Shell Scripting Toolkit 2.5

© 2004-2006 by Bill Stewart (bstewart@iname.com)

I frequent the **microsoft.public.win2000.cmdprompt.admin** and **alt.msdos.batch.nt** newsgroups, and I have noticed several common questions related to command-line scripting. In response to some of these questions and answers, and also to meet some of my own scripting needs, I have written a small set of tools to assist the shell script author.

All of these programs are copyrighted freeware. Except for one, they are all Win32 console (text-based) programs that run from a command prompt window.

In the remainder of this documentation, the Windows 9*x*/Me (Windows 95, Windows 98, Windows 98 SE, and Windows Millennium edition) platform will be referred to as Win9*x* and the Windows NT platform (e.g. Windows NT/2000/XP/2003/etc.) will be referred to as WinNT. In each case, I'm assuming you're using the default command shell (Command.com on Win9x, Cmd.exe on WinNT).

I'm also assuming that you are familiar with Windows NT shell scripting. For an excellent overview, see Tim Hill's book *Windows NT Shell Scripting* (April 1998, ISBN 1578700477). Even though the book is eight years old as of this writing, it is still an excellent overview of the Cmd.exe command shell on the Windows NT platform, although it has not been updated with the new capabilities added to Cmd.exe in Windows 2000 and later.

For some examples of what you can do with these utilities, see **Sample Applications** on page 15.

License Statement

These programs may be used and/or redistributed freely, provided you do the following:

- Give the author due credit.
- Let the author know how you're using the software.
- Do not modify the executables in any way.

These programs are distributed in the hope that they will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

Acknowledgments

The Win32 programs in this package were developed with Free Pascal (**http://www.freepascal.org/**), and this documentation was written using OpenOffice.org (**http://www.openoffice.org/**). The author would like to gratefully acknowledge the authors of these outstanding free tools.

General Notes

Capturing Standard Output in an Environment Variable

Most of these programs rely on standard input and output, which is a simple and standard way for text-based programs to interact with one another. Of particular use is to read a line from standard input and store it in an environment variable.

On the Win9x platform, you can use the helper program S2V (see page 11). It reads a line (up to 255 characters) of standard input and stores it in an environment variable.

On the WinNT platform, you can accomplish this by using the Cmd.exe For /f command. The general format of the command in a script is as follows:

```
for /f "delims=" %%v in ('command') do set ENVVAR=%%v
```

Note how the *delims* flag does not specify any delimiters; this causes the For command to read each line of the standard output of command into an environment variable without parsing it into tokens. Of course, you can also specify delimiters and tokens if you desire to parse the output differently. Enter the command For /? at a Cmd.exe prompt for more information.

Displaying Help for Commands

For all of the programs except EchoX, you may specify /? as the first command line argument on the command line and the program will display a short synopsis of its use. The help information will also appear if a program requires arguments but none are supplied.

Command-Line Arguments and Quoting

For each program except S2V, if a command-line argument contains spaces or tabs, enclose it in single or double quote marks. This indicates to the program's command-line parser that you are giving it a single argument. Since the quote marks simply delimit the beginning and end of the argument, they are not included in the argument itself. For example:

```
echox "Hello, world"
```

Hello, world is the contents of the argument; since it includes a space character, the quote marks tell the program that the space is part of the argument. The quotes are not themselves part of the argument; they simply tell the command-line parser where the argument starts and ends. You may use either single (') or double quote (") marks. If an argument needs to contain either type of quote mark, you must quote using the other; for example: 'This string contains "double" quotes' or "This string contains 'single' quotes".

Many command-line arguments consist of switches (e.g. -a, -b, -c, etc.) that modify the program's behavior. These may appear in any order on the command line. Some switches themselves require arguments, and the same case as above applies: If an argument to a switch contains spaces or tabs, enclose it in quotes.

After the program processes all of the command-line switches, the rest of the command line arguments (if any) are processed separately. If a program needs non-switch command-line arguments, they must appear *after* all switches on the command line. Non-switch arguments also need quotes if they contain spaces or tabs.

Reserved Shell Characters on Windows NT

Cmd.exe, has a set of seven special characters that have special meaning to the shell. These are:

```
( ) < > ^ & |
```

To use any of these characters literally in a shell command, you must use the escape character (^) before any of these characters (including the escape character itself). In some cases, it is useful to echo or parse the output of a command without worrying whether it contains any of the reserved characters. The ShellEsc utility (see page 12) is provided to work around this problem; it will perform the necessary "escaping" for the shell.

Console Applications on Win9x

Win32 console (text-based) programs are poorly behaved on Win9*x* because they are handled by a separate program called Conagent.exe. Many of the console APIs are buggy or implemented incompletely. In many cases, using the Win32 console functions to write colored text to standard output on Win9*x* does not function properly (particularly on the last line of the console window). Since the Win9*x* platform is obsolete and no longer being updated, Microsoft will not be fixing these bugs.

Exit Codes and the If Command

You must exercise care when checking the exit code of a program using If Errorlevel. In general, it is important to check exit codes in **descending order**, because "If Errorlevel *n*" really means "If Errorlevel >= *n*". This means that If Errorlevel 0 is always true.

To check for a specific exit code, you can use the following "double-if" construct:

```
if errorlevel n if not errorlevel n+1 ...
```

For example, to check if the exit code is exactly zero:

```
if errorlevel 0 if not errorlevel 1 ...
```

You can avoid this in Cmd.exe scripting by using "If Errorlevel EQU *n*", as the EQU operator performs an exact comparison. (You can also use the %ERRORLEVEL% variable.)

CCase

Description

Reads each line of standard input, converts it to upper or lowercase, and writes it to standard output.

Usage

ccase [**-u** | **-l**]

Arguments

-u Converts to uppercase (default).
-l Converts to lowercase.

Examples

1. Echo a string in uppercase:

   ```
   echo Hello, world | ccase
   ```

2. Uppercase an environment variable, Cmd.exe:

   ```
   for /f "delims=" %%v in ('echo %VAR% ^| ccase') do set VAR=%%v
   ```

   Note the use of the escape character (^) in the command string.

3. Uppercase an environment variable, Win9*x*:

   ```
   echo %VAR% | ccase | s2v VAR
   ```

ColorX

Description

Writes the current screen colors to standard output, sets the color for text subsequently written to standard output, or lists a table of colors. Setting colors does not work on Win9*x* due to console API issues.

Usage

colorx [**-c** *color* | [**-f** *fg*] [**-b** *bg*]]
colorx -l

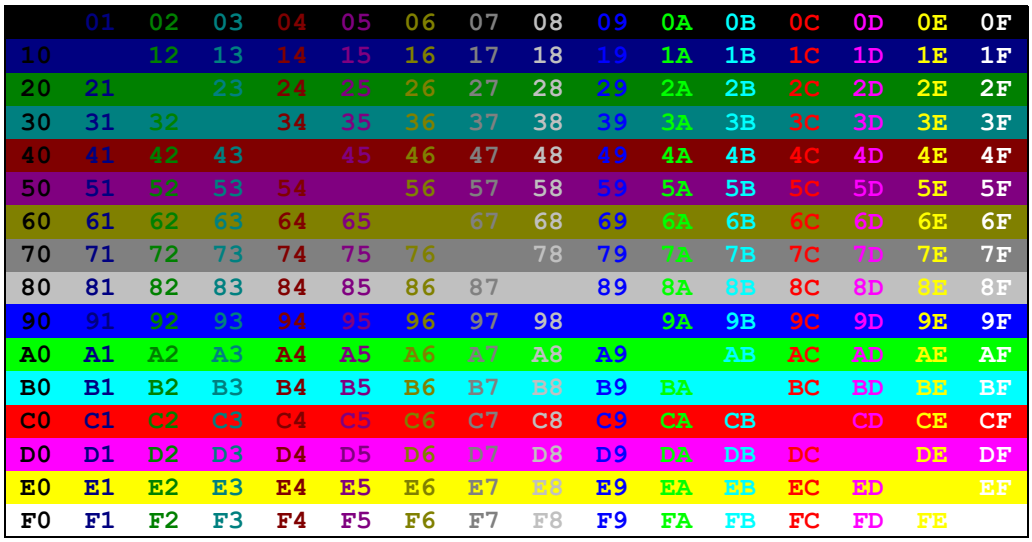
Arguments

color
Hexadecimal value (two hex digits) specifying both the foreground and background colors (01 through FE). The first hex digit is the background color, and the second hex digit is the foreground color.
fg Hex value (single hex digit) specifying a foreground color (0 through F).
bg Same as above, but specify a background color.
-l Lists all colors in a table (see below).

If you attempt to specify colors such that the foreground and background colors match, ColorX will ignore the request and return an exit code of 1.

Color Table

If you specify **-l**, ColorX will display a color table with color values in hexadecimal:



Note that each row represents a background color, and each column represents a foreground color. The first digit in each number is the background color, and the second digit is the foreground color.

Examples

1. Set the screen colors to white on blue:

   ```
   colorx -c 1F
   ```

2. Sets the background color to red:

   ```
   colorx -b 4
   ```

3. Cmd.exe script sample: Store the current colors, output a Dir command in bright white on cyan, and restore the colors again:

   ```
   for /f %%a in ('colorx') do set COLOR=%%a
   colorx -c 3F
   dir /w
   colorx -c %COLOR%
   ```

DateX

Description

Echoes to standard output the current date, a specified date, or an offset from the current or specified date in a variety of formats.

Usage

datex [**-d** *date*] [**-f** *format*] [**-i** *num*] [**-o** *offset*] [**-t**]

Arguments

date

Specifies a particular date/time. Specify year/month/day order. If you include the time, place it after the date in hours:minutes:seconds order (minutes and seconds are optional).

format

Specifies the date output format (see table below).

num
Specifies an integer representing a date (inverse of **-t**).
offset
Specifies a number-of-days difference. Can be a negative number.
-t Outputs the date as an integer.

Without arguments, DateX outputs the current date and time in the following format: yyyy/mm/dd hh:nn:ss. You can control the formatting with **-f** (see table below).

Date Formats

Date format specifiers are listed in the below table. They are similar to the options Windows provides in the Control Panel for controlling date and time formats.

Date Formatting Characters			
y	Year (two digits)	**d**	Day of month
yy	Year (two digits)	**dd**	Day of month (leading zero)
yyyy	Year (four digits)	**ddd**	Day of week (abbreviation)
m	Month	**dddd**	Day of week (full)
mm	Month (leading zero)		
mmm	Month (abbreviation)	**ddddd**	ShortDateFormat: **mm/dd/yy**
mmmm	Month (full)	**dddddd**	LongDateFormat: **mmmm d, yyyy**

Time Formatting Characters			
h	Hour	**a/p**	12-hour clock with a or p
hh	Hour (leading zero)	**am/pm**	12-hour clock with am or pm
n	Minute		
nn	Minute (leading zero)	**t**	ShortTimeFormat: **hh:nn**
s	Second	**tt**	LongTimeFormat: **hn:nn:ss**
ss	Second (leading zero)		

Combination Formats	
c	ShortDateFormat + " " + ShortTimeFormat

If you need spaces within a date format, enclose the entire format in quotes.

Examples

1. Output the current date similar to Cmd.exe's Date /t command:

   ```
   datex -f "ddd mm/dd/yyyy"
   ```

2. Output yesterday's date:

   ```
   datex -o -1
   ```

3. Output the current date and time as an integer:

   ```
   datex -t
   ```

   This is useful for comparing the date and time from a file (see FInfo on page 9) with the current date and time.

4. Display the date and time for a specified integer:

```
datex -i 2162688
```

The output of this command is:

```
1980/01/01 00:00:00
```

DriveX

Description

Returns the drive type for a specified drive as an exit code, or lists drive(s) of a specified type to standard output. It can list the current set of drives and types.

Usage

drivex [**-d** *drive*] [**-l** *type*] [**-v**]

Arguments

drive

Return an exit code representing the type of the specified drive (see the list of types below). The drive letter may be specified with or without the colon (:). Specify 0 (zero) to indicate the current drive. An invalid drive designation will return an exit code of 1.

type

Drives of the specified type will be listed to standard output. Drive types are 2 through 6, as follows:

2	Removable
3	Fixed
4	Remove (network)
5	CD-ROM
6	RAM disk

-v List available drives and types in a table format.

Examples

1. List fixed drives to standard output:

```
drivex -l 3
```

Sample output:

```
C
D
E
```

2. Return drive type of current drive as an exit code. Sample code:

```
drivex -d 0
if errorlevel 3 if not errorlevel 4 goto :HD
```

In this example, the script will branch to the :HD label if the exit code is exactly 3 (fixed drive). See **Exit Codes and the If Command** on page 3 for a description of this “double-if” construction.

3. List drives on the machine:

```
drivex -v
```

Sample output:

```
Drive  Type  Description
----------------------------
A:     2     Removable
C:     3     Fixed
D:     5     CD-ROM
E:     4     Remote (network)
R:     6     RAM disk
```

EchoX

Description

Echoes text to standard output in a selectable color. Also supports several escape sequences, a format width, and alignment. Writing colored text to standard output is very problematic on Win9*x* due to console API issues; in particular, writing a colored string on the last line of the console window will not reset the previous colors properly.

Usage

echox [**-c** *color* | [**-f** *fg*] [**-b** *bg*]] [**-n**] [**-w** *width* [**-r** | **-e**]] *message*
echox -l

Arguments

color
: Hexadecimal value (two hex digits) specifying both the foreground and background colors (01 through FE). The first hex digit is the background color, and the second hex digit is the foreground color.

fg Hex value (single hex digit) specifying a foreground color (0 through F).

bg Same as above, but specify a background color.

-n Do not skip to the next line (no carriage return/line feed).

width
: The text should be formatted to exactly *width* characters. If the text is longer than the specified width, it is truncated; if it is shorter, then it is padded with spaces on the right (left-justified), unless you specify **-r** or **-e**.

-r Right-justify the message with respect to *width*.

-e Centers the message with respect to *width*.

message
: Specifies the text to write to standard output. Enclose it in quotes if it contains spaces or tabs.

-l Lists all colors in a table (See **Color Table** on page 5).

If you attempt to specify colors such that the foreground and background colors match, EchoX will ignore the request and return an exit code of 1.

The string of text can contain the following “escape” sequences:

`~n`	Carriage return/line feed (move to next line)
`~r`	Carriage return (move cursor to beginning of current line)
`~t`	Tab character

If a ~ character is followed by any of n, r, or t, it will be interpreted as an escape sequence per the above table. You can bypass this interpretation by using ~~. Also, these escape sequences are ignored if you specify a format width (**-w**).

Examples

1. Display the string Hello, world in white on red:

```
echox -c 4F "Hello, world"
```

2. Display a centered message in bright cyan (assuming the console window has 80 columns):

```
echox -f B -w 80 -e "Centered Message"
```

3. Use the Cmd.exe For command to count to 100% on a single line, and display a message when finished:

```
for /l %%i in (1,1,100) do echox -n "%%i%% complete~r"
echox "Done          "
```

   In this example, the Cmd.exe For command will count from 1 to 100, and EchoX will print the iterator variable followed by a carriage return. After the iterations are complete, EchoX prints *Done* followed by spaces to overwrite the previous text.

FInfo

Description

Echoes selectable information about one or more files on standard output, including last modification date/time, filename or path (or both), and file size.

Usage

finfo [**-d** *format*] [**-f** | **-p** | **-n**] [**-t**] [**-z** [**-s**]] *filename* [...]

Arguments

format
: FInfo should output the last modification date/time in the specified format. (See **Date Formats** on page 6 for a list of formats.) Use quotes if the format contains spaces.

-f Output full pathnames.
-p Output paths (subdirectory names) only.
-n Output filenames only.
-t Output last modification date/time as an integer.
-z Output file size(s) (up to 2GB per file).
-s Include thousands separators in the size(s).

filename
: One or more files. You may use wildcards and repeat as many filenames on the command line as needed. Remember to use quotes if any filenames contain spaces.

If you specify more than one item to report on, the output will appear in the following order:

Formatted date/time, integer date/time, file size, filename or pathname

Each item will be separated by a single space.

Examples

1. Output the filename and size for all *.exe files in the current directory:

```
finfo -n -z -s *.exe
```

2. Output the full pathname for the file readme.txt:

```
finfo -f readme.txt
```

IfX

Description

Performs a case-insensitive comparison of two strings or a numeric comparison of two integers, and returns an exit code (not needed in Cmd.exe).

Usage

ifx *arg1 arg2*

Arguments

arg1 and *arg2* are the two arguments to compare. They can be either integers or strings. If both arguments are composed of all numeric digits, then IfX performs a numeric comparison; otherwise, it performs a case-insensitive string comparison. If you are comparing strings that contain spaces or tabs, enclose both arguments in quotes.

Exit Codes

0 The two arguments are equal.
1 Argument 1 is less than argument 2.
2 Argument 1 is greater than argument 2.

Examples

1. Compare two strings:

   ```
   ifx "hello, world" "Hello, World"
   ```

   This command will return an exit code of 0 because IfX is not case-sensitive. Note that IfX is not needed in WinNT, because Cmd.exe's If /i command supports case-insensitive comparisons:

   ```
   if /i "hello, world"=="Hello, World" ...
   ```

2. Compare two numbers:

   ```
   ifx 1023 2047
   if errorlevel 2 goto :GTR
   if errorlevel 1 goto :LSS
   echo The two numbers are equal
   ...
   ```

   Note that the If Errorlevel tests are performed in descending order (see **Exit Codes and the If Command** on page 3 for a discussion of why this is important).

LineX

Description

Counts the number of lines in standard input, print a specific line, or a specified number of lines from the beginning or end of standard input.

Usage

linex -c | **-n** | **-l** *n* [**-e** *n*] | **-h** *n* | **-t** *n* | **-r**

Arguments

-c Outputs the number of lines in standard input.
-n Outputs the contents of standard input with line numbers.
-l Outputs line *n* from standard input.
-e Also outputs the next *n* lines.
-h (Head) Outputs the first *n* lines from standard input.

-t (Tail) Outputs the last *n* lines from standard input.
-r Outputs in reverse order. Can only be used alone or with **-n**.

Examples

1. Count the number of files in the current directory:

   ```
   dir /a-d /b | linex -c
   ```

2. Capture the output of a single ping:

   ```
   ping -n 1 192.168.0.1 | linex -l 7
   ```

   The Windows Ping program adds superfluous carriage returns to its output. LineX sees these extra carriage returns as separate lines, so as a result it sees the output of the ping as being on line 7. To see the actual output of the ping command as seen by LineX:

   ```
   ping -n 1 192.168.0.1 | linex -l
   ```

3. Output the names of the three most recently modified in the current directory:

   ```
   dir /b /o-d | linex -h 3
   ```

   You could also use:

   ```
   dir /b /o-d | linex -l 1 -e 2
   ```

   This command outputs line one (**-l 1**) and also the next two lines (-**e 2**).

Notes

LineX reads all of standard input into memory, so it might perform slowly on inputs that exceed the computer's memory.

S2V

Description

A real-mode MS-DOS executable that reads the first line of standard input and stores it in an environment variable. Only for Win9*x*; does not work (and is not needed) on WinNT.

Usage

s2v *varname*

Arguments

varname specifies the name of the environment variable.

Examples

1. Store the name of the newest *.exe file in the current directory in the variable NE:

   ```
   dir /a-d /b /o-d *.exe | s2v NE
   ```

2. Store the last modification date/time of the file ccase.exe in the variable DT:

   ```
   finfo -t ccase.exe | s2v DT
   ```

ShellEsc

Description

For each line of standard input, inserts a shell escape character (^) before each reserved shell character, and writes the line to standard output. This is useful when you are capturing standard input and want to be able to use it without worrying about special shell characters. The reserved shell characters are:

```
( ) < > ^ & |
```

Note that ShellEsc is not useful on Win9*x* since Command.com only has three reserved characters (<, >, and |), and the only way to "escape" them is by quoting the entire string.

Usage

shellesc

Examples

1. Escapes the output of the Dir command, and writes it to an output file:

   ```
   dir | shellesc > out
   ```

   If you then open the output file in a text editor, you'll notice that each of the reserved shell characters is prefixed by the escape character (^).

2. Cmd.exe: Script sample to produce a time-stamped ping:

   ```
   for /f "delims=" %%d in ('datex') do set STAMP=%%d
   for /f "delims=" %%p in ('ping -n 1 192.168.200.200 ^| linex -l 7 ^|
      shellesc') do set OUT=%%p
   echo %STAMP% %OUT%
   ```

   Note that we need to escape the pipe symbols (|) in the command.

SleepX

Description

Pauses a script for specified delay period.

Usage

sleepx [**-k**] [**-m**] [**-p** *prompt*] *delay*

Arguments

-k Abort the delay and return a non-zero exit code if a key was pressed.
-m The delay is specified in millseconds, rather than seconds.
prompt
Displays the specified prompt. Use quotes if it contains spaces or tabs.

Examples

1. Pause a script for 5 seconds:

   ```
   sleepx 5
   ```

2. Pause a script for 30 seconds, with a prompt:

   ```
   sleepx -p "Please wait for 30 seconds..." 30
   ```

3. Script sample that pauses for 10 seconds, and allows a keystroke to interrupt the pause:

```
sleepx -p "Wait for 10 seconds, or press a key..." -k 10
if errorlevel 1 goto :KEYPRESSED
```

Str

Description

Counts characters in a string, echoes a string in upper or lowercase, and searches for one string in another.

Usage

str [**-c** | **-l** | **-u**] *string*
str [**-i**] **-p** *string1 string2*

Arguments

-c Echoes the number of characters in *string*.
-l Echoes *string* in lowercase.
-u Echoes *string* in uppercase.
-p Echoes the starting position of *string2* in *string1*. If *string2* is not found in *string1*, return -1.
-i Perform a case-insensitive comparison (**-i** must appear before **-p** on the command line).

Remember to enclose the string in quotes if it contains spaces.

Examples

1. Cmd.exe: Check %1 for wildcard characters (* or ?):

```
for /f %%p in ('str -p %1 *') do set POS=%%p
if %POS% GEQ 0 echo %1 contains the * character
for /f %%p in ('str -p %1 ?') do set POS=%%p
if %POS% GEQ 0 echo %1 contains the ? character
```

2. Cmd.exe: Check if %1 is at least eight characters:

```
for /f %%l in ('str -c %1') do set LEN=%%l
if %LEN% GEQ 8 (echo %1 is at least 8 characters) else (echo %1 too short)
```

Tee

Description

Writes each line from standard input to a specified file and to standard output simultaneously. Useful when you want to write information to standard output and to a file at the same time.

Usage

tee [**-a**] *filename*

Arguments

-a Append to the file if it exists (don't overwrite it).

Examples

1. Display a directory on the screen and to the file list.txt:

```
dir | tee list.txt
```

If the file list.txt already exists, Tee will overwrite it unless you specify **-a**.

2. Echo an error message on the screen and to a log file simultaneously:

```
echo Error! | tee -a errors.log
```

   In this case, the string Error! will appear on standard output, and it will also be appended to the file errors.log.

Notes

If Tee can't open the specified file, it will output a message to standard error:

```
Error number opening filename
```

The number will correspond to the Win32 error code; to see the textual representation of the error message in Windows 2000 or later, use the command Net Helpmsg *number*. Tee's exit code will match the error number.

In case of an error, Tee will still write standard input to standard output, and it will write the error message to standard error after it is finished.

TempName

Description

Generates a unique temporary file or directory name.

Usage

tempname [**-p** *path*] [**-e** *ext*] [**-d**]

Arguments

path
: The path for the temporary file or directory name. Enclose it in quotes if it contains spaces. If you don't specify a path, TempName will use the directory specified by the TEMP environment variable.

ext
: The extension for the file or directory. .TMP is the default extension. To omit an extension, specify a blank string after the **-e** option (e.g., **-e ""**).

-d The temporary name should be a directory rather than a file.

Examples

1. Cmd.exe: Create a temporary filename in the TEMP directory, and put it in the TEMPNAME variable.

```
for /f "delims=" %%t in ('tempname') do set TEMPNAME=%%t
```

2. Win9*x*: Create a temporary filename in the TEMP directory and put it in the TN variable:

```
tempname | s2v TN
```

Notes

TempName generates a pseudo-random hex number for a filename and verifies that the file doesn't exist before writing it to standard output. If you use the **-d** option, TempName verifies that the generated name isn't an existing directory (rather than a file).

Sample Applications

Except where otherwise noted, all of these examples are for Cmd.exe. Also, the line numbers in the script samples are for reference only; they are not a part of the actual scripts. Note that these scripts are merely short examples; there may be faster and/or more efficient ways to do the same things.

1. List files in the current directory modified in the last 24 hours.

   You can combine the DateX and FInfo utilities. Here is a sample Cmd.exe script:

```
1.  @echo off
2.  setlocal enableextensions
3.  for /f %%d in ('datex -t -o -1') do set DT=%%d
4.  for /f "tokens=1,2*" %%e in ('finfo -t -n *') do call :SHOW %%e "%%f"
5.  endlocal & goto :EOF
6.
7.  :SHOW
8.  setlocal
9.  set FN=%2
10. for /f "delims=" %%f in ('echo %FN% ^| shellesc') do SET FN=%%f
11. set FN=%FN:"=%
12. if %1 GEQ %DT% echo %FN%
13. endlocal & goto :EOF
```

   On line 3, we capture yesterday's date as a number (**-t -o -1**) to the variable DT. In line 4, we call the SHOW subroutine with the file's time (first token, %%e) and the filename (remaining tokens, %%f) as parameters. Note that we quote the filename in case it contains spaces.

   Inside the SHOW subroutine on line 9, we store the second parameter (%2) into the FN variable. on line 10, we filter the variable through ShellEsc in case a filename contains any of the reserved shell characters; then we remove the quotes from the filename on line 11. On line 12, we compare the file's time (%1) with the DT variable we created in line 3. If it's greater or equal, then echo the filename (line 12).

2. Retrieve a file's size into an environment variable.

   Cmd.exe in Windows 2000 and later has this capability via the For command:

```
for %%v in (filename) do set VARNAME=%~zv
```

   Cmd.exe in Windows NT 4.0 doesn't have the ~z modifier, so we can use FInfo instead:

```
for /f %%s in ('finfo -z finfo.exe') do set SIZE=%%s
```

   On Win9x, we can use FInfo and S2V:

```
finfo -z finfo.exe | s2v SIZE
```

3. Find a file on all local hard drives.

   We'll use DriveX to obtain a list of local hard drives, and use the Dir command to redirect a list of files from all hard drives to a temporary file. Then, we can parse the temporary file and use FInfo to output the date/time/size and full pathname of each file. Here's the script:

```
1.  @echo off
2.  if {%1}=={} goto :HELP
3.  if {%1}=={/?} goto :HELP
4.  goto :START
5.
```

```
6.  :HELP
7.  echo Usage: %~n0 filename
8.  echo.
9.  echo Searches for the specified file(s) on all fixed drives.
10. goto :EOF
11.
12. :START
13. setlocal enableextensions
14. set FN=%1
15. set FN=%FN:"=%
16. for /f "delims=" %%t in ('tempname') do set TF=%%t
17. set FS=0
18. for /f %%d in ('drivex -l 3') do dir /b /s "%%d:\%FN%" >> "%TF%" 2> NUL
19. for /f "delims=" %%f in ('type "%TF%"') do call :SHOW "%%f"
20. for /f %%c in ('linex -c ^< "%TF%"') do echox "~n%%c file(s) found"
21. echo %FS% byte(s)
22. if exist "%TF%" del "%TF%"
23. endlocal & goto :EOF
24.
25. :SHOW
26. setlocal
27. set FN=%1
28. for /f "delims=" %%f in ('echo %FN% ^| shellesc') do set FN=%%f
29. set FN=%FN:"=%
30. for /f "tokens=1,2,3*" %%a in ('finfo -d "yy/mm/dd hh:nn" -f -z "%FN%"')
    do (
31.   echox -n "%%a %%b  " & echox -n -w 12 -r %%c & echox "  %%d"
32.   set SIZE=%%c
33. )
34. endlocal & set /a FS+=%SIZE% & goto :EOF
```

In this script, we require a single command-line parameter: A filename pattern to search for. Lines 2 and 3 use curly braces to avoid quote problems; if we don't enter a parameter, or if the first parameter is /?, then display help information. Otherwise, branch to the START subroutine.

In the START subroutine, we store the command-line parameter (%1) into the FN variable. We also use TempName to generate a temporary filename and put it in the TF variable. On line 17, we initialize the FS variable (file sizes) to 0.

On line 18, we use the For command to to parse each line of the DriveX output, and we use the iterator variable (%%d) to specify the drive letter to use for the Dir command. We redirect the output to the temporary file. The end of the command line (2> NUL) tells the shell to redirect standard error output to the null device (don't show it); this is in case the Dir command reports any errors to standard error (e.g. "File Not Found").

On line 19, we parse each line of the temporary file, and pass each line as a parameter to the SHOW subroutine.

Inside the SHOW subroutine, we store the first parameter (%1) in the FN variable (line 27). Then we pass the variable's contents through ShellEsc in order to escape any reserved shell characters that the filename might contain. Then we strip off the quotes (line 29).

On line 30, we parse the output of the FInfo command (note that it's too long to fit in this document's margins, but it should all be on one line). Remember that each line of the temporary file we created on line 18 contains a single filename, which might contain spaces, so we quote the filename when using FInfo. On line 31, we use EchoX twice with the **-n** argument to keep the outputs on the same line. The first EchoX command prints the date and time; the second EchoX command prints the file's size (right-justified with a maximum width of 12), and the last EchoX command prints the file's name. We then store the file's size in the SIZE variable (line 32).

Line 34 takes advantage of the fact that the shell expands environment variables on the line before executing the commands, so it increments the FS variable we created on line 17 with the size of the file we just displayed before exiting the subroutine.

After finishing the SHOW subroutine, the script returns to line 20, where we count the lines in the temporary file and use EchoX to display the count (we use ~n to separate the last found file from the count with a new line). Line 21 displays the cumulative sizes of the files, and line 22 deletes the temporary file.

This script is provided in the distribution as WhereIs.cmd.

4. Compare the last modification date of two files.

   Here is a sample Cmd.exe script:

```
1.  @echo off
2.  setlocal enableextensions
3.  set F1=TEE.EXE
4.  set F2=CCASE.EXE
5.  for /f %%a in ('finfo -t %F1%') do set F1M=%%a
6.  for /f %%a in ('finfo -t %F2%') do set F2M=%%a
7.  if %F1M% EQU %F2M% (
8.    echo Date/time stamps for %F1% and %F2% match
9.  ) else (
10.   if %F1M% LSS %F2M% (
11.     echo %F1% is older than %F2%
12.   ) else (
13.     echo %F1% is newer than %F2%
14.   )
15. )
16. endlocal
```

   Below is a sample script that duplicates the above functionality on Win9*x*:

```
1.  @echo off
2.  set F1=TEE.EXE
3.  set F2=CCASE.EXE
4.  finfo -t %F1% | s2v F1M
5.  finfo -t %F2% | s2v F2M
6.  ifx %F1M% %F2M%
7.  if errorlevel 2 goto :GTR
8.  if errorlevel 1 goto :LSS
9.  echo Date/time stamps for %F1% and %F2% match
10. goto :END
11. :LSS
12. echo %F1% is older than %F2%
13. goto :END
14. :GTR
15. echo %F1% is newer than %F2%
16. :END
17. set F1=
18. set F2=
19. set F1M=
20. set F2M=
```

   This script uses IfX on line 6 to compare the files' times because Command.com's If command does not provide a way to perform a numeric comparison.