

Windows Interface

languageyesHTMLWILyeshtmlwilyesyesyesyesyes06/11/98

Table of Contents

[Main](#)

[Registering your software](#)

[Ordering Information](#)

[WILSON WINDOWWARE ORDER FORM](#)

[How to get technical support](#)

[Menu File Structure](#)

[Modifying Menus](#)

[Menu Hotkeys](#)

[Menu Items](#)

[What is WIL?](#)

[Debug WIL Scripts](#)

[Topic by Topic](#)

[The Complete Tutorial](#)

[Creating WIL Scripts Files](#)

[Running WIL Utilities](#)

[Running WIL System Utilities](#)

[What is a WIL Program?](#)

[Our First WIL Program](#)

[Functions and Parameters](#)

[Displaying Text](#)

[Getting Input](#)

[Using Variables](#)

[Making Decisions](#)

[Control of Program Flow](#)

[Exploring WIL](#)

[Running Programs](#)

[Display and Input](#)

[Manipulating Windows](#)

[Files and Directories](#)

[Handling Errors](#)

[Selection Methods](#)

[Wrap Lines in Messages](#)

[Running DOS Programs](#)

[Sending Keystrokes to Programs](#)

[Our Completed WIL File](#)

[The Tutorial Course](#)

[Tutor Example](#)

[Tutor Example 2](#)

[Tutor Example Script Stop](#)

[Tutor Example continued](#)

[Tutor Example continued...](#)

[Tutor Example continued 5](#)

[Tutor Example continued...](#)

[Tutor Example continued...](#)

[Tutor Example continued...](#)

[Tutor Example continued...](#)

[Tutor Example continued...](#)

[Tutor Example continued...](#)

[Tutor Example continued...](#)

[Tutor Example continued...](#)

[Tutor Example continued...](#)

[Tutor.wbt](#)

Reference

[Notational Conventions](#)

[Notes](#)

[Recovering from Cancel](#)

[Script Template](#)

[Extenders: Networking, Novell, TCP/IP, Microsoft HTMLWIL. ENN001](#)

[Terminating WIL processing](#)

[Carriage Return Line Feed](#)

[Extension Associations and the Run functions](#)

[Debug](#)

[Internal Control Functions](#)

[Partial Window Names](#)

[System.ini and its device= lines](#)

[Dividing Floating Point Numbers](#)

[File Delimiters](#)

[Sounds](#)

Reference Functions

[Standard WIL functions in alphabetical order](#)

Alphabetic Jumps for alphabetical listing.

[A](#)

[B](#)

[C](#)

[D](#)

[E](#)

[F](#)

[G](#)

[I](#)

[K](#)

[M](#)

[N](#)

[O](#)

[P](#)

[R](#)

[S](#)

[T](#)

[V](#)

[W](#)

[Y](#)

[Standard WIL functions by type](#)

[Arithmetic Functions](#)

[Binary Functions](#)

[Clipboard Handling](#)

[DDE Functions](#)

[Directory Management](#)

[Disk Drive Management](#)

[Displaying Information](#)

[DLLCall](#)
[File Management](#)
[Important Functions](#)
[Inputting Information](#)
[InterProgram Communication](#)
[Menu Management](#)
[Miscellaneous Functions](#)
[Multimedia Functions](#)
[Network Functions](#)
[OLE2.0](#)
[Process Control](#)
[Program Management](#)
[Registration Functions](#)
[String Handling](#)
[System Information](#)
[Time Functions](#)
[Window Management](#)

Reference

[WIL Language Elements](#)

[Constants](#)
[Identifiers](#)
[Variables](#)
[Substitution](#)
[Floating Point Constants](#)
[Integer Constants](#)
[String Constants](#)
[Predefined Constants](#)
[Predefined System Constants](#)
[Predefined Floating Point Constants](#)
[Predefined String Constants](#)
[Lists](#)
[Keywords](#)
[Operators](#)
[Precedence and Evaluation Order](#)
[Comments](#)
[Statements](#)
[Parameters](#)
[Error Handling](#)

Reference

[Error Messages](#)
[Minor Errors](#)
[Moderate Errors](#)
[Fatal Errors](#)
[Copyright ©1996 Wilson WindowWare, Inc.](#)

Windows Interface Language Help File



Getting started

[What is WIL?](#)

[What WIL is good for](#)

[Products that use WIL](#)

[Contacting Wilson WindowWare](#)

[Step by step guide to learning WIL](#)



The Windows Interface Language (WIL) is a scripting language supported by [many Wilson WindowWare products](#). There are [over 500 functions available](#) to help you get the job done -- what would take pages of code in other languages is often taken care of by a single WIL function.

Using WIL

[Batch files](#)

[Menu Files](#)

For more information [about WIL](#) and [what it can do](#), see the [getting started](#) section of this help file.

WIL Reference

Function list

[By type](#)

[Alphabetically](#)



We've included a [Step by step guide to learning WIL](#) so you can get started right away. And there's a complete [reference](#) to WIL in this help file.

[WIL Language Elements](#)

[Error Messages](#)

Wilson WindowWare products are available for every version of Windows, including Windows 3.1, Windows 95, Windows 98 and Windows NT, running on Intel, DEC Alpha, PowerPC and other platforms.

[Notes](#)

[Technical support](#) is available for registered users. If you



can't find what you're

looking for, or you're having problems with your WIL scripts, be sure to look at the [notes](#) section, which has solutions to many of the problems you're likely to encounter.

[Copyright ©1996 Wilson WindowWare, Inc.](#)

Copyright © 1988-1997 by Morrie Wilson.

All rights reserved.

No part of this manual may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying and recording, for any purpose without the express written permission of Wilson WindowWare, Inc. Information in this document is subject to change without notice and does not represent a commitment by Wilson WindowWare, Inc.

The software described herein is furnished under a license agreement. It is against the law to copy this software under any circumstances except as provided by the license agreement.

U.S. Government Restricted Rights

Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subdivision (b)(3)(ii) of the Rights in Technical Data and Computer Software clause at 252.227-7013. Contractor/manufacturer is Wilson WindowWare, Inc. / 5421 California Ave SW / Seattle, WA 98136 / Orders: 800-762-8383 / Support: 206-937-9335 / Fax: 206-935-7129.

Trademarks

Microsoft and MS-DOS are registered trademarks of Microsoft Corporation. Windows, Word for Windows, NotePad, Visual Basic and Excel are trademarks of Microsoft Corporation. WordPerfect, AmiPro are trademarks of IBM. Power Builder is a trademark of PowerSoft Corporation.

Clock Manager is a trademark of Graphical Dynamics.

WinBatch, WinEdit and WebBatch are trademarks of Wilson WindowWare, Inc.

Acknowledgments

This software designed by Morrie Wilson and Richard Merit.

Documentation written by Morrie Wilson, Richard Merit, Jim Stiles, Tina Browning and Deana Dahley .

Getting started

[What is WIL?](#)

[What WIL is good for](#)

[Products that use WIL](#)

[Contacting Wilson WindowWare](#)

[Step by step guide to learning
WIL](#)

Windows Interface Language (WIL) is an easy-to-use yet very powerful general-purpose programming language with [over 500 functions](#) for file management, sending keystrokes, disk drive management, directory management, binary file access, multimedia support, DDE support, clipboard handling, system control, program management, string handling, displaying information, user prompting, window management, floating point & integer arithmetic, execution control and more. Many operations that require pages of code in other programming languages can be accomplished with a single WIL function call.

WIL is supported by several Wilson WindowWare products, including WinBatch, WinBatch+Compiler, WebBatch, FileMenu, PopMenu and WinEdit. Some implementations of WIL include capabilities not described here (for example, WebBatch includes functions that are useful only in an Internet World Wide Web server environment, and leaves out those that don't make sense in that environment such as window management).

This help file describes the "standard" set of WIL functions; for details on a particular implementation of WIL please see that program's help file.

What is WIL?

Windows Interface Language (WIL) is an easy-to-use yet very powerful general-purpose programming language with over [500 functions](#) for file management, sending keystrokes, disk drive management, directory management, binary file access, multimedia support, DDE support, clipboard handling, system control, program management, string handling, displaying information, user prompting, window management, floating point & integer arithmetic, execution control and more. Many operations that require pages of code in other programming languages can be accomplished with a single WIL function call.

WIL scripts are written in a plain text file, which can be created by **WinBatch Studio**, Notepad or most word processors. (Of course, we recommend our own WinEdit, which has many features designed expressly for programmers, including a full-featured implementation of WIL itself.)

These text files can take one of two forms, depending on your particular implementation of WIL: [batch files](#) or [menu files](#).

Batch Files

A batch file is simply a list of WIL commands and function calls, executed in order (just like the old DOS batch language).

Menu Files

A menu file is similar to a batch file, except that multiple chunks of WIL code are organized into menu and sub-menus, and each routine is launched by pressing the appropriate keystroke or selecting an item from the menu. (The name and location of the menus vary depending on the particular implementation of WIL menu files.)

- [What WIL is good for](#)
- [Products that use WIL](#)
- [Using WIL](#)
- [Reference](#)
- [Step by step guide to learning WIL](#)

What WIL is good for

Using WIL, you can make small batch files to use as system management utilities. Connecting to network servers, printing batch jobs out at odd hours, upgrading software, and metering application use, are just a few of the chores handled by the system utilities made with WIL.

You can also write major business applications. Experience with small utilities encourages the leap to major projects. WIL is the common glue that can bind any off-the-shelf or custom Windows and DOS programs together. Software from any vendor can be combined to make a solution. Automated business solutions save time, save money, and make money for the companies using them to leverage their investment in hardware, software and people.

With the WebBatch implementation of WIL, your Windows NT-based web server can process orders, dynamically respond to visitor choices, manage group discussion lists, and public tech support pages; nearly anything you've seen done on the Internet using Perl or C can often be done much easier and faster using WIL.

WIL gives you some very powerful features:

- Traditional batch-language file management capabilities such as copying files, changing the directory, or launching applications
 - Structured programming, including GOSUB, SELECT CASE, FOR...NEXT, WHILE...ENDWHILE, IF...ELSE...ENDIF
 - Complete support for user interaction, including standard Windows dialog boxes, check boxes, radio buttons, and file list selectors
 - Time and date functions for scheduling operations
 - Send keystrokes and menu commands directly to applications
 - Network support: extenders included for Novell 3.x, Novell 4.x, Windows for Workgroups, Basic Windows 3.1. Win32 network extenders for Windows 95/98/NT.
 - Link to Windows DLLs, third party MAPI and TAPI DLLs, or your own C or C++ DLLs
 - Extensive string manipulation, integer and floating point arithmetic, including trig and log functions
 - System control -- read and change system values like the current wallpaper, keyboard repeat rate, etc.
 - Manipulate windows (change their size or location)
 - Control multimedia hardware (sound cards, CD-ROM drives, etc.)
 - Manipulate the DOS environment (e.g., PATH)
 - Launch a script via clicking an icon in the Explorer.
 - Complete binary access to any file
 - Your implementation of WIL may include all of these features and more; see your program's help file for complete details.

- [What is WIL?](#)
- [Products that use WIL](#)
- [Contacting Wilson WindowWare](#)
- [Using WIL](#)
- [Reference](#)
- [Step by step guide to learning WIL](#)

Some features aren't appropriate for a particular implementation; for example, WebBatch doesn't include any window-management functions, but it has many functions that are only appropriate in a web server environment.

Products that use WIL

Windows Interface Language (WIL) is an easy-to-use yet very powerful general-purpose programming language with over [500 functions](#) to automate nearly any task. Many operations that require pages of code in other programming languages can be accomplished with a single WIL function call.

WIL is supported by several Wilson WindowWare products, including WinBatch, WinBatch+Compiler, WebBatch, and WinEdit.

Some implementations of WIL include capabilities not described here. For example, WebBatch includes functions that are useful only in an Internet World Wide Web server environment, and leaves out those that don't make sense in that environment such as window management.

- [What is WIL?](#)
- [What WIL is good for](#)
- [Contacting Wilson WindowWare](#)
- [Step by step guide to learning WIL](#)

Batch file implementations of WIL

WinBatch
WinBatch+Compiler
WebBatch

Menu file implementations of WIL

WinEdit
Clock Manager
PopMenu (a utility included with WinBatch)
FileMenu (a utility included with WinBatch)

Contacting Wilson WindowWare

Wilson WindowWare, Inc.
5421 California Ave. SW
Seattle, WA 98136 USA

Orders: (800) 762-8383

Voice: (206) 938-1740

Fax: (206) 935-7129

Email: info@windowware.com

- [Registering your copy](#)
- [Ordering Information](#)
- [Order form](#)
- [Technical support](#)

.....

Registered users of our software receive: manuals, technical support, use of Wilson WindowWare on-line information services, and special offers on new versions of Wilson WindowWare products.

How to get technical support

The Wilson WindowWare website is an excellent technical resource. Access to the entire Technical Support Database is at your fingertips. In the Technical Support area use the keyword search to find answers to common problems, alternate scripting methods, and sample code. Or join the Wilson WindowWare Web BBS, a new Web forum. The BBS provides an outlet for registered users to share their experiences with other users.

See the information on [registering your copy](#) if you haven't done so yet.

The latest versions of our software are available on-line. The places here may change at any time -- check your installation sheet for the most recent addresses.

Internet Web page: <http://www.windowware.com>

Internet Technical Support Articles & Web BBS:
<http://techsupt.windowware.com>

Internet FTP: <ftp.windowware.com> in [/wwwftp/wilson](#)

- [Registering your copy](#)
- [Ordering Information](#)
- [Order form](#)

- [Reference](#)

- [Step by step guide to learning WIL](#)

Registering your software

Registered users of our software receive: manuals, [technical support](#), use of Wilson WindowWare on-line information services, and special offers on new versions of WinBatch and other Wilson WindowWare products.

You can register online, through our secure commerce server:

<http://commerce.windowware.com>

You can register your software by mailing your registration card, faxing your registration card, or calling Wilson WindowWare.

Wilson WindowWare, Inc.
5421 California Ave. SW
Seattle, WA 98136 USA

Orders: (800) 762-8383

Voice: (206) 938-1740

Fax: (206) 935-7129

- [Ordering Information](#)
- [Order form](#)
- [Contacting Wilson WindowWare](#)

Ordering Information

Licensing our products brings you wonderful benefits. Some of these are:

- Gets rid of that pesky reminder window that comes up when you start up the software.
- Entitles you to one hour free phone support for 90 days (Your dime).
- Ensures that you have the latest version of the product.
- Encourages the authors of these programs to continue bringing you updated/better versions and new products.
- Gets you on our mailing list so you are occasionally notified of spectacular updates and our other Windows products.
- And, of course, our 90-day money back guarantee.

▪ [Order form](#)

▪ [Contacting Wilson WindowWare](#)

International customers

Although we do prefer payment by Credit Card we can accept non-US-bank checks under certain conditions. The check MUST be in your currency -- NOT IN US\$ -- Just look in your newspaper for the current exchange rates, make out your check and send mail it to us. We will take care of the rest. No Eurocheques please.

Send to: Wilson WindowWare, Inc.
5421 California Ave. SW
Seattle, WA 98136
USA

or call: (800) 762-8383 (USA orders only)
(206) 938-1740 (customer service)
(206) 937-9335 (tech support)
(206) 935-7129 (fax)

(Please allow up to 2-3 weeks for delivery)

WILSON WINDOWWARE ORDER FORM

WILSON WINDOWWARE
5421 California Ave. SW
Seattle • WA • 98136
Ph - (206)938-1740
Fax - (206)935-7129

- Ordering Information
- Contacting Wilson WindowWare

Name: _____

Company: _____

Address: _____

City: _____ St: _____ Zip: _____

Phone: (____) _____

Country: _____

Products

___ WinBatch @ \$99.95 : _____

___ WinBatch Compiler @\$495.00 : _____

___ WinEdit @\$99.95 : _____

Shipping

___ US and Canada shipping @ \$5.00 : _____

___ Foreign air shipping
(except Canada) @ \$14.50 : _____

Total: _____

Please enclose a check payable to Wilson WindowWare or you may use Access, Amex, Visa, MasterCharge, or EuroCard. For credit cards, please enter the information below:

Card #: _____ - _____ - _____ - _____ **Expiration date:** ____/____

Signature: _____

Where did you hear about or get a copy of our products?

International customers please see note on previous page.

Step by step guide to learning WIL

The Windows Interface Language (WIL) is a scripting language. In order to use it, you must open up an ascii editor (such as **WinBatch Studio**) and [Create a Script](#) using the WIL commands. Once written, the script is saved and [run](#) with an extension already associated with the WIL interpreter. In our examples, we use the extension .WBT.

The WIL language is not hard to learn. A general knowledge of batch file programming is helpful, but not necessary.

Suggestions for Tutorial use

Everyone has different learning styles. The contents of the WIL Tutorial can be accessed in several ways.

[Topic by Topic](#) - Arranged so each new concept builds on the last. Scroll through the topics from the top or select the ones which catch your eye.

[Step by Step Tutorial Course](#) - For those who have the general idea and don't want to be bogged down with the absolute particulars. Follow along and write a working script.

[The Complete Tutorial](#) - For some, the printed word is mightier than the hypertext jump. Here the tutorial has been arranged for easy printing.

The WIL Tutorial

- [The Complete WIL Tutorial](#)
- [Topic by Topic](#)
- [The Tutorial Course](#)

- [Getting started](#)
- [Using WIL](#)
- [Reference](#)
- [Notational Conventions](#)
- [Notes](#)

Topic by Topic

The WIL Tutorial is an excellent place to start when learning how to use the WIL language to create batch scripts. The topics are arranged so that each new concept builds on the last. Scroll through them one by one or feel free to skip around.

[Creating WIL Scripts Files](#)

The WIL Tutorial

- [The Complete WIL Tutorial](#)
- [Topic by Topic](#)
- [The Tutorial Course](#)

[Running WIL Utilities](#)
[Running WIL System Utilities](#)
[What is a WIL Program?](#)
[Our First WIL Program](#)
[Functions and Parameters](#)
[Displaying Text](#)
[Getting Input](#)
[Using Variables](#)
[Making Decisions](#)
[Control of Program Flow](#)
[Exploring WIL](#)
[Running Programs](#)
[Display and Input](#)
[Manipulating Windows](#)
[Files and Directories](#)
[Handling Errors](#)
[Debug WIL Scripts](#)
[Selection Methods](#)
[Wrap Lines in Messages](#)
[Running DOS Programs](#)
[Sending Keystrokes to Programs](#)
[Our Completed WIL File](#)

- [Getting started](#)
- [Using WIL](#)
- [Reference](#)
- [Notes](#)

The Tutorial Course

When learning a new skill, it is often more effective to dive in and give it a try, rather than bury yourself in documentation. If you're an insomniac, reading the WIL Reference Manual in bed is a good alternative for a doctor's prescription. However, until your WIL scripts are managing your work load you probably don't have time to sleep.

The Tutorial Course will guide you through the absolute basics by taking you through the creation of a sample script. In some places the information may be further enhanced by reading the whole topic. Feel free to scroll to the top of any topic to get the full picture.



Each Tutorial Step is indicated by a numbered flag so you know where to start reading and so you don't stray off course. When selected they provide all kinds of additional information.

The actual hands on directions for each step are outlined in the center column.

To jump easily from step to step look for;



Next...

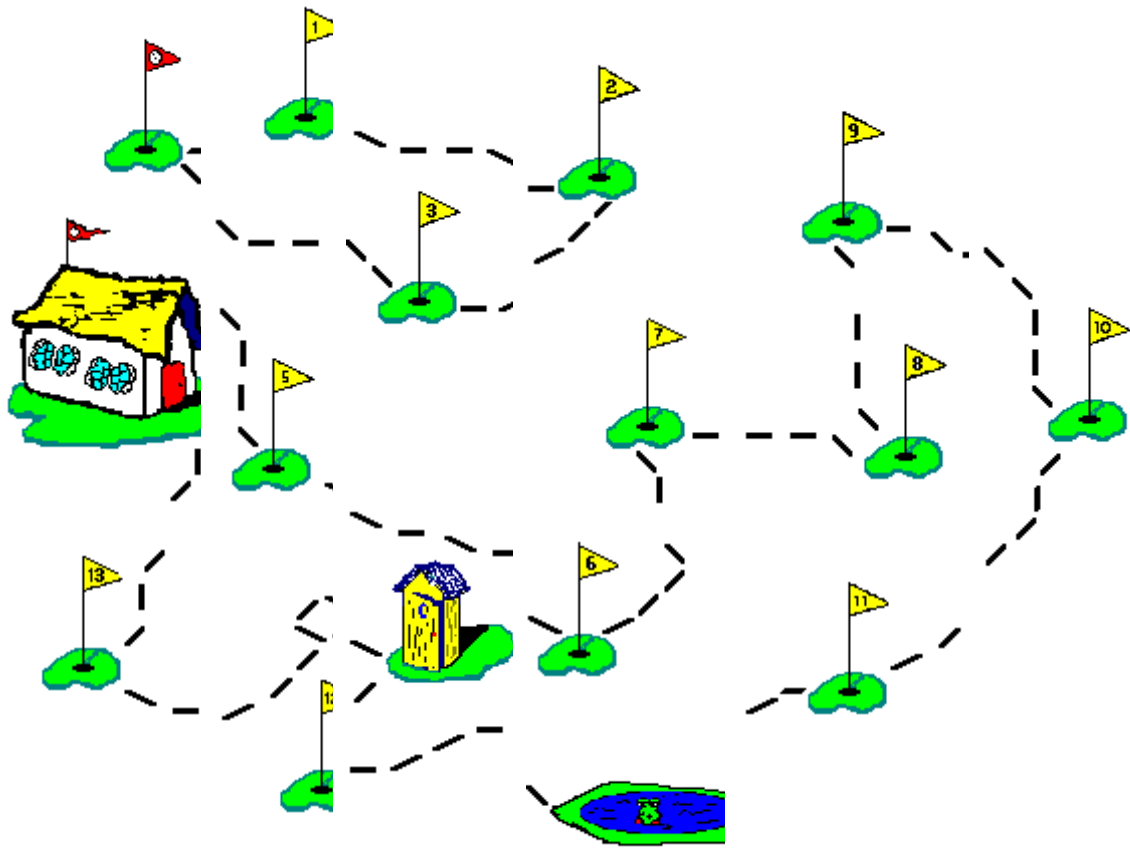
The Tutorial code examples are indicated with a different font.

`;retype example lines into your editor.`

The WIL Tutorial

- [The Complete WIL Tutorial](#)
- [Topic by Topic](#)
- [The Tutorial Course](#)

- [Getting started](#)
- [Using WIL](#)
- [Reference](#)
- [Notes](#)



The Complete WIL Tutorial

The WIL Tutorial

- [Topic by Topic](#)
- [The Tutorial Course](#)
- [Getting started](#)
- [Using WIL](#)
- [Reference](#)
- [Notes](#)

Late one evening around the flickering of the screensaver amidst greasy pizza boxes and crumpled diet soda cans, a haggard programmer relayed his vision for the future to his equally haggard teammates. They were astounded. Was it the late hour, the heaviness of the pepperoni turning somersaults in their gut or this ingenious theory which caused the sudden delirium in their minds? The next day the vision went public but is yet to be realized.

The myth of the infamous "paperless" office may never be seen in our lifetimes. If you'd like to give it a try, pop over to [Topic by Topic](#) and click your way through the tutorial.

If you'd prefer to learn WIL with the tutorial in hand rather than on screen, you can select print button at the top and print the entire tutorial in one swell foop.

Approximate size; 31 pgs.

Creating WIL Scripts Files

WIL is a script file interpreter. Before you can do anything useful with the WIL interpreter, you must have at least one WIL script file to interpret.

Your program installation puts several sample scripts into a "samples" directory. Sample scripts are located in a "samples" subdirectory in the WinBatch directory in Windows 95, Windows 98 and NT 4.0.

WIL script files must be formatted as plain text files. You can create them with **WinBatch Studio**, WinEdit (Wilson WindowWare's optional text editor for programmers), the Windows Notepad or another text editor.

Word processors like WordPerfect, AmiPro, and Word can also save scripts in plain text formatted files.

The .WBT (WinBatch) extension is used in this manual for batch file extensions, but, you can use others just as well. If you want to click on a batch file and have Windows run it, be sure that you associate it in Windows with your executable program file. When you installed your program, an association was automatically established between the interpreter and .WBT files.

Each line in a script file contains a statement written in WIL, Wilson WindowWare's Windows Interface Language.

A statement can be a maximum of 255 characters long. Indentation does not matter. A statement can contain functions, commands, and comments.

A single WinBatch script cannot exceed 64 kilobytes in file size.

You can give each script file a name which has an extension of WBT (e.g. TEST.WBT). We'll use the terms WinBatch script files and WBT files interchangeably.

Running WIL Utilities

WIL system utilities are very versatile. They can be run from icons in the Windows Explorer.

- from macros in word processors and spreadsheets.
- from a command line entry such as the "Run..." from the Explorer Start button.
- by double clicking.
- from menu items via the Filemenu and Popmenu utilities.
- from other WIL scripts to serve as single or multiple "agents", event handlers, or schedulers.
- from any Windows application or application macro language that can execute another Windows program.

Software suite macro languages and application builders like Visual Basic and PowerBuilder are examples of these.

Running WIL System Utilities

WIL utilities run like any other Windows programs. They can run from a command line, a shortcut, or from a file listing such as the Windows Explorer.

WIL utilities are usually run as files with the extension .WBT. When some WIL utilities are used, they need information passed to them when they run. This is easily done by passing command line parameters to them.

This capability can be used from the command line in the Run menu .

Parameters can be also be passed through the command line entry included in the item properties of any icon. Finally, an application can send parameters to a WIL utility it launches from a command line or from a function in a macro language.

A command like this runs a WinBatch system utility from a command line or an icon:

```
WINBATCHEXEC filename.wbt p1 p2... p9
```

This command line can be entered into a Command Line text entry box like this one:



The command line is longer than the dialog can show, but it can be easily edited with the arrow keys.

WINBATCHEXEC is the generic name of your WinBatch executable. The specific, or actual, name for the WinBatch application will change to reflect the operating system in use: Windows 3.1, Windows 95, Windows 98 and the different Windows NT versions.

"filename.wbt" is any valid WBT file, and is a required parameter.

"p1 p2 ... p9" are optional parameters (there are a maximum of nine of these) to be passed to the WBT file on startup. Each is delimited from the next by one space character.

Parameters passed to a WBT file will be automatically inserted into variables named **param1**, **param2**, etc. The WinBatch utility will be able to use these. An additional variable, **param0**, gives you the total number of command-line parameters.

Example:

Passing parameters on the command line to a WinBatch script.

Create a WinBatch script called "myscript.wbt" and put it in a temp directory. The contents of "myscript.wbt" should be the following:

```
Message (param1,param2)  
Exit
```

In Windows 95, 98 and NT 4.0, go to the Start Menu and Select Run. Enter the following commandline:

```
"c:\Program Files\WinBatch\System\WinBatch.exe" "c:\temp\myscript.wbt" "Example"
"Hello World"
```

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

Tutor Example

1. Open an editor, such as WinBatch Studio.

2. Type in the line.

*3. Save the program with an appropriate extension. If you're using WinBatch, this extension will be .WBT. We will refer to this script as **tutor.wbt**.*

*4. Run **tutor.wbt** by double-clicking on the filename.*

Our first WIL program will simply run our favorite Windows application: Solitaire. If you are using a menu script-based implementation of the WIL Interpreter, refer to the section on Menu Files for instructions on how to create and edit WIL menu items. 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 the **WinBatch Studio**, that is capable of saving text in pure ASCII format. In either case, 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, the User's Guide, for information on how to execute a WIL program). Presto! It's Solitaire.

Note: in order to run this script solitaire must be installed, make sure you installed GAMES with your windows platform .

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** ([Introduction](#)). 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 arguments 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", "")
```

Note: Strings are delimited by quotes, ("", "")

"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: (s) denotes the parameter requires a string.

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 WIL language standpoint, the second parameter is required. That is, if you omit it, you will get a syntax error, and your WIL program will halt. However, the program that you are running may not need parameters. Solitaire, for example, does not take 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.

Example 2

To illustrate this further, let's create a WIL program containing

1. Open WinBatch Studio.
2. Type in the line.
3. Save the file as notetest.wbt.
4. Close WinBatch Studio
5. Run the wbt by double-clicking on the filename.

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.

Is Notepad running? If it is - good. If not, check your script. WinBatch will look for Notepad.exe on the computer's search path. If it isn't there, you may need to locate it with the File Manager or the Windows Explorer. Add a complete pathname to the run statement and the script should work.

Now, edit the WIL program as follows:

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

Save the program, exit WinBatch Studio, 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")
```

Note: semi-colons denote comments.

The semi-colon 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 semi-colon. For example:

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

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

Displaying Text

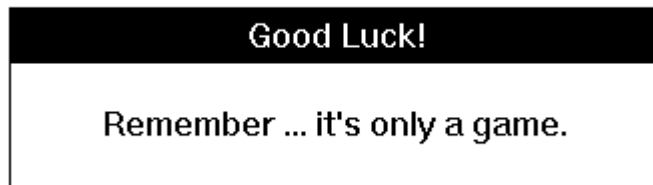
Tutor Example continued...

1. Exit Solitaire.
2. Activate or open your editor.
3. Load tutor.wbt.
4. Add the new lines.
5. Save the file.
6. Run the wbt by double-clicking on the filename.

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

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

And run it. Notice the little dialog 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. |

The **Display** function has three parameters. The first parameter 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). In our example above, we are displaying the box for 5 seconds. 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 **(s)** in front of a parameter indicates that it is a string. An **(i)** indicates that it is an integer and a **(f)** indicates a floating point number parameter.

Tutor Example continued...

1. Exit Solitaire.
2. Edit **tutor.wbt**
-Comment out the **Run** statement.
3. Save the file.
4. Run the wbt by double-clicking on the filename.

Now, exit Solitaire (if you haven't done so already), and edit the WIL program by placing a semi-colon 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:

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

Feel free to 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 2.9, and see what happens (it will be converted to an integer). 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, it is a floating point number). 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. 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.)

Tutor Example continued...

1. Edit **tutor.wbt**
-Delete the

Display
statement.
Add **AskYesNo**.
Delete semi-
colon from the

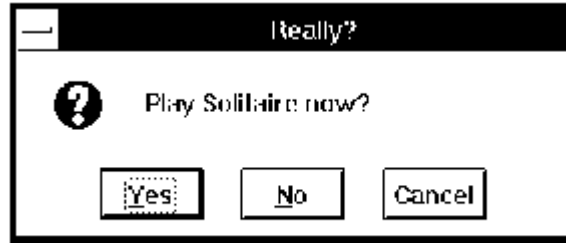
Now, let's modify our WIL program as follows:

```
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:

Run.

2. Save the file.
3. Run the wbt by double-clicking on the filename..



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.)

Tutor Example continued...

1. Exit Solitaire.
2. Edit **tutor.wbt**
-Add variable name..
3. Save the file.
4. Run the wbt by double-clicking on the filename..

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 ... Endif** command. Actually, there are several forms of the **If** statement -- the structured form and the single statement form.

Structured Forms

If expression
 series of statements

Endif

If expression
 series of statements

Else

 series of statements

Endif

Single Statement Forms

If expression **Then** statement.

If expression **Then** statement

Else statement

(We refer to **If ... Endif** as a **command**, rather than a **function**, because functions are followed by parameters in parentheses, while commands are not. Commands tend to be used to control the WIL interpreter.)

Tutor Example continued...

1. *Exit Solitaire.*

2. *Edit **tutor.wbt**
-Add **If/Endif**
command to
check return of
the variable.*

3. *Save the file.*

4. *Run the **wbt** by
double-clicking
on the filename.*

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

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

However, as this example is a single statement, rather than a series of statements, the single statement structure is more appropriate. There are generally many different ways to perform any task in WIL. With experience you will be able quickly decide the best way to do any task.

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

Experiment with additional If structures.

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 or works out to a non-zero value, then the statement(s) following are performed. If the condition following the **If** keyword is false or works out to a zero value, then the statement(s) following are ignored.

There is something extremely important that you should note about the syntax of these **If ... Endif** commands: 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

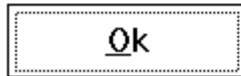
want to test whether it is equal to @YES. Therefore, we use the double equal signs (==), which is the **equality operator** in WIL. The statement above is saying, in English: "If the value of the variable named **response** is equal to @YES, then run the program SOL.EXE." If you used a single equal sign (=) here by mistake, you would get an error message:

3069: Assignment not legal here. (Use == for equality)



If response = @YES Then Run("sol

WinBatch 32 96M
WIL Version: 2.3mbn



Note:

(=) *this IS that*
(==) *this*
EQUALS *that*

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 statement(s) following the **If** condition are true. But what happens when the condition is false? Remember we said that when the **If** condition is false, the following statement(s) are 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  
    Run("sol.exe", "")  
Else  
    Display(5, "", "Game canceled")  
Endif
```

Using the single statement **If...Then...Else** structure the same code would look like:

```
response=AskYesNo("Really?", "Play Solitaire  
now?")  
If response == @YES      Then Run("sol.exe",  
    "")  
    Else Display(5, "", "Game canceled")
```

When you have only single statements to execute when conditions are true or false, the single statement form may be preferred. However, 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", "")
If response==@NO Then Display(5,"","Game
canceled")
```

Clearly, the best way of handling this is to use the **If... Else... Endif** structured form.

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

Control of Program Flow

The linear flow of statements (executing one statement after another) is not always preferred or possible. **WIL** provides the standard set of flow control **commands**: **For**, **While**, **Switch** and **GoSub**. These commands give you the ability to redirect the **flow of control** in your **WIL** programs.

The **For** command controls the looping of a block of code based on an incrementing index. The **While** command conditionally and/or repeatedly executes a series of statements. The **Switch** statement allows selection among multiple blocks of statements. **GoSub** transfers control to another point in the **WIL** program and saves the location for a **Return** statement.

Let's explore the use of these commands further. Perhaps you need to break your Solitaire habit by limiting your time of play (it has, by now, become obvious to your boss and co-workers that, ever since you got this program, all you do is play solitaire).

Tutor Example continued...

First you need to ask yourself how long you would like to play by adding the following line to the top of your script.

```
mins=AskLine("Solitaire","How many mins do you want to
play?","")
```


1. Exit Solitaire.
(If it is running.)
2. Edit **tutor.wbt**
-Add **AskLine** to
the top of the
script.
3. Save the file.
4. Run the wbt by
double-clicking
on the filename.

This will display a message box which prompts you for the number of minutes you would like to play.

Once you enter the desired number of minutes, you could display an additional message as a response to the specific amount of time entered. **Switch**, as you remember, allows selection from among multiple blocks of statements. Each block of statements is called a **case**. In the sample below, there are several case statement blocks. Selection of one of the cases is determined by the number of minutes stored in the variable **mins**. If the number is 3, then case 3 will be executed. All numbers not accounted for will be executed by the default case, **mins**.

1. Exit Solitaire.
2. Edit **tutor.wbt**
to add the
Switch
statement.

Type the following code into **tutor.wbt**. Test the Switch statement by entering a number into the **AskLine** dialog box. Try running it several times using various numbers.

Your code should look like the code below. Remove extra statements.

```
mins = AskLine("Solitaire", "How many mins do you want to play?", "")
mins = Int(mins)
Switch mins
  case 0
    Display(5, "", "Game canceled")
    exit
    break
  case 1
    Message("Only a minute?", "Wow! You've got willpower.")
    break
  case 2
    Message("2 Minutes?", "This isn't much of a break.")
    break
  case 3
    Message("3 Minutes?", "You've barely got time to shuffle.")
    break
  case 4
    Message("HA, HA, HA", "I dare you to try to beat me.")
    break
  case mins ;default case - must be last in the switch
    Message("THAT LONG!!!", "Where did you get all that time?")
    break
EndSwitch
Run("sol.exe", "")
```

In our example, each case statement block is composed of three parts; a **case** statement followed by a number, a series of one or more statements and the **break** command. If the number behind the **case** statement matches the number behind the **switch** statement, then the **case** block is executed. Once the correct message has been displayed, **break** terminates the case block and transfers control to the **EndSwitch** statement.

Now we need to create a timer to track the time elapsed and compare it to the time entered. The **While** command, which repeats execution of a series of statements by telling WIL, "Do the following while a condition is present," does this job nicely.

Tutor Example continued...

1. Exit Solitaire.
2. Edit **tutor.wbt**
-
Set up variables.
Add the timing code below the **Run** statement.
3. Save the file.
4. Run the wbt by double-clicking on the filename.

First let's set up a couple of variables.

```
goal = mins * 60  
timer = 0
```

Now for the **While** statement. The first line sets the condition, "While the **timer** is less than the **goal** execute this series of statements."

```
While timer < goal  
    remain = goal - timer  
    WinTitle("Solitaire", "Solitaire (%remain%  
seconds  
    left)")  
    TimeDelay(10)  
    timer = timer + 10  
EndWhile
```

The rest of our series of statements include: a computation of the time remaining (**remain**) to be displayed, a line to display the time remaining in the Solitaire window title bar, a **delay** statement to allow time to pass, and a statement to calculate the time elapsed. **EndWhile** marks the end of statements. **WIL** marches through the **While** loop until the variable **timer** exceeds the value of the variable **goal**.

So what happens if suddenly your time is up and you're four moves away from winning? Can't have that happening.

Tutor Example continued...

1. Exit Solitaire.
2. Edit **tutor.wbt**
-
Add new line to bottom of script.
3. Save the file.
4. Run the wbt by double-clicking on the filename.

We can give ourselves the opportunity to add more time by adding another **AskLine** statement.

```
mins=AskLine("More Time?", "Enter additional  
minutes.", 0)
```

If a time is entered the timer will need to be used again. Of course, it would be easy to copy that portion of the script and insert it after the new line. However, the same script can be utilized with the assistance of **GoSub**.

GoSub causes the flow of control to go to another point in the **WIL** program while remembering its point of origin. The name **GoSub** is an abbreviation of "Go To Subroutine". You must specify where you want the flow of control to be transferred -- the subroutine name, and you must mark this point with a **label**. A label is simply a destination address, or marker. The form of the **GoSub** command is:

GoSub 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.

Tutor Example continued...

1. Exit Solitaire.
2. Edit **tutor.wbt**-
Add **:dumdedum** label to bottom.
Move timing loop below label.

In our sample script, we move the timing loop to the bottom of the script, add a label marked **:dumdedum** above the timing script as the destination address. After **EndWhile**, add the statement, **Return** to allow the flow of control to return from the bottom of the **GoSub**.

We'll add a **GoSub** statement in after the **Run** statement. The **GoSub** statement is saying, in English "go to the line marked **:dumdedum**, and continue processing from there, but

Add **Return** after loop. After **Run** statement add **GoSub dumdedum**.

3. Save the file.
4. Run the wbt by double-clicking on the filename.

Tutor Example continued...

1. Exit Solitaire.
2. Edit **tutor.wbt**

-
Add check for return of **AskLine**.

Add **WinClose** and **Exit**. (See code below)

3. Save the file.
4. Run the wbt by double-clicking on the filename.

remember where you came from." When **Return** is reached, control will be transferred back to the statement after the original **GoSub**.

Notice that the label **dumdedum** is preceded by a colon as the address, but *not* on the line where it follows the **GoSub** keyword. This is important. Although you can have multiple lines in your WIL program which say **GoSub dumdedum**, you can have only one line marked **:dumdedum** (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.

In addition to changing the message displayed in the "mins=AskLine" statement, a default time has been added. The value returned will need to be checked. In the example below, "!=" signifies "not equal to". Therefore the line reads, "If mins is not equal to zero then GoSub dumdedum."

```
If mins!=0 then GoSub dumdedum
```

If a time is returned, **GoSub** will send execution to the **:dumdedum** label and the waiting process will begin again. After the time has elapsed, control will be returned to the statement following the **GoSub**.

The last thing we want to do is end the program with the **WinClose** function and display a final message.

The **Exit** command is used to keep the processing from "falling through" to the subroutine at the end of the program. In this case, the dumdedum subroutine sits at the end. **Exit** causes a WIL program to end immediately and not fall into the dumdedum loop.

Our altered script has the following appearance from the **Run** statement to the bottom.

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

GoSub dumdedum

mins=AskLine("More Time?", "Enter additional minutes", 0)
If mins!=0 then GoSub dumdedum

WinClose("Solitaire")
Message("Time's Up", "Get Back to Work!")

Exit

:dumdedum
goal = mins * 60
timer = 0
While timer < goal
    remain = goal - timer
    WinTitle("Solitaire", "Solitaire (%remain% seconds left)")
    TimeDelay(10)
    timer = timer + 10
EndWhile
Return
```

The sample script could be considered complete at this point. However, the **For** command has yet to be discussed. The **For** command is more complex than the previous commands. It controls the looping of a block of code based on an incrementing index. This command is handy if you want to perform a specific code block a particular number of times. The statement says, "Repeat the block of code for each value of a variable from the initial value to the final value, incrementing the variable after each pass of the loop".

In the sample below, the size of the Solitaire window is manipulated and displayed 10 times before the window is zoomed to full screen. Each time the loop is executed, the coordinate and size variables (j and k) are altered, and then used in a **WinPlace** statement (it's time to start looking up [functions](#) in the reference yourself now) to affect the position and size of the Solitaire window.

Tutor Example continued...

1. Edit tutor.wbt

-
Add code for the **For** statement directly after the **Run** statement.

```
Run("sol.exe", "")  
  
for i = 0 to 9  
    j=100-i*10  
    k=300+i*70  
    WinPlace(j,j,k,k, "Solitaire")  
next
```

2. Save the file.

```
WinZoom("Solitaire")
```

3. Run the wbt by double-clicking on the filename.

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.

See the [Completed WIL file, tutor.wbt](#), at the end of the WIL TUTORIAL.

Exploring WIL

If you take a moment and flip through the [WIL Function Reference](#) that takes up most of the back of the manual, you will notice that WIL uses a very convenient naming convention. [WIL functions](#) are named so that the object affected by the function is the first word in the function name -- any function dealing with Files starts with the word "File", and they can be found clumped together in the alphabetically arranged function reference. If you think you might want a function dealing with DDE, simply flip open the manual to the DDE section and scan the available functions.

What follows is just quick overview of the many functions and [commands](#) available in WIL. These should be sufficient to begin creating versatile and powerful WIL programs. For complete information on these and all WIL functions and commands, refer to the **WIL Function Reference** (see [Introduction](#)).

Running Programs

There are several [functions](#) that you can use to start an [application](#), most of which share a common syntax.

32 bit Windows 95/98/NT can use file names with embedded spaces. For handling this possibility, use quotation marks like this:

```
Run(' "My Program" ', "")
```

Notice: the use of single quotation marks outside of the double quotation marks.

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 system 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("excel", "bigsheet.xls")
```

RunIcon (program-name, parameters)

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

Example:

```
RunIcon("notepad", "")
```

All these Run functions simply launch the program and continue with WIL processing. If you need to wait until the program exits before continuing, then there are a number of other suitable functions also available.

RunWait (program-name, parameters)

RunWait starts a program and waits for it to exit before continuing. This is often required when the "program" is a DOS batch file.

RunZoomWait (program-name, parameters)

RunZoomWait starts a program as a full screen window and waits for it to exit before continuing.

RunIconWait (program-name, parameters)

RunIconWait starts a program as an icon at the bottom of the screen and waits for it to exit before continuing.

If all these Run functions are too much for you, there is also the combination **RunShell** function, which combines all the capabilities of the Run functions and adds additional functionality.

RunShell (program-name, parameters, working dir, view, waitflag)

RunShell is an advanced form of the Run function that even allows the specification of a working directory, along with the window view mode and whether or not to wait for completion of the run program in a single function.

Display and Input

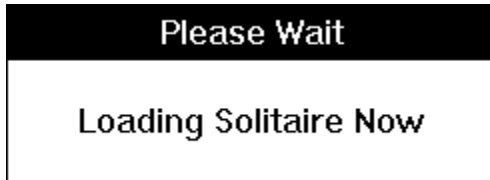
Here we have functions which display information to the user and prompt the user for information, plus a couple of relevant system functions.

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, "Please wait", "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")
```

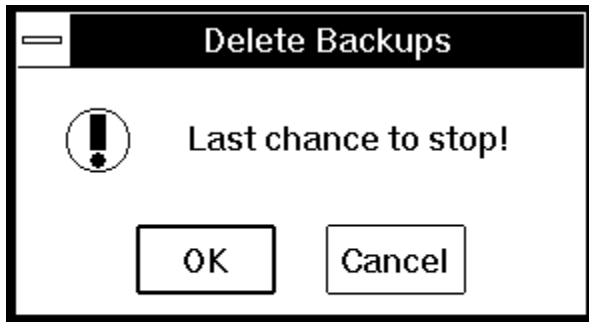


Pause (title, text)

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")
```

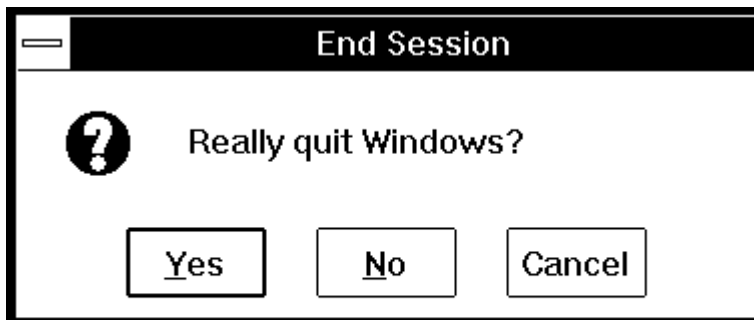


AskYesNo (title, question)

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?")
```

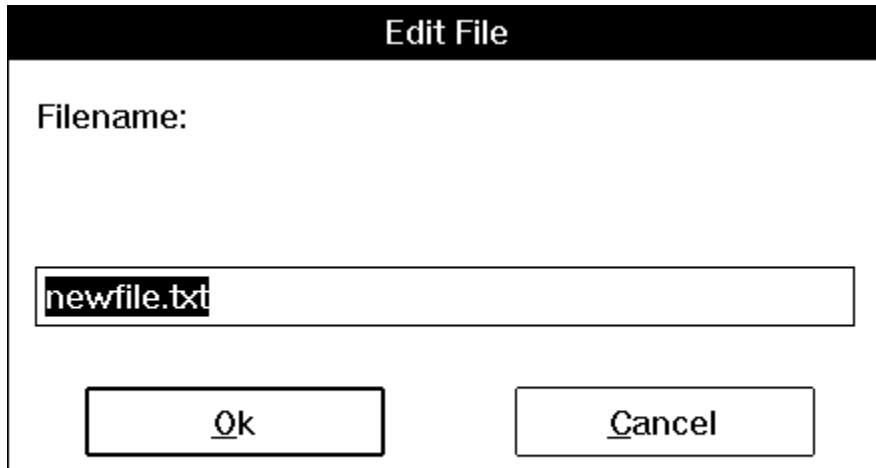


AskLine (title, prompt, default)

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

TimeDelay (seconds)

Pauses WIL program execution.

The **TimeDelay** 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)

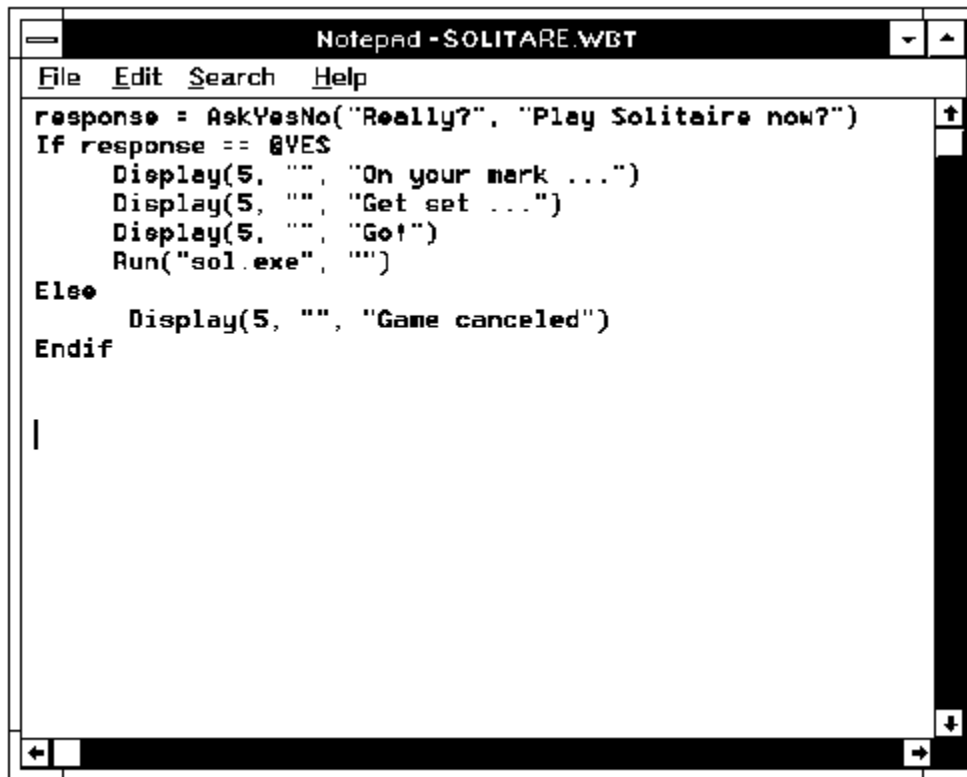
Shows a window in its "normal" state.

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**.



```
Notepad - SOLITARE.WBT
File Edit Search Help
response = AskYesNo("Really?", "Play Solitaire now?")
If response == @YES
    Display(5, "", "On your mark ...")
    Display(5, "", "Get set ...")
    Display(5, "", "Go!")
    Run("sol.exe", "")
Else
    Display(5, "", "Game canceled")
Endif
|
```

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 **C**lose 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.

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 as follows:

```
If WinExist("Solitaire") == @TRUE  
WinActivate("Solitaire")  
Else  
Run("sol.exe", "")  
Endif
```

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

```
If WinExist("Solitaire") == @FALSE  
Run("sol.exe", "")  
Else  
WinActivate("Solitaire")  
Endif
```

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
FileCopy("win.ini", "win.bak", @FALSE)
endif
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, along with selecting various options for copying the files. 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 (source-list, destination)

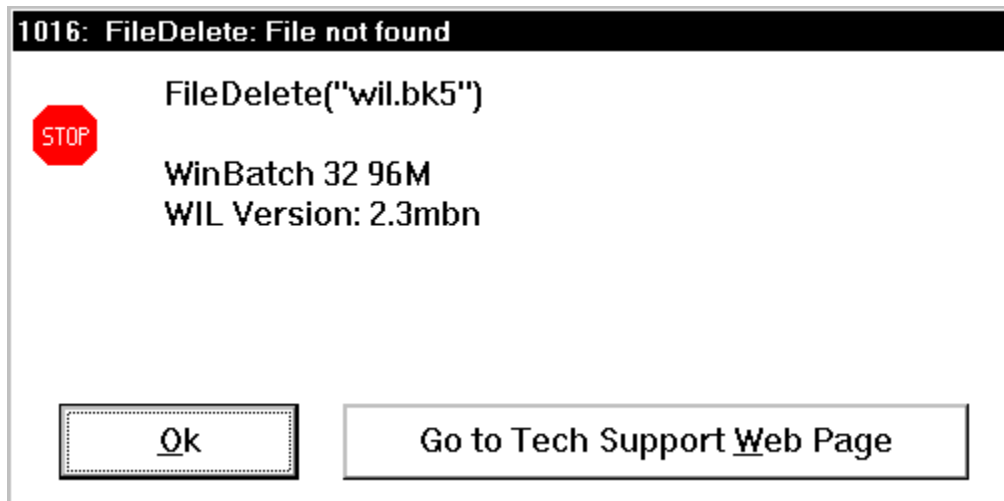
Renames files to another set of names.

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
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")
Endif
Run("winword.exe", "wil.doc")
Exit
```

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.

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 resulting in a message resembling the following:



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. Luckily, there is a WIL system function to help us here: **ErrorMode**.

Handling Errors

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
  ErrorMode(@OFF)                                 ;turn 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)                             ;reset ErrorMode to default
Endif
Run("winword.exe", "wil.doc")
Exit
```

Notice how we've used **ErrorMode(@OFF)** to prevent errors in the **If** statement section 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.

Note: Pay close attention when suppressing errors with the **ErrorMode** function. When an error occurs, the processing of the ENTIRE line is canceled. Setting the **ErrorMode()** to **@OFF** or **@NOTIFY** allows execution to resume at the next line. **Various parts of the original line may have not been executed.**

e.g.

```

ErrorMode(@OFF)
; The FileCopy will cause a file not found error,
; canceling the execution of the whole line.
; The variable A is set to @FALSE by default
A = FileCopy( "xxxxxxxx", "*.*", @FALSE)
;
;
; Now there is a NOT symbol in front of the FileCopy.
; Nonetheless, if an error occurs A is still set to @FALSE
; not @TRUE as might be assumed. When an error is suppressed
; with ErrorMode the line is canceled, and any assignment is
; simply set to the default @FALSE value.
;
A = !FileCopy("yyyyyyyyy", "*.*", @FALSE)

```

For this reason, **ErrorMode()** must be used with a great deal of care. The function for which the errors are being suppressed should be isolated from other functions and operators as much as possible.

e.g.

```

; INCORRECT USAGE of ErrorMode( )
; In this instance, when the copy has an error, the entire if
; statement is canceled.
; Execution begins (erroneously) at the next line, and states
; that the copy succeeded. Next a fatal error occurs as the
; "else" is found, since it does not have a matching if

```

```

ErrorMode(@OFF)
if FileCopy(file1,file2,@FALSE) == @TRUE
Message("Info", "Copy worked")
else
Message("Error", "Copy failed")
endif

```

```

; CORRECT USAGE
; In this case, the FileCopy is isolated from other statements
; and flow control logic. When the statement fails, execution
; can safely begin at the next line. The variable "a" will
; contain the default value of zero that a failed assignment
; returns.
; Results are not confused by the presence of other operators.
;

```

```

ErrorMode(@OFF)
a = FileCopy(file1,file2,@FALSE)
ErrorMode(@CANCEL)
if a == @TRUE
Message("Info", "Copy worked")
else
Message("Error", "Copy failed")
endif

```

Debug WIL Scripts

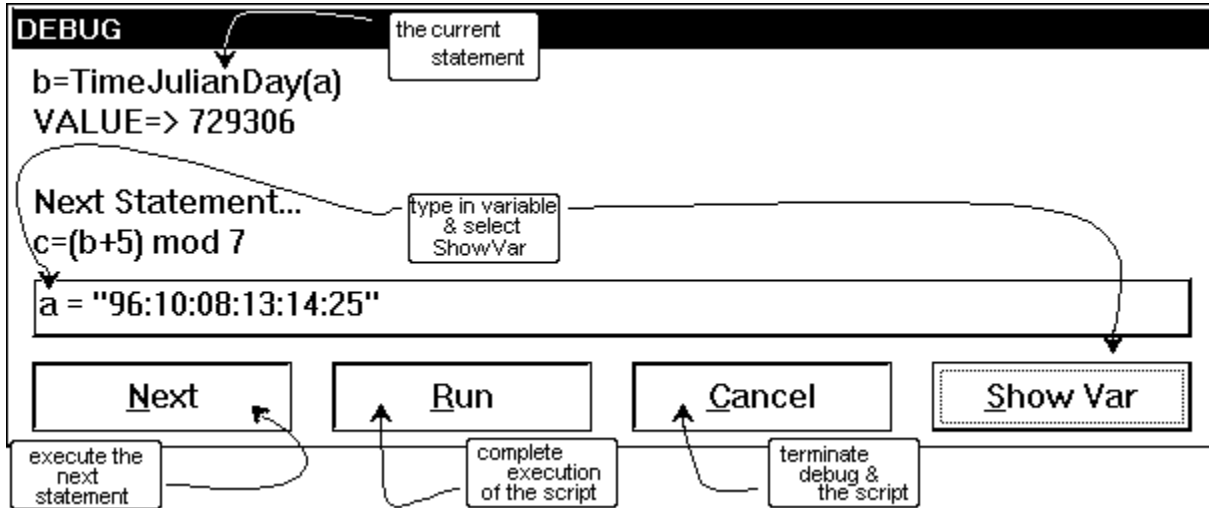
Apparently, there have been sightings of bug free code. However, chances are you will need to exterminate at least one of your WIL scripts. Luckily, WIL has a handy debug utility which comes with the WIL Interpreter.

Debug, a WIL function, is invaluable in determining logic errors. When **Debug** is initialized, a dialog box which controls the execution of each statement is displayed. **Debug** works line by line through the script, displaying the current statement, its value and the following statement. In addition, **Debug** can report on the value of your variables at any given point in the program. Just type the variable into the edit box and select "Show Var". The script will also be executed in conjunction with the display of statements.

Initialize **Debug** by adding **Debug(1)** or **Debug(@ON)** to a specific point in your script. Note: For specific instructions see [Debug](#) in the WIL Function Reference.

Example:

```
Debug(1)
a=TimeYmdHms( )
b=TimeJulianDay(a)
c=(b+5) mod 7
day=ItemExtract(c+1, "Sun Mon Tue Wed Thu Fri Sat", " ")
line=StrCat("Julian Date= ",b,@CRLF,"Day of week= ",day)
Message(TimeDate( ), line)
```



Selection Methods

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:

The image shows a standard Windows-style dialog box. At the top, the title bar is black. Below it, the text 'Enter Filename to edit?' is centered. Underneath the text is a single-line text input field with a vertical cursor on the left. At the bottom of the dialog, there are two buttons: 'Ok' on the left and 'Cancel' on the right. Both buttons have a thin black border and the text is centered within them.

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 several functions we can use for an improved file selection routine.

FileItemize (file-list)

Returns a delimited list of files.

This function compiles a **list** of filenames and separates the names with a delimiter. (The 16-bit version of WIL returns a space-delimited list. The 32-bit version returns a tab-delimited list.) 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:

AskItemList (title, list, delimiter, sort mode, select mode)

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

This is the function which actually displays the list box. Remember that **FileItemize** returns a file list delimited by a character. If the delimiter is a space, the file will look something like this:

```
FILE1.DOC FILE2.DOC FILE3.DOC
```

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

```
IntControl(29, " ", 0, 0, 0)
```

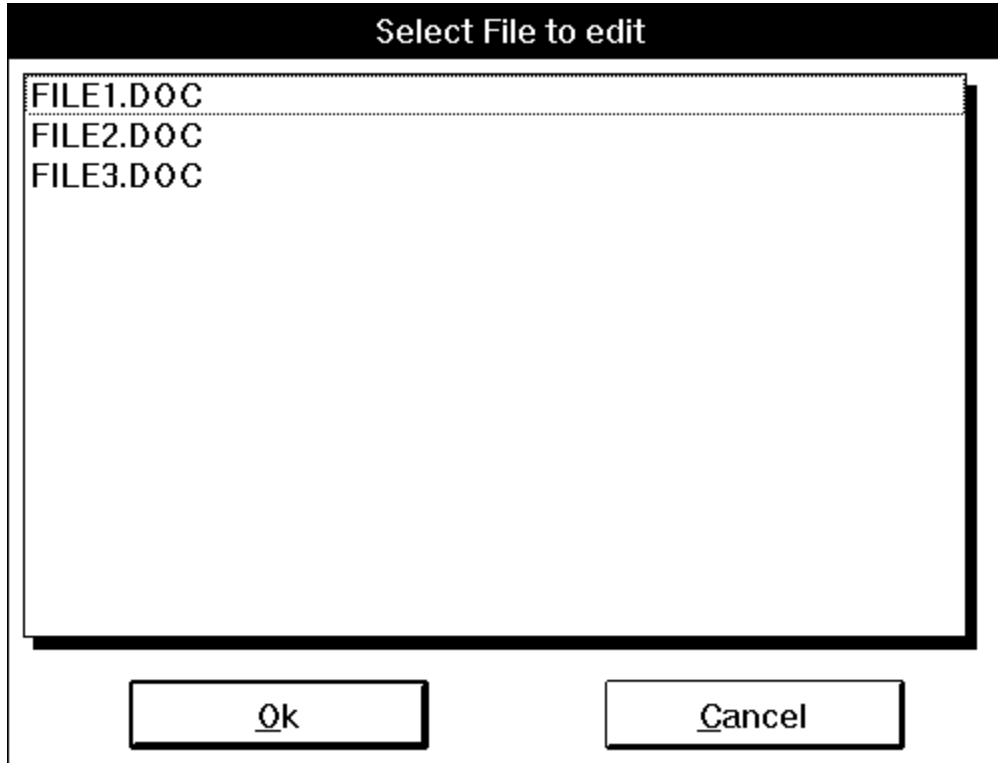


```

files = FileItemize("*.doc *.txt")
afile = AskItemList("Select File to edit", files, " ", @unsorted, @single)
Run("notepad.exe", afile)

```

which produces:



First, we use **FileItemize** to build a list of filenames with DOC and TXT extensions. We assign this list to the variable **files**. Then, we use the **AskItemList** function to build a list box, passing it the variable **files** as its second parameter. The third parameter we use for **AskItemList** is simply a space with quote marks around it; this tells **AskItemList** that the list in variable **files** 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.) Using the fourth parameter set the sort mode to choose how to **display** the text, sorted or unsorted. The fifth parameter sets the select mode allowing you to choose a single item or multiple items from the list. Finally, we assign the value returned by **AskItemList** to the variable **afile**, 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, **AskItemList** returns a null string. If you want, you can test for this condition:

```

IntControl(29, " ", 0, 0, 0)
files = FileItemize("*.doc *.txt")
while @TRUE
afile = AskItemList("Select File to edit", files, " ",
                  @unsorted, @single)
If afile != "" Then break
;break terminates the While structure transferring control to the ;statement
following the endwhile.
endwhile
Run("notepad.exe", afile)

```

DirItemize (dir-list)

Returns a 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. One way we can do this by first letting the user select the appropriate directory, using the **DirItemize** and **AskItemList** combination:

```
IntControl(29, " ", 0, 0, 0)
DirChange("C:\")
subdirs = DirItemize("*. *")
targdir = AskItemList("Select dir", subdirs, " ", @sorted, @single)
if targdir != "" then DirChange(targdir)
files = FileItemize("*. *")
afile = AskItemList("Select File", files, " ", @sorted, @single)
Run("notepad.exe", afile)
```

First we change to the root directory. Then we use **DirItemize** to get a list of all the sub-directories off of the root directory. Next, we use **AskItemList** to give us a list box of directories from which to select. Finally, we change to the selected directory, and use **FileItemize** and **AskItemList** 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 improve our file selection logic:

```
IntControl(29, " ", 0, 0, 0)
DirChange("C:\")
; Directory selection loop
while @TRUE ; Loop forever til break do us part
dirs = DirItemize("")
If dirs == "" Then break
targ = AskItemList("Select dir", dirs, " ", @sorted, @single)
If targ == "" Then break
DirChange(targ)
endwhile
;
; File selection loop
while @TRUE ; Loop forever til break do us part
files = FileItemize("*. *")
afile = AskItemList("Select File", files, " ", @sorted, @single)
If afile != "" Then break
endwhile
Run("notepad.exe", afile)
```

First of all, we set up a repeating **while** loop. The "While **@TRUE**" will repeat the loop forever. In the loop itself we use the **break** statement to exit the loop. After we use the **DirItemize** function to try to get a list of the directories at the current level, we test the returned value for a null string. If we have a null string, then we know that the current directory has no sub-directories, and so we proceed to the file selection logic by **breaking** out of the directory selection loop. If, however, **DirItemize** returns a non-blank list, then we know that there is, in fact, at least one sub-directory. In that case, we use **AskItemList** to present the user with a list box of directories. Then, we test the value returned by **AskItemList**. 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 by **breaking** out of the directory selection loop. On the other hand, a non-blank value returned by **AskItemList** indicates that the user has selected a sub-directory from the list box. In

that case, we change to the selected directory, and the **endwhile** causes the directory selection loop to be repeated. We continue this process until either (a) the user selects a directory, or (b) there are no directories left to select. Eventually, we move to the file selection loop.

Nicer File Selection

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

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.

`@CRLF`

`@TAB`

`@CRLF` and `@TAB` are string constants containing, respectively, a carriage-return line-feed pair and a tab character.

We want to be able to insert a carriage return/line feed combination at the end of each line in our output, and the `@CRLF` string constant will let us do that.. 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:

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

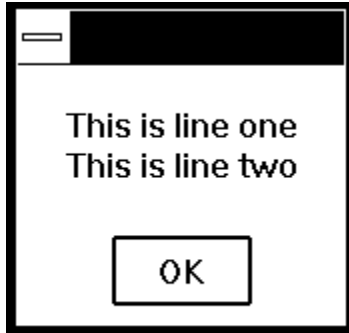
we would not get the desired effect. WIL would simply treat it as ordinary text:



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

```
Message("", "This is line one%@crlf%This is line two")
```

we will get what we want:



Note that there is no space after `%@crlf%`; 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).

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 your programs, just as if you were typing them directly from the keyboard.

SendKeysTo (parent windowname, sendkey string)

Activates the specified window and sends keystrokes to it.

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", "")
SendKeysTo("Notepad", "!fo")
```

The parameters you specify for **SendKeysTo** are the window-name (or at least the first unique part of it), and 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** (see SendKey). 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:



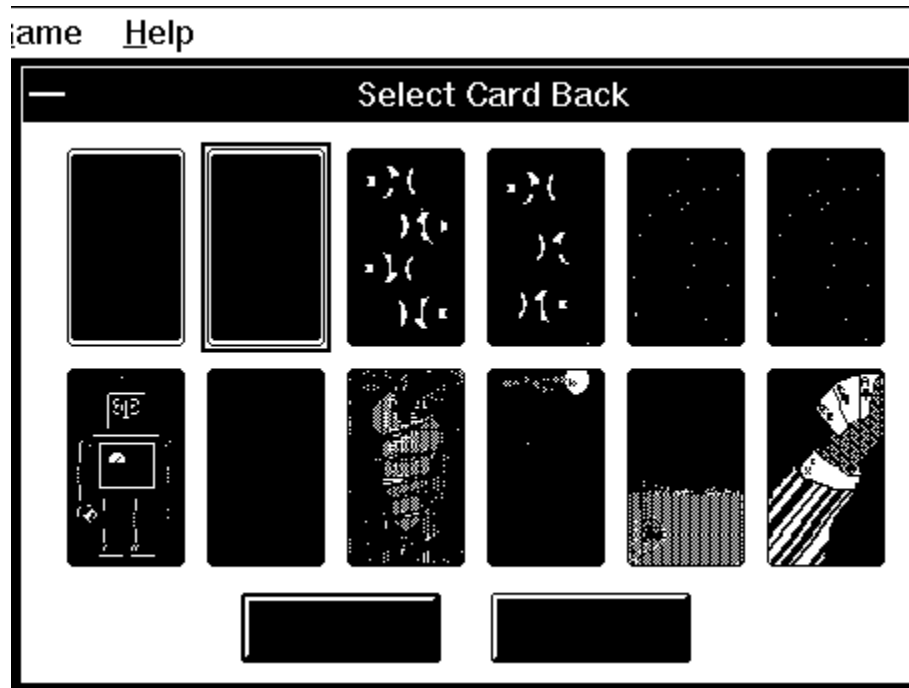
Here's another example:

```
RunZoom("sol.exe", "")
SendKeysTo("Solitaire", "!gc{RIGHT}{SP}~")
```

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

Game	Hel
<u>D</u> eal	
<u>U</u> ndo	
<u>D</u> eck...	
<u>O</u> ptions...	
<u>E</u> xit	

Then it moves the cursor to the next card back style on the right (**{RIGHT}**), selects that card back (**{SP}**), and then selects **OK** (~). (The tilde sign (~) is SendKey shorthand for the enter key.)



And *walla!* 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. 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 [Operators](#). 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!

```

; tutor.wbt
mins = AskLine("Solitaire", "How many mins do you want to play?", "")
mins = Int(mins)
Switch mins
  case 0
    Display(5, "", "Game canceled")
    exit
    break
  case 1
    Message("Only a minute?", "Wow! You've got willpower.")
    break
  case 2
    Message("2 Minutes?", "This isn't much of a break.")
    break
  case 3
    Message("3 Minutes?", "You're barely got time to shuffle")
    break
  case 4
    Message("HA,HA,HA", "I dare you to try to beat me.")
    break
  case mins ;default case - must be last in the switch
    Message("THAT LONG!!!", "Where did you get all that time?")
    break
EndSwitch

program ;;ADDED to demonstrate checking for the existence of a
;before running a second occurrence.

If WinExist("Solitaire") == @TRUE
  WinActivate("Solitaire")
  WinShow("Solitaire")
Else
  Run("sol.exe", "")
Endif

for i = 0 to 9
  j=100-i*10
  k=300+i*70
  WinPlace(j,j,k,k,"Solitaire")
next

WinZoom("Solitaire")

;ADDED to show an example of sending keystrokes.

SendKeysTo("Solitaire", "!gc{RIGHT}{SP}~")

GoSub dumdedum

mins=AskLine("More Time?", "Enter additional minutes", 0)
If mins!=0 then GoSub dumdedum

;ADDED as an example of a while loop.

while WinExist("Solitaire")
  WinClose("Solitaire") ;Make sure it closes
endwhile

```

```
Message("Time's Up", "Get Back to Work!")
```

```
Exit
```

```
:dumdedum  
goal = mins * 60  
timer = 0  
While timer < goal  
    remain = goal - timer  
    if WinExist("Solitaire")  
        WinTitle("Solitaire", "Solitaire (%remain% seconds left)")  
    else  
        exit  
    endif  
    TimeDelay(10)  
    timer = timer + 10  
EndWhile  
Return
```


Creating WIL Script Files

[Next Topic](#)

The WIL Tutorial

- [Topic by Topic](#)
- [The Tutorial Course](#)
- [The Complete WIL](#)

[Tutorial](#)

- [Getting started](#)
- [Using WIL](#)
- [Reference](#)
- [Notes](#)

WIL is a script file interpreter. Before you can do anything useful with the WIL interpreter, you must have at least one WIL script file to interpret.

Your program installation puts several sample scripts into your directory. Suitable icons for these scripts were added to the group in the Windows Program Manager, or to the usual place programs are accessed in your version of Windows.

WIL script files must be formatted as plain text files. You can create them with **WinBatch Studio**, WinEdit (Wilson WindowWare's optional text editor for programmers), the Windows Notepad or another text editor.

Word processors like WordPerfect, AmiPro, and Word can also save scripts in plain text formatted files.

The .WBT (WinBatch) extension is used in this manual for batch file extensions, but, you can use others just as well. If you want to click on a batch file and have Windows run it, be sure that you associate it in Windows with your executable program file. When you installed your program, an association was automatically established between the interpreter and .WBT files.

Each line in a script file contains a statement written in WIL, Wilson WindowWare's Windows Interface Language.

A statement can be a maximum of 255 characters long (refer to the WIL Reference Manual for information on the commands). Indentation does not matter. A statement can contain functions, commands, and comments.

A single Winbatch script cannot exceed 64 kilobytes in file size.

You can give each script file a name which has an extension of WBT (e.g. TEST.WBT). We'll use the terms WinBatch script files and WBT files interchangeably.

Running WIL Utilities

[Next Topic](#)

The WIL Tutorial

- [Topic by Topic](#)
- [The Tutorial Course](#)
- [The Complete WIL](#)

Tutorial

- [Getting started](#)
- [Using WIL](#)
- [Reference](#)
- [Notes](#)

WIL system utilities are very versatile. They can be run from icons in the Windows Explorer.

- as automatic execution macros from the Windows\Start menu\Startup directory
- from macros in word processors and spreadsheets.
- from a command line entry such as the "Start...Run..." in Windows.
- by double clicking or dragging and dropping file names in the Windows Explorer
 - from other WIL scripts to serve as single or multiple "agents", event handlers, or schedulers.
 - from any Windows application or application macro language that can execute another Windows program. Software suite macro languages and application builders like Visual Basic and PowerBuilder are examples of these.

Running WIL System Utilities

[Next Topic](#)

The WIL Tutorial

- [Topic by Topic](#)
- [The Tutorial Course](#)
- [The Complete WIL](#)

[Tutorial](#)

- [Getting started](#)
- [Using WIL](#)
- [Reference](#)
- [Notes](#)

WIL utilities run like any other Windows programs. They can run from a command line, a shortcut, or from a file listing such as the Windows Explorer.

WIL utilities are usually run as files with the extension .WBT. When some WIL utilities are used, they need information passed to them when they run. This is easily done by passing command line parameters to them.

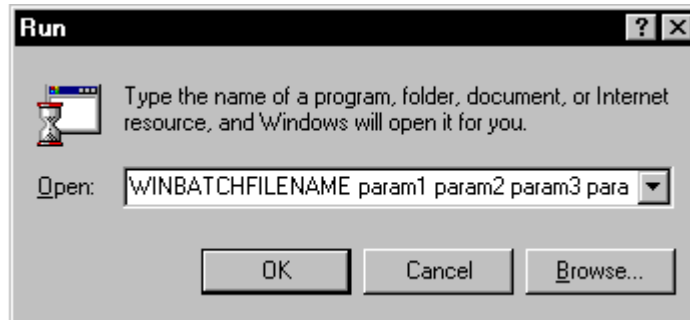
This capability can be used from the command line in Start... Run menu. An example dialog is shown below.

Parameters can be also be passed through the command line entry included in the item properties of any shortcut icon. Finally, an application can send parameters to a WIL utility it launches from a command line or from a function in a macro language.

A command like this runs a WinBatch system utility from a command line or an icon:

```
WinBatchfilename filename.wbt param1 param2 ... param9
```

This command line can be entered into a Command Line text entry box like this one from the task bar (Start..Run):



The command line is longer than the dialog can show, but it can be easily edited with the arrow keys.

WINBATCHFILENAME is the generic name of your WinBatch executable. The specific, or actual, name for the WinBatch application will change to reflect the operating system in use: Windows 3.1, Windows 95/98, and the different Windows NT versions.

"filename.wbt" is any valid WBT file, and is a required parameter.

"p1 p2 ... p9" are optional parameters (there are a maximum of nine of these) to be passed to the WBT file on startup. Each is delimited from the next by one space character.

Parameters passed to a WBT file will be automatically inserted into variables named **param1**, **param2**, etc. The WinBatch utility will be able to use these. An additional variable, **param0**, gives you the total number of command-line parameters.

What is a WIL Program?

[Next Topic](#)

The WIL Tutorial

- [Topic by Topic](#)
- [The Tutorial Course](#)
- [The Complete WIL](#)

[Tutorial](#)

- [Getting started](#)
- [Using WIL](#)
- [Reference](#)
- [Notes](#)

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

[Next Topic](#)

The WIL Tutorial

- [Topic by Topic](#)
- [The Tutorial Course](#)
- [The Complete WIL Tutorial](#)

Windows

Tutor Example

1. Open an editor, such as WinBatch Studio.
2. Type in the line.

Our first WIL program will simply run our favorite Windows application: Solitaire. If you are using a menu script-based implementation of the WIL Interpreter, refer to the preceding section on **Menu Files** for instructions on how to create and edit WIL menu items. 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 **WinBatch Studio**, that is capable of saving text in pure ASCII format. In either case, let's create a WIL program containing the following line of text:

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

- [Getting started](#)
- [Using WIL](#)
- [Reference](#)
- [Notes](#)

3. Save the program with an appropriate extension. If you're using WinBatch, this extension will be **.WBT**. We will refer to this script as **tutor.wbt**.

4. Run **tutor.wbt** by double-clicking on the filename.

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

Windows Int

[Next...](#)

Functions and Parameters

[Next Topic](#)

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

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

The WIL Tutorial

- [Topic by Topic](#)
- [The Tutorial Course](#)
- [The Complete WIL](#)

[Tutorial](#)

- [Getting started](#)
- [Using WIL](#)
- [Reference](#)
- [Notes](#)

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** ([Introduction](#)). 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 arguments 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", "")
```

Note: Strings are delimited by quotes, ("", "")

"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: (s)
denotes the
parameter
requires a string.

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 WIL language standpoint, the second parameter is required. That is, if you omit it, you will get a syntax error, and your WIL program will halt. However, the program that you are running may not need parameters. Solitaire, for example, does not take 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.

"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.

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 WIL language standpoint, the second parameter is required. That is, if you omit it, you will get a syntax error, and your WIL program will halt. However, the program that you are running may not need parameters. Solitaire, for example, does not take 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.



Example 2

1. Open WinBatch Studio.
2. Type in the line.
3. Save the file as *notetest.wbt*.

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.

Is Notepad running? If it is - good. If not, check your script.

4. Close *WinBatch Studio*.
5. Run the *wbt* by double-clicking on the filename.

Hint...

WinBatch will look for Notepad.exe on the computer's search path. If it isn't there, you may need to locate it with the File Manager or the Windows 95/98 Explorer. Add a complete pathname to the run statement and the script should work.

Now, edit the WIL program as follows:

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

Save the program, exit **WinBatch Studio**, 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")
```

Note: semi-colons denote comments.

The semi-colon 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 semi-colon. For example:

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

Windows Int
Next...

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

Displaying Text

[Next Topic](#)



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

Tutor
Example
continue
d...

1. *Exit*
Solitaire.

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

The WIL Tutorial

- [Topic by Topic](#)
- [The Tutorial Course](#)
- [The Complete WIL](#)

Tutorial

- [Getting started](#)
- [Using WIL](#)
- [Reference](#)
- [Notes](#)

2. *Activate or open your editor.*

3. *Load tutor.wbt.*

4. *Add the new lines.*

5. *Save the file.*

6. *Run the wbt by double-clicking on the filename.*

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

Windows Interface Language Help File

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. |

Windows Int

[Next...](#)

The **Display** function has three parameters. The first parameter 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). In our example above, we are displaying the box for 5 seconds. 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 **(s)** in front of a parameter indicates that it is a string. An **(i)** indicates that it is an integer and a **(f)** indicates a floating point number parameter.



[Tutor Example continued...](#)

Now, exit Solitaire (if you haven't done so already), and edit the WIL program by placing a semi-colon 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:

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

1. *Exit Solitaire.* Feel free to experiment with modifying the parameters in the **Display** function. Try adjusting the value of the first parameter.
2. *Edit **tutor.wbt** - Comment out the **Run** statement.* 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**.
3. *Save the file.*

4. *Run the wbt by double-clicking on the filename.* Also, try using a non-integer value, such as 2.9, and see what happens (it will be converted to an integer). Play around with the text in the two string parameters; try making one, or both, null strings ("").

Windows Int

[Next...](#)

Getting Input

[Next Topic](#)

The WIL Tutorial

- [Topic by Topic](#)
- [The Tutorial Course](#)
- [The Complete WIL](#)

[Tutorial](#)

- [Getting started](#)
- [Using WIL](#)
- [Reference](#)
- [Notes](#)

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:

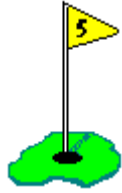
- | | |
|-----|--|
| (i) | @YES or @NO ,
depending on the button
pressed. |
|-----|--|

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, it is a floating point number). 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. Look up a list of all **predefined constants**.

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.)



[Tutor Example continued...](#)

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

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

1. Edit **tutor.wbt**
- Delete the
Display
statement.
Add **AskYesNo**.
Delete semi-
colon from the
Run.

2. Save the file.

3. Run the wbt by
double-clicking
on the filename..

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

Windows Interface Language Help File 

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.

Windows Int

[Next...](#)

Using Variables

[Next Topic](#)

The WIL Tutorial

- [Topic by Topic](#)
- [The Tutorial Course](#)
- [The Complete WIL](#)

Tutorial

- [Getting started](#)
- [Using WIL](#)
- [Reference](#)
- [Notes](#)



[Tutor Example continued...](#)

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?")
```

1. Exit Solitaire.
2. Edit **tutor.wbt**
-Add variable name..
3. Save the file.
4. Run the wbt by double-clicking on the filename..

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.

Windows Int

[Next...](#)

Making Decisions

[Next Topic](#)

The WIL Tutorial

- [Topic by Topic](#)
- [The Tutorial Course](#)
- [The Complete WIL](#)

[Tutorial](#)

- [Getting started](#)
- [Using WIL](#)
- [Reference](#)
- [Notes](#)



WIL provides a way to conditionally execute a statement, and that is by using the **If ... Endif** command. Actually, there are several forms of the **If** statement -- the structured form and the single statement form.

Structured Forms

If expression
series of statements

Endif

If expression
series of statements

Else

series of statements
Endif

Single Statement Forms

If expression **Then** statement.

If expression **Then** statement
Else statement

(We refer to **If ... Endif** as a **command**, rather than a **function**, because **functions** are followed by parameters in parentheses, while **commands** are not. Commands tend to be used to control the WIL interpreter.)

[Tutor Example](#) [continued...](#)

1. *Exit Solitaire.*

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

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

2. *Edit tutor.wbt*

-Add **If/Endif** command to check return of the variable.

3. *Save the file.*

4. *Run the wbt by*

However, as this example is a single statement, rather than a series of statements, the single statement structure is more appropriate. There are generally many different ways to perform any task in WIL. With experience you will be able quickly decide the best way to do any task.

*double-clicking
on the filename.*

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

*Experiment with
additional If
structures below
or*

Windows Int
[continue on...](#)

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 or works out to a non-zero value, then the statement(s) following are performed. If the condition following the **If** keyword is false or works out to a zero value, then the statement(s) following are ignored.

There is something extremely important that you should note about the syntax of these **If ... Endif** commands: 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:

Windows Interface Language Help File 

Note:
*(=) this IS that
(==) this
EQUALS that*

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 statement(s) following the **If** condition are true. But what happens when the condition is false? Remember we said that when the **If** condition is false, the following statement(s) are 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
    Run("sol.exe", "")
Else
    Display(5, "",
"Game canceled")
Endif
```

Using the single statement **If...Then...Else** structure the same code would look like:

```
response=AskYesNo("Really?", "Play
Solitaire now?")
If response == @YES      Then
    Run("sol.exe", "")
    Else Display(5, "", "Game
canceled")
```

When you have only single statements to execute when conditions are true or false, the single statement form may be preferred. However, 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", "")
    If response==@NO Then
        Display(5,"","Game canceled")
```


Clearly, the best way of handling this is to use the **If... Else... Endif** structured form.

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

Control of Program Flow

[Next Topic](#)

The WIL Tutorial

- [Topic by Topic](#)
- [The Tutorial Course](#)
- [The Complete WIL Tutorial](#)
- [Getting started](#)
- [Using WIL](#)
- [Reference](#)
- [Notes](#)



The linear flow of statements (executing one statement after another) is not always preferred or possible. WIL provides the standard set of flow control commands: **For**, **While**, **Switch** and **GoSub**. These commands give you the ability to redirect the **flow of control** in your WIL programs.

The **For** command controls the looping of a block of code based on an incrementing index. The **While** command conditionally and/or repeatedly executes a series of statements. The **Switch** statement allows selection among multiple blocks of statements. **GoSub** transfers control to another point in the WIL program and saves the location for a **Return** statement.

Let's explore the use of these commands further. Perhaps you need to break your Solitaire habit by limiting your time of play (it has, by now, become obvious to your boss and co-workers that, ever since you got this program, all you do is play solitaire).

[Tutor Example continued...](#)

First you need to ask yourself how long you would like to play by adding the following line to the top of your script.

```
mins=AskLine("Solitaire","How many mins do you want to play?","")
```

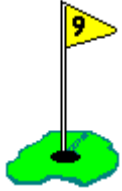
1. *Exit Solitaire.*
(If it is running.)
2. *Edit **tutor.wbt***
*-Add **AskLine** to the top of the script.*
3. *Save the file.*
4. *Run the wbt by double-clicking on the filename.*

This will **display** a message box which prompts you for the number of minutes you would like to play.

Once you enter the desired number of minutes, you could display an additional message as a response to the specific amount of time entered. **Switch**, as you remember, allows selection from among multiple blocks of statements. Each block of statements is called a **case**. In the sample below, there are several case statement blocks. Selection of one of the cases is determined by the number of minutes stored in the variable **mins**. If the number is 3, then case 3 will be executed. All numbers not accounted for will be

Windows Int executed by the default case, **mins**.

[Next...](#)



1. *Exit Solitaire.*
 2. *Edit **tutor.wbt** to add the **Switch** statement.*
- Type the following code into **tutor.wbt**. Test the Switch statement by entering a number into the **AskLine** dialog box. Try running it several times using various numbers.

Your code should look like the code below. Remove extra statements.

```
mins = AskLine("Solitaire", "How many mins do you want to play?", "")
mins = Int(mins)
Switch mins
  case 0
    Display(5, "", "Game canceled")
    exit
    break
  case 1
    Message("Only a minute?", "Wow! You've got willpower.")
    break
  case 2
    Message("2 Minutes?", "This isn't much of a break.")
    break
  case 3
    Message("3 Minutes?", "You've barely got time to shuffle.")
    break
  case 4
    Message("HA,HA,HA", "I dare you to try to beat me.")
    break
  case mins ;default case - must be last in the switch
    Message("THAT LONG!!!", "Where did you get all that time?")
    break
EndSwitch
Run("sol.exe", "")
```

Windows In

[Next...](#)

In our example, each case statement block is composed of three parts; a **case** statement followed by a number, a series of one or more statements and the **break** command. If the number behind the **case** statement matches the number behind the **switch** statement, then the **case** block is executed. Once the correct message has been displayed, **break** terminates the case block and transfers control to the **EndSwitch** statement.



Tutor Example continued...

1. Exit Solitaire.
2. Edit **tutor.wbt** Set up variables.

Now we need to create a timer to track the time elapsed and compare it to the time entered. The **While** command, which repeats execution of a series of statements by telling WIL, "Do the following while a condition is present," does this job nicely.

First let's set up a couple of variables.

```
goal = mins * 60
timer = 0
```

Add the variables and timing code below the **Run** statement.

Now for the **While** statement. The first line sets the condition, "While the **timer** is less than the **goal** execute this series of statements."

```
While timer < goal
  remain = goal - timer
  WinTitle("Solitaire","Solitaire (%remain
% seconds left)")
  TimeDelay(10)
  timer = timer + 10
EndWhile
```

3. Save the file.
4. Run the **wbt** by double-clicking on the filename..

The rest of our series of statements include: a computation of the time remaining (**remain**) to be displayed, a line to display the time remaining in the Solitaire window title bar, a **delay** statement to allow time to pass, and a statement to calculate the time elapsed. EndWhile marks the end of statements. WIL marches through the **While** loop until the variable **timer** exceeds the value of the variable **goal**.

Windows Int

Next...



Tutor Example continued...

1. Exit Solitaire.

So what happens if suddenly your time is up and you're four moves away from winning? Can't have that happening.

We can give ourselves the opportunity to add more time by adding another Askline statement.

```
mins=AskLine("More Time?","Enter
additional minutes.",0)
```

2. Edit **tutor.wbt** Add new line to bottom of script.
3. Save the file.
4. Run the **wbt** by

If a time is entered the timer will need to be used again. Of course, it would be easy to copy that portion of the script and insert it after the new line. However, the same script can be utilized with the assistance of **GoSub**.

GoSub causes the flow of control to go to another point in the WIL program while remembering its point of origin. The

double-clicking
on the filename.

Windows Int

[Next...](#)

name **GoSub** is an abbreviation of "Go To Subroutine". You must specify where you want the flow of control to be transferred -- the subroutine name, and you must mark this point with a **label**. A label is simply a destination address, or marker. The form of the **GoSub** command is:

GoSub 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.



[Tutor Example continued...](#)

1. Exit Solitaire.
2. Edit **tutor.wbt**-
Add **:dumdedum**
label to bottom.
Move timing loop
below label.
Add Return after
loop. After **Run**
statement add
GoSub
dumdedum.
3. Save the file.
4. Run the wbt by
double-clicking
on the filename.

Windows Int

[Next...](#)

In our sample script, we move the timing loop to the bottom of the script, add a label marked **:dumdedum** above the timing script as the destination address. After **EndWhile**, add the statement, **Return** to allow the flow of control to return from the bottom of the **GoSub**.

We'll add a **GoSub** statement in after the **Run** statement. The **GoSub** statement is saying, in English "go to the line marked **:dumdedum**, and continue processing from there, but remember where you came from." When **Return** is reached, control will be transferred back to the statement after the original **GoSub**.

Notice that the label **dumdedum** is preceded by a colon as the address, but *not* on the line where it follows the **GoSub** keyword. This is important. Although you can have multiple lines in your WIL program which say **GoSub dumdedum**, you can have only one line marked **:dumdedum** (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.



[Tutor Example continued...](#)

1. Exit Solitaire.

In addition to changing the message displayed in the "mins=AskLine" statement, a default time has been added. The value returned from the new AskLine will need to be checked. In the example below, "!" signifies "not equal to". Therefore the line reads, "If mins is not equal to zero then GoSub dumdedum."

```
If mins!=0 then
```

GoSub dumdedum

2. Edit **tutor.wbt**

-
Add check for
return of the 2nd
AskLine.
Add **WinClose**
and **Exit**. (See
code below)

3. Save the file.

4. Run the wbt by
double-clicking
on the filename.

If a time is returned, **GoSub** will send execution to the
:dumdedum label and the waiting process will begin again.
After the time has elapsed, control will be returned to the
statement following the **GoSub**.

The last thing we want to do is end the program with the
WinClose function and display a final message.

The **Exit** command is used to keep the processing from
"falling through" to the subroutine at the end of the
program. In this case, the dumdedum subroutine sits at the
end. **Exit** causes a WIL program to end immediately and
not fall into the dumdedum loop.

Windows Int

[Next...](#)



Our altered script has the following appearance from the **Run** statement to the bottom.

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

GoSub dumdedum

mins=AskLine("More Time?", "Enter additional minutes", 0)
If mins!=0 then GoSub dumdedum

WinClose("Solitaire")
Message("Time's Up", "Get Back to Work!")

Exit

:dumdedum
goal = mins * 60
timer = 0
While timer < goal
    remain = goal - timer
    WinTitle("Solitaire", "Solitaire (%remain% seconds left)")
    TimeDelay(10)
    timer = timer + 10
EndWhile
Return
```

Windows Interface 1

[Next...](#)



[Tutor Example continued...](#)

1. Edit *tutor.wbt*

Add code for the For statement directly after the Run statement.

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

for i = 0 to 9
  j=100-i*10
  k=300+i*70
  WinPlace(j,j,k,k, "Solitaire")
next

WinZoom("Solitaire")
```

2. Save the file.

3. Run the wbt by double-clicking on the filename.

The sample script could be considered complete at this point. However, the **For** command has yet to be discussed. The **For** command is more complex than the previous commands. It controls the looping of a block of code based on an incrementing index. This command is handy if you want to perform a specific code block a particular number of times. The statement says, "Repeat the block of code for each value of a variable from the initial value to the final value, incrementing the variable after each pass of the loop"

In the sample below, the size of the Solitaire window is manipulated and displayed 10 times before the window is zoomed to full screen. Each time the loop is executed, the coordinate and size variables (j and k) are altered, and then used in a **WinPlace** statement (it's time to start looking up [functions](#) in the reference yourself now) to affect the position and size of the Solitaire window.

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.

See the

[Windows Interface Language Help File](#)



[Completed WIL file](#), **tutor.wbt**, at the end of the WIL TUTORIAL.

Exploring WIL

[Next Topic](#)

The WIL Tutorial

- [Topic by Topic](#)
- [The Tutorial Course](#)
- [The Complete WIL](#)

[Tutorial](#)

- [Getting started](#)
- [Using WIL](#)
- [Reference](#)
- [Notes](#)

If you take a moment and look through the [WIL Function Reference](#) that takes up most of this help file, you will notice that WIL uses a very convenient naming convention. WIL [functions](#) are named so that the object affected by the function is the first word in the function name -- any function dealing with Files starts with the word "File", and they can be found clumped together in the alphabetically arranged function reference. If you think you might want a function dealing with DDE, simply search for the word DDE and scan the available functions.

What follows is just quick overview of the many functions and [commands](#) available in WIL. These should be sufficient to begin creating versatile and powerful WIL programs. For complete information on these and all WIL functions and commands, refer to the **WIL Function Reference** (see [Introduction](#)).

Running Programs

[Next Topic](#)

The WIL Tutorial

- [Topic by Topic](#)
- [The Tutorial Course](#)
- [The Complete WIL](#)

[Tutorial](#)

- [Getting started](#)
- [Using WIL](#)
- [Reference](#)
- [Notes](#)

There are several [functions](#) that you can use to start an [application](#), most of which share a common syntax. To name a few:

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 system 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("excel", "bigsheet.xls")
```

RunIcon (program-name, parameters)

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

Example:

```
RunIcon("notepad", "")
```

All these Run functions simply launch the program and continue with WIL processing. If you need to wait until the program exits before continuing, then there are a number of other suitable functions also available.

RunWait (program-name, parameters)

RunWait starts a program and waits for it to exit before continuing.

RunZoomWait (program-name, parameters)

RunZoomWait starts a program as a full screen window and waits for it to exit before continuing.

RunIconWait (program-name, parameters)

RunIconWait starts a program as an icon at the bottom of the screen and waits for it to exit before continuing.

If all these Run functions are too much for you, there is also the combination **RunShell** function, which combines all the capabilities of the Run functions and adds additional capability.

RunShell (program-name, parameters, working dir, view, waitflag)

RunShell is an advanced form of the Run function that even allows the specification of a working directory, along with the window view mode and whether or not to wait for completion of the run program in a single function.

Display and Input

Next Topic

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

The WIL Tutorial

- Topic by
Topic
- The Tutorial
Course
- The
Complete WIL Tutorial

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, "Please wait", "Loading Solitaire now")
```

- Getting started
- Using WIL
- Reference
- Notes

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
```

TimeDelay (seconds)

Pauses WIL program execution.

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

Manipulating Windows

[Next Topic](#)

The WIL Tutorial

- [Topic by Topic](#)
- [The Tutorial Course](#)
- [The Complete WIL](#)

[Tutorial](#)

- [Getting started](#)
- [Using WIL](#)
- [Reference](#)
- [Notes](#)

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.

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 **Alt-Tab** and seeing how many instances are open. 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 as follows:

```
If WinExist("Solitaire") == @TRUE
WinActivate("Solitaire")
Else
Run("sol.exe", "")
Endif
```

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

```
If WinExist("Solitaire") == @FALSE
Run("sol.exe", "")
Else
WinActivate("Solitaire")
Endif
```

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.

See Also: **IntControl** 66, 67, or 68.

Files and Directories

[Next Topic](#)

The WIL Tutorial

- [Topic by Topic](#)
- [The Tutorial Course](#)
- [The Complete WIL Tutorial](#)

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", "")
```

- [Getting started](#)
- [Using WIL](#)
- [Reference](#)
- [Notes](#)

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
FileCopy("win.ini", "win.bak", @FALSE)
endif
Run("notepad.exe", "win.ini")
```

FileCopy (source-list, destination, 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, along with selecting various options for copying the files. 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 (source-list, destination)

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
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")
Endif
Run("winword.exe", "wil.doc")
Exit
```

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.

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 resulting in a message resembling the following:

Windows Interface Language Help File 

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

Next Topic

The WIL Tutorial

- Topic by Topic
- The Tutorial Course
- The Complete WIL

Tutorial

- Getting started
- Using WIL
- Reference
- Notes

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
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)
Endif
Run("winword.exe", "wil.doc")
```

Exit

Notice how we've used **ErrorMode(@OFF)** to prevent errors in the **If** statement section 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.

Note: Pay close attention when suppressing errors with the **ErrorMode** function. When an error occurs, the processing of the ENTIRE line is canceled. Setting the **ErrorMode()** to **@OFF** or **@NOTIFY** allows execution to resume at the next line.

Various parts of the original line may have not been executed.

e.g.

```
ErrorMode(@OFF)
; The FileCopy will cause a file not found error,
; canceling the execution of the whole line.
; The variable A is set to @FALSE by default
  A = FileCopy( "xxxxxxxx", "*.*", @FALSE)
;
;
; Now there is a NOT symbol in front of the FileCopy.
; Nonetheless, if an error occurs A is still set to @FALSE
; not @TRUE as might be assumed. When an error is suppressed
; with ErrorMode the line is canceled, and any assignment is
; simply set to the default @FALSE value.
;
  A = !FileCopy("yyyyyyyyyy", "*.*", @FALSE)
```

For this reason, **ErrorMode()** must be used with a great deal of care. The function for which the errors are being suppressed should be isolated from other functions and operators as much as possible.

e.g.

```
; INCORRECT USAGE of ErrorMode( )
; In this instance, when the copy has an error, the entire if
; statement is canceled.
; Execution begins (erroneously) at the next line, and states
; that the copy succeeded. Next a fatal error occurs as the
; "else" is found, since it does not have a matching if
```

```
ErrorMode(@OFF)
if FileCopy(file1,file2,@FALSE) == @TRUE
  Message("Info", "Copy worked")
else
  Message("Error", "Copy failed")
endif
```

```
; CORRECT USAGE
; In this case, the FileCopy is isolated from other statements
; and flow control logic. When the statement fails, execution
; can safely begin at the next line. The variable "a" will
; contain the default value of zero that a failed assignment
; returns.
; Results are not confused by the presence of other operators.
;
```

```
ErrorMode(@OFF)
a = FileCopy(file1,file2,@FALSE)
ErrorMode(@CANCEL)
if a == @TRUE
    Message("Info", "Copy worked")
else
    Message("Error", "Copy failed")
endif
```

Debug WIL Scripts

[Next Topic](#)

The WIL Tutorial

- [Topic by Topic](#)
- [The Tutorial Course](#)
- [The Complete WIL](#)

[Tutorial](#)

- [Getting started](#)
- [Using WIL](#)
- [Reference](#)
- [Notes](#)

Apparently, there have been sightings of bug free code. However, chances are you will need to exterminate at least one of your WIL scripts. Luckily, WIL has a handy debug utility which comes with the WIL Interpreter.

Debug, a WIL function, is invaluable in determining logic errors. When **Debug** is initialized, a dialog box which controls the execution of each statement is displayed. **Debug** works line by line through the script, displaying the current statement, its value and the following statement. In addition, **Debug** can report on the value of your variables at any given point in the program. Just type the variable into the edit box and select "Show Var". The script will also be executed in conjunction with the display of statements.

Initialize **Debug** by adding **Debug(1)** or **Debug(@ON)** to a specific point in your script. Note: For specific instructions see [Debug](#) in the WIL Function Reference.

Example:

```
Debug(1)
a=TimeYmdHms( )
b=TimeJulianDay(a)
c=(b+5) mod 7
day=ItemExtract(c+1, "Sun Mon Tue Wed Thu Fri Sat", " ")
line=StrCat("Julian Date= ",b,@CRLF,"Day of week= ",day)
Message(TimeDate( ), line)
```

Selection Methods

[Next Topic](#)

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")
```

The WIL Tutorial

- [Topic by Topic](#)
- [The Tutorial Course](#)
- [The Complete WIL Tutorial](#)

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)
```

- [Getting started](#)
- [Using WIL](#)
- [Reference](#)
- [Notes](#)

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 several functions we can use for an improved file selection routine.

FileItemize (file-list)

Returns a delimited list of files.

This function compiles a **list** of filenames and separates the names with a delimiter. 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:

AskItemList (title, list, delimiter, sort mode, select mode)

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

This is the function which actually displays the list box.

Remember that **FileItemize** returns a file list delimited by a character. If the delimiter is a space, the file will look something like this:

```
FILE1.DOC FILE2.DOC FILE3.DOC
```

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

```
IntControl(29, " ", 0, 0, 0)
files= FileItemize("*.doc *.txt")
afile= AskItemList("Select File to edit",files," ",
                 @unsorted, @single)
Run("notepad.exe", afile)
```

which produces:

Windows Interface Language Help File

First, we use **FileItemize** to build a list of filenames with DOC and TXT extensions. We assign this list to the variable **files**. Then, we use the **AskItem**List function to build a list box, passing it the variable **files** as its second parameter. The third parameter we use for **AskItem**List is simply a space with quote marks around it; this tells **AskItem**List that the list in variable **files** 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.) Using the fourth parameter, set the sort mode to choose how to **display** the text, sorted or unsorted. The fifth parameter sets the select mode allowing you to choose a single item or multiple items from the list. Finally, we assign the value returned by **AskItem**List to the variable **afile**, 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, **AskItemList** returns a null string. If you want, you can test for this condition:

```
IntControl(29, " ", 0, 0, 0)
files = FileItemize("*.doc *.txt")
while @TRUE
  afile = AskItemList("Select File to edit",
                    files, " ", @unsorted, @single)
  If afile != "" Then break
;break terminates the While structure
;transferring control to the statement
;following the endwhile.
endwhile
Run("notepad.exe", afile)
```

DirItemize (dir-list)

Returns a delimited list of directories.

This function is similar to **FileItemize**, but instead of returning a list of files, it returns a list of directories. (The 16-bit version of WIL returns a space-delimited list. The 32-bit version returns a tab-delimited list.) 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. One way we can do this by first letting the user select the appropriate directory, using the **DirItemize** and **AskItemList** combination:

```
IntControl(29, " ", 0, 0, 0)
DirChange("C:\")
subdirs = DirItemize("*. *")
targdir=AskItemList("Select dir", subdirs, " ",@sorted,@single)
if targdir != "" then DirChange(targdir)
files = FileItemize("*. *")
afile=AskItemList("Select File to edit",files," ",@sorted,@single)
Run("notepad.exe", afile)
```

First we change to the root directory. Then we use **DirItemize** to get a list of all the sub-directories off of the root directory. Next, we use **AskItemList** to give us a list box of directories from which to select. Finally, we change to the selected directory, and use **FileItemize** and **AskItemList** 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 improve our file selection logic:

```

IntControl(29, " ", 0, 0, 0)
DirChange("C:\")
; Directory selection loop
while @TRUE ; Loop forever til break do us part
  dirs = DirItemize("")
  If dirs == "" Then break
  targ = AskItemList("Select dir", dirs, " ", @sorted, @single)
  If targ == "" Then break
  DirChange(targ)
endwhile
;
; File selection loop
while @TRUE ; Loop forever til break do us part
  Files = FileItemize("*. *")
  afile=AskItemList("Select File to edit",files," ",@sorted,@single)
  If afile != "" Then break
endwhile
Run("notepad.exe", afile)

```

First of all, we set up a repeating **while** loop. The "While **@TRUE**" will repeat the loop forever. In the loop itself we use the **break** statement to exit the loop. After we use the **DirItemize** function to try to get a list of the directories at the current level, we test the returned value for a null string. If we have a null string, then we know that the current directory has no sub-directories, and so we proceed to the file selection logic by **breaking** out of the directory selection loop. If, however, **DirItemize** returns a non-blank list, then we know that there is, in fact, at least one sub-directory. In that case, we use **AskItemList** to present the user with a list box of directories. Then, we test the value returned by **AskItemList**. 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 by **breaking** out of the directory selection loop. On the other hand, a non-blank value returned by **AskItemList** indicates that the user has selected a sub-directory from the list box. In that case, we change to the selected directory, and the **endwhile** causes the directory selection loop to be repeated. We continue this process until either (a) the user selects a directory, or (b) there are no directories left to select. Eventually, we move to the file selection loop.

Nicer File Selection

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

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.

@CRLF

@TAB

@CRLF and @TAB are string constants containing, respectively, a carriage-return line-feed pair and a tab character.

We want to be able to insert a carriage return/line feed combination at the end of each line in our output, and the @CRLF string constant will let us do that. 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:

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

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

Windows Interface Language Help File 

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

```
Message("", "This is line one%@crlf%This is line  
two")
```

we will get what we want:

Windows Interface Language Help File 

Note that there is no space after %@crlf%; 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).

Running DOS Programs

Next Topic

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

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

The WIL Tutorial

- Topic by Topic
- The Tutorial Course
- The Complete WIL Tutorial

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")
```

- Getting started
- Using WIL

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")
```

- Reference
- Notes

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

[Next Topic](#)

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

The WIL Tutorial

- [Topic by Topic](#)
- [The Tutorial Course](#)
- [The Complete WIL](#)

[Tutorial](#)

SendKeysTo (parent-windowname, sendkey string)

Activates the specified window and sends keystrokes to it.

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", "")
SendKeysTo("Notepad", "!fo")
```

- [Getting started](#)
- [Using WIL](#)
- [Reference](#)
- [Notes](#)

The parameters you specify for **SendKeysTo** are the window-name (or at least the first unique part of it), and 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** (see SendKey). 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", "")
SendKeysTo("Solitaire", "!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** (~). (The tilde sign (~) is SendKey shorthand for the enter key.

Windows Interface Language Help File 

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

Our Completed WIL File

The WIL Tutorial

- [Topic by Topic](#)
- [The Tutorial Course](#)
- [The Complete WIL](#)

Tutorial

- [Getting started](#)
- [Using WIL](#)
- [Reference](#)
- [Notes](#)

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

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 [Operators](#).

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!



Tutor.wbt

```
; tutor.wbt
mins = AskLine("Solitaire", "How many mins do you want to play?", "")
mins = Int(mins)
Switch mins
  case 0
    Display(5, "", "Game canceled")
    exit
    break
  case 1
    Message("Only a minute?", "Wow! You've got willpower.")
    break
  case 2
    Message("2 Minutes?", "This isn't much of a break.")
    break
  case 3
    Message("3 Minutes?", "You're barely got time to shuffle")
```

```

        break
    case 4
        Message("HA,HA,HA", "I dare you to try to beat me.")
        break
    case mins ;default case - must be last in the switch
        Message("THAT LONG!!!", "Where did you get all that time?")
        break
EndSwitch

program ;;ADDED to demonstrate checking for the existence of a
;before running a second occurrence.

If WinExist("Solitaire") == @TRUE
    WinActivate("Solitaire")
    WinShow("Solitaire")
Else
    Run("sol.exe", "")
Endif

for i = 0 to 9
    j=100-i*10
    k=300+i*70
    WinPlace(j,j,k,k,"Solitaire")
next

WinZoom("Solitaire")

;ADDED to show an example of sending keystrokes.

SendKeysTo("Solitaire", "!gc{RIGHT}{SP}~")

GoSub dumdendum

mins=AskLine("More Time?", "Enter additional minutes", 0)
If mins!=0 then GoSub dumdendum

;ADDED as an example of a while loop.

while WinExist("Solitaire")
    WinClose("Solitaire") ;Make sure it closes
endwhile
Message("Time's Up", "Get Back to Work!")

Exit

```

```
:dumdedum
goal = mins * 60
timer = 0
While timer < goal
    remain = goal - timer
    if WinExist("Solitaire")
        WinTitle("Solitaire", "Solitaire (%remain% seconds left)")
    else
        exit
    endif
    TimeDelay(10)
    timer = timer + 10
EndWhile
Return
```

Using WIL

WIL scripts are written in a plain text file, which can be created by Notepad or most word processors. (Of course, we recommend our own WinEdit, which has many features designed expressly for programmers, including a full-featured implementation of WIL itself.)

These text files can take one of two forms, depending on your particular implementation of WIL: [batch files](#) or [menu files](#).

Batch Files

A batch file is simply a list of WIL commands and function calls, executed in order (just like the old DOS batch language).

Menu Files

A menu file is similar to a batch file, except that multiple chunks of WIL code are organized into menu and sub-menus, and each routine is launched by pressing the appropriate keystroke or selecting an item from the menu. (The name and location of the menus vary depending on the particular implementation of WIL menu files.)

- [Products that use WIL](#)
- [Batch files](#)
- [Menu files](#)
- [Reference](#)
- [Step by step guide to learning WIL](#)
- [Notational Conventions](#)
- [Notes](#)

Batch Files

Windows Interface Language (WIL) scripts are written in a plain text file, which can be created by Notepad or most word processors. (Of course, we recommend our own WinEdit, which has many features designed expressly for programmers, including a full-featured implementation of WIL itself.)

These text files can take one of two forms, depending on your particular implementation of WIL: batch files or [menu files](#).

WinBatch and WebBatch each execute batch files. A batch file is simply a list of WIL commands and function calls, executed in order (just like the old DOS batch language). Each line must be less than 255 characters in length.

- [Using WIL](#)
- [Menu files](#)
- [Reference](#)
- [Step by step guide to learning WIL](#)
- [Notes](#)

Menu Files

WIL scripts can be implemented in two ways: via [batch files](#) or [menu files](#). In a batch process, WIL scripts are associated with the WIL processor, allowing them to be initiated and run on the desktop just as any true executable is launched and run.

WIL scripts can also be launched as menu items from a drop down menu. However, you must have an implementation of WIL with the capability of generating the menu either within one of our applications or as an enhancement to standard Windows applications. In Windows 95/98/NT, WIL adds menu capability to the Windows Task Bar and the Shortcut Menu in the Windows 95/98/NT Explorer.

Please see either the help file or printed documentation that came with your program for more information.

- [Menu file structure](#)
- [Modifying menus](#)
- [Menu hotkeys](#)
- [Menu items](#)
- [Batch files](#)
- [Products that use WIL](#)
- [Reference](#)
- [Step by step guide to learning WIL](#)

Menu File Structure

WIL menus are defined in a **menu file**, which is simply a standard ASCII text file (the kind created by Notepad). Each menu file consists of one or more lines of menu statements. Each line is terminated with a carriage return / line feed (CRLF) combination, and can be up to 255 characters long. Generally, these files have an extension of .MNV. See your product documentation for the name of the default menu file that it uses.

Every menu file contains one or more **menu items**. When activated they appear as 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** which the WIL Interpreter will execute when you choose the item.

Essentially, your menu file is an outline of menu options, formatted with specific spacing, and containing snippets of code.

There are two main parts of a menu file:

The first section, which is optional, is the **initialization** code. This section is executed once when the menu is first loaded and run. It's located before the first **menu item** declaration.

The remainder of the menu file consists of menu item titles and their associated statements. The code under each menu title is executed when the corresponding menu item is selected. Execution begins at the first statement under a menu item and continues up to the definition of the next item.

Menus can be up to four levels deep. Levels are determined by the position of the first letter in the menu title. The top level menu starts at Column 1, the second starts in Column 2, and so on. The WIL code must begin at Column 5 or greater. Same level menu items must be separated by WIL code.

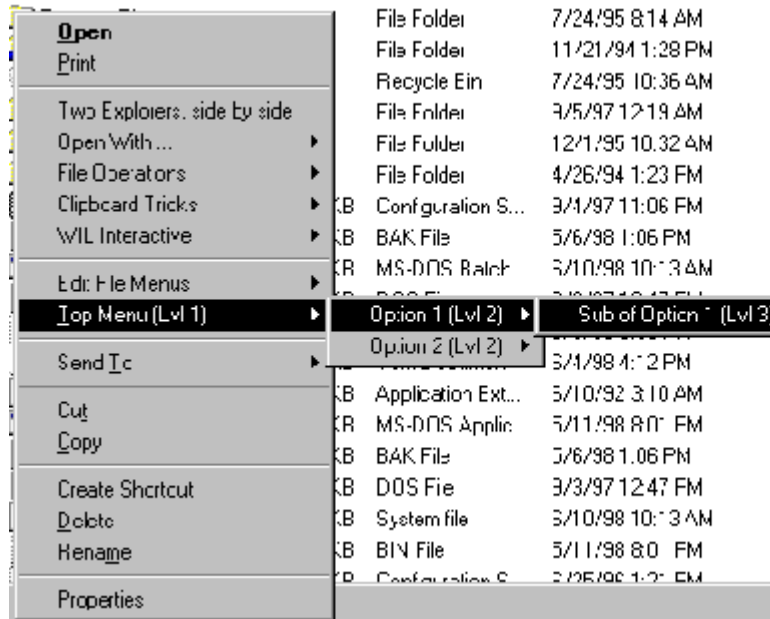
In the example below, Option #1 and Option #2 are separated by the WIL code which is to be executed.

```
&Top Menu (Lvl 1)
Option 1 (Lvl 2)
  Submenu of Option 1 (Lvl 3)
    Submenu of Option 1 (Lvl 4)
      WIL code

Option 2 (Lvl 2)
  Submenu (Lvl 3)
    WIL code
```

This script creates the following menu in the Windows 95/98/NT Explorer.

- [Menu files](#)
- [Modifying menus](#)
- [Menu hotkeys](#)
- [Menu items](#)
- [Reference](#)
- [Step by step guide to learning WIL](#)



Your application probably included a pre-defined sample menu. Refer to it for a practical example of correct menu structure.

Here is another example of 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. It 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). 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", "")
```

Here, 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**.

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 supported applications such as FileMenu, a comment can be displayed on the status bar in the Windows Explorer. This works only for top level menu items. The comment must be on the same line as the top level item. For example, the menu item below is a main menu for running Games. "Killers of Time" is the comment that appears in the status bar.

```
&Games      ;Killers of Time
  &Solitaire
    Run("sol.exe", "")
```

In addition to top-level menus, you can optionally define up to 3 additional levels of **submenus**. The titles for the first-level, second-level and third-level submenus must begin in columns 2,3 and 4, 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 (beginning in column 2), which contains two menu items (beginning 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.

Some applications allow you 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 earlier, menu files must be created and edited with an editor, such as **WinBatch Studio**, 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 files](#)
- [Menu file structure](#)
- [Menu hotkeys](#)
- [Menu items](#)
- [Reference](#)
- [Step by step guide to learning WIL](#)

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. (WinEdit has this capability, PopMenu and FileMenu do not.) Hotkeys 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.

- [Menu files](#)
- [Menu file structure](#)
- [Modifying menus](#)
- [Menu items](#)
- [Reference](#)
- [Step by step guide to learning WIL](#)

Menu Items

Menu titles can consist of letters, digits, spaces, punctuation marks in fact any displayable ANSI characters your text editor can create.

There are special characters you can use to modify the appearance of items in the menus.

- &** Causes the following character to be underlined in the menu item. The user can select the item by pressing the ALT key with the character instead of using the mouse.
- |** In a main menu, puts this item on a new line.
- |** In a drop down menu, this item starts a new column.
- _** Used to create a horizontal bar (in drop down menus only).
- @** Causes the item not to be displayed in the menu.

In order to identify a menu item within a WIL statement, each menu item you define has an associated **menu name**. The menu name is built using only the letters and digits that make up the menu title. Menu names are case-*insensitive*; you don't have to worry about how the actual menu title is capitalized in order to identify it.

For menu items in a **pop-up** menu, the menu name consists of its parent menu's name, plus the pop-up menu item's name concatenated at the end.

Top-level menu names must begin in column 1. Submenu names are optional, and if used must begin in column 2-4; each column of indentation represents an additional level of submenu nesting. Actual menu code must begin in column 5 or higher, and must appear directly under the menu name to which it belongs.

Example:

- [Menu files](#)
- [Menu file structure](#)
- [Modifying menus](#)
- [Menu hotkeys](#)
- [Reference](#)
- [Step by step guide to learning WIL](#)

```

Set &User Level
  PW=AskLine ("","Enter your password:", "")
  ;assuming the resident guru's pw is already in WIN.INI:
  RealPW = IniRead ("Our Company", "Tech Guru PW", "")
  if PW==RealPW
    MenuChange("SystemUtilitiesCleanupDir", @ENABLE)
    MenuChange("SystemUtilitiesEditBatFiles",@ENABLE)
    MenuChange("SystemUtilitiesEditWinIni", @ENABLE)
    Message ("Access", "You have FULL access")
  else
    MenuChange("SystemUtilitiesCleanupDir", @DISABLE)
    MenuChange("SystemUtilitiesEditBatFiles",@DISABLE)
    MenuChange("SystemUtilitiesEditWinIni", @DISABLE)
    Message ("Access", "You have LIMITED access")
  endif

&System Utilities ;name = "SystemUtilities"
&Cleanup Dir      ;name = "SystemUtilitiesCleanupDir"
...
&Edit BAT Files...;name = "SystemUtilitiesEditBatFiles"
...
&Edit WIN.INI     ;name = "SystemUtilitiesEditWinIni"
...

```

Windows Interface Language Reference

Function List

[By type](#)

[Alphabetically](#)

[Using WIL](#)

[Step by step guide to learning WIL](#)

[WIL Language Elements](#)

[Constants](#)

[Identifiers](#)

[Variables](#)

[Lists](#)

[Keywords](#)

[Operators](#)

[Precedence and Evaluation Order](#)

[Comments](#)

[Statements](#)

[Substitution](#)

[Parameters](#)

[Error Handling](#)

[Error Messages](#)

[Minor Errors](#)

[Moderate Errors](#)

[Fatal Errors](#)

[Script Template](#)

[Notational Conventions](#)

[Notes](#)

Windows Interface Language (WIL) is an easy-to-use yet very powerful general-purpose programming language with over 500 functions for file management, sending keystrokes, disk drive management, directory management, binary file access, multimedia support, DDE support, clipboard handling, system control, program management, string handling, displaying information, user prompting, window management, floating point & integer arithmetic, execution control and more.

What would take pages of code in other languages is often taken care of by a single WIL function.

WIL scripts are written in a plain text file, which can be created by Notepad or most word processors. (Of course, we recommend our own WinEdit, which has many features designed expressly for programmers, including a full-featured implementation of WIL itself.)

These text files can take one of two forms, depending on your particular implementation of WIL: [batch files](#) or [menu files](#). Either way, a program written in Windows Interface Language (WIL) consists of a series of statements. WIL statements are constructed from [constants](#), [variables](#), [operators](#), [functions](#), commands, and [comments](#).

In addition, WIL scripts can take advantage of advanced techniques such as [substitution](#), passing [parameters](#), and [error handling](#).

Notational Conventions

Throughout this documentation, 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.

`system`

Used for items in menus and dialogs, as they appear to the user.

`Small fixed-width`

Used for WIL sample code.

Italics

Used for emphasis, and to liven up the documentation just a bit.

- Reference

- Step by step guide to learning WIL

Notes

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

If you've been having trouble with your WIL scripts, or have a question that's not answered elsewhere, there's a pretty good chance we've answered it here.

[Recovering from Cancel](#)

[WIL Script Template](#)

[Terminating WIL processing](#)

[Carriage Return Line Feed](#)

[Extension Associations and the Run functions](#)

[Debug](#)

[Internal Control Functions](#)

[Partial Window Names](#)

[System.ini and its device= lines](#)

[Dividing Floating PointNumbers](#)

[File Delimiters](#)

[Sounds](#)

- [Reference](#)
- [Using WIL](#)
- [Technical Support](#)
- [Step by step guide to learning WIL](#)

Recovering from Cancel

If the user presses the **Cancel** button (in most 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**.

See the Basic Script Template below for an example of trapping more than one **:CANCEL** label.

- [Notes](#)
- [Reference](#)
- [Step by step guide to learning WIL](#)

Script Template

It is highly recommended that you use a standard template when creating WinBatch scripts. Here is a suggested template:

```
; TITLE :  
; DATE :  
; AUTHOR :  
; VERSION :  
; UPDATED :  
; PURPOSE :  
  
ONCANCEL="EXIT"  
  
:  
: CANCEL  
%ONCANCEL%  
EXIT
```

Extenders: Networking, Novell, TCP/IP, Microsoft

WIL language extenders enhance Microsoft, Novell and TCP/IP networking. They are among the most useful parts of the WIL language. Network connectivity, special arithmetic operations, and other capabilities can be added to WIL with extenders. Other extenders are often available on the web site : <http://www.windowware.com>. They may be downloaded from there. The most widely used extenders are included on the distribution diskettes.

Software developers who want to create custom extensions to the WIL language may purchase a development kit called WIL Extender SDK. The actual extender must be written as a DLL using C or C++.

Each extender has a help file that explains how to use the functions in it. The example here comes from the Novell extender and changes a password under Netware 4.x:

Syntax:

```
n4ChgPassword(server-name, user-name, old password, new password)
```

Parameters:

(s) server-name	name of a network file server or empty string.
(s) user-name	name of the current user.
(s) old password	the old password.
(s) new password	a new password.

Returns:

(i) @TRUE if successful.

When changing passwords the "Old-password" must be specified for Directory Services connections.

For bindery connections, if you have supervisor or equivalent rights, a blank string ("") can be specified for "old-password".

Passwords ARE case-sensitive for Directory Services connections, but NOT for bindery connections.

Example:

```
AddExtender("wnw4x32i.dll")
;Assuming user is attached to and logged into
server
N4ChgPassword("\\CHICAGO", "joe", "spam",
"spasm")
```

Changing passwords is easy with WinBatch and its Netware extender. There are additional functions for Netware 4.x functionality. Particularly important is the capability, only for 32 bit network clients, to log on and off of the network from within Windows. Similar connectivity enhancements work for Microsoft and other networks.

- [Notes](#)
- [Reference](#)
- [Step by step guide to learning WIL](#)

Network connectivity extensions are used to manage LANs and WANs. They are also used to test LAN operation at times when the network is not heavily used.

Terminating WIL processing

A currently-executing WIL program can be terminated immediately by pressing the **<CtrlBreak>** key combination. You may need to hold it a second or two. **IntControl (12, p1, p2, 0, 0)** can be used to suppress the ability of the user to terminate the batch file. One would suggest the batch file is completely debugged before doing this.

- [Notes](#)
- [Reference](#)
- [Step by step guide to learning WIL](#)

Carriage Return Line Feed

A commonly asked question is, "How do I get my long lines to wrap to the next line?". One way, besides using the built in `@crlf` and `@tab` string constants is to use the functions **Num2Char** or **StrCat** to accomplish this.

Example:

```
cr=Num2Char(13); 13 is a carriage-return
lf=Num2Char(10); 10 is a line feed
Message("", "This is line one %cr% %lf% This is line two")
Or..
cr=Num2Char(13)
lf=Num2Char(10)
crlf=StrCat(cr, lf)
Message("", "This is line one %crlf% This is line two")
```

Note: `@crlf` and `@tab` are explained in more detail in the WIL Tutorial section under the heading [Nicer Messages](#).

- [Notes](#)
- [Reference](#)
- [Step by step guide to learning WIL](#)

Extension Associations and the Run Functions

The [Run](#) function (and most of the related members of the **Run...** family of functions) allow you to run a data file if it is associated with a program.

They do not support running programs with unknown extensions.

- [Notes](#)
- [Reference](#)
- [Step by step guide to learning WIL](#)

Debug

WIL has a handy debug utility which comes with the WIL Interpreter. When **Debug** is initialized, a dialog box which controls the execution of each statement is displayed. **Debug** works line by line through the script, displaying the current statement, its value and the following statement. The script will also be executed in conjunction with the display of statements. Initialize **Debug** by adding **Debug(1)** or **Debug(@ON)** to a specific point in your script.

Note: For specific instructions see [Debug](#) in the **WIL Function Reference**.

- [Notes](#)
- [Reference](#)
- [Step by step guide to learning WIL](#)

Internal Control Functions

WinBatch has several Internal Control functions, **IntControl**, which permit numerous internal operations. If you are having trouble finding a specific command, you may find a solution here. For example, **IntControl** can perform a warm boot, restart windows and control whether a file list box has to return a file name. Check out the versatility of [IntControl](#) in the **WIL Function Reference**.

- [Notes](#)
- [Reference](#)
- [Step by step guide to learning WIL](#)

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 (~).

A tilde (~) used as the first character of the window name will match any window containing the specified string anywhere in its title. For example, **WinShow("~Notepad")** will match a window title of "(Untitled) - Notepad" and a window title of "My Notepad Application", as well as a window title of "Notepad - (Untitled)".

A tilde (~) used as the last character of the window name indicates that the name must match the window title through to the end of the title. For example, **WinShow("Note~")** would only match a window whose title was "Note"; it would **not** match "Notepad". Furthermore, **WinShow("~Notepad~")** will match a window title of "Notepad" and a window title of "(Untitled) -Notepad", but will **not** match a window title of "Notepad - (Untitled)".

When using partial windownames as parameters, you can specify the full name if you wish, but in most circumstances, it isn't necessary. Remember that the case (upper or lower) of the title is significant. If the case is not correct, a match will not be made.

- [Notes](#)

- [Reference](#)

- [Step by step guide to learning WIL](#)

System.ini and its device= lines

The "device=" lines in the System.ini cannot be edited using the normal **IniWritePvt** function. See [BinaryPokeStr](#) for a complete example of how to write "device=" lines into the System.ini.

- [Notes](#)
- [Reference](#)
- [Step by step guide to learning WIL](#)

Dividing Floating Point Numbers

This example might not work exactly how you think it will. If you take two integers, for example, 32 and 37, and divide 32 by 37, you will not necessarily get a floating point answer. This integer divide will result in an answer of 0. Add 0.0 to one of the numbers to get a true floating point answer.

- [Notes](#)
- [Reference](#)
- [Step by step guide to learning WIL](#)

Example:

```
;;Problem.wbt
a1= "An unexpected problem can occur when dividing numbers."
a2= "The problem is in deciding between an integer divide "
a3= "(where the remainder, if any, is discarded) and a floating "
a4= "point divide (where a floating point number is returned). "
a5= ""
a6= ""
a7= "Let's assume a test. There are 42 questions."
a8= "A student gets 37 of them correct,"
a9= "what is the student's score."
a10= " "
a11= "iQuestions = 42"
a12= "iCorrect = 37"
a13= "Score = iCorrect / iQuestions"

iQuestions = 42
iCorrect = 37
Score = iCorrect / iQuestions

a14= " "
a15= "The unexpected result is that the score is %Score%"
a16= "Reasonable problem? The trap is that WIL will perform an"
a17= "integer divide and return the unexpected answer of Zero."
a18= " "
a19= "To dig your code out of this trap, simply use floating point"
a20= "numbers when you want a floating point answer."
a21= " "
a22= "fQuestions = 42.0"
a23= "fCorrect = 37.0"

fQuestions = 42.0
fCorrect = 37.0
Score = fCorrect / fQuestions

a24= "Score = fCorrect / fQuestions"
a25= "The correct score is %Score%"
a26= " "
```

```
a27= "Or make the answer look nicer by using the Decimals function"
a28= "and a little formatting."
a29= ""
a30= "Decimals(0)
a31= "Score=Score*100"
Decimals(0)
Score=Score*100
a32= ""
a33= "The correct score is %Score%%%" ;Use 2 %'s to a single %

text=""
for i=1 to 15
    text=strcat(text,a%i%,@crlf)

next

text2=""
for i=16 to 33
    text2=strcat(text2,a%i%,@crlf)
next

Message("Integer Divide Problem",text)
Message("Floating point solution",text2)
```

File Delimiters

In order to support long file names in Windows NT and Windows 95/98, which can contain embedded spaces, we have changed the default file delimiter, used to delimit lists of files and directories, to a TAB in the 32-bit version of WIL. In the 16-bit version of WIL, the default delimiter has not changed, and remains a space.

This is the "default" file delimiter. We have added the ability to change the file delimiter to a character of your own choosing, using the new **IntControl (29,p1,0,0,0)**.

The most important functions affected by this change are:

DirItemize
DiskScan
FileItemize

which now return lists delimited by the current file delimiter character.

The following functions, which take file or directory lists as input parameters, now expect the lists to be delimited by the current file delimiter character. However, they now also accept lists delimited with a TAB or a vertical bar ("|", which may be easier to code in a WIL script):

DirItemize	FileDelete
DirRemove	FileItemize
DiskFree	FileMove
FileAppend	FileRename
FileAttrSet	FileSize
FileCopy	FileTimeSet
	FileTimeTouch

DiskFree will continue to accept space-delimited lists as input.

- [Notes](#)
- [Reference](#)
- [Step by step guide to learning WIL](#)

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, (in the 16-bit WIL Interpreter) these sounds are enabled. They are, however, disabled by default in the 32-bit WIL Interpreter.

If you want sounds to be off by default, enter the line:

Sounds=0

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. One of our developers is particularly enamored with the "BEAMUP.WAV" file available on various on-line services.

- [Notes](#)
- [Reference](#)
- [Step by step guide to learning WIL](#)

Standard WIL functions by type

Windows Interface Language (WIL) is an easy-to-use yet very powerful general-purpose programming language with over 500 functions for file management, sending keystrokes, disk drive management, directory management, binary file access, multimedia support, DDE support, clipboard handling, system control, program management, string handling, displaying information, user prompting, window management, floating point & integer arithmetic, execution control and more. Many operations that require pages of code in other programming languages can be accomplished with a single WIL function call.

[Arithmetic Functions](#)

[Binary Functions](#)

[Clipboard Handling](#)

[DDE Functions](#)

[Directory Management](#)

[Disk Drive Management](#)

[Displaying Information](#)

[DLLCall](#)

[File Management](#)

[Important Functions](#)

[Inputting Information](#)

[InterProgram Communication](#)

[Menu Management](#)

[Miscellaneous Functions](#)

[Multimedia Functions](#)

[Network Functions](#)

[OLE2.0](#)

[Process Control](#)

[Program Management](#)

[Registration Functions](#)

[String Handling](#)

[System Information](#)

[Time Functions](#)

[Window Management](#)

- [Functions listed alphabetically](#)

- [Reference](#)

- [Step by step guide to learning WIL](#)

- [Notes](#)

Arithmetic Functions

Function list

By type

Alphabetically

Abs(integer)

Returns the absolute value of a number.

Acos(fp_num)

Calculates the arccosine.

Asin(fp_num)

Calculates the arcsine.

Atan(fp_num)

Calculates the arc tangent.

Average(number [,number...])

Returns the average of a list of numbers.

Ceiling(fp_num)

Calculates the ceiling of a value.

Char2Num(string)

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

Cos(fp_num)

Calculates the cosine.

Cosh(fp_num)

Calculates the hyperbolic cosine.

Decimals(#digits)

Sets the number of decimal points used with floating point numbers.

Exp(fp_num)

Calculates the exponential.

Fabs(fp_num)

Calculates the absolute value of a floating-point argument.

Floor(fp_num)

Calculates the floor of a value.

Int(string/fp_num)

Converts a floating point number or a string to an integer.

IsFloat(value)

Tests whether a number can be converted to a floating point number.

IsInt(string)

Tests whether a number can be converted into a valid integer.

IsNumber(value)

Determines if a string represents a valid number.

Log10(fp_num)

Calculates the base-10 logarithm.

Loge(fp_num)

Calculates the natural logarithm.

Max(number [,number...])

Determines the highest number in a list.

Min(number [,number...])

Determines the lowest number in a list.

Num2Char(integer)

Converts a number to its character equivalent.

Random(integer)

Generates a positive random number.

Sin(fp_num)

Calculates the sine.

Sinh(fp_num)

Calculates the hyperbolic sine.

Sqrt(fp_num)

Calculates the square root.

Tan(fp_num)

Calculates the tangent.

Tanh(fp_num)

Calculates the hyperbolic tangent.

TimeAdd(YmdHms, YmdHms)

Adds two YmdHms variables.

TimeDiff(time-value-1, time-value-2)

Returns the difference between two points in time.

TimeDiffDays(Ymd[Hms], Ymd[Hms])

Returns the difference in days between the two dates.

TimeDiffSecs(YmdHms, YmdHms)

Returns the time difference in seconds between the two date times.

TimeJulianDay(Ymd[Hms])

Returns the Julian day given a date/time.

TimeJulToYmd(julian-date)

Returns a datetime given a Julian day.

TimeSubtract(datetime, datetime difference)

Subtracts one YmdHms variable from another.

Binary Functions

WIL contains a number of functions designed to allow direct access to areas - buffers - of computer memory. By managing and working with these buffers using the assorted Binary functions provided, you can implement a number of operations that otherwise would be tedious and time consuming.

For more detail see [Binary Operations](#).

Function list

[By type](#)

[Alphabetically](#)

BinaryAlloc(bufsize)

Allocates a memory buffer of the desired size.

BinaryAnd(target-handle,target-offset,source-handle,source-offset,count)

Performs a bitwise AND on portions of two binary buffers.

BinaryClipGet(handle, format)

Reads the contents of the Windows clipboard into a binary buffer.

BinaryClipPut(handle, format)

Writes a binary buffer to the Windows clipboard.

BinaryCompare(handle1, offset1, handle2, offset2, count)

Compares portions of two binary buffers.

BinaryConvert(handle, source-type, target-type, code-page, flags)

Converts a binary buffer.

BinaryCopy(handle targ, offset targ, handle src, offset src, bytecount)

Copies bytes of data from one binary buffer to another.

BinaryEodGet(handle)

Returns the offset of the free byte just after the last byte of stored data.

BinaryEodSet(handle, offset)

Sets the EOD value of a buffer.

BinaryFree(handle)

Frees a buffer previously allocated with BinaryAlloc.

BinaryHashRec(handle, rectx, key offset, key size, key value)

Returns a pointer to a record in a binary buffer.

BinaryIncr(handle, offset)

Peeks at a value, increments the value by 1 then pokes the value back into the buffer.

BinaryIndex(handle, offset, search string, direction)

Searches a buffer for a string.

BinaryIndexEx(handle, offset, string, direction, match-case)

Searches a buffer for a string. This function is very similar to BinaryIndex and BinaryIndexNc, except this function will return -1, if the specified string was not found.

BinaryIndexNC(handle, offset, string, direction)

Searches a buffer for a string. (case insensitive)

BinaryOleType(handle, type,reserved- 1, reserved-2, reserved-3)

Specifies how a binary buffer will be used by OLE functions.

BinaryOr(target-handle, target-offset, source-handle, source-offset, count)

Performs a bitwise OR on portions of two binary buffers.

BinaryPeek(handle, offset)

Returns the value of a byte from a binary buffer. Peeks an 8 bit value.

BinaryPeek2(handle, offset)

Returns the value of a byte from a binary buffer. Peeks a 16 bit value.

BinaryPeek4(handle, offset)

Returns the value of a byte from a binary buffer. Peeks a 32 bit value.

BinaryPeekFlt(handle, offset)

Returns a value from a binary buffer. Peeks a 64 bit floating point number.

BinaryPeekStr(handle, offset, maxsize)

Extracts a string from a binary buffer.

BinaryPoke(handle, offset, value)

Pokes a new value into a binary buffer at offset.
Pokes an 8 bit value.

BinaryPoke2(handle, offset, value)

Pokes a new value into a binary buffer at offset.
Pokes an 16 bit value.

BinaryPoke4(handle, offset, value)

Pokes a new value into a binary buffer at offset.
Pokes an 32 bit value.

BinaryPokeFlt(handle, offset, value)

Pokes a new value into a binary buffer at offset.
Pokes a 64 bit floating point number.

BinaryPokeStr(handle, offset, string)

Writes a string into a binary buffer.

BinaryRead(handle, filename)

Reads a file into a binary buffer.

BinaryReadEx(handle, binary-offset, filename, file-offset, count)

Reads a portion of a file into a binary buffer.

BinarySort(handle, reccount, key offset, key size, flags)

Sorts records in a binary buffer.

BinaryStrCnt(handle, start-offset, end-offset, string)

Counts the occurrences of a string in some or all of a binary buffer.

BinaryTagExtr(tag-struct, flags)

Returns the text between the last-returned pair of binary tags.

BinaryTagFind(tag-struct)

Finds the next binary tag.

BinaryTagInit(buffer, start-tag, end-tag)

Initializes a binary tag operation.

BinaryTagRepl(tag-struct, new-string)

Replaces a binary tag with text.

BinaryWrite(handle, filename)

Writes a binary buffer to a file.

BinaryWriteEx(handle, binary-offset, filename, file-offset, count)

Writes a portion of a binary buffer to a file.

BinaryXor(target-handle,target-offset,source-handle,source-offset,count)

Performs a bitwise XOR (exclusive OR) on portions of two binary buffers.

Clipboard Handling

Function list

By type

Alphabetically

BinaryClipGet(handle, format)

Reads the contents of the Windows clipboard into a binary buffer.

BinaryClipPut(handle, format)

Writes a binary buffer to the Windows clipboard.

ClipAppend(string)

Appends a string to the end of the Clipboard.

ClipGet()

Returns the Clipboard contents into a string.

ClipGetEx(format)

Returns the contents of the Windows clipboard in a specified format.

ClipPut(string)

Replaces the Clipboard contents with a string.

Snapshot(request#)

Takes a snapshot of the screen and pastes it to the clipboard.

DDE Functions

Function list

By type

Alphabetically

DDEExecute(channel, [commandstring])

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 in seconds)

Sets the DDE time-out value.

Directory Management

Function list

By type

Alphabetically

AskDirectory(prompt, browse-root, start-dir, confirm-prompt, flags)

Displays a directory browse dialog box, and returns the selected directory name.

DirAttrGet([d:]path)

Gets directory attributes.

DirAttrSet(dir-list, settings)

Sets directory attributes.

DirChange([d:]path)

Changes the current directory.

DirExist(pathname)

Determines if a directory exists.

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(dir-list)

Removes an existing empty directory.

DirRename([d:]oldpath, [d:]newpath)

Renames a directory.

DirSize(dir-name, flags)

This function returns the total size of a directory, including all files in the directory, and all files in all subdirectories under the directory.

DirWindows(request#)

Returns the name of the Windows or Windows System directory.

ShortcutDir(name)

Returns the name of the directory where shortcuts of type "name" are stored.

Disk Drive Management

Function list

By type

Alphabetically

DiskExist(drive letter)

Tests for the existence of a drive.

DiskFree(drive-list)

Finds the total available (unused) disk space on a group of drives.

DiskInfo (request#)

Returns information on the specified disk drive.

DiskScan(request#)

Returns a list of drives.

DiskSize (request#)

Finds the total size of the specified drive(s).

DiskVollInfo(root-path, request#)

Returns information on a file system volume.

LogDisk(drive letter)

Logs (activates) a disk drive.

NetInfo(request code)

Determines network(s) installed.

Displaying Information

Function list

By type

Alphabetically

About()

Displays the About message box.

AskDirectory(prompt, browse-root, start-dir, confirm-prompt, flags)

Displays a directory browse dialog box, and returns the selected directory name.

AskFileName(title, directory, filetypes, default filename, flag)

Returns the filename as selected by a FileOpen dialog box.

AskItemlist(title, list, delimiter, sort mode, select mode)

Allows the user to choose an item from a list box initialized with a list variable.

Debug(mode)

Turns the Debug mode on or off.

DebugData(string, string)

Writes data via the Windows OutputDebugString function to the default destination.

DebugTrace(mode, filename)

Creates and writes a line by line debugging file of each statement executed in a script.

Dialog(dialog-name)

Displays a user-defined dialog box.

Display(seconds, title, text)

Momentarily displays a string.

Message(title, text)

Displays text in a message box.

Pause(title, prompt)

Displays text in a message box.

Sounds(request#)

Turns sounds on or off.

Terminate(condition, title, text)

Conditionally ends a WIL program.

DLLCall

The DllCall functions are unlike all other WIL functions. They are designed to allow sophisticated users to either write their own extensions to the WIL language (using the Windows SDK), to call third party DLLs, or to access the Windows APIs directly.

Function list

By type

Alphabetically

DllCall(**dllfilename/dllhandle**, **returntype:entrypoint** [,**paramtype:param...**])

Calls an external DLL.

DllFree(**dllhandle**)

Frees a DLL that was loaded via the DllLoad function.

DllHinst(**partial-winname**)

Obtains an application instance handle for use in DllCall's when required.

DllHwnd(**partial-winname**)

Obtains a window handle for use in DllCall's when required.

DllLoad(**dllname**)

Loads a DLL for later use via the DllCall function.

File Management

Function list

By type

Alphabetically

AskDirectory(prompt, browse-root, start-dir, confirm-prompt, flags)

Displays a directory browse dialog box, and returns the selected directory name.

AskFileName(title, directory, filetypes, default filename, flag)

Returns the filename as selected by a FileOpen dialog box.

AskFileText(title, filename, sort mode, select mode)

Allows the user to choose an item from a list box initialized with data from a file.

BinaryRead(handle, filename)

Reads a file into a binary buffer.

BinaryReadEx(handle, binary-offset, filename, file-offset, count)

Reads a portion of a file into a binary buffer.

BinaryWrite(handle, filename)

Writes a binary buffer to a file.

BinaryWriteEx(handle, binary-offset, filename, file-offset, count)

Writes a portion of a binary buffer to a file.

ExeTypeInfo(EXEname)

Returns an integer describing the type of EXE file specified.

FileAppend(source-list, destination)

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.

FileCompare(filename1, filename2)

Compares two files.

FileCopy(source-list, destination, warning)

Copies files.

FileCopyAttr(source-list, destination, warning , attributes)

Copies files, and sets file attributes.

FileDelete(file-list)

Deletes files.

FileExist(filename)

Test for the existence of files.

FileExtension(filename)

Returns the extension of a file.

FileFullName(partial filename)

Returns a file name with drive and path information.

FileItemize(file-list)

Builds a list of files.

FileLocate(filename)

Finds a file within the current DOS path.

FileMapName(filename, mapping-data)

Transforms a filename with a file wild-card mask and returns a new filename.

FileMove(source-list, destination, warning)

Moves files to another set of path names.

FileMoveAttr(source-list, destination, warning)

Moves files, and sets file attributes.

FileNameLong {*32}(filename)

Returns the long version of a filename.

FileNameShort {*32}(filename)

Returns the short (i.e., 8.3) version of a filename.

FileOpen(filename, mode)

Opens a STANDARD ASCII (only) file for reading or writing.

FilePath(filename)

Returns path of a file.

FileRead(filehandle)

Reads data from a file.

FileRename(source-list, destination)

Renames files to another set of names.

FileRoot(filename)

Returns root of a file.

FileSize(file-list)

Adds up the total size of a set of files.

FileTimeCode(filename)

Returns a machine readable/computable code for a file time.

FileTimeGet(filename)

Returns file date and time.

FileTimeGetEx(filename, time-field)

Gets extended time information for a file or directory.

FileTimeSet(list, YmdHms)

Sets the date and time of one or more files.

FileTimeSetEx(file-list, YmdHms, time-field)

Sets extended time information for one or more files.

FileTimeTouch(file-list)

Sets file(s) to current time.

FileVerInfo(filename, language-key, resource-string)

Returns a version resource string from a binary file.

FileWrite(filehandle, output-data)

Writes data to a file.

FileYmdHms(filename)

Returns a file time in the YmdHms date/time format.

IconReplace(filename, icon filename)

Replaces an existing icon with a new icon.

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 file.

IniItemizePvt(section, filename)

List 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, data)

Writes a string to the WIN.INI file.

IniWritePvt(section, keyname, data, filename)

Writes a string to a private INI file.

InstallFile(filename, targname, default-targdir, delete-old, flags)

Installs a file.

Print(data file, directory, display mode, waitflag)

Instructs an application associated to a file to print the file on the default printer.

ShortcutDir (name) {*Explorer}

Returns the name of the directory where shortcuts of type "name" are stored. .

ShortcutEdit (link-name, target, params, start-dir, show-mode) {*Explorer}

Modifies the specified shortcut file.

ShortcutExtra(link-name, description, hotkey, icon-file, icon-index) {*Explorer}

Sets additional information for the specified shortcut file.

ShortcutInfo(link-name) {*Explorer}

Returns information on the specified shortcut file.

ShortcutMake(link-name, target, params, start-dir, show-mode) {*Explorer}

Creates a shortcut for the specified filename or directory.

Important Functions

Function list

By type

Alphabetically

AskDirectory(prompt, browse-root, start-dir, confirm-prompt, flags)

Displays a directory browse dialog box, and returns the selected directory name.

AskFileName(title, directory, filetypes, default filename, flag)

Returns the filename as selected by a FileOpen dialog box.

AskFileText(title, filename, sort mode, select mode)

Allows the user to choose an item from a list box initialized with data from a file.

AskItemList(title, list, delimiter, sort mode, select mode)

Allows the user to choose an item from a list box initialized with a list variable.

AskLine(title, prompt, default answer)

Lets the user enter a line of information.

AskYesNo(title, question)

Lets the user choose from Yes, No, or Cancel.

Debug(mode)

Turns the Debug mode on or off.

Dialog(dialog-name)

Displays a user-defined dialog box.

DirChange([d:]path)

Changes the current directory.

Display(seconds, title, text)

Momentarily displays a string.

ErrorMode(mode)

Specifies how to handle errors.

FileCopy(source-list, filename/mask, mode)

Copies files.

FileCopyAttr(source-list, destination, warning , attributes)

Copies files, and sets file attributes.

FileDelete(filelist)

Deletes files.

FileExist(filename)

Tests for the existence of files.

For varname = initial value to final value [by increment]

Controls the looping of a block of code base in an incrementing index.

GoSub

Transfers control of WIL processing while saving location of the next statement.

If ... Else ... Endifexpression

Conditionally performs a function.

IniReadPvt(section, keyname, default, filename)

Reads a string from a private INI file.

ItemCount(list, delimiter)

Returns the number of items in a list.

ItemExtract(index, list, delimiter)

Returns the selected item from a list.

Message(title, text)

Displays text in a message box.

NetInfo(request code)

Determines network(s) installed.

Pause(title, text)

Displays Text in a message box.

RegApp(program-name, path)

Creates registry entries for a program under "App Paths".

RegExistKey(handle, subkey-string)

Checks whether a registry key exists.

RegQueryValue(keyhandle, sub-key string)

Returns data item string at sub-key position.

RegSetValue(keyhandle, sub-key string, value)

Sets the value of a data item in the registration database.

Run(program-name, params)

Runs a program as a normal window.

RunShell(program-name, params, directory, displaymode, waitflag)

An advanced form of the Run function that even allows the specification of a working directory, along with the window view mode and whether or not to wait for completion of the run program in a single function.

RunWait(program-name, params)

Runs a program as a normal window, and waits for it to close.

SendKeysTo(parent-windowname, sendkey string)

Sends keystrokes to a "windowname".

SendMenusTo(partial-parent-windowname, menuname)

Activates a window and sends a specified menu option.

ShellExecute (program-name, params, directory, display mode, operation)

Runs a program via the Windows ShellExecute command

StrCat(string [,string])

Concatenates strings together.

StrIndex(string, sub-string, start, direction)

Searches a string for a sub-string.

StrLen(string)

Returns the length of a string.

StrReplace(string, old, new)

Replaces all occurrences of a sub-string with another.

StrSub(string, startpos, length)

Returns a sub-string from within a string.

Switch varname

Allows selection among multiple blocks of statements.

TimeDelay(seconds)

Pauses execution for a specified amount of time.

TimeWait(YmdHms)

Pauses execution and waits for the date/time to pass.

While expression

Conditionally and/or repeatedly executes a series of statements.

WinActivate(partial-winname)

Activates a previously running parent window.

WinClose(partial-winname)

Closes an application window.

WinExist(partial-winname)

Tells if a window exists.

WinMetrics(request#)

Returns Windows system information.

WinWaitChild(partial parent winname, partial child winname, timeout)

Waits for a child window to exist.

WinWaitExist(partial-winname, timeout)

Waits for a window to exist.

Inputting Information

Function list

By type

Alphabetically

AskDirectory(prompt, browse-root, start-dir, confirm-prompt, flags)

Displays a directory browse dialog box, and returns the selected directory name.

AskFileName(title, directory, filetypes, default filename, flag)

Returns the filename as selected by a FileOpen dialog box.

AskFileText(title, filename, sort mode, select mode)

Allows the user to choose an item from a list box initialized with data from a file.

AskItemList(title, list, delimiter, sort mode, select mode)

Allows the user to choose an item from a list box initialized with a list variable.

AskLine(title, prompt, default answer)

Lets the user enter a line of information.

AskPassword(title, prompt)

Prompts the user for a password.

AskYesNo(title, question)

Lets the user choose from Yes, No, or Cancel.

ButtonNames(Ok-name, Cancel-name)

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

Dialog(dialog-name)

Displays a user-defined dialog box.

Display(seconds, title, text)

Momentarily displays a string.

IsKeyDown(keycode)

Determines if the Shift key or the Ctrl key is currently down.

KeyToggleGet(keycode)

Returns the status of a toggle key.

KeyToggleSet(@key, value)

Sets the state of a toggle key and returns the previous value.

Pause(title, prompt)

Displays Text in a message box.

InterProgram Communication

Function list

By type

Alphabetically

AddExtender(dllfilename)

Installs a WIL extender DLL.

AppExist(program-name)

Tells if an application is running.

AppWaitClose(program-name)

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

ClipAppend(string)

Appends a string to the end of the Clipboard.

ClipGet()

Returns the Clipboard contents into a string.

ClipGetEx(format)

Returns the contents of the Windows clipboard in a specified format.

ClipPut(string)

Replaces the Clipboard contents with a string.

DDEExecute(channel, [commandstring])

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 in seconds)

Sets the DDE time-out value.

EnvironSet(env-varname, newvalue)

Changes LOCAL Environment variables.

IntControl(request#, p1, p2, p3, p4)

Special function which permits an internal operation.

MouseClick(click-type, modifiers)

Clicks mouse button(s).

MouseClickBtn(parent-windowname, child-windowname, button-text)

Clicks on the specified button control.

MouseMove(X, Y, parent-windowname, child-windowname)

Moves the mouse to the specified X-Y coordinates.

MsgTextGet(msgboxtitle)

Returns the contents of a Windows message box.

ObjectAccess(app.objname)

Opens or creates an OLE 2.0 Automation object.

ObjectClose(objecthandle)

Closes OLE 2.0 automation object.

ObjectOpen(objectname)

Opens or creates an OLE 2.0 automation object.

Print(data file, directory, display mode, waitflag)

Instructs an application associated to a file to print the file on the default printer.

SendKey(sendkey string)

Sends keystrokes to the currently active window.

SendKeysChild(partial-parent-windowname, partial-child-windowname, sendkey string)

Sends keystrokes to the active child window.

SendKeysTo(parent-windowname, sendkey string)

Sends keystrokes to a "windowname".

SendMenusTo(partial-parent-windowname, menuname)

Activates a window and sends a specified menu option.

WinActivate(partial-winname)

Makes an application window the active window.

WinActivChild(partial-parent-windowname, partial-child-windowname)

Activates a previously running child window.

WinClose(partial-winname)

Closes an application window.

WinExeName(partial-winname)

Returns the name of the executable file which created a specified window.

WinExist(partial-winname)

Tells if specified window exists.

WinExistChild(partial-parent-windowname, partial-child-windowname)

Tells if a specified child window exists.

WinGetActive()

Gets the title of the active window.

WinIdGet(partial-winname)

Returns a unique "Window ID" (pseudo-handle) for the specified window name.

WinIsDOS(partial-winname)

Tells whether or not a particular window is a DOS or console-type window.

WinItemChild(partial-parent-windowname)

Returns a list of all the child windows under this parent.

WinItemize()

Lists all the main windows currently running.

WinItemizeEx(partial-winname, multiple, hidden)

Returns the full name of window(s) matching a partial window name.

WinItemProclD(process-id, flags, return-type)

Returns a list of window ID's for the specified process.

WinItemNameId()

Returns a list of all open windows and their Window ID's.

WinState(partial-winname)

Returns the current state of a window.

Yield

Pauses WIL processing so other applications can process some messages.

Menu Management

These commands can only be implemented in one of the shell or file manager-type applications which use the WIL Interpreter. See [Menu Files](#) for more information.

Function list

[By type](#)

[Alphabetically](#)

CurrentFile()

Returns the filename of the selected item.

CurrFilePath()

Returns the path and full filename of the selected item.

CurrentPath()

Returns path of the selected filename.

IsMenuChecked(menuname)

Determines if a menu item has a check mark next to it.

IsMenuEnabled(menuname)

Determines if a menu item has been enabled.

MenuChange(menuname, flags)

Checks, unchecks, enables, or disables a menu item.

Miscellaneous Functions

Function list

By type

Alphabetically

About()

Displays the About message box.

AddExtender(dllfilename)

Installs a WIL extender Dll.

Beep

Beeps at the user.

DllCall(dllfilename/dllhandle, returntype:entrypoint [,paramtype:param..]

Calls an external Dll.

DllLoad(dllname)

Loads a Dll for later use via the DllCall function.

Drop(varname [,varname...])

Deletes variables to recover their memory.

DropWild(variable/wildcard)

Removes variables from memory.

ErrorMode(mode)

Sets what happens in the event of an error.

Exclusive(mode)

Controls whether or not other Windows programs will get any time to execute.

Execute statement

Directly executes a WIL statement.

ExeTypeInfo(exefilename)

Returns an integer describing the type of EXE file specified.

IconReplace(filename, iconfilename)

Replaces an existing icon with a new icon.

IntControl(request#, p1, p2, p3, p4)

Special function which permits an internal operation.

IsDefined (variable name)

Determines if a variable name is currently defined.

IsFloat(value)

Tests whether a number can be converted to a floating point number.

IsInt(string)

Tests whether a number can be converted into a valid integer.

IsKeyDown(keycode)

Determines if the Shift key or the Ctrl key is currently down.

IsLicensed()

Tells if the calling application is licensed.

IsNumber(value)

Determines if a string represents a valid number.

KeyToggleGet(@key)

Returns the status of a toggle key.

KeyToggleSet(@key, value)

Sets the state of a toggle key and returns the previous value.

LastError()

Returns the last error encountered.

MouseClicked(click-type, modifiers)

Clicks mouse button(s).

MouseClickedBtn(parent-windowname, child-windowname, button-text)

Clicks on the specified button control.

MouseInfo(request#)

Returns assorted mouse information.

MouseMove(X, Y, parent-windowname, child-windowname)

Moves the mouse to the specified X-Y coordinates.

MsgTextGet(window-name)

Returns the contents of a Windows message box.

NetInfo(request code)

Determines network(s) installed.

ParseData(string)

Parses a passed string.

Sounds(request#)

Turns sounds on or off.

Version()

Returns the version of the parent program currently running.

VersionDLL()

Returns the version of the WIL interpreter currently running.

WallPaper(bmp-filename, tilemode)

Changes the Windows wallpaper.

Multimedia Functions

Function list

By type

Alphabetically

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.

WallPaper(bmp/html-name, tile)

Changes the Windows wallpaper.

Network Functions

The Networking Functions are implemented through the use of WIL extender DLLs, special DLLs designed to extend the built-in function set of the WIL processor. These DLLs typically add functions not provided in the basic WIL set.

Network and Other extenders are documented fully in their own on-line help files.

Function list

By type

Alphabetically

AddExtender(dllfilename)

Installs a WIL extender DLL.

NetInfo(request#)

Determines network(s) installed. All network functionality for WIL is performed via "WIL Extenders", add-on DLLs for WIL, which contain Network commands for assorted networks. Documentation for the various network extenders are found in associated help files.

OLE2.0

Function list

By type

Alphabetically

ObjectAccess(app.objname)

Opens or creates an OLE 2.0 Automation object.

ObjectClose(objecthandle)

Closes OLE 2.0 automation object.

ObjectOpen(objectname)

Opens or creates an OLE 2.0 automation object.

Process Control

Function list

By type

Alphabetically

AddExtender(dllfilename)

Installs a WIL extender DLL.

AppWaitClose(program-name)

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

Break

Used to exit a conditional flow control statement.

Call(wilfilename, parameters)

Calls a WIL batch file as a subroutine.

Continue

Transfers control to the beginning of a For or While loop or to a different case statement.

Debug(mode)

Turns the Debug mode on or off.

DebugData(string, string)

Writes data via the Windows OutputDebugString function to the default destination.

DebugTrace(mode, filename)

Creates and writes a line by line debugging file of each statement executed in a script.

Drop (varname [,varname...])

Deletes variables to recover their memory.

DropWild(variable/wildcard)

Removes variables from memory.

ErrorMode(mode)

Sets what happens in the event of an error.

Execute statement

Directly executes a WIL statement.

Exit

Unconditionally ends a WIL program.

For varname = initial value to final value [by increment]

Controls the looping of a block of code base in an incrementing index.

GoSub label

Transfers control of WIL processing while saving location of the next statement.

Goto label

Changes the flow of control in a WIL program.

If...Else...Endif expression

Conditionally performs a function.

LastError()

Returns the last error encountered.

Return

Returns from a Call to the calling program or from a GoSub :label.

Select varname

Allows selection among multiple blocks of statements.

Switch varname

Allows selection among multiple blocks of statements.

Terminate(expression, title, message)

Conditionally ends a WIL program.

TimeDelay(seconds)

Pauses execution for a specified amount of time.

TimeWait(YmdHms)

Pauses execution and waits for the date/time to pass.

Version()

Returns the version of the parent program currently running.

VersionDLL()

Returns the version of the WIL interpreter currently running.

WaitForKey(key, key, key, key, key)

Waits for a specific key to be pressed.

While expression

Conditionally and/or repeatedly executes a series of statements.

WinResources(request#)

Returns information on available memory and resources.

WinState(partial-winname)

Returns the current state of a window.

WinSysInfo()

Returns system configuration information.

WinVersion(level)

Returns the version of Windows that is currently running.

WinWaitChild(partial parent winname, partial child winname, timeout)

Waits for a child window to exist.

WinWaitClose(partial-winname)

Waits until an application window is closed.

WinWaitExist(partial-winname, timeout)

Waits for a window to exist.

Yields(count)

Provides time for other windows to do processing.

Program Management

Function list

By type

Alphabetically

AppExist(program-name)

Tells if an application is running.

AppWaitClose(program-name)

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

DllCall(dllfilename/dllhandle, returntype:entrypoint [,paramtype:param...])

Calls an external Dll.

DllLoad(dllname)

Loads a Dll for later use via the DllCall function.

EnvironSet(env-varname, newvalue)

Changes LOCAL Environment variables.

ObjectAccess(app.objname)

Opens or creates an OLE 2.0 Automation object.

ObjectClose(objecthandle)

Closes OLE 2.0 automation object.

ObjectOpen(objectname)

Opens or creates an OLE 2.0 automation object.

Print(data file, directory, display mode, waitflag)

Instructs an application associated to a file to print the file on the default printer.

Run(program-name, parameters)

Runs a program as a normal window.

RunEnviron(program name, parameters, displaymode, waitflag)

Launches a program in the current environment as set with the EnvironSet command.

RunHide(program-name, parameters)

Runs a program as 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.

RunShell(program-name, params, directory, displaymode, waitflag)

An advanced form of the **Run** function that even allows the specification of a working directory, along with the window view mode and whether or not to wait for completion of the run program in a single function.

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.

ShellExecute (program-name, params, directory, display mode, operation)

Runs a program via the Windows ShellExecute command

Registration Functions

Function list

By type

Alphabetically

Registration Database Operations

RegApp(program-name, path) {*32}

Creates registry entries for a program under "App Paths".

RegCloseKey(keyhandle)

Closes a key to the registration database.

RegConnect(computer-name, handle)

Connects to a predefined registry handle on another computer.

RegCreateKey(keyhandle, sub-key string)

Returns a handle to a new registration database key.

RegDeleteKey(keyhandle, sub-key string)

Deletes a key and data items associated with the key.

RegDelValue(handle, subkey-string){*32}

Deletes a named value data item for the specified subkey from the registry.

RegEntryType(handle, subkey-string)

Returns the type of data for the specified subkey.

RegExistKey(handle, subkey-string)

Checks whether a registry key exists.

RegExistValue(handle, subkey-string) {*32}

Checks whether a value for a registry key exists.

RegLoadHive(handle, subkey, filename)

Loads a hive into a registry subkey

RegOpenKey(keyhandle, sub-key string)

Returns a handle to an existing registration database key.

RegQueryBin(handle, subkey-string){*32}

Returns binary value at subkey position.

RegQueryDword(handle, subkey-string){*32}

Returns DWORD value at subkey position.

RegQueryEx(handle, subkey-string, delimiter, type)

Retrieves a value from the registry.

RegQueryExpSz(handle, subkey-string){*32}

Retrieves a REG_EXPAND_SZ value from the registry.

RegQueryItem(handle, subkey-string){*32}

Returns a list of named data items for a subkey.

RegQueryKey(handle, index)

Returns sub keys of the specified key.

RegQueryKeys(handle)

Returns a tab-delimited list of subkey names under a specified key.

RegQueryMulSz(handle, subkey-string, delimiter) {*32}

Retrieves a REG_MULTI_SZ value from the registry.

RegQueryValue(keyhandle, sub-key string)

Returns data item string at sub-key position.

RegSetBin(handle, subkey-string, value) {*32}

Sets a binary value in the Registration Database.

RegSetDword(handle, subkey-string, value) {*32}

Sets a DWORD value in the Registration Database.

RegSetEx(handle, subkey-string, value, delimiter, type)

Sets a value in the registry.

RegSetExpSz(handle, subkey-string, value) {*32}

Sets a REG_EXPAND_SZ value in the registry.

RegSetMulSz(handle, subkey-string, value, delimiter){*32}

Sets a REG_MULTI_SZ value in the registry.

RegSetValue(keyhandle, sub-key string, value)

Sets the value of a data item in the registration database.

RegUnloadHive(handle, subkey)

Unloads a hive from the registry.

String Handling

Function list

By type

Alphabetically

BinaryPeekStr(handle, offset, maxsize)

Extracts a string from a binary buffer.

BinaryPokeStr(handle, offset, string)

Writes a string into a binary buffer.

BinaryTagExtr(tag-struct, flags)

Returns the text between the last-returned pair of binary tags.

BinaryTagFind(tag-struct)

Finds the next binary tag.

BinaryTagInit(buffer, start-tag, end-tag)

Initializes a binary tag operation.

BinaryTagRepl(tag-struct, new-string)

Replaces a binary tag with text.

Char2Num(string)

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

ClipAppend(string)

Appends a string to the end of the Clipboard.

ClipGet()

Returns the Clipboard contents into a string.

ClipGetEx(format)

Returns the contents of the Windows clipboard in a specified format.

ClipPut(string)

Replaces the Clipboard contents with a string.

Drop (varname [,varname...])

Deletes variables to recover their memory.

DropWild(variable/wildcard)

Removes variables from memory.

Environment(env-variable)

Gets a DOS environment variable.

FileExtension(filename)

Returns the extension of a file.

FileFullName(partial filename)

Returns a file name with drive and path information.

FileItemize(file-list)

Builds a list of files.

FileLocate(filename)

Finds a file within the current DOS path.

FileMapName(filename, mapping data)

Transforms a filename with a file wild-card mask and returns a new filename.

FilePath(filename)

Returns path of a file.

FileRoot(filename)

Returns root of a file.

FileYmdHms(filename)

Returns a file time in the YmdHms date/time format.

IsFloat(string)

Tests whether a number can be converted to a floating point number.

IsInt(string)

Tests whether a number can be converted into a valid integer.

IsNumber(value)

Determines if a string represents a valid number.

ItemCount(list, delimiter)

Returns the number of items in a list.

ItemExtract(index, 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.

ItemSortNc(list, delimiter)

Sorts a list, without regard to case.

Num2Char(integer)

Converts a number to its character equivalent.

ParseData(string)

Parses a passed string.

StrCat(string [,string])

Concatenates strings together.

StrCharCount(string)

Counts the number of characters in a string.

StrCmp(string1, string2)

Compares two strings.

StrFill(filler, length)

Builds a string from a repeated smaller string.

StrFix(base-string, pad-string, length)

Pads or truncates a string to a fixed length.

StrFixChars(base-string, pad-string, length)

Pads or truncates a string To a fixed length using characters.

StrFixCharsL(base-string, pad-string, length)

Pads or truncates the left side of a string to a fixed length using characters.

StrFixLeft(base-string, pad-string, length)

Pads or truncates the left side of a string to a fixed length using bytes.

StriCmp(string1, string2)

Compares two strings without regard to case.

StrIndex(base-string, sub-string, start, direction)

Locates a string within a larger string.

StrIndexNc(base-string, sub-string, start, direction)

Locates a string within a larger string, ignoring case.

StrIndexWild(string, pattern, start)

Finds wildcarded text within a larger string.

StrLen(string)

Returns the length of a string.

StrLenWild(string, pattern, start)

Returns the length of wildcarded text within a larger string.

StrLower(string)

Converts a string to all lower-case characters.

StrReplace(string, old, new)

Replaces all occurrences of a sub-string with another.

StrScan(string, delimiters, startpos, direction)

Finds an occurrence of one or more delimiter characters in a string.

StrSub(string, startpos, length)

Returns a sub-string from within a string.

StrSubWild(string, pattern, start)

Extracts a substring matching wildcarded text from within a larger string.

StrTrim(string)

Trims leading and trailing blanks from a string.

StrUpper(string)

Converts a string to all upper-case characters.

WinItemize()

Lists all the main windows currently running.

WinItemizeEx(partial-winname, multiple, hidden)

Returns the full name of window(s) matching a partial window name.

WinItemNameId()

Returns a list of all open windows and their Window ID's.

WinItemProcID(process-id, flags, return-type)

Returns a list of window ID's for the specified process.

System Information

Function list

By type

Alphabetically

About()

Displays the About message box.

AddExtender(dllfilename)

Installs a WIL extender Dll.

AppExist(program-name)

Tells if an application is running.

AppWaitClose(program-name)

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

Beep

Beeps at the user.

Debug(mode)

Turns the Debug mode on or off.

DebugData(string, string)

Writes data via the Windows OutputDebugString function to the default destination.

DebugTrace(mode, filename)

Creates and writes a line by line debugging file of each statement executed in a script.

DirHome()

Returns the initial directory path.

DirWindows(request#)

Returns the name of the Windows or Windows System directory.

DiskExist(drive letter)

Tests for the existence of a drive.

DiskFree(drive-list)

Finds the total available (unused) disk space on a group of drives.

DiskInfo(request#)

Returns information on the specified disk drive.

DiskScan(request#)

Returns a list of drives.

DiskSize (request#)

Finds the total size of the specified drive(s).

DiskVollInfo(root-path, request#)

Returns information on a file system volume.

DllCall(dllfilename/dllhandle, returntype:entrypoint [,paramtype:param...])

Calls an external DLL.

DllFree(dllhandle)

Frees a DLL that was loaded via the DllLoad function.

DllHinst(partial-winname)

Obtains an application instance handle for use in DllCall's when required.

DllHwnd(partial-winname)

Obtains a window handle for use in DllCall's when required.

DllLoad(dllname)

Loads a DLL for later use via the DllCall function.

DOSVersion(level)

Returns the version numbers of the current version of DOS.

Drop(varname [,varname...])

Deletes variables to recover their memory.

DropWild(variable/wildcard)

Removes variables from memory.

EndSession()

Ends the current Windows session.

Environment(env-variable)

Gets a DOS environment variable.

EnvironSet(env-varname, newvalue)

Changes LOCAL Environment variables.

Envltemize()

Returns a delimited list of the current environment.

ErrorMode(mode)

Sets what happens in the event of an error.

Exclusive(mode)

Controls whether or not other Windows programs will get any time to execute.

Executestatement

Directly executes a WIL statement.

ExeTypeInfo(exefilename)

Returns an integer describing the type of EXE file specified.

Exit

Unconditionally ends a WIL program.

FileFullName(partial filename)

Returns a file name with drive and path information.

FileMapName(filename, mapping-data)

Transforms a filename with a file wild-card mask and returns a new filename.

FileTimeCode(filename)

Returns a machine readable/computable code for a file time.

FileTimeGet(filename)

Returns file date and time.

FileTimeGetEx(filename, time-field)

Gets extended time information for a file or directory.

FileTimeSet(list, YmdHms)

Sets the date and time of one or more files.

FileTimeSetEx(file-list, YmdHms, time-field)

Sets extended time information for one or more files.

FileTimeTouch(file-list)

Sets file(s) to current time.

FileVerInfo(filename, language-key, resource-string)

Returns a version resource string from a binary file.

GetExactTime()

Returns the current time in hundredths of a second.

GetTickCount()

Returns the number of clock ticks used by Windows since Windows started.

IntControl(request#, p1, p2, p3, p4)

Special function which permits an internal operation.

KeyToggleGet(@key)

Returns the status of a toggle key.

KeyToggleSet(@key, value)

Sets the state of a toggle key and returns the previous value.

LastError()

Returns the last error encountered.

MouseInfo(request#)

Returns assorted mouse information.

MsgTextGet(window-name)

Returns the contents of a Windows message box.

NetInfo(request code)

Determines network(s) installed.

Sounds(request#)

Turns sounds on or off.

Terminate(expression, title, message)

Conditionally ends a WIL program.

Version()

Returns the version of the parent program currently running.

VersionDLL()

Returns the version of the WIL interpreter currently running.

WinExeName(partial-winname)

Returns the name of the executable file which created a specified window.

WinExist(partial-winname)

Tells if a window exists.

WinExistChild(partial-parent-windowname, partial-child-windowname)

Tells if a specified child window exists.

WinGetActive()

Gets the title of the active window.

WinHelp(helpfile, function, keyword)

Calls a Windows help file.

WinIsDOS(partial-winname)

Tells whether or not a particular window is a DOS or console-type window.

WinItemChild(partial-parent-windowname)

Returns a list of all the child windows under this parent.

WinItemize()

Returns a tab-delimited list of all open windows.

WinItemizeEx(partial-winname, multiple, hidden)

Returns the full name of window(s) matching a partial window name.

WinItemNameId()

Returns a list of all open windows and their Window ID's.

WinItemProclD(process-id, flags, return-type)

Returns a list of window ID's for the specified process.

WinMetrics(request#)

Returns Windows system information.

WinName()

Returns the name of the window calling the WIL Interpreter.

WinParmGet(request#)

Returns system information.

WinParmSet(request#, new-value, ini-control)

Sets system information.

WinResources(request#)

Returns information on available memory and resources.

WinState(partial-winname)

Returns the current state of a window.

WinSysInfo() { *32 }

Returns system configuration information.

WinVersion(level)

Returns the version of Windows that is currently running.

Yields(count)

Provides time for other windows to do processing.

Time Functions

Function list

By type

Alphabetically

FileTimeCode(filename)

Returns a machine readable/computable code for a file time.

FileTimeGet(filename)

Returns file date and time.

FileTimeGetEx(filename, time-field)

Gets extended time information for a file or directory.

FileTimeSet(list, ymdhms)

Sets the date and time of one or more files.

FileTimeSetEx(file-list, YmdHms, time-field)

Sets extended time information for one or more files.

FileTimeTouch(file-list)

Sets file(s) to current time.

FileYmdHms(filename)

Returns a file time in the YmdHms date/time format.

GetExactTime()

Returns the current time in hundredths of a second.

GetTickCount()

Returns the number of clock ticks used by Windows since Windows started.

TimeAdd(YmdHms, YmdHms)

Adds two YmdHms variables.

TimeDate()

Provides the current date and time in a readable format.

TimeDayofWeek

Example: Using the WIL language it is possible to grab the day of the week. The following example from TimeJulianDay grabs the current time and day of the week.

TimeDayofYear

Example: Using the WIL language it is possible to grab the day of the year. The following example from TimeJulianDay grabs the current time and day of the year.

TimeDelay(seconds)

Pauses execution for a specified amount of time.

TimeDiff(time-value-1, time-value-2)

Returns the difference between two points in time.

TimeDiffDays(Ymd[Hms], Ymd[Hms])

Returns the difference in days between the two dates.

TimeDiffSecs(YmdHms, YmdHms)

Returns the time difference in seconds between the two date times.

TimeJulianDay(Ymd[Hms])

Returns the julian day given a date/time.

TimeJulToYmd(julian-date)

Returns a datetime given a Julian day.

TimeSubtract(datetime, datetime difference)

Subtracts one YmdHms variable from another.

TimeWait(YmdHms)

Pauses execution and waits for the date/time to pass.

TimeYmdHms()

Returns current date/time in the date/time format.

Yields(count)

Provides time for other windows to do processing.

Window Management

Function list

By type

Alphabetically

DllHwnd(partial-winname)

Obtains a window handle for use in DllCall's when required.

FindWindow(window class name)

Returns a window title matching a specified class name.

WallPaper(bmp-filename, tilemode)

Changes the Windows wallpaper.

WinActivate(partial-winname)

Makes an application window the active window.

WinActivChild(partial-parent-windowname, partial-child-windowname)

Activates a previously running child window.

WinArrange(style)

Arranges all running application windows on the screen.

WinClose(partial-winname)

Closes an application window.

WinExeName(partial-winname)

Returns the name of the executable file which created a specified window.

WinExist(partial-winname)

Tells if a window exists.

WinExistChild(partial-parent-windowname, partial-child-windowname)

Tells if a specified child 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.

WinIdGet(partial-winname)

Returns a unique "Window ID" (pseudo-handle) for the specified window name.

WinIsDOS(partial-winname)

Tells whether or not a particular window is a DOS or console-type window.

WinItemChild(partial-parent-windowname)

Returns a list of all the child windows under this parent.

WinItemize()

Returns a tab-delimited list of all open windows.

WinItemizeEx(partial-winname, multiple, hidden)

Returns the full name of window(s) matching a partial window name.

WinItemNameId()

Returns a list of all open windows and their Window ID's.

WinItemProcID(process-id, flags, return-type)

Returns a list of window ID's for the specified process.

WinName()

Returns the name of the window calling the WIL Interpreter.

WinParmGet(request#)

Returns system information.

WinParmSet(request#, new-value, ini-control)

Sets system information.

WinPlace(x-ulg, y-ulg, x-brc, y-brc, 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 window in its "normal" state.

WinState(partial-winname)

Returns the current state of a window.

WinSysInfo() { *32 }

Returns system configuration information.

WinTitle(old-partial-winname, new-winname)

Changes the title of an application window.

WinWaitChild(partial parent winname, partial child winname, timeout)

Waits for a child window to exist.

WinWaitClose(partial-winname)

Waits until an application window is closed.

WinWaitExist(partial-winname, timeout)

Waits for a window to exist.

WinZoom(partial-winname)

Maximizes an application window to full-screen.

Standard WIL Functions in alphabetical order

A • B

- C
- D
- E
- F
- G
- H
- I
- J
- K
- L
- M
- N
- O
- P
- Q
- R
- S
- T
- U
- V
- W
- X
- Y
- Z

Windows Interface Language (WIL) is an easy-to-use yet very powerful general-purpose programming language with over 500 functions for file management, sending keystrokes, disk drive management, directory management, binary file access, multimedia support, DDE support, clipboard handling, system control, program management, string handling, displaying information, user prompting, window management, floating point & integer arithmetic, execution control and more. Many operations that require pages of code in other programming languages can be accomplished with a single WIL function call.

- Functions listed by type
- Reference
- Using WIL
- Step by step guide to learning WIL
- Notes

A

About()

Displays the About message box.

Abs(**integer**)

Returns the absolute value of a number.

Acos(**fp_num**)

Calculates the arccosine.

AddExtender(dllfilename)

Installs a WIL extender Dll.

AppExist(program-name)

Tells if an application is running.

AppWaitClose(program-name)

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

Asin(fp_num)

Calculates the arcsine.

AskDirectory(prompt, browse-root, start-dir, confirm-prompt, flags)

Displays a directory browse dialog box, and returns the selected directory name.

AskFileName(title, directory, filetypes, default filename, flag)

Returns the filename as selected by a FileOpen dialog box.

AskFileText(title, filename, sort mode, select mode)

Allows the user to choose an item from a list box initialized with data from a file.

AskItemList(title, list, delimiter, sort mode, select mode)

Allows the user to choose an item from a list box initialized with a list variable.

AskLine(title, prompt, default answer)

Lets the user enter a line of information.

AskPassword(title, prompt)

Prompts the user for a password.

AskYesNo(title, question)

Lets the user choose from Yes, No, or Cancel.

Atan(fp_num)

Calculates the arc tangent.

Average(number [,number...])

Returns the average of a list of numbers.

B

Beep

Beeps at the user.

Binary Operations

BinaryAlloc(bufsize)

Allocates a memory buffer of the desired size.

BinaryAnd(target-handle,target-offset,source-handle,source-offset,count)

Performs a bitwise AND on portions of two binary buffers.

BinaryClipGet(handle, format)

Reads the contents of the Windows clipboard into a binary buffer.

BinaryClipPut(handle, format)

Writes a binary buffer to the Windows clipboard.

BinaryCompare(handle1, offset1, handle2, offset2, count)

Compares portions of two binary buffers.

BinaryConvert(handle, source-type, target-type, code-page, flags)

Converts a binary buffer.

BinaryCopy(handle targ, offset targ, handle src, offset src, bytecount)

Copies bytes of data from one binary buffer to another.

BinaryEodGet(handle)

Returns the offset of the free byte just after the last byte of stored data.

BinaryEodSet(handle, offset)

Sets the EOD value of a buffer.

BinaryFree(handle)

Frees a buffer previously allocated with BinaryAlloc.

BinaryHashRec(handle, recsize, key offset, key size, key value)

Returns a pointer to a record in a binary buffer.

BinaryIncr(handle, offset)

Peeks at a 1 byte value, increments the value by 1 then pokes the value back into the buffer.

BinaryIncr2(handle, offset)

Peeks at a 2 byte value, increments the value by 1 then pokes the value back into the buffer.

BinaryIncr4(handle, offset)

Peeks at a 4 byte value, increments the value by 1 then pokes the value back into the buffer.

BinaryIncrFlt(handle, offset)

Peeks at an 8 byte floating point number, increments the value by 1 then pokes the value back into the buffer.

BinaryIndex(handle, offset, search string, direction)

Searches a buffer for a string.

BinaryIndexEx(handle, offset, string, direction, match-case)

Searches a buffer for a string. This function is very similar to BinaryIndex and BinaryIndexNc, except this function will return -1, if the specified string was not found.

BinaryIndexNC(handle, offset, string, direction)

Searches a buffer for a string. (case insensitive)

BinaryOleType(handle, type,reserved- 1, reserved-2, reserved-3)

Specifies how a binary buffer will be used by OLE functions.

BinaryOr(target-handle,target-offset,source-handle,source-offset,count)

Performs a bitwise OR on portions of two binary buffers.

BinaryPeek(handle, offset)

Returns a 1 byte value from a binary buffer.

BinaryPeek2(handle, offset)

Returns a 2 byte value from a binary buffer.

BinaryPeek4(handle, offset)

Returns a 4 byte value from a binary buffer.

BinaryPeekFlt(handle, offset)

Returns an 8 byte value from a binary buffer.

BinaryPeekStr(handle, offset, maxsize)

Extracts a string from a binary buffer.

BinaryPoke(handle, offset, value)

Pokes a new value into a binary buffer at offset.

BinaryPoke2(handle, offset, value)

Pokes a new value into a binary buffer at offset.
Pokes an 16 bit value.

BinaryPoke4(handle, offset, value)

Pokes a new value into a binary buffer at offset.
Pokes an 32 bit value.

BinaryPokeFlt(handle, offset, value)

Pokes a new value into a binary buffer at offset.
Pokes a 64 bit floating point number.

BinaryPokeStr(handle, offset, string)

Writes a string into a binary buffer.

BinaryRead(handle, filename)

Reads a file into a binary buffer.

BinaryReadEx(handle, binary-offset, filename, file-offset, count)

Reads a portion of a file into a binary buffer.

BinarySort(handle, reccount, key offset, key size, flags)

Sorts records in a binary buffer.

BinaryStrCnt(handle, start-offset, end-offset, string)

Counts the occurrences of a string in some or all of a binary buffer.

BinaryTagExtr(tag-struct, flags)

Returns the text between the last-returned pair of binary tags.

BinaryTagFind(tag-struct)

Finds the next binary tag.

BinaryTagInit(buffer, start-tag, end-tag)

Initializes a binary tag operation.

BinaryTagRepl(tag-struct, new-string)

Replaces a binary tag with text.

BinaryWrite(handle, filename)

Writes a binary buffer to a file.

BinaryWriteEx(handle, binary-offset, filename, file-offset, count)

Writes a portion of a binary buffer to a file.

BinaryXor(target-handle,target-offset,source-handle,source-offset,count)

Performs a bitwise XOR (exclusive OR) on portions of two binary buffers.

Break

Used to exit a conditional flow control statement.

ButtonNames(Ok-name, Cancel-name)

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

C

Call(WIL filename, parameters)

Calls a WIL batch file as a subroutine.

Ceiling(fp_num)

Calculates the ceiling of a value.

Char2Num(string)

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

ClipAppend(string)

Appends a string to the end of the Clipboard.

ClipGet()

Returns the Clipboard contents into a string.

ClipGetEx(format)

Returns the contents of the Windows clipboard in a specified format.

ClipPut(string)

Replaces the Clipboard contents with a string.

Continue

Transfers control to the beginning of a For or While loop or to a different case statement.

Cos(fp_num)

Calculates the cosine.

Cosh(fp_num)

Calculates the hyperbolic cosine.

CurrentFile() {*M}

Returns the filename of the selected item.

CurrFilePath() {*M}

Returns the path and full filename of the selected item.

CurrentPath() {*M}

Returns the path of the selected item.

D

DDEExecute(channel, [commandstring])

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 in seconds**)

Sets the DDE time-out value.

Debug(**mode**)

Turns the Debug mode on or off.

DebugData(**string, string**)

Writes data via the Windows OutputDebugString function to the default destination.

DebugTrace(**mode, filename**)

Creates and writes a line by line debugging file of each statement executed in a script.

Decimals(**#digits**)

Sets the number of decimal points used with floating point numbers.

Dialog(**dialog-name**)

Displays a user-defined dialog box.

DirAttrGet(**[d:]path**)

Gets directory attributes.

DirAttrSet(**dir-list, settings**)

Sets directory attributes.

DirChange(**[d:]path**)

Changes the current directory.

DirExist(**pathname**)

Determines if a directory exists.

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(dir-list)

Removes an existing empty directory.

DirRename([d:]oldpath, [d:]newpath)

Renames a directory.

DirSize(dir-name, flags)

This function returns the total size of a directory, including all files in the directory, and all files in all subdirectories under the directory.

DirWindows(request#)

Returns the name of the Windows or Windows System directory.

DiskExist(drive letter)

Tests for the existence of a drive.

DiskFree(drive-list)

Finds the total available (unused) disk space on a group of drives.

DiskInfo (request#)

Returns information on the specified disk drive.

DiskScan(request#)

Returns a list of drives.

DiskSize (request#)

Finds the total size of the specified drive(s).

DiskVollInfo(root-path, request#) {*32}

Returns information on a file system volume.

Display(seconds, title, text)

Momentarily displays a string.

DllCall(dllfilename/dllhandle, returntype:entrypoint [,paramtype:param...])

Calls an external DLL.

DllCall Additional information

DllFree(dllhandle)

Frees a DLL that was loaded via the DllLoad function.

DllHinst(partial-winname)

Obtains an application instance handle for use in DllCall's when required.

DllHwnd(partial-winname)

Obtains a window handle for use in DllCall's when required.

DllLoad(dllname)

Loads a DLL for later use via the DllCall function.

DOSVersion(level)

Returns the version numbers of the current version of DOS.

Drop(varname [,varname...])

Deletes variables to recover their memory.

DropWild(**variable/wildcard**)

Removes variables from memory.

E

EndSession()

Ends the current Windows session.

Environment(**env-variable**)

Gets a DOS environment variable.

EnvironSet(**env-varname, newvalue**)

Changes LOCAL Environment variables.

EnvItemize()

Returns a delimited list of the current environment.

ErrorMode(**mode**)

Sets what happens in the event of an error.

Exclusive(**mode**)

Controls whether or not other Windows programs will get any time to execute.

Executestatement

Directly executes a WIL statement.

ExeTypeInfo(**exefilename**)

Returns an integer describing the type of EXE file specified.

Exit

Unconditionally ends a WIL program.

Exp(**fp_num**)

Calculates the exponential.

F

Fabs(**fp_num**)

Calculates the absolute value of a floating-point argument.

FileAppend(**source-list, destination**)

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.

FileCompare(**filename1, filename2**)

Compares two files.

FileCopy(**source-list, destination, warning**)

Copies files.

FileCopyAttr(**source-list, destination, warning , attributes**)

Copies files, and sets file attributes.

FileDelete(**file-list**)

Deletes files.

FileExist(**filename**)

Test for the existence of files.

FileExtension(**filename**)

Returns the extension of a file.

FileFullName(**partial filename**)

Returns a file name with drive and path information.

FileItemize(**file-list**)

Builds a list of files.

FileLocate(**filename**)

Finds a file within the current DOS path.

FileMapName(**filename, mapping-data**)

Transforms a filename with a file wild-card mask and returns a new filename.

FileMove(**source-list, destination, warning**)

Moves files to another set of path names.

FileMoveAttr(**source-list, destination, warning**)

Moves files, and sets file attributes.

FileNameLong(**filename**) **{*32}**

Returns the long version of a filename.

FileNameShort(**filename**) **{*32}**

Returns the short (i.e., 8.3) version of a filename.

FileOpen(**filename, mode**)

Opens a STANDARD ASCII (only) file for reading or writing.

FilePath(**filename**)

Returns path of a file.

FileRead(**filehandle**)

Reads data from a file.

FileRename(**source-list, destination**)

Renames files to another set of names.

FileRoot(**filename**)

Returns root of a file.

FileSize(**file-list**)

Adds up the total size of a set of files.

FileTimeCode(**filename**)

Returns a machine readable/computable code for a file time.

FileTimeGet(**filename**)

Returns file date and time.

FileTimeGetEx(filename, time-field)

Gets extended time information for a file or directory.

FileTimeSet(list, YmdHms)

Sets the date and time of one or more files.

FileTimeSetEx(file-list, YmdHms, time-field)

Sets extended time information for one or more files.

FileTimeTouch(file-list)

Sets file(s) to current time.

FileVerInfo(filename, language-key, resource-string)

Returns a version resource string from a binary file.

FileWrite(handle, output-data)

Writes data to a file.

FileYmdHms(filename)

Returns a file time in the YmdHms date/time format.

FindWindow(window class name)

Returns a window title matching a specified class name.

Floor(fp_num)

Calculates the floor of a value.

For varname = initial value to final value [by increment]

Controls the looping of a block of code base in an incrementing index.

G

GetExactTime()

Returns the current time in hundredths of a second.

GetTickCount()

Returns the number of clock ticks used by Windows since Windows started.

GoSub

Transfers control of WIL processing while saving location of the next statement.

Gotolabel

Changes the flow of control in a WIL program.

I

IconReplace(filename, iconfilename)

Replaces an existing icon with a new icon.

If ... Else ... Endifexpression

Conditionally performs a function.

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.

IniRemove(section)

Lists keywords or sections in WIN.INI file.

IniRemovePvt(section, filename)

List 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, data)

Writes a string to the WIN.INI file.

IniWritePvt(section, keyname, data, filename)

Writes a string to a private INI file.

InstallFile(filename, targname, default-targdir, delete-old, flags)

Installs a file.

Int(string/fp_num)

Converts a floating point number or a string to an integer.

IntControl(request#, p1, p2, p3, p4)

Special function which permits an internal operation.

IsDefined(variable name)

Determines if a variable name is currently defined.

IsFloat(value)

Tests whether a number can be converted to a floating point number.

IsInt(string)

Tests whether a number can be converted into a valid integer.

IsKeyDown(keycode)

Determines if the Shift key or the Ctrl key is currently down.

IsLicensed()

Tells if the calling application is licensed.

IsMenuChecked(menuname) {*M}

Determines if a menu item has a check mark next to it.

IsMenuEnabled(menuname) {*M}

Determines if a menu item has been enabled.

IsNumber(value)

Determines if a string represents a valid number.

ItemCount(list, delimiter)

Returns the number of items in a list.

ItemExtract(index, 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.

ItemSortNc(list, delimiter)

Sorts a list, without regard to case.

K

KeyToggleGet(@key)

Returns the status of a toggle key.

KeyToggleSet(@key, value)

Sets the state of a toggle key and returns the previous value.

L

LastError()

Returns the last error encountered.

Log10(fp_num)

Calculates the base-10 logarithm.

LogDisk(drive letter)

Logs (activates) a disk drive.

LogE(fp_num)

Calculates the natural logarithm.

M

Max(number [,number...])

Determines the highest number in a list.

MenuChange {*M}(menuname, flags)

Checks, unchecks, enables, or disables a menu item.

Message(title, text)

Displays text in a message box.

Min(number [,number...])

Determines the lowest number in a list.

MouseClicked(click-type, modifiers)

Clicks mouse button(s).

MouseClickedBtn(parent-windowname, child-windowname, button-text)

Clicks on the specified button control.

MouseInfo(request#)

Returns assorted mouse information.

MouseMove(**X, Y, parent-windowname, child-windowname**)

Moves the mouse to the specified X-Y coordinates.

MsgTextGet(**window-name**)

Returns the contents of a Windows message box.

N

Net101

NetInfo(**requestcode**)

Determines network(s) installed.

Num2Char(**integer**)

Converts a number to its character equivalent.

[Goto Next Page](#)

O

Object101, Ole 2.0, and Applications

ObjectAccess(app.objname)

Opens or creates an OLE 2.0 Automation object.

ObjectClose(**objecthandle**)

Closes OLE 2.0 automation object.

ObjectOpen(**objectname**)

Opens or creates an OLE 2.0 automation object.

P

ParseData(**string**)

Parses a passed string.

Pause(**title, text**)

Displays text in a message box.

PlayMedia(**command-string**)

Controls multimedia devices.

PlayMidi(**filename, mode**)

Plays a MID or RMI sound file.

PlayWaveForm(**filename, mode**)

Plays a WAV sound file.

Print(**data file, directory, display mode, waitflag**)

Instructs an application associated to a file to print the file on the default printer.

R

Random(**integer**)

Generates a positive random number.

Registration Database Operations

RegApp(**program-name, path**) **{*32}**

Creates registry entries for a program under "App Paths".

RegCloseKey(**keyhandle**)

Closes a key to the registration database.

RegConnect(**computer-name, handle**)

Connects to a predefined registry handle on another computer.

RegCreateKey(**keyhandle, sub-key string**)

Returns a handle to a new registration database key.

RegDeleteKey(**keyhandle, sub-key string**)

Deletes a key and data items associated with the key.

RegDelValue(**handle, subkey-string**) **{*32}**

Deletes a named value data item for the specified subkey from the registry.

RegEntryType(**handle, subkey-string**)

Returns the type of data for the specified subkey.

RegExistKey(handle, subkey-string)

Checks whether a registry key exists.

RegExistValue(handle, subkey-string) {*32}

Checks whether a value for a registry key exists.

RegLoadHive(handle, subkey, filename) {*32}

Loads a hive into a registry subkey.

RegOpenKey(keyhandle, sub-key string)

Returns a handle to an existing registration database key.

RegQueryBin(handle, subkey-string) {*32}

Returns binary value at subkey position.

RegQueryDword(handle, subkey-string) {*32}

Returns DWORD value at subkey position.

RegQueryEx(handle, subkey-string, delimiter, type)

Retrieves a value from the registry.

RegQueryExpSz(handle, subkey-string) {*32}

Retrieves a REG_EXPAND_SZ value from the registry.

RegQueryItem(handle, subkey-string) {*32}

Returns a list of named data items for a subkey.

RegQueryKey(keyhandle, index)

Returns sub keys of the specified key.

RegQueryKeys(keyhandle)

Returns a tab-delimited list of subkey names under a specified key.

RegQueryMulSz(handle, subkey-string, delimiter) {*32}

Retrieves a REG_MULTI_SZ value from the registry.

RegQueryValue(keyhandle, keyname)

Returns data item string at sub-key position.

RegSetBin (handle, subkey-string, value) {*32}

Sets a binary value in the Registration Database.

RegSetDword (handle, subkey-string, value) {*32}

Sets a DWORD value in the Registration Database.

RegSetEx(handle, subkey-string, value, delimiter, type)

Sets a value in the registry.

RegSetExpSz(handle, subkey-string, value) {*32}

Sets a REG_EXPAND_SZ value in the registry.

RegSetMulSz(handle, subkey-string, value, delimiter) {*32}

Sets a REG_MULTI_SZ value in the registry.

RegSetValue(keyhandle, sub-key string, value)

Sets the value of a data item in the registration database.

RegUnloadHive(handle, subkey){*32}

Unloads a hive from the registry.

Reload {*M}

Reloads menu file(s).

Return

Returns from a Call to the calling program or from a GoSub :label.

Run(program-name, parameters)

Runs a program as a normal window.

RunEnviron(program-name, parameters, displaymode, waitflag)

Launches a program in the current environment as set with the EnvironSet command.

RunHide(program-name, parameters)

Runs a program as 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.

RunShell(program-name, params, directory, displaymode, waitflag)

An advanced form of the Run function that even allows the specification of a working directory, along with the window view mode and whether or not to wait for completion of the run program in a single function.

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.

S

Selectvarname

Allows selection among multiple blocks of statements.

SendKey(sendkey-string)

Sends keystrokes to the currently active window.

SendKeysChild(partial-parent-windowname, partial-child-windowname, sendkey-string)

Sends keystrokes to the active child window.

SendKeysTo(parent-windowname, sendkey-string)

Sends keystrokes to a "windowname".

SendMenusTo(partial-parent-windowname, menuname)

Activates a window and sends a specified menu option.

ShellExecute (program-name, params, directory, display mode, operation)

Runs a program via the Windows ShellExecute command

ShortcutDir(name)

Returns the name of the directory where shortcuts of type "name" are stored.

ShortcutEdit(link-name, target, params, start-dir, show-mode)

Modifies the specified shortcut file.

ShortcutExtra(link-name, description, hotkey, icon-file, icon-index)

Sets additional information for the specified shortcut file.

ShortcutInfo(link-name)

Returns information on the specified shortcut file.

ShortcutMake(link-name, target, params, start-dir, show-mode)

Creates a shortcut for the specified filename or directory.

Sin(fp_num)

Calculates the sine.

Sinh(fp_num)

Calculates the hyperbolic sine.

SnapShot(request#)

Takes a snapshot of the screen and pastes it to the clipboard.

Sounds(request#)

Turns sounds on or off.

Sqrt(fp_num)

Calculates the square root.

StrCat(string [,string])

Concatenates strings together.

StrCharCount(string)

Counts the number of characters in a string.

StrCmp(string1, string2)

Compares two strings.

StrFill(filler, length)

Builds a string from a repeated smaller string.

StrFix(base-string, pad-string, length)

Pads or truncates a string to a fixed length.

StrFixChars(base-string, pad-string, length)

Pads or truncates a string To a fixed length using characters.

StrFixCharsL(base-string, pad-string, length)

Pads or truncates the left side of a string to a fixed length using characters.

StrFixLeft(base-string, pad-string, length)

Pads or truncates the left side of a string to a fixed length using bytes.

StriCmp(**string1, string2**)

Compares two strings without regard to case.

StrIndex(**base-string, sub-string, start, direction**)

Locates a string within a larger string.

StrIndexNc(**base-string, sub-string, start, direction**)

Locates a string within a larger string, ignoring case.

StrIndexWild(**string, pattern, start**)

Finds wildcarded text within a larger string.

StrLen(**string**)

Returns the length of a string.

StrLenWild(**string, pattern, start**)

Returns the length of wildcarded text within a larger string.

StrLower(**string**)

Converts a string to all lower-case characters.

StrReplace(**string, old, new**)

Replaces all occurrences of a sub-string with another.

StrScan(**string, delimiters, startpos, direction**)

Finds an occurrence of one or more delimiter characters in a string.

StrSub(**string, startpos, length**)

Returns a sub-string from within a string.

StrSubWild(**string, pattern, start**)

Extracts a substring matching wildcarded text from within a larger string.

StrTrim(**string**)

Trims leading and trailing blanks from a string.

StrUpper(**string**)

Converts a string to all upper-case characters.

Switch**varname**

Allows selection among multiple blocks of statements.

T

Tan(**fp_num**)

Calculates the tangent.

Tanh(**fp_num**)

Calculates the hyperbolic tangent.

Terminate(**expression, title, message**)

Conditionally ends a WIL program.

TimeAdd(**YmdHms, YmdHms**)

Adds two YmdHms variables.

TimeDate()

Provides the current date and time in a readable format.

TimeDayofWeek

Example: Using the WIL language it is possible to grab the day of the week. The following example from TimeJulianDay grabs the current time and day of the week.

TimeDayofYear

Example: Using the WIL language it is possible to grab the day of the year. The following example from TimeJulianDay grabs the current time and day of the year.

TimeDelay(**seconds**)

Pauses execution for a specified amount of time.

TimeDiff(**time-value-1**, **time-value-2**)

Returns the difference between two points in time.

TimeDiffDays(**Ymd[Hms]**, **Ymd[Hms]**)

Returns the difference in days between the two dates.

TimeDiffSecs(**YmdHms**, **YmdHms**)

Returns the time difference in seconds between the two date times.

TimeJulianDay(**Ymd[Hms]**)

Returns the julian day given a date/time.

TimeJulToYmd(**julian-date**)

Returns a datetime given a Julian day.

TimeSubtract(**datetime**, **datetime difference**)

Subtracts one YmdHms variable from another.

TimeWait(**YmdHms**)

Pauses execution and waits for the date/time to pass.

TimeYmdHms()

Returns current date/time in the date/time format.

V

Version()

Returns the version of the parent program currently running.

VersionDLL()

Returns the version of the WIL interpreter currently running.

W

WaitForKey(**key**, **key**, **key**, **key**, **key**)

Waits for a specific key to be pressed.

WallPaper(**bmp-filename**, **tilemode**)

Changes the Windows wallpaper.

While **expression**

Conditionally and/or repeatedly executes a series of statements.

WinActivate(**partial-winname**)
Makes an application window the active window.

WinActivChild(**partial-parent-windowname, partial-child-windowname**)
Activates a previously running child window.

WinArrange(**style**)
Arranges all running application windows on the screen.

WinClose(**partial-winname**)
Closes an application window.

WinExeName(**partial-winname**)
Returns the name of the executable file which created a specified window.

WinExist(**partial-winname**)
Tells if a window exists.

WinExistChild(**partial-parent-windowname, partial-child-windowname**)
Tells if a specified child window exists.

WinGetActive()
Gets the title of the active window.

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

WinHide(**partial-winname**)
Hides an application window.

WinIconize(**partial-winname**)
Turns an application window into an icon.

WinIdGet(**partial-winname**)
Returns a unique "Window ID" (pseudo-handle) for the specified window name.

WinIsDOS(**partial-winname**)
Tells whether or not a particular window is a DOS or console-type window.

WinItemChild(**partial-parent-windowname**)
Returns a list of all the child windows under this parent.

WinItemize()
Returns a tab-delimited list of all open windows.

WinItemizeEx(**partial-winname, multiple, hidden**)
Returns the full name of window(s) matching a partial window name.

WinItemNameId()
Returns a list of all open windows and their Window ID's.

WinItemProclD(**process-id, flags, return-type**)
Returns a list of window ID's for the specified process.

WinMetrics(**request#**)

Returns Windows system information.

WinName()
Returns the name of the window calling the WIL Interpreter.

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

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

WinPlace(**x-ulg, y-ulg, x-brc, y-brc, 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.

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

WinShow(**partial-winname**)
Shows a window in its "normal" state.

WinState(**partial-winname**)
Returns the current state of a window.

WinSysInfo() { *32 }
Returns system configuration information.

WinTitle(**old-partial-winname, new-winname**)
Changes the title of an application window.

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

WinWaitChild(**partial parent winname, partial child winname, timeout**)
Waits for a child window to exist.

WinWaitClose(**partial-winname**)
Waits until an application window is closed.

WinWaitExist(**partial-winname, timeout**)
Waits for a window to exist.

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

Y

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

Yields

Provides time for other windows to do processing.

WIL Language Elements

A program written in Windows Interface Language (WIL) consists of a series of statements. WIL statements are constructed from constants, variables, operators, functions, commands, and comments.

In addition, WIL scripts can take advantage of advanced techniques such as substitution, passing parameters, and error handling.

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

[Constants](#)

[Identifiers](#)

[Variables](#)

[Lists](#)

[Keywords](#)

[Operators](#)

[Precedence and Evaluation Order](#)

[Comments](#)

[Statements](#)

[Substitution](#)

[Parameters](#)

[Error Handling](#)

• [Reference](#)

• [Step by step guide to learning WIL](#)

• [Using WIL](#)

• [Notes](#)

Constants

Windows Interface Language (WIL) supports three types of constants: floating point, integer and string constants. In addition, there is an extensive list of predefined constants to make your scripts easier to read.

- [Floating Point Constants](#)
- [Integer Constants](#)
- [String Constants](#)
- [Predefined Constants](#)
- [Reference](#)
- [Step by step guide to learning WIL](#)
- [Notes](#)

Floating Point Constants

Floating point constants are built from the digits 0 through 9, the plus and minus signs, a period '.' and the letter 'E'. They can range in magnitude from negative to positive 1.0E+300 (a very large number). Constants larger than these permissible magnitudes will produce unpredictable errors. Floating point constants must begin with a digit.

Examples of floating point constants:

3.14159

-8.92E-45

0.0001724

8.95e294

- [Integer Constants](#)
- [String Constants](#)
- [Predefined Constants](#)
- [Reference](#)
- [Step by step guide to learning WIL](#)
- [Notes](#)

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

- [Floating Point Constants](#)
- [String Constants](#)
- [Predefined Constants](#)
- [Reference](#)
- [Step by step guide to learning WIL](#)
- [Notes](#)

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'

`Look at these backquotes`

"This constant has an embedded "" mark"

'This constant also has an embedded " mark'

- [Floating Point Constants](#)
- [Integer Constants](#)
- [Predefined Constants](#)
- [Reference](#)
- [Step by step guide to learning WIL](#)
- [Notes](#)

Predefined Constants

WIL has a number of built-in constants that can be used for various purposes, including making your scripts easier to read. These start with the @-sign, and are case-insensitive.

Some predefined constants:

@FALSE
@NO
@STACK
@TILE
@TRUE
@YES

A list of all the predefined constants can be found in [Predefined System Constants](#).

- [Constants](#)
- [Predefined System Constants](#)
- [Predefined Floating Point Constants](#)
- [Predefined String Constants](#)
- [Reference](#)
- [Step by step guide to learning WIL](#)
- [Notes](#)

Predefined System Constants

WIL has a number of built-in integer constants that can make your scripts much more readable. These start with the @-sign, and are case-insensitive.

Logical Conditions

@NO
@OFF
@TRUE
@YES
@ON
@FALSE

Window Arranging

@NORESIZE
@ABOVEICONS
@STACK
@ARRANGE
@TILE
@ROWS
@COLUMNS

Window Status

@NORMAL
@ZOOMED
@ICON
@HIDDEN

Menu Handling

@CHECK
@UNCHECK
@DISABLE
@ENABLE

OS Dependent

@CAPSLOCK
@NUMLOCK
@REGCLASSES
@REGCURRENT
@REGMACHINE
@REGROOT
@REGUSERS
@SCROLLLOCK
@WHOLESECTION

BinarySort Control

@ASCENDING
@DESCENDING
@STRING
@WORD1
@WORD2
@WORD4
@FLOAT8

INI File Management

@WHOLESECTION

Mouse Control

@LBUTTON
@RBUTTON
@MBUTTON
@LCLICK
@RCLICK
@MCLICK

- [Constants](#)

- [Predefined Floating Point Constants](#)

- [Predefined String Constants](#)

- [Reference](#)

- [Step by step guide to learning WIL](#)

- [Notes](#)

String Handling

@FWDSCAN
@BACKSCAN

System Control

@MAJOR
@MINOR

Error Handling

@CANCEL
@NOTIFY
@OFF

Keyboard Status

@SHIFT
@CTRL

@LDBLCLICK
@RDBLCLICK
@MDBLCLICK

Miscellaneous

@MULTIPLE
@NOWAIT
@OPEN
@ROWS
@SAVE
@SINGLE
@SORTED
@STACK
@TILE
@UNSORTED
@WAIT

Predefined Floating Point Constants

WIL has several predefined floating point constants that can be handy for scientific and engineering scripts. Like all WIL constants, they start with the @-sign, and are case-insensitive.

@AMC

Atomic Mass
Constant
1.66043E-27

@AVOGADRO

Avogadro's
Constant
6.02252E23

@BOLTZMANN

Boltzmann Entropy
Constant
1.38054E-23

@DEG2RAD

Degrees to
Radians Conversion
Constant

0.017453292519943

@e

Base of natural or
Napierian logarithms

2.718281828459045

@ELECTRIC

Electric Field
Constant
8.8541853E-12

@EULERS

@GOLDENRATIO

Golden Ratio
1.6180339887498948

@GRAVITATION

Gravity Constant
6.670E-11

@LIGHTMPS

Speed of Light
(miles/sec)
186272

@LIGHTMTPS

Speed of Light
(meters/sec)
2.997925E8

@MAGFIELD

Magnetic Field
Constant
1.256637

@PARSEC

Parsec (AU)
206.265

@PI

Pi
3.141592653589793

@PLANCKERGS

Planck's Constant

• [Constants](#)

• [Predefined System Constants](#)

• [Predefined String Constants](#)

• [Reference](#)

• [Step by step guide to learning WIL](#)

• [Notes](#)

Eulers's Constant
0.5772156649015338

@FARADAY
Faraday Constant
9.64870E4

@GFTSEC
Gravitational
Acceleration (feet/sec²)
32.174

@GMTSEC
Gravitational
Acceleration
(meters/sec²)
9.80665

(ergs)
6.6252E-27

@PLANCKJOULES
Planck's Constant
(joules)
6.6256E-34

@RAD2DEG
Radians to Degrees
Conversion Constant
57.29577951308232

Predefined String Constants

WIL defines a few string constants that are handy for formatting and processing text data. These start with the @-sign, and are case-insensitive.

@CRLF
0x13,0x10 cr,lf

@CR
0x13, cr

@LF
0x10, lf

@TAB
0x09 tab

- [Constants](#)
- [Predefined System Constants](#)
- [Predefined Floating Point Constants](#)
- [Reference](#)
- [Step by step guide to learning WIL](#)
- [Notes](#)

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(MyTiTIE, pRoMpT, dEfAuLt)
```

- [WIL Language Elements](#)
- [Reference](#)
- [Step by step guide to learning WIL](#)
- [Notes](#)

Variables

A variable may contain an integer, a floating point number, a string, a list, or a string representing an integer or a floating point number . Automatic conversions between numbers and strings are performed as a matter of course during execution.

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

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

To simulate arrays, use variable substitution.

- [WIL Language Elements](#)
- [Reference](#)
- [Step by step guide to learning WIL](#)
- [Notes](#)

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.

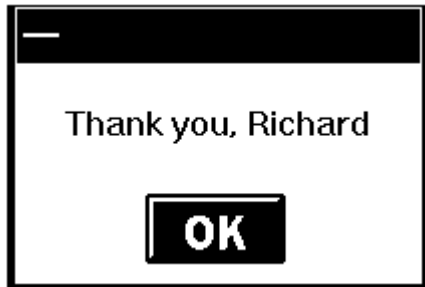
Examples:

```
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  
line0=""  
While line%num% != "*EOF*"  
    num = num + 1  
    line%num% = FileRead(handle)  
EndWhile  
FileClose(handle)  
handle = FileOpen("c:\config.rev", "WRITE")  
While num > 1  
    num = num - 1
```

- [WIL Language Elements](#)
- [Reference](#)
- [Step by step guide to learning WIL](#)
- [Notes](#)

```
        FileWrite(handle, line%num%)
    EndWhile
    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.

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 AskItemizeList, you will need to know what character is being used to delimit the list.

In order to support long file names in Windows NT and Windows 95/98, which can contain embedded spaces, the default file delimiter used to delimit lists of files and directories has been changed to a TAB in the 32-bit version of WIL. In the 16-bit version of WIL, the default delimiter has not changed, and remains a space.

To change the file delimiter to a character of your own choosing, use **IntControl (29,p1,0,0,0)**. When using the 32-bit version of WIL consider making the file delimiter a space for compatibility with existing scripts.

- [WIL Language Elements](#)
- [Reference](#)
- [Step by step guide to learning WIL](#)
- [Notes](#)

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

- [WIL Language Elements](#)
- [Reference](#)
- [Step by step guide to learning WIL](#)
- [Notes](#)

Operators

Windows Interface Language operators take one operand ("unary operators") or two operands ("binary operators").

Unary operators (integers and floating point numbers)

- Arithmetic Negation (Two's complement)
- + Identity (Unary plus)

Unary operators (integers only)

- ~ 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 logical operators (integers only)

- << Left Shift
- >> Right Shift
- & Bitwise And
- | Bitwise Or
- ^ Bitwise Exclusive Or (XOR)
- && Logical And
- || Logical Or

Binary arithmetic operators (integers and floating point numbers)

- ** Exponentiation
- * Multiplication
- / Division
- mod Modulo
- + Addition
- Subtraction

Binary relational operators

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

- [Precedence and evaluation order](#)

- [WIL Language Elements](#)

- [Reference](#)

- [Step by step guide to learning WIL](#)

- [Notes](#)

Assignment operator

= 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
**	Exponentiation
* / mod	Multiplication & Division
-	Addition & Subtraction
<< >>	Shift operators
< <= == >= > !=	Relational operators
<>	Bit manipulation operators
& ^	Logical operators
&&	

- [Operators](#)
- [WIL Language Elements](#)
- [Reference](#)
- [Step by step guide to learning WIL](#)
- [Notes](#)

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 semi-colon 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
```

- [WIL Language Elements](#)

- [Reference](#)

- [Step by step guide to learning WIL](#)

- [Notes](#)

Statements

Assignment Statements

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

Examples:

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

- [WIL Language Elements](#)
- [Reference](#)
- [Step by step guide to learning WIL](#)
- [Notes](#)

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
While
For
Yield
Switch
Return
```

Function Parameters

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

- Integer
- Floating point number
- String
- List
- Variable name

The interpreter performs automatic conversions between strings, integers and floating point numbers, so that you can use them interchangeably. In general, the only case to be careful of is comparing two floating point numbers -- floating point numbers have a habit of never being quite equal when you want them to.

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

Floating point parameters may be any of the following:

- A floating point number (i.e. 3.141569)
- A string representing an integer (i.e. "314.1569E-2")
- A variable containing a floating point number
- A variable containing a string representing a floating point number

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
- A variable containing a floating point number

- [WIL Language Elements](#)
- [Reference](#)
- [Step by step guide to learning WIL](#)
- [Notes](#)

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.

Note: You must read the section on the `ErrorMode` function completely before attempting to use the function to suppress run-time errors. For an example of using `ErrorMode`, see the function [LastError](#).

- [Error Messages](#)
- [LastError](#)
- [WIL Language Elements](#)
- [Reference](#)
- [Step by step guide to learning WIL](#)
- [Notes](#)

Error Messages

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 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.

For moderate errors, if the error mode is @NOTIFY or @OFF, the user has the option of continuing with the WIL program or canceling it.

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).

For an example of trapping an error message, see the function [LastError](#).

- [Minor Errors](#)
- [Moderate Errors](#)
- [Fatal Errors](#)
- [Error Handling](#)
- [Reference](#)
- [Step by step guide to learning WIL](#)
- [Notes](#)

Minor 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 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.

- 1002 File List Processing - No Match
- 1003 FileMove: Failed
- 1004 FileMove: FROM file open failed
- 1005 FileMove: TO file open failed
- 1006 FileMove: I/O error
- 1007 FileMove: Could not delete FROM file
- 1008 FileCopy: Failed
- 1009 FileCopy: FROM file open failed
- 1010 FileCopy: TO file open failed
- 1011 FileCopy: I/O error
- 1012 FileAppend: FROM file open failed
- 1013 FileAppend: TO file open failed
- 1014 FileAppend: I/O error
- 1015 FileRename: Failed
- 1016 FileDelete: File not found
- 1017 TimeDiff: Time parameter error - bad value
- 1018 TimeDiff: Out of Range (over 60 years)
- 1019 TimeAdd: Cannot add supplied times
- 1020 TimeAdd: Time parameter error - bad value
- 1021 WaitLong: Could not properly compute time for delay
- 1022 WaitUntil: Passed time not in proper format
- 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
- 1034 Clipboard owned by another app. Cannot open.
- 1035 Clipboard does not contain text for CLIPAPPEND.
- 1036 Clipboard cannot hold that much text (>64000 bytes)
- 1037 Unable to get memory for clipboard. Close some apps
- 1039 WinClose: Window not found
- 1040 WinHide: Window not found
- 1041 WinIconize: Window not found
- 1042 WinZoom: Window not found

Moderate Errors

- _____
Fatal Errors

Error Handling • _

Step by step guide to learning WIL • _

- Notes [{vflid2336630925664678512}](#)
- [{vflid7308609264982843757}Á{vflid7503110703112418419}](#)
- [{vflid7453010347607089512}](#)
- [{vflid7600212964535396708}](#)
- [{vflid8386104319403257449}](#)
- [{vflid8030604370215531381}](#)
- [{vflid7956018242947871344}](#)
- [{vflid7863416269698918775}](#)
- [{vflid7308609264982843757}](#)
- [{vflid7453010381916564582}](#)
- [{vflid2316574742769397616}](#)
- [{vflid7306920471174914086}](#)
- [{vflid8387230176444510817}](#)
- [{vflid8028075858292209765}](#)
- [{vflid2336364844068794211}](#)
- [{vflid7310027659661307489}](#)
- [{vflid7309464380489228621}](#)
- [{vflid8028075772644520047}](#)
- [{vflid8171062582650693748}](#)
- [{vflid2334397779128968562}](#)
- [{vflid7381153994278134128}](#)
- [{vflid2334382411428685423}](#)
- [{vflid8243109542261452393}](#)
- [{vflid7885065666585129584}](#)
- [{vflid7306916077306274156}](#)
- [{vflid6998705293059055971}](#)
- [{vflid7596570286789714785}](#)
- [{vflid8295737305435761015}](#)
- [{vflid7308330019228688481}](#)
- [{vflid7164793138682022231}](#)
- [{vflid199387537763}](#)
- [{vflid6998704207841945964}](#)
- [{vflid8387218051550964833}](#)
- [{vflid2337479757427407202}](#)
- [{vflid8031079655356855655}](#)
- [{vflid7958530579750743924}](#)
- [{vflid541870423}](#)

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
1126 BinaryAlloc: Could not allocate binary buffer
1128 FileTimeTouch: File not found
1129 OleInitiate: Initiate Failed
1133 OleExecute: Could not process OLE command
1134 OleExecute: Function syntax error
1141 OleExecute: Not enough format ids for all parameters
1143 OleExecute: Format id problem
1144 DDETerminate: Channel not open
1150 DDEExec: DDE Post failed
1155 DDEReq: DDE Post failed
1158 RegOpenKey: Function Failed
1159 RegCreateKey: Function 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
1226 DirExist: Invalid path specified
1227 WinIsDos: Window not found or bad window
1229 RegDeleteKey: Function Failed
1230 RegDeleteKey: Access Denied
1231 RegCloseKey: Function Failed
1232 RegSetValue: Function Failed
1233 RegQueryValue: Function Failed
1240 ExeTypeInfo/RunEnviron: Cannot Locate File
1241 RunEnviron: Not a Windows EXE file
1242 EnvironSet: Not enough environment space left.
1248 Ole:System Ole Dll's not found (not installed?)
1249 Ole Object: Could not process value returned from object
1251 Ole: WIL Ole interface Dll not found (WBOLExxx.DLL)
1253 Function not supported on this platform.
1254 Ole: Unknown interface
1255 Ole: Member not found
1256 Ole: Param not found
1257 Ole: Type mismatch
1258 Ole: Unknown name
1259 Ole: No named args
1260 Ole: Bad variable type
1261 Ole: Exception
1262 Ole: Overflow
1263 Ole: Bad index
1264 Ole: Unknown LCID
1265 Ole: Array is locked
1266 Ole: Bad param count
1267 Ole: Param not optional
1268 Ole: Bad callee
1269 Ole: Not a collection
1270 Ole: IO error
1271 Ole: Compile error
1272 Ole: Cannot create tempfile
1273 Ole: Illegal index
1274 Ole: Id not found
1275 Ole: Buffer too small
1276 Ole: Read only
1277 Ole: Invalid data read
1278 Ole: Unsupported format
1279 Ole: Already contains name
1280 Ole: No matching arity
1281 Ole: Registry access problem

1282 Ole: Lib not registered
1283 Ole: Duplicate definition
1284 Ole: Usage
1285 Ole: Dest not known
1286 Ole: Undefined type
1287 Ole: Qualified name disallowed
1288 Ole: Invalid state
1289 Ole: Wrong type kind
1290 Ole: Element not found
1291 Ole: Ambiguous name
1292 Ole: Invoke function mismatch
1293 Ole: DLL function not found
1294 Ole: Bad module kind
1295 Ole: Wrong platform
1296 Ole: Already being laidout
1297 Ole: Cannot load library
1298 Ole: Error code not recognized
1299 Dll: DLL file not found
1300 Dll: File not loadable
1301 DllCall: Bad Entrypoint name
1302 DllCall: Bad Global Pointer returned from called DLL
1303 IconReplace: EXE file not found
1304 IconReplace: ICO file not found
1305 IconReplace: ICO file open failed
1306 IconReplace: Invalid ICO file
1307 IconReplace: Memory Alloc Error
1308 IconReplace: EXE file open failed (in use?)
1309 IconReplace: Unrecognized EXE file
1310 IconReplace: Not a Windows EXE file
1311 IconReplace: No resources in EXE file
1312 IconReplace: New Icon is larger than old icon
1313 IconReplace: Invalid EXE file
1314 IconReplace: No icons found in EXE file
1315 SendMenusTo: Window menu not accessible
1316 SendMenusTo: MenuItem name not found
1317 SendMenusTo: PostMessage Failed
1318 Ole: WBOLExxx.DLL LoadLibrary failure
1319 OS2Sound: Could not communicate with OS2
1323 FileFullName: Filename cannot be legally expanded
1324 FileMapName: Filename cannot be legally mapped to mask
1330 BinaryRead: File size larger than binary buffer size
1334 WinItemChild: Parent Window cannot be found
1335 IniPrivate functions: Illegal to access [386Enh] Device= keywords
1336 Ask Multiple: More than 99 items selected. Too Many.
1337 AskFileName: Dialog Box creation error
1341 FP Math: Argument to function outside domain of function
1342 FP Math: Result is too large to be represented
1343 FP Math: Partial loss of significance occurred
1344 FP Math: Illegal value passed to function. (Singularity)
1345 FP Math: Total loss of significance occurred

1346 FP Math: Result too small to be represented
1347 FP Math: Undocumented library error passed to matherr
1348 FP Math: Non-Integer Power of Negative Number is not defined
1349 FP Math: Square Root of a Negative Number
1350 FP Math: Cannot Take Log of Zero or a Negative Number
1351 FP Math: Fact args must be positive whole numbers <=170
1368 ActivateChild: Child windows does not exist
1369 DIICall: Invalid DIIName as Param1
1370 DIICall: Invalid DIIEnterPoint
1371 DIICall: Bad punctuation found
1372 DIICall: Too many parameters (max 2 + 15 args)
1373 DIICall: Must have at least 3 parameters
1374 DIICall: Number of DLL parameters and type string do not agree
1375 DIICall: Parameter cannot be forced to 'SHORT'
1376 DIICall: Parameter cannot be forced to 'LONG'
1377 DIICall: Return type invalid - WORD LONG or LPSTR
1378 DIICall: Bad parm code. Only WORD, LONG, LPSTR, LPBINARY or LPNULL
1379 DIICall: Bad type list caused stack problems. Check types carefully.
1380 DIICall: Missing ':' after type code
1381 DIICall: Param cannot be converted to string for LPSTR
1388 Request Ignored: NT Security violation
1389 FileCompare: FileOpen failure - First File
1390 FileCompare: FileOpen failure - Second File
1398 List Allocation: Could not itemize all items
1399 AddExtender: Extender detects error, will not load
1400 Call: File not found
1402 REG Functions: Bad Data Item [name]
1404 FileCopy: Insufficient free space on target drive
1405 FileMove: Insufficient free space on target drive
1406 FileAppend: Insufficient free space on target drive
1407 IntControl 29: Invalid delimiter character
1408 WinIdGet: Window not found
1409 Shortcut functions require Windows 95
1410 Shortcut functions: Shortcut files must have an extension of '.LNK'
1411 Shortcut functions: Shortcut file not found
1412 ShortcutMake: Shortcut file already exists
1413 Shortcut Make/Edit: Invalid show mode
1414 ShortcutExtra: Invalid hotkey
1415 Shortcut functions: Colnitialize failed
1416 Shortcut functions: CoCreateInstance failed
1417 Shortcut functions: QueryInterface failed
1418 Shortcut functions: Error loading shortcut file
1419 Shortcut functions: Error reading shortcut file
1420 Shortcut functions: Error saving shortcut file
1421 DirAttrGet: Directory not found
1422 DirAttrSet: Directory not found or access denied
1423 FileNameLong: File not found
1424 FileNameShort: File not found
1425 WIL Internal Error

1426 IntControl 30: Source file not found
1427 IntControl 30: Error parsing target spec
1428 IntControl 30: Cannot move file to a different drive
1429 RegApp: Function not supported in 16-bit version
1430 RegApp: File not found
1431 RegApp: Error writing to registry
1432 RegDelValue: Function not supported in 16-bit version
1433 InstallFile: Function not supported in 16-bit version
1434 InstallFile: Source file not found (or not specified)
1435 InstallFile: Target file name cannot contain a path
1436 InstallFile: Target directory not found (or not specified)
1438 WinSysInfo: Function not supported in 16-bit version
1439 Mouse Functions: Invalid click-type
1440 Mouse Functions: Unable to determine window containing mouse
1441 Mouse Functions: Child window specified with no parent
1442 Mouse Functions: Parent window not found
1443 Mouse Functions: Child window not found
1444 MouseClickBtn: Button not found
1445 IconReplace: Unable to create file mapping
1446 IconReplace: Unable to map view of file
1447 IconReplace: New icon is smaller than old icon
1448 IntControl 32: Invalid data type
1449 ShellExecute: Error launching specified file
1450 RegQueryItem: Function not supported in 16-bit version
1451 RegQueryItem: Unable to open specified subkey
1452 REG Functions: Unable to open (or create) specified subkey
1453 RegQueryValue: Binary data found. Use RegQueryBin instead.
1454 RegQueryBin: Data is not binary
1455 RegQueryBin: Unable to allocate or lock memory
1456 RegQueryDword: Data is not a DWORD
1457 RegSetBin: Invalid binary value string
1458 RegSetBin: Binary value string too long
1459 RegSetDword: Invalid DWORD value
1460 REG functions: Subkey string too long
1461 TimeJulToYmd: Invalid Julian date
1462 TimeSubtract: Cannot subtract supplied times
1463 TimeSubtract: Time parameter error - bad value
1464 IntControl 36: Window not found
1465 IgnoreInput: Function not supported in 32-bit version
1466 IconReplace: Invalid icon index
1468 DiskInfo: Invalid drive letter
1469 DiskInfo: Invalid request number
1470 DiskInfo: Function failed
1471 IntControl 39/40: Invalid share mode
1474 BinaryHash/Sort: Invalid record size
1475 BinaryHash/Sort: Invalid key offset
1476 BinaryHash/Sort: Invalid key size
1477 BinaryHash/Sort: Specified key extends beyond end of specified
record
1478 BinaryHash/Sort: Specified record size is larger than binary buffer

1479 BinaryHash/Sort: Unable to allocate (or lock) memory
1480 BinaryHash: Key value must be specified
1481 BinaryHash: Specified key value is larger than specified key size
1482 BinaryHash: No available space in binary buffer
1483 RegQueryExpSz: Registry value is not a REG_EXPAND_SZ
1484 RegQueryMulSz: Registry value is not a REG_MULTI_SZ
1485 RegQuery[.]: Function not supported in 16-bit version
1486 RegSet[.]: Function not supported in 16-bit version
1487 RegQueryMulSz/RegSetMulSz: Invalid delimiter
1488 RegQueryValue: Registry value is a REG_EXPAND_SZ. Use
RegQueryExpSz instead.
1489 RegQueryValue: Registry value is a REG_MULTI_SZ. Use
RegQueryMulSz instead.
1490 RegQueryValue: Registry value could not be converted to a string
1491 Ole: Unable to allocate (or lock) memory
1492 RegQueryEx: Invalid 'type'
1493 RegSetEx: Invalid 'type'
1494 RegEntryType: Unable to access registry value
1495 IntControl 41: Invalid parameter
1496 FileVerInfo: File not found
1497 FileVerInfo: Unable to access version information for file
1498 FileVerInfo: Unable to allocate (or lock) memory
1499 FileVerInfo: Specified resource string not found
1500 Undefined error
1501 IntControl 43: Invalid parameter
1502 RegConnect: Remote registry service not installed
1503 RegConnect: Invalid computer name
1504 RegConnect: Invalid handle
1505 RegConnect: Function failed
1506 RegConnect: Function not supported in 16-bit version
1507 FileCopyPort: Error accessing file
1508 FileCopyPort: Error accessing port
1509 FileCopyPort: Error reading file
1510 FileCopyPort: Error writing to port
1511 FileCopyPort: Function not supported in 16-bit version
1512 InstallFile: Buffer too small (WIL internal error)
1513 Ole: Out of memory / Buffer too small
1514 BinaryOleType: Invalid 'type'
1515 BinaryOleType: 'type' does not include a data type
1516 BinaryOleType: 'type' does not include a direction
1517 Ole Object: Must first use 'BinaryOleType' to identify the binary buffer
1518 Ole Object: Unsupported binary buffer type (specified using
'BinaryOleType')
1519 DiskVolInfo: Invalid root path
1520 DiskVolInfo: Invalid request number
1521 DiskVolInfo: Function not supported in 16-bit version
1522 DiskVolInfo: Function failed
1523 RegLoadHive: Specified file not found
1524 RegLoad/UnloadHive: Unable to set NT security privilege for
operation
1525 RegLoad/UnloadHive: Invalid key handle

1526 RegLoad/UnloadHive: Access denied
1527 RegLoadHive: Sharing violation (specified file is in use)
1528 RegUnloadHive: Invalid subkey
1529 RegLoad/UnloadHive: Function failed
1530 RegLoad/UnloadHive: Function not supported in 16-bit version
1531 RegLoadHive: Specified file is not a valid hive
1532 RegLoadHive: Bad key, or duplicate subkey
1533 RegUnloadHive: Bad key or subkey

Moderate 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.

For 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
- 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
- 2331 BinaryStrGet: Data request extends beyond end of binary buffer
- 2332 BinaryStrSet: Data to store would overrun binary buffer
- 2392 DIIILoad: Too many open DIIILoads

- [Minor Errors](#)
- [Fatal Errors](#)
- [Error Handling](#)
- [Reference](#)
- [Step by step guide to learning WIL](#)
- [Notes](#)

Fatal 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.

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).

- 3023 BinaryData: Invalid Binary Data handle passed
- 3024 BinaryData: Too many open Binary Data Handles
- 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
- 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 No IF to relate to THEN or ELSE is currently valid
- 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 a valid number
- 3059 Illegal Bounds for STRSUB function
- 3061 Illegal Syntax
- 3062 Attempt to divide by zero
- 3063 Binary operation not legal for type of number
- 3064 Unary operation not legal for type of number
- 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

- [Minor Errors](#)
- [Moderate Errors](#)
- [Error Handling](#)
- [Reference](#)
- [Step by step guide to learning WIL](#)
- [Notes](#)

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
3127 BinaryEODSet: Set value beyond end of buffer
3130 OleTerminate: Bad Ole Channel
3131 OleExecute: Bad Ole Channel
3132 Ole: Ole has not been initialized
3135 OleExecute: Syntax Error - Needs more parameters
3136 DDEInitiate: Undefined Error
3137 DDEInitiate: Nobody Around to talk to
3138 DDEInitiate: Too many DDE conversations
3139 DDEInitiate: Bad Channel Number
3140 DDEInitiate: Create String Failure
3142 DDETerminate: Channel does not exist

3145 DDEExec: GlobalAlloc failed
3146 DDEExec: Global Lock failed
3147 DDEExec: Bad channel number
3148 OleInitiate: Application does not support Ole
3149 DDEExec: Internal Error 3149
3151 DDEReq: Undefined Error
3152 DDEReq: Bad channel number
3153 DDEReq: Null handle returned
3154 DDEReq: Create String Failed
3156 DDEReq: GlobalLock failed
3157 OleInitiate: Too many open channels
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
3228 Function not available in Windows NT
3234 RunExit: Not EXE, COM or BAT file
3235 RunExit: EXE name too long (max 127)
3236 RunExit: Params too long (max 126)
3237 RunEnviron: Params too long (max 119)
3238 RunEnviron: EXE file NOT a Windows file
3239 RunExit: Cannot locate file to run
3243 Ole Object: Object Name too long
3244 Ole Object: Property name too long
3245 Ole Object: Method name too long
3246 Ole Object: Object does not exist
3247 Ole Object: Method has more parameters then WIL supports

3250 Ole Object: Problem occurred when formatting parameters
3252 RunExit: Not supported under Windows NT (No DOS)
3320 EvalDisk: Disk Drive specification error
3321 EvalDisk: Internal Error 1
3322 File Name Parsing: Function does not allow wildcards
3325 BinaryPeekPoke: Offset is beyond end of binary buffer
3326 BinaryPoke: Value to poke is outside the 8 bit range
3327 BinaryCopy: Offset(s) beyond end of binary buffer(s)
3328 BinaryCopy: Data to be copied will not fit in buffer
3329 BinaryCopy: Data to copy extends beyond end of buffer
3333 BinaryIndex: Offset is beyond end of binary buffer
3338 FP Math: Illegal floating point number. Too many dots.
3339 FP Math: Illegal floating point number. Too many E's.
3340 FP Math: Variable could not be converted to floating point
3352 Internal Error: Command or Structop not defined
3353 Struct Error: Nesting of structures is too complex
3354 END Error: No matching End found
3355 STRUCT Error: 'Break' not in a Structure
3356 STRUCT Error: 'Continue' not in a Structure
3357 End Error: No match found
3358 Else Error: No matching If found
3359 Break/Continue: Not in a While, Switch, or For structure
3360 For Error: Bad Syntax. e.g. 'For x = 1 to 10'
3361 For Error: For counters must be numbers, not strings
3362 Misplaced 'TO' found without a DoFor
3363 Unidentified 'END': Must be followed by If, While, Switch or For
3364 SWITCH/CASE Error: Switch/Case can only accept integers
3365 CASE Error: No matching Switch found
3366 AskBox Error: Single/Multiple value incorrect
3367 AskBox Error: Sorted/Unsorted value incorrect
3382 DIICall: Internal Error - cannot accept lpbinary return
3383 Execute function error: 'Wait' parameter bad
3384 Execute function error: 'Display type' parameter bad
3385 DiskExist: Invalid Disk Argument. Try a single letter
3386 Iniltemize: Null section name not valid in NT
3387 ShellPrint: @WAIT not supported in NT
3391 DllFree: Bad Dll handle passed. Must use handle returned by DllLoad
3393 AddExtender: Too many extenders added
3394 AddExtender: Extender dll not found
3395 AddExtender: Not a valid extender
3396 AddExtender: Extender table full
3397 List Itemization: Could not allocate global buffer
3401 RunEnviron: @NOWAIT not supported in 16 bit versions. Use
@WAIT.
3437 AddExtender: Extender DLL load failed (make sure 16/32-bit type
matches WIL)
3467 FileWrite: Write failed
3472 Function not available in this product
3473 FileCopy/Move: Cannot specify warning-@TRUE in background
mode.

yesTRUEC&opy&Printnono&AboutyesyesyesyesWIL L-Z Jump File
WILLZyes06/11/98

Table of Contents

[LastError](#)
[Log10](#)
[LogDisk](#)
[LogE](#)
[Max](#)
[MenuChange {*M}](#)
[Message](#)
[Min](#)
[MouseClicked](#)
[MouseClickedBtn](#)
[MouseInfo](#)
[MouseMove](#)
[MsgTextGet](#)
[Net101](#)
[NetInfo](#)
[Num2Char](#)
[Object101, Ole 2.0, and Applications](#)
[ObjectClose](#)
[ObjectOpen](#)
[Operators](#)
[ParseData](#)
[Pause](#)
[PlayMedia](#)
[PlayMidi](#)
[PlayWaveForm](#)
[Print](#)
[Random](#)
[Registration Database Operations](#)
[RegApp {*32}](#)
[RegCloseKey](#)
[RegConnect](#)
[RegCreateKey](#)
[RegDeleteKey](#)
[RegDelValue {*32}](#)
[RegEntryType](#)
[RegExistKey](#)
[RegExistValue](#)
[RegLoadHive](#)
[RegOpenKey](#)
[RegQueryBin {*32}](#)
[RegQueryDword {*32}](#)
[RegQueryEx](#)
[RegQueryExpSz {*32}](#)
[RegQueryItem {*32}](#)
[RegQueryKey](#)
[RegQueryKeys](#)
[RegQueryMulSz {*32}](#)
[RegQueryValue](#)
[RegSetBin {*32}](#)
[RegSetDword {*32}](#)
[RegSetEx](#)
[RegSetExpSz {*32}](#)
[RegSetMulSz {*32}](#)
[RegSetValue](#)
[RegUnloadHive](#)
[Reload {*M}](#)
[Return](#)
[Run](#)
[RunEnviron](#)
[RunHide](#)
[RunHideWait](#)
[RunIcon](#)
[RunIconWait](#)
[RunShell](#)
[RunWait](#)
[RunZoom](#)
[RunZoomWait](#)

Select
SendKey
SendKeysChild
SendKeysTo
SendMenusTo
ShellExecute
ShortcutDir
ShortcutEdit
ShortcutExtra {*Explorer}
ShortcutInfo {*Explorer}
ShortcutMake {*Explorer}
Sin
Sinh
SnapShot
Sounds
Sqrt
StrCat
StrCharCount
StrCmp
StrFill
StrFix
StrFixChars
StrFixLeft
StrFixCharsL
StriCmp
StrIndex
StrIndexNc
StrIndexWild
StrLen
StrLenWild
StrLower
StrReplace
StrScan
StrSub
StrSubWild
StrTrim
StrUpper
Switch
Tan
Tanh
Terminate
TextBox
TextBoxSort
TextSelect
TimeFunctions
TimeAdd
TimeDate
TimeDayofWeek
TimeDayofYear
TimeDelay
TimeDiff
TimeDiffDays
TimeDiffSecs
TimeJulianDay
TimeJulToYmd
TimeSubtract
TimeWait
TimeYmdHms
Version
VersionDLL
WaitForKey
WallPaper
While
WinActivate
WinActivChild
WinArrange
WinClose
WinExeName
WinExist
WinExistChild

WinGetActive
WinHelp
WinHide
WinIconize
WinIdGet
WinIsDOS
WinItemChild
WinItemize
WinItemizeEx
WinItemNameId
WinMetrics
WinName
WinParmGet
WinParmSet
WinPlace
WinPlaceGet
WinPlaceSet
WinPosition
WinResources
WinShow
WinState
WinSysInfo() { *32 }
WinTitle
WinVersion
WinWaitChild
WinWaitClose
WinWaitExist
WinZoom
Yield
Yields
ObjectAccess
WinItemProclD

LastError

Returns the most-recent error encountered during the current WIL program.

Syntax:

```
LastError ( )
```

Parameters:

(none)

Returns:

(i) 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 WIL program 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 WIL program is in Appendix B.

Of course, if you use the **LastError** function to trap errors, then extensive script testing -- including all error conditions -- is highly recommended.

Example:

```
ErrorMode (@OFF)
LastError ( )
FileCopy("data.dat", "c:\backups\*.*", @FALSE)
ErrorMode (@CANCEL)
If LastError( ) == 1006
    Message("Error", "Please call Tech Support at 555-9999.")
endif
Message("LastError", " Done.")
```

See Also:

[Debug](#), [ErrorMode](#)

Log10

Calculates the base-10 logarithm.

Syntax:

`Log10(x)`

Parameters:

(f) *x* floating point number.

Returns:

(f) the logarithm of the argument .

The **Log10** function calculates the base-10 logarithm of the argument. If the argument is negative or zero, an error will occur.

Example:

```
a = Log10(123.45)
Message("Base-10 log of 123.45 is", a)
```

See Also:

[Loge](#), [Exp](#), [operator **](#)

LogDisk

Logs (activates) a disk drive.

Syntax:

```
LogDisk (drive-letter)
```

Parameters:

(s) drive-letter the disk drive to log into.

Returns:

(i) **@TRUE** if the current drive was changed.

Use this function to change to a different disk drive.

Example:

```
LogDisk("C:")  
Message("LogDisk","Disk Logged.")
```

See Also:

[DirChange](#), [DiskScan](#)

LogE

Calculates the natural logarithm.

Syntax:

LogE(x)

Parameters:

(f) x floating point number.

Returns:

(f) the logarithm of the argument x.

The **LogE** function calculates the natural logarithm of the argument. If the argument is negative or zero, an error will occur.

Example:

```
a = LogE(123.45)
Message("Natural log of 123.45 is", a)
```

See Also:

[Log10](#), [Exp](#), [operator **](#)

Max

Returns largest number in a list of numbers.

Syntax:

Max (number [, number...])

Parameters:

(f) number floating point number(s).

Returns:

(f) largest number.

Use this function to determine the largest of a set of comma-delimited numbers.

Example:

```
a = Max(5, -3.57, 125, 34E3, 2345.12, -32767)
Message("Largest number is", a)
```

See Also:

[Abs](#), [Average](#), [Min](#), [Random](#)

MenuChange {*M}

Checks, unchecks, enables, or disables a menu item.

Syntax:

MenuChange (menuname, flags)

Parameters:

(s) menuname menu item whose status you wish to change.
(s) flags **@CHECK, @UNCHECK, @ENABLE, or @DISABLE.**

Returns:

(i) always 1.

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.

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.

Example:

```
; For menu based versions of WIL only  
  
; Assuming the application has a "File" "Print"  
; menu item, this code will disable it  
; Disable Print  
    MenuChange("FilePrint", @Disable)
```

See Also:

[IsMenuChecked](#), [IsMenuEnabled](#)

Message

Displays a message to the user.

Syntax:

Message (title, text)

Parameters:

(s) title title of the message box.
(s) text text to display in the message box.

Returns:

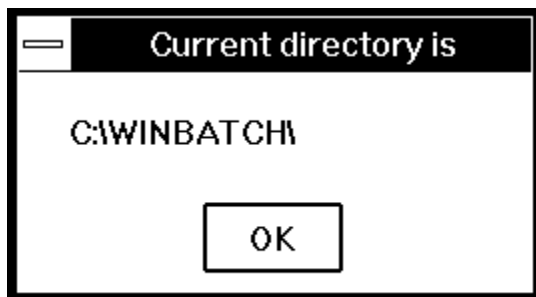
(i) always 1.

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( ))
```

which produces:



See Also:

[Display](#), [Pause](#)

Min

Returns lowest number in a list of numbers.

Syntax:

Min (number [, number...])

Parameters:

(f) number floating point number(s).

Returns:

(f) lowest number.

Use this function to determine the lowest of a set of comma-delimited numbers.

Example:

```
a = Min( 5, -37.5, 125, 34.26, 2345E4, -32767)
Message("Smallest number is", a)
```

See Also:

[Abs](#), [Average](#), [Max](#), [Random](#)

MouseClicked

Clicks mouse button(s).

Syntax:

MouseClicked(click-type, modifiers)

Parameters:

- (i) click-type a mouse button press.
- (i) modifiers click variations for mouse button presses.

Returns:

- (i) **@TRUE** on success.

This function performs a mouse click at the current cursor position.

"Modifiers" can be set to 0 if none are needed.

Click-types:

@LCLICK	left click
@RCLICK	right click
@MCLICK	middle click
@LDBLCLICK	left double-click
@RDBLCLICK	right double-click
@MDBLCLICK	middle double-click

Modifiers (multiple modifiers can be linked together with a logical OR, "|"):

@SHIFT	hold down shift key
@CTRL	hold down control key
@LBUTTON	hold down left mouse button
@RBUTTON	hold down right mouse button
@MBUTTON	hold down middle mouse button

NOTE: The modifiers are in effect until the click type is executed. [For example, a Shift key plus a right mouse click would be: `MouseClicked(@RCLICK, @SHIFT)`]

Example:

```
Run("notepad.exe", "")
Message("Notepad Running", "We will now close it via a mouseclick on the close
utton")
winpos = WinPlaceGet(@NORMAL, "~Notepad")
; get coordinates for upper right corner of window
x = ItemExtract(3, winpos, " ") - 10
y = ItemExtract(2, winpos, " ") + 10
WinActivate("~Notepad")
MouseMove(x - 10, y + 10, "", "")
MouseClicked(@LCLICK, 0)
```

See Also:

[MouseClickedBtn](#), [MouseMove](#)
[MouseInfo](#), [SendKey](#)

MouseClickedBtn

Clicks on the specified button control.

Syntax:

```
MouseClickedBtn(parent-windowname, child-windowname, button-text)
```

Parameters:

(s) parent-windowname	the initial part of, or an entire parent window name.
(s) child-windowname	the initial part or, or an entire child window name.
(s) button-text	text specifying a button control.

Returns:

(i) **@TRUE** on success.

This function clicks on the pushbutton, radio button, or checkbox whose text is specified by "button-text".

If the button is located within a top-level window, specify the window name in "parent-windowname" and specify a blank string ("") for "child-windowname".

If the button is located within a child window, specify the top-level window name in "parent-windowname" and the child window name in "child-windowname".

Example:

```
SendMenusTo("Exploring", "Tools | Map Network Drive")
TimeDelay(1)
MouseClickedBtn("Map Network Drive", "", "Reconnect at logon")
Message("MouseClickedBtn", "Reconnect at logon button clicked")
```

See Also:

[MouseClicked](#), [MouseMove](#)
[MouseInfo](#), [SendKey](#)

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:		
	<u>Binary</u>	<u>Decimal</u>	<u>Meaning</u>
	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
5	returns mouse coordinates relative to the client area of the window under the cursor, in virtual (1000x1000) screen units.		
6	returns mouse coordinates relative to the client area of the window under the cursor, in virtual (1000x1000) client units.		

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

0	"Clock"
1	"Clock"
2	"500 500"
3	"320 240"
4	"4"

Example:

```
Display(1, "", "Press a mouse button to continue")
buttons = 0
while buttons == 0
    buttons = MouseInfo(4)
endwhile
If buttons & 4
    Display(1, "", "Left button was pressed")
endif
If buttons & 1
    Display(1, "", "Right button was pressed")
endif
```

See Also:

[WinMetrics](#), [WinParmGet](#)
[MouseClick](#), [MouseClickBtn](#), [MouseMove](#)

MouseMove

Moves the mouse to the specified X-Y coordinates.

Syntax:

```
MouseMove(X, Y, parent-windowname, child-windowname)
```

Parameters:

- | | |
|-----------------------|---|
| (i) X | integer specifying the coordinate X. |
| (i) Y | integer specifying the coordinate Y. |
| (s) parent-windowname | the initial part of, or an entire parent window name. |
| (s) child-windowname | the initial part of, or an entire child window name. |

Returns:

- | | |
|-----|--------------------------|
| (i) | @TRUE on success. |
|-----|--------------------------|

If "parent-windowname" specifies a top-level window and "child-windowname" is a blank string, the specified X-Y coordinates are relative to "parent-windowname".

If "parent-windowname" specifies a top-level window and "child-windowname" specifies a child window of "parent-windowname", the specified X-Y coordinates are relative to "child-windowname".

If "parent-windowname" and "child-windowname" are both blank strings, the specified X-Y coordinates are relative to the Windows desktop.

All coordinates are based on a virtual 1000 x 1000 screen.

Example:

```
MouseMove(335, 110, "", "")
```

See Also:

[MouseDown](#), [MouseDownBtn](#),
[MouseDownInfo](#), [SendKey](#)

MsgTextGet

Returns the contents of a Windows message box.

Syntax:

```
MsgTextGet(window-name)
```

Parameters:

(s) window-name full title of the message box window.

Returns:

(s) contents of the message box.

This function returns the text contents of a standard Windows message box. "Window-name" must be the full title of the message box window, and is case-sensitive.

Note1: This function may not work with the types of message boxes created by the application you wish to control if it is not a standard Windows Message box. However, if this function does work, it is the easiest way to keep tabs on an application.

Note2: This function will not work with the types of message boxes created by most **WIL** functions, since they are not standard Windows message boxes.

Example:

```
msg = MsgTextGet("Microsoft Word")
If msg == "Search text not found"
    SendKey("~")
endif
Message("MsgTextGet", " Message Text Gotten.")
```

See Also:

[WinGetActive](#)

Net101

All network functionality for WIL is performed via "WIL Extenders", add-on DIIs for WIL, which contain Network commands for assorted networks.

NetInfo is the only WIL network function. It returns the types of the networks currently active on the local machine, and can be used to help determine which network extenders should be loaded in multi-network environments.

Documentation for the various network extenders are found either in a manual for a particular extender or in an associated disk file.

See Also:

[AddExtender](#), [DIICall](#), [NetInfo](#)

NetInfo

Determines network(s) installed.

Syntax:

NetInfo(requestcode)

Parameters:

- (i) requestcode 0 for primary network name
 1 for secondary subnet list

Returns:

- (s) Primary network name for request code 0, or
 Secondary network list for request code 1.

Use this function to determine the network type(s) running on a workstation. When running in a mixed network environment, it may be important to be able to determine the types of networks running on a workstation so as to be able to load the appropriate network extender DLLs and issue the corresponding commands.

NetInfo(0) will return then name of the primary network, or will return "**MULTINET**" , which indicates the Windows multinet driver is active and the secondary subnet list should be queried. **NetInfo(0)** will return one of the following strings:

NetInfo(0) return values:

NONE	No network installed
MULTINET	Multinet driver installed, see subnet codes.
MSNET	Microsoft Network
LANMAN	LAN Manager
NETWARE	Novell NetWare
VINES	Banyan Vines
10NET	10 Net
LOCUS	Locus
SUNPCNFS	SUN PC NFS
LANSTEP	LAN Step
9TILES	9 Tiles
LANTASTIC	Lantastic
AS400	IBM AS/400
FTPNFS	FTP NFS
PATHWORK	DEC PathWorks
OTHER1	Other (code 1)
OTHER2	Other(code 2)
UNKNOWN	Other (unknown)

If **NetInfo(0)** returned "**MULTINET**" then **NetInfo(1)** will return one or more of the following in a space delimited list:

NetInfo(1) return values:

NONE	No networks active
MSNET	Microsoft Network
LANMAN	LAN Manager
WINNET	Windows Network (Windows for Workgroups, etc)
NETWARE	Novell Netware
VINES	Banyan Vines
OTHER2	Other (code 0x20)
OTHER4	Other (code 0x40)
OTHER8	Other (code 0x80)

32 Bit Windows

NetInfo(0) will always return the string "WINNT" for 32 bit Windows platforms, regardless of whether the platform is Windows95 or Windows NT.

Under Windows 95, NetInfo(1) will return a list of installed network client ID's, delimited with the standard file delimiter (by default, a tab).

Possible client ID's, with their corresponding descriptions, are:

<u>Client ID</u>	Description
3OPEN	3Com 3+Open (all versions)
3SHARE	3Com 3+Share (all versions)
DLR	IBM OS/2 LAN Server (versions below 1.2)
DLR12	IBM OS/2 LAN Server (version 1.2)
DLR13	IBM OS/2 LAN Server (versions 1.2, 1.3, and 1.2 without /API)
DLR13CSD	IBM OS/2 LAN Server (version 1.3 CSD 5015/5050)
DLR20	IBM OS/2 LAN Server (version 2.0)
FTPNFS	FTP Software NFS Client (InterDrive 95)
LANMAN	Microsoft Real Mode LAN Manager
LANT5	Artisoft LANtastic (version 5.X and above)
MSNET	Real mode MS-Net Compatible
NETWARE3	Novell NetWare (Workstation Shell 3.X [NETX])
NETWARE4	Novell NetWare (Workstation Shell 4.0 and above [VLM])
NOVELL32	Novell NetWare Client 32
NWREDIR	Client for NetWare Networks
PATHWKS	DEC PATHWORKS (versions below 4.0)
PATHWKS40	DEC PATHWORKS (version 4.x)
PCLP	IBM PC LAN Program (all versions)
PCNFS50	SunSoft PC-NFS (version 5.0)
VINES552	Banyan DOS/Windows 3.1 client
VREDIR	Client for Microsoft Networks

Under Windows NT, NetInfo(1) will return a list of installed network provider ID's, delimited with the standard file delimiter (by default, a tab).

Possible providers, with their corresponding descriptions, are:

<u>Provider ID</u>	Description
LanmanWorkstation	Microsoft Windows Network
NetWareWorkstation	NetWare Services
NWCWorkstation	NetWare or Compatible Network

Example:

```
a=NetInfo(0)
if a=="MULTINET"
    b=NetInfo(1)
    count=ItemCount(b," ")
    Message("Multinet supporting %count% networks", b)
else
    Message("Installed Network", a)
endif
```

See Also:

[AddExtender](#), [DIICall](#), [Net101](#)

Num2Char

Converts a number to its character equivalent.

Syntax:

Num2Char (integer)

Parameters:

(i) number any number from 0 to 255.

Returns:

(s) one-byte string containing the character which 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](#), [IsNumber](#)

Object101, Ole 2.0, and Applications

The ability to control and assist the movement of data between applications is one of the key strengths of WIL. In early versions of WIL, the Clipboard and **SendKey** functions were the only way to transfer data. More recently, dynamic: data exchange (DDE) support allowed both the transfer of data to and the control of other applications.

Now, with support for OLE Automation, you can do much more than share data. From within your WIL script, you can access and manipulate OLE objects that are supplied by other applications. With OLE Automation, you can use WIL to produce custom solutions that utilize data and features from applications that support OLE Automation.

What Is OLE Automation?

OLE Automation is an industry standard that applications use to expose their OLE objects to development tools, macro languages, and container applications that support OLE Automation. For example, a spreadsheet application may expose a worksheet, chart, cell, or range of cells -- all as different types of objects. A word processor might expose objects such as applications, paragraphs, sentences, bookmarks, or selections.

When an application supports OLE Automation, the objects it exposes can be accessed by WIL. You use WIL scripts to manipulate these objects by invoking methods (subroutines) on the objects, or by getting and setting the objects' properties (values).

Accessing OLE Objects

You can manipulate other applications' OLE objects directly by first opening the object with the **ObjectOpen** function. The **ObjectOpen** function is used to open the object. This function requires a single parameter -- a string that indicates the application name and the type of object you want to create. Use the following syntax to specify an object to create:

Application.ObjectType

For example, let's say there is a orgchart application named ORGCHART.EXE that supports a single object: an orgchart. Furthermore, the OrgChart object supports two sub-objects: a box and a line. The object might be defined as:

```
OrgChart.Chart
```

Once you know the type of object you want to create, you use the **ObjectOpen** function to create the object. Set the value returned by the **ObjectOpen** function to a variable. Here's an example:

```
MyChart = ObjectOpen("OrgChart.Chart")
```

Once you have the primary object in hand - in the MyChart variable in this case, you can create the sub-objects and assign them to their own variables.

```
TopBox = MyChart.NewBox  
BottomBox = MyChart.NewBox  
TheLine = MyChart.NewLine
```

When this code executes, the application providing the object is started (if it is not already running) and an object is created. The object belongs to the application that created it. This object can be referenced in WIL scripts using the variable you placed the return value of the **ObjectOpen** function into. For example, after creating the object, you could write code such as this to open sub-objects, change the background color, set a default font, set a title, and save the object to a file:

```

MyChart.Color = "White"
MyChart.FontName = "Arial"
MyChart.FontSize = 12
MyChart.Title = "Tinas Org Chart"
;
TopBox.Position(2,2)
TopBox.Text = "The Boss"
BottomBox.Position(2,8)
BottomBox.Text = "Tina"
;
TheLine.Begin(2,2)
TheLine.End(2,8)
;
MyChart.SaveAs("C:\ORGCHART\TINA.ORG")

```

When you are through with an object, use **ObjectClose** to tell the WIL processor that you are done with the object.

```

ObjectClose(TheLine)
ObjectClose(TopBox)
ObjectClose(BottomBox)
ObjectClose(MyChart)

```

Note1: To get a list of objects that an application supports, you must consult that application's documentation. It may also help to poke around in the Windows registration database. Be aware, though, that intentional, unintentional, or accidental changes to the registration database may completely destroy a Windows installation and require a complete re-installation of ALL your software to recover.

Note2: When creating an object, some applications require that the application providing the object is either active or on the system's path.

Accessing an Object's Properties

To assign a value to a property of an object, put the object variable and property name on the left side of an equation and the desired property setting on the right side. For example:

```

MyChart.Title = "Tinas Org Chart"

```

You can also retrieve property values from an object:

```

TheTitle = MyChart.Title

```

Performing Object Methods

In addition to getting and setting properties, you can manipulate an object using the methods it supports. Some methods may return a value.

```

MyChart.Position(2,2)
TheLine.End(2,8)
MyChart.SaveAs("C:\ORGCHART\TINA.ORG")
;
a= MyChart.Print( )
if a == @FALSE
    Message("Error", "Print of MyChart failed")
endif

```

Methods that do not return a value return 0.

Sub-Objects

Some objects contain sub-objects. For example, a box is a sub-object of an orgchart object. You cannot include multiple objects, properties, and methods on the same line of code. Each object must have its own variable. For example:

```
TopBox = MyChart.NewBox
```

Closing an Object

All OLE Automation objects support some method that closes the object and the application that created it. Since OLE objects can use a significant amount of memory, it is a good idea to explicitly close an object when you no longer need it. To close an object, use the appropriate method (most objects support the Close method or the Quit method). For example:

```
Closes the object.  
MyChart.Close  
Closes the application that created the object.  
MyChart.Quit
```

When WIL processing for an object is complete, use the **ObjectClose** function to free WIL processor memory.

```
ObjectClose("MyChart")
```

Note3: The **ObjectClose** function will suggest to the application that owns the object that its services are no longer required, and that, if it has nothing better to do, it might as well close up shop and exit. For these applications, the "MyChart.Quit" as shown above is not required.

OLE 2.0 Limitations in WIL

Some OLE objects support features that can't be accessed using WIL. This section discusses known limitations.

Arrays

Some objects have properties and methods that return an array of data or take an array as an argument. WIL cannot process these types of properties or methods.

Named Arguments

You cannot use named arguments when calling an object's methods in WIL. You must specify each argument in the correct order. If you want to omit an optional argument, leave it blank.

ObjectAccess

Opens or creates an OLE 2.0 Automation object.

Syntax:

ObjectAccess(app.objname, create-flag)

Parameters:

- (s) app.objname name of the desired object.
- (i) create-flag @TRUE, it will create an instance of the object, and return a handle to it. @FALSE, it will fail and return an error.

Returns:

- (i) a special object handle to be used when referring to the object. See discussion in Object101 section.

If there is already a running instance of the object specified by "app.objname", this function will return a handle to that object.

If there is not a running instance of the specified object, the behavior will depend on the value of 'create-flag':

If 'create-flag' == @TRUE, it will create an instance of the object, and return a handle to it.

If 'create-flag' == @FALSE, it will fail and return an error.

This function is similar to ObjectOpen, except that ObjectOpen always creates a new object and cannot be used to access an existing one.

See Object101 section for information on OLE Automation.

See Also:

[Object101, Ole 2.0, and Applications](#)

ObjectClose

Closes an OLE 2.0 Automation object

Syntax:

```
ObjectClose(objecthandle)
```

Parameters:

(i) objecthandle handle of object to close.

Returns:

(i) **@TRUE** (always)

The **ObjectClose** function closes an object and frees WIL processor memory. The parameter passed to **ObjectClose** must be the same variable that the return value from the corresponding **ObjectOpen** was placed into. Otherwise the function will fail.

Example:

```
MyChart = ObjectOpen("OrgChart.Chart")
a = MyChart.Load("C:\ORGCHART\TINA.ORG")
if a == @TRUE
    MyChart.Print
endif
ObjectClose(MyChart)
```

See Also:

[Object101](#), [ObjectOpen](#)

ObjectOpen

Opens or creates an OLE 2.0 Automation object

Syntax:

ObjectOpen(app.objname)

Parameters:

(s) app.objname name of the desired object.

Returns:

(i) a special object handle to be used when referring to the object. See discussion in Object101 section.

The **ObjectOpen** function returns a handle to be used when referring to an OLE 2.0 Automation object. If the Object does not exist, the function will fail.

Example:

```

; Note: This example shows how to use Ole Automation to
; drive Netscape to obtain a web page. However, if your
; intent is to obtain web pages, a more suitable approach
; would be to try our Internet extender which can be
; downloaded from our Web site.
;
BoxOpen("Object Open Example","Initializing")
url = "http://www.windowware.com/index.html"
bufsize = 100000

ErrorMode(@off)
LastError( )
web = ObjectOpen("Netscape.Network.1")
err=LastError( )
ErrorMode(@cancel)
if err!=0
    Message("Netscape not found","Cannot run this sample program")
    exit
endif

BoxText("Opening netscape Object")
TimeDelay(2)
outfile=strcat(DirGet( ),"oletest.html")
bigbuf=BinaryAlloc(bufsize)
BinaryOleType(bigbuf,202,0,0,0)

rc = web.Open(url, 0, 0, 0, 0)

BoxText(strcat("Retrieving url contents of",@crlf,url))

if rc!=0
    While !(web.IsFinished)
        contsize=web.GetContentLength
        if contsize==0
            TimeDelay(1)
        else
            getnowsize=min(bufsize,contsize)
            gotsize=web.Read(bigbuf,getnowsize)
            BinaryEODSet(bigbuf,gotsize)
            break
        endif
    endwhile
    BinaryWrite(bigbuf,outfile)

else
    Message("Uh-oh", "web.Open failed")
endif

web.Close

BoxText("Complete")
Message("OLE Test", strcat("Finished",@crlf,"Data saved in",@crlf,outfile))

```

See Also:

[Object101](#), [ObjectClose](#)

Operators

Windows Interface Language Help File 

Windows Interface Language operators take one operand ("unary operators") or two operands ("binary operators").

Unary operators (integers and floating point numbers)

- Arithmetic Negation (Two's complement)
- + Identity (Unary plus)

Unary operators (integers only)

- ~ 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 logical operators (integers only)

- << Left Shift
- >> Right Shift
- & Bitwise And
- | Bitwise Or
- ^ Bitwise Exclusive Or (XOR)
- && Logical And
- || Logical Or

Binary arithmetic operators (integers and floating point numbers)

- ** Exponentiation
- Multiplication
- / Division
- mod Modulo
- Addition
- Subtraction

Binary relational operators

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

Assignment operator

- = Assigns evaluated result of an expression to a variable

ParseData

Parses the passed string.

Syntax:

ParseData (string)

Parameters:

(s) string string to be parsed.

Returns:

(i) 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)
Message("ParseData", " Done.")
```

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:

(s) title title of pause box.
(s) text text of the message to be displayed.

Returns:

(i) always 1.

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:")
```

which produces:



See Also:

[Display](#), [Exit](#), [Message](#), [Terminate](#)

PlayMedia

Controls multimedia devices.

Syntax:

PlayMedia (mci-string)

Parameters:

(s) mci-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 MIDI 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
; drive, from start to finish
stat = PlayMedia("status cdaudio mode")
answer = 1
If stat == "playing"
    answer = AskYesNo("CD Audio", "CD is Playing. Stop?")
    If answer == 0 Then Exit
endif
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:

[Beep](#), [PlayMidi](#), [PlayWaveForm](#), [Sounds](#)

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.

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("C:\windows\media\canyon.mid", 1)  
Message("PlayMidi","Midi Played")
```

See Also:

[Beep](#), [PlayMedia](#), [PlayWaveForm](#), [Sounds](#)

PlayWaveForm

Plays a WAV sound file.

Syntax:

```
PlayWaveForm (filename, mode)
```

Parameters:

(s) filename name of the WAV sound file.
(i) mode play mode (see below).

Returns:

(i) **@TRUE** if successful.

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.

Examples:

```
PlayWaveForm("tada.wav", 0)
```

```
PlayWaveForm("SystemDefault", 1 | 16)  
Message("PlayWaveForm", " Done.")
```

See Also:

[Beep](#), [PlayMedia](#), [PlayMidi](#), [Sounds](#)

Print

Instructs the application responsible for a file to print the file on the default printer.

Syntax:

Print(data file, directory, display mode, waitflag)

Parameters:

- | | |
|------------------|--|
| (s) data file | the name of the file to print. |
| (s) directory | current working directory (if applicable). |
| (i) display mode | @NORMAL, @ICON, @ZOOMED, @HIDDEN. |
| (i) waitflag | @WAIT, @NOWAIT. |

Returns:

- (i) **@TRUE** if the function completed.

Instructs the application responsible for a file to print the file on the default printer. The Windows ShellExecute API is used. It examines the extension of the data file, looks the extension up in the Windows registry to determine the owning application, starts the owning application, and instructs it, also according to data specified in the registry, to print the data file. Most applications will send the printout to the default printer, however the exact action taken by the application is under the applications own control.

Applications that support this command or their setup programs will generally make the necessary modifications to the Windows registry to allow this function to perform successfully.

Note: The **@WAIT** parameter is not supported in 32 bit versions of this product.

Example:

```
FileCopy("C:\config.sys", "xxx.txt", 0)
a=Print("xxx.txt", DirGet( ), @NORMAL, @WAIT)
FileDelete("xxx.txt")
```

See Also:

[RunShell](#)

Random

Computes a pseudo-random number.

Syntax:

Random (max)

Parameters:

(i) max largest desired integer number.

Returns:

(i) 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)
```

See Also:

[Average](#), [Max](#), [Min](#)

Registration Database Operations

In the early days of Windows, there was a single INI file, WIN.INI. As Windows advanced, the WIN.INI file became cluttered, and it was then subdivided into SYSTEM.INI, WIN.INI and a large number of application specific INI files.

With the advent of OLE, Windows NT, and other advancements in operating system technology, the simple INI files could not hold or organize the new and vast amounts of information required to run a modern operating system. For this reason, a new data storage structure was developed. Sometimes called the Registry or the Registration Database, this new file was designed to be able to hold and organize large amounts of seeming random information.

The Registration Database is organized in a tree structure, much like a file system. At every level "keys" to the data exist. The keys are analogous to the sub-directories in a file system. A set of keys to a data item look very much like a path to a filename.

In Windows, the Registration Database may be viewed and altered with the "RegEdit" utility. It requires a "/v" parameter, as in "regedit.exe /v", to enable the edit mode of the utility. In Windows NT and Windows 95, there exists the "RegEdt32" utility that allows access to the Registration Database. Neither of these utilities can be found in the normally installed Program Manager groups, and must be ferreted out on your own.

CAUTION: The reason that these utilities are not made easily accessible is that it is trivially easy to make a modification to the database that will completely ruin a Windows, Windows 95 or Windows NT installation, and may require a complete re-install of the Windows version to get the system running again. It is best to study the database and understand what is going on, instead of perhaps using a somewhat common "trial and error" method of making changes.

There are two ways to query and set information in the Registration Database. The easy way is to simply base all operations on an always open **root** key. Using just the **RegQueryValue** and **RegSetValue** functions you can alter all data associated with pre-defined keys.

The other more complicated and more flexible method is to open or create a desired key, using the **RegOpenKey** or **RegCreateKey** functions, modify the database with other registration functions, passing it a handle to the key, and then finally close the database with the **RegCloseKey** function.

Most of the registration functions accept both a handle to a key and a subkey string which further defines a lower key. Oftentimes the subkey string is simply set to null (empty quotes), and the handle points directly to the destination. At other times, one of the pre-defined roots of the database is passed as the handle and the subkey string points all the way down to the desired data item.

Pre-defined keys are provided. Windows has a single root key that is always open. Its handle can be accessed via the built-in WIL constant **@REGROOT**. Windows NT and Windows 95 provides several keys, as shown in the table below:

Windows handles to always open keys

@REGROOT	Root of the Registration Database.
----------	------------------------------------

32 bit Windows handles to always open keys

@REGMACHINE	Root of the machine section of the Registration Database.
@REGCLASSES	Shortcut to the classes sub-section.
@REGUSERS	Root of the user section of the Registration Database.

@REGCURRENT Shortcut to the current users
sub-section.

Note: Windows NT and Windows 95 added named data types to the registration database entries. As a result there is a special way to access the named data entries in Windows NT and Windows 95 registration databases. The steps are as follows:

- 1) Open a key pointing to the group of data items that contains the desired data item.
- 2) Use the RegSetValue or the RegQueryValue functions to access the data value. The "subkey-string" must contain only the data item name enclosed in square brackets.
- 3) Be sure to close the key when operations are complete.

For example, here is a WIL script which modifies the default printer in Windows NT.

```
newprt = "LJ3,winspool,LPT1:" ;the printer you want to assign as the
                                ;default
regkey = RegOpenKey(@REGCURRENT, "Software\Microsoft\Windows
                                NT\CurrentVersion\Windows")
defprt = RegQueryValue(regkey, "[Device]")
Message("Previous Default printer", defprt)
RegSetValue(regkey, "[Device]", newprt)
defprt = RegQueryValue(regkey, "[Device]")
Message("New Default printer", defprt)
RegCloseKey(regkey)
```

RegApp {*32}

Creates registry entries for a program under "App Paths".

Syntax:

```
RegApp(program-name, path)
```

Parameters:

- (s) program-name the name of an executable program (EXE), optionally containing a path.
- (s) path optional desired "PATH" setting for the specified program.

Returns:

- (i) **@TRUE** Entry was created.

This function creates (or updates) a sub-key in the registration database for the specified program, of the form PROGRAMNAME.EXE, under the key:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\App Paths\
```

If "program-name" does not contain a path, the function will search for it on the path.

The function creates a "(Default)" value for the key, containing the full path to the specified program.

If the "path" parameter is not a blank string (""), the function also creates a "Path" value for the key. This should contain one or more directories (separated by semi-colons) which you want to be prepended to the existing "PATH" environment variable when the program is run.

Example:

```
residence="c:\msoffice\excel.exe"  
path="c:\msoffice\excel;c:\msoffice\winword"
```

```
RegApp(residence, path)  
Message("Excel.EXE", "Registered at %residence% and use path %path%")
```

See Also:

[InstallFile](#), [RegOpenKey](#), [RegCloseKey](#), [RegDeleteKey](#), [RegSetValue](#), [RegQueryValue](#), [RegQueryKey](#), and the section on [Registration Database Operations](#)

RegCloseKey

Closes a key to the Registration Database.

Syntax:

```
RegCloseKey(handle)
```

Parameters:

(i) handle handle to a registration database key.

Returns:

(i) **@TRUE** Database was closed.

The **RegCloseKey** function closes a key to the Registration Database. The key is opened or created with the **RegOpenKey** or the **RegCreateKey** functions. Registration Database changes made using a key are saved when the key is closed.

Example:

```
key=RegOpenkey(@RegClasses, "txtfile")
b=RegQueryValue(key, "shell\open\command")
RegCloseKey(key)
Message("Default textfile editor is", b)
```

See Also:

[RegOpenKey](#), [RegCreateKey](#), [RegSetValue](#), [RegQueryValue](#), and the section on [Registration Database Operations](#)

RegConnect {*32}

Connects to a predefined registry handle on another computer.

Syntax:

```
RegConnect(computer-name, handle)
```

Parameters:

(s) computer-name the name of the computer in the form of "\\computername".
(i) handle either @REGMACHINE OR @REGUSERS

Returns:

(s) a key handle to the specified handle on the remote computer.

Connects to a predefined registry handle on another computer. The key can be closed with RegCloseKey.

When a handle returned by RegConnect is no longer needed, it should be closed by calling RegCloseKey. However, this may not terminate the session. To terminate the session, call w95CancelCon or wntCancelCon with \\server\IPC\$ as the name of the resource to disconnect (where "\\server" is the name of the server).

Example:

```
Fred=RegConnect("\\FRED", @REGMACHINE)
Iver=RegQueryValue(Fred, "SOFTWARE\Microsoft\Internet Explorer[Iver]")
RegCloseKey(Fred)
Message("Internet Explorer Version Installed on \\Fred.", Iver)
```

See Also:

[RegCreateKey](#), [RegCloseKey](#), [RegDeleteKey](#), [RegSetValue](#), [RegQueryValue](#), [RegQueryKey](#), and the section on [Registration Database Operations](#)

RegCreateKey

Returns a handle to a new registration database key.

Syntax:

```
RegCreateKey(handle, subkey-string)
```

Parameters:

- (i) handle handle to a registration database key.
- (s) subkey-string a path from the key provided to the desired key.

Returns:

- (i) handle a handle to the new key.

The **RegCreateKey** function will create and open a desired key into the Registration Database. If the key already exists, **RegCreateKey** will open it. When using **RegCreateKey** you must pass a pre-existing, open key to create a new key. A pre-defined key may be used.

Example:

```
; Associate DIZ files with the default textfile editor
key=RegCreatekey(@REGCLASSES, ".diz")
RegSetValue(key, "", "txtfile")
RegClosekey(key)
Message("RegCreatekey", "*.DIZ files now associated with your text editor")
```

See Also:

[RegOpenKey](#), [RegCloseKey](#), [RegDeleteKey](#), [RegSetValue](#), [RegQueryValue](#), [RegQueryKey](#), and the section on [Registration Database Operations](#)

RegDeleteKey

Deletes a key and data items associated with the key.

Syntax:

```
RegDeleteKey(handle, subkey-string)
```

Parameters:

- (i) handle an open registration database key (see below).
- (s) subkey-string a path from the key provided to the desired key.

Returns:

- (i) **@TRUE** Key was deleted.

The **RegDeleteKey** function will delete a pre-existing key from the Registration Database. If the key does not exist, **RegDeleteKey** will fail. When using **RegDeleteKey** you must pass a pre-existing, open key to access the desired key. A pre-defined key may be used.

Example:

```
; Delete default operation for *.DIZ files
; from the registration database
ErrorMode(@off)
RegDeleteKey(@REGCLASSES, ".diz")
ErrorMode(@cancel)
Message("Regdeletekey", "*.DIZ files no longer associated with text editor")
```

See Also:

[RegOpenKey](#), [RegCreateKey](#), [RegCloseKey](#), [RegDelValue](#) and the section on [Registration Database Operations](#)

RegDelValue {*32}

Deletes a named value data item for the specified subkey from the registry.

Syntax:

```
RegDelValue(handle, subkey-string)
```

Parameters:

- (i) handle handle to a registration database key.
- (s) subkey-string a path from the key provided to the desired key.

Returns:

- (i) **@TRUE** Data item was deleted.

"Subkey-string" must be enclosed in square brackets (see **RegSetValue**). "Subkey-string" of "" deletes the "default" value.

Example:

```
;To turn Sounds ON in the case Sounds was disabled.  
ErrorMode(@off)  
RegDelValue(@REGMACHINE, "SOFTWARE\Wilson WindowWare\Settings\WWWBATCH\  
MAIN[SOUNDS]")  
ErrorMode(@cancel)  
Message("Sounds", "Sound Settings - Sounds is on.")
```

See Also:

[RegOpenKey](#), [RegCreateKey](#), [RegCloseKey](#), [RegSetValue](#), [RegQueryValue](#), and the section on [Registration Database Operations](#)

RegEntryType

Returns the type of data for the specified subkey.

Syntax:

RegEntryType(handle, subkey-string)

Parameters:

- (i) handle an open registration database key. (see below).
- (s) subkey-string a path from the key provided to the desired key.

Returns:

- (s) data type of specified subkey. (see below).

In the 16-bit version, the return value will always be 1. In the 32-bit version, the return value will be one of the following:

<u>Value</u>	<u>Meaning</u>	
0	(REG_NONE)	No value type
1	(REG_SZ)	Unicode null terminated string
2	(REG_EXPAND_SZ)	Unicode null terminated string (with environment variable references)
3	(REG_BINARY)	Free form binary
4	(REG_DWORD)	32-bit number
5	(REG_DWORD_BIG_ENDIAN)	32-bit number
6	(REG_LINK)	Symbolic Link (Unicode)
7	(REG_MULTI_SZ)	Multiple Unicode strings
8	(REG_RESOURCE_LIST)	Resource list in the resource map
9	(REG_FULL_RESOURCE_DESCRIPTOR)	Resource list in the hardware description
10	(REG_RESOURCE_REQUIREMENTS_LIST)	

Example:

```

;Registry Cloning Script
;This file traverses the Registry and copies the data items
;to a new key, finds the data item type and value and then
;resets it in the new key.
;

tot=0
totdir=0
level=1

; Standardize on tabs as file/dir delimiters
IntControl(29,@tab,0,0,0)

;;establish which registration database key is to be copied.
whol="Control Panel"
;;create a new key for the data to be copied.
there=RegCreatekey(@regcurrent,"boink")

dir1=RegOpenKey(@Regcurrent,whol)
BoxOpen("Reg Rinker","Processing top level")
gosub copydadata
sub%level%=RegQueryKeys(dir%level%)
numdir%level% = ItemCount(sub%level%, @tab)
index%level% = 0

:dsloop
If index%level% == numdir%level% Then Goto upalevel
nextlevel=level+1
index%level% = index%level% + 1
who%nextlevel%=ItemExtract(index%level%,sub%level%,@tab)

dir%nextlevel%=RegOpenKey(dir%level%,who%nextlevel%)
BoxText(strcat("Level=",level,@crlf,"Total=",tot,@crlf,dir%level%))
totdir=totdir+1
level = level + 1
gosub copydadata
sub%level%=RegQueryKeys(dir%level%)
numdir%level% = ItemCount(sub%level%, @tab)
index%level% = 0
goto dsloop

:upalevel
RegCloseKey(dir%level%)
drop(dir%level%,sub%level%,index%level%,numdir%level%)

level=level-1
if level!=0 then goto dsloop

; -----
; Termination
; -----
BoxShut ( )
;Message("%totdir% Sub-directories", "Total size %tot% bytes.")
Exit

:copydadata
curkey=dir%level% ; current source key
zork=whol
for xxx=2 to level
zork=strcat(zork,"\",who%xxx%)

```

```

next

dataitems=RegQueryItem(curkey,"")
;message(zork,who%level%)
grackle=RegCreateKey(there,zork) ;target key

count=ItemCount(dataitems,@tab)

For i = 1 to count
    data=ItemExtract(i,dataitems,@tab)
    type=RegEntryType(curkey,"[%data%]")
    a=RegQueryEx(curkey,"[%data%]",@tab,type)
    ;message(type,a)
    RegSetEx(grackle,"[%data%]",a,@tab,type)
next

;copy dataitems here
RegClosekey(grackle)
AskItemList(zork,dataitems,@tab,@sorted,@single)

tot=tot+ItemCount(dataitems,@tab)
return

```

See Also:

[RegQueryEx](#), [RegSetEx](#) and the section on Registration Database Operations

RegExistKey

Checks whether a registry key exists.

Syntax:

```
RegExistKey(handle, subkey-string)
```

Parameters:

- (i) handle handle to a registration database key.
- (s) subkey-string a path from the key provided to the desired key.

Returns:

- (i) **@TRUE** if the specified registry key exists and is accessible (ie, can be opened with RegOpenKey). Otherwise, it returns **@FALSE**.

Example:

```
key=RegExistkey(@RegClasses, "txtfile")
if key == 1
    who=RegQueryValue(@REGClasses, "txtfile\shell\open\command")
else
    message("Sorry", "The specified key does not exist!")
    break
endif
Display(3, "Default text file editor is", who)
```

See Also:

RegOpenKey

RegExistValue

Checks whether a value for a registry key exists.

Syntax:

```
RegExistValue(handle, subkey-string)
```

Parameters:

- (i) handle handle to a registration database key.
- (s) subkey-string a path from the key provided to the desired key.

Returns:

- (i) This function returns @TRUE if the specified registry value (data item) exists and is accessible (ie, can be read with RegQuery[.]). Otherwise, it returns @FALSE.

Example:

```
reg=RegExistValue(@REGMachine, "Software\Wilson
Windowware\Settings\WWWBATCH\DllUsage[WB32i]")
if reg == 0
    Message("Does the Value exist?", "No")
    exit
else
    Message("Does the Value exist?", "Yes")
endif
regkey =RegQueryValue(@REGMachine, "Software\Wilson
Windowware\Settings\WWWBATCH\DllUsage[WB32i]")
Message("Dll Usage of WinBatch", regkey)
```

See Also:

[RegQueryValue](#)

RegLoadHive {*32}

Loads a hive into a registry subkey.

Syntax:

```
RegLoadHive(handle, subkey, filename)
```

Parameters:

(i) handle	either @REGUSERS or @REGMACHINE.
(s) subkey	the name of the key to be created
(s) filename	specifies the name of a hive file.

Returns:

(s)	always 1.
-----	-----------

This function creates a subkey under HKEY_USERS or HKEY_LOCAL_MACHINE and stores registration information from the specified file into that subkey. This registration information is in the form of a hive. A hive is a discrete body of keys, subkeys, and values that is rooted at the top of the registry hierarchy.

Example:

```
RegLoadHive(@REGMACHINE, "WinWare", "winware")
```

See Also:

[RegUnloadHive](#)

RegOpenKey

Returns a handle to an existing registration database key.

Syntax:

```
RegOpenKey(handle, subkey-string)
```

Parameters:

- (i) handle handle to a registration database key.
- (s) subkey-string a path from the key provided to the desired key.

Returns:

- (i) key a handle to the new key.

The **RegOpenKey** function will open a desired key into the Registration Database. If the key does not exist, **RegOpenKey** will fail. When using **RegOpenKey** you must pass a pre-existing, open key to create a new key. A pre-defined key may be used.

Example:

```
; Find default text editor
key=RegOpenkey(@RegClasses, "txtfile")
who=RegQueryValue(key, "shell\open\command")
RegClosekey(key)
Message("Default text file editor is", who)
;
; The preceding is actually a hard way to do ...
who=RegQueryValue(@REGClasses, "txtfile\shell\open\command")
Message("Default text file editor is", who)
```

See Also:

[RegCreateKey](#), [RegCloseKey](#), [RegDeleteKey](#), [RegSetValue](#), [RegQueryValue](#), [RegQueryKey](#), and the section on [Registration Database Operations](#)

RegQueryBin {*32}

Returns binary value at subkey position.

Syntax:

```
RegQueryBin(handle, subkey-string)
```

Parameters:

- (i) handle handle to a registration database key.
- (s) subkey-string a path from the key provided to the desired key.

Returns:

- (s) contents of data item at key position desired.

The value is returned as a space-delimited string of hex bytes; e.g.:
"AB 45 3E 01".

Example:

```
value = RegQueryBin(@REGCURRENT, "Control Panel\Appearance\[CustomColors]")  
Message("CustomColors", value)
```

See Also:

[RegApp](#), [RegEntryType](#), [RegQueryEx](#), [RegQueryExpSz](#), [RegQueryDword](#), [RegQueryItem](#),
[RegQueryMulSz](#), [RegQueryValue](#), and the section on [Registration Database Operations](#)

RegQueryDword {*32}

Returns DWORD value at subkey position.

Syntax:

```
RegQueryDword(handle, subkey-string)
```

Parameters:

- (i) handle handle to a registration database key.
- (s) subkey-string a path from the key provided to the desired key.

Returns:

- (i) contents of data item at key position desired.

Example:

```
value = RegQueryDword(@REGCURRENT, "Control Panel\Desktop[ScreenSaveUsePassword] ")  
Message("ScreenSaveUsePassword", value)
```

See Also:

[RegApp](#), [RegEntryType](#), [RegQueryBin](#), [RegQueryEx](#), [RegQueryExpSz](#), [RegQueryItem](#), [RegQueryMulSz](#), [RegQueryValue](#), and the section on [Registration Database Operations](#)

RegQueryEx

Retrieves a value from the registry.

Syntax:

RegQueryEx (handle, subkey-string, delimiter, type)

Parameters:

- (i) handle an open registration database key (see below).
- (s) subkey-string a path from the key provided to the desired key.
- (s) delimiter a character to act as a delimiter between items.
"Delimiter" is ignored unless "type" is 7.
- (i) type see below.

Returns:

- (s) a value from the registry.

In the 16-bit version, "type" must be 1. In the 32-bit version, "type" can be one of the following:

Value	Meaning	
1	(REG_SZ)	Unicode null terminated string
2	(REG_EXPAND_SZ)	Unicode null terminated string (with environment variable references)
3	(REG_BINARY)	Free form binary
4	(REG_DWORD)	32-bit number
7	(REG_MULTI_SZ)	Multiple Unicode strings

Example:

See the Registry Cloning Script under RegEntryType.

See Also:

[RegApp](#), [RegEntryType](#), [RegQueryBin](#), [RegQueryExpSz](#), [RegQueryDword](#), [RegQueryItem](#), [RegQueryMulSz](#), [RegQueryValue](#), and the section on [Registration Database Operations](#)

RegQueryExpSz {*32}

Retrieves a REG_EXPAND_SZ value from the registry.

Syntax:

```
RegQueryExpSz(handle, subkey-string)
```

Parameters:

- (i) handle an open registration database key (see below).
- (s) subkey-string a path from the key provided to the desired key.

Returns:

- (i) contents of the data item at the key position desired.

Example:

```
; Note: For 32-bit Path Modifications See RegApp
; Following is location of users path in Windows NT.
userenv="Environment[path]"
sysenv="SYSTEM\CurrentControlSet\Control\Session Manager\Environment[path]"

if WinMetrics(-4)==4 ; Win NT
  lasterror( )
  ErrorMode(@off)
  oldpath=RegQueryExpSz(@RegCurrent, userenv)
  ErrorMode(@cancel)

  If lasterror( ) == 1233
    ErrorMode(@off)
    oldpath=RegQueryExpSz(@RegMachine, sysenv)
    ErrorMode(@cancel)
    if oldpath==0
      Message("Environment Variable - Users Path", "No path found")
    else
      Message("System Environment Variable - Users Path", oldpath)
    endif
  else
    Message("User Environment Variable - Users Path", oldpath)
  exit
endif

if WinMetrics(-4) !=4
  Message("RegQueryExpSz","Not supported in Non-NT versions of Windows")
  exit
endif
```

See Also:

[RegApp](#), [RegEntryType](#), [RegQueryBin](#), [RegQueryEx](#), [RegQueryDword](#), [RegQueryItem](#), [RegQueryMulSz](#), [RegQueryValue](#), and the section on [Registration Database Operations](#)

RegQueryItem {*32}

Returns a list of named data items for a subkey.

Syntax:

```
RegQueryItem(handle, subkey-string)
```

Parameters:

- (i) handle handle to a registration database key.
- (s) subkey-string a path from the key provided to the desired key.

Returns:

- (s) tab-delimited list of named data items for the specified subkey-string.

Example:

```
items = RegQueryItem(@REGCURRENT,  
                    "Software\Microsoft\Windows\CurrentVersion\Extensions")  
item = AskItemList("Select an item", items, @TAB,@SORTED,@SINGLE)  
value = RegQueryValue(@REGCURRENT,  
                      "Software\Microsoft\Windows\CurrentVersion\Extensions[%item%]")  
Message(item, value)
```

See Also:

[RegApp](#), [RegEntryType](#), [RegQueryBin](#), [RegQueryEx](#), [RegQueryExpSz](#), [RegQueryDword](#), [RegQueryMulSz](#), [RegQueryValue](#), and the section on [Registration Database Operations](#)

RegQueryKey

Returns subkeys of the specified key.

Syntax:

```
RegQueryKey(handle, index)
```

Parameters:

- (i) handle handle to a registration database key.
- (i) index zero-based index into list of subkeys.

Returns:

- (s) name of desired subkey.

Use this function to enumerate the subkeys of a desired key. The first subkey is referenced by index number 0, the second key by 1, and so on. If the key does not exist, a null string will be returned.

Example:

```
For x=1 to 999
a=RegQueryKey(@REGMACHINE, x)
Display(2, "HKEY_LOCAL_MACHINE subkey number %x%", a)
If a="" then break
next
```

See Also:

[RegOpenKey](#), [RegCreateKey](#), [RegCloseKey](#), [RegDeleteKey](#), [RegSetValue](#), [RegQueryKeys](#), [RegQueryValue](#), and the section on [Registration Database Operations](#)

RegQueryKeys

Returns a tab-delimited list of subkey names under a specified key.

Syntax:

```
RegQueryKeys(handle)
```

Parameters:

(i) handle handle to a registration database key.

Returns:

(s) tab-delimited list of subkey names.

Example:

```
key=RegOpenkey (@RegMachine, "SOFTWARE\Microsoft")
who=RegQueryKeys (key)
RegClosekey (key)
who=StrReplace (who,@tab,@crlf)
Message("Registry Keys under SOFTWARE\Microsoft", who)
```

See Also:

[RegOpenKey](#), [RegCreateKey](#), [RegCloseKey](#), [RegDeleteKey](#), [RegSetValue](#), [RegQueryKey](#), [RegQueryValue](#), and the section on [Registration Database Operations](#)

RegQueryMulSz {*32}

Retrieves a REG_MULTI_SZ value from the registry.

Syntax:

```
RegQueryMulSz(handle, subkey-string, delimiter)
```

Parameters:

- (i) handle an open registration database key (see below).
- (s) subkey-string a path from the key provided to the desired key.
- (s) delimiter a character to act as a delimiter between items.

Returns:

- (s) a list of strings, delimited with the specified delimiter character.

Example:

```
key=RegCreatekey(@REGMACHINE, "sample")
RegSetMulSz(key, "testmulz[foods]", "apple|pear|grapes|toast", "|")
RegClosekey(key)
a1=RegOpenkey(@REGMACHINE, "sample")
c=RegQueryMulSz(a1, "testmulz[foods]",@tab)
RegClosekey(a1)
Message("Foods",c)
```

See Also:

[RegApp](#), [RegEntryType](#), [RegQueryBin](#), [RegQueryEx](#), [RegQueryExpSz](#), [RegQueryDword](#), [RegQueryItem](#), [RegQueryValue](#), and the section on [Registration Database Operations](#)

RegQueryValue

Returns data item string at subkey position.

Syntax:

RegQueryValue(handle, subkey-string)

Parameters:

- (i) handle handle to a registration database key.
- (s) subkey-string a path from the key provided to the desired key.

Returns:

- (s) contents of data item at key position desired.

Use this function to retrieve data items from the Registration Database. The function will fail if the data item does not exist.

Note: Named data entries as found in Windows 95 and NT are specified by enclosing the data item name in square brackets at the end of the key string. For example:

```
RegDelValue (@REGMACHINE, "SOFTWARE\Wilson WindowWare\Settings \WWWBATCH\
MAIN[SOUNDS]")
```

For example, here is a WIL script which modifies the default printer in Windows NT.

```
newprt = "LJ3,winspool,LPT1:" ;the printer you want to assign as the
;default
regkey = RegOpenKey (@REGCURRENT, "Software\Microsoft\Windows
NT\CurrentVersion\Windows")
defprt = RegQueryValue (regkey, "[Device]")
Message("Previous Default printer", defprt)
RegSetValue (regkey, "[Device]", newprt)
defprt = RegQueryValue (regkey, "[Device]")
Message("New Default printer", defprt)
RegCloseKey (regkey)
```

Example:

```
a1=RegOpenkey (@RegClasses, "crdfile")
a2=RegOpenkey (a1, "shell\print")
c=RegQueryValue (a2, "command")
RegCloseKey (a2)
RegClosekey (a1)
Message("Cardfile Print Processor is", c)
;
; The preceding is the hard way to do the following
b=RegQueryValue (@REGClasses, "crdfile\shell\print\command")
Message("Cardfile Print Processor is", b)
```

See Also:

[RegOpenKey](#), [RegCreateKey](#), [RegCloseKey](#), [RegDeleteKey](#), [RegDelValue](#), [RegSetValue](#), [RegQueryValue](#), and the section on [Registration Database Operations](#)

RegSetBin {*32}

Sets a binary value in the Registration Database.

Syntax:

```
RegSetBin(handle, subkey-string, value)
```

Parameters:

- | | |
|-------------------|---|
| (i) handle | handle to a registration database key. |
| (s) subkey-string | a path from the key provided to the desired key. |
| (s) value | data to be stored into the database at desired key. |

Returns:

- | | |
|-----|------------------------------------|
| (i) | @TRUE Data item was stored. |
|-----|------------------------------------|

The value is specified as a space-delimited string of hex bytes; e.g.:
"AB 45 3E 01".

Example:

```
RegSetBin(@REGCURRENT, "A Test Key\[My Binary Value]", "00 01 22 AB FF 00")  
Message("RegSetBin", " Done.")
```

See Also:

[RegApp](#), [RegEntryType](#), [RegSetDword](#), [RegSetEx](#), [RegSetExpSz](#), [RegSetMulSz](#), [RegSetValue](#)
and the section on [Registration Database Operations](#)

RegSetDword {*32}

Sets a DWORD value in the Registration Database.

Syntax:

```
RegSetDword(handle, subkey-string, value)
```

Parameters:

- | | |
|-------------------|---|
| (i) handle | handle to a registration database key. |
| (s) subkey-string | a path from the key provided to the desired key. |
| (s) value | data to be stored into the database at desired key. |

Returns:

- | | |
|-----|------------------------------------|
| (i) | @TRUE Data item was stored. |
|-----|------------------------------------|

Example:

```
RegSetDword(@REGCURRENT, "A Test Key[My DWORD Value]", 32)  
Message("RegSetDWord", "Done.")
```

See Also:

[RegApp](#), [RegEntryType](#), [RegSetBin](#), [RegSetEx](#), [RegSetExpSz](#), [RegSetMulSz](#), [RegSetValue](#) and the section on [Registration Database Operations](#)

RegSetEx

Sets a value in the registry.

Syntax:

RegSetEx(handle, subkey-string, value, delimiter, type)

Parameters:

- (i) handle an open registration database key (see below).
- (s) subkey-string a path from the key provided to the desired key.
- (s) value data to be stored in the database at desired key.
- (s) delimiter a character to act as a delimiter between items.
"Delimiter" is ignored unless "type" is 7.
- (i) type see below.

Returns:

- (s) **@TRUE** Data item was stored.

In the 16-bit version, "type" must be 1. In the 32-bit version, "type" can be one of the following:

<u>Value</u>	<u>Meaning</u>	
1	(REG_SZ)	Unicode null terminated string
2	(REG_EXPAND_SZ)	Unicode null terminated string (with environment variable references)
3	(REG_BINARY)	Free form binary
4	(REG_DWORD)	32-bit number
7	(REG_MULTI_SZ)	Multiple Unicode strings

Example:

See the Registry Cloning Script under RegEntryType.

See Also:

[RegApp](#), [RegEntryType](#), [RegSetBin](#), [RegSetDword](#), [RegSetExpSz](#), [RegSetMulSz](#), [RegSetValue](#) and the section on [Registration Database Operations](#)

RegSetExpSz {*32}

Sets a REG_EXPAND_SZ value in the registry.

Syntax:

RegSetExpSz(handle, subkey-string, value)

Parameters:

- (i) handle an open registration database key (see below).
- (s) subkey-string a path from the key provided to the desired key.
- (i) value data to be stored in the database at desired key.

Returns:

- (i) **@TRUE** Data item was stored.

Example:

```
; Note: To modify path in Windows 95+ and NT 4.0+
; See the RegApp function

; The Following is the location of users path in Windows NT
; For the User Environment Path variable use:
userenv="Environment[path]"
; For the System Environment Path variable use:
sysenv="SYSTEM\CurrentControlSet\Control\Session Manager\Environment[path]"

; Directory to add to the path
addpath="Q:\"

if WinMetrics(-4)==4        ;Check to make sure system is Windows NT.
  GetLastError( )         ;clear the errors
  ErrorMode(@off)         ;tell WB we'll handle the errors
  oldpath=RegQueryExpSz(@RegCurrent, userenv)
  ErrorMode(@cancel)

  If GetLastError( ) == 1233
    Message("Users Path", "No path found, replacement not made.")
  else
    newpath=strcat(oldpath, ";", addpath)
    c=RegSetExpSz(@RegCurrent, userenv, newpath)
    Message("Users new path is", newpath)
    WININICHG=26
    DaDll=Strcat(DirWindows(1), "USER32.DLL")
    DllCall(DaDll, long:"SendMessageA", long:-
1, long:WININICHG, long:0, lpstr:"Environment")
  endif
  exit
else
  Message("RegQueryExpSz", "Not supported by Non-NT registry")
endif
```

See Also:

[RegApp](#), [RegEntryType](#), [RegSetBin](#), [RegSetDword](#), [RegSetEx](#), [RegSetMulSz](#), [RegSetValue](#) and the section on [Registration Database Operations](#)

RegSetMulSz {*32}

Sets a REG_MULTI_SZ value in the registry.

Syntax:

```
RegSetMulSz(handle, subkey-string, value, delimiter)
```

Parameters:

- (i) handle an open registration database key (see below).
- (s) subkey-string a path from the key provided to the desired key.
- (i) value a list of strings, delimited with the specified delimiter character.
- (s) delimiter a character to act as a delimiter between items.

Returns:

- (i) **@TRUE** Data item was stored.

Example:

```
key=RegCreatekey(@REGMACHINE, "sample")
RegSetMulSz(key, "testmulz[foods]", "apple|pear|grapes|toast", "|")
RegClosekey(key)
a1=RegOpenkey(@REGMACHINE, "sample")
c=RegQueryMulSz(a1, "testmulz[foods]",@tab)
RegClosekey(a1)
Message("Foods",c)
```

See Also:

[RegApp](#), [RegEntryType](#), [RegSetBin](#), [RegSetDword](#), [RegSetEx](#), [RegSetExpSz](#), [RegSetValue](#) and the section on [Registration Database Operations](#)

RegSetValue

Sets the value of a data item in the Registration Database.

Syntax:

```
RegSetValue(handle, subkey-string, value)
```

Parameters:

- (i) handle handle to a registration database key.
- (s) subkey-string a path from the key provided to the desired key.
- (s) value data to be stored into the database at desired key.

Returns:

- (i) **@TRUE** Data item was stored.

Use this function to store data items into the Registration Database. If the desired key does not exist, the function will create it.

Note: Windows NT and Windows 95 have added named data types to the registration database entries. As a result there is a special way to access the named data entries in Windows NT and Windows 95 registration databases. The steps are as follows:

- 1) Open a key pointing to the group of data items that contains the desired data item.
- 2) Use the **RegSetValue** or the **RegQueryValue** functions to access the data value. The "subkey-string" must contain only the data item name enclosed in square brackets.
- 3) Be sure to close the key when operations are complete.

For example, here is a WIL script which modifies the default printer in Windows NT.

```
newprt = "LJ3,winspool,LPT1:" ;the printer you want to assign as the
                                ;default
regkey = RegOpenKey(@REGCURRENT, "Software\Microsoft\Windows
                                NT\CurrentVersion\Windows")
defprt = RegQueryValue(regkey, "[Device]")
Message("Previous Default printer", defprt)
RegSetValue(regkey, "[Device]", newprt)
defprt = RegQueryValue(regkey, "[Device]")
Message("New Default printer", defprt)
RegCloseKey(regkey)
```

Example:

```
; Associate DIZ files with the default textfile editor
key=RegCreatekey(@REGCLASSES, ".diz")
RegSetValue(key, "", "txtfile")
RegClosekey(key)

; The preceding is actually a hard way to do ...
RegSetValue(@REGCLASSES, ".diz", "txtfile")
Message("RegSetValue", "Done")
```

See Also:

[RegOpenKey](#), [RegCreateKey](#), [RegCloseKey](#), [RegDeleteKey](#), [RegDelValue](#), [RegQueryValue](#), [RegQueryKey](#), and the section on [Registration Database Operations](#)

RegUnloadHive {*32}

Unloads a hive from the registry.

Syntax:

```
RegUnloadHive(handle, subkey)
```

Parameters:

(i) handle	either @REGUSERS or @REGMACHINE.
(s) subkey	the name of a key which is mapped to a hive file.

Returns:

(s)	@TRUE on success.
-----	--------------------------

Example:

```
RegUnloadHive (@REGMACHINE, "WinWare")
```

See Also:

[RegLoadHive](#)

Reload {*M}

Reloads menu file(s).

Syntax:

```
Reload ( )
```

Parameters:

(none)

Returns:

(i) always 1.

This function is used to reload the WIL Interpreter's menu file(s). It is useful after editing a menu file, to cause the changes to immediately take effect.

Note1: This command does not take effect until the WIL program has completed, regardless of where the command may appear in the program.

Note2: 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.

Example:

```
RunZoomWait("notepad.exe", "c:\win\cmdpost.cpm")  
Reload( )
```

Return

Used to return from a **Call** to the calling program or to return from a **GoSub** *:label*.

Syntax:

Return

Parameters:

(none)

Returns:

(not applicable)

The **Return** statement returns to the statement following the most recently executed **Call** or **GoSub** statement. If there is no matching **Call** or **GoSub**, an **Exit** is assumed.

Example:

```
Display(2, "End of subroutine", "Returning to MAIN.WBT")  
Return
```

See Also:

[Call](#), [Exit](#), [GoSub](#)

Run

Runs a program as a normal window.

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.

Returns:

- (i) **@TRUE** if the program was found.

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, 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" has an extension other than .EXE, .COM, .PIF, or .BAT, it will be run with the associated application. When this happens, any "parameters" you specified are ignored.

Examples:

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

```
Run ("paint.exe", "pict.msp")
```

See Also:

[RunShell](#), [AppExist](#), [RunHide](#), [RunIcon](#), [RunWait](#), [RunZoom](#), [ShellExecute](#), [WinClose](#), [WinExeName](#), [WinWaitClose](#)

RunEnviron

Launches a program and has it inherit the current environment as set with the **EnvironSet** command.

Syntax:

```
RunEnviron(program-name, params, display mode, waitflag)
```

Parameters:

- (s) program-name the name of the desired Windows EXE file.
- (i) params optional parameters as required by the application.
- (i) display mode **@NORMAL, @ICON, @ZOOMED, @HIDDEN.**
- (i) waitflag **@WAIT, @NOWAIT.**

Returns:

- (i) **@TRUE** Function was executed normally.

Use this function to launch a program with the current environment.

Only Windows EXEs may be executed with this command. It is possible to change the environment for DOS programs by launching a DOS BAT file that includes DOS SET statements to alter the environment settings before executing the DOS program. Use the **Run** commands to launch DOS programs and BAT files.

When running 32 bit versions of WinBatch, this function is identical to the **RunShell** function. However, in the 16-bit version, only the **@WAIT** flag is implemented. The **@NOWAIT** flag cannot be used.

To alter the path for DOS programs, all that is required is a simple batch file, and the usual WIL **Run** command. Assuming the case where one wished to run "command.com" with the path "c:\special", a generic batch file as shown below will suffice, along with passing all the information required as parameters in the WIL **Run** command.

```
DoPath.bat   file listing
SET PATH=%1
ECHO %PATH%
PAUSE
%2 %3 %4 %5 %6 %7 %8 %9
```

```
WIL Run Command
Run("dopath.bat", "c:\special command.com")
```

HINT: Use the WWENVMAN.DII, WIL Environment extender, for enhanced environment management. Further explanations are in WWWENV.HLP.

If the "program-name" has an extension other than .EXE, .COM, .PIF, or .BAT, it will be run with the associated application.

Example:


```
;Note
;16-bit users should use the
; "Environment Manager Extender"
; instead of this function .
;
;NT Users note: Permanent changes may be made by
;using the registry functions to update the registry.
;
;Note that this function will not affect the environment for
; DOS or Console applications
;
;Note: Modifying the path does not allow assist WIL in
;locating the EXE file. The path WIL uses has already
;been pre-determined. However the child process (excel)
;will inherit the PATH environment variable in this case.

EnvironSet("PATH","c:\Program Files\Excel\")
RunEnviron("c:\Program Files\Excel\Excel.exe", "", @NORMAL, @WAIT)
```

See Also:

[RunShell](#), [Run](#), [RunWait](#), [ShellExecute](#) [Environment](#), [EnvironSet](#)

RunHide

Runs a program as a hidden window.

Syntax:

RunHide (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..

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, 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" has an extension other than .EXE, .COM, .PIF, or .BAT, it will be run with the associated application.

Note: When this command launches an application, it merely 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.

Examples:

```
;RunHide("notepad.exe", "abc.txt")  
  
;RunHide("paint.exe", "pict.msp")
```

See Also:

[RunShell](#), [Run](#), [RunHideWait](#), [RunIcon](#), [RunZoom](#), [ShellExecute](#), [WinClose](#), [WinExeName](#), [WinHide](#), [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..

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" has an extension other than .EXE, .COM, .PIF, or .BAT, it will be run with the associated application.

Note: When this command launches an application, it merely 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:

```
RunHideWait(Environment("COMSPEC"), "/c dir *.exe> temp.txt")
Print("temp.txt", "", @NORMAL, @NOWAIT)
```

See Also:

[RunShell](#), [RunHide](#), [RunIconWait](#), [RunWait](#), [RunZoomWait](#), [ShellExecute](#) [WinWaitClose](#)

RunIcon

Runs a program as an iconic (minimized) window.

Syntax:

RunIcon (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.

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, 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" has an extension other than .EXE, .COM, .PIF, or .BAT, it will be run with the associated application.

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.

Examples:

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

See Also:

[RunShell](#), [Run](#), [RunHide](#), [RunIconWait](#), [RunZoom](#), [ShellExecute](#), [WinClose](#), [WinExeName](#), [WinIconize](#), [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.

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" has an extension other than .EXE, .COM, .PIF, or .BAT, it will be run with the associated application.

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:

```
RunIconWait(Environment("COMSPEC"), "/c dir *.exe> temp.txt")
Message("RunIconWait","It Ran, It Iconned, It Waited.")
```

See Also:

[RunShell](#), [RunHideWait](#), [RunIcon](#), [RunWait](#), [RunZoomWait](#), [ShellExecute](#) [WinWaitClose](#)

RunShell

An advanced form of the **Run** function that even allows the specification of a working directory, along with the window view mode and whether or not to wait for completion of the run program in a single function.

Syntax:

RunShell(program-name, params, directory, display mode, waitflag)

Parameters:

- | | |
|------------------|---|
| (s) program-name | the name of the desired .EXE, .COM, .PIF, .BAT file or a data file. |
| (s) params | optional parameters as required by the application. |
| (s) directory | current working directory (if applicable). |
| (i) display mode | @NORMAL, @ICON, @ZOOMED, @HIDDEN |
| (i) waitflag | @WAIT, @NOWAIT, @GETPROCID. |

Returns:

- (i) **@TRUE** if the program was found.

If a data file is specified instead of an executable file (i.e. EXE, COM, PIF, or BAT file), the function examines the extension of the data file, looks the extension up in the Windows registry to determine the owning application and starts the owning application, passing the data file name as a parameter.

Applications that support this command or their setup programs will generally make the necessary modifications to the Windows registry to allow this function to perform successfully.

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 **@WAIT** parameter is used, the WIL program will suspend processing until the application is closed.

If the **@GETPROCID** parameter is used, a process-id will be returned that can be used with the function **WinItemProcID**.

Note: When this command launches an application, it merely informs it how you wish it to appear on the screen. Whether or not the application honors your wish is beyond this function's control.

Example:

```
RunShell("NOTEPAD.EXE", "CONFIG.SYS", "C:\", @NORMAL, @NOWAIT)
```

See Also:

[WinItemProcID](#), [Print](#), [Run](#), [RunWait](#), [ShellExecute](#)

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.

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" has an extension other than .EXE, .COM, .PIF, or .BAT, it will be run with the associated application.

Example:

```
RunWait(Environment("COMSPEC"), "/c dir *.exe> temp.txt")
Print("temp.txt", "", @NORMAL, @NOWAIT)
WinWaitClose("~Notepad")
FileDelete("temp.txt")
```

See Also:

[RunShell](#), [AppWaitClose](#), [Run](#), [RunHideWait](#), [RunIconWait](#), [RunZoomWait](#), [ShellExecute](#)
[WinWaitClose](#)

RunZoom

Runs a program as a full-screen (maximized) window.

Syntax:

RunZoom (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.

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, 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" has an extension other than .EXE, .COM, .PIF, or .BAT, it will be run with the associated application.

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.

Examples:

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

See Also:

[RunShell](#), [Run](#), [RunHide](#), [RunIcon](#), [RunZoomWait](#), [ShellExecute](#), [WinClose](#), [WinExeName](#), [WinWaitClose](#), [WinZoom](#)

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.

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" has an extension other than .EXE, .COM, .PIF, or .BAT, it will be run with the associated application.

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:

```
RunZoomWait (Environment ("COMSPEC"), "/c dir *.exe> temp.txt")
Print ("temp.txt", "", @NORMAL, @NOWAIT)
```

See Also:

[RunShell](#), [RunHideWait](#), [RunIconWait](#), [RunWait](#), [RunZoom](#), [ShellExecute](#) [WinWaitClose](#)

Select

The **Select** statement allows selection among multiple blocks of statements.

Syntax:

```
Select expression
case expression
    statements
    break
case expression
    statements
    break
EndSelect
```

Parameters:

(s) expression an expression that must evaluate to an integer.

The **Select** statement allows selection among multiple blocks of statements, depending on the value of an expression. The expression must evaluate to an integer.

The **Select** statement causes the statements in the select body to be scanned by the parser as it attempts to find a **case** statement. When a **case** statement is found, the expression following the **case** statement is evaluated, and if the expression evaluates to the same value as the expression following the **Select** statement, execution of the following statements is initiated. The **EndSelect** statement terminates the **Select** structure.

If a matching **case** expression was found, and execution was initiated, the following statements will affect continued execution:

Break	Terminates the Select structure and transfers control to the statement following the next matching EndSelect .
Continue	Stops execution and resumes scanning for a case statement.
Case	Ignored. Treated as a comment.
EndSelect	Terminates the Select structure and transfers control to the next statement.

Note: **Switch** and **Select** may be used interchangeably. They are synonyms for the same statement.

EndSwitch, **EndSelect**, **"End Switch"**, and **"End Select"** may be used interchangeably.

Example:

```
response=AskLine("Select", "Enter a number between one and three", 1)
Select response
    case 1
        Message("Select", "Case 1 entered")
        break
    case 2
        Message("Select", "Case 2 entered")
        break
    case 3
        Message("Select", "Case 3 entered")
        break
    case response ; default case
        Message("Select", "Default case entered")
        break
End Select
```

See Also:

If, For, GoSub, While

SendKey

Sends keystrokes to the currently active window.

Syntax:

SendKey (char-string)

Parameters:

(s) char-string string of regular and/or special characters.

Returns:

(i) always 0.

Note: **SendKey** will send keystrokes to the currently active window. For many applications, the related functions, **SendKeysChild**, **SendKeysTo** or **SendMenusTo** may be better alternatives. There is no way to emulate the number pad with Sendkey or Sendkeysto

This function is used to send keystrokes to the active 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:

<u>Key</u>	<u>SendKey equivalent</u>
~	{~} ; This is how to send a ~
!	{!} ; This is how to send a !
^	{^} ; This is how to send a ^
+	{+} ; This is how to send a +
{	{{} ; This is how to send a {
}	{}} ; This is how to send a }
Alt	{ALT}
Backspace	{BACKSPACE} or {BS}
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} or {INS}
Left Arrow	{LEFT}
Page Down	{PGDN}
Page Up	{PGUP}
Right Arrow	{RIGHT}
Space	{SPACE} or {SP}
Tab	{TAB}
Up Arrow	{UP}

Additional special characters to the SendKey (and SendKeysTo and SendKeysChild) functions:

<u>Key</u>	<u>SendKey equivalent</u>
------------	---------------------------

0 on numeric keypad	{NUMPAD0}
1 on numeric keypad	{NUMPAD1}
2 on numeric keypad	{NUMPAD2}
3 on numeric keypad	{NUMPAD3}
4 on numeric keypad	{NUMPAD4}
5 on numeric keypad	{NUMPAD5}
6 on numeric keypad	{NUMPAD6}
7 on numeric keypad	{NUMPAD7}
8 on numeric keypad	{NUMPAD8}
9 on numeric keypad	{NUMPAD9}
* on numeric keypad	{NUMPAD*}
+ on numeric keypad	{NUMPAD+}
- on numeric keypad	{NUMPAD-}
. on numeric keypad	{NUMPAD.}
/ on numeric keypad	{NUMPAD/}
Enter on numeric keypad	{NUMPAD~}

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}")
```

To toggle a checkbox:

```
SendKey("{+}") ;always checks a checkbox
SendKey("{-}") ;always unchecks a checkbox
```

Examples:

```
; start Notepad, and use *.* for filenames
Run("notepad.exe", "")
SendKey("!fo*.~")
```

In those cases where you have an application which can accept text pasted in from the clipboard, it will often be more efficient to use the **ClipGet** function:

```
Run("notepad.exe", "")
CrLf = StrCat(Num2Char(13), Num2Char(10))
; copy some text to the clipboard
ClipPut("Dear Sirs:%CrLf%%CrLf%")
; paste the text into Notepad (using Ctrl-v)
SendKey("^v")
```

A **WIL** program cannot send keystrokes to its own **WIL** Interpreter window.

Notes:

You should, in general, use lower-case letters to represent Alt-key combinations and other menu shortcut keys as that is the normal keys used when typing to application. For example "lfo" is interpreted as Alt-f-o, as one might expect. However "lFO" is interpreted as Alt-Shift-f-o, which is not a normal keystroke sequence.

If your **SendKey** statement doesn't seem to be working (e.g., all you get are beeping noises), you may need to place a **WinActivate** statement before the **SendKey** statement to insure that you are sending the keystrokes to the correct window, or you may try using the **SendKeysTo** or **SendKeysChild** function.

When sending keystrokes to a DOS box, the DOS box must be in a window (Not Full Screen). Most keystrokes can be sent to a full screen DOS box, however, **SendKey** can only send the ENTER key to a Windowed DOS Box.

See Also:

[SendKeysTo](#), [SendKeysChild](#), [SendMenusTo](#), [SnapShot](#), [WinActivate](#)

SendKeysChild

Sends keystrokes to the active child window.

Syntax:

SendKeysChild(main-windowname, child windowname, sendkey string)

Parameters:

- (s) main- windowname the initial part of, or an entire parent window name.
- (s) child-windowname the initial part of, or an entire child window name.
- (s) sendkey string string of regular and/or special characters.

Returns:

- (i) always 0

Use this function to send keystrokes to a particular child window. This function is similar to **SendKey**, but the desired parent and child windows will be activated before sending any keys in lieu of using **WinActivChild**. Consequently, a previous **WinActivChild** command will be overridden by this function. See the **SendKey** function for a description of the "sendkey string".

Note: "main-windowname" and "child-windowname" are the initial parts of their respective window names, and may be complete window names. They are case-sensitive. You should specify enough characters so that the window names will match only one existing window of its type. If a windowname matches more than one window, the most recently accessed window which it matches will be used.

Example:

```
; Start Windows File Manager - the hard way
; This code activates Program Manager, then
; activates the "Main" child window. Sending an
; "f" should (hopefully) activate the File Manager
; icon. The Enter key (abbreviated as ~ ) runs it.
SendKeysChild("Program Manager", "Main", "f~")
```

See Also:

[SendKeysTo](#), [SendKey](#), [SendMenusTo](#), [SnapShot](#), [WinActivate](#)

SendKeysTo

Sends keystrokes to a "parent-windowname".

Syntax:

```
SendKeysTo(parent-windowname, sendkey string)
```

Parameters:

- (s) parent-windowname the initial part of, or an entire parent window name.
- (s) sendkey string string of regular and /or special characters.

Returns:

- (i) always 0.

Use this function to send keystrokes to a particular window. This function is similar to **SendKey**, but the correct "parent-windowname" will be activated before sending any keys in lieu of using **WinActivate**. Consequently, a previous **WinActivate** command will be overridden by this function. See the **SendKey** function for a description of the "sendkey string". There is no way to emulate the number pad with **Sendkey** or **Sendkeysto**.

Note: "parent-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 "windowname" matches only one existing window. If it matches more than one window, the most recently accessed window which it matches will be used.

Example:

```
run ("notepad.exe", "")
SendKeysTo ("~Notepad", "aBcDeF")
```

See Also:

[SendKey](#), [SendKeysChild](#), [SendMenusTo](#), [SnapShot](#), [WinActivate](#)

SendMenusTo

Activates a window and sends a specified menu option.

Syntax:

```
SendMenusTo(windowname, menuname)
```

Parameters:

(s) windowname the initial part of, or an entire parent window name.
(s) menuname windows message to be posted or performed.

Returns:

(i) always 0.

Use this function to access drop down menus on a window. The function activates the "windowname" application window, searches its menus and sends the specified windows message for the menu operation.

To construct the "menuname" parameter simply string together all the menu options selected to access the desired function. All punctuation and special characters are ignored, as well as any possible "hotkeys" used to access the function via the keyboard. For example, most Windows applications have a "File" menu and an "Open" menu. To construct the "menu name" parameter, simply string together the words, making "FileOpen", or for better readability use "File Open" - the spaces are ignored.

In 32-bit version, SendMenusTo attempts to wait until the specified application window is ready to accept input (like SendKey[...] does). This can be disabled with IntControl(43).

Note: "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 "windowname" matches only one existing window. If it matches more than one window, the most recently accessed window which it matches will be used.

Example:

```
Run("notepad.exe", "c:\config.sys")  
SendMenusTo("~Notepad", "Edit Select All")  
SendMenusTo("~Notepad", "Edit Copy")
```

See Also:

[SendKeysTo](#), [SendKeysChild](#), [SendKey](#), [SnapShot](#)

ShellExecute

Runs a program via the Windows ShellExecute command

Syntax:

ShellExecute(program-name, params, directory, display mode, operation)

Parameters:

- (s) program-name the name of the desired .EXE, .COM, .PIF, .BAT file or a data file.
- (s) params optional parameters as required by the application.
- (s) directory current working directory (if applicable).
- (i) display mode **@NORMAL**, **@ICON**, **@ZOOMED**, **@HIDDEN.**; or 0 for the default mode.
- (i) operation operation to perform on the specified file.

Returns:

- (i) **@TRUE** on success.

This function uses the Windows ShellExecute API to launch the specified file. The similar **RunShell** function also uses the ShellExecute API in the 16-bit version, but uses the CreateProcess API in the 32-bit version. Note that **RunShell** has a "wait" parameter, while this function does not.

"operation" is the operation to perform on the file ("Open", "Print", etc.), which may or may not correspond to an available "verb" on the context menu for the file. This parameter may be case-sensitive. Specify a blank string "" for the file's default operation.

Note: If you use this function to launch a shortcut, and the shortcut points to an invalid path, Windows will display a "Missing Shortcut" dialog box asking if you wish to update the shortcut. This would not be suitable to use in unattended operation. Instead, you could use one of the **Run..** functions to launch the shortcut, which would return an error #1932 if the shortcut could not be launched. This error can be trapped using the **ErrorMode** function.

Example:

```
; launches a shortcut to a "Dial-Up Networking" item on the desktop
ShellExecute("d:\win95\desktop\netcom.lnk", "", "", @NORMAL, "")
WinWaitClose("Connect To")
Message("ShellExecute", " Executed.")
```

See Also:

[RunShell](#)

ShortcutDir {*32}

Returns the name of the directory where shortcuts of type "name" are stored.

Syntax:

ShortcutDir(name)

Parameters:

(s) name see list below.

Returns:

(i) Returns a directory name on success, or a blank string ("") if no corresponding value was found.

This information is retrieved from the registry, under the key:

HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Explorer\Shell Folders

Standard values for "name" include:

Desktop
Programs
Fonts
Recent
SendTo
Personal
Startup
Start Menu
NetHood
Favorites
Templates

Example:

```
startdir=ShortcutDir("Desktop")
DirChange(startdir)
ShortcutMake("system~1.LNK", "c:\Program Files\winbatch\system~1.wbt", "", "",
@NORMAL)
ShortcutEdit("system~1.LNK", "", "", "c:\Win95\desktop", @NORMAL)
```

See Also:

[ShortCutExtra](#), [ShortcutInfo](#), [ShortcutMake](#)

ShortcutEdit {*Explorer}

Modifies the specified shortcut file.

Syntax:

ShortcutEdit(link-name, target, params, start-dir, show-mode)

Parameters:

- | | |
|---------------|--|
| (s) link-name | the name of shortcut .LNK file to be edited. |
| (s) target | file or directory name which "link-name" will point to. |
| (s) params | optional command-line parameters for "target" |
| (s) start-dir | "Start in" directory for "target". |
| (i) show-mode | "Run" mode for "target": @NORMAL , @ZOOMED , or @ICON . |

Returns:

- (i) **@TRUE** if the shortcut was successfully modified.

See **ShortcutMake** for further information on these parameters.

Note: In the ShortcutMake and ShortcutEdit functions, if the "target" parameter specifies a UNC, and it does not exist at the time this function is called, then the "target" field in the shortcut may not be properly set. Specifically, it may get truncated after the share name. For example, if you use:

```
ShortcutMake("test.lnk", "\\server\public\testdir\myprog.exe", "", "", 0)
```

and \\server\public\testdir\myprog.exe doesn't exist, then the target of the share may get set to:

```
\\server\public
```

This is due to the behavior of the API function we are calling, and is consistent with the behavior when trying to create or edit a shortcut interactively from Explorer (ie, you are prevented from specifying an invalid target).

Example:

```
scfile="system~1.LNK"
startdir=RegQueryValue(@REGCURRENT,"Software\Microsoft\Windows\
CurrentVersion\Explorer\Shell Folders[Desktop]")
DirChange(startdir)
ShortcutMake(scfile, "c:\Program Files\winbatch\system~1.wbt", "", "", @NORMAL)
ShortcutEdit(scfile, "", "", "c:\Win95\desktop", @NORMAL)
```

See Also:

[ShortCutExtra](#), [ShortcutInfo](#), [ShortcutMake](#)

ShortcutExtra

Sets additional information for the specified shortcut file.

Syntax:

```
ShortcutExtra(link-name, description, hotkey, icon-file, icon-index)
```

Parameters:

(s) link-name	the name of shortcut .LNK file to be modified.
(s) description	the internal description for the shortcut.
(s) hotkey	the "shortcut key" to be assigned to the shortcut.
(s) icon-file	a file containing an icon to be used for the shortcut, with optional path.
(i) icon-index	the 0-based index position of the desired icon within "icon-file".

Returns:

- (i) **@TRUE** if the shortcut was successfully modified.

The "description" parameter only sets an internal description, which is not actually displayed anywhere.

If "hotkey" is not a blank string (""), it specifies the hotkey ("shortcut key") for the shortcut. This can be an alphanumeric or special character (see **SendKey** for a list of special key characters), optionally preceded by one or more of the following modifiers:

!	(Alt)
^	(Control)
+	(Shift)

This function can be used to set hotkeys which would be impossible to set from within the shortcut properties dialog in Explorer.

"Icon-file" can be used to specify an .EXE (or .DLL) file or an .ICO file containing an icon which you want to be used for the shortcut. If "icon-file" specifies an .EXE (or .DLL) file (which can contain multiple icons), then "icon-index" can be used to specify the offset of a particular icon within "icon-file", where 0 indicates the first icon in the file, 1 indicates the second icon, etc. If "icon-file" specifies an .ICO file, then "icon-index" should be 0.

You can specify a blank string ("") for "icon-file", and 0 for "icon-index", to use the default icon.

Example:

```
startdir=RegQueryValue(@REGCURRENT,"Software\Microsoft\Windows\
    CurrentVersion\Explorer\Shell Folders\Desktop")
DirChange(startdir)
ShortcutMake("system~1.LNK", "c:\Program Files\winbatch\system~1.wbt","",
c:\Program Files\Winbatch", @NORMAL)
ShortcutExtra("system~1.LNK", "WinBatch Version Info", "^!j", "", 0)
```

See Also:

[ShortcutEdit](#), [ShortcutInfo](#), [ShortcutMake](#)

ShortcutInfo {*Explorer}

Returns information on the specified shortcut file.

Syntax:

ShortcutInfo(link-name)

Parameters:

(s) link-name the name of shortcut .LNK file.

Returns:

(s) a TAB delimited list of information on the shortcut file.

ShortcutInfo returns a TAB-delimited list containing the following items (some of which may be blank):

target	file or directory name which the shortcut points to.
params	command-line parameters for "target".
start-dir	"Start in" directory for "target".
show-mode	"Run" mode for "target": 1 (@ ICON), 2 (@ NORMAL), or 3 (@ ZOOMED).
description	the internal description for the shortcut.
hotkey	the "shortcut key" for the shortcut.
icon-file	the name of the icon file being used by the shortcut.
icon-index	the 0-based index position within "icon-file" of the icon being used.

Example:

```
startdir=RegQueryValue(@REGCURRENT,"Software\Microsoft\Windows\CurrentVersion\
Explorer\Shell Folders\Desktop")
DirChange(startdir)
if FileExist("system~1.LNK") then FileDelete("system~1.LNK")
ShortcutMake("system~1.LNK", "c:\Program Files\winbatch\system~1.wbt",
            "", "", NORMAL)
ShortcutExtra("system~1.LNK", "WinBatch Version Info", "!j", "", 0)
info=ShortcutInfo("system~1.LNK")

target= ItemExtract(1, info, @tab)
params= ItemExtract(2, info, @tab)
workdir= ItemExtract(3, info, @tab)
showmode= ItemExtract(4, info, @tab)
desc= ItemExtract(5, info, @tab)
hotkey= ItemExtract(6, info, @tab)

iconfile= ItemExtract(7, info, @tab)
iconindex= ItemExtract(8, info, @tab)

editinfo=StrCat("filename=",target,@crlf,"params=",params,@crlf,"workdir=",
               workdir,@crlf,"showmode=",showmode)
extrainfo=StrCat("desc=",desc,@crlf,"hotkey=",hotkey,@crlf,"iconfile=",
                iconfile,@crlf,"iconindex=",iconindex)
Message("ShortcutInfo Syntax", StrCat(editinfo,@crlf, extrainfo))
```

See Also:

[ShortcutEdit](#), [ShortCutExtra](#), [ShortcutMake](#)

ShortcutMake {*Explorer}

Creates a shortcut for the specified filename or directory.

Syntax:

ShortcutMake(link-name, target, params, start-dir, show-mode)

Parameters:

(s) link-name	the name of shortcut .LNK file to be created.
(s) target	file or directory name which "link-name" will point to.
(s) params	optional command-line parameters for "target".
(s) start-dir	"Start in" directory for "target".
(i) show-mode	"Run" mode for "target": 1 (@ICON), 2 (@NORMAL), or 3 (@ZOOMED).

Returns:

(i) @TRUE if the shortcut was successfully created.

This function can be used to create a shortcut file which points to a filename or to a directory.

"Params" and "start-dir" are optional, and can be set to blank strings (""). "Show-mode" is optional, and can be set to 0.

If "target" specifies a directory, the other parameters are meaningless.

Note: In the ShortcutMake and ShortcutEdit functions, if the "target" parameter specifies a UNC, and it does not exist at the time this function is called, then the "target" field in the shortcut may not be properly set. Specifically, it may get truncated after the share name. For example, if you use:

```
ShortcutMake("test.lnk", "\\server\public\testdir\myprog.exe", "", "", 0)
```

and \\server\public\testdir\myprog.exe doesn't exist, then the target of the share may get set to:

```
\\server\public
```

This is due to the behavior of the API function we are calling, and is consistent with the behavior when trying to create or edit a shortcut interactively from Explorer (ie, you are prevented from specifying an invalid target).

Example:

```
startdir=RegQueryValue(@REGCURRENT,"Software\Microsoft\Windows\
CurrentVersion\Explorer\Shell Folders\Desktop")
DirChange(startdir)
ShortcutMake("system~1.LNK", "c:\Program Files\winbatch\system~1.wbt","",
"c:\Program Files\Winbatch", @NORMAL)
```

See Also:

[ShortcutEdit](#), [ShortCutExtra](#), [ShortcutInfo](#),

Sin

Calculates the sine.

Syntax:

Sin(x)

Parameters:

(f) x angle in radians.

Returns:

(f) The **Sin** function returns the sine of x.

Calculates the sine. If the passed parameter is large, a loss in significance in the result or significance error may occur.

Note: To convert an angle measured in degrees to radians, simply multiply by the constant **@Deg2Rad**.

Example:

```
real=AskLine("Sine", "Enter an angle between 0 and 360", "45")
answer=sin(real * @Deg2Rad)
Message("Sine of %real% degrees is", answer)
```

See Also:

[Acos](#), [Asin](#), [Atan](#), [Cos](#), [Tan](#), [Sinh](#)

Sinh

Calculates the hyperbolic sine.

Syntax:

`Sinh(x)`

Parameters:

(f) *x* angle in radians.

Returns:

(f) the hyperbolic sine of *x*.

Calculates the hyperbolic sine. If the passed parameter is large, a loss in significance in the result or significance error may occur.

Note: To convert an angle measured in degrees to radians, simply multiply by the constant `@Deg2Rad`.

Example:

```
real=AskLine("SinH", "Enter an angle between 0 and 360", "45")
answer=sinh(real * @Deg2Rad)
Message("Hyperbolic Sine of %real% degrees is", answer)
```

See Also:

[Acos](#), [Asin](#), [Atan](#), [Cos](#), [Cosh](#), [Sin](#), [Tan](#), [Tanh](#)

Snapshot

Takes a bitmap snapshot of the screen and pastes it to the clipboard.

Syntax:

Snapshot (request#)

Parameters:

(i) request# see below.

Returns:

(i) always 0.

<u>Req#</u>	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:

```
;This code will snapshot the screen to the clipboard,  
; Add a BMP file header and write out the data as  
; a legal bmp file. Requires WinBatch 98A or newer  
Snapshot(0)  
size=BinaryClipGet(0,8)  
bb=BinaryAlloc(size)  
BinaryClipGet(bb,8)  
  
bb2=BinaryAlloc(size+14)  
BinaryPokeStr(bb2,0,"BM")  
BinaryPoke4(bb2,2,size+14)  
tableloc=BinaryPeek4(bb,0)+14  
BinaryPoke4(bb2,10,tableloc)  
BinaryCopy(bb2,14,bb,0,size)  
BinaryWrite(bb2,"c:\temp\screenshot.bmp")  
BinaryFree(bb)  
BinaryFree(bb2)  
Message("All","Done")
```

See Also:

[ClipPut](#)

Sounds

Turns sounds on or off.

Syntax:

Sounds (request#)

Parameters:

(i) request# 0 to turn sounds off; 1 to turn sounds on.

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 16-bit WIL Interpreter makes noise. You can override this by entering:

```
Sounds=0
```

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

Sounds are disabled by default in the 32-bit WIL Interpreter.

Example:

```
Sounds(0)  
Message("Sounds", " Sounded.")
```

See Also:

[Beep](#), [PlayMedia](#), [PlayMidi](#), [PlayWaveForm](#)

Sqrt

Calculates the square root.

Syntax:

Sqrt(x)

Parameters:

(f) x floating point number.

Returns:

(f) the square root result.

The **Sqrt** function calculates the square root of the passed parameter. If the passed parameter is negative, a domain error occurs.

Example:

```
real=AskLine("Square Root", "Enter a positive number", "269")
answer=sqrt(real)
Message("Square root of %real% is", answer)
```

See Also:

[Operator **](#)

StrCat

Concatenates two or more strings.

Syntax:

```
StrCat (string1, string2[, ..., stringN])
```

Parameters:

(s) string1, etc. at least two strings you want to concatenate.

Returns:

(s) concatenation of the entire list of input strings.

Use this command to stick character strings together, or to format display messages. Although the substitution feature of the WIL (putting percent signs on both side of a variable name) is a little quicker and easier than the strcat function, substitution should only be used for simple, short cases. Use **StrCat** when concatenating large strings.

Example:

```
user = AskLine("Login", "Your Name:", "")
msg = StrCat("Hi, ", user)
Message("Login", msg)
```

```
; note that this is the same as the second line above:
msg = "Hi, %user%"
```

See Also:

[StrFill](#), [StrFix](#), [StrTrim](#)

StrCharCount

Counts the number of characters in a string.

Syntax:

```
StrCharCount(string)
```

Parameters:

(s) string any text string.

Returns:

(i) the number of characters in a string,

Use this function to count the number of characters in a string. This function is useful when dealing with double-byte character sets such as those containing Kanji characters. When using single byte character sets, such as those found in English versions of Windows, this function is identical to the **StrLen** function.

Example:

```
name = AskLine("Data Entry", "Please enter your name", "")
len = StrLen(name)
chars = StrCharCount(name)
Message(name, "Is %len% bytes long and %CRLF% has %chars% characters")
```

See Also:

[StrLen](#), [StrScan](#), [StrReplace](#), [StrFill](#)

StrCmp

Compares two strings.

Syntax:

```
StrCmp (string1, string2)
```

Parameters:

(s) string1, string2 strings to compare.

Returns:

(i) -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)
c = c + 1
d = StrSub("less than    equal to    greater than", (c * 12)+ 1, 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:

- (s) filler a string to be repeated to create the return string.
 If the filler string is null, spaces will be used instead.
- (i) length the length of the desired string.

Returns:

- (s) 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:



See Also:

[StrCat](#), [StrFix](#), [StrLen](#), [StrTrim](#)

StrFix

Pads or truncates a string to a fixed length using bytes.

Syntax:

```
StrFix (base-string, pad-string, length)
```

Parameters:

- (s) base-string string to be adjusted to a fixed length.
- (s) pad-string appended to **base-string** if needed to fill out the desired length. If **pad-string** is null, spaces are used instead.
- (i) length length of the desired string.

Returns:

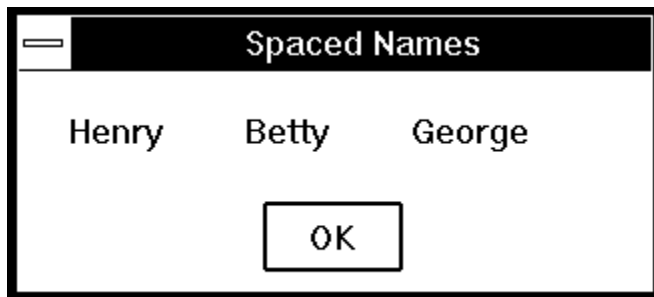
- (s) fixed size string.

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:



See Also:

[StrFill](#), [StrLen](#), [StrTrim](#)

StrFixChars

Pads or truncates a string to a fixed length using characters.

Syntax:

```
StrFixChars(base-string, pad-string, length)
```

Parameters:

- (s) base-string string to be adjusted to a fixed length.
- (s) pad-string appended to **base-string** if needed to fill out the desired length. If **pad-string** is null, spaces are used instead.
- (i) length character count of the desired string.

Returns:

- (s) fixed size string.

This function is similar to **StrFix** in that it "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. However, **StrFixChars** works based on characters rather than bytes. This function is useful when dealing with double-byte character sets such as those containing Kanji characters. When using single byte character sets, such as those found in English versions of Windows, this function is identical to the **StrFix** function.

Example:

```
a = StrFixChars("Henry", " ", 15)
b = StrFixChars("Betty", " ", 15)
c = StrFixChars("George", " ", 15)
Message("Spaced Names", StrCat(a, b, c))
```

See Also:

[StrFix](#), [StrFill](#)

StrFixLeft

Pads or truncates the left side of a string to a fixed length using bytes.

Syntax:

```
StrFixLeft(base-string, pad-string, length)
```

Parameters:

- (s) base-string string to be adjusted to a fixed length.
- (s) pad-string appended to base-string if needed to fill out the desired length. If pad-string is null, spaces are used instead.
- (i) length length of the desired string.

Returns:

- (s) fixed size string.

This function "fixes" the length of a string, either by truncating it on the left, or by prepending enough copies of pad-string to achieve the desired length.

Example:

```
a = StrFixLeft("Henry", " ", 15)
b = StrFixLeft("Betty", " ", 15)
c = StrFixLeft("George", " ", 15)
Message("Spaced Names", StrCat(a, b, c))
```

See Also:

[StrFill](#), [StrFix](#), [StrFixCharsL](#), [StrLen](#), [StrTrim](#)

StrFixCharsL

Pads or truncates the left side of a string to a fixed length using characters.

Syntax:

```
StrFixCharsL(base-string, pad-string, length)
```

Parameters:

- (s) base-string string to be adjusted to a fixed length.
- (s) pad-string appended to base-string if needed to fill out the desired length. If pad-string is null, spaces are used instead.
- (i) length length of the desired string.

Returns:

- (s) fixed size string.

This function is similar to **StrFixLeft** in that it "fixes" the length of a string, either by truncating it on the left, or by prepending enough copies of pad-string to achieve the desired length. However, **StrFixCharsL** works based on characters rather than bytes. This function is useful when dealing with double-byte character sets such as those containing Kanji characters. When using single byte character sets, such as those found in English versions of Windows, this function is identical to the **StrFixLeft** function.

Example:

```
a = StrFixCharsL("Henry", " ", 15)
b = StrFixCharsL("Betty", " ", 15)
c = StrFixCharsL("George", " ", 15)
Message("Spaced Names", StrCat(a, b, c))
```

See Also:

[StrFill](#), [StrFix](#), [StrFixLeft](#), [StrLen](#), [StrTrim](#)

StriCmp

Compares two strings without regard to case.

Syntax:

```
StriCmp (string1, string2)
```

Parameters:

(s) string1, string2 strings to compare.

Returns:

(i) -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", "")
c = StriCmp(a, b)
c = c + 1
d = StrSub("less than    equal to    greater than", (c * 12)+ 1, 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 sub-string.

Syntax:

StrIndex (string, sub-string, start, direction)

Parameters:

- (s) string the string to be searched for a sub-string.
- (s) sub-string the string to look for within the main string.
- (i) start the position in the main string to begin search.
The first character of a string is position **1**.
- (i) direction the search direction. **@FWDSCAN** searches forward,
while **@BACKSCAN** searches backwards.

Returns:

- (i) position of **sub-string** within **string**, or 0 if not found.

This function searches for a sub-string 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 "sub-string" 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
daend = StrIndex(instr, " ", start, @FWDSCAN)
If daend == 0
    Message("Sorry...", "No spaces found")
else
    a = StrCat("First word is: ", StrSub(instr, start, daend - 1))
    Message("STRINDEX", a)
endif
```

See Also:

[StrLen](#), [StrScan](#), [StrSub](#)

StrIndexNc

Searches a string for a sub-string, ignoring case.

Syntax:

StrIndexNc (string, sub-string, start, direction)

Parameters:

- (s) string the string to be searched for a sub-string.
- (s) sub-string the string to look for within the main string.
- (i) start the position in the main string to begin search.
The first character of a string is position **1**.
- (i) direction the search direction. **@FWDSCAN** searches forward,
while **@BACKSCAN** searches backwards.

Returns:

- (i) position of **sub-string** within **string**, or 0 if not found.

This function searches for a sub-string 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 "sub-string" within the "target" string, and returns its position. It is not case-sensitive.

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
daend = StrIndexNc(instr, " ", start, @FWDSCAN)
If daend == 0
    Message("Sorry...", "No spaces found")
else
    a = StrCat("First word is: ", StrSub(instr, start, daend - 1))
    Message("STRINDEX", a)
endif
```

See Also:

[StrLen](#), [StrScan](#), [StrSub](#), [StrIndexWild](#)

StrIndexWild

Searches a string for a sub-string, ignoring case.

Syntax:

```
StrIndexWild(string, pattern, start)
```

Parameters:

- (s) string the string to be searched for a sub-string.
- (s) pattern the wildcarded text to search for.
- (i) start the position in the main string to begin search.
The first character of a string is position **1**.

Returns:

- (i) position of **sub-string** within **string**, or 0 if not found.

This function searches for a sub-string within a "target" string. Starting at the "start" position. It stops when it finds the "sub-string" within the "target" string, and returns its position.

In the wildcard pattern, "*" matches zero or more characters, and "?" matches any one character.

Example:

```
StrIndexWild("abcd1234wxyz", "ab*wx", 1)    ; would return 1  
StrIndexWild("abcd1234wxyz", "1*9", 1)     ; would return 0 (no match)  
StrIndexWild("abcd1234wxyz", "1*4", 1)     ; would return 5  
StrIndexWild("abcd1234wxyz", "1?34*z", 1)  ; would return 5
```

See Also:

[StrLen](#), [StrScan](#), [StrSub](#), [StrIndexNc](#)

StrLen

Provides the length of a string.

Syntax:

StrLen (string)

Parameters:

(s) string any text string.

Returns:

(i) 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
    Message("Error", "Filename too long!")
endif
```

See Also:

[StrFill](#), [StrFix](#), [StrIndex](#), [StrIndexNc](#), [StrScan](#), [StrTrim](#)

StrLenWild

Finds wildcarded text within a larger string.

Syntax:

StrLen (string, pattern, start)

Parameters:

- | | |
|-------------|---|
| (s) string | any text string. |
| (s) pattern | The wildcarded text to match. |
| (i) start | The position (in characters) within the string to begin matching. |

Returns:

- | | |
|-----|--|
| (i) | The length (in characters) of the portion of the string that matches the wildcard pattern, or 0 if no match at the specified position. |
|-----|--|

In the wildcard pattern, "*" matches zero or more characters, and "?" matches any one character.

Example:

```
StrLenWild("abcd1234wxyz", "ab*wx", 1) ; would return 10
StrLenWild("abcd1234wxyz", "1*9", 1) ; would return 0
StrLenWild("abcd1234wxyz", "1*4", 5) ; would return 4
StrLenWild("abcd1234wxyz", "1?34*z", 5) ; would return 8
```

See Also:

[StrFill](#), [StrLen](#), [StrFix](#), [StrIndex](#), [StrIndexNc](#), [StrScan](#), [StrTrim](#)

StrLower

Converts a string to lowercase.

Syntax:

StrLower (string)

Parameters:

(s) string any text string.

Returns:

(s) 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 occurrences of a sub-string with another.

Syntax:

StrReplace (string, old, new)

Parameters:

(s) string	string in which to search.
(s) old	target sub-string.
(s) new	replacement sub-string.

Returns:

(s) updated string, with **old** replaced by **new**.

StrReplace scans the "string", searching for occurrences of "old" and replacing each occurrence with "new".

Example:

```
IntControl(29, @TAB,0,0,0)
; Copy all INI files to clipboard
a = FileItemize("*.ini")
;in file list replace tabs with crlf's.
b = StrReplace(a, @tab, @CRLF)
ClipPut(b)
newlist=StrCat("Before", @CRLF, a,@CRLF,@CRLF,"After", @CRLF, b)
Message("StrReplace",newlist)
```

See Also:

[StrIndex](#), [StrScan](#), [StrSub](#), [StrIndexWild](#)

StrScan

Searches string for occurrence of delimiters.

Syntax:

StrScan (string, delimiters, start, direction)

Parameters:

- | | |
|----------------|---|
| (s) string | the string that is to be searched. |
| (s) delimiters | a string of delimiters to search for within string . |
| (i) start | the position in the main string to begin search.
The first character of a string is position 1. |
| (i) direction | the search direction. @FWDSCAN searches forward,
while @BACKSCAN searches backwards. |

Returns:

- | | |
|-----|---|
| (i) | 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:

```
; Parse a string with multiple delimiters into standard param format
thestr = "123,456.789:abc"
length=StrLen(thestr)
start = 1
count=0
while @TRUE
    finish = StrScan(thestr, ",.:", start, @FWDSCAN)
    If finish == 0
        break
    else
        count = count+1
        param%count% = StrSub(thestr, start, finish - start)

start=finish+1
    Message("Parameter number %count% is", param%count%)
    If finish == length then Break
    endif
endwhile
If start <= length
    finish = length+1
    count = count+1
    param%count% = StrSub(thestr, start, finish - start)
    Message("Parameter number %count% is", param%count%)
endif
param0 = count
Message("Parameter count is",param0)
```

See Also:

[StrLen](#), [StrSub](#), [StrIndexWild](#)

StrSub

Extracts a sub-string out of an existing string.

Syntax:

StrSub (string, start, length)

Parameters:

- (s) string the string from which the sub-string is to be extracted.
- (i) start character position within **string** where the sub-string starts.
(The first character of the string is at position 1).
- (i) length length of desired sub-string. If you specify a length of zero
it will return a null string. If you specify a length of -1 it will extract
the rest of the string.

Returns:

- (s) sub-string of parameter string.

This function extracts a sub-string from within a "target" string. Starting at the "start" position, it copies up to "length" characters into the sub-string.

Example:

```
a = "My dog has fleas"  
animal = StrSub(a, 4, 3)  
Message("STRSUB", "My animal is a %animal%")
```

See Also:

[StrLen](#), [StrScan](#)

StrSubWild

Extracts a substring matching wildcarded text from within a larger string.

Syntax:

```
StrSubWild(string, pattern, start)
```

Parameters:

- (s) string the string from which the sub-string is to be extracted.
- (i) start character position within **string** where the sub-string starts.
(The first character of the string is at position 1).
- (i) length length of desired sub-string. If you specify a length of zero it will return a null string. If you specify a length of -1 it will extract the rest of the string.

Returns:

- (s) The portion of the string matched by the wildcard pattern.

In the wildcard pattern, "*" matches zero or more characters, and "?" matches any one character.

Example:

```
StrSubWild("abcd1234wxyz", "ab*wx", 1)    ; would return "abcd1234wx"  
StrSubWild("abcd1234wxyz", "1*9", 1)     ; would return ""  
StrSubWild("abcd1234wxyz", "1*4", 5)     ; would return "1234"  
StrSubWild("abcd1234wxyz", "1?34*z", 5)  ; would return "1234wxyz"
```

See Also:

[StrLen](#), [StrScan](#), [StrSub](#)

StrTrim

Removes leading and trailing blanks from a character string.

Syntax:

StrTrim (string)

Parameters:

(s) string a string with unwanted spaces at the beginning and/or end.

Returns:

(s) string devoid of leading and trailing spaces.

This function removes spaces and tab characters from the beginning and end of a text string.

Example:

```
mydata = ""
while mydata != "exit"
    mydata = AskLine("STRTRIM", "Type stuff ('exit' quits)", "")
    mydata = StrTrim(mydata)
    Display(4,"STRTRIM",>%mydata%<)
endwhile
exit
```

See Also:

[StrFill](#), [StrFix](#), [StrLen](#)

StrUpper

Converts a string to uppercase.

Syntax:

StrUpper (string)

Parameters:

(s) string any text string.

Returns:

(s) 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](#)

Switch

The **Switch** statement allows selection among multiple blocks of statements.

Syntax:

```
Switch expression
  case expression
    statements
    break
  case expression
    statements
    break
```

EndSwitch

Parameters:

(s) expression an expression that must evaluate to an integer.

The **Switch** statement allows selection among multiple blocks of statements, depending on the value of an expression. The expression must evaluate to an integer.

The **Switch** statement causes the statements in the switch body to be scanned by the parser as it attempts to find a **case** statement. When a **case** statement is found, the expression following the **case** statement is evaluated, and if the expression evaluates to the same value as the expression following the **Switch** statement, execution of the following statements is initiated. The **EndSwitch** statement terminates the **Switch** structure.

If a matching **case** expression was found, and execution was initiated, the following statements will affect continued execution:

Break	Terminates the Switch structure and transfers control to the statement following the next matching EndSwitch .
Continue	Stops execution and resumes scanning for a case statement.
Case	Ignored. Treated as a comment
EndSwitch	Terminates the Switch structure and transfers control to the next statement.

Note: **Switch** and **Select** may be used interchangeably. They are synonyms for the same statement.

EndSwitch, **EndSelect**, **"End Switch"**, and **"End Select"** may be used interchangeably.

Example:

```
response=AskLine("Switch", "Enter a number between one and three", 1)
Switch response
  case 1
    Message("Switch", "Case 1 entered")
    break
  case 2
    Message("Switch", "Case 2 entered")
    break
  case 3
    Message("Switch", "Case 3 entered")
    break
  case response ; default case
    Message("Switch", "Default case entered")
    break
EndSwitch
```

See Also:

If, For, GoSub, While

Tan

Calculates the tangent.

Syntax:

Tan(x)

Parameters:

(f) x angle in radians.

Returns:

(f) the **Tan** function returns the tangent of x .

Calculates the tangent. If x is large, a loss in significance in the result or significance error may occur.

Note: To convert an angle measured in degrees to radians, simply multiply by the constant **@Deg2Rad**.

Example:

```
real=AskLine("Tangent", "Enter an angle between 0 and 360", "45")
answer=tan(real * @Deg2Rad)
Message("Tangent of %real% degrees is", answer)
```

See Also:

[Acos](#), [Asin](#), [Atan](#), [Cos](#), [Sin](#), [Tanh](#)

Tanh

Calculates the hyperbolic tangent.

Syntax:

Tanh(*x*)

Parameters:

(f) *x* angle in radians.

Returns:

(f) the **Tanh** function returns the hyperbolic tangent of *x*.

Calculates the hyperbolic tangent. There is no error value.

Note: To convert an angle measured in degrees to radians, simply multiply by the constant **@Deg2Rad**.

Example:

```
real=AskLine("TanH", "Enter an angle between 0 and 360", "45")
answer=Tanh(real * @Deg2Rad)
Message("Hyperbolic Tangent of %real% degrees is", answer)
```

See Also:

[Acos](#), [Asin](#), [Atan](#), [Cos](#), [Cosh](#), [Sin](#), [Sinh](#), [Tan](#)

Terminate

Conditionally ends a WIL program.

Syntax:

Terminate (expression, title, message)

Parameters:

- (s) expression any logical expression.
- (s) title the title of a message box to be displayed before termination.
- (s) message the message in the message box.

Returns:

- (i) always 1.

This command ends processing for the WIL program if "expression" is nonzero. Note that many functions return **@TRUE (1)** or **@FALSE (0)**, which you can use to decide whether to cancel a menu item.

If either "title" or "message" contains a string, a message box with a title and a message is displayed before exiting.

Examples:

```
; basically a no-op:
Terminate(@FALSE, "", "This will never terminate")
Message("Terminate", " Terminated.")

; exits w/o message if answer isn't "YES":
Terminate(answer != "YES", "", "")
Message("Terminate", " Terminated.")

; unconditional termination w/o message box (same as Exit)
Terminate(@TRUE, "", "")
Message("Terminate", " Terminated.")

; exits with message if variable is less than zero:
Terminate(a < 0, "Error", "Cannot use negative numbers")
Message("Terminate", " Terminated.")
```

See Also:

[Display](#), [Exit](#), [Message](#), [Pause](#)

TimeAdd

Adds two YmdHms variables

Syntax:

```
TimeAdd(datetime, datetime difference)
```

Parameters:

- | | |
|-------------------------|---|
| (s) datetime | a datetime using the format of YYYY:MM:DD:HH:MM:SS. |
| (s) datetime difference | a datetime to be added to the original using the same format. |

Returns:

- | | |
|--------------|----------------|
| (s) datetime | a new datetime |
|--------------|----------------|

Use this function to add a specified date/time to an original date/time. **TimeAdd** uses normalized conversion so a valid date/time will be returned.

Example:

```
Now=TimeYmdHms ( )
AddTime = "00:00:00:157:00:00" ; 157 hours
Later=TimeAdd(Now, AddTime)
Message("157 hours from now will be", Later)
```

See Also:

[FileTimeGet](#), [TimeDate](#), [TimeYmdHms](#), [TimeDiffSecs](#), [TimeDelay](#), [TimeWait](#), [TimeSubtract](#)

TimeDate

Provides the current date and time in a **human-readable format**. For computations with times and dates the **TimeYmdHms** function should be used instead.

Syntax:

```
TimeDate ( )
```

Parameters:

none

Returns:

(s) the current date and time.

This function will return the current date and time in a pre-formatted string. The format of the string depends on the current settings in the [Intl] section of the WIN.INI file or depending on what is set in the user's current International/Regional settings.

```
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)
```

Windows provides even more formatting options than this.

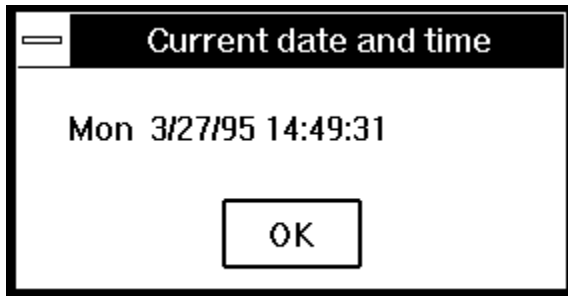
The WIN.INI file will be examined to determine which format to use. You can adjust the WIN.INI file via the [Intl] section of **Control Panel** if the format isn't what you prefer.

Note: This function is the same as the **DateTime** function, which it replaces.

Example:

```
a=Timedate( )
Message("Current date and time", a)
```

would produce:



See Also:

[FileTimeGet](#), [TimeAdd](#), [TimeYmdHms](#), [TimeDiffSecs](#), [TimeDelay](#), [TimeWait](#)

TimeDayofWeek

Using the WIL language it is possible to grab the day of the week. The following example from TimeJulianDay grabs the current time and day of the week.

Example:

```
;This example grabs the Time/Day of the Week.  
a=TimeYmdHms( )  
b=TimeJulianDay(a)  
c=(b+5) mod 7  
day=ItemExtract(c+1, "Sun Mon Tue Wed Thu Fri Sat", " ")  
line=StrCat("Julian Date-> ", b,@CRLF,"Day of week-> ",day)  
Message(TimeDate( ), line)
```

See Also:

[TimeDayofYear](#), [TimeJulianDay](#)

TimeDayofYear

Using the WIL language it is possible to grab the day of the year. The following example from TimeJulianDay grabs the current time and day of the year.

Example:

```
;TimeDayOfYear  
  
now=TimeYmdHms()  
year=ItemExtract(1,now,":")  
Jan01=strcat(year,":01:01:00:00:00")  
dayofyear=TimeDiffDays(now,Jan01)+1  
Message("Day of Year",dayofyear)
```

See Also:

[TimeDayofWeek](#)

TimeDelay

Pauses execution for a specified amount of time.

Syntax:

```
TimeDelay(seconds)
```

Parameters:

(i) seconds seconds to delay (1 - 3600).

Returns:

(i) always 1.

This function causes the currently-executing WIL program to be suspended for the specified period of time. **Seconds** must be a number between 0 and 3600. Seconds can be a floating point number: TimeDelay(2.5).

Example:

```
Message("Wait", "About 15 seconds")
TimeDelay(15)
Message("Hi", "I'm Baaaaaaack")
```

See Also:

[TimeWait](#), [Yield](#)

TimeDiff

Returns the difference between two points in time.

Syntax:

```
TimeDiff( time-value-1, time-value-2)
```

Parameters:

- (s) valid date-time strings. (YMDHMS format)
- (s) valid date-time strings. (YMDHMS format)

Returns:

- (s) the difference between the two times, in YmdHms format.

"time-value-1" and "time-value-2" must be valid date-time strings, in YmdHms format. "time-value-1" must be the later (more recent) of the two times.

Because some months have more days than others, an adjustment may need to be made when converting the resulting "day" field into months. In the example:

```
TimeDiff("1998:09:30:00:00:00", "1998:08:31:00:00:00")
```

the result is, logically, "0000:00:30:00:00:00" (30 days). But in this example:

```
TimeDiff("1998:10:01:00:00:00", "1998:08:31:00:00:00")
```

where the operation wraps past the end of the month, there is some question what the result should be, since there is no such date as September 31. This function handles this by treating the period from August 31 to September 30 as one month, so the result would be "0000:01:01:00:00:00" (one month and one day).

Example:

```
How long has it been since the beginning of the decade
diff = TimeDiff(TimeYmdHms(), "1990:01:01:00:00:00")
Message("How long since beginning of decade?",diff)
```

See Also:

[FileTimeGet](#), [TimeDiffDays](#), [TimeDiffSecs](#)

TimeDiffDays

Returns the difference in days between the two dates.

Syntax:

TimeDiffDays (datetime1, datetime2)

Parameters:

(s) datetime1 uses format YYYY:MM:DD:HH:MM:SS, or YYYY:MM:DD
(s) datetime2 uses format YYYY:MM:DD:HH:MM:SS, or YYYY:MM:DD

Returns:

(i) integer the difference in days between the two dates.

Use this function to return the difference in days between two dates. Hours, mins, secs, if specified, are ignored.

Example:

```
;Shopping days til Christmas
Now=TimeYmdHms( )            ; Get current time
Year=ItemExtract(1, Now, ":")
Xmas=strcat(Year, ":12:25:00:00:00")
Shopping=TimeDiffDays(Xmas, Now)
if Shopping>0
    Message("Shopping Days to Christmas", Shopping)
else
    if Shopping<0
        Message("You missed it by", abs(Shopping))
    else
        Message("Merry Christmas", "And a Happy New year")
    endif
endif
```

See Also:

[FileTimeGet](#), [TimeDate](#), [TimeAdd](#), [TimeYmdHms](#), [TimeDiffSecs](#), [TimeDelay](#), [TimeWait](#)

TimeDiffSecs

Returns time difference in seconds between the two datetimes.

Syntax:

```
TimeDiffSecs(datetime1, datetime2)
```

Parameters:

(s) datetime1 use format YYYY:MM:DD:HH:MM:SS.
(s) datetime2 use format YYYY:MM:DD:HH:MM:SS.

Returns:

(i) integer the difference in seconds between the two times.

Use this function to return the time difference between two datetimes. The time difference should not exceed 68 years or else an error will occur.

Example:

```
Now=TimeYmdHms ( )  
Midnight=strcat (strsub (Now,1,11), "00:00:00")  
Seconds=TimeDiffSecs (Now, Midnight)  
Message ("Seconds since midnight", Seconds)
```

See Also:

[FileTimeGet](#), [TimeDate](#), [TimeAdd](#), [TimeDiffDays](#), [TimeYmdHms](#), [TimeDelay](#), [TimeWait](#)

TimeJulianDay

Returns the Julian day given a datetime.

Syntax:

```
TimeJulianDay(datetime)
```

Parameters:

(s) datetime use format YYYY:MM:DD.

Returns:

(i) the Julian day.

Use this function to return the Julian date given a datetime. The Julian date is often used in banking and similar calculations as it provides an easy way to compute the difference between two dates.

Example:

```
;This example grabs the Time/Day of the Week.  
a=TimeYmdHms( )  
b=TimeJulianDay(a)  
c=(b+5) mod 7  
day=ItemExtract(c+1, "Sun Mon Tue Wed Thu Fri Sat", " ")  
line=StrCat("Julian Date-> ", b,@CRLF,"Day of week-> ",day)  
Message(TimeDate( ), line)
```

See Also:

[FileTimeGet](#), [TimeDate](#), [TimeAdd](#), [TimeDiffDays](#), [TimeYmdHms](#), [TimeDelay](#), [TimeWait](#),
[TimeJulToYmd](#)

TimeJulToYmd

Returns the datetime given a Julian day.

Syntax:

```
TimeJulToYmd(julian-date)
```

Parameters:

(i) julian-date a Julian date.

Returns:

(s) the datetime corresponding to the specified Julian date.

This function converts the specified (numeric) Julian date value to a datetime in YmdHms format. The "Hms" portion of the returned YmdHms string will always be "00:00:00".

Example:

```
today = TimeYmdHms( )
jul_today = TimeJulianDay(today)
jul_lastweek = jul_today - 7
lastweek = TimeJulToYmd(jul_lastweek)
FileTimeSet("stuff.txt", lastweek)
Message("TimeJulToYmd", "Done.")
```

See Also:

[TimeJulianDay](#)

TimeSubtract

Subtracts one YmdHms variable from another.

Syntax:

```
TimeSubtract( time-value, time- difference)
```

Parameters:

- (s) time-value a datetime using the format of YYYY:MM:DD:HH:MM:SS.
- (s) time difference a datetime to be subtracted from the original using the same format

Returns:

- (s) a time string in YmdHms format.

Use this function to subtract a specified date/time from an original date/time. TimeSubtract uses normalized conversion so a valid date/time will be returned. "datetime difference" can not be larger than "datetime".

TimeSubtract is designed to subtract a time-difference from a time-value:

"time-difference" is an amount of time to be subtracted from time-value", in YmdHms format.

TimeSubtract is NOT designed to determine the difference between two points in time. For that, use TimeDiffDays, TimeDiffSecs, or the new TimeDiff function

Example:

```
; Subtract 1 day from 9/1/98 (at midnight)
TimeSubtract("1998:09:01:00:00:00", "00:00:01:00:00:00")

; Subtract 48 hours from the current time
TimeSubtract(TimeYmdHms(), "00:00:00:48:00:00")

Message("TimeSubtract","Time Subtracted.")
```

See Also:

[TimeAdd](#)

TimeWait

Pauses execution and waits for the date/time to pass.

Syntax:

TimeWait (YmdHms)

Parameters:

(s) YmdHms use format YYYY:MM:DD:HH:MM:SS.

Returns:

(i) always 1.

Use this function to pause execution to wait for the datetime to pass.

To wait for the next occurrence of the specified time, (i.e., today or tomorrow), specify "00:00:00:H:M:S" for the date.

Example:

```
a=TimeYmdHms( ) ; Gets Current Time
b=TimeAdd(a,"00:00:00:00:00:07") ; Adds 7 seconds to current time
TimeWait(b) ; Waits for that time to occur
Display(3, "Time now should be", b)
```

See Also:

[FileTimeGet](#), [TimeDate](#), [TimeAdd](#), [TimeDiffDays](#), [TimeDiffSecs](#), [TimeYmdHms](#), [TimeDelay](#)

TimeYmdHms

Returns current date/time in the datetime format.

Syntax:

```
TimeYmdHms ( )
```

Parameters:

none

Returns:

(s)datetime uses format YYYY:MM:DD:HH:MM:SS.

Use this function to return the current date and time in the datetime format.

Example:

```
a=TimeYmdHms ( )  
Message("Time is", a)
```

See Also:

[FileTimeGet](#), [TimeDate](#), [TimeAdd](#), [TimeDiffSecs](#), [TimeDelay](#), [TimeWait](#)

Version

Returns the version number of the parent program currently running.

Syntax:

```
Version ( )
```

Parameters:

(none)

Returns:

(s) parent program version number.

Use this function to determine the version of the parent program that is currently running.

Example:

```
ver = Version( )  
Message("Version number", ver)
```

See Also:

[DOSVersion](#), [Environment](#), [FileVerInfo](#), [VersionDLL](#) [WinVersion](#)

VersionDLL

Returns the version number of the WIL Interpreter currently running.

Syntax:

```
VersionDLL( )
```

Parameters:

(none)

Returns:

(s) WIL Interpreter version number.

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 latest version of the language will operate properly on what may be a different machine with a different version of the WIL Interpreter installed.

Example:

```
ver = VersionDLL( )
If ver < "2.0a"
    Message("Sorry", "WIL version 2.0a or higher required")
    Exit
endif
Message("VersionDll", "Done.")
;AddExtender("extender.dll")
```

See Also:

[DOSVersion](#), [Environment](#), [FileVerInfo](#), [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.

Note: Certain keys, such as **{ALT}** and **{F10}** may not work with this function and should be avoided.

Example:

```
k = WaitForKey("{F11}", "{F12}", "{INSERT}", "", "")
switch k
  case 1
    Message("WaitForKey", "You pressed the F11 key")
    break
  case 2
    Message("WaitForKey", "You pressed the F12 key")
    break
  case 3
    Message("WaitForKey", "You pressed the Insert key")
    break
endswitch
```

See Also:

[IgnoreInput](#), [IsKeyDown](#)

WallPaper

Changes the Windows wallpaper.

Syntax:

WallPaper (bmp/html-name, tile)

Parameters:

- (s) bmp/html-filename Name of the BMP/HTML wallpaper file.
- (i) tile **@TRUE** if wallpaper should be tiled;
@FALSE if wallpaper should not be tiled.
2 - if the wallpaper should be stretched.
(only available in Windows 98 and in Windows 95 with "Plus")

Returns:

- (i) always 0.

This function immediately changes the Windows wallpaper. It can even be used for wallpaper "slide shows".

This function supports the Active Desktop wallpaper in Internet Explorer 4.0.

Example:

```
IntControl(29, @TAB, 0, 0, 0)
DirChange("c:\windows")
a = FileItemize("*.bmp")
a = AskItemList("Select New paper", a, @TAB, @unsorted, @single)
tile = @FALSE
If FileSize(a) < 40000 Then tile = @TRUE
Wallpaper(a, tile)
```

See Also:

[WinParmSet](#)

While

Conditionally and/or repeatedly executes a series of statements.

Syntax:

```
While termination-condition
    series
    of
    statements
EndWhile
```

Parameters:

(s) termination-condition	an expression to be evaluated.
(s) series of statements	statements to be executed repeatedly until the condition following the While keyword evaluates to @FALSE .

The **While** statement causes a series of statements to be repeatedly executed until the termination condition evaluates to zero or **@FALSE**. The test of the termination condition takes place before each execution of the loop. A **While** loop executes zero or more times, depending on the termination condition.

The following statements affect continued execution:

Break	Terminates the While structure and transfers control to the statement following the next matching EndWhile .
Continue	Returns to the While statement and re-evaluates the expression.
EndWhile	Returns to the While statement and re-evaluates the expression.

Note: **EndWhile** and "**End While**" may be used interchangeably.

Example:

```
a=10
while a>5
    Display(3, "The value of a is now", a)
    a=a-1
endwhile
Message("The value of a should now be 5",a)
```

See Also:

[If](#), [For](#), [GoSub](#), [Switch](#), [Select](#)

WinActivate

Activates a previously running parent window.

Syntax:

```
WinActivate (partial-winname)
```

Parameters:

(s) partial-winname either an initial portion of, or an entire window name. The most-recently used window whose title matches the name will be activated.

Returns:

(i) **@TRUE** if a window was found to activate.

Use this function to activate windows for user input.

This function works only with top-level (parent) application windows.

Example:

```
Run ("notepad.exe", "")  
WinActivate ("~Notepad")
```

See Also:

[WinActivChild](#), [WinGetActive](#), [WinName](#), [WinShow](#)

WinActivChild

Activates a previously running child window.

Syntax:

```
WinActivChild(main windowname, child windowname)
```

Parameters:

- (s) main windowname the initial part of, or an entire parent window name.
- (s) child windowname the initial part of, or an entire child window name.

Returns:

- (i) **@TRUE** if the window was found to activate.

Use this function to activate a child window for user input. The most recently used window whose title matches the name will be activated.

Note: The partial window name you give must match the initial portion of the window name (as it appears in the title bar) exactly, including proper case (upper or lower) and punctuation. The parent window must exist or this function will return an error.

Example:

```
WinActivChild("Program Manager", "Main")
```

See Also:

[WinActivate](#), [WinGetActive](#), [WinShow](#)

WinArrange

Arranges, tiles, and/or stacks application windows.

Syntax:

WinArrange (style)

Parameters:

- (i) style one of the following: **@STACK**, **@TILE** (or **@ARRANGE**), **@ROWS**, or **@COLUMNS**.

Returns:

- (i) always 1.

Use this function to rearrange the open windows on the screen. (Any iconized programs are unaffected.) If there are more than four open windows and you specify **@ROWS**, or if there are more than three open windows and you specify **@COLUMNS**, **@TILE** will be used instead. This function works only with top-level (parent) application windows.

Example:

```
; Reveal all windows  
WinArrange (@TILE)
```

See Also:

[WinHide](#), [WinIconize](#), [WinItemize](#), [WinPlace](#), [WinShow](#), [WinZoom](#)

WinClose

Closes an open window.

Syntax:

WinClose (partial-winname)

Parameters:

(s) partial-winname either an initial portion of, or an entire window name. The most-recently used window whose title matches the name will be closed.

Returns:

(i) @TRUE if a window was found to close.

Use this function to close windows.

WinClose will not close the window which contains the currently executing WIL program. You can, however, use **EndSession** to end the current Windows session.

This function works only with top-level (parent) application windows.

Example:

```
Run("notepad.exe", "")
Message("WinClose","Window is about to close.")
WinClose("~Notepad")
```

See Also:

[EndSession](#), [WinHide](#), [WinIconize](#), [WinItemize](#), [WinWaitClose](#)

WinExeName

Returns the name of the executable file which created a specified window.

Syntax:

WinExeName (partial-winname)

Parameters:

(s) partial-winname the initial part of, or an entire, window name.

Returns:

(s) name of the EXE file.

Returns the name of the EXE file which created the first window found whose title matches "partial-winname".

"Partial-winname" 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-winname" matches only one existing window.

A partial-winname of "" (null string) specifies the window making the current call to the WIL Interpreter.

This function works only with top-level (parent) application windows.

Example:

```
prog = WinExeName("~Notepad")
WinClose("~Notepad")
TimeDelay(5)
Run(prog, "")
```

See Also:

[AppExist](#), [AppWaitClose](#), [Run](#), [WinExist](#), [WinGetActive](#), [WinName](#)

WinExist

Tells if specified window exists.

Syntax:

WinExist (partial-winname)

Parameters:

(s) partial-winname the initial part of, or an entire, window name.

Returns:

(i) **@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. This function works only with top-level (parent) application windows.

Example:

```
if WinExist("~Notepad") == @FALSE Then RunIcon("Notepad", "")
```

See Also:

[AppExist](#), [WinActivate](#), [WinClose](#), [WinExeName](#), [WinExistChild](#), [WinGetActive](#), [WinItemize](#),
[WinState](#)

WinExistChild

Tells if specified child window exists.

Syntax:

```
WinExistChild ("partial-parent-windowname", "partial-child-windowname")
```

Parameters:

- (s) partial-parent- windowname the initial part of, or an entire parent window name.
- (s) partial-child-windowname the initial part or, or an entire child window name.

Returns:

- (i) **@TRUE** if a matching window is found;
 @FALSE if a matching window is not found.

Use this function to test for the existence of a child window.

Note: The partial window names you give must match the initial portion of the window name exactly, as it appears in the title bar, including proper case (upper or lower) and punctuation. The parent window must exist or this function will return an error.

Example:

```
ans=WinExistChild("Program Manager", "Main")
if ans==@TRUE then Message("Main Group", "exists in Program Manager")
                  else Message("Main Group", "seems to have been deleted")
```

See Also:

[AppExist](#), [WinActivate](#), [WinClose](#), [WinExeName](#), [WinGetActive](#), [WinItemize](#), [WinItemChild](#),
[WinState](#)

WinGetActive

Gets the title of the active window.

Syntax:

```
WinGetActive ( )
```

Parameters:

(none)

Returns:

(s) title of active window.

Use this function to determine which window is currently active.

Example:

```
currentwin = WinGetActive( )  
Message("WinGetActive", "Current window is %currentwin%.")
```

See Also:

[WinActivate](#), [WinExeName](#), [WinItemize](#), [WinName](#), [WinPlaceGet](#), [WinPosition](#), [WinTitle](#)

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.

This command can be used to perform several functions from a Windows help (.HLP) file. 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 (""), 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", "AskItemList")
```

WinHide

Hides a window.

Syntax:

WinHide (partial-winname)

Parameters:

(s) partial-winname either an initial portion of, or an entire window name.
The most-recently used window whose title matches
the name will be hidden.

Returns:

(i) **@TRUE** if a window was found to hide.

Use this function to hide windows. The programs are still running when they are hidden.
A partial-window name of "" (null string) hides the window making the current call to the WIL Interpreter.
This function works only with top-level (parent) application windows.

Example:

```
Run("notepad.exe", "")
WinHide("~Notepad")
TimeDelay(3)
WinShow("~Notepad")
```

See Also:

[RunHide](#), [WinClose](#), [WinIconize](#), [WinPlace](#)

WinIconize

Iconizes a window.

Syntax:

WinIconize (partial-winname)

Parameters:

(s) partial-winname either an initial portion of, or an entire window name. The most-recently used window whose title matches the name will be iconized.

Returns:

(i) @TRUE if a window was found to iconize.

Use this function to turn a window into an icon at the bottom of the screen.

A partial-window name of "" (null string) iconizes the current WIL Interpreter window.

This function works only with top-level (parent) application windows.

Example:

```
Run("notepad.exe", "")  
WinIconize("~Not") ; partial window name used here
```

See Also:

[RunIcon](#), [WinClose](#), [WinHide](#), [WinPlace](#), [WinShow](#), [WinZoom](#)

WinIdGet

Returns a unique "Window ID" (pseudo-handle) for the specified window name.

Syntax:

```
WinIdGet(partial-winname)
```

Parameters:

(s) partial-winname the initial part of, or an entire, window name.

Returns:

(s) the unique "Window ID".

Use this function to obtain the unique "Window ID" (pseudo-handle) for the specified parent window name. All functions which accept a partial window name as a parameter now accept the Window ID obtained with **WinIdGet**. This can be useful to distinguish between multiple windows with the same name, or to track a window whose title changes.

Example:

```
Run("notepad.exe", "")
winid1 = WinIdGet("~Notepad") ; gets the most-recently-accessed Notepad
Run("notepad.exe", "")
winid2 = WinIdGet("~Notepad") ; gets the most-recently-accessed Notepad
WinPlace(0, 0, 500, @ABOVEICONS, winid1)
WinPlace(500, 0, 1000, @ABOVEICONS, winid2)
WinActivate(winid1)
```

See Also:

[DllHwnd](#), [WinExist](#), [WinGetActive](#), [WinItemNameId](#), [WinTitle](#)

WinIsDOS

Tells whether or not a particular window is a DOS or console-type window.

Syntax:

```
WinIsDOS("partial-winname")
```

Parameters:

(s) partial-winname the initial part of, or an entire, window name.

Returns:

(i) **@TRUE** if the window is a DOS window.
 @FALSE if it is not a DOS window.

Use this function to determine if the application is in DOS or Windows.

Note: "Partial-winname" 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-winname" matches only one existing window. If it matches more than one window, the most recently accessed window which it matches will be used.

Example:

```
Run("command.com", "")
delay(2)
title="COMMAND"
if WinExist("MS-DOS") then title="MS-DOS"
if WinExist("Command") then title="Command"
a=WinIsDOS(title)
if a=@true then message(title, "is a DOS window")
              else message(title, "is NOT a DOS window")
```

See Also:

[WinExeName](#), [WinExist](#), [WinGetActive](#), [WinItemize](#), [WinName](#), [WinState](#), [WinTitle](#)

WinItemChild

Returns a list of all the child windows under this parent.

Syntax:

```
WinItemChild("partial-parent-windowname")
```

Parameters:

(s) partial-parent-windowname the initial part of, or an entire, window name.

Returns:

(s) a list of all the child windows under the parent.

Use this function to return a tab-delimited list of all child windows existing under a given parent window.

Note: "Partial-parent-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-parent-windowname" matches only one existing window. If it matches more than one window, the most recently accessed window which it matches will be used.

Example:

```
grplist=WinItemChild("Program Man")
AskItemList("Progman Groups", grplist, @TAB, @SORTED, @SINGLE)
```

See Also:

[AppExist](#), [WinActivate](#), [WinClose](#), [WinExeName](#), [WinGetActive](#), [WinItemize](#), [WinState](#)

WinItemize

Returns a tab-delimited list of all open windows.

Syntax:

```
WinItemize ( )
```

Parameters:

(none)

Returns:

(s) 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 **AskItemList** function, which enables the user to choose an item from such a tab-delimited list.

Note: Window titles regularly contain embedded spaces causing **WinItemize** to behave differently from **FileItemize** and **DirItemize** which can create space-delimited lists.

This function works only with top-level (parent) application windows. See [WinItemChild](#) to work with child windows.

Example:

```
; Find a window
allwins = WinItemize( )
mywind = AskItemList("Windows", allwins, @TAB, @unsorted, @single)
WinActivate(mywind)
```

See Also:

[DirItemize](#), [FileItemize](#), [AskItemList](#), [WinClose](#), [WinGetActive](#), [WinItemNameId](#), [WinName](#), [WinPlaceGet](#), [WinPosition](#)

WinItemizeEx

Returns the full name of window(s) matching a partial window name.

Syntax:

WinItemizeEx (partial-winname, multiple, hidden)

Parameters:

- (s) partial-winname specifies a partial window name to look for. It can be a blank string (""), in which case it will match any window.
- (i) multiple @TRUE or @FALSE. See below.
- (i) hidden @TRUE or @FALSE See below.

Returns:

- (s) list of the titles of all open windows.

If 'multiple' is @TRUE, then this function returns a tab-delimited list of all open windows that match 'partial-winname'. If 'multiple' is @FALSE, then it returns the name of the first open window found that matches 'partial-winname'. If no appropriate matching windows are found, it returns a blank string ("").

If 'hidden' is @TRUE, then hidden windows are included in the list of returned windows. If 'hidden' is @FALSE, then hidden windows are not included. If 'hidden' is @FALSE and 'multiple' is @FALSE, then this function returns the name of the first visible (non-hidden) open window found that matches 'partial-winname'.

Note: this function does not accept a Window ID as the 'partial-winname' parameter.

Example:

```
; Find all windows that are not hidden
wins = WinItemizeEx("",@TRUE,@False)
mywind = AskItemList("Windows", wins, @TAB, @unsorted, @single)
WinActivate(mywind)
```

See Also:

[DirItemize](#), [FileItemize](#), [AskItemList](#), [WinClose](#), [WinGetActive](#), [WinItemNameId](#), [WinName](#), [WinPlaceGet](#), [WinPosition](#), [WinItemize](#)

WinItemNameId

Returns a list of all open windows and their Window ID's.

Syntax:

```
WinItemNameId( )
```

Parameters:

(none)

Returns:

(s) list of the titles and Window ID's of all open windows.

This function returns a list of top-level window titles and their corresponding "Window ID's", in the form:
"window1-name|window1-ID|window2-name|window2-ID|..."

Example:

```
winlist = WinItemNameId( )  
AskItemList("Windows and ID's", winlist, "|", @SORTED,@SINGLE)
```

See Also:

[WinIdGet](#), [WinItemize](#)

WinItemProcID

Returns a list of window ID's for the specified process.

Syntax:

WinItemProcID (process-id, flags, return-type)

Parameters:

- (i) process-id returned from RunShell.
- (i) flags (see below).
- (i) return-type (see below).

Returns:

- (s) a tab-delimited list of all top-level (parent) window ID's owned by the process specified by "process-id".

You can obtain the process ID of an application that is launched with the RunShell function by specifying **@GETPROCID** as the "waitflag". This is the same as specifying **@NOWAIT**, except that on success the function will return the process ID of the application that was launched. This process ID can be used with the **WinItemProcID** function. The process ID may be a negative number. If a process ID cannot be obtained a 1 will be returned to indicate success. In all cases, a return value of 0 indicates failure.

"flags" can be 0, or one or more of the following values combined with the binary OR ("|") operator:

Flag	Meaning
1	Include windows with blank titles
2	Include hidden windows
4	Include windows which are not enabled for keyboard and mouse input
8	Include windows with the title "WinOldAp"

"return-type" specifies the format in which the list of windows will be returned, and can be one of the following:

Type	Meaning
0	Tab-delimited list of window ID's
1	Tab-delimited list of window titles
2	List of top-level window titles and their corresponding "Window ID's", in the form: "window1-name window1-ID window2-name window2-ID ..."

Example:

```
procid = RunShell("calc.exe", "", "", @NORMAL, @GETPROCID)
```

```
If (procid != 0) && (procid != 1) ; if we got a valid process ID
    winids = WinItemProcId(procid, 2, 0)
    Message("Window ID(s)", winids)
Endif
```

See Also:

[RunShell](#)

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
-6	pixels per horizontal dialog unit (rounded to an integer)
-5	pixels per vertical dialog unit (rounded to an integer)
-4	Windows Platform; 0 = Other 1 = Windows 2 = Windows for Workgroups 3 = Win32s 4 = Windows NT 5 = Windows 95
-3	WIL EXE type; 0=Win16, 1=Intel32, 2=Alpha32, 3=Mips32, 4=PowerPC32
-2	WIL platform; 1=Win16, 2=Win32
-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

Additional request #'s for WinMetrics (32-bit version only):

- 41 TRUE or non-zero if the Microsoft Windows for Pen computing extensions are installed; zero, or FALSE, otherwise.
- 42 TRUE or non-zero if the double-byte character set (DBCS) version of USER.EXE is installed; FALSE, or zero otherwise.
- 43 Number of buttons on mouse, or zero if no mouse is installed.
- 44 (Win95 only) TRUE if security is present, FALSE otherwise.
- 63 (Win95 only) The least significant bit is set if a network is present; otherwise, it is cleared. The other bits are reserved for future use.
- 67 (Win95 only) Value that specifies how the system was started:
 - 0 - Normal boot
 - 1 - Fail-safe boot
 - 2 - Fail-safe with network bootFail-safe boot (also called SafeBoot) bypasses the user's startup files.
- 70 TRUE or non-zero if the user requires an application to present information visually in situations where it would otherwise present the information only in audible form; FALSE, or zero, otherwise.
- 73 (Win95 only) TRUE if the computer has a low-end (slow) processor.
- 74 (Win95 only) TRUE if the system is enabled for Hebrew/Arabic languages.

There are a number of other request #'s which can be specified, but are of limited usefulness and therefore not documented here. Details on these can be obtained from Win32 programming references, available from Microsoft (and others).

Example:

```
mouse = "NO"  
If WinMetrics(19) == 1 Then mouse = "YES"  
Message("Is there a mouse installed?", mouse)
```

See Also:

[Environment](#), [MouseInfo](#), [WinSysInfo](#), [NetInfo](#), [WinParmGet](#), [WinResources](#)

WinName

Returns the name of the window calling the WIL Interpreter.

Syntax:

```
WinName ( )
```

Parameters:

(none)

Returns:

(s) window name.

Returns the name of the window making the current call to the WIL Interpreter.

Example:

```
allwins = WinItemize( )
win = AskItemList("Close window", allwins, @tab, @sorted, @single)
If win == WinName( )
    Message("Sorry", "I can't close myself")
else
    WinClose(win)
endif
Exit
```

See Also:

[WinActivate](#), [WinExeName](#), [WinGetActive](#), [WinItemize](#), [WinTitle](#)

WinParmGet

Returns system information.

Syntax:

WinParmGet (request#)

Parameters:

(i) request# see below.

Returns:

(s) see below.

Note: This function requires Windows 3.1 or higher.

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 DbIcClick
17	DoubleClickHeight	Allowable vert. movement in pixels for DbIcClick
18	DoubleClickSpeed	Max time in millisecs between clicks for DbIcClick
19	MouseButtonSwap	0 = Not swapped, 1 = swapped
20	Fast Task Switch	0 = Off, 1 = On

Example:

```
If WinParmGet(8) == 1 Then Message("", "Screen saver is active")
```

See Also:

[Environment](#), [MouseInfo](#), [NetInfo](#), [WinMetrics](#), [WinParmSet](#), [WinSysInfo](#), [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:

(i) previous value of the setting.

Note: This function requires Windows 3.1 or higher.

See **WinParmGet** 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  
Message("WinParmSet", " Window Parameters Set.")
```

See Also:

[WallPaper](#), [WinParmGet](#)

WinPlace

Places a window anywhere on the screen.

Syntax:

WinPlace (x-ulg, y-ulg, x-brc, y-brc, partial-winname)

Parameters:

- (i) x-ulg how far from the left of the screen to place the upper-left corner (0-1000).
- (i) y-ulg how far from the top of the screen to place the upper-left corner (0-1000).
- (i) x-brc how far from the left of the screen to place the bottom-right corner (10-1000) or **@NORESIZE**.
- (i) y-brc how far from the top of the screen to place the bottom-right corner (10-1000) or **@NORESIZE** or **@ABOVEICONS**.
- (s) partial-winname either an initial portion of, or an entire windowname. The most-recently used window whose title matches the name will be moved to the new position.

Returns:

- (i) **@TRUE** if a window was found to move.

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-ulg", "y-ulg", "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**

This function works only with top-level (parent) application windows.

Example:

```
Run ("Notepad.exe", "")  
WinPlace(0, 0, 200, 200, "~Notepad")
```

See Also:

[WinArrange](#), [WinHide](#), [WinIconize](#), [WinPlaceSet](#), [WinPosition](#), [WinShow](#), [WinZoom](#)

WinPlaceGet

Returns window coordinates.

Syntax:

WinPlaceGet (win-type, partial-winname)

Parameters:

- (i) win-type **@ICON**, **@NORMAL**, or **@ZOOMED**
- (s) partial-winname 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-winname" 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-winname" 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.

This function works only with top-level (parent) application windows.

Examples:

```
Run("notepad.exe", "")
pos = WinPlaceGet(@NORMAL, "~Notepad")
TimeDelay(2)
WinPlaceSet(@NORMAL, "~Notepad", "250 250 750 750")
TimeDelay(2)
WinPlaceSet(@NORMAL, "~Notepad", pos)
```

See Also:

[WinGetActive](#), [WinItemize](#), [WinPlaceSet](#), [WinPosition](#), [WinState](#)

WinPlaceSet

Sets window coordinates.

Syntax:

WinPlaceSet (win-type, partial-winname, position-string)

Parameters:

- (i) win-type **@ICON**, **@NORMAL**, or **@ZOOMED**
- (s) partial-winname 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-winname" 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-winname" 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.

This function works only with top-level (parent) application windows.

Examples:

```
Run ("Notepad.exe", "")
WinPlaceSet (@ICON, "~Notepad", "10 950")
```

See Also:

[WinActivate](#), [WinArrange](#), [WinPlace](#), [WinPlaceGet](#), [WinState](#)

WinPosition

Returns Window position.

Syntax:

WinPosition (partial-winname)

Parameters:

(s) partial-winname the initial part of, or an entire, winname.

Returns:

(s) 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).

This function works only with top-level (parent) application windows.

Example:

```
Run("notepad.exe", "") ; start Notepad
WinPlace(0,0,300,300, "~Notepad") ; place Notepad
pos = WinPosition("~Notepad") ; save position
TimeDelay(2)
WinPlace(200,200,300,300, "~Notepad") ; move Notepad
delay(2)
WinPlace(%pos%, "~Notepad") ; restore Notepad
```

See Also:

[WinGetActive](#), [WinItemize](#), [WinPlace](#), [WinPlaceGet](#), [WinState](#)

WinResources

Returns information on available memory and resources.

Syntax:

WinResources (request#)

Parameters:

(i) request# see below

Returns:

(i) see below.

The value of request# determines 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

for 32-bit version:

10	returns a number between 0 and 100 giving a general idea of current memory utilization, in which 0 indicates no memory use and 100 indicates full memory use.
11	total number of bytes of physical memory.
12	number of bytes of physical memory available.
13	total number of bytes that can be stored in the paging file. Note that this number does not represent the actual physical size of the paging file on disk.
14	number of bytes available in the paging file.
15	total number of bytes that can be described in the user mode portion of the virtual address space of the calling process.
16	number of bytes of unreserved and uncommitted memory in the user mode portion of the virtual address space of the calling process.

Example:

```
mem = WinResources(0)
Message("Available memory", "%mem% bytes")
```

See Also:

[WinMetrics](#), [WinParmGet](#), [WinSysInfo](#)

WinShow

Shows a window in its "normal" state.

Syntax:

WinShow (partial-winname)

Parameters:

(s) partial-winname either an initial portion of, or an entire window name. The most-recently used window whose title matches the name will be shown.

Returns:

(i) **@TRUE** if a window was found to show.

Use this function to restore a window to its "normal" size and position.

A partial-window name 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-winname)

Parameters:

(s) partial-winname the initial part of, or an entire, window name.

Returns:

(i) window state (see below).

"Partial-winname" 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-winname" 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.

<u>Value</u>	<u>Symbolic name</u>	<u>Meaning</u>
-1	@HIDDEN	Specified window exists, but is hidden
0	@FALSE	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)

This function works only with top-level (parent) application windows.

Example:

```
RunIcon("Notepad.exe", "")  
If WinState("~Notepad") == @ICON Then WinShow("~Notepad")
```

See Also:

[Run](#), [WinExist](#), [WinGetActive](#), [WinHide](#), [WinIconize](#), [WinItemize](#), [WinPlace](#), [WinPlaceGet](#), [WinPlaceSet](#), [WinPosition](#), [WinShow](#), [WinZoom](#)

WinSysInfo() {*32}

Returns system configuration information.

Syntax:

```
WinSysInfo( )
```

Parameters:

(none)

Returns:

(s) a TAB delimited list of system configuration information.

WinSysInfo returns a TAB-delimited list containing the following items:

1. computer name of the current system.
2. processor architecture.
3. page size (specifies granularity of page protection and commitment).
4. mask representing the set of processors configured into the system.
5. number of processors in the system.
6. processor type.
7. granularity in which memory will be allocated.
8. system's architecture-dependent processor level.
9. architecture-dependent processor revision.

Note: This function should be used instead of WinConfig in the 32-bit version.

Example:

```
sysinfo = WinSysInfo( )
computer = ItemExtract(1, sysinfo, @TAB)
processor = ItemExtract(6, sysinfo, @TAB)
Message(computer, "is a %processor%")
```

See Also:

[WinMetrics](#), [WinParmGet](#), [WinResources](#)

WinTitle

Changes the title of a window.

Syntax:

```
WinTitle (partial-winname, new-name)
```

Parameters:

- (s) partial-winname either an initial portion of, or an entire window name. The most-recently used window whose title matches the name will be shown.
- (s) new-name the new name of the window.

Returns:

- (i) **@TRUE** if a window was found to rename.

Use this function to change a window's title.

A partial-window name 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.

This function works only with top-level (parent) application windows.

Example:

```
; Capitalize title of window
allwinds = WinItemize( )
mywin = AskItemList("Uppercase Windows", allwinds, @TAB, @unsorted, @single)
WinTitle(mywin, StrUpper(mywin))
Drop(allwinds, mywin)
```

See Also:

[WinGetActive](#), [WinItemize](#), [WinName](#)

WinVersion

Provides the version number of the current Windows system.

Syntax:

WinVersion (level)

Parameters:

(i) level see below.

Returns:

(i) Windows version number.

Level	Returns
0	(i) Minor version: returns the decimal part of the Windows version number; i.e. 1.0, 2.11, 3.0, etc.
1	(i) Major version: returns the integer part of the Windows version number; i.e. 1.0, 2.11, 3.0, etc.
2	(i) Build number
3	(s) CSD version

Under Windows NT, CSD version is a string indicating the latest service pack that has been installed (eg, "Service Pack 1"), or a blank string ("") if no service pack has been installed. Under Windows 95, it is a string that may indicate arbitrary additional information about the operating system, or may be blank.

Example:

```
minorver = WinVersion(0)
majorver = WinVersion(1)
buildver = WinVersion(2)
csdver = WinVersion(3)
Message("Windows Version", StrCat(majorver, ".", minorver, " ", buildver, " ",
csdver))
```

See Also:

[Version](#), [DOSVersion](#), [FileVersionInfo](#)

WinWaitChild

Waits for a child window to exist.

Syntax:

WinWaitChild(partial parent winname, partial child winname, timeout)

Parameters:

- | | |
|----------------------------|---|
| (s) partial parent winname | the initial part of, or an entire parent window name. |
| (s) partial child winname | the initial part of, or an entire child window name. |
| (i) timeout | the number of seconds to wait (maximum = 86400).
Specify 0 to return immediately (no wait).
Specify -1 for no timeout (wait forever). |

Returns:

- (s) **@TRUE** if the window appeared; **@FALSE** if it didn't.

Example:

```
Run("c:\program files\winedit\winedit.exe", "")
WinWaitExist("WinEdit", 5)
TimeDelay(1)
SendMenusTo("WinEdit", "File Open")
SendKeysTo("File Open", "c:\config.sys~")
WinWaitChild("WinEdit", "c:\config.sys", 5)
SendKeysTo("WinEdit", "lastdrive=z~")
```

See Also:

[AppWaitClose](#), [TimeDelay](#), [RunWait](#), [WinExist](#), [WinWaitClose](#), [WinWaitExist](#), [Yield](#)

WinWaitClose

Suspends the WIL program execution until a specified window has been closed.

Syntax:

WinWaitClose (partial-winname)

Parameters:

(s) partial-winname either an initial portion of, or an entire window name. WinWaitClose suspends execution until all matching windows have been closed.

Returns:

(i) **@TRUE** if at least one window was found to wait for;
@FALSE if no windows were found.

Use this function to suspend the WIL program's execution until the user has finished using a given window and has manually closed it.

This function works only with top-level (parent) application windows.

Example:

```
Run("notepad.exe", "")
Display(4, "Note", "Close Notepad to continue")
WinWaitClose("~Notepad")
Message("Continuing...", "Notepad closed")
```

See Also:

[AppWaitClose](#), [TimeDelay](#), [RunWait](#), [WinExist](#), [WinWaitChild](#), [WinWaitExist](#), [Yield](#)

WinWaitExist

Waits for a window to exist.

Syntax:

```
WinWaitExist(partial-winname, timeout)
```

Parameters:

- (s) partial-winname the initial part of, or an entire, window name.
- (i) timeout the number of seconds to wait (maximum = 86400).
Specify 0 to return immediately (no wait).
Specify -1 for no timeout (wait forever).

Returns:

- (s) **@TRUE** if the window appeared; **@FALSE** if it didn't.

Example:

```
Run("notepad.exe", "")  
WinWaitExist("~Notepad", 5)  
TimeDelay(1)  
SendMenuTo("~Notepad", "File Open")  
SendKeysTo("File Open", "c:\config.sys~")
```

See Also:

[AppWaitClose](#), [TimeDelay](#), [RunWait](#), [WinExist](#), [WinWaitChild](#), [WinWaitClose](#), [Yield](#)

WinZoom

Maximizes a window to full-screen.

Syntax:

WinZoom (partial-winname)

Parameters:

(s) partial-winname either an initial portion of, or an entire window name. The most-recently used window whose title matches the name will be shown.

Returns:

(i) @TRUE if a window was found to zoom.

Use this function to "zoom" windows to full screen size.

A partial-winname of "" (null string) zooms the current WIL interpreter window.

This function works only with top-level (parent) application windows.

Example:

```
Run("notepad.exe", "")
WinZoom("~Notepad")
TimeDelay(3)
WinShow("~Notepad")
```

See Also:

[RunZoom](#), [WinHide](#), [WinIconize](#), [WinPlace](#), [WinShow](#)

Yield

Provides time for other windows to do processing.

Syntax:

Yield

Parameters:

(none)

Returns:

(not applicable)

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:

[TimeDelay](#), [TimeWait](#), [Exclusive](#), [WinWaitExist](#)

Yields

Provides time for other windows to do processing.

Syntax:

Yields(count)

Parameters:

(s) count number of yields to perform.

Returns:

(not applicable)

Use this command to give other running windows time to process. This command will allow each open window to process 20 or more messages.

This function is the same as the "Yield" command, but allows you to specify the number of yields to perform; ie, "Yields(5)" is the same as 5 consecutive "Yield" commands.

Example:

```
; run Excel and give it some time to start up
sheet = AskLine ("Excel", "File to run:", "")
Run("excel.exe", sheet)
Yields(3)
```

See Also:

Yield

yesTRUEnono&About&PrintyesyesyesyesWIL A TO K
JUMP FILEWILAKyes20/10/98

Table of Contents

[Introduction](#)
[About](#)
[Abs](#)
[Acos](#)
[AddExtender](#)
[AppExist](#)
[AppWaitClose](#)
[Asin](#)
[AskDirectory](#)
[AskFileName](#)
[AskFileText](#)
[AskItemList](#)
[AskLine](#)
[AskPassword](#)
[AskYesNo](#)
[Atan](#)
[Average](#)
[Beep](#)
[Binary Operations](#)
[BinaryAlloc](#)
[BinaryAnd](#)
[BinaryClipGet](#)
[BinaryClipPut](#)
[BinaryCompare](#)
[BinaryConvert](#)
[BinaryCopy](#)
[BinaryEodGet](#)
[BinaryEodSet](#)
[BinaryFree](#)
[BinaryHashRec](#)
[BinaryIncr](#)
[BinaryIndex](#)
[BinaryIndexEx](#)
[BinaryIndexNC](#)
[BinaryOleType](#)
[BinaryOr](#)
[BinaryPeek](#)
[BinaryPeekStr](#)
[BinaryPoke](#)
[BinaryPokeStr](#)
[BinaryRead](#)
[BinaryReadEx](#)
[BinarySort](#)
[BinaryStrCnt](#)
[BinaryTagExtr](#)
[BinaryTagFind](#)
[BinaryTagInit](#)

BinaryTagRepl
BinaryWrite
BinaryWriteEx
BinaryXor
Break
ButtonNames
Call
Ceiling
Char2Num
ClipAppend
ClipGet
ClipGetEx
ClipPut
Continue
Cos
Cosh
CurrentFile {*M}
CurrFilePath {*M}
CurrentPath {*M}
DateTime
DDEExecute
DDEInitiate
DDEPoke
DDERequest
DDETerminate
DDETimeout
Debug
DebugData
DebugTrace
Decimals
Delay
Dialog
DirAttrGet
DirAttrSet
DirChange
DirExist
DirGet
DirHome
DirItemize
DirMake
DirRemove
DirRename
DirSize
DirWindows
DiskExist
DiskFree
DiskInfo
DiskSize
DiskScan
DiskVollInfo

Display
DllCall
DllCall_Additional_information
DllFree
DllHinst
DllHwnd
DllLoad
DOSVersion
Drop
DropWild
EndSession
Environment
EnvironSet
EnvItemize
ErrorMode
Exclusive
Execute
ExeTypeInfo
Exit
Exp
Fabs
FileAppend
FileAttrGet
FileAttrSet
FileClose
FileCompare
FileCopy
FileCopyAttr
FileDelete
FileExist
FileExtension
FileFullName
FileItemize
FileLocate
FileMapName
FileMove
FileMoveAttr
FileNameLong {*32}
FileNameShort {*32}
FileOpen
FilePath
FileRead
FileRename
FileRoot
FileSize
FileTimeCode
FileTimeGet
FileTimeGetEx{*32}
FileTimeSet
FileTimeTouch

FileVerInfo
FileWrite
FileYmdHms
FindWindow
Floor
For
GetExactTime
GetTickCount
GoSub
Goto
IconReplace
If ... Else ... Endif...If ... Then ... Else
IniDelete
IniDeletePvt
IniItemize
IniItemizePvt
IniRead
IniReadPvt
IniWrite
IniWritePvt
InstallFile
Int
IntControl
IsDefined
IsFloat
IsInt
IsKeyDown
IsLicensed
IsMenuChecked {*M}
IsMenuEnabled {*M}
IsNumber
ItemCount
ItemExtract
ItemInsert
ItemLocate
ItemRemove
ItemSort
ItemSortNc
KeyToggleGet
KeyToggleSet
FileTimeSetEx{*32}

Help file produced by **HELLLP!** v2.3a , a product of Guy Software, on 10/20/98 for WILSON WINDOWWARE, INC..

The above table of contents will be automatically completed and will also provide an excellent cross-reference for context strings and topic titles. You may leave it as your main table of contents for your help file, or you may create your own and cause it to be displayed instead by using the I button on the toolbar. This page will not be displayed as a topic. It is given a context string of __ and a HelpContextID property of 32517, but these are not presented for jump selection.

HINT: If you do not wish some of your topics to appear in the table of contents as displayed to your users (you may want them ONLY as PopUps), move the lines with their titles and contexts to below this point. If you do this remember to move the whole line, not part. As an alternative, you may wish to set up your own table of contents, see Help under The Structure of a Help File.

Do not delete any codes in the area above the Table of Contents title, they are used internally by HELLLP!

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 (shown in *Courier* type), and related functions you may want to **See Also**.

The following symbols will be used to indicate a function or a section of the manual which applies only to a specific implementation of the WIL Interpreter.

{***Explorer**} Windows 95 or Windows NT 4.0 or newer, for use with the Microsoft Explorer.

{***32**} 32 bit Windows operating systems. (95 or NT)

{***M**} menu-based implementations of WIL Interpreter.

Note: WinBatch, WebBatch and Clock Manager are **batch file** applications, FileMenu, PopMenu and WinEdit are **menu file** applications.

Additional Symbols

(**i**) indicates an integer parameter or return value.

(**s**) indicates a string parameter or return value.

(**f**) indicates a floating point parameter or return value.

(**t**) indicates special type information described in the functions text.

About

Displays the about message box which gives program information.

Syntax:

About()

Parameters:

none

Returns:

(i) always 1.

This function displays a message box containing copyright and version information.

Example:

About ()

See Also:

[Version](#), [VersionDLL](#)

Abs

Returns the absolute value of an integer.

Syntax:

Abs (integer)

Parameters:

(i) integer integer whose absolute value is desired.

Returns:

(i) 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. If a floating point number is passed as a parameter, it will be converted to an integer.

Example:

```
y1=1993
y2=1996
dy = Abs(y1 - y2)
Message("Years", "There are %dy% years 'twixt %y1% and %y2%")
```

See Also:

[Average](#), [Fabs](#), [IsNumber](#), [Max](#), [Min](#)

Acos

Calculates the arccosine.

Syntax:

`Acos(x)`

Parameters:

(f) x floating point number whose arccosine is desired.

Returns:

(f) the Acos function returns the arccosine result of x .

The **Acos** function returns the arccosine of x in the range 0 to π radians. The value of x must be between -1 and 1, otherwise a domain error will occur.

Example:

```
real=AskLine("ArcCos", "Enter a real number between -1 and 1", "0.5")
answer=Acos(real)
Message("Arccos of %real% is",answer)
```

See Also:

[Asin](#), [Atan](#), [Cos](#), [Sin](#), [Tan](#)

AddExtender

Installs a WIL extender dll.

Syntax:

```
AddExtender(filename)
```

Parameters:

(s) filename WIL extender Dll filename

Returns:

(i) **@TRUE** if function succeeded

WIL extender Dlls are special Dlls designed to extend the built-in function set of the WIL processor. These Dlls typically add functions not provided in the basic WIL set, such as network commands for particular networks (Novell, Windows for WorkGroups, LAN Manager and others), MAPI, TAPI, and other important Application Program Interface functions as may be defined by the various players in the computer industry from time to time. These Dlls may also include custom built function libraries either by the original authors, or by independent third party developers. (An Extender SDK is available). Custom extender Dlls may add nearly any sort of function to the WIL language, from the mundane network, math or database extensions, to items that can control fancy peripherals, including laboratory or manufacturing equipment.

Use this function to install extender Dlls as required. Up to 10 extender Dlls may be added. The total number of added items may not exceed 200 functions and constants. The **AddExtender** function must be executed before attempting to use any functions in the extender library. The **AddExtender** function should be only executed once in each WIL script that requires it.

The documentation for the functions added are supplied either in a separate manual or disk file that accompanies the extender Dll.

Example:

```

; Add vehicle radar processing dll controlling billboard visible to
; motorists, and link to enforcement computers.
; The WIL Extender SPEED.DLL adds functions to read a radar speed
; detector(GetRadarSpeed) , put a message on a billboard visible to
; the motorist (BillBoard), take a video of the vehicle (Camera), and
; send a message to alert enforcement personnel (Alert) that a
; motorist in violation along with a picture id number to help
; identify the offending vehicle and the speed which it was going.
;
AddExtender("SPEED.DLL")
BillBoard("Drive Safely")
While @TRUE
    ; Wait for next vehicle
    while GetRadarSpeed( )<5 ; if low, then just radar noise
        Yield ; wait a bit, then look again
    endwhile
    speed=GetRadarSpeed( ) ; Something is moving out there
    if speed < 58
        BillBoard("Drive Safely") ; Not too fast.
    Else
        if speed < 63
            BillBoard("Watch your Speed") ; Hmmm a hot one
        else
            if speed < 66
                BillBoard("Slow Down") ; Tooooo fast
            else
                BillBoard("Violation Pull Over")
                pictnum = Camera( ) ; Take Video
            Snapshot
                Alert(pictnum, speed); Pull this one over
            endif
        endif
    endif
endwhile

```

See Also:

[DllCall](#)

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).

Note: In 32-bit versions of WIL, only 32-bit applications will be detected.

Example:

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

See Also:

[AppWaitClose](#), [RunWait](#), [RunShell](#), [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).

Note: In 32-bit versions of WIL, only 32-bit applications will be detected.

Example:

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

See Also:

[AppExist](#), [TimeDelay](#), [RunShell](#), [RunWait](#), [WinExeName](#), [WinWaitChild](#),
[WinWaitClose](#), [WinWaitExist](#), [Yield](#)

Asin

Calculates the arcsine.

Syntax:

`Asin(x)`

Parameters:

(f) *x* floating point number whose arcsine is desired.

Returns:

(f) the **Asin** function returns the arcsine result of *x*.

The **Asin** function returns the arcsine of *x* in the range $-\pi/2$ to $\pi/2$ radians. The value of *x* must be between -1 and 1 otherwise a domain error will occur.

Example:

```
real=AskLine("ArcSin", "Enter a real number between -1 and 1", "0.5")
answer=Asin(real)
Message("Arcsin of %real% is", answer)
```

See Also:

[Acos](#), [Atan](#), [Cos](#), [Sin](#), [Tan](#)

AskDirectory {*32-bit}[Win95/NT40 only]

Displays a directory browse dialog box, and returns the selected directory name.

Syntax:

AskDirectory(prompt, browse-root, start-dir, confirm-prompt, flags)

Parameters:

- (s) prompt: Prompt to be displayed in the dialog box above the list of directories. This can be a blank string ("") if no prompt is desired. Note that the title of the dialog box ("Browse for Folder") cannot be changed.
- (s) browse-root: Directory under which the user can browse for directories. The user will not be able to browse above this level. You can specify a blank string ("") to allow the entire file system (all drives, directories, and network shares) to be browsed.
- (s) start-dir: Directory which will be selected by default when the dialog box is initially displayed. This can be a blank string (""), in which case the top of the tree will be selected (same as 'browse-root').
- (s) confirm-prompt: The title of the confirmation message box (see 'flags' #2, below).
- (i) flags: One or more of the optional flags (see below), combined using the binary OR ("|") operator, or 0 if none are desired: title of the file name select box. It does not mean "or".

Returns:

- (s) string The string containing the directory path selected in the browse dialog. If the user presses "Cancel", standard WIL "cancel" processing will be performed

Flag	Meaning
----	-----
1	Display an edit field in the dialog box, in which the user can type the name of a directory which may or may not exist. This name will be relative to the currently selected directory name in the browse list. NOTE: This is supported only in Windows 98, Windows NT 5.0, or in earlier versions of Windows that have Internet Explorer 4.0 installed. In other versions of Windows, this flag is ignored.
2	If the user types a name in the edit field (see flag #1), of a directory which does not exist, this flag causes a confirmation

message box to be displayed, showing the name of the directory that would be returned by the function, and containing three buttons: Yes, No, and Cancel. If the user selects 'Yes', the function returns. If the user selects 'No', the directory browse dialog remains displayed, and the user can re-edit the name or select a different directory. 'Cancel' causes the function to return, and standard WIL "cancel" processing to be performed.

The title of the message box is specified by 'confirm-prompt'.

Example:

```
flags=1|2
dir1=AskDirectory("Select Working Directory for Winbatch", "C:\Program
Files\WinBatch", "", "Are you sure?", flags)
Message("Directory Path selected was", dir1)
```

which produces:



and then:



See Also:

[AskFileName](#), [AskLine](#), [DirItemize](#), [AskItemizeList](#)

AskFileName

Returns the filename with its path as selected by a FileOpen dialog box.

Syntax:

```
AskFileName(title, directory, filetypes, default filename, flag)
```

Parameters:

- | | |
|----------------------|--|
| (s) title | title of the file name select box. |
| (s) directory | initial drive and directory. |
| (s) filetypes | file type selection definition (see below)*.*
text highlighted in the File Name field of the
Open/Save dialog box or file filter.
(format: description mask description mask etc.).
Note: The " " is required to separate the descriptive
phrase from the mask. It does not mean "or". |
| (s) default filename | default filename or mask. |
| (i) flag | 0 for Save style ; 1 for Open style. |

Returns:

- | | |
|------------|--|
| (s) string | A string containing the filename and path. |
|------------|--|

This function is the equivalent of a standard Common Dialog FileOpen or a FileSave dialog box. The initial drive and directory is logged, and either a FileSave or a FileOpen dialog box is presented to the user. The default filename or mask is filled in, as well as a selection of filetypes.

The user can either type in a filename or select one via the file list box. File types displayed may be selected from a "File Type" drop down list box. The File Type contents in the drop down list box is specified by the filetypes parameter. The filetype parameter is defined as follows:

```
filetypes := Description|Mask| [ Description|Mask| ...]
Description := Any human readable string
Mask := filespec [; filespec ...]
filespec := DOS File wildcard mask
```

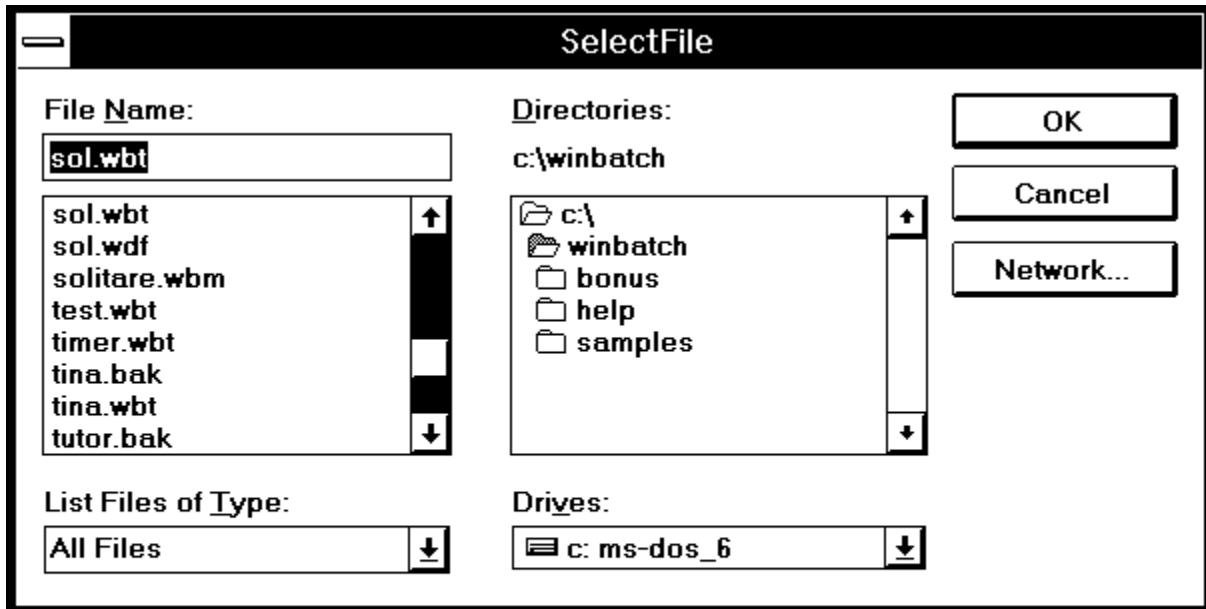
Basically, a description - visible to the user in the drop down list box, followed by the vertical bar symbol(|), followed by a file mask - for the computer, followed by another vertical bar. This description may be repeated for each desired file type selection.

The filetype parameter field must be enclosed in quotes "". No additional quotes are needed inside these. You can use several descriptions and file type masks. For only one file type, a second "|" is optional. Spaces and alphanumeric characters can be used in file type descriptions. An example is: Text files|*.txt|WinBatch|*.wbt" Several file type masks can be used for a single description. Separate them with a semicolon. An example of this: "Batch files|*.wbt;*.bat;*.cmd".

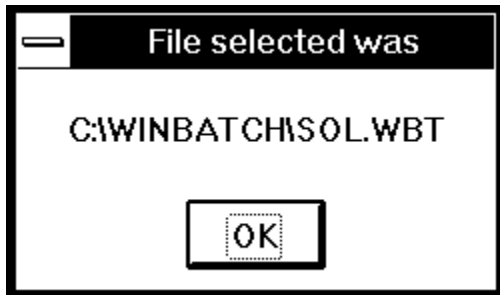
Example:

```
types="All Files|*.*|WIL Files|*.wbt;*.mnu|Text Files|*.txt|"
fn1=AskFileName("SelectFile", "C:\WinBatch", types, "Sol.wbt", 1)
Message("File selected was", fn1)
```

which produces:



and then:



See Also:

[AskFileText](#), [AskItemList](#), [AskLine](#)

AskFileText

Allows the user to choose an item from a list box initialized with data from a file.

Syntax:

```
AskFileText(title, filename, sort mode, select mode)
```

Parameters:

- | | |
|-----------------|---|
| (s) title | title of the list box. |
| (s) filename | file containing the contents of list box. |
| (i) sort mode | @sorted for an alphabetic list.
@unsorted to display the text as is. |
| (i) select mode | @single to limit selection to one item.
@multiple allow selection of more than one item. |

Returns:

- | | |
|-----|--|
| (s) | the selected item or items. If more than one item is selected it will be returned as a tab delimited list. |
|-----|--|

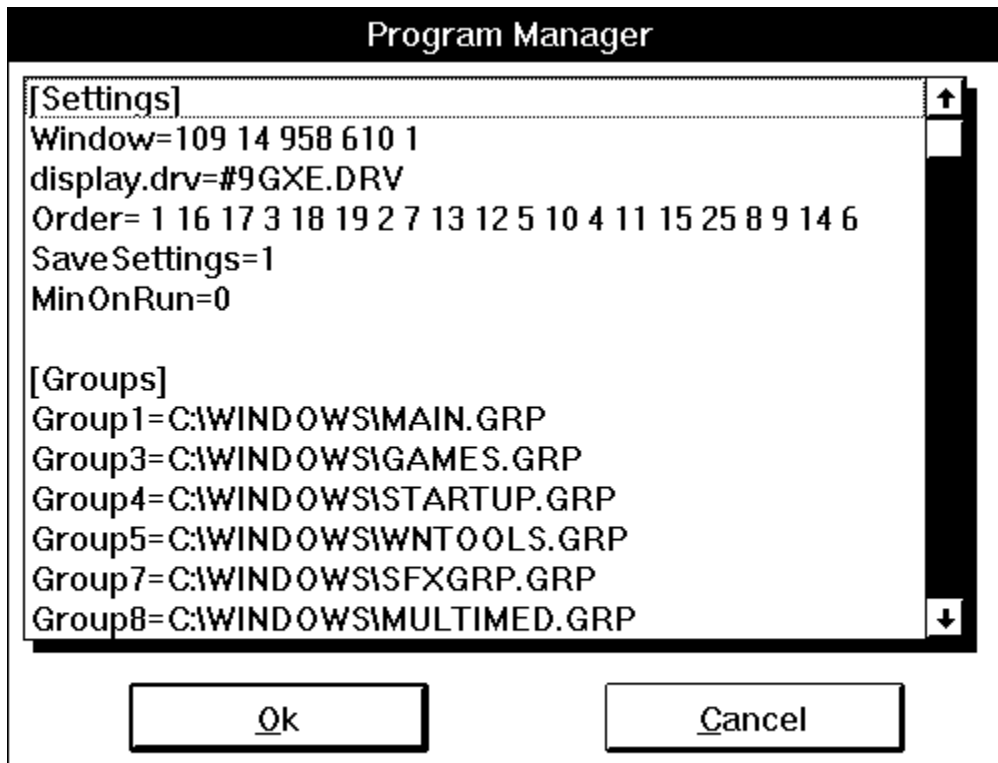
Note: This function replaces **TextBox** and **TextBoxSort**

This function loads a file into a Windows list box, either as is or sorted alphabetically, and displays the list box to the user. The line or lines highlighted by the user (if any) will be returned to the program as a tab delimited list (see [ItemExtract](#)). **AskFileText** 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 selected lines, it may be used as a multiple choice question box. 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 Environment PATH will be searched to find the desired file.

Example:

```
A=AskFileText("Program Manager", "progman.ini", @unsorted, @single)
Message("The line chosen was", A)
```

which produces:



See Also:

[AskItemList](#), [AskFileName](#), [AskLine](#)

AskItemList

Allows the user to choose an item from a list box initialized with a list variable.

Syntax:

```
AskItemList(title, list variable, delimiter, sort mode, select mode)
```

Parameters:

- | | |
|-------------------|--|
| (s) title | title of the list box. |
| (s) list variable | a string containing a list of items. |
| (s) delimiter | a character to act as a delimiter between items in the list. |
| (i) sort mode | @sorted for an alphabetic list.
@unsorted to display the list of items as is. |
| (i) select mode | @single to limit selection to one item.
@multiple to allow selection of more than one item. |

Returns:

- | | |
|-----|-----------------------------|
| (s) | the selected item or items. |
|-----|-----------------------------|

This function displays a list box. The list box is filled with a list of items, sorted or unsorted, 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 (we suggest using Tabs). The user selects one of the items by either double clicking 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 your delimiter will be determined by the version of WIL. The 16-bit returns a space-delimited list and the 32-bit version returns a tab-delimited list. **WinItemize**, automatically creates a tab-delimited list of window titles since titles can have embedded blanks.

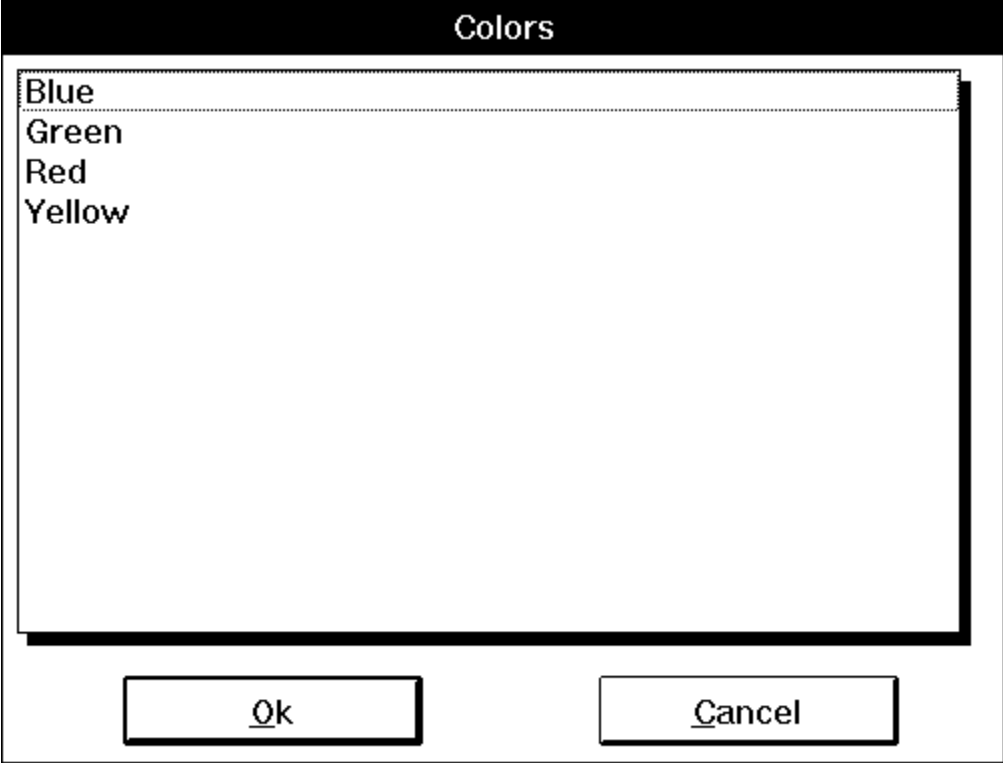
The line(s) highlighted by the user (if any) will be returned to the program. If multiple lines are selected, they will be separated by the specified delimiter. If the user does not make a selection, a null string ("") is returned.

Note: In the 32-bit version of WIL, the "default" file delimiter used to delimit lists of files and directories, has been changed to a TAB. In the 16-bit version of WIL, the "default" delimiter has not changed, and remains a space. We have added the ability to change the file delimiter to a character of your own choosing, using **IntControl (29,p1,0,0,0)**.

Example:

```
list = StrCat("Red",@tab,"Blue",@tab,"Yellow",@tab,"Green")
A = AskItemList("Colors", list, @tab, @sorted, @single)
Message("The item selected is", A)
```

produces:



See Also:

[AskFileText](#), [AskFileName](#)

AskLine

Prompts the user for one line of input.

Syntax:

AskLine (title, prompt, default)

Parameters:

- (s) title title of the dialog box.
- (s) prompt question to be put to the user.
- (s) default default answer.

Returns:

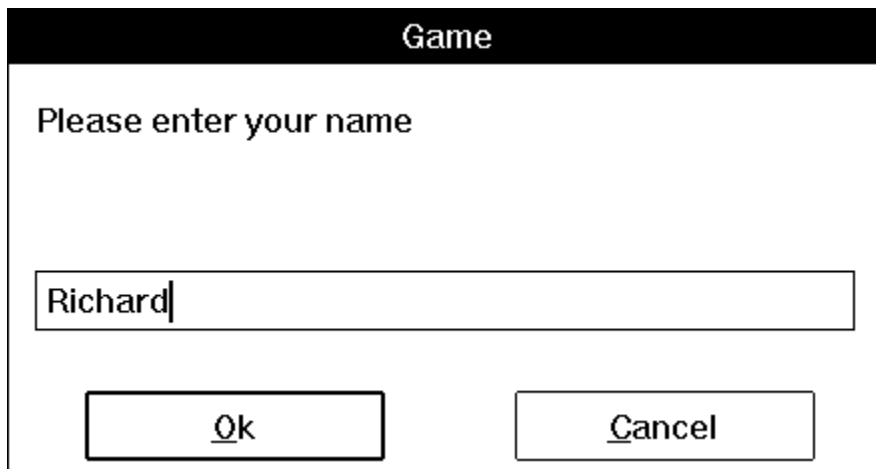
- (s) user response.

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

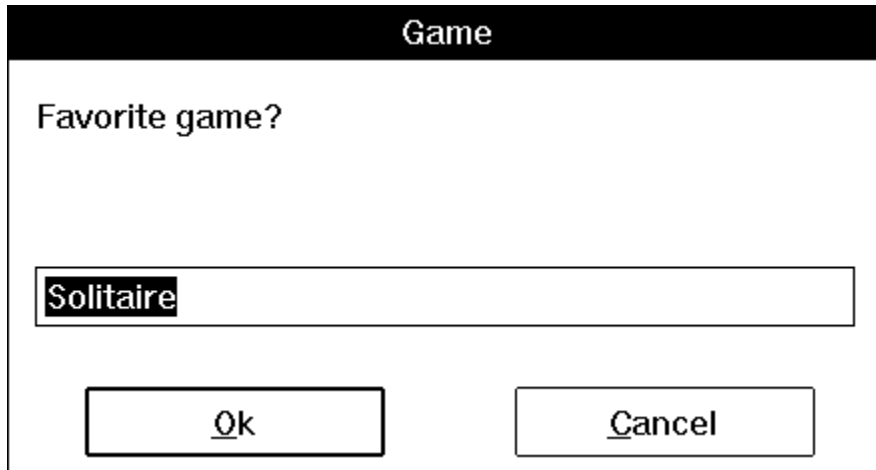
Example:

```
name = AskLine("Game", "Please enter your name", "")
game = AskLine("Game", "Favorite game?", "Solitaire")
message(StrCat(name,"'s favorite game is "), game)
```

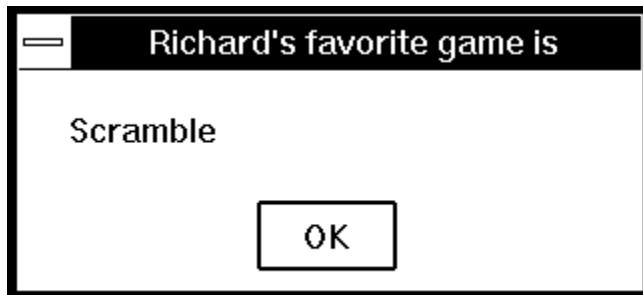
produces:



The image shows a screenshot of a dialog box titled "Game". The dialog box has a black title bar with the text "Game" in white. Below the title bar, the text "Please enter your name" is displayed. Underneath this text is a text input field containing the word "Richard" followed by a vertical cursor. At the bottom of the dialog box, there are two buttons: "Ok" and "Cancel". The "Ok" button has a small underline under the 'k' and the "Cancel" button has a small underline under the 'c'.



And then, if Richard types "Scramble" and clicks on the OK button:



See Also:

[AskPassword](#), [AskYesNo](#), [Dialog](#), [Display](#), [AskItem](#), [Message](#), [Pause](#), [AskFileText](#)

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 a password. An asterisk (*) is echoed for each character that the user types; the actual characters entered are not displayed. The entire user response will be returned if the user presses the `OK` button or the `Enter` key. If the user presses the `Cancel` button or the `Esc` key, the processing of the WIL program is canceled.

Example:

```
pw = AskPassword("Security check", "Please enter your password")
If StriCmp(pw, "Erin") == 0
    Run(Environment("COMSPEC"), "")
    Exit
else
    Pause("Security breach", "Invalid password entered")
endif
```

See Also:

[AskLine](#), [AskYesNo](#), [Dialog](#)

AskYesNo

Prompts the user for a Yes or No answer.

Syntax:

AskYesNo (title, question)

Parameters

- (s) title title of the question box.
- (s) question question to be put to the user.

Returns:

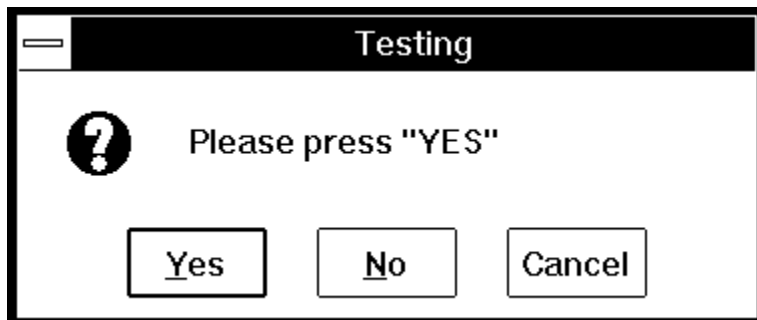
- (i) **@YES** or **@NO**, depending on the button pressed.

This function displays a message box with three pushbuttons - Yes, No, and Cancel.

Example:

```
q = AskYesNo('Testing', 'Please press "YES"')
If q == @YES Then Exit
Display(3, 'ERROR', 'I said press "YES"')
```

produces:



... and then, if the user presses **N**o:



See Also:

[AskLine](#), [AskPassword](#), [Dialog](#), [Display](#), [AskItemList](#), [Message](#), [Pause](#), [AskFileText](#)

Atan

Calculates the arc tangent.

Syntax:

Atan(x)

Parameters:

(f) x floating point number whose arc tangent is desired.

Returns:

(f) The **Atan** function returns the arc tangent result of x .

The **Atan** function calculates the arc tangent of x , which returns a value in the range $-\pi/2$ to $\pi/2$ radians. If x is 0 a domain error occurs.

Example:

```
real=AskLine("ArcTan", "Enter a real number ", "34.6")
answer=Atan(real)
Message("ArcTan of %real% is", answer)
```

See Also:

[Acos](#), [Asin](#), [Cos](#), [Sin](#), [Tan](#)

Average

Returns the average of a list of numbers.

Syntax:

```
Average(list)
```

Parameters:

(f) list comma delimited floating point numbers to be averaged.

Returns:

(f) the average of the numbers.

Use this function to compute the mean average of a series of numbers, delimited by commas. It adds the numbers provided as parameters, and then divides by the number of parameters. This function returns a floating point value.

Example:

```
avg = Average(1.7, 2.6, 3, 4, 5, 6, 7, 8, 9, 10.6, 11, 12)
Message("The average is", avg)
```

See Also:

[Abs](#), [Fabs](#), [Max](#), [Min](#), [Random](#)

Beep

Beeps once.

Syntax:

```
Beep
```

Parameters:

(none)

Returns:

(not applicable)

Use this command to produce a short beep, generally to alert the user to an error situation or to get the user's attention.

Example:

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

See Also:

[PlayMedia](#), [PlayMidi](#), [PlayWaveForm](#), [Sounds](#)

Binary Operations

File manipulation in fast RAM memory.

WIL contains a number of functions designed to allow direct access to areas - buffers - of computer memory. By managing and working with these buffers using the assorted Binary functions provided, you can implement a number of operations that otherwise would be tedious and time consuming. Using the Binary functions, you can perform edits on any sort of file, build new files, build data structures, and do high-speed editing of multiple files. If you understand the structure of a data file, you can perform fast look-ups of data with these functions.

The principal, required Binary functions are **BinaryAlloc** and **BinaryFree**. The **BinaryAlloc** function allows you to allocate a buffer of almost any size. You may have up to ten separate buffers allocated at one time. When operations are complete with a particular buffer, the **BinaryFree** function is used to return it to the system.

There are **BinaryRead** and **BinaryWrite** functions to read files into allocated buffers, and to write the contents of buffers back to files. A **BinaryCopy** function can move sections of one buffer to another one, allowing buffers to be broken up and re-combined in different fashions.

A set of peek and poke functions, **BinaryPeek**, **BinaryPeekStr**, **BinaryPoke**, and **BinaryPokeStr** allow direct editing and modification of the buffers. These functions can initialize a buffer that can be passed to a third party DLL via the **DllCall** function.

A **BinaryIndex** function is available to assist in searching buffers for known data patterns, and a **BinaryStrCnt** function can quickly scan a buffer and return the number of occurrences of particular data patterns. A couple of functions to get and set the End-of-Data point of each buffer (which is automatically tracked), **BinaryEodGet** and **BinaryEodSet**, and the average unreconstructed hacker has all the tools necessary to become a real hazard to the community at large.

BinaryAlloc

Allocates a memory buffer of the desired size.

Syntax:

BinaryAlloc(size)

Parameters:

(i) size size in bytes of the desired memory buffer.

Returns:

(i) returns a handle to a buffer of the desired size.

Use this function to allocate a memory buffer for [Binary operations](#). Up to 10 separate buffers may be allocated concurrently. Nearly any reasonably sized buffer may be allocated. Windows users may allocate over 10 million bytes, assuming sufficient system memory and page file space is available. Users of 32 bit versions of Windows may allocate, theoretically, over 2 trillion bytes, the real, practical bounds are not established and will vary with system configuration.

When operations on a particular buffer are complete, it should be released with the **BinaryFree** function.

Example:

```
; This example edits the Config.sys file
; by adding a new line to the bottom of the file.
;
fs=FileSize("C:\CONFIG.SYS")
; Allocate a buffer the size of your file + 100 bytes.
binbuf = BinaryAlloc(fs+100)
if binbuf == 0
    Message("Error", "BinaryAlloc Failed")
else
    ; Read the file into the buffer.
    BinaryRead(binbuf, "C:\CONFIG.SYS")
    ; Append a line to the end of the file in buffer.
    BinaryPokeStr(binbuf, fs, "DEVICE=C:\FLOOGLE.SYS%  
")
    ; Write modified file back to the file from the buffer.
    BinaryWrite(binbuf, "C:\CONFIG.SYS")
    binbuf=BinaryFree(binbuf)
endif
Message("BinaryAlloc", "Done.")
```

See Also:

[Binary Operations](#), [BinaryCopy](#), [BinaryFree](#), [BinaryRead](#), [DllCall](#)

BinaryAnd

Performs a bitwise AND on portions of two binary buffers.

Syntax:

```
BinaryAnd(target-handle, target-offset, source-handle, source-offset, count)
```

Parameters:

- (i) target-handle: handle of target buffer.
- (i) target-offset: zero-based offset into the target buffer specifying where the data to be processed starts, and where the result of the operation should be placed.
- (i) source-handle: handle of source buffer.
- (i) source-offset: zero-based offset into the source buffer specifying where the data to be processed starts.
- (i) count: the number of bytes to process.

Returns:

- (i) Returns 1.

The specified sections of the buffers are processed on a byte-by-byte basis, and the results are written to the buffer specified by "target-handle". I.e., the byte at "source-offset" is AND'ed with the byte at "target-offset", and the result of the AND operation is stored in the byte at "target-offset", then the bytes at "source-offset + 1" and "target-offset + 1" are AND'ed, and so on.

"target-handle" and "source-handle" may both reference the same buffer, in which case two sections of the buffer can be processed.

Example:

```

buf1 = BinaryAlloc(10)
buf2 = BinaryAlloc(10)

for i = 0 to 9
  BinaryPoke(buf1, i, 5)      ; this stuffs 5's into buf1
  BinaryPoke(buf2, i, 6)      ; this stuffs 6's into buf2
next

rc = BinaryAnd(buf1, 0, buf2, 0, 3) ; This replaces values in buf1

a=BinaryPeek( buf1, 0 )
b=BinaryPeek( buf1, 1 )
c=BinaryPeek( buf1, 2 )
d=BinaryPeek( buf1, 3 )
e=BinaryPeek( buf1, 4 )

binaryWrite(buf1, "zzbin1.txt") ; if you want to see the values
binaryWrite(buf2, "zzbin2.txt") ; you can write it out with BinaryWrite

; AND LOGIC:
; 5 = 0101
; 6 = 0110
; =====
; 4 = 0100 = result

Message("Variables a, b, and c should now = 4", "a = %a%%crlf%b =
%b%%crlf%c = %c%%crlf%")

```

See Also:

[Binary Operations](#), [BinaryAlloc](#), [BinaryCopy](#), [BinaryFree](#), [BinaryRead](#), [DIICall](#),
[BinaryCompare](#)

BinaryClipGet

Reads the contents of the Windows clipboard into a binary buffer.

Syntax:

```
BinaryClipGet(handle, format)
```

Parameters:

- (i) handle: handle of buffer.
- (i) format: format of clipboard data.

Returns:

- (i) returns the number of bytes read from the clipboard.

Note: 0 in the first parameter of BinaryClipGet returns size of buffer needed for BinaryAlloc, but doesn't stuff contents of clipboard into buffer yet

The following is a list of possible clipboard formats. Note that some of them may not be supported, because the clipboard contains a pointer or handle to external data instead of the data itself.

1	F_TEXT	
2	F_BITMAP	(not supported)
3	F_METAFILEPICT	
4	F_SYLK	
5	F_DIF	
6	CF_TIFF	
7	CF_OEMTEXT	
8	CF_DIB	
9	CF_PALETTE	
10	CF_PENDATA	
11	CF_RIFF	
12	CF_WAVE	
13	CF_UNICODETEXT	
14	CF_ENHMETAFILE	
15	CF_HDROP	
16	CF_LOCALE	
128	CF_OWNERDISPLAY	
129	CF_DSPTEXT	
130	CF_DSPBITMAP	
131	CF_DSPMETAFILEPICT	
142	CF_DSPENHMETAFILE	

Example:

```
;This code will snapshot the screen to the clipboard,  
; Add a BMP file header and write out the data as  
; a legal bmp file. Requires WinBatch 98A or newer  
Snapshot(0)  
size=BinaryClipGet(0,8)  
bb=BinaryAlloc(size)  
BinaryClipGet(bb,8)  
  
bb2=BinaryAlloc(size+14)  
BinaryPokeStr(bb2,0,"BM")  
BinaryPoke4(bb2,2,size+14)  
tableloc=BinaryPeek4(bb,0)+14  
BinaryPoke4(bb2,10,tableloc)  
BinaryCopy(bb2,14,bb,0,size)  
BinaryWrite(bb2,"c:\temp\screenshot.bmp")  
BinaryFree(bb)  
BinaryFree(bb2)  
Message("All","Done")
```

See Also:

[Binary Operations](#), [BinaryAlloc](#), [BinaryWrite](#), [BinaryFree](#), [BinaryClipPut](#)

BinaryClipPut

Writes a binary buffer to the Windows clipboard.

Syntax:

```
BinaryClipPut(handle, format)
```

Parameters:

- (i) handle: handle of buffer.
- (i) format: format of clipboard data.

Returns:

- (i) returns 1.

Note that this function destroys the previous contents of the clipboard.

The following is a list of possible clipboard formats. Note that some of them may not be supported, because the clipboard contains a pointer or handle to external data instead of the data itself.

1	CF_TEXT	
2	CF_BITMAP	(not supported)
3	CF_METAFILEPICT	
4	CF_SYLK	
5	CF_DIF	
6	CF_TIFF	
7	CF_OEMTEXT	
8	CF_DIB	
9	CF_PALETTE	
10	CF_PENDATA	
11	CF_RIFF	
12	CF_WAVE	
13	CF_UNICODETEXT	
14	CF_ENHMETAFILE	
15	CF_HDROP	
16	CF_LOCALE	
128	CF_OWNERDISPLAY	
129	CF_DSPTEXT	
130	CF_DSPBITMAP	
131	CF_DSPMETAFILEPICT	
142	CF_DSPENHMETAFILE	

Example:

```
file = "c:\b\1clip.bmp"

format = 8

size = FileSize(file)
buf = BinaryAlloc(size)

BinaryRead(buf, file)

rc = BinaryClipPut(buf, format)

Message("BinaryClipPut returned", rc)
```

See Also:

[Binary Operations](#), [BinaryAlloc](#), [BinaryWrite](#), [BinaryFree](#), [BinaryClipGet](#)

BinaryCompare

Compares portions of two binary buffers.

Syntax:

```
BinaryCompare ( handle1, offset1, handle2, offset2, count)
```

Parameters:

- (i) handle1: handle of first buffer.
- (i) offset1: zero-based offset into the first buffer specifying where the data to be compared starts.
- (i) handle2: handle of second buffer.
- (i) offset2: zero-based offset into the second buffer specifying where the data to be compared starts.
- (i) count: the number of bytes to compare.

Return:

- (i) @TRUE if the sections are identical; @FALSE otherwise.

The specified sections of the buffers are compared on a byte-by-byte basis. This function is case-sensitive.

"handle1" and "handle2" may both reference the same buffer, in which case two sections of the buffer can be compared.

Example:

```
buf1 = BinaryAlloc(10)
buf2 = BinaryAlloc(10)

BinaryPokeStr(buf1, 0, "Hello")
BinaryPokeStr(buf2, 0, "Hello")

rc = BinaryCompare(buf1, 0, buf2, 0, 4)
Message("BinaryCompare returned", rc)
```

See Also:

[Binary Operations](#), [BinaryAlloc](#), [BinaryCopy](#), [BinaryFree](#), [BinaryRead](#), [DllCall](#)

BinaryConvert

Converts a binary buffer.

Syntax:

BinaryConvert(handle, source-type, target-type, code-page, flags)

Parameters:

- (i) handle: handle of buffer.
- (i) source-type: format of existing data in buffer.
- (i) target-type: format of data which the buffer will be converted to.
- (i) code-page: code page for 8-bit <-> Unicode conversions.
- (i) flags: additional options.

Returns:

- (i) Returns the new binary EOD (end of data) for the buffer.

This function can be used to perform the following types of conversions:

8-bit <-> Unicode

Multibyte <-> Unicode

ANSI <-> OEM

-> Uppercase

-> Lowercase

"Source-type" and **"target-type"** can be one of the following:

- 0 8-bit ANSI
- 1 8-bit OEM
- 2 Multibyte (eg, double-byte)
- 3 Unicode

Conversions from 8-bit (types 0 or 1) to or from multibyte (type 2) are not supported. If you need to perform this type of conversion, you can do it in two steps, using Unicode as an intermediate stage (eg, ANSI to Unicode, then Unicode to multibyte).

For conversions to or from Unicode, a code page must be specified for the 8-bit (non-Unicode) character set. "Code-page" can be any valid code page on your system, or one of the following default code pages:

- 0 ANSI
- 1 OEM
- 2 Macintosh

For conversions which don't involve Unicode, "code-page" is ignored.

"Flags" can be one of the following:

- 1 Convert to uppercase

2 Convert to lowercase

Note: that Unicode uses two bytes for each character. Therefore, for conversions to Unicode, the binary buffer must be large enough to hold at least twice as much data as is currently in the buffer. Ie, if you are trying to convert a buffer which contains a 40-character string to Unicode, the buffer must be at least 80 bytes in size, because the resulting Unicode string will be 80-bytes long.

Example:

```
buf = BinaryAlloc(20)

BinaryPokeStr(buf, 0, "Helloß")

rc = ""

string = BinaryPeekStr(buf, 0, BinaryEodGet(buf))
Message(rc, string)

rc = BinaryConvert(buf, 2, 3, 0, 1)

string = BinaryPeekStr(buf, 0, BinaryEodGet(buf))
Message(rc, string)
```

See Also:

[Binary Operations](#), [BinaryAlloc](#), [BinaryCopy](#), [BinaryFree](#), [BinaryRead](#), [DllCall](#)

BinaryCopy

Copies bytes of data from one binary buffer to another.

Syntax:

```
BinaryCopy(handle targ, offset targ, handle src, offset src, count)
```

Parameters:

- (i) handle targ handle of target buffer.
- (i) offset targ zero-based offset into the target buffer specifying where the data to be copied should be placed.
- (i) handle src handle of the source buffer.
- (i) offset src zero-based offset into the source buffer specifying where the data to be copied starts.
- (i) count the number of bytes to copy.

Returns:

- (i) number of bytes actually copied. The byte count may be lower than that specified in the command if the source block does not contain sufficient data.

Use this function to move blocks of data from one binary buffer to another one. **Count** bytes are transferred from the **handle-src** buffer starting at **offset-src** to the **handle-targ** buffer starting at **offset-targ**.

Example:

```
; This example edits the config.sys file
; and adds a new line at the top of the file.
;
fs1 = FileSize("C:\CONFIG.SYS")
binbuf1 = BinaryAlloc(fs1)
BinaryRead(binbuf1, "C:\CONFIG.SYS")
binbuf2=binaryalloc(fs1 + 200)
n = BinaryPokeStr(binbuf2, 0, "Rem Note new line at top")
a2=BinaryCopy(binbuf2, n, binbuf1, 0, fs1)
BinaryWrite(binbuf2, "C:\CONFIG.SYS")
binbuf2 = BinaryFree(binbuf2)
binbuf1 = BinaryFree(binbuf1)
Message("BinaryCopy", "Done.")
```

See Also:

[Binary Operations](#), [BinaryAlloc](#), [BinaryFree](#), [BinaryRead](#), [BinaryWrite](#)

BinaryEodGet

Returns the offset of the free byte just after the last byte of stored data.

Syntax:

BinaryEodGet(handle)

Parameters:

(i) handle handle of buffer.

Returns:

(i) offset of the free byte just after the last byte of stored data.

Use this function to determine the beginning of the free area just past the already initialized data in a buffer. This value is automatically set by any Binary function that modifies the buffer.

Example:

```
; This example adds three lines to the end of the
; config.sys file.
;
fs1 = FileSize( "C:\CONFIG.SYS" )
binbuf1 = BinaryAlloc( fs1 + 100 )
BinaryRead( binbuf1, "C:\CONFIG.SYS" )
a = BinaryEodGet( binbuf1 )
BinaryPokeStr( binbuf1, a, "REM ADDING FIRST NEW LINE TO END%
```

See Also:

[Binary Operations](#), [BinaryAlloc](#), [BinaryEodSet](#), [BinaryIndex](#),

BinaryEodSet

Sets the EOD (end of data) value of a buffer.

Syntax:

```
BinaryEodSet(handle, offset)
```

Parameters:

- (i) handle handle of buffer.
- (i) offset desired offset to set the end-of-data value to.

Returns:

- (i) previous value.

Use this function to update the EOD value. This can be done when data at the end of a buffer is to be discarded, or when the buffer has been modified by an external program - such as via a **DIICall**.

Example:

```
; This function extracts the first line from the  
; config.sys file and writes it to a new file.
```

```
fs1 = FileSize("C:\CONFIG.SYS")  
binbuf1 = binaryalloc( fs1 + 100)  
BinaryRead(binbuf1, "C:\CONFIG.SYS")  
a = BinaryIndex(binbuf1, 0, @CRLF, @FWDSCAN)  
; we just found the end of first line  
a = a + 2                    ; add 2 to skip crlf  
BinaryEodSet(binbuf1, a)  
BinaryWrite(binbuf1, "firstlin.txt")  
binbuf1 = BinaryFree(binbuf1)  
Message("BinaryEODSet", "Done.")
```

See Also:

[Binary Operations](#), [BinaryAlloc](#), [BinaryEodGet](#), [BinaryIndex](#), [DIICall](#)

BinaryFree

Frees a buffer previously allocated with **BinaryAlloc**.

Syntax:

```
BinaryFree(handle)
```

Parameters:

(i) handle handle of buffer to free.

Returns:

(i) always 0.

Use this function to free a binary buffer previously allocated by the **BinaryAlloc** function. After freeing the buffer, no further operations should be performed on the buffer or its handle.

Example:

```
fs=FileSize("C:\CONFIG.SYS")
binbuf = BinaryAlloc(fs+100)
if binbuf == 0
    Message("Error", "BinaryAlloc Failed")
else
    BinaryRead(binbuf, "C:\CONFIG.SYS")
    BinaryPokeStr(binbuf, fs, "DEVICE=C:\FLOOGLE.SYS%@crlf%")
    BinaryWrite(binbuf, "C:\CONFIG.SYS")
    binbuf=BinaryFree(binbuf)
endif
Message("BinaryFree", "Done.")
```

See Also:

[Binary Operations](#), [BinaryAlloc](#)

BinaryHashRec

Returns a pointer to a record in a binary buffer.

Syntax:

BinaryHashRec(handle, rectxize, key offset, key size, key value)

Parameters:

- (i) handle handle of buffer.
- (i) rectxize specifies the fixed length of each record.
- (i) key offset the offset within a record of the key field (where the first byte in the record is 0).
- (i) key size specifies the size of the key field, in bytes.
- (i) key value the value of the key field to be searched for.

Returns:

- (i) the starting position of a record in a binary buffer.

This function uses a hashing algorithm to calculate a hash value for the specified "keyvalue", which provides an offset into the binary buffer. It starts searching at that offset for either (1) a record with a key field whose first byte is a 0, or (2) a record with a key field whose value is the same as "keyvalue". For case (1), it stores "keyvalue" in the key field of the found record, and returns the offset of the beginning of that record. For case (2), the offset of the record is returned but not stored. If an appropriate record cannot be found (i.e., if the buffer is full), it returns an error.

Note: The binary buffer must consist of fixed-length records. Each record must contain a fixed-length key field in a fixed position.

Example:


```
; In this example, we are going to choose a bunch of fruits at random,  
; look the fruit up in a hash table, increment how many times each  
; fruit occurs, sort the table when we are done, and write a report to  
; a file.
```

```
fruits="apple pear banana apricot kiwi orange peach grape grapefruit"  
fruitcount=ItemCount(fruits," ")-1
```

```
namesize=20  
countsize=4  
recsize=namesize+countsize  
nameoffset=0  
countoffset=20
```

```
tableentries=100 ; note hash tables should be 20-40 percent bigger  
than your data  
tablesize=tableentries*recsize
```

```
hash=BinaryAlloc(tablesize)
```

```
For x=1 to 1000  
  afruit=ItemExtract(Random(fruitcount)+1,fruits," ")  
  offset=BinaryHashRec(hash,recsize,nameoffset, namesize, afruit)  
  BinaryIncr4(hash,offset+countoffset)  
next
```

```
BinarySort(hash,recsize,countoffset,countsize,@WORD4|@ASCENDING)
```

```
offset=0  
Report=""  
While offset<tablesize  
if offset==0  
  if BinaryPeek(hash,0)==0  
    offset=BinaryIndex(hash,recsize,"",@FWDSCAN)  
  endif  
else  
  offset=BinaryIndex(hash,offset,"",@FWDSCAN)
```

```
endif  
if offset==0 then break  
afruit=BinaryPeekStr(hash,offset,namesize)  
acount=BinaryPeek4(hash,offset+countoffset)  
offset=offset+recsize  
Report=strcat(report, afruit, " ", acount, @crlf)
```

```
endwhile  
BinaryFree(hash)  
Message("Random Fruit Report", Report)
```

See Also:

[BinarySort](#), [BinaryIncr](#), [BinaryPeek](#), [BinaryPoke](#)

BinaryIncr

Peeks at a value, increments the value by 1 then pokes the value back into the buffer.

Syntax for BinaryIncr Functions:

BinaryIncr(handle, offset)	Increments an 8 bit value.
BinaryIncr2(handle, offset)	Increments a 16 bit value.
BinaryIncr4(handle, offset)	Increments a 32 bit value.
BinaryIncrFlt(handle, offset)	Increments a 64 bit floating point number.

Parameters:

- (i) handle handle of buffer.
- (i) offset offset in the buffer to obtain byte from.

Returns:

- (i) new byte value.

BinaryIncr is equivalent to doing a **BinaryPeek[...]**, incrementing the extracted value by 1, and then doing a **BinaryPoke[...]** to store the new value.

Example:

```
Binbuf=BinaryAlloc(10)
BinaryPoke( binbuf, 5, -14 ) ;Pokes a new value into the buffer.
a=BinaryPeek( binbuf, 5 ) ;Finds the value of a byte.
Message("Hmmm", "Returned value is %a%" )
; Value will be 242 which is (256 - 14). 242 and -14 map
; to the same 8bit number.
```

See Also:

[Binary Operations](#), [BinaryCopy](#), [BinaryPeek](#), [BinaryPeekStr](#), [BinaryPoke](#), [BinaryPokeStr](#)

BinaryIndex

Searches a buffer for a string. (case sensitive)

Syntax:

BinaryIndex (handle, offset, string, direction)

Parameters:

- (i) handle handle of buffer.
- (i) offset offset in the buffer to begin search.
- (s) string the string to search for within the buffer.
- (i) direction the search direction. **@FWDSCAN** searches forwards, while **@BACKSCAN** searches backwards.

Returns:

- (i) offset of string within the buffer, or 0 if not found.

This function searches for a string within a buffer. Starting at the offset position, it goes forwards or backwards depending on the value of the direction parameter. It stops when it finds the string within the buffer and returns the string's beginning offset.

Notes:

The string parameter may be composed of any characters except the null (00) character. A blank string ("") can be specified for the "string" parameter, in which case the function will return the offset of the first non-NULL character found, starting at "offset".

The return value of this function is possibly ambiguous. A zero return value may mean the string was not found, or it may mean the string was found starting at offset 0. If there is a possibility that the string to be searched for could begin at the beginning of the buffer, you must determine some other way of resolving the ambiguity, such as using

BinaryPeekStr.

Example:

```
; Find line number of line in config.sys where HIMEM occurs
fsl = FileSize( "C:\CONFIG.SYS" )
binbuf1 = binaryalloc( fsl )
a1 = BinaryRead( binbuf1, "C:\CONFIG.SYS" )
a = BinaryIndex( binbuf1, 0, "HIMEM", @FWDSCAN ) ; find HIMEM
if a == 0
    Message("Hmmm", "HIMEM not found in CONFIG.SYS file")
else
    c = BinaryStrCnt( binbuf1, 0, a, @CRLF) + 1
    Message("Hmmm", "HIMEM found on line %c%")
endif
```

See Also:

[Binary Operations](#), [BinaryCopy](#), [BinaryIndexEx](#), [BinaryIndexNc](#) [BinaryEodGet](#),
[BinaryEodSet](#), [BinaryStrCnt](#)

BinaryIndexEx

Searches a buffer for a string. This function is very similar to BinaryIndex and BinaryIndexNc, except this function will return -1, if the specified string was not found.

Syntax:

BinaryIndexEx(handle, offset, string, direction, match-case)

Parameters:

- (i) handle handle of buffer.
- (i) offset offset in the buffer to begin search.
- (s) string the string to search for within the buffer.
- (i) direction the search direction. **@FWDSCAN** searches forwards, while **@BACKSCAN** searches backwards.
- (i) match-case This function is almost the same as BinaryIndex (if "match-case" = 1) and BinaryIndexNc (if "match-case" =0).

Returns:

- (i) offset of string within the buffer, if the specified string is not found, this function returns -1 (unlike BinaryIndex and BinaryIndexNC , which return 0).

Note:This function is almost the same as BinaryIndex (if "match-case" = @TRUE) and BinaryIndexNc (if "match-case" = @FALSE). The one difference: if the specified string is not found, this function returns -1 (unlike those other functions, which return 0).

Example:

```
; Find line number of line in config.sys where HIMEM occurs
fs1 = FileSize( "C:\CONFIG.SYS" )
binbuf1 = binaryalloc( fs1 )
a1 = BinaryRead( binbuf1, "C:\CONFIG.SYS" )
a = BinaryIndexEx( binbuf1, 0, "HIMEM", @FWDSCAN ,0);find HIMEM
if a == 0
    Message("Hmmm", "HIMEM not found in CONFIG.SYS file")
else
    c = BinaryStrCnt( binbuf1, 0, a, @CRLF) + 1
    Message("Hmmm", "HIMEM found on line %c%")
endif
```

See Also:

[Binary Operations](#), [BinaryCopy](#), [BinaryIndexNc](#) [BinaryEodGet](#), [BinaryEodSet](#), [BinaryStrCnt](#), [BinaryIndex](#)

BinaryIndexNc

Searches a buffer for a string, ignoring case.

Syntax:

BinaryIndexNc(handle, offset, string, direction)

Parameters:

- (i) handle handle of buffer.
- (i) offset offset in the buffer to begin search.
- (s) string the string to search for within the buffer.
- (i) direction the search direction. **@FWDSCAN** searches forwards, while **@BACKSCAN** searches backwards.

Returns:

- (i) offset of string within the buffer, or 0 if not found.

This function is like BinaryIndex, but performs a case-insensitive search for a string within a buffer. Starting at the offset position, it goes forwards or backwards depending on the value of the direction parameter. It stops when it finds the string within the buffer and returns the string's beginning offset.

Notes:

The string parameter may be composed of any characters except the null (00) character. A blank string ("") can be specified for the "string" parameter, in which case the function will return the offset of the first non-NULL character found, starting at "offset".

The return value of this function is possibly ambiguous. A zero return value may mean the string was not found, or it may mean the string was found starting at offset 0. If there is a possibility that the string to be searched for could begin at the beginning of the buffer, you must determine some other way of resolving the ambiguity, such as using

BinaryPeekStr.

Example:

```
; Find line number of line in config.sys where HIMEM occurs
fs1 = FileSize( "C:\CONFIG.SYS" )
binbuf1 = binaryalloc( fs1 )
a1 = BinaryRead( binbuf1, "C:\CONFIG.SYS" )
a = BinaryIndexNc( binbuf1, 0, "HIMEM", @FWDSCAN ) ; find HIMEM
if a == 0
    Message("Hmmm", "HIMEM not found in CONFIG.SYS file")
else
    c = BinaryStrCnt( binbuf1, 0, a, @CRLF) + 1
    Message("Hmmm", "HIMEM found on line %c%")
endif
```

See Also:

[Binary Operations](#), [BinaryCopy](#), [BinaryIndex](#), [BinaryIndexEx](#), [BinaryEodGet](#), [BinaryEodSet](#), [BinaryStrCnt](#)

BinaryOleType

Specifies how a binary buffer will be used by OLE functions.

Syntax:

BinaryOleType(handle, type, reserved-1, reserved-2, reserved-3)

Parameters:

- (i) handle handle of buffer.
- (i) type specifies what type of parameter the buffer will represent, see below.
- (i) reserved-1 reserved for future use, specify as 0.
- (i) reserved-2 reserved for future use, specify as 0.
- (i) reserved-3 reserved for future use, specify as 0.

Returns:

- (i) @TRUE if successful.

BinaryOleType defines the attributes of the binary buffer specified by "handle", which determines what happens when you use the binary buffer handle as a parameter to an OLE object (ie, one created with the "ObjectOpen" function).

When a binary buffer is created with **BinaryAlloc**, it has a "type" of 0 (undefined). You must use **BinaryOleType** to set the buffer's type before using it as a parameter to an OLE object. This function does not modify any data in the binary buffer itself.

When an OLE object is called, and one of the parameters is a binary buffer handle, the following actions are automatically taken:

1. An internal OLE buffer is created which is (at least) as large as the binary buffer.
2. If the binary buffer is defined as an input (or input/output) parameter (see "type" below), any data in the binary buffer (up to the binary EOD) is converted as necessary to the specified format and copied to the OLE buffer.
3. The OLE buffer is passed to the OLE object.

When the OLE object returns:

4. If the binary buffer is defined as an output (or input/output) parameter (see "type" below), then any data in the OLE buffer is copied back to the binary buffer. If it is possible to determine how much data was returned in the OLE buffer, then the binary EOD is set to the end of the data; otherwise, the binary EOD is set to the end of the binary buffer.
5. The OLE buffer is freed.

TYPE

"type" specifies what type of parameter the buffer will represent, and whether it will be used as an input or output parameter (or both). It consists of one entry from each of the following groups, added together:

data type:

0	undefined	
1	BSTR	(VT_BSTR)
2	BSTR*	(VT_BSTR VT_BYREF)

direction:

100	input parameter
200	output parameter
300	input/output parameter

Example:

```

; Note: This example shows how to use Ole Automation to
; drive Netscape to obtain a web page. However, if your
; intent is to obtain web pages, a more suitable approach
; would be to try our Internet extender which can be
; downloaded from our Web site.
;
BoxOpen("Object Open Example","Initializing")
url = "http://www.windowware.com/index.html"
bufsize = 100000

ErrorMode(@off)
LastError( )
web = ObjectOpen("Netscape.Network.1")
err=LastError( )
ErrorMode(@cancel)
if err!=0
    Message("Netscape not found","Cannot run this sample program")
    exit
endif

BoxText("Opening netscape Object")
TimeDelay(2)
outfile=strcat(DirGet( ),"oletest.html")
bigbuf=BinaryAlloc(bufsize)
BinaryOleType(bigbuf,202,0,0,0)

rc = web.Open(url, 0, 0, 0, 0)

BoxText(strcat("Retrieving url contents of",@crlf,url))

if rc!=0
    While !(web.IsFinished)
        contsize=web.GetContentLength
        if contsize==0
            TimeDelay(1)
        else
            getnowsize=min(bufsize,contsize)
            gotsize=web.Read(bigbuf,getnowsize)
            BinaryEODSet(bigbuf,gotsize)
            break
        endif
    endwhile
    BinaryWrite(bigbuf,outfile)

else
    Message("Uh-oh", "web.Open failed")
endif

web.Close

BoxText("Complete")
Message("OLE Test", strcat("Finished",@crlf,"Data saved
in",@crlf,outfile))

```

See Also:

[Binary Operations](#), [BinaryCopy](#), [BinaryFree](#), [BinaryRead](#), [DllCall](#)

BinaryOr

Performs a bitwise OR on portions of two binary buffers.

Syntax:

BinaryOr(target-handle, target-offset, source-handle, source-offset, count)

Parameters:

- (i) target-handle: handle of target buffer.
- (i) target-offset: zero-based offset into the target buffer specifying where the data to be processed starts, and where the result of the operation should be placed.
- (i) source-handle: handle of source buffer.
- (i) source-offset: zero-based offset into the source buffer specifying where the data to be processed starts.
- (i) count: the number of bytes to process.

Returns:

- (i) Returns 1.

The specified sections of the buffers are processed on a byte-by-byte basis, and the results are written to the buffer specified by "target-handle". I.e., the byte at "source-offset" is OR'ed with the byte at "target-offset", and the result of the OR operation is stored in the byte at "target-offset", then the bytes at "source-offset + 1" and "target-offset + 1" are OR'ed, and so on.

"target-handle" and "source-handle" may both reference the same buffer, in which case two sections of the buffer can be processed.

Example:

```

buf1 = BinaryAlloc(10)
buf2 = BinaryAlloc(10)

for i = 0 to 9
  BinaryPoke(buf1, i, 5)      ; this stuffs 5's into buf1
  BinaryPoke(buf2, i, 6)      ; this stuffs 6's into buf2
next

BinaryOr(buf1, 0, buf2, 0, 3) ; This replaces values in buf1

a=BinaryPeek( buf1, 0 )
b=BinaryPeek( buf1, 1 )
c=BinaryPeek( buf1, 2 )
d=BinaryPeek( buf1, 3 )
e=BinaryPeek( buf1, 4 )

binaryWrite(buf1, "zzbin1.txt") ; if you want to see the values
binaryWrite(buf2, "zzbin2.txt") ; you can write it out with BinaryWrite

; OR LOGIC:
; 5 = 0101
; 6 = 0110
; =====
; 3 = 0111

Message("Variables a, b, and c should now = 3", "a = %a%%crlf%b =
%b%%crlf%c = %c%%crlf%")

```

See Also:

[Binary Operations](#), [BinaryAlloc](#), [BinaryCopy](#), [BinaryFree](#), [BinaryRead](#),
[BinaryCompare](#), [BinaryAnd](#)

BinaryPeek

Returns the value of a byte from a binary buffer.

Syntax for BinaryPeek Functions:

BinaryPeek(handle, offset)	Peeks an 8 bit value.
BinaryPeek2(handle, offset)	Peeks a 16 bit value.
BinaryPeek4(handle, offset)	Peeks a 32 bit value.
BinaryPeekFlt(handle, offset)	Peeks a 64 bit floating point number.

Parameters:

- (i) handle handle of buffer.
- (i) offset offset in the buffer to obtain byte from.

Returns:

- (i) new byte value.

Use these functions to return the value of a number in the binary buffer. The **BinaryPeek**'s extract 1, 2, or 4 bytes (respectively) beginning at "offset". **BinaryPeekFlt** treats the 8 bytes it extracts as a 64 bit floating point number.

Example:

```
binbuf=BinaryAlloc(10)
BinaryPoke( binbuf, 5, -14 ) ;Pokes a new value into the buffer.
a=BinaryPeek( binbuf, 5 ) ;Finds the value of a byte.
Message("Hmmm", "Returned value is %a%" )
; Value will be 242 which is (256 - 14). 242 and -14 map
; to the same 8bit number.
```

See Also:

[Binary Operations](#), [BinaryCopy](#), [BinaryIncr](#), [BinaryPeekStr](#), [BinaryPoke](#),
[BinaryPokeStr](#)

BinaryPeekStr

Extracts a string from a binary buffer.

Syntax:

```
BinaryPeekStr(handle, offset, maxsize)
```

Parameters:

- (i) handle handle of buffer.
- (i) offset offset in the buffer the string starts at.
- (i) maxsize maximum number of bytes in string.

Returns:

- (s) string starting offset location in binary buffer.
String consists of all non-zero bytes up to the first zero byte or maxsize number of bytes.

This function is used to extract string data from a binary buffer. The desired starting offset and a maxsize are passed to the function. The function returns a string of bytes, starting at the specified offset, and continuing until either a zero byte is found (which terminates the string) or the maxsize number of bytes have been copied into the return string.

Example:

```
; This example searches the Config.sys for the first
; occurrence of the string HIMEM. It then extracts
; the line containing the string and prints it out.
;
fs = FileSize( "C:\CONFIG.SYS" )
binbuf = BinaryAlloc( fs )
BinaryRead( binbuf, "C:\CONFIG.SYS" )
;
; Search for first occurrence of HIMEM.
himem = BinaryIndex( binbuf, 0, "HIMEM", @FWDSCAN)
; Single out beginning of line which contains HIMEM string,
; skipping over the @crlf.
linebegin = BinaryIndex( binbuf, himem, @CRLF, @BACKSCAN) + 2
;
; Search for the end of the line which contains the HIMEM string.
lineend = BinaryIndex( binbuf, himem, @CRLF, @FWDSCAN)
linelen = lineend-linebegin+1
;
; Extract the line with HIMEM string.

linedata=BinaryPeekStr(binbuf, linebegin, linelen)
binbuf=BinaryFree(binbuf)
Message("Himem.sys line in config.sys reads", linedata)
```

See Also:

[Binary Operations](#), [BinaryCopy](#), [BinaryPeek](#), [BinaryPoke](#), [BinaryPokeStr](#)

BinaryPoke

Pokes a new value into a binary buffer at offset returning the previous value.

Syntax for BinaryPoke Functions:

BinaryPoke(handle, offset, value)	Pokes an 8 bit value.
BinaryPoke2(handle, offset, value)	Pokes a 16 bit value.
BinaryPoke4(handle, offset, value)	Pokes a 32 bit value.
BinaryPokeFlt(handle, offset, value)	Pokes a 64 bit floating point number.

Parameters:

- (i) handle handle of buffer.
- (i) offset offset in the buffer to obtain byte from.
- (i) value value to store.

Returns:

- (i) previous value.

This function pokes a value into the binary buffer at the offset specified.

Example:

```
binbuf=BinaryAlloc(10)
BinaryPoke( binbuf, 5, -14 ) ;Pokes a new value into the buffer.
a=BinaryPeek( binbuf, 5 ) ;Finds the value of a byte.
Message("Hmmm", "Returned value is %a%" )
; Value will be 242 which is (256 - 14). 242 and -14 map
; to the same 8bit number.
```

See Also:

[Binary Operations](#), [BinaryCopy](#), [BinaryIncr](#), [BinaryPeek](#), [BinaryPeekStr](#),
[BinaryPokeStr](#)

BinaryPokeStr

Writes a string into a binary buffer.

Syntax:

```
BinaryPokeStr(handle, offset, string)
```

Parameters:

- (i) handle handle of buffer.
- (i) offset offset in the buffer to store string.
- (s) string string to store into buffer.

Returns:

- (i) number of bytes stored.

This function is used to write string data into a binary buffer. There must be sufficient space in the buffer between the offset and the allocated end of the buffer to accommodate the string.

Note: The string parameter may be composed of any characters except the null (00) character. If a null character is found, it will be assumed that the string ends at that point. If you need to store a null character into a binary buffer, use the **BinaryPoke** function.

Example:

```
; This example writes a new device= line to SYSTEM.INI
; It is *very* fast
NewDevice = "DEVICE=COOLAPP.386"
;
; Change to the Windows Directory
DirChange(DirWindows(0))
;
; Obtain filesize and allocate binary buffers
fs1=FileSize("SYSTEM.INI")
srcbuf = BinaryAlloc(fs1)
editbuf = BinaryAlloc(fs1+100)
;
; Read existing system.ini into memory
BinaryRead( srcbuf, "SYSTEM.INI")
;
; See if this change was already installed. If so, quit
a = BinaryIndexNc( srcbuf, 0, "COOLAPP.386", @FWDSCAN)
if a != 0 then goto AlreadyDone
;
; Find 386Enh section.
a = BinaryIndexNc( srcbuf, 0, "[386Enh]", @FWDSCAN)
;
;
; Find beginning of next line ( add 2 to skip over our crlf )
cuthere = BinaryIndexNc( srcbuf, a, @CRLF, @FWDSCAN) + 2
;
; Copy data from beginning of file to just after [386Enh}
; to the edit buffer
BinaryCopy( editbuf, 0, srcbuf, 0, cuthere)
;
; Add the device= line to the end of the edit buffer, and add a CRLF
BinaryPokeStr(editbuf,BinaryEodGet(editbuf), Strcat(NewDevice,@CRLF))
```

```
;
; Copy remaining part of source buffer to the edit buffer
a = BinaryEodGet(editbuf)
b = BinaryEodGet(srcbuf)
BinaryCopy( editbuf, a, srcbuf, cuthere, b-cuthere)
;
; Save file out to disk. Use system.tst until it is
; completely debugged
BinaryWrite( editbuf, "SYSTEM.TST")
;
; Close binary buffers
:AlreadyDone
BinaryFree(editbuf)
BinaryFree(srcbuf)
Message("BinaryPokeStr", "Done.")
```

See Also:

[Binary Operations](#), [BinaryCopy](#), [BinaryPeek](#), [BinaryPeekStr](#), [BinaryPoke](#)

BinaryRead

Reads a file into a binary buffer.

Syntax:

```
BinaryRead(handle, filename)
```

Parameters:

- (i) handle handle of buffer.
- (s) filename file to read into buffer.

Returns:

- (i) the number of bytes read.

This function reads the entire contents of a file into a buffer then returns the number of bytes read. The buffer must be large enough to hold the entire file. The file is placed into the buffer starting at offset 0.

Example:

```
; This example edits the Config.sys file by adding a line
; to the bottom of the file.
fs=FileSize("C:\CONFIG.SYS")
binbuf = BinaryAlloc(fs+100)
if binbuf == 0
    Message("Error", "BinaryAlloc Failed")
else
    BinaryRead(binbuf, "C:\CONFIG.SYS")
    BinaryPokeStr(binbuf, fs, "DEVICE=C:\FLOOGLE.SYS%@crlf%")
    BinaryWrite(binbuf, "C:\CONFIG.SYS")
    binbuf=BinaryFree(binbuf)
endif
Message("BinaryRead", "Done.")
```

See Also:

[Binary Operations](#), [BinaryAlloc](#), [BinaryReadEx](#) , [BinaryFree](#), [BinaryWrite](#)

BinaryReadEx

Reads a portion of a file into a binary buffer.

Syntax:

```
BinaryReadEx(handle, binary-offset, filename, file-offset, count)
```

Parameters:

- (i) handle: handle of buffer.
- (i) binary-offset: zero-based offset into the buffer specifying where the data should be stored.
- (i) filename: file to read into buffer.
- (i) file-offset: zero-based offset into the file specifying where the data to be read starts.
- (i) count: the number of bytes to read.

Returns:

- (i) Returns the number of bytes read.

This function reads "count" bytes from "filename", beginning at "file-offset". It then writes the data to the buffer specified by "handle", beginning at "binary-offset". Any existing data in the buffer within the range "binary-offset + count" is overwritten with the new data; any existing data outside that range is left untouched.

Example:

```
file = "hello.txt"

buf = BinaryAlloc(100)

BinaryPokeStr(buf, 0, "12345")

rc = BinaryReadEx(buf, 2, file, 4, 1)
Message("BinaryReadEx returned", rc)

eod = BinaryEodGet(buf)
Message("BinaryEod is", eod)

string = BinaryPeekStr(buf, 0, eod)
Message("BinaryPeekStr returned", string)
```

See Also:

[Binary Operations](#), [BinaryRead](#), [BinaryAlloc](#), [BinaryFree](#), [BinaryWrite](#)

BinarySort

Sorts records in a binary buffer.

Syntax:

BinarySort(handle, reccount, key offset, key size, flags)

Parameters:

- | | |
|----------------|--|
| (i) handle | handle of buffer. |
| (i) reccount | specifies the size of each record. |
| (i) key offset | the offset within a record of the key field (where the first byte in the record is 0). |
| (i) key size | specifies the size of the key field, in bytes. |
| (i) flags | see below. |

Returns:

- (i) @TRUE if successful.

BinarySort sorts records in a binary buffer in place, updating the existing buffer. String sorts are case-insensitive.

The flags consist of a maximum of one value from each of the following groups, combined with the binary OR ("|") operator. If either (or both) is not specified, the default value will be used:

Sort sequence:

- @ASCENDING (default)
- @DESCENDING

Key type:

- @STRING (default)
- @WORD1 (i.e., BYTE)
- @WORD2 (i.e., WORD)
- @WORD4 (i.e., DWORD)
- @FLOAT8

Note: The binary buffer must consist of fixed-length records. Each record must contain a fixed-length key field in a fixed position.

Example:

```

fh=FileOpen("test.in","WRITE")
; Note: Each record is 40 chars (recsize+crlf
;          1111111111222222222233333333334
;offsets    ;012345678901234567890123456789012345678901234567890
FileWrite(fh,"0001 Flintstone Fred (111)222-1334")
FileWrite(fh,"0002 Duck Donald (271)333-2334")
FileWrite(fh,"0003 Duck Daffy (222)444-3334")
FileWrite(fh,"0004 Hedgehog Sonic (215)555-4334")
FileWrite(fh,"0005 Mario Super (212)666-5334")
FileWrite(fh,"0006 Kent Clark (234)777-6334")

FileWrite(fh,"0007 Lane Lois (987)888-7334")
FileWrite(fh,"0008 Mouse Mickey (765)999-8334")
FileWrite(fh,"0009 Coyote Wiley (853)111-9334")
; length    1234567890123456789012345678901234567890
; Note that offset position and ordinal position are
; not the same.
FileClose(fh)

RecSize=40
NameOffset=5
NameSize=20
dbsize=FileSize("test.in")
db=BinaryAlloc(dbsize)
BinaryRead(db,"test.in")
BinarySort(db,RecSize,NameOffset,NameSize,@STRING|@ASCENDING)

BinaryWrite(db,"test.out")
BinaryFree(db)
Message("Sort Complete","Launching Notepad")
Run("notepad.exe","test.in")
Run("Notepad.exe","test.out")

```

See Also:

[BinaryHashRec](#)

BinaryStrCnt

Counts the occurrences of a string in some or all of a binary buffer.

Syntax:

BinaryStrCnt(handle, start-offset, end-offset, string)

Parameters:

- (i) handle handle of buffer.
- (i) start-offset offset for start of search.
- (i) end-offset offset for end of search.
- (s string string to search for.

Returns:

- (i) number of occurrences of string found.

This function will search all or a portion of a binary buffer for a string and will return a count of the occurrences of the string found. The buffer will be searched from the start-offset to the end-offset.

Note: The string parameter may be composed of any characters except the null (00) character. This function cannot process a null character.

Example:

```
; Find number of Device, DEVICE= and device= lines in config.sys
fs1 = FileSize( "C:\CONFIG.SYS" )
binbuf1 = binaryalloc( fs1 )
BinaryRead( binbuf1, "C:\CONFIG.SYS" )
a = BinaryStrCnt( binbuf1, 0, fs1 - 1, "Device=")
b= BinaryStrCnt( binbuf1, 0, fs1 - 1, "DEVICE=")
c= BinaryStrCnt( binbuf1, 0, fs1 - 1, "device=")
BinaryFree( binbuf1 )
d = a + b + c
Message( "Hmmm", "Total Device= lines found in Config.Sys is %d% " )
```

See Also:

[Binary Operations](#), [BinaryEodGet](#), [BinaryEodSet](#), [BinaryIndex](#), [BinaryPeek](#),
[BinaryPeekStr](#), [BinaryPoke](#), [BinaryPokeStr](#)

BinaryTagExtr

Returns the text between the last-returned pair of binary tags.

Syntax:

BinaryTagExtr(tag-struct, flags)

Parameters:

(s) tag-struct Structure returned from BinaryTagInit.
(i) flags see below.

Returns:

(s) Returns the text, or "" on failure.

Note: This function was designed to work in correlation with the HTML extender.

Flags	Meaning
0	default.
1	Strip out tabs, carriage returns, and line feeds.

Example:

```
;BinaryTag Example 3 - Replace <b>xxx</b> in HTML files with
;<strong>xxx</strong> per html style guidelines
;Set up test case

;Setup input and output file names
filedata=AskFilename("HTML file to convert","", "HTML files|
*.html;*.htm","*.html",1)
filerslt=FileMapName(filedata, "*.new")

; Allocate a buffer much larger than required.
fsize=Filesize(filedata)
bb=BinaryAlloc(fsize+10000)

;Find <b> and </b> for open and close tags around strings.
;Note BinaryTagFind is NOT case sensitive
structure=BinaryTagInit(bb, "<b>", "</b>")

;Not necessary in this example, but allows reuse of a
;BinaryBuffer if this code is placed in a loop editing
;multiple files.
BinaryEODSet(bb,0)

;Read data into the BinaryBuffer
BinaryRead(bb, filedata)

while 1
  structure=BinaryTagFind(structure)
  if structure=="" then break ; All done
  strongdata=BinaryTagExtr(structure,1)
  Value=strcat("<STRONG>",strongdata, "</STRONG>")
  BinaryTagRepl(structure, Value)
endwhile
```

```
BinaryWrite(bb,filerslt)
BinaryFree(bb)
Message("Result in",filerslt)
exit
```

See Also:

[BinaryTagInit](#), [BinaryTagFind](#), [BinaryTagRepl](#), **HTML Extender**

BinaryTagFind

Finds the next binary tag.

Syntax:

```
BinaryTagFind(tag-struct)
```

Parameters:

(s) tag-struct Structure returned from BinaryTagInit.

Returns:

(s) Returns a binary tag structure string, or "" on failure.

Note: This function was designed to work in correlation with the HTML extender.

Example:


```

;BinaryTag Example 2 - Substitution in GoSubs
;Set up test case

;Setup input and output file names
filedata=strcat(DirGet(),"filedata.txt")
filerslt=strcat(DirGet(),"filerslt.txt")

;Set up a sample file for this example.
;Presumably you could have a library of these
;to choose from, making form letters easy to
;automate.
fhandle=FileOpen(filedata,"WRITE")
FileWrite(fhandle,"{{date}}")
FileWrite(fhandle,"")
FileWrite(fhandle,"Dear {{name}}")
FileWrite(fhandle,"Thank you for your recent purchase of")
FileWrite(fhandle,"our new {{product}}.")
FileWrite(fhandle,"")
FileWrite(fhandle,"Please feel free to call if you have ")
FileWrite(fhandle,"any questions.")
FileWrite(fhandle,"")
FileWrite(fhandle,"{{salesperson}}")
FileWrite(fhandle,"{{salespersonphone}}")
FileClose(fhandle)

; Allocate a buffer much larger than required.
bb=BinaryAlloc(10000)

;Using {{ and }} for open and close tags around keywords
structure=BinaryTagInit(bb,"{","}")

;Not necessary in this example, but allows reuse of a
;BinaryBuffer if this code is placed in a loop editing
;multiple files.
BinaryEODSet(bb,0)

;Read data into the BinaryBuffer
BinaryRead(bb,filedata)

while 1
    structure=BinaryTagFind(structure)
    if structure="" then break ; All done
    keyword=BinaryTagExtr(structure, 1)

    Value="???" ; keyword not found error value
    gosub %keyword%

    BinaryTagRepl(structure,Value)
endwhile

BinaryWrite(bb,filerslt)
BinaryFree(bb)
Message("Result in",filerslt)
exit

:date
    Value=TimeDate()
    return

:name
    Value="Ms. Jamie Dough"

```

```
return

:product
Value="Analog Osscilosophilator"
return

:salesperson
Value="Fred Ficklemeier"
return

:salespersonphone
Value="888.555.1234"
return
```

See Also:

[BinaryTagInit](#), [BinaryTagExtr](#), [BinaryTagRepl](#), [**HTML Extender**](#)

BinaryTagInit

Initializes a binary tag operation.

Syntax:

```
BinaryTagInit ( handle, start-tag, end-tag)
```

Parameters:

- | | |
|---------------|---|
| (i) handle | Handle of buffer. (handle returned from BinaryAlloc). |
| (s) start-tag | Specifies what beginning string (tag) to search for (ie. "{") |
| (s) end-tag | Specifies what ending string (tag) to search for (ie. "}") |

Returns:

- | | |
|-----|--|
| (s) | Returns a binary tag structure string, or "" on failure. |
|-----|--|

Note: This function was designed to work in correlation with the HTML extender.

It searches the buffer for the start and end tags and returns a binary tag structure string that is used with the other binary tag functions.

Example:

```

;BinaryTag Example 1 - Basic Code
;Set up test case

;Setup input and output file names
filedata=strcat(DirGet(),"filedata.txt")
filerslt=strcat(DirGet(),"filerslt.txt")

;Set up a sample file for this example.
;Presumably you could have a library of these
;to choose from, making form letters easy to
;automate.
fhandle=FileOpen(filedata,"WRITE")
FileWrite(fhandle,"{{date}}")
FileWrite(fhandle,"")
FileWrite(fhandle,"Dear {{name}}")
FileWrite(fhandle,"Thank you for your recent purchase of")
FileWrite(fhandle,"our new {{product}}.")
FileWrite(fhandle,"")
FileWrite(fhandle,"Please feel free to call if you have ")
FileWrite(fhandle,"any questions.")
FileWrite(fhandle,"")
FileWrite(fhandle,"{{salesperson}}")
FileWrite(fhandle,"{{salespersonphone}}")
FileClose(fhandle)

; Allocate a buffer much larger than required.
bb=BinaryAlloc(10000)

;Using {{ and }} for open and close tags around keywords
structure=BinaryTagInit(bb,"{","}")

;Not necessary in this example, but allows reuse of a
;BinaryBuffer if this code is placed in a loop editing
;multiple files.
BinaryEODSet(bb,0)

;Read data into the BinaryBuffer
BinaryRead(bb,filedata)

while 1
    structure=BinaryTagFind(structure)
    if structure="" then break ; All done
    keyword=BinaryTagExtr(structure,1)

    Value="???" ; keyword not found error value
    if keyword=="date" then Value=TimeDate()
    if keyword=="name" then Value="Ms. Jamie Dough"
    if keyword=="product" then Value="Analog Osscilosophilator"
    if keyword=="salesperson" then Value="Fred Ficklemeyer"
    if keyword=="salespersonphone" then Value="888.555.1234"

    BinaryTagRepl(structure,Value)
endwhile

BinaryWrite(bb,filerslt)
BinaryFree(bb)
Message("Result in",filerslt)

```

See Also:

[BinaryTagFind](#), [BinaryTagExtr](#), [BinaryTagRepl](#), [HTML Extender](#)

BinaryTagRepl

Replaces a binary tag with text.

Syntax:

BinaryTagRepl (tag-struct, new-string)

Parameters:

(s) tag-struct	Structure returned from BinaryTagInit.
(s) new-string	text string to replace binary-tag with.

Returns:

(s)	Returns a binary tag structure string. If BinaryTagRepl fails, the replace operation is ignored and the strings are not replaced.
-----	--

Note: This function was designed to work in correlation with the HTML extender.

Example:

```

;BinaryTag Example 4 - Replace - passing parameters and other tricks
;Set up test case

;Setup input and output file names
filedata=strcat(DirGet(),"filedata.txt")
filerslt=strcat(DirGet(),"filerslt.txt")

;Set up a sample file for this example.
;Presumably you could have a library of these
;to choose from, making the generation of
;customized HTML pages easy to automate.
fhandle=FileOpen(filedata,"WRITE")
FileWrite(fhandle,"<HTML>")
FileWrite(fhandle,"<HEAD><TITLE>{{Title}}</TITLE></HEAD>")
FileWrite(fhandle,"<BODY>")
FileWrite(fhandle,"<p>Random number in range 0 thru 10 = {{RANDOM 0
10}}")
FileWrite(fhandle,"<p>Random number in range 92 thru 114 = {{RANDOM 92
114}}")
FileWrite(fhandle,"<p>Random number in range -20 thru 4 = {{RANDOM -20
4}}")
FileWrite(fhandle,"<p>Value of <b>filedata</b> variable=<br>{{VALUE
filedata}}")
FileWrite(fhandle,"<p>Value of <b>filerslt</b> variable=<br>{{VALUE
filerslt}}")
FileWrite(fhandle,"<p>Current time/date in YmdHms format = {{DATE
YMDHMS}}")
FileWrite(fhandle,"<p>Current time/date in display format = {{DATE
DISPLAY}}")
FileWrite(fhandle,"</BODY>")
FileWrite(fhandle,"</HTML>")
FileClose(fhandle)

; Allocate a buffer much larger than required.
bb=BinaryAlloc(10000)

;Using {{ and }} for open and close tags around keywords
structure=BinaryTagInit(bb,"{","}")

;Not necessary in this example, but allows reuse of a
;BinaryBuffer if this code is placed in a loop editing
;multiple files.
BinaryEODSet(bb,0)

;Read data into the BinaryBuffer
BinaryRead(bb,filedata)

while 1
    structure=BinaryTagFind(structure)
    if structure="" then break ; All done
    ;Get tag with any extra spaces removed
    extract=strtrim(BinaryTagExtr(structure, 1))
    ParseData(extract)

    HTMLValue="???" ; keyword not found error value
    gosub %param1%

    BinaryTagRepl(structure,HTMLValue)
endwhile

BinaryWrite(bb,filerslt)
BinaryFree(bb)

```

```
Message("Result in",filerslt)
exit

:Value
; Use WinBatch substitution to capture variable contents
HTMLValue = %param2%
return

:TITLE
HTMLValue="BinaryTagRepl Example"
return

:DATE
if param2=="YMDHMS"
HTMLValue=TimeYmdHms()
else
if param2=="DISPLAY"
HTMLValue=TimeDate()
else
HTMLValue="???Unknown DATE format???"
endif
endif
return

:RANDOM
randomlow=param2
randomhigh=param3
randomrange=randomhigh-randomlow
HtmlValue=Random(randomrange)+randomlow
return
```

See Also:

[BinaryTagInit](#), [BinaryTagFind](#), [BinaryTagExtr](#), [HTML Extender](#)

BinaryWrite

Writes a binary buffer to a file.

Syntax:

```
BinaryWrite(handle, filename)
```

Parameters:

- (i) handle handle of buffer.
- (s) filename filename to write buffer to.

Returns:

- (i) number of bytes written.

This function writes the contents of a binary buffer out to a file and returns the number of bytes written. Data written to the file starts at offset 0 in the buffer and extends to the end of data - not necessarily the end of the buffer. The end of data may be inspected or modified with the [BinaryEodGet](#) and [BinaryEodSet](#) functions.

Example:

```
; This example edits the Config.sys file by adding a line
; to the bottom of the file.
fs=FileSize("C:\CONFIG.SYS")
binbuf = BinaryAlloc(fs+100)
if binbuf == 0
    Message("Error", "BinaryAlloc Failed")
else
    BinaryRead(binbuf, "C:\CONFIG.SYS")
    BinaryPokeStr(binbuf, fs, "DEVICE=C:\FLOOGLE.SYS%@crlf%")
    BinaryWrite(binbuf, "C:\CONFIG.SYS")
    binbuf=BinaryFree(binbuf)
endif
Message("BinaryWrite", "Done.")
```

See Also:

[Binary Operations](#), [BinaryAlloc](#), [BinaryWriteEx](#) , [BinaryFree](#), [BinaryRead](#)

BinaryWriteEx

Writes a portion of a binary buffer to a file.

Syntax:

BinaryWriteEx(handle, binary-offset, filename, file-offset, count)

Parameters:

- (i) handle: handle of buffer.
- (i) binary-offset: zero-based offset into the buffer specifying where the data to be read starts.
- (i) filename: file to write to.
- (i) file-offset: zero-based offset into the file specifying where the data should be stored.
- (i) count: the number of bytes to write.

Returns:

- (i) Returns the number of bytes written.

This function reads "count" bytes from the buffer specified by "handle", beginning at "binary-offset". It then writes the data to "filename", beginning at "file-offset". Any existing data in the file within the range "file-offset + count" is overwritten with the new data; any existing data outside that range is left untouched.

If "filename" does not exist, it will be created. If the existing file is smaller than "file-offset", NULL bytes will be added to the end of the file to increase its size to "file-offset", and then the specified data will be written beginning at "file-offset".

Example:

```
file = "lbinout.txt"

buf = BinaryAlloc(10)

BinaryPokeStr(buf, 0, "01234")

rc = BinaryWriteEx(buf, 1, file, 50, 3)
Message("BinaryWriteEx returned", rc)
```

See Also:

[Binary Operations](#), [BinaryWrite](#), [BinaryAlloc](#), [BinaryFree](#), [BinaryRead](#)

BinaryXor

Performs a bitwise XOR (exclusive OR) on portions of two binary buffers.

Syntax:

```
BinaryXor(target-handle, target-offset, source-handle, source-offset, count)
```

Parameters:

- (i) target-handle: handle of target buffer.
- (i) target-offset: zero-based offset into the target buffer specifying where the data to be processed starts, and where the result of the operation should be placed.
- (i) source-handle: handle of source buffer.
- (i) source-offset: zero-based offset into the source buffer specifying where the data to be processed starts.
- (i) count: the number of bytes to process.

Returns:

- (i) Returns 1.

The specified sections of the buffers are processed on a byte-by-byte basis, and the results are written to the buffer specified by "target-handle". I.e., the byte at "source-offset" is XOR'ed with the byte at "target-offset", and the result of the XOR operation is stored in the byte at "target-offset", then the bytes at "source-offset + 1" and "target-offset + 1" are XOR'ed, and so on.

"target-handle" and "source-handle" may both reference the same buffer, in which case two sections of the buffer can be processed.

Example:

```
buf1 = BinaryAlloc(10)
buf2 = BinaryAlloc(10)

for i = 0 to 9
    BinaryPoke(buf1, i, 0)
    BinaryPoke(buf2, i, 1)
next

BinaryPoke(buf1, 0, 1)

rc = BinaryXor(buf1, 0, buf2, 0, 3)

binaryWrite(buf1, "zzbin1")
binaryWrite(buf2, "zzbin2")

Message("BinaryAnd returned", rc)
```

See Also:

[Binary Operations](#), [BinaryAlloc](#), [BinaryOr](#), [BinaryCopy](#), [BinaryFree](#), [BinaryRead](#), [DllCall](#), [BinaryCompare](#), [BinaryAnd](#)

Break

The Break statement is used to exit a **While**, **Switch**, **Select**, or **For/Next** structure.

Syntax:

```
break
```

Parameters:

```
none
```

Use the **Break** statement to exit a **While**, **Switch**, **Select**, or **For/Next** structure. It transfers control to the statement immediately following the nearest enclosing **EndWhile**, **EndSwitch**, **EndSelect**, or **Next**. It is used to terminate loops and to exit **Switch** statements - usually just before the next **case** statement.

Example:

```
a=15
b=21
while (a<100)
    a=a+1
    if a==4 then break
    b=b+1
endwhile
Message("Break", "It Broke.")
```

See Also:

[Continue](#), [For](#), [While](#), [Switch](#), [Select](#)

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 many of the 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").

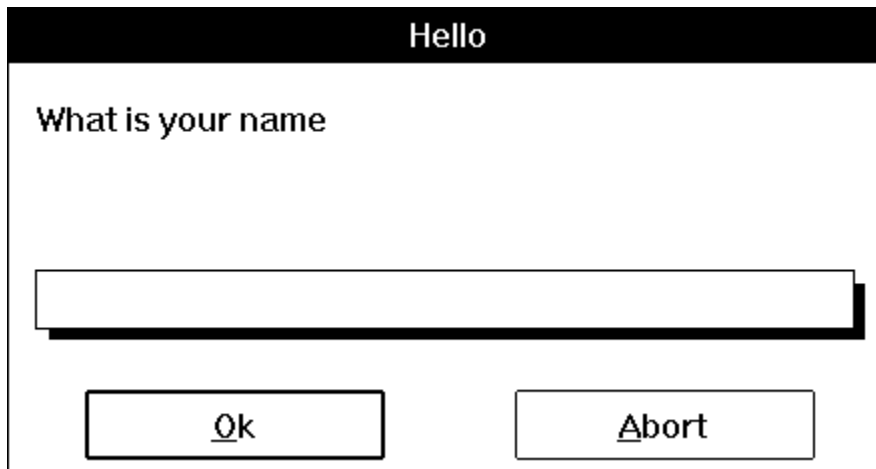
You can place an ampersand before the character which you want to be the underlined character in the dialog.

Note: **Buttonnames** does not change the name in all functions which display buttons. Supported functions are: **AskLine**, **AskFileText**, **AskItemList**, **AskPassword**.

Example:

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

would produce:



See Also:

n/a

Call

Calls a WIL batch file as a subroutine.

Syntax:

Call (filename, parameters)

Parameters:

- (s) filename the WIL batch file you are calling (including extension).
- (s) parameters the parameters to pass to the file, if any, in the form "p1 p2 p3 ... pn".

Returns:

- (i) always 0.

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

If a string of parameters is passed to the secondary WIL batch 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:

```
; File MAIN.WBT
;
; This example asks for user input, their name and age,
; and then calls another WinBatch job to verify if their
; age is between 0 & 150.
;
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")
Exit

;FILE CHEK-AGE.WBT
;
; This subroutine checks if the age inputted is between 0 & 150.
; If this is true, a global parameter is set to a value of 1.
userage = param1
really = AskYesNo("", "%name%, are you really %userage%?")
If really == @YES
    If (userage > 0) && (userage < 150)
        valid = @YES
    endif
endif
Return
```

See Also:

[ParseData](#), [Return](#)

Ceiling

Calculates the ceiling of a value

Syntax:

Ceiling(x)

Parameters:

(f) x value **Ceiling** is calculated from.

Returns:

(f) a floating point number whose value represents the smallest integer that is greater than or equal to x. (rounds up to the nearest integer#)

Use this function to calculate the ceiling of a value.

Example:

```
; This example accepts a value from the user to calculate
; the ceiling and floor.
;
a=AskLine("Ceiling and Floor", "Please enter a number", "1.23")
c=Ceiling(a)
f=Floor(a)
Message("Ceiling and Floor of %a%", "Ceiling: %c%    Floor: %f%")
```

ie.	A=	Ceiling=	Floor=
	25.2	26.0	25.0
	25.7	26.0	25.0
	24.9	25.0	24.0
	-14.3	-14.0	-15.0

See Also:

[Abs](#), [Fabs](#), [Floor](#), [Min](#), [Max](#)

Char2Num

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

Syntax:

Char2Num (string)

Parameters:

(s) string any text string. Only the first character will be converted.

Returns:

(i) 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:

[IsNumber](#), [Num2Char](#)

ClipAppend

Appends a string to the Clipboard.

Syntax:

```
ClipAppend (string)
```

Parameters:

(s) string text string to add to Clipboard.

Returns:

(i) @TRUE if string was appended;

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)  
Message ("ClipAppend", "Clip Appended.")
```

See Also:

[ClipGet](#), [ClipPut](#)

ClipGet

Returns the contents of the Clipboard.

Syntax:

```
ClipGet ( )
```

Parameters:

(none)

Returns:

(s) 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](#)

ClipGetEx

Returns the contents of the Windows clipboard in a specified format.

Syntax:

```
ClipGetEx(format)
```

Parameters:

(none)

Returns:

(s) Clipboard contents.

This function is similar to ClipGet, but it allows to you specify the format in which the text is returned. Valid formats are:

<u>Format</u>	<u>Meaning</u>
1	(CF_TEXT) Standard ANSI text (same as ClipGet).
7	(CF_OEMTEXT) Text containing characters in the OEM character set.
13	(CF_UNICODETEXT) Unicode text format (32-bit version under Windows NT only)

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

Example:

```
clipput("abc")
clipappend("def")
a=clipgetEx(1)
Display(3, "ClipGetEx using Format 1 Returns:", a)
if a!="abcdef" then goto ahem

Run("Command.com", "")
    ClipPut(StrCat("dir", Num2char(13)))
    WinActivate("MS-DOS Prompt")
    SendKeysto("MS-DOS Prompt","!{sp}EP")
    TimeDelay(15)
    SendKeysto("MS-DOS Prompt","!{sp}Ek") ;to mark the position
    SendKeysto("~MS-DOS Prompt", "+{RIGHT 60}+{DOWN 24}")
    SendKeysto("~MS-DOS Prompt","!{sp}Ey{ENTER}") ;copy to
clipboard
    SendKeysto("~MS-DOS Prompt","exit{ENTER}")

a=clipgetEx(7)
Display(3, "ClipGetEx using Format 7 Returns:", a)

;NOTE: Unable to test ClipGetEx(13) (Unicode text format) under
Window95--only under NT
if WinMetrics(-4) == 4
    clipput("abc")
    clipappend("def")
    a=clipgetEx(1)
    Display(3, "ClipGetEx using Format 1 Returns:", a)
    if a!="abcdef" then goto ahem
endif
```

See Also:

[ClipAppend](#), [ClipPut](#)

ClipPut

Copies a string to the Clipboard.

Syntax:

ClipPut (string)

Parameters:

(s) string any text string.

Returns:

(i) **@TRUE** if string was copied.

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](#), [SnapShot](#)

Continue

The **Continue** statement in a **While** or **For** loop causes a transfer of control back to the beginning of the loop so that the controlling expressions can be re-evaluated. In a **Switch** or **Select** statement, execution of a particular case is terminated and a search for the next matching case is initiated.

Syntax:

Continue

Parameters:

none

In **While** or **For** statements, use the **Continue** statement to immediately stop execution and re-evaluate the **While** or **For** statement to determine if the loop should be repeated. In **For** statements, the index variable is also incremented. In **Switch** or **Select** statements, if a case is being executed, execution of that case is terminated, and a search is started for another case statement whose expression evaluates to the same integer as the expression controlling the **Switch** or **Select** statement.

Example:

```
a=0
b=0
while (a<100)
    a=a+1
    if a>10 then continue
    b=b+1
endwhile
Message("Continue", "Continued")
```

See Also:

[Break](#), [For](#), [While](#), [Switch](#), [Select](#)

Cos

Calculates the cosine.

Syntax:

`Cos(x)`

Parameters:

(f) *x* angle in radians.

Returns:

(f) the **Cos** function returns the cosine of *x*.

Calculates the cosine. If *x* is large, a loss in significance in the result or a significance error may occur.

Note: To convert an angle measured in degrees to radians, simply multiply by the constant **@Deg2Rad**.

Example:

```
real=AskLine("Cosine", "Enter an angle in degrees ( 0 to 360)", "45")
answer=cos(real * @Deg2Rad)
Message("Arccos of %real% degrees is",answer)
```

See Also:

[Acos](#), [Asin](#), [Atan](#), [Cosh](#), [Sin](#), [Tan](#)

Cosh

Calculates the hyperbolic cosine.

Syntax:

Cosh(x)

Parameters:

(f) x angle in radians.

Returns:

(f) the **Cosh** function returns the hyperbolic cosine of x.

Calculates the hyperbolic cosine. If the result is too large, the function will return an error.

Note: To convert an angle measured in degrees to radians, simply multiply by the constant **@Deg2Rad**.

Example:

```
real=AskLine("Cosh", "Enter an angle in degrees (0 to 360)", "45")
answer=cosh(real * @Deg2Rad)
Message("Hyperbolic cosine of %real% degrees is",answer)
```

See Also:

[Acos](#), [Asin](#), [Atan](#), [Cos](#), [Sin](#), [Sinh](#), [Tan](#), [Tanh](#)

CurrentFile {*M}

Returns the selected filename.

Syntax:

CurrentFile ()

Parameters:

(none)

Returns:

(s) currently-selected file's name.

When a WIL menu shell displays the files in the current directory, one of them may be "selected". This function returns the name of that file, if any.

This is different than 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 many of the shell or file manager-type applications which use the WIL Interpreter.

Example:

```
;The CurrentFile function is designed for menu based
;implementations of WIL, such as FileMenu. It is not
;supported in the WinBatch Product.
;Ask which program to show (default = current file)
;Edit Current File
    thefile = AskLine("Show File", "Program:",CurrentFile( ))
    Run("browser.exe",thefile)
```

See Also:

[CurrentPath](#), [DirGet](#), [DirItemize](#), [FileItemize](#)

CurrFilePath {*M}

Returns the full path plus filename of the currently-selected file.

Syntax:

```
CurrFilePath( )
```

Parameters:

(none)

Returns:

(s) path and filename of currently-selected file.

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.

Example:

```
;The CurrFilePath function is designed for menu based
;implementations of WIL, such as FileMenu. It is not
;supported in the WinBatch Product.
;Get the filename before changing directories.
;Edit Current File
    myfile=CurrFilePath( )
    DirChange("c:\temp")
    Run("notepad.exe", myfile)
```

See Also:

[CurrentFile](#), [CurrentPath](#)

CurrentPath {*M}

Returns path of the selected filename.

Syntax:

```
CurrentPath ( )
```

Parameters:

(none)

Returns:

(s) path of currently-selected file.

When a WIL menu shell displays the files in the current directory, one of them may be "selected." This function returns the drive and path of that file, if any.

This is different than 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 many of the shell or file manager-type applications which use the WIL Interpreter.

Example:

```
; Builds full filename before changing directories.  
; For Menu based systems only
```

```
;Build FileName  
myfile = StrCat(CurrentPath( ), CurrentFile( ))  
DirChange("c:\temp")  
Run("notepad.exe", myfile)
```

See Also:

[CurrentFile](#), [DirGet](#), [FilePath](#)

DateTime

Provides the current date and time.

Note: This function has been replaced by [TimeDate](#), but will still work in this version for compatibility reasons. See **TimeDate** for more information.

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 app.

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("report.exe", "sales.dat")                    ;Run Report
channel = DDEInitiate("report", "YTD")        ;Initialize DDE
If channel != 0                                    ;If DDE OK Execute DDE Command
    result = DDEExecute(channel, '[Act:p="ABCco", t=580.00]')
    DDETerminate(channel)                        ;Close DDE
    WinClose("Reports")                         ;Close Report
    If result == @FALSE
        Message("DDE Execute", "Failed")
    else
        Message("DDE Execute", "Operation complete")
        Exit
    endif
else
    Message("DDE operation unsuccessful", "Check your syntax")
endif
```

See Also:

[DDEInitiate](#), [DDEPoke](#), [DDERequest](#), [DDETerminate](#), [DDETimeout](#)

DDEInitiate

Opens a DDE channel.

Syntax:

DDEInitiate (app name, topic name)

Parameters:

- (s) app name name of the application (without the **EXE** extension).
- (s) topic name name of the topic you wish to access.

Returns:

- (i) communications channel, or **0** on error.

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) 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.

Example:

```
Run("report.exe", "sales.dat")            ;Run Report
channel = DDEInitiate("report", "YTD")   ;Initialize DDE
If channel != 0                           ;If DDE OK Execute DDE Command
    result = DDEExecute(channel, '[Act:p="ABCco", t=580.00]')
    DDETerminate(channel)                ;Close DDE
    WinClose("Reports")                 ;Close Report
    If result == @FALSE
        Message("DDE Execute", "Failed")
    else
        Message("DDE Execute", "Operation complete")
        Exit
    endif
else
    Message("DDE operation unsuccessful", "Check your syntax")
endif
```

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", "")                                    ;Run Reminder
channel = DDEInitiate("Reminder", "items")                ;Initialize DDE
If channel != 0                                             ;If DDE OK
                                                           ;Do DDE Poke
    result = DDEPoke(channel, "all", "11/3/92 Misc Vote!!!!")
    DDETerminate(channel)                                 ;Close DDE
    WinClose("Reminder")                                 ;Close Application
    If result == @FALSE
        Message("DDE Poke", "Failed")
    else
        Message("DDE Poke", "Operation complete")
        Exit
    endif
else
    Message("DDE operation unsuccessful", "Check your syntax")
endif
```

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 from the server if successful, ******NACK****** on failure.

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("report.exe", "sales.dat")                    ;Run Report
channel = DDEInitiate("report", "YTD")           ;Initialize DDE
If channel != 0                                    ;If DDE OK
                                                 ;Do DDE Request
    result = DDERequest(channel, 'TotalSales')
    DDETerminate(channel)                        ;Close DDE
    WinClose("Reports")                         ;Close Report
    If result == @FALSE
        Message("DDE Execute", "Failed")
    else
        Message("DDE Request", "Total Sales is %result%")
        Exit
    endif
else
    Message("DDE operation unsuccessful", "Check your syntax")
endif
```

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("report.exe", "sales.dat")                    ;Run Report
channel = DDEInitiate("report", "YTD")           ;Initialize DDE
If channel != 0                                    ;If DDE OK
                                                 ;Do DDE Request
    result = DDERequest(channel, 'TotalSales')
    DDETerminate(channel)                        ;Close DDE
    WinClose("Reports")                         ;Close Report
    If result == @FALSE
        Message("DDE Execute", "Failed")
    else
        Message("DDE Request", "Total Sales is %result%")
        Exit
    endif
else
    Message("DDE operation unsuccessful", "Check your syntax")
endif
```

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) ;Set Timeout to 5 secs
Run("report.exe", "sales.dat") ;Run Report
channel = DDEInitiate("report", "YTD") ;Initialize DDE
If channel != 0 ;If DDE OK
    ;Do DDE Request
    result = DDERequest(channel, 'SortByCity')
    DDETerminate(channel) ;Close DDE
    WinClose("Reports") ;Close Report
    If result == @FALSE
        Message("DDE Execute", "Failed")
    else
        Message("DDE Request", "Database sorted")
        Exit
    endif
else
    Message("DDE operation unsuccessful", "Check your syntax")
endif
```

See Also:

[DDEExecute](#), [DDEInitiate](#), [DDEPoke](#), [DDERequest](#), [DDETerminate](#)

Debug

Controls the debug mode.

Syntax:

Debug (mode)

Parameters:

(i) mode **@ON** or **@OFF**

Returns:

(i) previous debug mode

Use this function to turn the debug mode on or off. The default is **@OFF**.

When debug mode is on, the WIL 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 which gives the user four options: **N**ext, **R**un, **C**ancel and **S**how **V**ar.

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 WIL program.

Show **V**ar displays the contents of a variable whose name the user entered in the edit box.

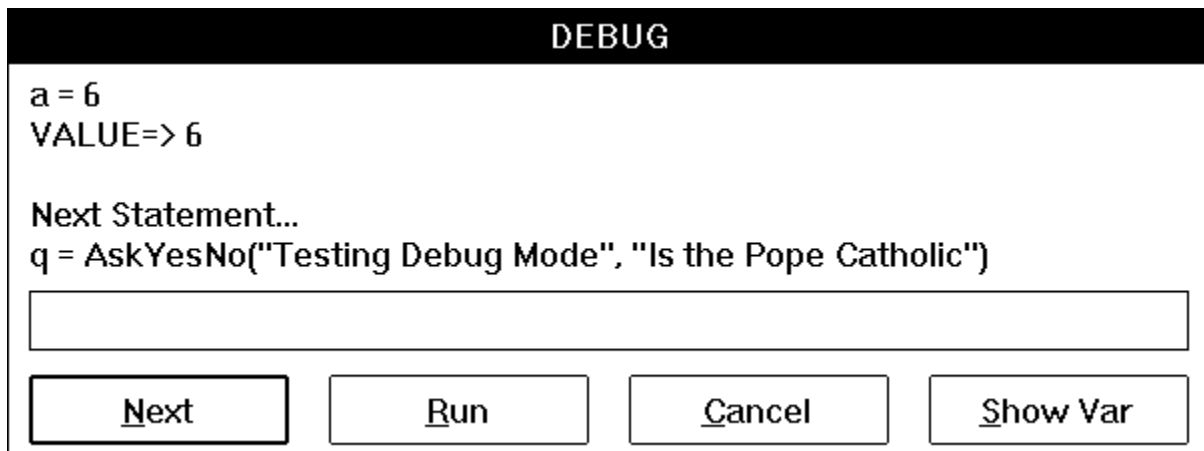
Example:

```
Debug (@ON)
a = 6
q = AskYesNo("Testing Debug Mode", "Is the Pope Catholic")
Debug (@OFF)
b = a + 4
```

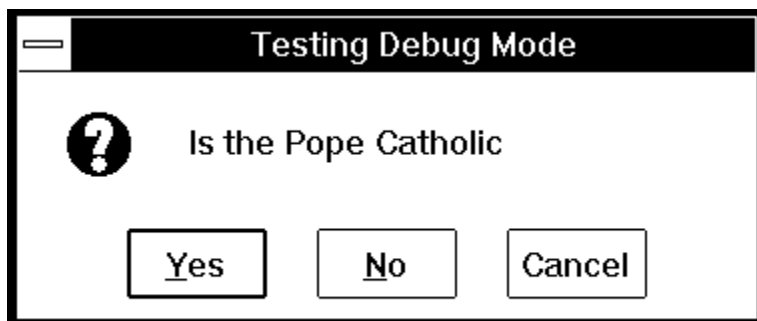
produces:

The image shows a dialog box titled "DEBUG". Inside the box, the text reads: "*** Debug Initialized ***", "VALUE=> 0", "Next Statement...", and "a = 6". Below this text is a large empty rectangular box for input. At the bottom of the dialog box are four buttons: "Next", "Run", "Cancel", and "Show Var".

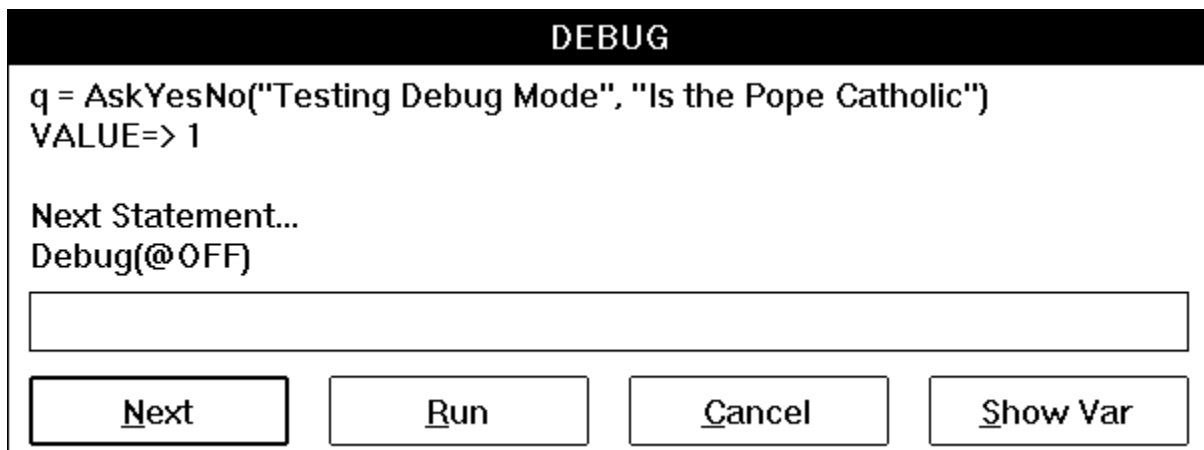
... then, if the user presses **Next**:



... and presses **Next** again:



... and then presses **Yes**:



etc. (If the user had pressed **No** it would have said "VALUE=>0".)

See Also:

ErrorMode, DebugData, DebugTrace, LastError

DebugData

Writes data via the Windows OutputDebugString function to the default destination.

Syntax:

```
DebugData(string, string)
```

Parameters:

(s) string desired data.
(i) string more desired data.

Returns:

(i) always zero

Writes data via the Windows OutputDebugString function to the default destination. The function is generally only useful if you have the proper tools and hardware to debug Windows applications. In general, for standard retail Windows, the default destination is COM1. The Windows SDK provides tools (DBWIN) to allow you to capture the debug data to an alternate device or to a special window.

Use of this function in standard retail Windows may interfere with any device, such as a mouse or modem connected to COM1.

For users without sophisticated (and expensive) debugging tools, the WIL **Debug** function and the WIL **Message** function work incredibly well.

Example:

```
a=45  
DebugData("Value of a is", a)  
; or for those without expensive tools  
Message("Value of a is", a)
```

See Also:

[Debug](#), [DebugTrace](#), [Message](#)

DebugTrace

Creates and writes a line by line debugging file of each statement executed in a script.

Syntax:

```
DebugTrace( mode, filename)
```

Parameters:

- (i) mode **@ON** or **@OFF**
- (s) filename file to write debugging info to.

Returns:

- (i) Always 0.

Note: DebugTrace will create a file showing each line executed and the return value of the line. It will consume considerable processing time and is generally reserved for particularly hard-to-debug problems.

Example:

```
DebugTrace(@ON,"C:\temp\trace.txt")
for xx=1 to 10
    a=5
next
Message("Text","Complete")
exit
```

See Also:

[ErrorMessage](#), [Debug](#), [DebugData](#)

Decimals

Sets the number of decimal places to be used when displaying floating point numbers.

Syntax:

Decimals(places)

Parameters:

(i) places number of decimals to be displayed.

Returns:

(i) previously set value.

Use this function to set the number of decimal places to be displayed when viewing a floating point number. The floating point number will be rounded to the specified number of decimals. If you are doing computations on US currency -- mortgage or financial calculations -- use **Decimals(2)**. Use **-1** for full precision, dropping of trailing zeros.

Example:

```
a=1.23456789
for d = 0 to 10
    Decimals(d)
    Message("Decimals = %d%", a)
Next
```

See Also:

<none>

Delay

Pauses execution for a specified amount of time.

Note: This function has been replaced by [TimeDelay](#), but will still work in this version for compatibility reasons. See [TimeDelay](#) for more information.

Dialog

Displays a user-defined dialog box.

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.

Note: The DialogEditor has been included to create your dialogs. The following information is for technical reference only.

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="WWDLGED,5.0"
```

where <name> is the dialog name. "WWDLGED,5.0" is the hard coded format which identifies this dialog box as using the WIL interpreter Version 5.0. 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>Caption="<box-caption>"  
<name>Y=<y-origin>  
<name>Width=<box-width>  
<name>Height=<box-height>  
<name>NumControls=<ctrl-count>
```

where:

<name>	is the internal name of the dialog box, as described above.
<box-caption>	is the text which will appear in the title bar of the dialog box.
<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.
<ctrl-count>	is the total number of controls in the dialog box (see below).

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 file-list boxes and item boxes.]
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. For pushbuttons, "var" should be DEFAULT. **Note:** Every dialog box must contain at least one pushbutton.

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. When a dialog box opens, the default radiobutton in each group (i.e., the one which is pressed) is the one which has the lowest ordinal position. 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** (unchecked) or **1** (checked). Each checkbox in a dialog should use a unique "var". Normally, when a dialog box opens, every checkbox defaults to being unchecked. 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.

Note: Variable names that begin with "PW_", will be treated as password fields causing asterisks to be echoed for the actual characters that the user types.

STATICTEXT

Descriptive text, which does not change. This can be used to display titles, instructions, etc. For static text 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.

ITEMBOX

A selection list box. The variable "var" is assumed to contain a tab delimited list. The list is loaded into the list box in the original order (Use the **ItemSort** function if a sorted list is desired.). The user may choose none, one, or more items in the list. When the dialog box is closed, the selected items are returned via the "var" variable as a tab delimited list. If the user selects more than 99 items, an error will occur.

FILELISTBOX

A file selection list box. 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, filelist boxes

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. Normally, if a dialog contains a filelistbox, you must select a file from the list box before you can exit the dialog. You can change this behavior by placing the statement **IntControl(4, 0, 0, 0, 0)** anywhere in your WIL program prior to the **Dialog** statement. 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.

You can have a maximum of 100 controls in a dialog.

Example:

```
; Define the dialog format
EditFormat=`WWDDLGED,5.0`
EditCaption=`Edit INI File`
EditX=80
EditY=40
EditWidth=150
EditHeight=170
EditNumControls=14
Edit01=`5,3,40,DEFAULT,STATICTEXT,DEFAULT,"&Directory:"`
Edit02=`42,3,100,DEFAULT,VARYTEXT,editfile,"" `
Edit03=`5,15,80,DEFAULT,EDITBOX,editfile,"" `
Edit04=`5,30,40,DEFAULT,STATICTEXT,DEFAULT,"&File:"`
Edit05=`5,43,80,125,FILELISTBOX,editfile,DEFAULT`
Edit06=`98,17,44,DEFAULT,CHECKBOX,backup,"Make &BAK",1`
Edit07=`98,40,40,DEFAULT,RADIOBUTTON,state,"No&rmal",1`
Edit08=`98,52,40,DEFAULT,RADIOBUTTON,state,"&Zoomed",2`
Edit09=`98,64,40,DEFAULT,RADIOBUTTON,state,"&Iconized",3`
Edit10=`95,82,44,DEFAULT,PUSHBUTTON,DEFAULT,"&Notepad",1`
Edit11=`95,98,44,DEFAULT,PUSHBUTTON,DEFAULT,"&WinEdit",2`
Edit12=`95,114,44,DEFAULT,PUSHBUTTON,DEFAULT,"Wri&te",3`
Edit13=`95,130,44,DEFAULT,PUSHBUTTON,DEFAULT,"WinW&ord",4`
Edit14=`91,151,52,DEFAULT,PUSHBUTTON,DEFAULT,"&Cancel",0`

editfile = "*.INI"           ; Set default mask for filelistbox
backup = 1                   ; Set the checkbox to be on by default
state = 2                    ; Set the 2nd radio button as the default

; Display the dialog, and wait for the user to press one of the
; pushbuttons. The variable "retval" will be equal to the value of
; whichever pushbutton is pressed.
while @TRUE
    retval = Dialog("Edit")
    ; If the user didn't select a valid file, re-display the dialog
    If FileExist(editfile) Then break
endwhile
```

```

; Find out if the checkbox was checked, and proceed accordingly
If backup == 1
    bakfile = StrCat(FileRoot(editfile), ".BAK")
    FileCopy(editfile, bakfile, @TRUE)
endif

; Find out which radio button was pressed, and set the variable
; "runcmd" to the name of the appropriate member of the Run "family"
Switch state
    case 1
        runcmd = "Run"
        break
    case 2
        runcmd = "RunZoom"
        break
    case 3
        runcmd = "RunIcon"
        break
endswitch

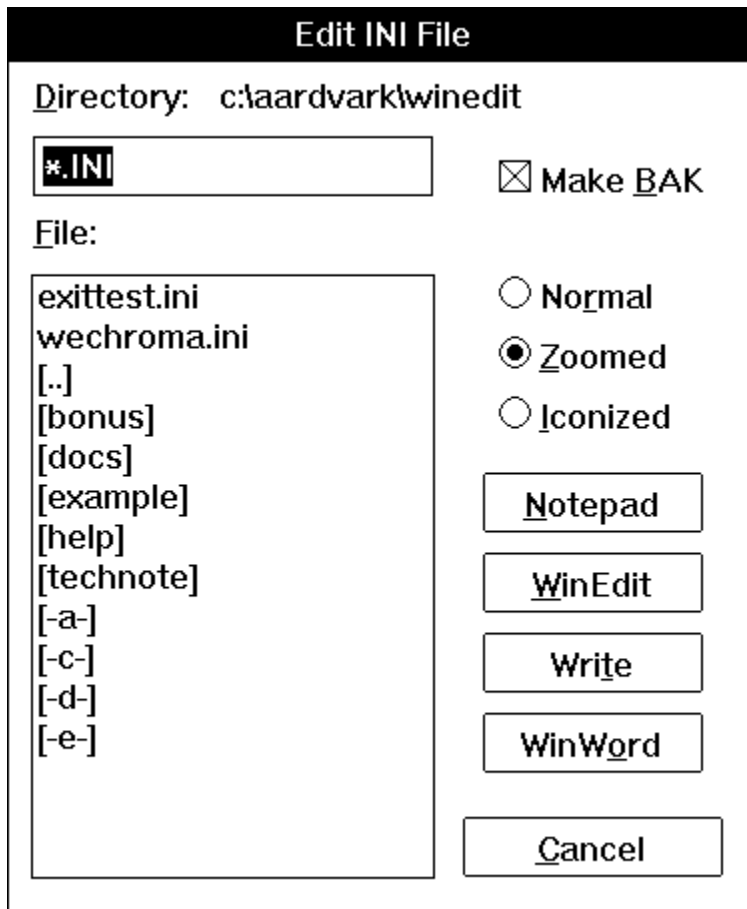
; Set the variable "editor", based on the pushbutton that was pressed
Switch retval
    case 1
        editor = "notepad.exe"
        break
    case 2
        editor = "c:\win\edit\winedit.exe"
        break
    case 3
        editor = "write.exe"
        break
    case 4
        editor = "c:\word\winword.exe"
        break
endswitch

; Execute the appropriate command (using variable substitution)
%runcmd%(editor, editfile)
Exit

:cancel
; If we got here, it means the user pressed the Cancel pushbutton
Message(EditCaption, "Operation cancelled")

```

produces:



See Also:

[AskLine](#), [AskPassword](#), [AskYesNo](#), [IntControl](#), [AskItemList](#)

DirAttrGet

Gets directory attributes.

Syntax:

```
DirAttrGet( [d:]path )
```

Parameters:

(s) [d:]path directory pathname whose attributes you want to determine.

Returns:

(s) the attributes of the specified directory pathname.

Returns attributes for the specified directory, in a string of the form "RASH". This string is composed of four individual attribute characters, as follows:

<u>Char</u>	<u>Symbol</u>	<u>Meaning</u>
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 directory which has the Archive and Hidden attributes set.

Example:

```
dir = "c:\temp"  
attr = DirAttrGet(dir)  
Message("Attributes of Directory, %dir%", attr)
```

See Also:

[DirAttrSet](#), [FileAttrGet](#), [FileAttrSet](#), [FileTimeGet](#)

DirAttrSet

Sets directory attributes.

Syntax:

```
DirAttrSet(dir-list, settings)
```

Parameters:

- (s) dir-list a list of one or more sub-directory names.
- (s) settings new attribute settings for the directories.

Returns:

- (s) always 1.

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):

<u>Symbol</u>	<u>Meaning</u>
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:

```
DirAttrSet("c:\windows\temp", "rASh")  
Message("DirAttrSet","Directory attributes set.")
```

See Also:

[DirAttrGet](#), [FileAttrGet](#), [FileAttrSet](#) , [FileTimeGet](#), [FileTimeTouch](#)

DirChange

Changes the current directory. Can also log a new drive.

Syntax:

```
DirChange ([d:]path)
```

Parameters:

(s) [d:] path an optional disk drive to log onto and the desired path.

Returns:

(i) **@TRUE** if directory was changed,

Use this function to change the current working directory to another directory, either on the same or a different disk drive.

Example:

```
DirChange("c:\")
a = AskFileText("Your CONFIG.SYS", "config.sys", @unsorted, @single)
Message("Contents of selected line, if any", a)
```

See Also:

[DirExist](#), [DirGet](#), [DirHome](#), [LogDisk](#)

DirExist

Tests for the existence of a directory.

Syntax:

```
DirExist(pathname)
```

Parameters:

(s) pathname complete drive and path.

Returns:

(i) **@TRUE** if the directory exists;
 @FALSE if it doesn't exist or if the pathname is invalid.

You can use this function to determine whether a specified drive is valid by checking for the existence of the root directory on that drive.

This function will accept a UNC path.

Examples:

```
; This example checks to see if a directory c:\wp exists. If it
; doesn't one is created.
wmdir = "c:\wp"
If !DirExist(wmdir) Then DirMake(wmdir)
DirChange(wmdir)

; This section asks the user to input a drive, and then checks its
; existence.
while @TRUE                    ; Loop forever, until break or exit
    drive = AskLine("Run Excel", "Enter a drive letter", "")
    If drive == "" Then Exit
    drive = StrSub(drive, 1, 1)
    If DirExist("%drive%:\") then Break
endwhile
Message("Selected Drive", drive)
```

See Also:

[DirChange](#), [DirMake](#), [DirRemove](#), [DirRename](#), [AppExist](#), [FileExist](#), [DiskExist](#)

DirGet

Gets the current working directory.

Syntax:

```
DirGet ( )
```

Parameters:

(none)

Returns:

(s) 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)
Message("DirGet", "Current working directory retrieved and restored.")
```

See Also:

[CurrentFile](#), [CurrentPath](#), [DirHome](#), [DirWindows](#)

DirHome

Returns directory containing the WIL Interpreter's executable files.

Syntax:

```
DirHome ( )
```

Parameters:

(none)

Returns:

(s) pathname of the home directory.

Use this function to determine the directory where the current WIL Interpreter's executable files are stored.

Example:

```
a = DirHome( )  
Message("WIL Executable is in ", a)
```

See Also:

[DirGet](#), [DirWindows](#)

DirItemize

Returns a delimited list of directories.

Syntax:

DirItemize (dir-list)

Parameters:

(s) dir-list a string containing a set of sub-directory names, which may be wildcarded.

Returns:

(s) list of directories.

This function compiles a list of sub-directories and separates the names with a delimiter determined by the WIL version. The 16-bit version returns a space-delimited list and the 32-bit version returns a tab-delimited list.

This is especially useful in conjunction with the **AskItemList** function, which enables the user to choose an item from such a delimited list.

DirItemize(".*")** returns all sub-directories under the current directory.

Note: Some shell or file manager applications using the WIL Interpreter allow an empty string ("") to be used as the "dir-list" parameter, in which case all sub-directories highlighted in the file display are returned. However, if there are any directory names or wildcards in the string, all sub-directories matching the pathnames are returned, regardless of which ones are highlighted.

Note: In the 32-bit version of WIL, the "default" file delimiter used to delimit lists of files and directories, has been changed to a TAB. In the 16-bit version of WIL, the "default" delimiter has not changed, and remains a space. **IntControl(29,0,0,0)** allows the file delimiter to be changed to a character of your own choosing. To standardize a TAB delimiter for both 16 and 32 bit scripts add the **IntControl(29,0,0,0)** to the top of the script.

```
IntControl(29, @TAB, 0, 0, 0)
```

Example:

```
IntControl(29, @TAB, 0, 0, 0)
DirChange(DirWindows(0))
a = DirItemize("**.*")
AskItemList("Directories", a, @TAB, @unsorted, @single)
```

See Also:

[CurrentFile](#), [FileItemize](#), [AskItemList](#), [AskFileText](#), [WinItemize](#)

DirMake

Creates a new directory.

Syntax:

```
DirMake ([d:]path)
```

Parameters:

(s) [d:] path the desired disk drive and the path to create.

Returns:

(i) **@TRUE** if the directory was successfully created.

Use this function to create a new directory. This function will also accept a UNC path.

Example:

```
DirMake("c:\xstuff")  
Message("DirMake", "Directory Made")
```

See Also:

[DirExist](#), [DirRemove](#), [DirRename](#)

DirRemove

Removes an empty directory.

Syntax:

```
DirRemove (dir-list)
```

Parameters:

(s) dir-list a delimited list of directory pathnames.

Returns:

(i) **@TRUE** if the directory was successfully removed.

Use this function to delete empty directories. You can delete one or more at a time by separating directory names with spaces. You cannot, however, use wildcards.

Note: In the 32-bit version of WIL, the "default" file delimiter used to delimit lists of files and directories, has been changed to a TAB. In the 16-bit version of WIL, the "default" delimiter has not changed, and remains a space. **IntControl 29** allows the file delimiter to be changed to a character of your own choosing. To standardize a TAB delimiter for both 16 and 32 bit scripts add the **IntControl(29,@TAB,0,0,0)** to the top of the script.

Examples:

```
DirRemove("c:\xstuff")
Message("DirRemove","Directory removed")

;or
;DirRemove("tempdir1 tempdir2 tempdir3")
```

See Also:

[DirExist](#), [DirMake](#), [DirRename](#)

DirRename

Renames a directory.

Syntax:

```
DirRename ([d:]oldpath, [d:]newpath)
```

Parameters:

(s) oldpath existing directory name, with optional drive.
(s) newpath new name for directory.

Returns:

(i) **@TRUE** if the directory was successfully renamed.

Example:

```
;Do a dirmake so the example runs smoothly.  
  
dirmake("c:\wwwtemp")  
  
;Once 'old' directory is created execute the example  
;This is the real example.  
  
old="c:\wwwtemp"  
new="c:\wwwwork"  
DirRename("c:\wwwtemp", "c:\wwwwork")  
Message("DirRename", "%old% Directory renamed to %new%")
```

See Also:

[DirExist](#), [DirMake](#), [DirRemove](#)

DirSize

Finds the total size of a directory.

Syntax:

```
DirSize (dir-name, flags)
```

Parameters:

(s) dir-name directory name
(i) flags 0 or 1. see below.

Returns:

(i) size in bytes. The return value will be a floating point number if it is larger than 2 gigabytes.

This function returns the total size of a directory, including all files in the directory, and all files in all subdirectories under the directory.

Flags	Meaning
----	-----
0	Actual size of the files.
1	Instead of returning the actual size of the files, return the amount of disk space they occupy. This is based upon the disk's cluster size. This flag is valid only if the specified directory is on a local drive or a network drive mapped to a drive letter; it does not support UNC's.

Example:

```
a=Dirsize("C:\Windows",0)
Message("DirSize : Actual size",a)

b=Dirsize("C:\Windows",1)
Message("DirSize :The amount of disk space occupied",b)
```

See Also:

[DirExist](#), [DirItemize](#)

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.

<u>Req#</u>	<u>Return value</u>
0	Windows directory
1	Windows System directory

Example:

```
IntControl(29, @TAB, 0, 0, 0)
DirChange(DirWindows(0))
files=FileItemize("*.ini")
ini = AskItemList("Select file", files, @TAB,@unsorted, @single)
Run("notepad.exe", ini)
```

See Also:

[DirGet](#), [DirHome](#)

DiskExist

Tests for the existence of a drive.

Syntax:

```
DiskExist(driveletter)
```

Parameters:

(s) driveletter drive being tested.

Returns:

(i) **@TRUE** if the drive was found;
 @FALSE if the drive was not found.

Use this function to test for the existence of a specific disk drive.

Example:

```
b="A:"  
a=DiskExist(b)  
if a  
    Message("Disk Exists", b)  
else  
    Message("Disk Does Not Exist", b)  
endif
```

See Also:

[AppExist](#), [FileExist](#), [DirExist](#), [DiskScan](#), [DiskFree](#), [LogDisk](#)

DiskFree

Finds the total available (unused) disk space on a group of drives.

Syntax:

DiskFree (drive-list)

Parameters:

(s) drive-list one or more drive letters, separated by a delimiter.

Returns:

(i) 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.

Results larger than 2 gigabytes will be returned as a floating point number. However, it should be noted that in Windows 95 prior to OSR 2 incorrect values may be returned for volumes larger than 2 gigabytes.

Example:

```
size = DiskFree("c d")
Message("Space Available on C: and D:", size)
```

See Also:

[DiskScan](#), [FileSize](#)

DiskInfo

Returns information on the specified disk drive.

Syntax:

```
DiskInfo(drive-letter, request#)
```

Parameters:

(s) drive-letter a drive letter
(i) request# see below.

Returns:

(i) information per request#

Req#	Return value
1	sectors per cluster
2	bytes per sector
3	number of free clusters
4	total number of clusters

Example:

```
spc = DiskInfo("c", 1)
bps = DiskInfo("c", 2)
bpc = spc * bps
Message("Cluster size on drive C:", bpc)
```

See Also:

[DiskSize](#), [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 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 removable (floppy) drives
2	List of local fixed (hard) drives
4	List of remote (network) drives
8	CD-ROM (32 bit versions of WIL only)
16	RamDisk (32 bit version of WIL only)

Note: In the 32-bit version of WIL, the "default" file delimiter used to delimit lists of files and directories, has been changed to a TAB. In the 16-bit version of WIL, the "default" delimiter has not changed, and remains a space. **IntControl(29,0,0,0)** allows the file delimiter to be changed to a character of your own choosing. To standardize a TAB delimiter for both 16 and 32 bit scripts add the **IntControl(29,0,0,0)** to the top of the script.

```
IntControl(29, @TAB, 0, 0, 0)
```

Example:

```
hd = DiskScan(2)
Message("Hard drives on system", hd)
```

See Also:

[DiskFree](#), [LogDisk](#)

DiskSize

Finds the total size of the specified drive(s).

Syntax:

DiskSize (drive-list)

Parameters:

(s) drive-list one or more drive letters, separated by a delimiter.

Returns:

(i) the total size of a selected disk.

This function takes a string consisting of drive letters, separated by the current file delimiter character. See [File Delimiters](#). 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. Results larger than 2 gigabytes will be returned as a floating point number.

However, it should be noted that in Windows 95 prior to OSR 2 incorrect values may be returned for volumes larger than 2 gigabytes.

Example:

```
size = DiskSize("c")
Message("Size of C:", size)
```

See Also:

[DiskScan](#), [FileSize](#)

DiskVollInfo {*32}

Returns information on a file system volume.

Syntax:

```
DiskVollInfo(root-path, request#)
```

Parameters:

- (s) root-path specifies the root directory of a file system volume (ie "C:\\" or "\\SERVER\PUBLIC\")
- (i) request # the request number for desired information.

Returns:

- (s) Request numbers which indicate a return type of "(b)" (Boolean) return a value of 1 (@TRUE) or 0 (@FALSE).

Req#	Return value
1	(s) volume name
2	(s) file system name (eg, "FAT" or "NTFS")
3	(s) volume serial number (hex string; eg, "1234-ABCD")
4	(i) volume serial number (decimal)
5	(i) max length of filename component (portion between backslashes)
6	(b) volume preserves the case of filenames when storing files
7	(b) volume supports case-sensitive filenames
8	(b) volume supports Unicode in filenames
9	(b) volume preserves and enforces ACL's
10	(b) volume supports file-based compression
11	(b) volume is a compressed volume (eg, a DoubleSpace volume)

Example:

```
vol = DiskVollInfo("c:\", 1)
Message("Volume name of drive C:", vol)
```

See Also:

[DiskInfo](#)

Display

Displays a message to the user for a specified period of time.

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. |

Returns:

- | | |
|-----|---|
| (i) | @TRUE if terminated by user;
@FALSE otherwise. |
|-----|---|

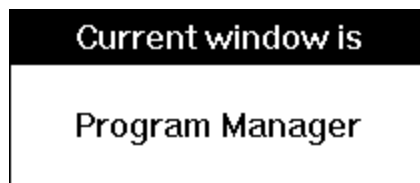
Use this function to display a message for a few seconds, and then continue processing without user input. **Seconds** must be an integer between 1 and 3600. Smaller or larger numbers will be adjusted accordingly.

The user can make the displayed message disappear before the designated time has elapsed by clicking a mouse button, or by pressing the space or enter key. If the user terminates the function in this manner, it will return a value of **@TRUE**; otherwise, it will return **@FALSE**.

Example:

```
Display(3, "Current window is", WinGetActive( ))
```

which produces something like this:



See Also:

[Message](#), [Pause](#)

DllCall

Calls an external DLL.

Syntax:

```
DllCall(dllname, returntype:epname, paramtype:parameter [paramtype:parameter ...] )
```

Parameters:

(s) dllname	The name of the Dll to be called, or a handle returned by the DllLoad function.
(t) returntype:	Type of value the Dll entry point will return (see below).
(s) epname	Entry point name into the Dll.
(t) paramtype	Type of parameter (see below).
(?) parameters	Parameters as required by the entry point.

Returns:

Value returned by the DllCall depends on the external Dll. It may be either a integer or a string. See discussion below.

The DllCall function is unlike all other WIL functions. It is designed to allow sophisticated users to either write their own extensions to the WIL language (using the Windows SDK), to call third party Dlls, or to access Windows APIs directly.

In order to use this function properly, a little background is necessary. There exists a number of very specific reasons one would want to call an external DLL to process some code. Examples may include calling Dlls to interface with certain hardware devices, to perform special compute-intensive algorithms, or to perform a series of functions not possible using the WIL language. In many cases, the user has no control over the DLLs to be called, so that the WIL **DllCall** statement must be able to call a wide variety of Dlls, to be able to pass an assortment of different parameter types, and to be able to process a number of different return values.

For this reason, the **DllCall** syntax is complicated and initially confusing. Use of the **DllCall** requires detailed understanding of Windows programming and complete documentation for the Dll and the Dll entry point being called. If you need tech support help with the **DllCall** statement, you must fax pertinent documentation before calling for help.

To call an external Dll, the user must first determine the following information:

- 1) Name of the DLL.
- 2) Entry point name of the desired function within the Dll.
- 3) Type of the return value from the Dll.
- 4) Number of passed parameters the Entry point requires.
- 5) Type of each of the passed parameters.

WIL supports the following types of return types from a Dll:

- | | | |
|----|-------|----------------------------|
| 1) | word | 16 bit integer |
| 2) | long | 32 bit integer |
| 3) | lpstr | 32 bit pointer to a string |
| 4) | void | no return value |

WIL supports the following parameter types to pass data to a Dll:

- | | | |
|----|----------|--|
| 1) | word | 16 bit integer |
| 2) | long | 32 bit integer |
| 3) | lpstr | 32 bit pointer to a string |
| 4) | lptr | 32 bit NULL pointer |
| 5) | lpbinary | 32 bit pointer to a memory block allocated with the BinaryAlloc function. See section on <u>Binary Operations</u> . |

Note: If **lpbinary** is used to pass information from a Dll back to a WIL script via a **DllCall**, then be sure to use **BinaryEodSet** to manually set the end of data point so that the other binary functions can reference the returned data.

The DllCall parameters are as follows:

First: The first parameter defined the Dll to be used. It can either be a dllname or a dllhandle. A dllname may be used for "oneshot" types of calls where a single call to the Dll is all that is required, or when each call to the Dll is independent of all other calls. A dllhandle is used for multiple calls to a Dll, where the calls are interrelated -- perhaps the first call to initialize the Dll, other calls use it, and a final call to terminate processing. In such cases the Dll must first be loaded with the **DllLoad** function, and freed with the **DllFree** function. The first parameter must be one of the following:

dllname: Simply the filename of the Dll that contains the desired entry point name. A single Dll may contain one to many separate entry points. Each entry point may have its own unique return type and parameter list.

dllhandle: A handle do a Dll obtained from the **DllLoad** function.

Second: The second parameter consists of two parts, the first part is the return type of the entry point desired, followed by a colon (:), and the second part is the entry point name itself.

Note: Only use the **lpstr** return type for text strings. Even though some other documentation might suggest using a **lpstr** as a return type for its structures, dont. Use **long** instead.

For each parameter the entry point requires, an additional parameter is added to the **DllCall** parameter list. If the entry point has no parameters, then the **DllCall** function uses only the first and second parameters as described above.

Additional: For each parameter that the entry point in the Dll requires, additional **DllCall** parameters are added. Each additional parameter consists

of two parts, the first part is the parameter type of the required parameter, followed by a colon (:), and the second part is the parameter itself.

Example:

```
; DllCall example.
; This example calls the CharUpperA API in the Windows User module.
; The CharUpperA function requires a 32 bit pointer to a string
(lpstr).
; It converts the string to uppercase and passes back a 32 bit
; pointer (also lpstr) to the uppercased string.
; The CharUpperA function is found in the Windows USER32.Dll.
; Note: Dll Name, being a normal string is in quotes.
;           Entry point name, also being a string, is also in
quotes
;           Parameter a0, being a normal variable is not in quotes.

a0="Hello Dolly"
dllname=strcat(dirwindows(1),"USER32.DLL")
a1=DllCall(dllname, lpstr:"CharUpperA", lpstr:a0)
Message(a0,a1)
```

For an example on how to use binary buffers to receive data returned through a passed pointer, see the DllHwnd example.

See Also:

[Binary Operations](#), [DllCall](#) [Additional information](#), [DllLoad](#), [DllFree](#), [DllHwnd](#), [DllHinst](#)

DllCall Additional information

In 16 bit versions of Windows, functions that are called using **DllCall** must use the `__pascal` calling convention (declared as FAR PASCAL or WINAPI). In 32 bit versions of Windows, they must use the `__stdcall` calling convention (declared as WINAPI). Otherwise, **DllCall** will return a "Bad Entrypoint" or "Bad Parameter List" error message, even though you have specified the correct function name and parameter types; this would likely indicate that the function is using an unsupported calling convention.

Problem:

Under 32-bit Windows, using **DllCall** to call a function in a custom DLL that you've developed produces the error message:

```
"NT DllCall: Bad Parameter List"
```

followed by the error message:

```
"1379: DllCall: Bad type list caused stack problems.  
Check types carefully."
```

First check the number of parameters and the parameter types carefully to make sure that they are indeed correct. If they are, it is likely that the problem is due to your function using the `__cdecl` calling convention, instead of the required `__stdcall`. To change this, follow these steps:

1. Add the keyword "WINAPI" to your function prototype and declaration:

```
LONG WINAPI MyFunction(LPSTR);  
LONG WINAPI MyFunction(LPSTR lpString)
```

This ensures that the function will use the `__stdcall` calling convention, instead of the default `__cdecl` convention. **DllCall** requires `__stdcall`, in which the called function is responsible for removing the parameters from the stack (similar to `__pascal` in 16 bit versions of Windows). The WIL program checks the stack pointer before and after the **DllCall**; if they are not the same, this indicates that either (1) you did not specify the correct parameters to **DllCall**, or (2) the called function did not clean up the stack properly (probably because it wasn't using `__stdcall`).

Alternatively, in Visual C++ you can use the `/Gz` compiler option (or set Calling Convention to `__stdcall` under "Project | Settings | C/C++ | Category: Code Generation" in the IDE) to make all your functions use `__stdcall`, but it's wise to specify WINAPI in the declarations as well.

2. Add the option `/EXPORT:MyFunction` to the (VC++) LINK command line.

Or, if you have more than one exported function, it may be easier to create a module definition (.DEF) file with an EXPORTS section (or add an EXPORTS section to your existing .DEF file):

```
EXPORTS
    MyFunctionA
    MyFunctionB
```

If you use the .DEF file method, you will also need to add the option "/DEF:filename" to the (VC++) LINK command line, where "filename" is the name of your module definition file (by default, VC++ 2.x does not create or use .DEF files).

This is necessary, even if you have specified "__declspec(dllexport)" in the function declaration, because __stdcall "decorates" (mangles) the function name when it is exported, so that in the DLL it becomes:

```
_MyFunction@4
```

where the number following the '@' symbol is the stack space used by the function (the parameter count * 4). This prevents **DllCall** from accessing the function. Exporting the function using the /EXPORT option (or via the EXPORTS section) causes the real, un-decorated name to be exported.

After you've done this, it's no longer necessary to declare the function as "__declspec(dllexport)", although it certainly wouldn't hurt to do so.

DllFree

This function frees a DLL that was loaded via the **DllLoad** function.

Syntax:

```
DllFree(dllhandle)
```

Parameters:

(i) dllhandle handle of the DLL to be freed.

Returns:

(i) always 0.

Use this function to free DLLs that were loaded with the **DllLoad** function. Failure to free such DLLs will use up system resources.

Example:

```
a0="Hello Dolly"  
dllhandle=DllLoad(strcat(dirwindows(1),"USER32.DLL"))  
a1=DllCall(dllhandle, lpstr:"CharUpperA", lpstr:a0)  
DllFree(dllhandle)  
Message(a0,a1)
```

See Also:

[Binary Operations](#), [DllCall](#), [DllCall Additional information](#), [DllLoad](#)

DllHinst

Obtains an application instance handle for use in **DllCalls** when required.

Syntax:

```
DllHinst(partial-winname )
```

Parameters:

(s) partial-winname the initial part of, or an entire, window name.

Returns:

(i) an application instance handle.

Use this function to return a valid application instance handle (hInst) of the application owning the specified window.

Note: "Partial-winname" 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 windowname "" may be used as a shorthand way of referring to the WIL parent application window.

Example:

```
binbuf=BinaryAlloc(100)
hInst=DllHinst("")
dllname=strcat(dirwindows(1),"KERNEL32.DLL")
DllCall(dllname,word:"GetModuleFileNameA",long:hInst,lpbinary:binbuf,
long:100)

; Note DllCalls do not set EOD point in buffer.
; EOD Point MUST be set manually with BinaryEODSet
BinaryEodSet(binbuf, 100) ;

a=BinaryPeekStr(binbuf, 0, 100)
BinaryFree(binbuf)
Message("Window module filename is", a)
```

See Also:

[Binary Operations](#), [DllCall](#), [DllCall Additional information](#), [DllHwnd](#)

DllHwnd

Obtains a window handle for use in **DllCalls** when required.

Syntax:

```
DllHwnd(partial-winname)
```

Parameters:

(s) partial-winname the initial part of, or an entire, window name.

Returns:

(i) a window handle.

Use this function to return a valid window handle (hWnd) of specified window. Some Dlls require a window handle. This function will provide - in most cases - a usable window handle.

Note: "Partial-winname" 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 windowname "" may be used as a shorthand way of referring to the WIL parent application window.

Example:

```
binbuf=BinaryAlloc(100)
hWnd=DllHwnd("")
DllCall("USER.EXE", word:"GetClassName", word:hWnd, lpbinary:binbuf, word:100)
; Note DllCalls do not set EOD point in buffer.
BinaryEodSet(binbuf, 100)
a=BinaryPeekStr(binbuf, 0, 100)
BinaryFree(binbuf)
Message("Window class name is", a)
```

See Also:

[Binary Operations](#), [DllCall](#), [DllCall Additional information](#), [DllHinst](#)

DllLoad

This function loads a Dll for later use via the **DllCall** function

Syntax:

```
DllLoad(dllname)
```

Parameters:

(s) dllname The name of the Dll to be called.

Returns:

(s) a handle to a Dll for use in **DllCalls**.

When multiple calls are to be made to a Dll, and the calls are interdependent, the Dll should be first loaded via the **DllLoad** command, and the return value - a dllhandle - should be passed to the **DllCall** function instead of a filename.

Example:

```
a0="Hello Dolly"  
dllhandle=DllLoad(strcat(dirwindows(1),"USER32.DLL"))  
a1=DllCall(dllhandle, lpstr:"CharUpperA", lpstr:a0)  
DllFree(dllhandle)  
Message(a0, a1)
```

See Also:

[Binary Operations](#), [DllCall](#), [DllCall Additional information](#), [DllFree](#)

DOSVersion

Returns the version numbers of the current version of DOS.

Syntax:

DOSVersion (level)

Parameters:

(i) level **@MAJOR** or **@MINOR**.

Returns:

(i) 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 5.0, then:

```
DOSVersion(@MAJOR) == 5
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](#), [FileVerInfo](#), [Version](#), [VersionDLL](#), [WinVersion](#)

Drop

Removes variables from memory.

Syntax:

```
Drop (var, [var...])
```

Parameters:

(i) var variable names to remove.

Returns:

(i) always 1.

This function removes variables from the WIL Interpreter's variable list, and recovers the memory associated with the variable (and possibly related string storage).

A variable is defined the first time it appears to the left of an equal sign in a statement. It stays defined until it is explicitly dropped with the **Drop** function, or until the current invocation of the WIL Interpreter gets closed.

Generally speaking: in batch file-based implementations of WIL, all variables are dropped automatically at the end of every batch file; and in menu-based implementations of WIL, variables stay defined until explicitly dropped.

Example:

```
a = "A variable"  
b = "Another one"  
Drop(a, b)            ; This removes A and B from memory  
Message("Drop", "Dropped from memory.")
```

See Also:

[DropWild](#), [IsDefined](#)

DropWild

Removes variables from memory.

Syntax:

DropWild (variable/wildcard)

Parameters:

(s) variable or wildcarded variable.

Returns:

(i) always 1.

This function removes variables from the WIL Interpreter's variable list, and recovers the memory associated with the variable (and possibly related string storage).

A variable is defined the first time it appears to the left of an equal sign in a statement. It stays defined until it is explicitly dropped with the **DropWild** function, or until the current invocation of the WIL Interpreter gets closed.

Generally speaking: in batch file-based implementations of WIL, all variables are dropped automatically at the end of every batch file; and in menu-based implementations of WIL, variables stay defined until explicitly dropped.

Example:

```
a1= "A variable"  
a2= "Another variable"  
Drop("a*") ; This removes A1 and A2 from memory  
Message("DropWild","Dropped from memory.")
```

See Also:

[IsDefined](#), [Drop](#)

EndSession

Ends the Windows session.

Syntax:

```
EndSession ( )
```

Parameters:

(none)

Returns:

(i) always 0.

Use this command to end the current Windows session, just like selecting **Close** from Program Manager's control menu. If any active applications pop up dialog boxes in response to this command (such as prompting to save data files which have changed), the user will need to respond to them before Windows will close.

Note: For additional methods of rebooting or restarting Windows, see `IntControl(66,p1,0,0,0)` , `IntControl(67,p1,0,0,0)` , and `IntControl(68,p1,0,0,0)`.

Example:

```
while AskYesNo ("End Session", "You want to exit Windows?")
    EndSession( )
endwhile
:cancel
Message("", "Exit Windows canceled")
```

See Also:

[Exit](#), [WinClose](#)

Environment

Gets a System environment variable.

Syntax:

Environment (env-variable)

Parameters:

(s) env-variable any defined environment variable.

Returns:

(s) environment variable contents.

Use this function to get the value of a system environment variable.

Note: It is **not** possible to change a DOS environment variable from within Windows.

The maximum value length is 1280 bytes. It will return a blank string ("") for environment variables whose values are longer than 1280 bytes.

HINT: Under 16-bit Windows platforms, use the WWENVMAN.DLL, WIL Environment extender, for enhanced environment management. Further explanations are in WWWENV.HLP.

Example:

```
; Display the PATH for this session
currpath = Environment("PATH")
Message("Current Path", currpath)
```

See Also:

[FileVerInfo](#), [IniRead](#), [Version](#), [WinMetrics](#), [WinParmGet](#), [RegApp](#)

EnvironSet

Changes LOCAL Environment variables.

Syntax:

```
EnvironSet(name, value)
```

Parameters:

(s) name name of environment variable. (See Note 1).
(s) value desired value.

Returns:

(i) **@TRUE** Environment variable was modified.

Use this function to change the LOCAL environment variables.

32 bit versions

32 bit versions of WIL will always pass the local environment to any programs they launch. In the 32 bit versions, the **RunEnviron** command is identical to the **RunShell** Command.

Note 1: DOS expects UPPERCASE environment variable names. Windows NT and Windows 95/98 allow mixed upper and lowercase names. If you are using WIL with DOS, be sure to use uppercase names.

Note 2: To alter the path for DOS programs, all that is required is a simple batch file, and the usual WIL **Run** command. Assuming the case where one wished to run "command.com" with the path "c:\special", a generic batch file as shown below will suffice, along with passing all the information required as parameters in the WIL **Run** command.

```
DoPath.bat  file listing
            SET PATH=%1
            ECHO %PATH%
            PAUSE
            %2 %3 %4 %5 %6 %7 %8 %9
```

```
WIL Run Command
      Run("dopath.bat", "c:\special  command.com")
```

Example:

```
;Note
;16-bit users should use the
; "Environment Manager Extender"
; instead of this function .
;
;NT Users note: Permanent changes may be made by
;using the registry functions to update the registry.
;
;Note that this function will not affect the environment for
; DOS or Console applications
;
;Note: Modifying the path does not allow assist WIL in
;locating the EXE file. The path WIL uses has already
;been pre-determined. However the child process (excel)
;will inherit the PATH environment variable in this case.

EnvironSet("PATH","c:\Program Files\Tmp\")
RunEnviron("c:\Program Files\Tmp\abc.exe", "", @NORMAL, @WAIT)
```

See Also:

[Environment](#), [Envltemize](#), [RunEnviron](#)

EnvItemize

Returns a delimited list of the current environment.

Syntax:

```
EnvItemize( )
```

Parameters:

none

Returns:

(s) a list containing all variables in the current environment (See Note).

Use this function to return a list of the variables in the current environment.

Note: This list is delimited by the newline character (ASCII 10), which can be generated with the [Num2Char](#) function. The returned list is suitable for a message box display. Tabs are not used as a delimiter as they seem to be legal characters within the environment. The [StrReplace](#) function may be used to change the delimiter to any other character.

HINT: Use the WWENVMAN.DII, WIL Environment extender, for enhanced environment management. Further explanations are in WWWENV.HLP.

Example:

```
env=EnvItemize( )
Message("The Environment is", EnvItemize( ))
env=StrReplace(env, Num2Char(10), @TAB)
a=AskItemList("Select a Variable", env, @TAB, @SORTED, @SINGLE)
b=Environment(a)
Message(a, b)
```

See Also:

[Environment](#), [EnvironSet](#)

ErrorMode

Specifies how to handle errors.

Syntax:

ErrorMode (mode)

Parameters:

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

Returns:

(i) previous error setting.

Use this function to control the effects of runtime errors. The default is **@CANCEL**, meaning the execution of the WIL program will be canceled upon 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 WIL program for others to use. You can always suppress errors you expect will occur and then re-enable **ErrorMode(@CANCEL)**.

ONLY put error mode around SINGLE statements where you are handling the errors yourself. If it is obvious that there is no way a statement could fail it should be run with **ErrorMode(@CANCEL)**

Note: Pay close attention when suppressing errors with the ErrorMode function.

When an error occurs, the processing of the ENTIRE line is canceled. The value returned from the function is 0. Setting the **ErrorMode()** to **@OFF** or **@NOTIFY** allows execution to resume at the next line. Various parts of the original line may have not been executed.

Please read examples very carefully before running.

e.g.

```
ErrorMode(@off)
; The FileCopy will cause a file not found error,
; canceling the execution of the whole line.
; The variable A is set to @FALSE by default
    A = FileCopy( "xxxxxxxx", "*.*", @FALSE)
;
; Now there is a NOT symbol in front of the FileCopy.
; Nonetheless, if an error occurs A is still set to @FALSE
; not @TRUE as might be assumed. When an error is suppressed
; with ErrorMode the line is canceled, and any assignment is
; simply set to the default @FALSE value.
    A = !FileCopy("yyyyyyyyy", "*.*", @FALSE)
Message("ErrorMode","ErrorMode executed.")
```

For this reason, **ErrorMode()** must be used with a great deal of care. The function for

which the errors are being suppressed should be isolated from other functions and operators as much as possible.

e.g.

```
; INCORRECT USAGE of ErrorMode( )  
; In this instance, when the copy has an error, the entire if  
; statement is canceled.  
; Execution begins (erroneously) at the next line, and states  
; that the copy succeeded. Next a fatal error occurs as the  
; "else" is found, since it does not have a matching if  
ErrorMode(@OFF)  
if FileCopy(file1,file2,@FALSE) == @TRUE  
    Message("Info", "Copy worked")  
else  
    Message("Error", "Copy failed")  
endif  
  
; CORRECT USAGE  
; In this case, the FileCopy is isolated from other statements  
; and flow control logic. When the statement fails, execution  
; can safely begin at the next line. The variable "a" will  
; contain the default value of zero that a failed assignment  
; returns.  
; Results are not confused by the presence of other operators.
```

```
ErrorMode(@OFF)  
a = FileCopy(file1,file2,@FALSE)  
ErrorMode(@CANCEL)  
if a == @TRUE  
    Message("Info", "Copy worked")  
else  
    Message("Error", "Copy failed")  
endif
```

See Also:

[Debug](#), [Execute](#), [LastError](#)

Exclusive

Controls whether or not other Windows programs will get any time to execute.

Syntax:

Exclusive (mode)

Parameters:

(i) mode **@ON** or **@OFF**.

Returns:

(i) previous **Exclusive** mode.

Exclusive(@OFF) is the default mode. In this mode, the WIL Interpreter is well-behaved toward other Windows applications.

Exclusive(@ON) allows WIL programs to run somewhat faster, but causes the WIL 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.

Note: This function is generally useful in the 16 bit versions of Windows. In versions of Windows with true multi-tasking, the effects of this function are negligible.

Example:

```
Exclusive(@ON)
x = 0
start = TimeDate( )
:add
x = x + 1
If x < 1000 Then Goto add
stop = TimeDate( )
crlf = StrCat(Num2Char(13), Num2Char(10))
Message("Times", "Start: %start%%crlf%Stop: %stop%")
Exclusive(@OFF)
```

See Also:

[Yield](#)

Execute

Executes a statement in a protected environment. Any errors encountered are recoverable.

Syntax:

Execute statement

Parameters:

(s) statement any executable WIL statement.

Returns:

(not applicable)

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: 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%
```

See Also:

[ErrorMode](#)

ExeTypeInfo

Returns an integer describing the type of EXE file specified.

Syntax:

ExeTypeInfo(EXENAME)

Parameters:

(s)EXENAME the name of the desired .EXE, .COM, .PIF, .BAT file or data file.

(Returns:

(i) integer 0 = not an EXE file.
 1 = Old style DOS EXE.
 2 = New Style DOS EXE.
 3 = Windows EXE.
 10 = Windows NT EXE

Use this function to return an integer describing and identifying the type of EXE file specified.

Example:

```
a=ExeTypeInfo(FileLocate("notepad.exe"))
switch a
    case 0
        b="Not an EXE file"
        break
    case 1
        b="Old DOS EXE"
        break
    case 2
        b="New DOS EXE"
        break
    case 3
        b="Windows EXE"
        break
    case 10
        b="Windows NT EXE"
        break
    case a
        b="Unknown file type, value = %a%"
        break
endswitch
Message("File Type", b)
```

See Also:

[AskFileName](#), [FileFullName](#), [FileMapName](#)

Exit

Unconditionally ends a WIL program.

Syntax:

Exit

Parameters:

(none)

Returns:

(not applicable)

Use this command to immediately terminate a WIL program. An **Exit** is implied at the end of each top-level WIL program, and so is not necessary there.

Example:

```
a = 100
Message("The value of a is", a)
Exit
```

See Also:

[IntControl](#), [Pause](#), [Return](#), [Terminate](#)

Exp

Calculates the exponential.

Syntax:

Exp(x)

Parameters:

(f) x floating point number.

Returns:

(f) the value of the exponential (e^{**x}).

The exp function returns the exponential function of the floating point argument (x).

Example:

```
real=AskLine("Exponential", "Enter a number", "1.23")
answer=Exp(real)
Message("Exponential of %real% is",answer)
```

See Also:

[LogE](#)

Fabs

Calculates the absolute value of a floating-point argument.

Syntax:

```
Fabs(x)
```

Parameters:

(f) *x* floating point number.

Returns:

(f) returns the absolute value of the argument.

Use this function to calculate the absolute value of a floating point argument. There is no error return.

Example:

```
a = -1.23
Message("Fabs(%a%) is", Fabs(a))
```

See Also:

<none>

FileAppend

Appends one or more files to another file.

Syntax:

```
FileAppend (source-list, destination)
```

Parameters:

- (s) source-list a string containing one or more filenames, which may be wildcarded.
- (s) destination target file name.

Returns:

- (i) **@TRUE** if all files were appended successfully.

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.

Note: In the 32-bit version of WIL, the "default" file delimiter used to delimit lists of files and directories, has been changed to a TAB. In the 16-bit version of WIL, the "default" delimiter has not changed, and remains a space. **IntControl(29,0,0,0)** allows the file delimiter to be changed to a character of your own choosing. To standardize a TAB delimiter for both 16 and 32 bit scripts add the **IntControl(29,0,0,0)** to the top of the script.

```
IntControl(29, @TAB, 0, 0, 0)
```

Examples:

```
testfile1=FileLocate("win.ini")
testfile2=FileLocate("system.ini")
testfile3="c:\winini.bak"
```

```
FileAppend(testfile1,testfile3)
FileAppend(testfile2,testfile3)
```

```
Message("FileAppend",strcat(testfile3,@crlf,"now
contains",@crlf,testfile1,@crlf,"and",@crlf,testfile2))
```

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:

<u>Char</u>	<u>Symbol</u>	<u>Meaning</u>
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 = strcat(DirWindows(0), "win.ini")
attr = FileAttrGet(editfile)
If StrSub(attr, 1, 1) == "R"
    Message("File is read-only", "Cannot edit %editfile%")
else
    Run("notepad.exe", editfile)
endif
```

See Also:

[FileAttrSet](#), [FileTimeGet](#)

FileAttrSet

Sets file attributes.

Syntax:

```
FileAttrSet (file-list, settings)
```

Parameters:

- (s) file-list a delimited list of files.
- (s) settings new attribute settings for those file(s).

Returns:

- (i) always 1.

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

File list may contain * and ? wildcards.

Note: In the 32-bit version of WIL, the "default" file delimiter used to delimit lists of files and directories, has been changed to a TAB. In the 16-bit version of WIL, the "default" delimiter has not changed, and remains a space. **IntControl(29,0,0,0)** allows the file delimiter to be changed to a character of your own choosing. To standardize a TAB delimiter for both 16 and 32 bit scripts add the **IntControl(29,0,0,0)** to the top of the script.

```
IntControl(29, @TAB, 0, 0, 0)
```

Examples:

```
testfile1=strcat(DirWindows(0), "win.ini")
FileAttrSet(testfile1, "rAsh")
Message("FileAttrSet", "File Attributes Set")
```

See Also:

[FileAttrGet](#), [FileTimeTouch](#)

FileClose

Closes a file.

Syntax:

FileClose (filehandle)

Parameters:

(i) filehandle same integer that was returned by **FileOpen**.

Returns:

(i) always 0.

Example:

```
; the hard way to copy an ASCII file
testfile1=FileLocate("win.ini")
testfile2=strcat(FilePath(testfile1),"winini.bak")
old = FileOpen(testfile1, "READ")
new = FileOpen(testfile2, "WRITE")
while @TRUE                    ; Loop till break do us end
    x = FileRead(old)
    If x == "*EOF*" Then Break
    FileWrite(new, x)
endwhile
FileClose(new)
FileClose(old)
Message("FileClose",strcat(testfile1,@crlf,"backed up
to",@crlf,testfile2))
```

See Also:

[FileOpen](#), [FileRead](#), [FileWrite](#)

FileCompare

Compares two files and reports on the result.

Syntax:

```
FileCompare(filename1, filename2)
```

Parameters:

(s) filename1 name of first file to compare
(s) filename2 name of second file to compare

Returns:

- (i) compare result. Possible values are:
- 0 Files contents are identical.
 - 1 Files are same size but different - first file is newer.
 - 1 Files are same size but different - second file is newer.
 - 2 Files are different - first file is newer.
 - 2 Files are different - second file is newer.
 - 3 Second file missing - only first file exists.
 - 3 First file missing - only second file exists.
 - 4 Neither file exists.

Use this function to compare two files to determine if they are identical or not. If the return value is zero, the file contents are identical. If the return value is not zero, the actual value provides additional information on why they didnt compare. An actual byte by byte compare is performed only if the file sizes are identical, otherwise it is obvious that the files must be different.

Example:

```
;Assuming a copy of win.ini has been previously made to win.sav  
a=FileCompare("WIN.INI", "WIN.SAV")  
if a==0  
    Message("Info", "WIN.INI not modified")  
else  
    Message("Alert!", "WIN.INI has been modified")  
endif
```

See Also:

[ExeTypeInfo](#), [FileCopy](#), [FileMove](#), [FileDelete](#)

FileCopy

Copies files.

Syntax:

FileCopy (source-list, destination, warning)

Parameters:

- (s) source-list a string containing one or more filenames, which may be wildcarded.
- (s) destination target file name.
- (i) warning **@TRUE** if you want a warning before overwriting existing files;
@FALSE if no warning desired.

Returns:

- (i) **@TRUE** if all files were copied successfully (see below).

Note: If warning parameter is set to **@TRUE**, then the return value will be **@TRUE**, even if it was modified by the user dialog.

Use this function to copy an individual file, a group of files using wildcards, or several groups of files by separating the names with a delimiter.

You can also copy files to any **COM** or **LPT** device, but **do not** place a colon after the name of the device.

Source-list and destination may contain * and ? wildcards.

Note: In the 32-bit version of WIL, the "default" file delimiter used to delimit lists of files and directories, has been changed to a TAB. In the 16-bit version of WIL, the "default" delimiter has not changed, and remains a space. **IntControl(29,0,0,0)** allows the file delimiter to be changed to a character of your own choosing. To standardize a TAB delimiter for both 16 and 32 bit scripts add the **IntControl(29,0,0,0)** to the top of the script.

```
IntControl(29, @TAB, 0, 0, 0)
```

Examples:

```
temp=Environment("TEMP")
DirChange(DirWindows(0))
FileCopy("*.ini", temp, @TRUE)
Message("FileCopy","Windows ini files backed up to %temp%")
```

```
;Print file to LPT1 example
;file to print
myfile="C:\WINDOWS\desktop\myfile.txt"
;Create a file with a formfeed to send to printer
home_dir=DirHome()
formfeed_file="%home_dir%print.txt"
file_handle=FileOpen(formfeed_file,"WRITE")
;inserts a form feed into the file
FileWrite(file_handle, Num2Char(12))
FileClose(file_handle)
;execute file copys to LPT1
FileCopy(myfile,"LPT1",@false)
FileCopy(formfeed_file,"LPT1",@false)
;cleanup
FileDelete(formfeed_file)
message("Print Job", "Complete")
exit
```

See Also:

[FileDelete](#), [FileExist](#), [FileLocate](#), [FileMove](#), [FileRename](#)

FileCopyAttr

Copies files, and sets file attributes..

Syntax:

FileCopyAttr (source-list, destination, warning, attributes)

Parameters:

- (s) source-list a string containing one or more filenames, which may be wildcarded.
- (s) destination target file name.
- (i) warning **@TRUE** if you want a warning before overwriting existing files;
 @FALSE if no warning desired.
- (s) attributes attributes to be modified

Returns:

- (i) **@TRUE** if all files were copied successfully (see below).

Note: If warning parameter is set to **@TRUE** , then the return value will be **@TRUE**, even if it was modified by the user dialog.

Use this function to copy an individual file, a group of files using wildcards, or several groups of files by separating the names with a delimiter.

You can also copy files to any **COM** or **LPT** device, but **do not** place a colon after the name of the device.

Source-list and destination may contain * and ? wildcards.

attribute: 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

Note: In the 32-bit version of WIL, the "default" file delimiter used to delimit lists of files and directories, has been changed to a TAB. In the 16-bit version of WIL, the "default" delimiter has not changed, and remains a space. **IntControl(29,0,0,0)** allows the file delimiter to be changed to a character of your own choosing. To standardize a TAB delimiter for both 16 and 32 bit scripts add the **IntControl(29,0,0,0)** to the top of the script.

```
IntControl(29, @TAB, 0, 0, 0)
```

Examples:

```
temp=Environment("TEMP")
DirChange(DirWindows(0))
FileCopy("*.ini", temp, @TRUE, "R")
Message("FileCopy", "files backed up with Read-Only
attribute set")
```

See Also:

[FileCopy](#), [FileMoveAttr](#)

FileDelete

Deletes files.

Syntax:

FileDelete (file-list)

Parameters:

(s) file-list a string containing one or more filenames, which may be wildcarded.

Returns:

(i) **@TRUE** if all the files were deleted.

File-list may contain * and ? wildcards.

Use this function to delete an individual file, a group of files using wildcards, or several groups of files by separating the names with a delimiter.

Note: In the 32-bit version of WIL, the "default" file delimiter used to delimit lists of files and directories, has been changed to a TAB. In the 16-bit version of WIL, the "default" delimiter has not changed, and remains a space. **IntControl(29,0,0,0)** allows the file delimiter to be changed to a character of your own choosing. To standardize a TAB delimiter for both 16 and 32 bit scripts add the **IntControl(29,0,0,0)** to the top of the script.

```
IntControl(29, @TAB, 0, 0, 0)
```

Example:

```
temp=Environment("TEMP")
DirChange(temp)
testfile1="test1.bak"
testfile2="temp123.fil"
handle = fileopen(testfile1, "write")
filewrite(handle, "dummy data one")
fileclose(handle)
handle = fileopen(testfile2, "write")
filewrite(handle, "dummy data two")
fileclose(handle)

Pause("FileDelete","OK to delete *.bak and temp123.fil files from %temp
%")
FileDelete("*.bak|temp123.fil")
```

See Also:

[FileExist](#), [FileLocate](#), [FileMove](#), [FileRename](#)

FileExist

Tests for the existence of files.

Syntax:

FileExist (filename)

Parameters:

(s) filename either a fully qualified filename with drive and path,
or just a filename and extension.

Returns:

(i) **@TRUE** if the file exists;
@FALSE if it doesn't exist or if the pathname is invalid.
2 - if the specified file exists but is currently
open by another application in read deny mode.

This function is used to test whether or not a specific 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.

FileExist returns "2" if the specified file exists but is currently open by another application in read deny mode. If you try to access this file using (most of) the other "File..." functions, it will cause a sharing violation.

FileExist does not support wild-carded file names. Here is a workaround.

```
if FileItemize("c:\somedir\*.*)"=="  
    message("", "No files were found")  
else  
    message("", "Files were found")  
  
endif
```

Examples:

```
; 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:

[DirExist](#), [FileLocate](#)

FileExtension

Returns the extension of a file.

Syntax:

FileExtension (filename)

Parameters:

(s) filename [optional path]full file name, including extension.

Returns:

(s) file extension.

This function parses the passed filename and returns the extension part of the filename.

Note: The extension must be in uppercase.

Example:

```
; prevent the user from editing a COM or EXE file
IntControl(29, @TAB, 0, 0, 0)
allfiles = FileItemize("*.*)
editfile = AskItemList("Select file to edit", allfiles, @TAB,
@unsorted,
single)
ext = FileExtension(editfile)
If (ext == "COM") || (ext == "EXE")
    Message ("Sorry", "You may not edit a program file")
else
    Run("notepad.exe", editfile)
endif
```

See Also:

[Dialog](#), [FilePath](#), [FileRoot](#)

FileFullName

Fleashes out a file name with drive and path information .

Syntax:

```
FileFullName(partial filename)
```

Parameters:

(s) partial filename possibly incomplete filename - missing drive and/or path.

Returns:

(s) a complete file name.

Use this function to return the complete file name from a partial file name. Drive and path information will be added to the file name to create a full file name. If both drive and path are missing, the currently logged drive and path will be used. If only drive is missing, the currently logged drive will be used. If drive is specified without a path, then the currently logged directory on that drive will be used.

Example:

```
DirChange("C:\TEMP")
a="Test.abc"
b=FileFullName(a)
Message(a,b)
; b will equal C:\TEMP\TEST.ABC
```

See Also:

[AskFileName](#), [ExeTypeInfo](#), [FileMapName](#), [FileLocate](#)

FileItemize

Returns a delimited list of files.

Syntax:

FileItemize (file-list)

Parameters:

(s) file-list a string containing a list of filenames, which may be wildcarded.

Returns:

(s) a 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 **AskItemList** function, which lets the user choose an item from such a delimited list.

Note: Some shell or file manager applications using the WIL Interpreter allow an empty string ("") to be used as the "file-list" parameter, in which case all files highlighted in the file display are returned. However, if there are any file names or wildcards in the string, all files matching the file names are returned, regardless of which ones are highlighted.

Note: In the 32-bit version of WIL, the "default" file delimiter used to delimit lists of files and directories, has been changed to a TAB. In the 16-bit version of WIL, the "default" delimiter has not changed, and remains a space. **IntControl(29,0,0,0)** allows the file delimiter to be changed to a character of your own choosing. To standardize a TAB delimiter for both 16 and 32 bit scripts add the **IntControl(29,0,0,0)** to the top of the script.

```
IntControl(29, @TAB, 0, 0, 0)
```

Examples:

```
IntControl(29, @TAB, 0, 0, 0)
a = FileItemize("*.bak")                    ;all BAK files
b = FileItemize("*.arc|*.zip|*.lzh")        ;compressed files

; Get which .INI file to edit
ifiles = FileItemize("c:\windows\*.ini")
ifile = AskItemList(".INI Files", ifiles, @TAB, @unsorted, @single)
RunZoom("notepad", ifile)
Drop(ifiles, ifile)
```

See Also:

[CurrentFile](#), [DirItemize](#), [AskItemList](#), [AskFileText](#), [WinItemize](#)

FileLocate

Finds file in current directory or along the DOS path.

Syntax:

FileLocate (filename)

Parameters:

(s) filename full file name, including extension.

Returns:

(s) 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 == ""
    Message("???", "WIN.INI not found")
else
    Run("notepad.exe", winini)
endif
```

See Also:

[FileExist](#)

FileMapName

Transforms a filename with a file wildcard mask and returns a new filename.

Syntax:

FileMapName(filename, mapping-data).

Parameters:

- (s) filename full or partial file name.
- (s) mapping data mapping and wildcard definition string (see below).

Returns:

- (s) transformed file name.

Use this function to generate a new filename based on an old filename. It can be used to generate *.bak filenames easily, or to perform assorted wildcard transformations on a filename. The mapping-data consists of the normal (optional) drive, path, legal filename characters, the period, and two special wildcard characters, the asterisk (*) and the question mark (?). The following algorithm is used to transform the file name:

- 1) If drive is specified in the mapping-data use specified drive, else use current drive.
- 2) If path is specified in the mapping-data use specified path, else use current path on the drive selected above.
- 3) Examine root of the filename and root position of mapping-data, sequencing through the root characters together, one character at a time.

map-char root transformation rule

- . If the mapping position character is a period, stop processing the root filename, add a period to the end of the new filename string and proceed to process the extension as outlined below.
- * If mapping data position is a asterisk, copy remainder of root file name to new filename string and proceed to process the extension as outlined below.
- ? If mapping data position is a question mark, copy the current character from the root filename to the new filename string.
- other If the mapping data character is not one of the above, copy the map character to the new filename string and ignore the corresponding character in the root filename.
- <none> If there are no more characters in the mapping-data string, filename generation is complete. Return with the new string.

- 4) Examine extension of the filename and extension position of mapping-data, sequencing through the extension characters together, one character at a time.

map-char extension transformation rule

- * If mapping data position is an asterisk, copy remainder of extension file name to new filename string and return.
- ? If mapping data position is a question mark, copy the current character from the extension filename to the new filename string.
- other If the mapping data character is not one of the above, copy the map character to the new filename string and ignore the corresponding character in the extension filename.
- <none> If there are no more characters in the mapping-data string, filename generation is complete. Return with the new string.

Example:

```
DirChange("C:\TEMP")
a=FileMapName("d:\sample\xxx.txt", "*.bak")
Message("New filename", a)
; This will return C:\TEMP\XXX.BAK
;
a=FileMapName("d:\sample\xxx.txt", "c:\demo\??Q.bak")
Message("New filename", a)
; This will return C:\DEMO\XXQ.BAK
```

See Also:

[AskFileName](#), [FileFullName](#), [FileCopy](#), [FileMove](#)

FileMove

Moves files.

Syntax:

FileMove (source-list, destination, warning)

Parameters:

- (s) source-list one or more filenames separated by a delimiter.
- (s) destination target filename.
- (i) warning **@TRUE** if you want a warning before overwriting existing files;
 @FALSE if no warning desired.

Returns:

- (i) **@TRUE** if the file was moved. (see note).

Note: If warning parameter is set to **@TRUE** , then the return value will be **@TRUE**, even if it was modified by the user dialog.

Use this function to move an individual file, a group of files using wildcards, or several groups of files by separating the names with a delimiter.

You can move files to another drive. You can also move a file to a **COM** or **LPT** port, which would cause the file to be copied to the port and then deleted (**do not** put a colon after the name of the port).

Source-list and destination may contain * and ? wildcards.

Note: In the 32-bit version of WIL, the "default" file delimiter used to delimit lists of files and directories, has been changed to a TAB. In the 16-bit version of WIL, the "default" delimiter has not changed, and remains a space. **IntControl(29,0,0,0)** allows the file delimiter to be changed to a character of your own choosing. To standardize a TAB delimiter for both 16 and 32 bit scripts add the **IntControl(29,0,0,0)** to the top of the script.

```
IntControl(29, @TAB, 0, 0, 0)
```

Examples:

```
temp=Environment("TEMP")
DirChange(temp)
testfile1="test1.txt"
testfile2="test2.txt"
handle = fileopen(testfile1, "write")
filewrite(handle, "dummy data one")
fileclose(handle)

FileMove(testfile1, testfile2, @FALSE)
Message("FileMove", "File Moved")
```

See Also:

[FileCopy](#), [FileDelete](#), [FileExist](#), [FileLocate](#), [FileRename](#)

FileMoveAttr

Moves files, and sets file attributes..

Syntax:

FileMoveAttr (source-list, destination, warning, attributes)

Parameters:

- | | |
|-----------------|---|
| (s) source-list | one or more filenames separated by a delimiter. |
| (s) destination | target filename. |
| (i) warning | @TRUE if you want a warning before overwriting existing files;
@FALSE if no warning desired. |
| (s) attributes | attributes to be modified. |

Returns:

- | | |
|-----|---|
| (i) | @TRUE if the file was moved. (see note). |
|-----|---|

Note: If warning parameter is set to **@TRUE** , then the return value will be **@TRUE**, even if it was modified by the user dialog.

Use this function to move an individual file, a group of files using wildcards, or several groups of files by separating the names with a delimiter.

You can move files to another drive. You can also move a file to a **COM** or **LPT** port, which would cause the file to be copied to the port and then deleted (**do not** put a colon after the name of the port).

Source-list and destination may contain * and ? wildcards.

attribute: 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

Note: In the 32-bit version of WIL, the "default" file delimiter used to delimit lists of files and directories, has been changed to a TAB. In the 16-bit version of WIL, the "default" delimiter has not changed, and remains a space. **IntControl(29,0,0,0)** allows the file delimiter to be changed to a character of your own choosing. To standardize a TAB delimiter for both 16 and 32 bit scripts add the **IntControl(29,0,0,0)** to the top of the script.

```
IntControl(29, @TAB, 0, 0, 0)
```

Examples:

```
temp=Environment("TEMP")
DirChange(temp)
testfile1="test1.txt"
testfile2="test2.txt"
handle = fopen(testfile1, "write")
fwrite(handle, "dummy data one")
fclose(handle)

FileMoveAttr(testfile1, testfile2, @FALSE, "R")
Message("FileMove", "File Moved with Read Only attribute set")
```

See Also:

[FileCopy](#), [FileCopyAttr](#), [FileMove](#)

FileNameLong {*32}

Returns the long version of a filename.

Syntax:

```
FileNameLong(filename)
```

Parameters:

(s) filename fully qualified file name, path optional.

Returns:

(s) the long version of a filename.

FileNameLong searches the path for the filename specified, returning the long filename if found.

Example:

```
DirChange("C:\windows")
a=FileNameLong("carved~1.bmp")
message("Long Filename", a)
```

See Also:

[FileFullName](#), [FileNameShort](#)

FileNameShort {*32}

Returns the short (ie, 8.3) version of a filename.

Syntax:

```
FileNameShort(filename)
```

Parameters:

(s) filename fully qualified file name, path optional.

Returns:

(s) the short version of a filename.

FileNameShort searches the path for the filename specified, returning the short filename if found.

Example:

```
DirChange("C:\windows")
a=FileNameShort("carved stone.bmp")
message("Short Filename", a)
```

See Also:

[FileFullName](#), [FileNameLong](#)

FileOpen

Opens a STANDARD ASCII / ANSI (only) file for reading, writing or appending.

Syntax:

```
FileOpen (filename, mode)
```

Parameters:

(s) filename	name of the file to open.
(s) mode	"READ", "WRITE". or "APPEND"

Returns:

(i)	filehandle.
-----	-------------

The **filehandle** returned by the **FileOpen** function may be subsequently used by the **FileRead**, **FileWrite**, and **FileClose** functions. If the file cannot be opened as requested, and errors are suppressed with **Errormode** then **FileOpen** returns a filehandle of 0.

You may have a maximum of five files open at one time.

FileOpen on an existing file in "WRITE" mode, will overwrite (zero out) the file. If you specify a non-existing file in "WRITE" or "APPEND" mode, it will create the file.

FileOpen by default expects each line to be terminated with the DOS standard CRLF combination. Using `IntControl 53` you can change the expected line termination character. The file must be some type of standard text file. The binary operation maybe used for non text files.

Examples:

```
; To open for reading:
handle = FileOpen("stuff.txt", "READ")

; To open for writing:
handle = FileOpen("stuff.txt", "WRITE")

; To open for appending:
handle = FileOpen("stuff.txt", "APPEND")

Temp = environment("temp")
if strsub(temp, strlen(temp), 1) != "\" then temp=strcat(temp, "\"")
testfile1=strcat(temp, "test1.txt")
handle=fileopen(testfile1, "write")
filewrite(handle, "dummy data one")
fileclose(handle)
Message("FileOpen", "File %testfile1% created")
```

See Also:

[Binary Operations](#), [BinaryRead](#), [BinaryWrite](#), [FileClose](#), [FileRead](#), [FileWrite](#)

FilePath

Returns the path of a file.

Syntax:

```
FilePath (filename)
```

Parameters:

(s) filename fully qualified file name, including path.

Returns:

(s) 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", compath)
```

See Also:

[CurrentPath](#), [FileExtension](#), [FileRoot](#)

FileRead

Reads data from a file.

Syntax:

FileRead (filehandle)

Parameters:

(i) filehandle same integer that was returned by **FileOpen**.

Returns:

(s) line of data read from file.

When the end of the file is reached, the string ***EOF*** will be returned.

Note: This function expects standard DOS CR-LF terminated lines in the file. It will read the next unread line from the file and return that line without the CR-LF on the end.

Example:

```
; the hard way to copy an ASCII file
testfile1=FileLocate("win.ini")
testfile2=strcat(FilePath(testfile1),"winini.bak")
old = FileOpen(testfile1, "READ")
new = FileOpen(testfile2, "WRITE")
while @TRUE                    ; Loop till break do us end
    x = FileRead(old)
    If x == "*EOF*" Then Break
    FileWrite(new, x)
endwhile
FileClose(new)
FileClose(old)
Message("FileRead",strcat(testfile1,@crlf,"backed up
to",@crlf,testfile2))
```

See Also:

[FileClose](#), [FileOpen](#), [FileWrite](#)

FileRename

Renames files.

Syntax:

```
FileRename (source-list, destination)
```

Parameters:

- (s) source-list one or more filenames, separated by a delimiter.
- (s) destination target filename.

Returns:

- (i) **@TRUE** if the file was renamed.

Use this function to rename an individual file, a group of files using wildcards, or several groups of files by separating the names with a delimiter.

Note: Unlike **FileMove**, you cannot make a file change its resident disk drive with **FileRename**.

Source-list and **destination** may contain * and ? wildcards.

Note: In the 32-bit version of WIL, the "default" file delimiter used to delimit lists of files and directories, has been changed to a TAB. In the 16-bit version of WIL, the "default" delimiter has not changed, and remains a space. **IntControl(29,0,0,0)** allows the file delimiter to be changed to a character of your own choosing. To standardize a TAB delimiter for both 16 and 32 bit scripts add the **IntControl(29,0,0,0)** to the top of the script.

```
IntControl(29, @TAB, 0, 0, 0)
```

Examples:

```
FileRename("c:\config.sys", "c:\config.old")
```

```
FileRename("c:\temp\*.txt", "c:\temp\*.bak")
```

See Also:

[FileCopy](#), [FileExist](#), [FileLocate](#), [FileMove](#)

FileRoot

Returns root of file.

Syntax:

FileRoot (filename)

Parameters:

(s) filename [optional path] full file name, including extension.

Returns:

(s) file root.

FileRoot parses the passed filename and returns the root part of the filename.

Example:

```
IntControl(29, @TAB, 0, 0, 0)
allfiles = FileItemize("*.*.*)
editfile = AskItemList("Select file to edit", allfiles, @TAB, @unsorted,
single)
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:

(s) file-list zero or more filenames, separated by a delimiter.

Returns:

(i) total bytes taken up by the specified file(s).

This function returns the total size of the specified files. **File-list may** contain * and ? wildcards.

Note: In the 32-bit version of WIL, the "default" file delimiter used to delimit lists of files and directories, has been changed to a TAB. In the 16-bit version of WIL, the "default" delimiter has not changed, and remains a space. **IntControl(29,0,0,0)** allows the file delimiter to be changed to a character of your own choosing. To standardize a TAB delimiter for both 16 and 32 bit scripts add the **IntControl(29,0,0,0)** to the top of the script.

```
IntControl(29, @TAB, 0, 0, 0)
```

Example:

```
size = FileSize("*. *")
Message("Size of All Files in Directory", size)
```

See Also:

[DiskFree](#)

FileTimeCode

Returns a machine readable/computable code for a file time.

Syntax:

```
FileTimeCode(filename)
```

Parameters:

(s) filename file name to get the time code from.

Returns:

(i) file time code.

Use this function to return an 32 bit integer representing the current file time stamp. This number may be compared to other file times to compare ages of files. It is basically the DOS 16 bit date and the DOS 16 bit time in a 32 bit integer. This function returns a valid, comparable time through the year 2044.

Example:

```
a=FileTimeCode("C:\AUTOEXEC.BAT")
b=FileTimeCode("C:\CONFIG.SYS")
if a == b
    ans="Same Time"
else
    if a > b
        ans = "AutoExec newer than Config"
    else
        ans = "AutoExec older than Config"
    endif
endif
Message("Comparing file times", ans)
```

See Also:

[FileTimeGet](#), [FileYmdHms](#)

FileTimeGet

Returns file date and time in a human readable format.

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 of the string depends on the current settings in the [Intl] section of the WIN.INI file or depending on what is set in the user's current International/Regional settings in the Control panel.

mm/dd/yy hh:mmXX
dd/mm/yy hh:mmXX
yy/mm/dd hh:mmXX

Where:

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
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** icon in **Control Panel** if the format isn't what you prefer.

Note: If you must parse the time data returned by this function, use the **ParseData** function to break the day, date, and time into separate components. However you should check the **FileYmdHms** and **FileTimeCode** functions first

Example:

```
oldtime = FileTimeGet("C:\windows\win.ini")
RunWait("notepad.exe", "win.ini")
newtime = FileTimeGet("C:\windows\win.ini")
If StrCmp(oldtime, newtime) == 0
    Message("WIN.INI not changed", "Last change was %oldtime%")
else
    Message("WIN.INI modified", "New time stamp is %newtime%")
endif
```

See Also:

[FileTimeCode](#), [TimeDate](#), [FileAttrGet](#), [FileTimeTouch](#), [FileTimeSet](#), [FileYmdHms](#)

FileTimeGetEx{*32}

Gets extended time information for a file or directory.

Syntax:

```
FileTimeGetEx(filename, time-field)
```

Parameters:

(s) filename can specify a file or directory name.
(i) time-field 1=file created , 2=file last modified, 3=file last accessed

Returns:

(s)datetime Returns file time in YmdHms format:
(YYYY:MM:DD:HH:MM:SS.)

This function is like FileYmdHms, but works with directories as well as files, and lets you specify which time field you want to get.

"Time-field" can be one of the following:

- 1 file created
- 2 file last modified
- 3 file last accessed

Example:

```
oldtime = FileTimeGetEx("C:\windows\win.ini",2)
RunWait("notepad.exe", "win.ini")
newtime = FileTimeGetEx("C:\windows\win.ini",2)
If StrCmp(oldtime, newtime) == 0
    Message("WIN.INI not changed", "Last change was %oldtime%")
else
    Message("WIN.INI modified", "New time stamp is %newtime%")
endif
```

See Also:

[FileTimeCode](#), [TimeDate](#), [FileAttrGet](#), [FileTimeTouch](#), [FileTimeSet](#), [FileYmdHms](#)

FileTimeSet

Sets the date and time of one or more files.

Syntax:

```
FileTimeSet(list, ymdhms)
```

Parameters:

(s) list filename, list of files, or list of wildcards of files reset.
(s) ymdhms date time in the YmdHms format

Returns:

(i) datetime **@TRUE** All files specified were time stamped.

Use this function to reset the date and time of a specific file or list of files.

Note: In the 32-bit version of WIL, the "default" file delimiter used to delimit lists of files and directories, has been changed to a TAB. In the 16-bit version of WIL, the "default" delimiter has not changed, and remains a space. **IntControl(29,0,0,0)** allows the file delimiter to be changed to a character of your own choosing. To standardize a TAB delimiter for both 16 and 32 bit scripts add the **IntControl(29,0,0,0)** to the top of the script.

```
IntControl(29, @TAB, 0, 0, 0)
```

Example:

```
; Alter time of the WIN.INI file
winini= FileLocate("Win.ini")
b=FileYmdhms(winini)
Message("File Time for %winini% is", b)
a="94:02:14:09:38:26"
;
FileTimeSet(winini, a)
b=FileYmdhms(winini)
Message("File Time for %winini% is now", b)
;
;Alter the time of all files in the temp directory
a=TimeYmdHms( )
Temp = environment("temp")
if strstr(temp,strlen(temp),1)!="\" then temp=strcat(temp, "\"
FileTimeSet("%temp%*.\"", a)
```

See Also:

[GetExactTime](#), [FileYmdHms](#), [TimeDiffDays](#), [TimeDiffSecs](#), [TimeYmdHms](#)

FileTimeSetEx{*32}

Sets extended time information for one or more files.

Syntax:

```
FileTimeSetEx(file-list, YmdHms, time-field)
```

Parameters:

(s) list	filename, list of files, or list of wildcards of files reset.
(s) ymdhms	date time in the YmdHms format
(i) time-field	1=file created , 2=file last modified, 3=file last accessed

Returns:

(i) datetime **@TRUE** All files specified were time stamped.

Use this function to reset the date and time of a specific file or list of files.

This function is like FileTimeSet, but lets you specify which time field you want to set. "Time-field" can be one of the following:

- 1 file created
- 2 file last modified
- 3 file last accessed

Note: In the 32-bit version of WIL, the "default" file delimiter used to delimit lists of files and directories, has been changed to a TAB. In the 16-bit version of WIL, the "default" delimiter has not changed, and remains a space. **IntControl(29,0,0,0)** allows the file delimiter to be changed to a character of your own choosing. To standardize a TAB delimiter for both 16 and 32 bit scripts add the **IntControl(29,0,0,0)** to the top of the script.

```
IntControl(29, @TAB, 0, 0, 0)
```

Example:

```
; Alter time of the WIN.INI file
winini= FileLocate("Win.ini")
b=FileYmdhms(winini)
Message("File Time for %winini% is", b)
a="94:02:14:09:38:26"
;
FileTimeSetEx(winini, a, 2)
b=FileYmdhms(winini)
Message("File Time for %winini% is now", b)
;
;Alter the time of all files in the temp directory
a=TimeYmdHms( )
Temp = environment("temp")
if strsub(temp,strlen(temp),1)!="\" then temp=strcat(temp,"\\")
FileTimeSetEx("%temp%*.\"", a, 2)
```

See Also:

[GetExactTime](#), [FileYmdHms](#), [TimeDiffDays](#), [TimeDiffSecs](#), [FileTimeSet](#), [TimeYmdHms](#)

FileTimeTouch

Sets file(s) to current date and time.

Syntax:

```
FileTimeTouch (file-list)
```

Parameters:

(s) file-list a delimited list of files

Returns:

(i) always 1.

File-list is a delimited list of files, which may 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**.

Note: In the 32-bit version of WIL, the "default" file delimiter used to delimit lists of files and directories, has been changed to a TAB. In the 16-bit version of WIL, the "default" delimiter has not changed, and remains a space. **IntControl(29,0,0,0)** allows the file delimiter to be changed to a character of your own choosing. To standardize a TAB delimiter for both 16 and 32 bit scripts add the **IntControl(29,0,0,0)** to the top of the script.

```
IntControl(29, @TAB, 0, 0, 0)
```

Example:

```
;Alter the time of all files in the temp directory

Temp = environment("temp")
if strsub(temp, strlen(temp), 1) != "\" then temp=strcat(temp, "\"")
FileTimeTouch("%temp%*.**")
a=TimeYmdHms( )
Message("FileTimeTouch", "All files in %temp% set to %a%")
```

See Also:

[FileAttrSet](#), [FileTimeGet](#)

FileVerInfo

Returns a version resource string from a binary file.

Syntax:

```
FileVerInfo(filename, language-key, resource-string)
```

Parameters:

- (s) filename name of the EXE, DLL, or other binary file containing version resource information.
- (s) language-key a hexadecimal string identifying the desired language and character-set block within the file's resource table.
- (s) resource-string specifies the item to return. (see below.)

Returns:

- (s) version resource string from binary file or a blank string ("") if the specified item was not found.

"language-key" is a hexadecimal string identifying the desired language and character-set block within the file's resource table. U.S. English (w/ Windows Multilingual character set) is "040904E4". You can specify a blank string ("") to indicate U.S. English, which will almost always be appropriate. If a blank string is specified, it will first try to look up the item under the language key "040904E4" (U.S. English w/ Windows Multilingual character set). If that fails, it will then try the key "040904B0" (U.S. English w/ Unicode character set).

"resource-string" specifies the item to return. The following are standard item names (although some of these are optional and will not necessarily be present in any particular file):

"Comments"	"LegalCopyright"
"CompanyName"	"LegalTrademarks"
"FileDescription"	"OriginalFilename"
"FileVersion"	"ProductName"
"InternalName"	"ProductVersion"

Note: The 16-bit version can only access 16-bit files, and the 32-bit version can only access 32-bit files.

Example:

```

; Version Information Example WinBatch Script

FileN = FileLocate("explorer.exe")

ErrorMode(@OFF)
FComments = FileVerInfo(FileN, "", "Comments")
FCompany = FileVerInfo(FileN, "", "CompanyName")
FDescription = FileVerInfo(FileN, "", "FileDescription")
FVersion = FileVerInfo(FileN, "", "FileVersion")
FInternalName = FileVerInfo(FileN, "", "InternalName")
FLegalCopyright = FileVerInfo(FileN, "", "LegalCopywrite")
FOriginalFilename = FileVerInfo(FileN, "", "OriginalFilename")

FProductName = FileVerInfo(FileN, "", "ProductName")
FProductVersion = FileVerInfo(FileN, "", "ProductVersion")
ErrorMode(@Cancel)

MyDialogFormat=`WWDDLGED,5.0`

MyDialogCaption=`WIL Dialog`
MyDialogX=4
MyDialogY=21
MyDialogWidth=241
MyDialogHeight=147
MyDialogNumControls=19

MyDialog01=`70,2,168,DEFAULT,VARYTEXT,FComments,"" `
MyDialog02=`2,2,66,DEFAULT,STATICTEXT,DEFAULT,"Comments:"`
MyDialog03=`2,16,64,DEFAULT,STATICTEXT,DEFAULT,"Company:"`
MyDialog04=`70,16,168,DEFAULT,VARYTEXT,FCompany,"" `

MyDialog05=`2,30,64,DEFAULT,STATICTEXT,DEFAULT,"Description:"`
MyDialog06=`70,30,168,DEFAULT,VARYTEXT,FDescription,"" `
MyDialog07=`2,44,64,DEFAULT,STATICTEXT,DEFAULT,"Version:"`
MyDialog08=`70,44,168,DEFAULT,VARYTEXT,FVersion,"" `
MyDialog09=`2,58,64,DEFAULT,STATICTEXT,DEFAULT,"Internal Name:"`
MyDialog10=`70,58,168,DEFAULT,VARYTEXT,FInternalName,"" `
MyDialog11=`2,72,64,DEFAULT,STATICTEXT,DEFAULT,"Legal Copyright:"`
MyDialog12=`70,72,168,DEFAULT,VARYTEXT,FLegalCopyright,"" `

MyDialog13=`2,86,64,DEFAULT,STATICTEXT,DEFAULT,"Original Filename:"`
MyDialog14=`70,86,168,DEFAULT,VARYTEXT,FOriginalFilename,"" `
MyDialog15=`2,100,64,DEFAULT,STATICTEXT,DEFAULT,"Product Name:"`
MyDialog16=`70,100,168,DEFAULT,VARYTEXT,FProductName,"" `
MyDialog17=`2,114,64,DEFAULT,STATICTEXT,DEFAULT,"Product Version:"`
MyDialog18=`70,114,168,DEFAULT,VARYTEXT,FProductVersion,"" `
MyDialog19=`86,130,64,DEFAULT,PUSHBUTTON,DEFAULT,"Ok",1`

ButtonPushed=Dialog("MyDialog")

```

See Also:

[DOSVersion](#), [Environment](#), [Version](#), [VersionDLL](#), [WinVersion](#)

FileWrite

Writes data to a file.

Syntax:

```
FileWrite (filehandle, output-data)
```

Parameters:

- (i) filehandle same integer that was returned by **FileOpen**.
- (s) output-data data to write to file.

Returns:

- (i) always 0.

Note: This function will append a CR-LF line terminator to the end of each line written to the file.

Example:

```
handle = FileOpen("stuff.txt", "WRITE")
FileWrite(handle, "Gobbledygook")
FileClose(handle)
Message("FileWrite", "File Written.")
```

See Also:

[FileClose](#), [FileOpen](#), [FileRead](#)

FileYmdHms

Returns a file time in the YmdHms date time format.

Syntax:

```
FileYmdHms(filename)
```

Parameters:

(s) filename filename.

Returns:

(s) file time in YmdHms format.

Use this function to retrieve a file time in the YmdHms format.

Example:

```
b=FileYMDHMS("C:\CONFIG.SYS")  
Message("File Time is", b)
```

See Also:

[FileTimeGet](#), [TimeDate](#), [TimeAdd](#), [TimeDiffDays](#), [TimeYmdHms](#), [TimeDelay](#),
[TimeWait](#)

FindWindow

Returns a window title matching a specified class name.

Syntax:

```
FindWindow(window class name)
```

Parameters:

(s) window class name the full unique window class name.

Returns:

(s) a window title.

FindWindow returns the window title of the first window whose class name matches the specified "class-name". It can be used to manipulate windows when the title is either unknown or there are several windows with the same title. A blank string ("") is returned if no matching window is found.

Note: "Class-name" is not case sensitive but must be the full class name (ie, not a partial name).

Example:

```
run("explorer.exe",strcat("/n,/e,",DirGet( )))
while FindWindow("ExploreWClass")=="" ;wait for it to come up
  Yield
endwhile
```

See Also:

[IntControl 44](#)

Floor

Calculates the floor of a value.

Syntax:

Floor(x)

Parameters:

(f) x value **Floor** is calculated from.

Returns:

(f) a floating point number whose value represents the largest integer that is less than or equal to x.

Use this function to calculate the floor of a value.

Example:

```
; This example accepts a value from the user to calculate
; the ceiling and floor.
a=AskLine("Ceiling and Floor", "Please enter a number", "1.23")
c=Ceiling(a)
f=Floor(a)
Message("Ceiling and Floor of %a%", "Ceiling: %c%   Floor: %f%")
```

ie.

A=	Ceiling=	Floor=
25.2	26.0	25.0
25.7	26.0	25.0
24.9	25.0	24.0
-14.3	-14.0	-15.0

See Also:

[Abs](#), [Ceiling](#), [Fabs](#), [Min](#), [Max](#)

For

Controls the looping of a block of code based on an incrementing index.

Syntax:

```
For var-name = initial-value to last-value [ by increment]
```

Parameters:

- (s) var-name a variable name to be used for the loop index.
- (f) initial-value an expression used to set the initial value of the loop index.
- (f) last-value an expression that defines last value to be used in the loop. When the initial value is incremented past the last value, the loop terminates.
- (f) increment an expression that defines the amount to increment the loop index on each pass through the loop. The default is one. The increment may be negative.

Use the **For** statement to execute a block of code a fixed number of times. When the **For** statement is executed, it initializes the specified variable *var-name* to the *initial-value*. This variable is called the *loop index*. It then tests the *loop index* with the *last value*. If the *increment* is positive and the *loop index* is greater than the last value, or if the *increment* is negative and the *loop index* is less than the last value, then the loop terminates and control is passed to the statement after the **Next** statement.

Otherwise the statements below the **For** are executed until the **Next** statement is reached. When the **Next** statement is reached, control returns to the **For** statement so that the *loop index* may be incremented and the test for *last value* repeated.

Example:

```
; Compute sum of numbers between 1 and selected number
a=AskLine("Sums", "Please enter a number", 5)
f=0
For j = 1 to a
    f = f + j
Next
Message("Sum [ 1 to %a% ] is", f)

;Compute factorials
a=AskLine("Factorials", "Please enter a number", 5)
f=1
For j = a to 2 by -1
    f=f*j
Next
Message("%a% Factorial is", f)
```

See Also:

[Break](#), [Continue](#), [If](#), [Select](#), [Switch](#), [While](#)

GetExactTime

Returns current time in hundredths of a second.

Syntax:

```
GetExactTime( )
```

Parameters:

(none)

(Returns:

(s) the current time in hundredths of a second.

Use this function to obtain the current time in hundredths of seconds.

Example:

```
a=GetExactTime( )  
Message("Time is", a)
```

See Also:

[TimeDate](#), [TimeYmdHms](#), [GetTickCount](#)

GetTickCount

Returns number of clock ticks used by Windows since Windows started.

Syntax:

```
GetTickCount( )
```

Parameters:

(none)

Returns:

(s) The number of clock ticks.

Use this function to obtain the number of clock ticks since Windows started.

Example:

```
a=GetTickCount( )  
Message("Clock Ticks", a)
```

See Also:

[TimeDate](#), [GetExactTime](#), [TimeYmdHms](#)

GoSub

Transfers control to another point in a WIL program and saves the location of the next statement.

Syntax:

```
GoSub label
```

Parameters:

(s) *label* user-defined identifier

GoSub *label* causes an unconditional transfer of control to the line in the program marked *:label* where the identifier is preceded by a colon (:). The location of the next statement after the **GoSub** statement is retained, and control may be transferred back to that statement with a **Return** statement.

Example:

```
a=1
b=2
c=3
x=21
GoSub Poly
Message("Polynomial evaluates to", y)
a=3
b=4
c=6
x=45
GoSub Poly
Message("Polynomial evaluates to", y)
exit

; Polynomial Computation Subroutine here
:Poly
y = a*(x**2) + b*x + c
return
```

See Also:

[For](#), [Goto](#), [Return](#), [Switch](#), [Select](#), [While](#)

Goto

Changes the flow of control in a WIL program.

Syntax:

Goto *label*

Parameters:

(s) *label* user-defined identifier.

Goto *label* causes an unconditional branch to the line in the program marked *:label*, where the identifier is preceded by a colon (:).

Note: Program flow control structures, such as **For/Next**, **While/EndWhile**, **Switch/EndSwitch**, **If/EndIf** must not be "jumped" by a **Goto** statement. If a **Goto** is used inside of one of these structures to send the flow of control outside of the structure, or if a **Goto** is used to send the flow of control inside a structure, errors will result.

Example:

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

See Also:

[For](#), [If](#), [Switch](#), [While](#)

IconReplace

Replaces an existing icon with a new icon.

Syntax:

```
IconReplace(filename, iconfilename)
```

Parameters:

- | | |
|------------------|--|
| (s) filename | either a fully qualified filename with drive and path, or just a filename and extension. |
| (s) iconfilename | the filename of the icon. |

Returns:

- (i) **@TRUE** if successful.

Use this function to replace icons. **IconReplace** will perform surgery on an EXE file and replace the first icon in the ICO file. The icon in the ICO file must be the same size or smaller than the icon in the EXE file. It is suggested that due caution be used when using this command, keeping the following points in mind:

- 1) The EXE file might become damaged and be unable to run. This is especially true of some programs that checksum themselves to verify the EXE. **KEEP BACKUPS.**
- 2) System anti-virus tools might detect the alteration of an EXE file and complain. If this is true, then either the anti-virus program must be disabled, or another work around must be used. Some anti-virus programs allow the specification of a "trusted" program - the trusted feature may be used with due caution.
- 3) The application whose icon is being modified must not be running while its EXE file is being modified.

Example:

```
IconReplace("FILENAME.EXE", "ICONFILE.ICO")
```

See Also:

<none>

If ... Else ... Endif

If ... Then ... Else

Conditionally performs a function or series of statements.

Syntax:

Note: There are several forms of the **if** statement:

if ... endif (structured):

```
if expression
series
of
statements
endif
```

if ... else ... endif (structured):

```
if expression
series
of
statements
else
series
of
statements
endif
```

if ... then (single statement):

```
if expression then statement
```

if ... then ... else ... (single statement):

```
if expression then statement
else statement
```

Parameters:

- (s) expression a condition to be evaluated.
- (s) statement any valid WIL function or command.
- (s) *series of statements* multiple lines of valid WIL statements.

The **if** statement evaluates the expression following it. The expression must evaluate to an integer.

In the structured forms of the **if** syntax, if the expression evaluates to a non zero value (@TRUE) the series of statements after the **if** statement up to the first matching **else** or **endif** are executed, otherwise they are skipped. In the **if ... else ... endif** syntax, the series of statements after the **else** are executed if the result of evaluating the expression is zero (@FALSE).

In the single statement forms of the **if** syntax, if the expression evaluates to a non zero value (@TRUE) the statement following the **then** keyword is executed, otherwise it is skipped. In the **if ... then ... else ...** syntax, the statement following the **else** is executed if the result of evaluating the expression is zero (@FALSE).

Example:

```
; This example guesses a # between 1 and 1023.
Message("Think of a number", "Any number between 0 and 1023")
start = 0
stop = 1023
for i = 1 to 10
  guess = (start+stop+1) /2
  if AskYesNo("Hmmm", "Is your number smaller than %guess%")
    stop = guess - 1
  else
    start = guess
  endif
next
guess = (start+stop+1) /2
;
if guess==0 || guess==1023
  Message("Hmmm", "%guess% eh? Testing the limits again I assume")
else
  if guess==13
    Message("Hmmm", "%guess% seems rather unlucky to me")
  else
    a = guess mod 2
    if a==0 then Message("Hmmm", "Even I can figure %guess%")
      else Message("Hmmm", "It must be %guess%", oddly enough")
    endif
  endif
endif
```

See Also:

[For](#), [Select](#), [Switch](#), [While](#)

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.

Examples:

```
IniDelete("Desktop", "Wallpaper")
```

```
IniDelete("Quicken", @WHOLESECTION)  
Message("IniDelete", "Ini Deleted.")
```

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.

Note: A full path must be specified if the ini file is not in the Windows directory.

Example:

```
IniDeletePvt("Current Users", "Excel", "meter.ini")  
Message("IniDeletePvt"," Pvt Ini Deleted.")
```

See Also:

[IniDelete](#), [IniItemizePvt](#), [IniReadPvt](#), [IniWritePvt](#)

Iniltemize

Lists keywords or sections in WIN.INI.

Syntax:

Iniltemize (section)

Parameters:

(s) section the major heading to itemize.

Returns:

(s) list of keywords or sections.

Iniltemize will scan the specified section in WIN.INI, and return a tab-delimited list of all keyword names contained within that section. If a null string ("") is given as the section name, **Iniltemize** will return a list of all section names contained within WIN.INI. It returns the string "**(NONE)**" if the specified section does not exist, and returns a null string ("") if the section exists but is empty. Case is not significant in section names.

Examples:

```
; Returns all keywords in the [Extensions] section
keywords = Iniltemize("Extensions")
AskItemList("Keywords in [Extension] of in.ini",keywords,@tab,@sorted,
@single)
```

See Also:

[IniDelete](#), [IniltemizePvt](#), [IniRead](#), [IniWrite](#)

IniltemizePvt

Lists keywords or sections in a private INI file.

Syntax:

IniltemizePvt (section, filename)

Parameters:

(s) section the major heading to itemize.
(s) filename name of the INI file.

Returns:

(s) list of keywords or sections.

IniltemizePvt will scan the specified section in a private INI file, and return a tab-delimited list of all keyword names contained within that section. If a null string ("") is given as the section name, **IniltemizePvt** will return a list of all section names contained within the file. It returns the string "**(NONE)**" if the specified section does not exist, and returns a null string ("") if the section exists but is empty. Case is not significant in section names.

Note: A full path must be specified if the ini file is not in the Windows directory.

Example:

```
; Returns all keywords in the [Boot] section of SYSTEM.INI
keywords = IniltemizePvt("Boot", "system.ini")
Message("IniltemizePvt","Pvt Ini Itemized.")
```

See Also:

[IniDeletePvt](#), [Iniltemize](#), [IniReadPvt](#), [IniWritePvt](#)

IniRead

Reads data from the WIN.INI file.

Syntax:

IniRead (section, keyname, default)

Parameters:

(s) section	the major heading to read the data from.
(s) keyname	the name of the item to read.
(s) default	string to return if the desired item is not found.

Returns:

(s) 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:

[Environment](#), [IniDelete](#), [IniItemize](#), [IniReadPvt](#), [IniWrite](#)

IniReadPvt

Reads data from a private INI file.

Syntax:

```
IniReadPvt (section, keyname, default, filename)
```

Parameters:

(s) section	the major heading to read the data from.
(s) keyname	the name of the item to read.
(s) default	string to return if the desired item is not found.
(s) filename	name of the INI file.

Returns:

(s)	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.

Note: A full path must be specified if the ini file is not in the Windows directory.

Example:

```
a=IniReadPvt("Main", "Lang", "English", "WB.INI")  
Message("IniReadPvt",a)
```

Given the following segment from WB.INI:

```
[Main]  
Lang=French
```

The statement above would return:

```
French
```

See Also:

[Environment](#), [IniDeletePvt](#), [IniItemizePvt](#), [IniRead](#), [IniWritePvt](#)

IniWrite

Writes data to the WIN.INI file.

Syntax:

IniWrite (section, keyname, data)

Parameters:

(s) section	major heading to write the data to.
(s) keyname	name of the data item to write.
(s) data	string to write to the WIN.INI file.

Returns:

(i)	always 1.
-----	-----------

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:

[IniDelete](#), [IniItemize](#), [IniRead](#), [IniWritePvt](#)

IniWritePvt

Writes data to a private INI file.

Syntax:

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

Parameters:

(s) section	major heading to write the data to.
(s) keyname	name of the data item to write.
(s) data	string to write to the INI file.
(s) filename	name of the INI file.

Returns:

(i)	always 1.
-----	-----------

Writes a value in the "filename" .INI file.

Note: You cannot use this function to add or update any of the "Device=" entries in the [386Enh] section of SYSTEM.INI, because that section contains multiple entries with the same keyword. See [BinaryPokeStr](#) for an example on how to modify the device= lines of the SYSTEM.INI file.

Note: A full path must be specified if the ini file is not in the Windows directory.

Example:

```
IniWritePvt("Main", "Lang", "French", "MYFILE.INI")
```

This would create the following entry in MYFILE.INI:

```
[Main]
Lang=French
```

See Also:

[Binary Operations](#), [BinaryPokeStr](#), [IniDeletePvt](#), [IniItemizePvt](#), [IniReadPvt](#), [IniWrite](#)

InstallFile {*32}

Installs and uncompresses a file.

Syntax:

InstallFile(filename, targname, default-targdir, delete-old, flags)

Parameters:

- | | |
|---------------------|---|
| (s) filename | source file to be installed. (path optional, short , 8.3 version required) |
| (s) targname | the name of the target file to be created. (without path) |
| (s) default-targdir | directory where the file is to be installed. |
| (i) delete-old | @TRUE - to delete existing same name files.
@FALSE - to ignore existing same name files. |
| (i) flags | 1 - shared file
2 - force install |

Returns:

- | | |
|-----|---------------------------------|
| (s) | "result tempname", or "result " |
|-----|---------------------------------|

When installing 32-bit image files (EXE's, DLL's, etc.), this function uses the version information embedded in the files to determine whether a file being installed is newer than an existing file with the same name. When installing any other type of file, which does not contain appropriate version information, this function uses the time stamps of the respective files instead.

The return value is in the form:

"result|tempname", or
"result|"

where "result" is the value returned by the "VerInstallFile" Windows API function; and "tempname" is the name of the temporary file that was created if the file could not be installed, or blank otherwise.

"Default-targdir" is the directory where you want the file to be installed. The file will be installed to this directory, unless it is a shared file or a file with the same name already exists elsewhere.

If "Delete-old" is **@TRUE** (or non-zero), and a file with the same name as the file being installed already exists, it will be deleted, even if it is located in a directory (on the path) other than the target directory. If "delete-old" is **@FALSE**, such a file will not be deleted.

"Flags" specifies other optional flags that affect the operation of this function, combined with the OR ('|') operator. They are:

- 1 - shared file (file should be installed to a shared directory)
- 2 - force install (install file even if older than existing file)

Note: InstallFile can uncompress files compressed by the Microsoft setup compress utility. The image version can only be interpreted by a corresponding platform version,

ie. 32-bit images by a 32-bit platform.

Example:

```
InstallFile("a:\ctl3d.dl_", "ctl3d.dll", DirWindows(1), @TRUE, 1)
```

;or in the case of a long filename:

```
InstallFile(FileNameShort("a:\carved stone.bm_"),  
"carved~1.bmp", DirWindows(0), @TRUE, 0)  
DirChange(DirWindows(0))  
FileRename("carved~1.bmp", "carved stone.bmp")
```

See Also:

[FileCopy](#), [RegApp](#)

Int

Converts a floating point number or a string to an integer.

Syntax:

`Int(x)`

Parameters:

(s) *x* value to be converted.

Returns:

(i) an integer.

Use this function to convert a floating point number or a string to an integer. If the argument is a string, it is first converted to a number- if possible. If the argument is a number within integer range, it will be converted to the closest integer.

Example:

```
a=int(5.1) + int("123")
Message("Result is", a)
; a= 5+123 = 128
```

See Also:

[IsInt](#), [IsFloat](#), [IsNumber](#)

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 the various products. The first parameter of IntControl defines exactly what the function does, the other parameters are possible arguments to the function.

Refer to your product documentation for any further information on this 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, p2, p3, p4)

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-list box in it has to return a file name, or may return merely a directory name or nothing.

<u>P1</u>	<u>Meaning</u>
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.

<u>P1</u>	<u>Meaning</u>
0	System & Hidden files not used (default)
1	System & Hidden files seen and used

IntControl (12, p1, p2, 0, 0)

IntControl 12 is used to direct WIL and it's parent application (if the parent application supports this function) as to how to handle users either terminating WinBatch via the "Ctrl-Break" keystroke sequence or perhaps a menu item, or by simply exiting windows.

P1 codes: Add desired code in each group together.

Exit Windows group codes (choose one).

P1 Meaning

- 0 Pop up message box giving user a chance to either cancel bat file or continue.
- 1 Allow Windows to be exited with no warning.
- 2 Refuse any attempt to exit Windows. If P2 is not "" and not 0, display p2 in a message box. E.G. `IntControl(12,2,"Attention! Close all apps first",0,0)`
- 3 Reserved

P2 Codes: When a "2" is included in the P1 (Exit Windows group code), P2 provides the message to display to the user. Use "" or "0" to clear any previously-set exit message.

Terminate Group (choose one).

Used to direct WIL to allow itself to be terminated without warning or to simply refuse any termination request (such as Ctrl-Break).

P1 Meaning

- 0 - Provide notification message when program terminated by user.
- 4 - Allow quiet termination.
- 8 - Refuse to terminate.

Example:

```
; We want to refuse termination requests and refuse any attempt to
; exit Windows until the WIL script is complete
; Add codes 2 and 8 making 10
IntControl(12,10,"Close Net apps before exiting Windows", 0, 0 )
Message("IntControl"," Done.")
```

IntControl (20, 0, 0, 0, 0)

Returns window handle of current parent window. (Similar to **DllHwnd**)

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, p1, p2, p3, p4)

Issues a Windows "PostMessage"

- p1 Window handle
- p2 Message ID number (in decimal)
- p3 wParam
- p4 lParam assumed to be numeric

IntControl(26, 0, 0, 0, 0)

Re-assesses the language currently being used and makes any necessary changes to the language strings used by the WIL Interpreter. Normally, this is done at program startup.

IntControl(28, p1, 0, 0, 0)

Selects system font used in list boxes.

<u>P1</u>	<u>Meaning</u>
0	proportional font (default)
1	fixed pitch font
2	GUI font (Windows 95+ and NT 4.0+ only).

Note: that this option applies to AskLine (and AskPassword), as well as to AskFileText and AskItemList.

Returns the current font type (0 or 1, as above)

IntControl(29, p1, 0, 0, 0)

Changes the default file delimiter.

- p1 New delimiter

We have added the ability to change the file delimiter to a character of your own choosing, using the new **IntControl(29,0,0,0)**. If you are using the 32-bit version of WIL, and want to make the file delimiter a space for compatibility with existing scripts, you can place the following line at the beginning of each of your scripts:

```
IntControl(29, " ", 0, 0, 0)
```

Conversely, if you want to standardize on a TAB delimiter, you can use:

```
IntControl(29, @TAB, 0, 0, 0)
```

The first parameter for IntControl is the new file delimiter you want to use, and must be a single character. The return value of the function is the previous file delimiter character. If you specify an empty string ("") as the first parameter, the function will return the current file delimiter character but the file delimiter will not be changed.

IntControl(30, p1, p2, 0, 0) {*NT}

Performs a delayed file move.

- p1 source file
- p2 destination

The file is not actually moved until the operating system is restarted. This can be useful for replacing system files. "Sourcefile" must be a single file name, with no wildcards. "Destination" may be a file name (which may contain wildcards) or a directory name. The destination file MUST be on the same drive as the source

file. If the destination file exists, it will be replaced without warning. "Destination" can also be a NULL string (""), in which case the source file will be deleted when the operating system is restarted.

Under Windows 95, and in the 16-bit version, this function performs a regular (non-delayed) **FileMove**.

This function returns "1" on success, "2" if it performed a regular **FileMove** instead, and "0" on failure.

IntControl(31, 0, 0, 0, 0) {*Explorer}

Returns "Window ID's" for all Explorer windows.

This function returns a tab-delimited list of Window ID's for all open Windows 95 Explorer windows.

IntControl(32, address, "data type", 0, 0)

Returns the contents of the memory location specified by "address".

"Data type" specifies the type of data to be retrieved:

"BYTE" - returns a byte

"WORD" - returns a word

"LONG" - returns a long integer

IntControl(33, p1, 0, 0, 0)

Controls whether a listbox control in a dialog box allows multiple items to be selected.

<u>P1</u>	<u>Meaning</u>
0	Single selection
1	Multiple selection (default)

IntControl(34, p1, 0, 0, 0)

Returns the error message string which corresponds to the specified WIL error.

p1 error number.

IntControl(35, p1, 0, 0, 0) {*32}

Slows down **SendKey**.

p1 amount of time to delay between each keypress, in milliseconds
(1000 milliseconds = 1 second);

0 = no delay (default).

Returns previous delay setting. The default delay (in the 32-bit version) is 25. Specify a larger number to slow **SendKey** down, or a smaller number to speed it up

IntControl(36, p1, p2, 0, 0) {*32}

Waits until an application is waiting for user input.

p1 = window name associated with application

p2 = time-out, in milliseconds (-1 = no time-out)

This function waits until the process which created the specified window is waiting for user input with no input pending, or until the specified time-out interval has elapsed. It can only be used with 32-bit GUI applications. It returns **@TRUE** if it has successfully waited, or **@FALSE** if a time-out has occurred (or if it was unable to initiate a wait).

IntControl(37, p1, p2, p3, 0)

Replaces the specified icon in an EXE file.

p1 = EXE file
p2 = icon file
p3 = ordinal position of icon to replace

This is similar to the **IconReplace** function, except it allows you to change an icon other than the first icon in the file.

Note: This **IntControl** is subject to change in future versions.

IntControl(39, p1, 0, 0, 0)

Sets the file sharing mode for file reads.

p1 = share mode for file reads (default = 1)

Valid share modes are:

<u>P1</u>	<u>Meaning</u>
-1	don't change (keep current setting)
0	prevent the file from being shared
1	allow other open operations for read access
2	allow other open operations for write access
3	allow other open operations for read and write access

This **IntControl** will affect the **FileOpen**("READ") and **BinaryRead** functions. Returns previous setting.

IntControl(40, p1, 0, 0, 0)

Sets the file sharing mode for file writes.

P1 share mode for file writes (default = 0)

This affects the **FileOpen**("WRITE"/"APPEND") and **BinaryWrite** functions. Returns previous setting. See **IntControl(39,p1,0,0,0)** for a list of valid share modes.

IntControl(41, p1, 0, 0, 0)

Controls YmdHms format returned by time functions.

<u>P1</u>	<u>Meaning</u>
0	2-digit year (YY:MM:DD:HH:MM:SS)
1	4-digit year (YYYY:MM:DD:HH:MM:SS) (default)
-1	Don't change (just return current setting)

This affects the following functions which return a YmdHms string: **FileYmdHms**, **TimeAdd**, **TimeJulToYmd**, **TimeSubtract**, **TimeYmdHms**. Returns previous setting.

IntControl(42, p1, 0, 0, 0)

Get the pointer to a binary buffer.

p1 specifies the binary buffer (see **BinaryAlloc**).

This returns the address of a pointer to a binary buffer. The address will be within the memory space of the WIL Interpreter.

IntControl(43, p1, 0, 0, 0) (32-bit only)

Enable/disable "WaitForInputIdle".

<u>p1</u>	<u>Meaning</u>
0	Don't wait for application to be ready.
1	Wait for application to be ready (default).
-1	Don't change (just return current setting)

This controls whether **SendKey**, **SendKeysTo** and **SendKeysChild** attempt to wait until the active application is ready to accept input before sending each keystroke. Returns the previous setting.

IntControl(44, p1, 0, 0, 0)

Returns class name for specified window handle.

p1 = window handle.

The window handle can be obtained with the **DllHwnd** function. Returns a blank string ("") if the window doesn't exist.

IntControl(45, p1, 0, 0, 0) {*32}

Enable/disable "**SendKey** speedup" in DOS programs in Windows NT.

<u>p1</u>	<u>Meaning</u>
-1	Don't change (returns current setting)
0	Don't speed up SendKey in DOS programs in Windows NT.
1	Speed up SendKey in DOS programs in Windows NT (default).

Normally, **SendKey**, **SendKeysTo** and **SendKeysChild** yield processor time between keystrokes allowing other Windows programs to multitask. However, when sending keystrokes to a DOS program in Windows NT, this can yield can result in excessive delays between keystrokes. When this occurs, **SendKey** goes into "speedup" mode and dominates the processors time.

SendKey speed is also affected by the delay inserted between keystrokes. By default, the delay between keystrokes is 25 milliseconds. (This length can be adjusted with **IntControl(35,p1,0,0,0)**). However, when "**SendKey** speedup" mode is in effect, no such delay is inserted.

IntControl(45,p1,0,0,0) allows you to disable the "**SendKey** speedup" behavior in the case that it causes problems or is not desired.

Returns previous setting.

IntControl(46, p1, 0, 0, 0) {*32}

Sets window retry timeout.

<u>p1</u>	<u>Meaning</u>
-1	Don't change (returns current setting)
0	No retries
secs	The number of seconds to wait (default = 9)

In the 32-bit version, WIL functions which take a window title as a parameter (except for WinExist) will wait up to 9 seconds for the specified window to appear. **IntControl(46,p1,0,0,0)** allows the timeout setting to be changed. Returns previous setting.

IntControl(47, p1, 0, 0, 0)

Closes another WinBatch program.

IntControl 47 allows a WIL script to close another WinBatch program. ("**WinClose**" cannot close WinBatch programs.)

p1 = full or partial windowname of a WinBatch program window.

If p1 is the current window, this function is the same as an Exit. Returns **@TRUE** on apparent success, or **@FALSE** on apparent failure.

IntControl(48, p1, 0, 0, 0)

Closes a DOS program.

IntControl 48 allows an MS-DOS program to be closed. ("**WinClose**" cannot close DOS windows.)

p1 = full or partial windowname of a DOS program window.

If p1 is not a DOS window, this function has no effect.

Returns **@TRUE** on apparent success, or **@FALSE** on apparent failure.

Note: You will not receive any warning from the operating system before the window is closed, even if the DOS application contains unsaved work. Use this function carefully. This function is not supported in Windows NT.

IntControl(49, p1, p2, 0, 0)

Adds system menus to WIL popup windows.

"p1" specifies which popup windows will have system menus:

p1	Meaning
-1	Don't change (return current setting)
0	No system menus (default).
1	Dialog boxes (created with " Dialog " or the WIL Dialog Editor).
2	Item listboxes (e.g., created with " AskItemList ", " AskFileText " or related functions).
3	Both Dialog boxes (1) and Item list boxes (2).

p2 **Meaning**

specifies the value that a dialog box (created using the "Dialog" function) will return if the user closes the dialog without pressing one of the pushbuttons in the dialog (eg, by pressing <Alt-F4>, or by clicking on the "Close" icon in the title bar). The default is 1. If a dialog returns 0, processing will be transferred to the label in the script marked ":Cancel" (if any).

IntControl 49 allows certain types of WIL popup windows to have system menus, (the drop down menu of minimize/maximize options accessed via the small icon or box to the left side of a windows title bar).

Returns previous setting for p1.

IntControl(50, p1, p2, 0, 0)

Controls whether "Go to web page" button is added to WIL error boxes.

<u>p1</u>	<u>Meaning</u>
-1	Don't change (return current setting)
0	Remove "Go to web page" button from error boxes.
1	Add "Go to web page" button to error boxes (default).

When an error occurs in a WIL program an error message box is displayed. Prior to display, WIL checks the system to see if a Web browser has been configured; if so, a "Go to Tech Support Web Page" push-button is added to the error message box. When pressed, the button launches the Web browser and opens to the Wilson WindowWare Tech Support Web Page.

IntControl 50 prevents the "Go to Tech Support Web Page" button from being added.

Returns previous setting.

<u>p2</u>	<u>Url</u>
default	"http://www.windowware.com/"
user defined	"http://www.domain.com"

P2 can specify the URL of the web page to be launched when the user clicks on the "Go to Tech Support Web Page" button. If p2 is "" or 0, no change is made to URL that is currently set. The default is:

IntControl(50, 1, "http://techsupt.windowware.com/", 0, 0)

IntControl(51, p1, 0, 0, 0) {*32}

Specifies flags for CreateProcess.

P1 = one or more of the following flags, combined using the binary "OR" ("|") operator. The default setting is 32.

In the 32-bit version, all the **Run[.]** commands call the Windows API "CreateProcess" function to launch the application. This **IntControl** lets you specify certain flags that control the priority class and the creation of the process, and affects all subsequent **Run[.]** commands. Under normal circumstances, you should not need to use this **IntControl** at all. It should be used only by knowledgeable users, and completely at your own risk. Some of these flags can have unexpected or undesirable results. We cannot provide any support for

their use.

Returns previous setting

Value	Flag Name	Meaning
1	DEBUG_PROCESS	If this flag is set, the calling process is treated as a debugger, and the new process is a process being debugged. The system notifies the debugger of all debug events that occur in the process is being debugged. If you create a process with this flag set, only the calling thread (the thread that called CreateProcess) can call the WaitForDebugEvent function.
2	DEBUG_ONLY_THIS_PROCESS	If not set and the calling process is being debugged, the new process becomes another process being debugged by the calling processs debugger. If the calling process is not a process being debugged, no debugging-related actions occur.
4	CREATE_SUSPENDED	The primary thread of the new process is created in a suspended state, and does not run until the ResumeThread function is called.
8	DETACHED_PROCESS	For console processes, the new process does not have access to the console of the parent process. The new process can call the AllocConsole function at a later time to create a new console. This flag cannot be used with the CREATE_NEW_CONSOLE flag.
16	CREATE_NEW_CONSOLE	The new process has a new console, instead of inheriting the parents console. This flag cannot be used with the DETACHED_PROCESS flag.
512	CREATE_NEW_PROCESS_GROUP	The new process is the root process of a new process group. The process group includes all processes that are descendants of this root process. The process identifier of the new process group is the same as the process identifier, which is returned in the lpProcessInformation parameter. Process groups are used by the GenerateConsoleCtrlEvent function to enable sending a CTRL+C or CTRL+BREAK signal to a group of console processes.
1024	CREATE_UNICODE_ENVIRONMENT	If set, the environment block pointed to by lpEnvironment uses Unicode characters. If clear, the environment block uses ANSI characters.
2048	CREATE_SEPARATE_WOW_VDM	Windows NT only: This flag is valid only when starting a 16-bit Windows-based application. If set, the new process is run in a private Virtual DOS Machine (VDM). By default, all 16-bit Windows-based

applications are run as threads in a single, shared VDM. The advantage of running separately is that a crash only kills the single VDM; any other programs running in distinct VDMs continue to function normally. Also, 16-bit Windows-based applications that are run in separate VDMs have separate input queues. That means that if one application hangs momentarily, applications in separate VDMs continue to receive input.

- 4096 CREATE_SHARED_WOW_VDM
Windows NT only: The flag is valid only when starting a 16-bit Windows-based application. If the DefaultSeparateVDM switch in the Windows section of WIN.INI is TRUE, this flag causes the CreateProcess function to override the switch and run the new process in the shared Virtual DOS Machine.
- 67108864 CREATE_DEFAULT_ERROR_MODE
The new process does not inherit the error mode of the calling process. Instead, CreateProcess gives the new process the current default error mode. An application sets the current default error mode by calling SetErrorMode. This flag is particularly useful for multi-threaded shell applications that run with hard errors disabled. The default behavior for CreateProcess is for the new process to inherit the error mode of the caller. Setting this flag changes that default behavior.

These flags control the new process's priority class, which is used in determining the scheduling priorities of the process's threads. If none of the following priority class flags are specified, the priority class defaults to NORMAL_PRIORITY_CLASS unless the priority class of the creating process is IDLE_PRIORITY_CLASS. In this case, the default priority class of the child process is IDLE_PRIORITY_CLASS.

One of the following flags can be specified:

Value	Flag Name	Meaning
32	NORMAL_PRIORITY_CLASS	Indicates a normal process with no special scheduling needs.
64	IDLE_PRIORITY_CLASS	Indicates a process whose threads run only when the system is idle and are preempted by the threads of any process running in a higher priority class. An example is a screen saver. The idle priority class is inherited by child processes.
128	HIGH_PRIORITY_CLASS	Indicates a process that performs time-critical tasks that must be executed immediately for it to run correctly. The threads of a high-priority class process preempt the threads of normal-priority or idle-priority class processes. An example is Windows Task List, which must respond quickly when called by the user, regardless of the load on the

operating system. Use extreme care when using the high-priority class, because a high-priority class CPU-bound application can use nearly all available cycles.

256 REALTIME_PRIORITY_CLASS

Indicates a process that has the highest possible priority. The threads of a real-time priority class process preempt the threads of all other processes, including operating system processes performing important tasks. For example, a real-time process that executes for more than a very brief interval can cause disk caches not to flush or cause the mouse to be unresponsive.

IntControl (52, p1, 0, 0, 0) (32 bit only)

Set dialog font. This IntControl lets you specify whether the standard system font or GUI font is used by dialog boxes created with the Dialog command. By default, the GUI font is used.

Note: The GUI font is available only in Windows 95 and Windows NT 4.0.

<u>p1</u>	<u>Meaning</u>
-1	Don't change (just return current setting)
0	Use standard system font.
1	Use GUI font (default).

IntControl (53, p1, 0, 0, 0)

Set line terminator type for FileWrite.

Note: Normally, FileWrite adds a carriage return and line feed (CR/LF) after writing each line to the file. This IntControl lets you select a different line terminator, or none at all, as specified by "p1".

<u>p1</u>	<u>Terminator</u>
0	No line terminator
1	CR/LF (DOS) (default)
2	LF (UNIX)
3	CR (Macintosh)
4	Tab

This function returns the previous setting.

IntControl(54, p1, p2, 0, 0)

Keep window on top.

This IntControl tells the specified window to remain on top of all other windows, or to return to a normal (non-topmost) state.

P1 = partial windowname of the window to be affected.

<u>P2</u>	<u>Meaning</u>
--	-----

0	Don't stay on top
1	Stay on top

Returns @TRUE on success, @FALSE on failure.

IntControl(56, p1, 0, 0, 0)

Terminate an application.

This IntControl lets you terminate an application abruptly. Normally, you should use WinClose to close an application gracefully, but sometimes that may not be possible. Using this IntControl is a drastic, last-resort method of terminating an app.

Note: You will not receive any warning from the operating system before the application is closed, even if it contains unsaved work. Also, this function may leave the system in an unstable state. Use this function carefully.

P1 = partial windowname of the application you wish to terminate.

Returns @TRUE on success, @FALSE on failure.

IntControl(57, p1, 0, 0, 0) {*32 Windows 95 only}

Disable/enable system keys.

This IntControl can be used to disable (or re-enable) the following system key-combinations:

<Alt-Tab>
<Alt-Esc>
<Ctrl-Alt-Del>

It works by telling the operating system that the screen saver is running (even though it isn't), which disables those keys.

P2	Meaning
--	-----
0	Enable system keys
1	Disable system keys

Note: This function is not supported in Windows NT.

Returns @TRUE on success, @FALSE on failure.

IntControl(58, p1, 0, 0, 0) {*32}

Set system time.

P1 = new system time, in YmdHms format.

This IntControl changes the time of the computer's system clock.

Returns @TRUE on success, @FALSE on failure.

IntControl(59, p1, p2, 0, 0)

Sends a WM_WININICHANGE or WM_SETTINGCHANGE message.

This IntControl sends a message to one or all application windows, informing them that a change has been made to WIN.INI or to the registry.

p1 = window handle of the window to send the message to, or -1 for all top-level windows.

p2 = name of the WIN.INI section or registry key that has been changed. If this parameter specifies a WIN.INI section, it should not include square brackets around the name. If it specifies a registry key, it typically indicates only the leaf node in the registry, not the whole path, but this will be application-dependent. You can specify a blank string ("") to request that the application re-read all sections or keys that affect it.

In the 16-bit version, this sends a WM_WININICHANGE message. In the 32-bit version, it sends a WM_SETTINGCHANGE message.

Returns @TRUE on success; @FALSE on error.

IntControl(60, p1, p2, 0, 0)

Sends a WM_DEVMODECHANGE message.

This IntControl sends a message to one or all application windows, informing them that a change has been made to device mode settings.

p1 = window handle of the window to send the message to, or -1 for all top-level windows.

p2 = name of the device (as specified in the [Devices] section of WIN.INI) whose settings have been changed.

Returns @TRUE on success; @FALSE on error.

IntControl (61, p1, 0, 0, 0)

Sets WinActivate() method.

In Windows 98 and NT 5.0, the standard method that WinActivate (and SendKeysTo and MouseClick) uses to activate an application window is no longer supported. Therefore, we must use an alternate method to activate windows under these operating systems. This IntControl lets you change the current method being used. Some methods may work with some particular windows and not others, so you may need to experiment.

P1 Method to use

--	-----
-1	Don't change (just return current setting).
0	Standard method (SetForegroundWindow API call).
1	Switch to the window, then click on it.
2	Iconize the window, then switch to it.

- 3 Change foreground lock timeout.
- 4 Attach to the input thread of the foreground window.

The default method is "3" when running under Windows 98 or NT 5.0. The default is "0" when running under previous versions of Windows.

Returns previous setting.

IntControl (62, p1, 0, 0, 0)

Sets dialog activation method.

This IntControl lets you change the method used to activate dialog windows that are displayed by the WIL Interpreter. See IntControl(61) for more information.

IntControl (63, p1, p2, p3, p4)

Sets coordinates for AskFileText and AskItemList windows.

This function sets the window coordinates for the dialog displayed by the next AskFileText or AskItemList function call. The coordinates will be reset to default values after the Ask[...] function is called.

Coordinates are based on a virtual 1000 x 1000 screen:

- p1 = left (upper-x)
- p2 = top (upper-y)
- p3 = right (lower-x)
- p4 = bottom (lower-y)

To explicitly indicate that default values should be used, use:

IntControl(63, 0, 0, 0, 0)

Returns 1.

IntControl (64, 0, 0, 0, 0)

Gets the exit code returned by the last program run.

This function gets the exit code (also known as the "errorlevel") that was returned by the program that was most-recently run using RunWait (or using RunShell with the @WAIT flag specified). If no program has been run, or if the last-run program was not run in @WAIT mode, this function will return 99999.

IntControl (66, p1, 0, 0, 0)

In 32 bit versions, this function logs the user out of the current session

p1	Meaning
0	default. Normal shutdown. User is prompted for open files.
1	Force. Forces the restart of Windows by not giving notice to any running applications about the restart. This doesn't allow the applications to save information or to display shut down messages.

In 16 bit version, this function 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. (Similar to the function EndSession.)

Be very careful using IntControl(66,1,0,0,0) , this can cause the loss of data.

IntControl (67, p1, 0, 0, 0)

This will cause of reboot of Windows NT machines.

p1	Meaning
0	default. Normal shutdown. User is prompted for open files.
1	Force. Forces the reboot of the system by not giving notice to any running applications about the reboot. This doesn't allow the applications to save information or to display shut down messages.

In 16 bit version, 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.

Be very careful using IntControl(67,1,0,0,0) , this can cause the loss of data.

IntControl (68, p1, 0, 0, 0)

This will cause a shutdown of the machine, awaiting power off.

p1	Meaning
0	default. Normal shutdown. User is prompted for open files.
1	Force. Forces the reboot of the system by not giving notice to any running applications about the reboot. This doesn't allow the applications to save information or to display shut down messages.

In 16 bit version, 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.

Be very careful using IntControl(68,1,0,0,0) , this can cause the loss of data.

IntControl(1000, p1, 0, 0, 0)

Sets the exit code returned by WinBatch.

p1 exit code

This specifies the value returned by WinBatch's "WinMain" (entry point) function. Returns the previous setting.

IntControl(1001, p1, 0, 0, 0) {*32}

Set service process status.

p1	Meaning
0	Run as a normal application (default).
1	Run as a service.

If WinBatch is running as a service, it will not be closed when the user logs out.

Under Windows 95, this function actually registers the currently executing WinBatch program as a service. Under Windows NT, it just causes the WinBatch program to behave as a service (ie, don't close when the user logs out); in order to launch a WinBatch program as a service under Windows NT, you

will need to use a utility such as the SRVANY.EXE program which is included with the Windows NT Resource Kit.

Returns **@TRUE** on success, **@FALSE** on failure.

IntControl(1002, p1, 0, 0, 0)

Sets WinBatch icon display state.

p1	Meaning
-1	Don't change (returns current setting)
0	Hide WinBatch icon.
1	Show WinBatch icon (default).

This function sets the display state of the WinBatch icon for the duration of the script. The default display state is minimized.

This IntControl is equivalent to using **WinShow()** or **WinHide()** on the current WinBatch window. However, **WinShow** and **WinHide** do not have the capability of hiding the icon permanently. If any Box[...] functions are used to display information, the WinBatch icon will return to its previous display state.

Returns previous setting.

IntControl(1003, p1, 0, 0, 0)

Prevents the WinBatch icon from being opened.

This function sets the "open-able" flag for the WinBatch icon. If the flag is on (default), then the WinBatch icon can be opened (restored) to a normal window by clicking on the icon. If the flag is off, then it is not possible to open the WinBatch icon by any method, including **WinShow()** or **BoxOpen()**.

p1	Meaning
-1	Don't change (returns current setting)
0	WinBatch icon cannot be opened.
1	WinBatch icon can be opened (default).

Returns previous setting.

IntControl(1004, 0, 0, 0, 0)

Returns the full path and file name of the current WinBatch program.

If the current WinBatch script is a child program that was called with the **Call()** function, this **IntControl** will return the name of the main (calling) program.

If the current WinBatch script is a child program that was called with the **CallExt()** function, this **IntControl** will return the name of the current (called) program.

IntControl(1005, 0, 0, 0, 0)

Gets or starts shutdown script mode.

This IntControl allows you to specify a script that will run when Windows is shut down or (in Windows 95 and Windows NT) when the user logs off.

<u>P1</u>	<u>Meaning</u>
0	Get shutdown state (@TRUE if in shutdown mode, @FALSE if normal script)
1	Enter shutdown mode

If "p1" == 0, the function will return a TRUE/FALSE flag indicating whether the current WinBatch script is running in shutdown mode.

If "p1"== 1, the current script will stop processing and will enter shutdown mode.

When WinBatch enters shutdown mode, it stops processing the current script, closes the normal WinBatch window, and stays running in the background waiting for Windows to shut down (or the user to log off). When this shutdown event occurs, WinBatch re-loads the script (the same script that was running when shutdown mode was entered) and runs it again from the beginning. If the script that was previously running was launched via a "Call" command, WinBatch will re-load and run the parent (calling) script.

The actual shutdown procedure varies a bit among different versions of Windows. Here is what happens, step-by-step, when WinBatch is in shutdown mode, and the user initiates a Windows shutdown (or logoff):

Windows 3.1 / Windows for Workgroups:

Windows queries all running applications, asking whether they agree to close. WinBatch agrees to close. If all the other apps agree to close as well, then Windows informs all the apps that the session is ending. At that time, WinBatch runs the script. When the script is finished, Windows shuts down.

Windows 95:

Windows queries all running applications, asking whether they agree to close. WinBatch agrees to close. If all the other apps agree to close as well, then Windows informs all the apps that the session is ending. At that time, WinBatch runs the script. When the script is finished, the Windows session ends. If the script takes more than about 20 seconds to run, Windows unconditionally terminates the script and the Windows session ends.

Windows NT:

Windows queries each running application in turn, asking whether it agrees to close. When it queries WinBatch, WinBatch displays a message box, asking the user to first close all other open windows. When the user presses the "OK" button in the message box, WinBatch agrees to close, then Windows informs WinBatch that the session is ending, and then WinBatch runs the script. (If, however, the user presses the "Cancel" button in the message box, WinBatch refuses to close, and the shutdown process is cancelled.) If the script takes more than about 20 seconds to run, Windows displays a message box, giving the user an opportunity to terminate WinBatch, but the script continues processing until the user responds to the message. When the script is finished, if there are any other apps still running, Windows continues querying each app in turn, asking whether it agrees to close. After all the apps agree to close, the Windows session ends. If any app refuses to close, the shutdown process is cancelled.

Example:

```
IntControl(1005... FUNCTION:

rc = IntControl(1005, 0, 0, 0, 0) ; get shutdown state
Message("ShutDown State=", rc)

If rc == @FALSE ; if we're running as a normal
script
new=IntControl(1005, 1, 0, 0, 0) ; enter shutdown mode
Message("ShutDown State=", new)
Exit ; this 'Exit' is not really
necessary
Endif

; if we got here, we're running in shutdown mode
Display(1, "Windows Shutdown", "WinBatch shutdown processing
begins now")
ErrorMode(@OFF)
FileDelete("c:\temp\*.tmp")
ErrorMode(@CANCEL)
```

IntControl(1006, 0, 0, 0, 0)

Returns the un-parsed WinBatch command line.

This returns the actual command line string that WinBatch was launched with.

Example:

```
;put this in a WBT file and run it):

rc=IntControl(1006, 0, 0, 0, 0) ; get un-parsed WinBatch
command line
Message("Un-parsed Command Line String =", rc)
```

IntControl(1007, p1, p2, p3, p4)

Add/remove/check tray icon. (This function requires Windows 95+ or NT 4.0+).

p1 Meaning

-- -----

- 0 Check whether tray icon has been clicked (and reset "clicked" state)
- 1 Add currently-running script to the system tray
- 2 Remove currently-running script from the system tray

If p1 == 1 (add icon), then p2 can be one or more of the following flags, combined with the binary OR ("|") operator:

p2 Meaning

- 1 Hide regular WinBatch icon while the icon is in the system tray
- 2 Suspend script until user clicks on the tray icon

p3 Tool tip (ie, string that is displayed when the mouse passes over the tray icon), or "" for none.

p4 Icon file, or "" for the default icon. If a file name is specified, it

can be a .ICO file, or an .EXE or .DLL (or similar resource file) containing icons. If it is a resource containing multiple icons, by default the first icon in the file is used.

You can specify a different icon using the following format:

`"filename|#"`

where "filename" is the name of the file, followed by a vertical bar and then the offset of the desired icon in the file ("#"). The first icon in a file has an offset of 0. If an invalid icon file is specified, the default icon will be used.

If the "suspend script" flag is specified in p2, then the WinBatch script will be suspended until the user clicks on the tray icon, at which point the script will continue, and this function will return one of the click values listed below. Otherwise, the WinBatch script will continue running, and you can periodically check to see whether the user has clicked on the tray icon by calling this function with p1 == 0.

When called with p1 == 0 (or when returning from being suspended), this function will return one of the following values:

Value	Meaning
0	Not clicked
1	Left click
2	Right click

Each time you call this function with p1 == 0, the "click state" will be reset to 0 (not clicked).

Returns @TRUE (success) or @FALSE (failure), or a click value.

Example:

```
IntControl(1007, 1, 2, "Click me!", "shell32.dll|41")
```

IsDefined

Determines if a variable name is currently defined.

Syntax:

```
IsDefined (var)
```

Parameters:

(s) var a variable name.

Returns:

(i) **@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 to the left of an equal sign in a statement. It stays defined until it is explicitly dropped with the **Drop** function, or until the current invocation of the WIL Interpreter gets closed.

Generally speaking: in batch file-based implementations of WIL, all variables are dropped automatically at the end of every batch file; and in menu-based implementations of WIL, variables stay defined until explicitly dropped.

Example:

```
if IsDefined(thisvar)
    Message("Value of thisvar is", thisvar)
else
    Message("ERROR!", "Variable not defined")
endif
```

See Also:

[Drop](#)

IsFloat

Tests whether a variable can be converted to a floating point number.

Syntax:

```
IsFloat(x)
```

Parameters:

(s) *x* value to be tested.

Returns:

(i) **@TRUE** if the data can be converted to a floating point number;
@FALSE if the data cannot be converted to a floating point number.

Use this function to test whether a variable can be converted into a floating point number.

Example:

```
A=IsFloat(4)
Message("Is 4 a floating point number", A)
B=IsFloat("Hamburger")
Message('Is "Hamburger" a floating point number', B)
C=IsFloat(4.5)
Message("Is 4.5 a floating point number", C)
```

See Also:

[IsInt](#), [IsNumber](#)

IsInt

Tests whether a variable is or can be converted into a valid integer.

Syntax:

```
IsInt(x)
```

Parameters:

(s) *x* value to be tested.

Returns:

(i) **@TRUE** if the data is or can be converted to a valid integer;
@FALSE if the data is not or cannot be converted to a valid integer.

Use this function to test whether a variable can be converted into a valid integer.

Example:

```
A=IsInt(4)
Message("Is 4 an integer", A)
B=IsInt("Hamburger")
Message('Is "Hamburger" an integer', B)
C=IsInt(4.5)
Message("Is 4.5 an integer", C)
```

See Also:

[IsFloat](#), [IsNumber](#)

IsKeyDown

Tells about keys/mouse.

Syntax:

```
IsKeyDown(keycodes)
```

Parameters:

(i) keycodes **@SHIFT** and/or **@CTRL**.

Returns:

(i) **@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.

Examples:

```
IsKeyDown (@SHIFT)
```

```
IsKeyDown (@CTRL)
```

```
IsKeyDown (@CTRL | @SHIFT)
```

```
IsKeyDown (@CTRL & @SHIFT)  
Message ("IsKeyDown", "Key Is Down.")
```

See Also:

[WaitForKey](#)

IsLicensed

Tells if the calling application is licensed.

Syntax:

```
IsLicensed ( )
```

Parameters:

(none)

Returns:

(i) **@YES** if it is licensed;
 @NO if it is not licensed.

Returns information on whether or not the currently-running version of the calling application is a licensed copy.

Example:

```
a=IsLicensed( )
If a == @true
    Message("IsLicensed","It's Licensed.")
else
    Message("IsLicensed","No, it's not licensed.")
endif
```

See Also:

[Version](#), [VersionDLL](#)

IsMenuChecked {*M}

Determines if a menu item has a checkmark next to it.

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.

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.

Example:

```
; assume we've defined a "Misc | Prompt Often" menu item elsewhere
; For menu based implementations of WIL only

Delete Backup File
  prompt = IsMenuChecked("MiscPromptOften")
  if prompt==@TRUE
    confirmed = AskYesNo("Delete backups???", "REALLY do this?")
  else
    confirmed = @YES
  endif
  if confirmed ==@ YES
    ; some risky operation the user has just confirmed
    FileDelete("C:\temp\backup\*. *" )
  endif
```

See Also:

[IsMenuEnabled](#), [MenuChange](#)

IsMenuEnabled {*M}

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.

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.

Example:

```
; Assuming a menu based implementation of WIL, and also
; assuming a menu item "Utilities" with a submenu item of
; "Edit Bat File" that may be enabled or disabled.
;
; This code only allows editing of autoexec.bat file
; if the menu item is enabled
rslt=IsMenuEnabled("UtilitiesEditBatFile")
if rslt==@TRUE
    Run("notepad.exe", "c:\autoexec.bat")
else
    Message("Sorry","Bat file editing disabled")
endif
```

See Also:

[IsMenuChecked](#), [MenuChange](#)

IsNumber

Tests whether a variable is or can be converted into a valid number.

Syntax:

```
IsNumber(x)
```

Parameters:

(s) *x* value to be tested

Returns:

(i) **@TRUE** if the data is or can be converted to a valid number;
@FALSE if the data is not or cannot be converted to a valid number.

Use this function to test whether a variable can be converted into a valid number, either an integer or a floating point number.

Example:

```
A=IsNumber(4)
Message("Is 4 a number", A)
B=IsNumber("Hamburger")
Message('Is "Hamburger" a number', B)
C=IsNumber(4.5)
Message("Is 4.5 a number", C)
```

See Also:

[IsFloat](#), [IsInt](#),

ItemCount

Returns the number of items in a list.

Syntax:

```
ItemCount (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:

- (i) the number of items in the list.

If you create the list with the [FileItemize](#) or [DirItemize](#) functions your delimiter will be determined by the version of WIL. The 16-bit returns a space-delimited list and the 32-bit version returns a tab-delimited list. [WinItemize](#), however, creates a tab-delimited list of window titles since titles can have embedded spaces.

Example:

```
IntControl(29, @TAB, 0, 0, 0)
a = FileItemize("*.*)
n = ItemCount(a, @TAB)
Message("Note", "There are %n% files")
```

See Also:

[ItemExtract](#), [AskItemList](#)

ItemExtract

Returns the selected item from a list.

Syntax:

ItemExtract (index, list, delimiter)

Parameters:

- (i) index the position in **list** of the item to be selected.
- (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) the selected item.

Note: Using -1 as the index will return the last item in the list

If you create the list with the **FileItemize** or **DirItemize** functions your delimiter will be determined by the version of WIL. The 16-bit returns a space-delimited list and the 32-bit version returns a tab-delimited list. **WinItemize**, however, creates a tab-delimited list of window titles since titles can have embedded spaces.

Example:

```
IntControl(29, @TAB, 0, 0, 0)
DirChange(DirWindows(0))
bmpfiles = FileItemize("*.bmp")
bmpcount = ItemCount(bmpfiles, @TAB)
pos = (Random(bmpcount - 1)) + 1
paper = ItemExtract(pos, bmpfiles, @TAB)
Wallpaper(paper, @FALSE)
Message("ItemExtract", "Wallpaper changed to %paper% - not tiled")
```

See Also:

[ItemCount](#), [ItemLocate](#), [AskItemList](#), [ItemSort](#)

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 your delimiter will be determined by the version of WIL. The 16-bit returns a space-delimited list and the 32-bit version returns a tab-delimited list. **WinItemize**, however, creates a tab-delimited list of window titles since titles can have embedded spaces.

Example:

```
item="apricots"  
list="apples pears oranges grapes"  
newlist = ItemInsert(item, -1, list, " ")  
message("List after ItemInsert", newlist)
```

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**. (this is case sensitive)
- (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 your delimiter will be determined by the version of WIL. The 16-bit returns a space-delimited list and the 32-bit version returns a tab-delimited list. **WinItemize**, however, creates a tab-delimited list of window titles since titles can have embedded spaces.

Example:

```
list="apples pears oranges grapes"  
index=ItemLocate("oranges", list, " ")  
message("The item is located at index #", index)
```

See Also:

[ItemExtract](#), [ItemRemove](#)

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 **Dirlitemize** functions your delimiter will be determined by the version of WIL. The 16-bit returns a space-delimited list and the 32-bit version returns a tab-delimited list. **WinItemize**, however, creates a tab-delimited list of window titles since titles can have embedded spaces.

Example:

```
Exlist="apples pears oranges grapes"  
index=ItemLocate("oranges", list, " ")  
newlist = ItemRemove(index, list, " ")  
message("List after item is removed", newlist)
```

See Also:

[ItemCount](#), [ItemInsert](#), [ItemLocate](#)

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 your delimiter will be determined by the version of WIL. The 16-bit returns a space-delimited list and the 32-bit version returns a tab-delimited list. **WinItemize**, however, creates a tab-delimited list of window titles since titles can have embedded spaces.

Example:

```
list=" one two three four "  
newlist = ItemSort(list, " ")  
message("List generated by ItemSort", newlist)
```

See Also:

[ItemSortNc](#), [ItemExtract](#)

ItemSortNc

Sorts a list, without regard to case.

Syntax:

```
ItemSortNc (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 your delimiter will be determined by the version of WIL. The 16-bit returns a space-delimited list and the 32-bit version returns a tab-delimited list. [WinItemize](#), however, creates a tab-delimited list of window titles since titles can have embedded spaces.

Example:

```
list=" one two three four "  
newlist = ItemSortNc(list, " ")  
message("List generated by ItemSort", newlist)
```

See Also:

[ItemExtract](#), [ItemSort](#)

KeyToggleGet

Returns the status of a toggle key.

Syntax:

```
KeyToggleGet(@key)
```

Parameters:

(i) @key	the toggle key in question. Values may be one of:
	@CAPSLOCK for the CapsLock key
	@NUMLOCK for the NumLock key
	@SCROLLLOCK for the ScrollLock key

Returns:

(i)	@ON	Key was set.
	@OFF	Key was not set.

Use this function to obtain the state of one of the toggle keys - the CapsLock, NumLock, and ScrollLock keys.

Note: On DOS based systems, this function will return the keys state for all applications. For 32 bit Windows based systems, the return value will reflect the key state of the application issuing the **KeyToggleGet**.

Example:

```
a1=KeyToggleGet (@NUMLOCK)
a2=KeyToggleGet (@CAPSLOCK)
a3=KeyToggleGet (@SCROLLLOCK)
b= strcat(a1," ",a2," ",a3)
Message("NumLock CapsLock ScrollLock", b)
```

See Also:

[KeyToggleSet](#), [SendKey](#)

KeyToggleSet

Sets the state of a toggle key and returns the previous value.

Syntax:

```
KeyToggleSet(@key, value)
```

Parameters:

- (i) @key the toggle key in question. Values may be one of:
- | | |
|--------------------|------------------------|
| @CAPSLOCK | for the CapsLock key |
| @NUMLOCK | for the NumLock key |
| @SCROLLLOCK | for the ScrollLock key |
- (i) value The new value of the toggle key.
- | | |
|-------------|---------------------|
| @OFF | for the unset state |
| @ON | for the set state |

Returns:

- (i) Previous toggle state of the key It may be:
- | | |
|-------------|-----------------|
| @ON | Key was set |
| @OFF | Key was not set |

Use this function to alter the state of one of the toggle keys - the CapsLock, NumLock, and ScrollLock keys.

Note: On DOS based systems, this function will alter the keys state for all applications. For 32 bit Windows based systems, only the application issuing the **KeyToggleSet** command will be affected.

Example:

```
KeyToggleSet(@NUMLOCK, @ON)  
KeyToggleSet(@CAPSLOCK, @ON)  
KeyToggleSet(@SCROLLLOCK, @ON)
```

See Also:

[KeyToggleGet](#), [SendKey](#)

yesyesTRUEyesyesyesWIL JUMPSWILjumpyesyes22/09/98

Table of Contents

[Application](#)
[Branching](#)
[Clipboard](#)
[Close](#)
[Commands](#)
[Display](#)
[DOS](#)
[Functions](#)
[Hide](#)
[Input](#)
[Introduction](#)
[Macro](#)
[Prompt](#)
[Rearrange](#)
[Resize](#)
[Run](#)
[String](#)
[WIL](#)
[Windows](#)
[FileItemize \(file-list\)](#)
[DirItemize \(dir-list\)](#)
[Nicer File Selection](#)
[Nicer Messages](#)
[Hint:](#)
[Hint: Run](#)
[Tutorial Course - Hole #1](#)
[Tutorial Course - Hole #2](#)
[Tutorial Course - Hole #3](#)
[Tutorial Course - Driving Range](#)
[Tutorial Course Hole #5](#)
[Tutorial Course Hole #6](#)
[Tutorial Course Hole #7](#)
[Tutorial Course Hole #8](#)
[Tutorial Course - Hole #9](#)
[Tutorial Course - Hole #10](#)
[Tutorial Course - Hole #11](#)
[Tutorial Course - Water Hazard](#)
[Tutorial Course - Hole #12](#)
[Tutorial Course - Hole #13](#)
[Tutorial Course - Script Stop](#)
[Tutorial Course - Clubhouse](#)

Application

A large computer program dedicated to performing a major task. It can accomplish its work alone. A utility program, in contrast, works on other programs or operating systems.

Branching

A process of first testing for the truth of a condition during execution of a computer program and then going to another location in that program based on the results of the test.

Clipboard

A temporary storage place for text or graphic data. Often used to transfer data within or between application programs. Putting information into the clipboard normally erases the prior contents. WIL has a ClipAppend feature for permitting data to be added.

Close

Remove a window and end the process that owned that window. See [Hide](#).

Commands

Commands perform an action without receiving back a result. For instance, the Yield command relinquishes processing time to other processes without receiving any sort of a report on how successful the operation was.

Display

The action of showing information on a computer screen. The screen, itself, can also be called the "display".

DOS

Disk Operating System for personal microcomputers. Several companies produce proprietary versions of DOS. It forms the operational foundation of the personal computer.

Functions

Functions perform actions and return a value that can be captured in a variable for further use.

Hide

Hide a window from view but keep the associated program running in the background. See [Close](#).

Input

Enter information into an edit dialog box and signal the computer that the information is ready.

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 (shown in *Courier* type), and related functions you may want to **See Also**.

Items marked **{*M}** are available only in menu script implementations.

(i) indicates an integer parameter or return value.

(s) indicates a string parameter or return value.

(f) indicates a floating point parameter or return value.

(t) indicates special type information described in the function's text.

Macro

A sequence of operations in a computer program executed through one command.

Prompt

Request information from the operator of a computer. This is usually done in a dialog box called a "message box".

Rearrange

Change the order or position of windows on a computer screen. This is important for visibility and because the topmost window is the location where user interaction, like data entry, takes place.

Resize

Change the size of a program window on a computer display.

Run

Launch a computer program. Running a program can be simply starting it. In addition, optional information the program needs to operate can be included in a Run command.

String

Strings are lengths of text that contain characters, numbers, or both. Strings are, however, treated in the computer as text. For example, a "string" of numbers cannot be used in arithmetic operations.

WIL

Windows Interface Language: a procedural language optimized for system control operations. Can be used as a macro language common to all application programs running under Microsoft Windows versions 3.1 and above.

Windows

A system of boxes called "windows" used to represent computer programs and operations for controlling them. This system serves as an interface for personal computers. It is a product of Microsoft Corporation.

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:

AskItemList (**title, list, delimiter, sort mode, select mode**)

Displays a list box 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 **AskItemList**, we need to tell it that the delimiter is a space. We do this as follows:

```
files = FileItemize("*.doc *.txt")
afile = AskItemList("Select File to edit", files, " ", @unsorted, @single)
Run("notepad.exe", afile)
```

which produces:



First, we use **FileItemize** to build a list of filenames with DOC and TXT extensions. We assign this list to the variable **files**. Then, we use the **AskItemList** function to build a list box, passing it the variable **files** as its second parameter. The third parameter we use for **AskItemList** is simply a space with quote marks around it; this tells **AskItemList** that the list in variable **files** 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.) Using the fourth parameter set the sort mode to choose how to display the text, sorted or unsorted. The fifth parameter sets the select mode allowing you to choose a single item or multiple items from the list. Finally, we assign the value returned by **AskItemList** to the variable **afile**, 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, **AskItemList** returns a null string. If you want, you can test for this condition:

```
files = FileItemize("*.doc *.txt")
while @TRUE
  afile = AskItemList("Select File to edit", files, " ",
    @unsorted, @single)
  If afile != "" Then break
```

```
    ;break terminates the While structure transferring control to the ;statement
following the endwhile.
endwhile
Run("notepad.exe", afile)
```

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. One way we can do this by first letting the user select the appropriate directory, using the **DirItemize** and **AskItemList** combination:

```
DirChange("C:\")
subdirs = DirItemize("*. *")
targdir = AskItemList("Select dir", subdirs, " ", @sorted, @single) if targdir != ""
then DirChange(targdir)
files = FileItemize("*. *")
afile = AskItemList("Select File to edit", files, " ", @sorted, @single)
Run("notepad.exe", afile)
```

First we change to the root directory. Then we use **DirItemize** to get a list of all the sub-directories off of the root directory. Next, we use **AskItemList** to give us a list box of directories from which to select. Finally, we change to the selected directory, and use **FileItemize** and **AskItemList** 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 improve our file selection logic:

```
DirChange("C:\")
; Directory selection loop
while @TRUE ; Loop forever til break do us part
    dirs = DirItemize("*. *")
    If dirs == "" Then break
    targ = AskItemList("Select dir", dirs, " ", @sorted, @single)
    If targ == "" Then break
    DirChange(targ)
endwhile
;
; File selection loop
while @TRUE ; Loop forever til break do us part
    files = FileItemize("*. *")
    afile = AskItemList("Select File to edit", files, " ",
        @sorted, @single)
    If afile != "" Then break
endwhile
Run("notepad.exe", afile)
```

First of all, we set up a repeating **while** loop. The "While @TRUE" will repeat the loop forever. In the loop itself we use the **break** statement to exit the loop. After we use the **DirItemize** function to try to get a list of the directories at the current level, we test the returned value for a null string. If we have a null string, then we know that the current directory has no sub-directories, and so we proceed to the file selection logic by **breaking** out of the directory selection loop. If, however, **DirItemize** returns a non-blank list, then we know that there is, in fact, at least one sub-directory. In that case, we use **AskItemList** to present the user with a list box of directories. Then, we test the value returned by **AskItemList**. 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 by **breaking** out of the directory selection loop. On the other hand, a non-blank value returned by **AskItemList** indicates that the user has selected a sub-directory from the list box. In that case, we change to the selected directory, and the **endwhile** causes the directory selection loop to be repeated. We continue this process until

either (a) the user selects a directory, or (b) there are no directories left to select. Eventually, we move to the file selection loop.

Nicer File Selection

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

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.

@CRLF

@TAB

@CRLF and @TAB are string constants containing, respectively, a carriage-return line-feed pair and a tab character.

We want to be able to insert a carriage return/line feed combination at the end of each line in our output, and the @CRLF string constant will let us do that.. 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:

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

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

Windows Interface Language Help File 

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

```
Message("", "This is line one%@crlf%This is line two")
```

we will get what we want:

Windows Interface Language Help File 

Note that there is no space after **%@crlf%**; 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).

Windows: **Tutorial Course - Hole #1**

Starting Simple: The goal here is open your editor and use a simple WIL statement to run an application.

Hint: highlighting

Highlight the **Run** statement with your mouse and copy / paste to your editor. This will save you some typing and keep you from getting killed off by typographical errors.

Window

Tutorial Course - Hole #2

The **Run** statement can take additional parameters. Follow the directions to run Notepad and automatically load the autoexec.bat.

Additionally, comments are introduced to demonstrate how descriptions and additional information can easily be added to WIL scripts.

Hint: Run

All WIL functions have a specific format which must be used. The function name is followed by required parameters which are enclosed by parenthesis and quotes. Documentation for a WIL function will look like this;

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.

Here's our line with the parenthesis and quotes.

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

If you need more information before you proceed. Scroll to the top of this topic page, Functions and Parameters.

Windows

Tutorial Course - Hole #3

Now that we have the **Run** statement under our belt, we'll move on.

Along with the original **Run** statement, your script should now have a comment and a line displaying information to the user.



Tutorial Course - Driving Range

Time for a little practice and the opportunity to use your new skills to execute WIL functions, change parameters and create comments.

By commenting out the **Run** statement, you can test the **Display** statement with different time intervals.

Window

Tutorial Course - Hole #5

The WIL Language has numerous variations for accomplishing any given goal.

AskYesNo is one of many ways in which you can display information and ask a user for input.

Window

Tutorial Course - Hole #6

Using variables is an important skill. To help you understand this concept let's set a variable to the **AskYesNo** statement.

If you'd like additional information on variables, look under the WIL Reference section, WIL Elements - Substitution.

Window

Tutorial Course - Hole #7

WIL is capable of making decisions based on pre-specified conditions.

If..else...endif is the first of several sophisticated command structures available for decision making.

Be sure and take a glance at the several different ways to structure the same **If** statement.

Window

Tutorial Course - Hole #8

Throughout the next section we will acquaint you with other types of command structures used to control the flow of a program; **For**, **While**, **Switch** and **GoSub**.

Window Tutorial Course - Hole #9

In this code example **Switch** is used to conditionally select one of several options.

Each case statement block is composed of three parts; a **case** statement followed by a number, a series of one or more statements and the **break** command. If the number behind the **case** statement matches the number behind the **switch** statement, then the **case** block is executed. Once the correct message has been displayed, **break** terminates the case block and transfers control to the **EndSwitch** statement.

FYI, this structure is sometimes called **Select**. It is documented in the WIL Reference under both names; **Switch** and **Select**.

Windows

Tutorial Course - Hole #10

The **While** command is excellent for creating loops. Here it is used to create a timing loop. Now you know three handy structures; **if..else...endif**, **Switch/Select** and **While**.

Window

Tutorial Course - Hole #11

GoSub is like a yo-yo. Your processing starts at the top, jumps down the string to the bottom, rolls around and then zips back to the top where it started.

With **GoSub** it is not necessary to add redundant code throughout your program. A routine can be added once and accessed multiple times.

Tutorial Course - Water Hazard

Windows Interfa

Beware!! The directions may be a little hazardous here.

GoSub needs to know where to send the processing. The label `:dumdedum`, becomes the "street address" of the timing subroutine.

After the **Run** statement add the line

```
GoSub dumdedum
```

Add the **GoSub** label to the very bottom of the script.

```
:dumdedum
```

Cut out the timing loop and move it below the `:dumdedum` label.

Add a Return statement at the bottom of the timing loop.

```
return
```

Windows I

Tutorial Course - Hole #12

The additional **AskLine** prompts the user for extra time. Therefore, a corresponding check must be added for the new time. Place the check after the new **AskLine**.

```
If mins!=0 then GoSub dumdedum
```

Now for the finishing touches. Add a **WinClose** and an **Exit** after the check but before the :dumdedum subroutine. This tightens up the program ensuring nothing is left hanging.

```
WinClose("Solitaire")  
Message("Time's Up", "Get Back to Work!")
```

Windows Int Tutorial Course - Script Stop

The last couple of steps were more difficult. Hopefully, you haven't lost your ball. Check out our code. Is it similar to yours? Differences may indicate concepts which need a second look.

Window

Tutorial Course - Hole #13

The last structure left to discuss is also the most complex. The For command controls the number of times a snippet of code is executed. The variable is incremented each time through until the value is reached.



Remember the infamous question posed, "How many licks does it take to reach the center of the tootsie roll tootsie pop?" In case you've forgotten your Saturday morning cartoons, the answer is "Three, it takes three licks to reach the tootsie roll center of the tootsie roll".

Let's use the **For** command to ask this very important question.

```
center="3"
For licks = 1 to center
  Display(2, "Tootsie Roll Center", "Taking lick # %licks%")
Next
Message("Answer to the Universe", "%licks% licks to reach the tootsie roll center.")
```

Windows Interface Lar Tutorial Course - Clubhouse

Congratulations! You've finished the course.

Let's take a look at the completed script we've been writing, `tutor.wbt`. A few things have been added to polish it up, but the majority of the code should match your example.

