, but not all, Windows applications. Any keystroke that is pressed is also passed on to the underlying application.

Example:

```
k = WaitForKey("(F11)", "(F12)", "(INSERT)", "", "")

If k == 1 Then Message("WaitForKey", "You pressed the F11 key")

If k == 2 Then Message("WaitForKey", "You pressed the F12 key")

If k == 3 Then Message("WaitForKey", "You pressed the Insert key")
```

See Also:

IsKeyDown

WallPaper

Changes the Windows wallpaper.

Syntax:

WallPaper (bmp-name**,** tile**)**

Parameters:

"bmp-name" = Name of the BMP wallpaper file.

tile = **@TRUE** if wallpaper should be tiled.

@FALSE if wallpaper should not be tiled.

Returns:

(integer) always **0**

This function immediately changes the Windows wallpaper. It can even be used for wallpaper "slide shows."

Example:

```
DirChange("c:\windows")

a = FileItemize("*.bmp")

a = ItemSelect("Select New paper", a, " ")

tile = @FALSE

If FileSize(a) < 40000 Then tile = @TRUE

Wallpaper(a, tile)
```

WinActivate

Activates a previously running window.

Syntax:

WinActivate (partial-windowname**)**

Parameters:

"partial-windowname" =

either an initial portion of, or an entire window name. The most-recently used window whose title matches the name will be activated.

Returns:

(integer) **@TRUE** if a window was found to activate;

@FALSE if no windows were found.

Use this function to activate windows for user input.

Example:

```
Run("notepad.exe", "")

Run("clock.exe", "")

WinActivate("Notepad")
```

See Also:

WinCloseNot, **WinGetActive**, **WinShow**

WinArrange

Arranges, tiles, and/or stacks application windows.

Syntax:

WinArrange (style**)**

Parameters:

style = one of the following: **@STACK**, **@TILE** (or **@ARRANGE**), **@ROWS**, or **@COLUMNS**.

Returns:

(integer) always **@TRUE.**

Use this function to rearrange the open windows on the screen. (Any iconized programs are unaffected.)

When you specify **@ROWS** and you have more than four open windows, or if you specify **@COLUMNS** and you have more than three open windows, WIL will revert to **@TILE**.

Example:

```
; Reveal all windows

WinArrange(@TILE)
```

See Also:

WinItemize, **WinHide**, **WinIconize**, **WinPlace**, **WinShow**, **WinZoom**

WinClose

Closes an open window.

Syntax:

WinClose (partial-windowname**)**

Parameters:

"partial-windowname" =

either an initial portion of, or an entire window name. The most-recently used window whose title matches the name will be closed.

Returns:

(integer) **@TRUE** if a window was found to close;

@FALSE if no windows were found.

Use this function to close windows.

WinClose will not close the window which contains the currently-executing WIL file. You can, however, use **EndSession** to end the current Windows session.

Example:

```
Run("notepad.exe", "")

WinClose("Notepad")
```

See Also:

WinCloseNot, **WinHide**, **WinIconize**, **WinWaitClose**

WinCloseNot

Closes all windows, *except* those provided as parameters.

Syntax:

WinCloseNot (partial-windowname [**,** partial-windowname]...**)**

Parameters:

"partial-windowname" =

either an initial portion of, or an entire window name. *Any* windows whose titles match the partial names will stay open.

Returns:

(integer) always **@TRUE**.

Use this function to close all windows *except* those specifically listed in the parameter strings.

At least one partial windowname must be given. A null-string parameter would match all windows, or, in other words, close nothing.

Example:

```
; The statement below will close all windows except:

; 1) Program Manager (starts with 'Program')

; 2) Clock  (starts with 'Clo' )

WinCloseNot("Program", "Clo")
```

See Also:

WinItemize, **WinClose**, **WinHide**, **WinIconize**, **WinWaitClose**

WinConfig

Returns WIN3 mode flags.

Syntax:

WinConfig **( )**

Parameters:

(none)

Returns:

(integer) sum of windows configuration bits.

Returns Windows configuration information as a number, which is the sum of the following individual bits:

1	Protected Mode
2	80286 CPU
4	80386 CPU
8	80486 CPU
16	Standard Mode
32	Enhanced Mode
64	8086 CPU
128	80186 CPU
256	Large PageFrame
512	Small PageFrame
1024	80x87 Installed

You will need to use bitwise operators to extract the individual bits.

Example:

```
cfg = WinConfig()
  If cfg & 32 Then Display(2, "Windows Mode", "Enhanced Mode")
If cfg & 16 Then Display(2, "Windows Mode", "Standard Mode")
If !(cfg & 1) Then Display(2, "Windows Mode", "Real Mode")
cfg = WinConfig()
If cfg & 1024 Then Display(2, "Math co-processor", "Yes")
```

```
If !(cfg & 1024) Then Display(2, "Math co-processor", "No")
```

WinExeName

Returns the name of the executable file which created a specified window.

Syntax:

WinExeName (partial-windowname)

Parameters:

(s) partial-windowname the initial part of, or an entire, window name.

Returns:

(s) name of the E file.

Returns the name of the E file which created the first window found whose title matches "partial-windowname".

"Partial-windowname" is the initial part of a window name, and may be a complete window name. It is case-sensitive. You should specify enough characters so that "partial-windowname" matches only one existing window.

Example:

```
prog = WinExeName("WinCheck")

WinClose("WinCheck")

Delay(5)

Run(prog, "")
```

See Also:

Run, WinExist, WinGetActive, WinName

WinExist

Tells if Window exists.

Syntax:

WinExist (partial-windowname)

Parameters:

"partial-windowname" =

either an initial portion of, or an entire window name.

Returns:

(integer) **@TRUE** if a matching window is found.

@FALSE if a matching window is not found.

Note: The partial window name you give must match the initial portion of the window name (as appears in the title bar) exactly, including proper case (upper or lower) and punctuation.

Example:

If WinExist("Clock") == @FALSE Then RunIcon("Clock", "")

WinGetActive

Gets the title of the active window.

Syntax:

WinGetActive ( )

Returns:

(string) title of active window.

Use this function to determine which window is currently active.

Example:

currentwin = WinGetActive()

See Also:

WinItemize, **WinActivate**

WinHelp

Calls a Windows help file.

Syntax:

WinHelp (help-file, function, keyword)

Parameters:

(s) help-file name of the Windows help file, with an optional full path.

(s) function function to perform (see below).

(s) keyword keyword to look up in the help file (if applicable), or "".

Returns:

(i) **@TRUE** if successful; **@FALSE** if unsuccessful.

This command can be used to perform several functions from a Windows help (.HLP) file.

You can use the Search function in any Windows Help application to find out what keywords point to what page of help. You can then use these keyowords to access help pages from your WIL scripts. Windows includes help applications for general functions. These can be used effectively as a source of glossary items and hints on how to accomplish operations.

It requires that the Windows help program WINHELP.EXE be accessible. The desired function is indicated by the "function" parameter (which is not case-sensitive). The possible choices for "function" are:

"Contents" Brings up the Contents page for the help file.

"Key" Brings up help for the keyword specified by the "keyword" parameter. You must specify a complete keyword, and it must be spelled correctly. If there is more than one occurrence of "keyword" in the help file, a search box will be displayed which allow you to select the desired topic from the available choices.

"PartialKey" Brings up help for the keyword specified by the "keyword" parameter. You may specify a partial keyword name: if it matches more than one keyword in the help file, a search box will be displayed which allow you to select the desired one from the available choices. You may also specify a null string (**""**) for "keyword", in which case you will get a search dialog containing all keywords in the help file.

"Command" Executes the help macro specified by the "keyword" parameter.

"Quit" Closes the WINHELP.EXE window, unless another application is still using it.

"HelpOnHelp" Brings up the help file for the Windows help program (WINHELP.HLP).

For the functions which do not require a keyword (i.e., "Contents", "Quit", and "HelpOnHelp"), specify a null string (**""**) for the "keyword" parameter.

Example:

```
WinHelp("wil.hlp", "Key", "ItemSelect")
```

WinHide

Hides a window.

Syntax:

WinHide (partial-windowname**)**

Parameters:

"partial-windowname" =

Either an initial portion of, or an entire window name can be used. The most-recently used window whose title matches the name will be hidden.

Returns:

(integer) **@TRUE** if a window was found to hide;

@FALSE if no windows were found.

Use this function to hide windows. The programs are still running when they are hidden.

A "partial-windowname" of **""** (null string) hides the owning window.

Example:

```
Run("notepad.exe", "")
WinHide("Notepad")
Delay(3)
WinShow("Notepad")
```

See Also:

WinClose, **WinIconize**, **WinPlace**

WinIconize

Iconizes a window.

Syntax:

WinIconize (partial-windowname**)**

Parameters:

"partial-windowname" =

either an initial portion of, or an entire window name. The most-recently used window whose title matches the name will be iconized.

Returns:

(integer) **@TRUE** if a window was found to iconize;

@FALSE if no windows were found.

Use this function to turn a window into an icon at the bottom of the screen.

A "partial-windowname" of **""** (null string) iconizes the current WIL window.

Example:

```
Run("clock.exe", "")

WinIconize("Clo")    ; partial window name used here
```

See Also:

WinClose, **WinHide**, **WinPlace**, **WinShow**, **WinZoom**

WinItemize

Returns a tab-delimited list of all open windows.

Syntax:

WinItemize ( )

Parameters:

(none)

Returns:

(string) list of the titles of all open windows.

This function compiles a list of all the open application windows' titles and separates the titles by tabs. This is especially useful in conjunction with the **ItemSelect** function, which enables the user to choose an item from such a tab-delimited list.

Note this behaves somewhat differently than **FileItemize** and **DirItemize**, which create space-delimited lists. This is because window titles regularly contain embedded spaces.

Example:

```
; Find a window

allwins = WinItemize()

  htab = Num2Char(9)

  mywind = ItemSelect("Windows", allwins, htab)

  WinActivate(mywind)
```

See Also:

DirItemize, **FileItemize**, **ItemSelect**

WinMetrics

Returns Windows system information.

Syntax:

WinMetrics (request#)

Parameters:

(i) request# see below.

Returns:

(i) see below.

The request# parameter determines what piece of information will be returned.

Req#	Return value
-1	Number of colors supported by video driver
0	Width of screen, in pixels
1	Height of screen, in pixels
2	Width of arrow on vertical scrollbar
3	Height of arrow on horizontal scrollbar
4	Height of window title bar
5	Width of window border lines
6	Height of window border lines
7	Width of dialog box frame
8	Height of dialog box frame
9	Height of thumb box on scrollbar
10	Width of thumb box on scrollbar
11	Width of an icon
12	Height of an icon
13	Width of a cursor
14	Height of a cursor
15	Height of a one line menu bar

16	Width of full screen window
17	Height of a full screen window
18	Height of Kanji window (Japanese)
19	Is a mouse present (0 = No, 1 = Yes)
20	Height of arrow on vertical scrollbar
21	Width of arrow on horizontal scrollbar
22	Is debug version of Windows running (0 = No, 1 = Yes)
23	Are Left and Right mouse buttons swapped (0 = No, 1 = Yes)
24	Reserved
25	Reserved
26	Reserved
27	Reserved
28	Minimum width of a window
29	Minimum height of a window
30	Width of bitmaps in title bar
31	Height of bitmaps in title bar
32	Width of sizeable window frame
33	Height of sizeable window frame
34	Minimum tracking width of a window
35	Minimum tracking height of a window

Example:

```
mouse = "NO"
If WinMetrics(19) == 1 Then mouse = "YES"
  Message("Is there a mouse installed?", mouse)
```

See Also:

MouseInfo, **NetGetCaps**, **WinConfig**, **WinParmGet**, **WinResources**

WinName

Returns the name of the current WIL Interpreter window.

Syntax:

WinName ( )

Parameters:

(none)

Returns:

(s)window name.

Returns the name of the current WIL interpreter (eg, the program you use to run WIL scripts) window.

Example:

```
tab = Num2Char(9)
allwins = WinItemize()
win = ItemSelect("Close window", allwins, tab)
If win == WinName() Then Goto nocando
WinClose(win)
Exit
:nocando
Message("Sorry", "I can't close myself")
```

See Also:

WinActivate, **WinExeName**, **WinGetActive**, **WinItemize**, **WinTitle**

WinParmGet

Returns system information.

Syntax:

WinParmGet (request#)

Parameters:

(i) request# see below.

Returns:

(s)see below.

The request# parameter determines what piece of information will be returned.

Req#	Meaning	Return value
1	Beeping	0 = Off, 1 = On
2	Mouse sensitivity	"threshold1 threshold2 speed"
3	Border Width	Width in pixels
4	Keyboard Speed	Keyboard Repeat rate
5	LangDriver	name of LANGUAGE.DLL
6	Horiz. Icon Spacing	Spacing in pixels
7*	Screen Save Timeout	Timeout in seconds
8*	Is screen saver enabled	0 = No, 1 = Yes
9	Desktop Grid size	Grid Size
10	Wallpaper BMP file	BMP file name
11	Desktop Pattern	Pattern codes (string of 8 space-delimited nums.)
12*	Keyboard Delay	Delay in milliseconds
13	Vertical Icon Spacing	Spacing in pixels
14	IconTitleWrap	0 = No, 1 = Yes
15*	MenuDropAlign	0 = Right, 1 = Left
16	DoubleClickWidth	Allowable horiz. movement in pixels for DblClick
17	DoubleClickHeight	Allowable vert. movement in pixels for DblClick

18	DoubleClickSpeed	Max time in millisecs between clicks for DblClick
19	MouseButtonSwap	0 = Not swapped, 1 = swapped
20*		Fast Task Switch 0 = Off, 1 = On

Items marked with an asterisk (*) require Windows 3.1 or higher.

Example:

```
If WinParmGet(8) == 1 Then Message("", "Screen saver is active")
```

See Also:

MouseInfo, NetGetCaps, WinConfig, WinMetrics, WinParmSet, WinResources

WinParmSet

Sets system information.

Syntax:

WinParmSet (request#, new-value, ini-control)

Parameters:

(i) request# see **WinParmGet**

(s) new-value see **WinParmGet**

(i) ini-control see below.

Returns:

(int) previous value of the setting.

See **WinParmSet** for a list of valid request#'s and values.

The "ini-control" parameter determines to what extent the value gets updated:

0 Set system value in memory only for future reference

1 Write new value to appropriate INI file

2 Broadcast message to all applications informing them of new value

3 Both 1 and 2

Example:

```
WinParmSet(9, "2", 3)          ; sets desktop grid size to 2
```

See Also:

WallPaper, **WinParmGet**

WinPlace

Places a window anywhere on the screen.

Syntax:

WinPlace (x-ulc**,** y-ulc**,** x-brc**,** y-brc**,** partial-windowname**)**

Parameters:

x-ulc = how far from the left of the screen to place the upper-left corner (**0**-**1000**).

y-ulc = how far from the top of the screen to place the upper-left corner (**0**-**1000**).

x-brc = how far from the left of the screen to place the bottom-right corner (**10**-**1000**) or **@NORESIZE**.

y-brc = how far from the top of the screen to place the bottom-right corner (**10**-**1000**) or **@NORESIZE** or **@ABOVEICONS**.

"partial-windowname" =

either an initial portion of, or an entire window name. The most-recently used window whose title matches the name will be moved to the new position.

Returns:

(integer) **@TRUE** if a window was found to move;

@FALSE if no windows were found.

Use this function to move windows on the screen. (You cannot, however, move icons or windows that have been maximized to full screen.)

The "x-ulc", "y-ulc", "x-brc", and "y-brc" parameters are based on a logical screen that is 1000 points wide by 1000 points high.

You can move the window without changing the width and/or height by specifying **@NORESIZE** for the "x-brc" and/or "y-brc" parameters, respectively.

You can fix the bottom of the window to sit just above the line of icons along the bottom of the screen by specifying a "y-brc" of **@ABOVEICONS**.

Some sample parameters:

Upper left quarter of the screen: **0, 0, 500, 500**

Upper right quarter: **500, 0, 1000, 500**

Center quarter: **250, 250, 750, 750**

Lower left eighth: **0, 750, 500, 1000**

A handy utility program is included with WIL, called **WININFO.EXE**. This program lets you take an open window that is sized and positioned the way you like it, and automatically create the proper **WinPlace**

statement for you. It puts the text into the Clipboard, from which you can paste it into your batch code:

You'll need a mouse to use **WinInfo**. While **WinInfo** is the active window, place the mouse over the window you wish to create the **WinPlace** statement for, and press the spacebar. The new statement will be placed into the Clipboard. Then press the Esc key to close **WinInfo**.

Example:

```
WinPlace(0, 0, 200, 200, "Clock")
```

See Also:

WinArrange, **WinHide**, **WinIconize**, **WinShow**, **WinZoom**

WinPlaceGet

Returns window coordinates.

Syntax:

WinPlaceGet (win-type, partial-windowname)

Parameters:

(i) win-type	@ICON, @NORMAL, or @ZOOMED
(s) partial-windowname	the initial part of, or an entire, window name.

Returns:

(s)	window coordinates (see below).

This function returns the coordinates for an iconized, normal, or zoomed window.

"Partial-windowname" is the initial part of a window name, and may be a complete window name. It is case-sensitive. You should specify enough characters so that "partial-windowname" matches only one existing window. If it matches more than one window, the most recently accessed window which it matches will be used.

The returned value is a string of either 2 or 4 numbers, as follows:

Iconic windows	"x y"	(upper left corner of the icon)
Normal windows	"upper-x upper-y lower-x lower-y"	
Zoomed windows	"x y"	(upper left corner of the window)

All coordinates are relative to a virtual 1000x1000 screen.

Example:

```
Run("clock.exe", "")
pos = WinPlaceGet(@NORMAL, "Clock")
Delay(2)
WinPlaceSet(@NORMAL, "Clock", "250 250 750 750")
Delay(2)
WinPlaceSet(@NORMAL, "Clock", pos)
```

See Also:

WinGetActive, **WinItemize**, **WinPlaceSet**, **WinPosition**, **WinState**

WinPlaceSet

Sets window coordinates.

Syntax:

WinPlaceSet (win-type, partial-windowname, position-string)

Parameters:

(i) win-type	@ICON, @NORMAL, or @ZOOMED
(s) partial-windowname	the initial part of, or an entire, window name.
(s) position-string	window coordinates (see below).

Returns:

(s)previous coordinates.

This function sets the coordinates for an iconized, normal, or zoomed window. The window does not have to be in the desired state to set the coordinates; for example, you can set the iconized position for a normal window so that when the window is subsequently iconized, it will go to the coordinates that you've set.

"Partial-windowname" is the initial part of a window name, and may be a complete window name. It is case-sensitive. You should specify enough characters so that "partial-windowname" matches only one existing window. If it matches more than one window, the most recently accessed window which it matches will be used.

"Position-string" is a string of either 2 or 4 numbers, as follows:

Iconic windows	"x y"	(upper left corner of the icon)
Normal windows	"upper-x upper-y lower-x lower-y"	
Zoomed windows	"x y"	(upper left corner of the window)

All coordinates are relative to a virtual 1000x1000 screen.

Example:

```
WinPlaceSet(@ICON, "Clock", "10 950")

WinPlaceSet(@NORMAL, "Clock", "250 250 750 750")

WinPlaceSet(@ZOOMED, "Clock", "-5 -5")
```

See Also:

IconArrange, **WinActivate**, **WinArrange**, **WinPlace**, **WinPlaceGet**, **WinState**

WinPosition

Returns Window position.

Syntax:

WinPosition (partial-windowname)

Parameters:

"partial-windowname" =

either an initial portion of, or an entire window name.

Returns:

(string) window coordinates, delimited by commas.

Returns the current Window position information for the selected Window. It returns 4 comma-separated numbers (see **WinPlace** for details).

Example:

```
Run("clock.exe", "")          ; start Clock
WinPlace(0,0,300,300, "Clock")     ; place Clock
pos = WinPosition("Clock")          ; save position
delay(2)
WinPlace(200,200,300,300, "Clock")     ; move Clock
delay(2)
WinPlace(%pos%, "Clock")          ; restore Clock
```

See Also:

WinPlace

WinResources

Returns information on available memory and resources.

Syntax:

WinResources (request#)

Parameters:

(i) request# see below

Returns:

(i) see below.

The value of request# determined the piece of information returned.

Req#	Return value
0	Total available memory, in bytes
1	Theoretical maximum available memory, in bytes
2	Percent of free system resources (lower of GDI and USER)
3	Percent of free GDI resources
4	Percent of free USER resources

Example:

```
mem = WinResources(0)
Message("Available memory", "%mem% bytes")
```

See Also:

WinConfig, **WinMetrics**, **WinParmGet**

WinShow

Shows a window in its "normal" state.

Syntax:

WinShow (partial-windowname**)**

Parameters:

"partial-windowname" =

either an initial portion of, or an entire window name. The most-recently used window whose title matches the name will be shown.

Returns:

(integer) **@TRUE** if a window was found to show;

@FALSE if no windows were found.

Use this function to restore a window to its "normal" size and position.

A "partial-windowname" of **""** (null string) restores the current WIL interpreter window.

Example:

```
RunZoom("notepad.exe", "")
; other processing...
WinShow("Notepad")
```

See Also:

WinArrange, **WinHide**, **WinIconize**, **WinZoom**

WinState

Returns the current state of a window.

Syntax:

WinState (partial-windowname)

Parameters:

(s) partial-windowname the initial part of, or an entire, window name.

Returns:

(i) window state (see below).

"Partial-windowname" is the initial part of a window name, and may be a complete window name. It is case-sensitive. You should specify enough characters so that "partial-windowname" matches only one existing window. If it matches more than one window, the most recently accessed window which it matches will be used.

Possible return values are as follows.

Value	Symbolic name	Meaning
-1		Specified window exists, but is hidden
0		Specified window does not exist
1	@ICON	Specified window is iconic (minimized)
2	@NORMAL	Specified window is a normal window
3	@ZOOMED	Specified window is zoomed (maximized)

Example:

```
If WinState("Notepad") == @ICON Then WinShow("Notepad")
```

See Also:

Run, WinExist, WinGetActive, WinHide, WinIconize, WinItemize, WinPlace, WinPlaceGet, WinPlaceSet, WinPosition, WinShow, WinZoom

WinTitle

Changes the title of a window.

Syntax:

WinTitle (partial-windowname**,** new-name**)**

Parameters:

"partial-windowname" =

either an initial portion of, or an entire window name. The most-recently used window whose title matches the name will be shown.

"new-name" = the new name of the window.

Returns:

(integer) **@TRUE** if a window was found to rename;

@FALSE if no windows were found.

Use this function to change a window's title.

A "partial-windowname" of **""** (null string) refers to the current WIL interpreter window.

Warning: Some applications may rely upon their window's title staying the same! Therefore, the **WinTitle** function should be used with caution and adequate testing.

Example:

```
; Capitalize title of window

htab = Num2Char(9)

allwinds = WinItemize()

mywin = ItemSelect("Uppercase Windows", allwinds, htab)

WinTitle(mywin, StrUpper(mywin))

Drop(htab, allwinds, mywin)
```

See Also:

WinItemize

WinVersion

Provides the version number of the current Windows system.

Syntax:

WinVersion (level**)**

Parameters:

level = either **@MAJOR** or **@MINOR.**

Returns:

(integer) either major or minor part of the Windows version number.

Use this command to determine which version of Windows is currently running.

@MAJOR returns the integer part of the Windows version number; i.e. **1**.0, **2**.11, **3**.0, etc.

@MINOR returns the decimal part of the Windows version number; i.e. 1.**0**, 2.**11**, 3.**0**, etc.

Example:

```
minorver = WinVersion(@MINOR)
majorver = WinVersion(@MAJOR)
Message("Windows Version", StrCat(majorver, ".", minorver))
```

See Also:

Version, **DOSVersion**

WinWaitClose

Suspends the batch file execution until a specified window has been closed.

Syntax:

WinWaitClose (partial-windowname**)**

Parameters:

"partial-windowname" =

either an initial portion of, or an entire window name. **WinWaitClose** suspends execution until all matching windows have been closed.

Returns:

(integer) **@TRUE** if at least one window was found to wait for;

@FALSE if no windows were found.

Use this function to suspend the batch file's execution until the user has finished using a given window and has manually closed it.

Example:

```
Run("clock.exe", "")

Display(4, "Note", "Close Clock to continue")

WinWaitClose("Clock")

Message("Continuing...", "Clock closed")
```

See Also:

Delay, **Yield**

WinZoom

Maximizes a window to full-screen.

Syntax:

WinZoom (partial-windowname**)**

Parameters:

"partial-windowname" =

either an initial portion of, or an entire window name. The most-recently used window whose title matches the name will be shown.

Returns:

(integer) **@TRUE** if a window was found to zoom;

@FALSE if no windows were found.

Use this function to "zoom" windows to full screen size.

A partial-windowname of **""** (null string) zooms the current WIL interpreter window.

Example:

```
Run("notepad.exe", "")
WinZoom("Notepad")
Delay(3)
WinShow("Notepad")
```

See Also:

WinHide, WinIconize, WinPlace, WinShow

Yield

Provides time for other windows to do processing.

Syntax:

Yield

Use this command to give other running windows time to process. This command will allow each open window to process 20 or more messages.

Example:

```
; run Excel and give it some time to start up

  sheet = AskLine("Excel", "File to run:", "")

  Run("excel.exe", sheet)

Yield

Yield

Yield
```

See Also:

Delay, **Exclusive**