Using Compress and Expand

You can use the Microsoft File Compression Utility (Compress) and Microsoft File Expansion Utility (Expand) to prepare your application files for distribution or as part of the installation process for your application. Compress reduces the size of files so that more files can fit on a disk. Expand restores compressed files to their original size. Both Compress and Expand are run from the operating-system command line.

Compress Utility

You can use the Compress utility (COMPRESS.EXE) to reduce one or more files to 25 to 45 percent of their original size.

To use Compress, use the following syntax:

COMPRESS [**/?**] [[**/r**] *source destination*

Compress provides the following command-line options:

/?
Displays information on using Compress.

/r
Specifies that compressed files should be renamed.

-z
Specifies the MS-ZIP compression used for files distributed by Microsoft.

source
Specifies the source filename. The filename can include a drive letter, a directory path, or both; it can also contain wildcards.

destination
Specifies the destination. This parameter can consist of a directory (with optional drive letter), a filename, or any combination of the two.

If the *source* parameter contains wildcards and the *destination* parameter does not specify a directory, the **/r** option must be used.

If the *destination* parameter does not contain a filename, the filename specified in the source option will be used when the file is copied to the location specified by the *destination* option.

Expand Utility

The Expand utility (EXPAND.EXE) restores compressed files to their original size.

To use Expand, use the following syntax:

EXPAND [**/?**] [[**/r**] *source destination*

Expand provides the following command-line options:

/?
Displays information on using Expand.

/r
Specifies that expanded files should be renamed.

Unlike the Compress utility (COMPRESS.EXE), Expand does not have a -z switch. Expand determines the compression type used and automatically decompresses it. You can also use **VerInstallFile** to expand MS_ZIP files.

source
Specifies the source filename. The filename can include a drive letter, a directory path, or both; it can also contain wildcards.

destination
Specifies the destination. This parameter can consist of a directory (with optional drive letter), a filename, or any combination of the two.

If the *source* parameter contains wildcards and the *destination* parameter does not specify a directory, the file must be renamed using the /r option.

If the *destination* parameter does not contain a filename, the filename specified in the source option will be used when the file is copied to the location specified by the destination parameter.

The following example shows how to expand all of the files on drive A and copy them to a directory on drive C.

```
expand a:*.* c:\mydir
```

Command Window Interface

This section describes the commands that you can run in the Command window. It describes the commands that apply to debugging user-mode applications, as well as those that apply to debugging kernel-mode drivers.

You can use standard editing keys when you enter a command. Use the UP ARROW and DOWN ARROW keys to retrieve previous commands. Edit the current command line with the BACKSPACE, DELETE, INSERT, and LEFT ARROW and RIGHT ARROW keys. Press the ESC key to clear the current line.

To copy text from a previous line in the Command window, highlight the text by holding down the left mouse button and dragging the mouse pointer. Click the right mouse button to copy the highlighted text to the Clipboard. Click the right button again to paste the text into the current command line. You can use the same technique to copy text from other windows to the Command window. You can also press CTRL+INS to copy highlighted text to the Clipboard and press SHIFT+INS to paste text from the Clipboard to the insertion point.

Command Conventions

Many commands in this section use a specific range syntax or process and thread syntax.

Range Syntax

You can specify a range as either of the following:

- *startaddress endaddress*
- *startaddress* `L` *length*

With the first method, you specify the start and end addresses of the range. With the second method, you specify the start address, the letter "L," and the number of data items in the range.

Note With the **DC** (Display Memory) and the **U** (Unassemble) commands, you can also use the following to dump *line* instructions starting at *startaddress*:

startaddress **l** *line*

Process and Thread Syntax

You can specify a process in the following ways:

Symbol	Description
|.	The current process.
|number	The process number.
|*	All processes.

You can specify a thread in the following ways:

Symbol	Description
~.	The current thread.
~number	The thread ID.
~*	All threads.

If you do not specify a process or thread with a command, the command will act on the current process or thread. To set the current process or thread, enter a process or thread specifier alone on a command line.

To display the state of the processes or threads, type pipeline (|) or tilde (~) alone on the command line.

In kernel debugging, threads represent the physical processors of the target computer: a two-processor computer would have two threads, one for each processor. The single process represents the target computer itself.

Command List

The following table lists the commands that you can run in the Command window:

Command	Definition
*	Comment
;	Command Separator
?	Evaluate Expression
!	User Extension DLL
|	Display Process State
~	Display Thread State
#	Search for Disassembly Pattern
%	Change Context
.attach	Attach to Process
.cache	Cache Size
.list	Display Source/Assembly Listing
.logappend	Append Log File
.logclose	Close Log File
.logopen	Open Log File
.reboot	Reboot Target Machine
.reload	Reload Symbols
BC	Breakpoint Clear
BD	Breakpoint Disable
BE	Breakpoint Enable
BL	Breakpoint List
BP	Set Breakpoint
C	Compare Memory
DA, DB, DC, DD, DI, DS, DT, DU, DW	Display Memory
EA, EB, ED, EI, ES, ET, EU, EW	Enter Values
F	Freeze Thread
FIA, FIB, FID, FII, FIS, FIT, FIU, FIW	Fill Memory
FR	Floating-Point Registers
G	Go
GH	Go-- Exception Handled
GN	Go-- Exception not Handled
K, KB, KN, KS, KV	Display Stack Backtrace
L	Restart Debuggee
LM	List Loaded Modules
LN	List Nearest Symbols
M	Move Memory

N	Set Number Base
P	Program Step
Q	Quit WinDbg
R	Registers
REMOTE	Start Remote Server
RT	Register Display Toggle
S+, S-	Set Source/Assembly Mode
SA, SB, SD, SI, SS, ST, SU, SW	Search Memory
SEB, SEW	Set Error Break, Set Error Warning
SX, SXD, SXE, SXN	Set Exceptions
T	Trace
U	Unassemble
X	Examine Symbols
Z	Unfreeze Thread

* (Comment)

Syntax

*

Description

If this character is at the start of a line, then the rest of the line is treated as a comment.

; (Command Separator)

Syntax

;

Description

Use this character to separate multiple commands on a single line. Commands are executed sequentially from left to right. All commands on a single line refer to the current thread, unless otherwise specified. If a command causes the thread to execute, the remaining commands on the line will be deferred until that thread stops on a debug event.

? (Evaluate Expression)

Syntax

? *expression* [**,***format*]

Parameters

expression
Expression to be evaluated with the expression evaluator.

format
C-style format characters. For example, to display a long int, use **Ll** as the *format* specifier. Do not use percent signs (**%**).

Description

Evaluates and displays the value of the expression or symbol in the context of the current thread and process.

Note The C++ expression evaluator cannot calculate expressions that use the conditional operator (**?:**).

! (User Extension DLL)

Syntax

! [*filename***.**]*function* [*string*]

Parameters

filename

Name of the DLL to call (without the .DLL extension), followed by a period (.). The default filename is NTSDEXTS.DLL.

In kernel debugging, the default extension DLL depends upon the target platform:

Target platform	**Default extension DLL**
x86	KDEXTX86.DLL
MIPS	KDEXTMIP.DLL
ALPHA	KDEXTALP.DLL

function

Name of the DLL function to call.

string

String to pass to the DLL function.

Description

Calls a DLL function from WinDbg. Use this command to call special routines while debugging. For a complete list of the built-in extension commands, enter **!?** at the WinDbg prompt.

(Search for Disassembly Pattern)

Syntax

[pattern] [address]

Parameters

pattern
The pattern to search for in the Disassembly window.

address
The address after which to search.

Description

Displays the first line of assembly code after *address* that contains the specified pattern.

% (Change Context)

Syntax

%[*frame*]

Parameters

frame
The number of the frame on the call stack to change context to. The current procedure is number 0 on the stack.

Description

Changes the context to the specified frame. Updates the Locals window with the variables of the new frame.

Use the **KN** command to list the call stack with the frame numbers.

.ATTACH (Attach to Process)

Syntax

.ATTACH *process*

Parameter

process
Process ID of task to debug.

Description

Starts debugging the process specified by *process*. You can use PView to find process IDs of running programs or use the **Attach** command from the Run menu to select a task.

This command is similar to the **/P** command-line option.

.CACHE (Cache Size)

Syntax

.CACHE [*cachesize*]

Parameter

cachesize
The size of the kernel debugging cache, in MB. If you do not specify *cachesize*, the command displays the status of the cache. If you issue **.cache 0**, the cache is disabled.

Description

Sets the size of the cache for WinDbg KD to use for memory values. Normal operations, such as single-stepping or the **G** command, invalidate the cache. If WinDbg KD has frozen the target computer, but hardware on the target can change memory (for example, through shared memory or DMA), you must disable the cache to see the memory changes.

.LIST (Display Source/Assembly Listing)

Syntax

.LIST [*address*]

Parameter

address
The address at which to start displaying the listing.

Description

Displays source-code lines and their corresponding assembly code, starting at *address*. If you do not specify *address*, the display starts from the current address.

.LOGAPPEND (Append Log file)

Syntax

.LOGAPPEND [*filename*]

Parameter

filename
Filename of the log file. The default filename is WINDBG.LOG.

Description

Sends a copy of the events and commands from the Command window to the given log file. If you are already logging, that log file will be closed. If *filename* already exists, new information will be appended to it.

.LOGCLOSE

Syntax

.LOGCLOSE

Description

Closes any currently open log file.

.LOGOPEN (Open Log File)

Syntax

.LOGOPEN [*filename*]

Parameter

filename
Filename of the log file. The default filename is WINDBG.LOG.

Description

Sends a copy of the events and commands from the Command window to the given log file. If you are already logging, the current log file will be closed. If *filename* already exists, its contents will be overwritten.

.REBOOT (Reboot Target Machine)

Syntax

.REBOOT

Description

Reboots the target computer (WinDbg KD only).

.RELOAD (Reload Symbols)

Syntax

.RELOAD [*modulename*]

Parameter

modulename
The name of the module whose symbols you want to reload.

Description

Loads symbols for the specified module on the target system. If you do not specify *modulename*, the symbols for all loaded modules will be reloaded. This command is useful in the event of a system crash, which can result in the loss of symbols for the target computer.

BC (Breakpoint Clear)

Syntax

BC {*breakpoint* [*breakpoint*...] | ***}

Parameter

breakpoint

The number of the breakpoint to be cleared. Use the Breakpoint List (**BL**) command to display currently set breakpoints and their numbers. Specify ranges of breakpoints with a hyphen. Separate multiple breakpoints with spaces or commas.

Description

Removes one or more previously set breakpoints from the system. **BC*** clears all breakpoints.

BD (Breakpoint Disable)

Syntax

BD {*breakpoint* [*breakpoint*...] | ***}

Parameter

breakpoint
The number of the breakpoint to be disabled. Use the Breakpoint List (**BL**) command to display currently set breakpoints and their numbers. Specify ranges of breakpoints with a hyphen. Separate multiple breakpoints with spaces or commas.

Description

Disables, but does not delete, one or more breakpoints. **BD*** disables all breakpoints. While a breakpoint is disabled, the system does not check to see if the conditions specified in the breakpoint are valid.

Use the Breakpoint Enable (**BE**) command to reenable a disabled breakpoint.

BE (Breakpoint Enable)

Syntax

BE {*breakpoint* [*breakpoint*...] | ***}

Parameter

breakpoint
The number of the breakpoint to be enabled. Use the Breakpoint List (**BL**) command to display currently set breakpoints and their numbers. Specify ranges of breakpoints with a hyphen. Separate multiple breakpoints with spaces or commas.

Description

Restores one or more breakpoints that were temporarily disabled by the Breakpoint Disable (**BD**) command. **BE*** enables all breakpoints.

BL (Breakpoint List)

Syntax

BL

Description

Lists all breakpoints. For each breakpoint, the command displays the following:

- The breakpoint number
- The breakpoint status, where "E" is for enabled, "D" is for disabled, "V" is for virtual, and "U" is for unknown address. A virtual breakpoint is a breakpoint for code that is not currently loaded.
- The conditional information specifying the breakpoint, such as the address, expression, and length. If a breakpoint has a pass count, the remaining number of times that the breakpoint will be ignored is listed in parentheses.

BL also displays commands to execute, message/message classes (in the case of message breakpoints), thread and process number.

BP (Set Breakpoint)

Syntax

[*thread*] **BP** [*breakpoint*] [*location*] [*condition*] [*option*...]

Parameters

thread
The thread that the breakpoint will apply to. See Process and Thread Syntax for more information on the *thread* syntax.

breakpoint
The breakpoint number to be set. If the given breakpoint number already exists, the new breakpoint will replace the old.

location
The memory location of the breakpoint, in the format given in the table below.

condition
One of the breakpoint conditions given in the table below. You can specify multiple conditions for a breakpoint.

option
One of the breakpoint options given in the table below. Separate multiple options with spaces.

Description

Sets a breakpoint. You can combine locations, conditions, and options to set different kinds of breakpoints. If you do not specify a thread, the breakpoint will apply to all threads.

If you want to put a breakpoint on a C++ public, enclose the expression in parentheses. For example, "BP (??MyPublic)" or "BP (operator new)".

On x86 computers, WinDbg will use debug registers to implement watchpoints if:

- There is a debug register available
- Memory size is 1
- Memory size is 2 and address is **WORD** aligned
- Memory size is 4 and address is **DWORD** aligned

You can use the following options when setting a breakpoint:

Location	**Description**
[**{**[*procedure*], [*module*], [*exe*]**}**]*address*	The address for the breakpoint.
[**{**[*procedure*], [*module*], [*exe*]**}**]**@***line*	The line number for the breakpoint.

Condition	**Description**
?*expression*	Break if *expression* is true.
=*address* [/R*count*]	Break if memory at *address* has changed. Use the **/R** option to specify the number of bytes to check (default is 1).

Option	**Description**
/P*count*	Ignore the breakpoint *count* times.

/C*cmdlist*	Execute *cmdlist* when the breakpoint is hit. The *cmdlist* parameter is a semicolon-separated list of one or more debugger commands. If *cmdlist* includes multiple commands, enclose it in quotes (**"**).
/M*messagename* **/M***messageclass*	Break only if the given message name or message class has been received.
/Q	Suppress the unresolved-breakpoint dialog box for this breakpoint.
/H*process*	Specify the process number to attach the breakpoint to. Defaults to all threads in *process* if **/T** is not used. If **/H** is not specified and no debuggee is running, the default is process 0.
/T*thread*	Specify the thread number to attach the breakpoint to. Defaults to current process if **/H** is not used.

C (Compare Memory)

Syntax

C *range address*

Parameters

range
The first memory area to compare. See Range Syntax for information on the *range* syntax.

address
The starting address of the second memory area.

Description

Compares the values held in two memory areas. Specify the first area with the *range* parameter. Specify the starting address of the second area with *address*. The second area is the same length as the first. If the two areas are not identical, WinDbg will display all memory addresses in *range* where they do not agree.

DA, DB, DC, DD, DI, DS, DT, DU, DW (Display Memory)

Syntax

D{**A** | **B** | **C** | **D** | **I** | **S** | **T** | **U** | **W**} [*range*]

Parameters

range
The memory area to display. See Range Syntax for information on the *range* syntax.

Description

Displays the contents of memory in the given range. Each line shows the address of the first byte in the line, followed by the contents of memory at that and following locations.

If you omit *range*, the command will display memory starting at the ending location of the last Display command. This allows you to continuously scan through memory.

When WinDbg is displaying ANSI or Unicode characters, it will stop displaying characters at the first null byte. When displaying ANSI characters, all characters, including non-printable characters, are displayed using the current code page character set. With Unicode, all nonprintable and nonmappable characters are displayed as dots.

Command	Definition	Displays
DA	Display ANSI	ANSI (extended ASCII) characters
DB	Display Bytes (**char**)	Byte values and ANSI characters
DC	Display Code	Assembly-language instructions (disassembly)
DD	Display Doublewords (**long**)	Doubleword (4-byte) values and ANSI characters
DI	Display 8-Byte Reals (**double**)	8-byte hexadecimal values and floating-point representations
DS	Display 4-Byte Reals (**float**)	4-byte hexadecimal values and floating-point representations
DT	Display 10-Byte Reals (**long double**)	10-byte hexadecimal values and floating-point representations
DU	Display Unicode	Unicode characters
DW	Display Words (**short**)	Word values and Unicode characters

Note With the **DC** (Dump Code) command, you can use the standard *range* syntax or *startaddress* **l** *line* to dump *line* instructions starting at *startaddress*.

EA, EB, ED, EI, ES, ET, EU, EW (Enter Values)

Syntax

E{**A** | **B** | **D** | **I** | **S** | **T** | **U** | **W**} *address* [*values*]

Parameters

address
The starting address to enter values.

values
One or more values to enter into memory. Separate multiple numeric values with spaces.

Description

Enters the values that you specify into memory. If you do not specify any values, the current address and the value at that address will be displayed. You can then enter a new value, preserve the current value in memory by pressing the space bar, or stop entering data by pressing ENTER alone.

When entering numeric values, you can use C-style radix prefixes to override the default radix. Prefix octal constants with a "0o" (for example 0o1776), hexadecimal constants with "0x" (for example 0xF000), and decimal constants with "0t" (for example 0t199).

When entering ANSI or Unicode values, you can include space (**" "**) characters by enclosing the character string in quotation marks (**"** or **'**). If you enclose the string in double quotation marks, WinDbg will automatically null-terminate the string. Single quotation marks (**'**) will not add a null character. You can enter standard C escape characters, such as **t**, **007**, and **"**.

Command	Definition	Enter
EA	Enter ANSI	ANSI (extended ASCII) characters
EB	Enter Bytes (**char**)	Byte values
ED	Enter Doublewords (**long**)	Doubleword (4-byte) values
EI	Enter 8-Byte Reals (**double**)	Floating-point numbers
ES	Enter 4-Byte Reals (**float**)	Floating-point numbers
ET	Enter 10-Byte Reals (**long double**)	Floating-point numbers
EU	Enter Unicode	Unicode characters
EW	Enter Words (**short**)	Word values

F (Freeze Thread)

Syntax

[*thread*] **F**

Parameter

thread
The thread to be frozen. See Process and Thread Syntax for information on the *thread* syntax.

Description

Freezes the given thread, causing it to wait until it is unfrozen. Other threads will continue to execute. If no thread is specified, the current thread is frozen.

Use the **Z** (Unfreeze) command to reenable the thread.

FIA, FIB, FID, FII, FIS, FIT, FIU, FIW (Fill Memory)

Syntax

FI{**A** | **B** | **D** | **I** | **S** | **T** | **U** | **W**} *range pattern*

Parameters

range
The memory area to fill. See Range Syntax for information on the *range* syntax.

pattern
One or more values to fill memory with. Separate multiple numeric values with spaces.

Description

Fills the memory area specified by *range* with the user-specified *pattern*. The entire *range* will be filled by repeatedly storing *pattern* into memory.

When entering numeric values, you can use C-style radix prefixes to override the default radix. Prefix octal constants with a "0o" (for example 0o1776), hexadecimal constants with "0x" (for example 0xF000), and decimal constants with "0t" (for example 0t199).

When entering ANSI or Unicode values, you can include space (**" "**) characters by enclosing the character string in quotation marks (**"** or **'**). If you enclose the string in double quotation marks, WinDbg will automatically null-terminate the string. Single quotation marks (**'**) will not add a null character. Standard C escape characters (such as **\t**, **\007**, and **\"**) are also allowed.

Command	Definition	Fill
FIA	Fill ANSI	ANSI (extended ASCII) characters
FIB	Fill Bytes (**char**)	Byte values
FID	Fill Doublewords (**long**)	Doubleword (4-byte) values
FII	Fill 8-Byte Reals (**double**)	Floating-point numbers
FIS	Fill 4-Byte Reals (**float**)	Floating-point numbers
FIT	Fill 10-Byte Reals (**long double**)	Floating-point numbers
FIU	Fill Unicode	Unicode characters
FIW	Fill Words (**short**)	Word values

FR (Floating-Point Registers)

Syntax

[*thread*] **FR** [*register* [**=***value*]]

Parameters

thread
The thread from which the floating-point registers are to be read. See Process and Thread Syntax for information on the *thread* syntax.

register
The floating-point register to display or modify.

value
The floating-point value to assign to *register*.

Description

Displays or modifies floating-point registers. If no thread is specified, the current thread is used.

If you do not specify a register, all of the floating-point registers are displayed. If you specify a register, the command displays the current value of the register and prompts for a new value. If you specify both a register and a value, the command changes the register to contain the value.

Use the **R** command to view and modify standard registers.

G (Go)

Syntax

[*process*|*thread*] **G** [*breakaddress*]

Parameters

process
The process to execute. See Process and Thread Syntax for information on the *process* syntax.

thread
The thread to execute. See Process and Thread Syntax for information on the *thread* syntax.

breakaddress
The address for an unconditional breakpoint.

Description

Starts executing the given process or thread. Execution will halt at the end of the program, when *breakaddress* is hit, or when another event causes the debugger to stop.

Note The breakpoint associated with *breakaddress* will only be hit by the current thread. Other threads that execute the code at that location will not be stopped.

GH (Go Exception Handled)

Syntax

[*thread*] **GH**

Parameter

thread

The thread to execute. This thread must have been stopped by an exception. See Process and Thread Syntax for information on the *thread* syntax.

Description

Marks the given thread's exception as being handled and allows the thread to continue execution at the instruction that caused the exception. This allows the debuggee to handle the exception. Use the **GN** (Go Exception Not Handled) command to continue execution without marking the exception as handled.

GN (Go Exception Not Handled)

Syntax

[*thread*] **GN**

Parameter

thread

The thread to execute. This thread must have been stopped by an exception. See Process and Thread Syntax for information on the *thread* syntax.

Description

Continues execution of the given thread without marking the exception as having been handled. This allows the debuggee's exception handler to handle the exception. Use the **GH** (Go Exception Handled) command to continue execution after marking the exception as having been handled.

K, KB, KN, KS, KV (Display Stack Backtrace)

Syntax

[*thread*] **K**[**BNSV**] [*framecount*]

Parameters

thread
The thread whose stack is to be displayed. See Process and Thread Syntax for information on the *thread* syntax.

framecount
Number of stack frames to display.

Description

Displays the stack frame of the given thread. Each display line shows the name or address of the procedure called, the arguments used on the call, and the address of the statement that called it. You can use any or all of the options in a single command; for example, **K**, **KB**, and **KBNSV** are valid commands. The following table describes the effect of the options:

Option	Effect
none	The **K** command without any options displays the basic call stack based on debugging information in the executable. It displays the frame pointer, return address, and function names.
B	The **B** option causes the **K** command to additionally display the first three parameters to the functions.
N	The **N** option causes the **K** command to additionally display the frame numbers for the calls.
S	The **S** option causes the **K** command to additionally display source module and line number information for the calls.
V	The **V** option causes the **K** command to additionally display runtime function information.

L (Restart Debuggee)

Syntax

L [*parameters*]

Parameter

parameters
Command-line options for the debuggee.

Description

Restarts the debuggee with optional *parameters* supplied as command-line options.

LM (List Loaded Modules)

Syntax

LM [/s [/o] [/f] [*modulename*]

Parameter

parameters

/s List segmented modules.

/f List flat modules.

/o Sorts segmented modules by selector.

modulename Module to list

Description

This command lists the specified loaded modules. If you do not specify a modulename, **LM** will list all loaded modules. If **/f** and **/s** are both absent from the command line, **LM** will assume both; the following two commands are equivalent:

> LM

> LM /f /s

Segmented modules are sorted by module name, then selector, unless you specified **/o**. Flat modules are sorted by base address.

LN (List Nearest symbols)

Syntax

LN *address*

Parameter

address
The address to search for symbols.

Description

Displays the symbols at or near the given *address*. You can use this command to help determine what a pointer is pointing to. It also can be useful when looking at a corrupted stack to determine what procedure made a call.

M (Move Memory)

Syntax

M *range address*

Parameters

range
The memory area to copy. See Range Syntax for information on the *range* syntax.

address
The starting address of the destination memory area.

Description

Copies the contents of memory from one location to another. It is legal for *address* to be part of *range*. Overlapping moves are handled correctly.

N (Set Number Base)

Syntax

N *radix*

Parameter

radix
The default number base used for numeric display and entry. Legal values are 8 (octal), 10 (decimal), and 16 (hexadecimal).

Description

Sets the default number base to *radix*. You can use C-style radix prefixes to override the default radix.

WinDbg's default is base 16.

P (Program Step)

Syntax

[*thread*] **P** [*count*]

Parameters

thread
The thread to step through. See Process and Thread Syntax for information on the *thread* syntax.

count
The number of instructions or source lines to step through before stopping.

Description

Executes the instruction (in ASM mode) or source line (in SRC mode) at the instruction pointer. If WinDbg encounters a CALL instruction or interrupt while stepping, the called subroutine will execute before control returns to the debugger.

Use the **T** (Trace) command to have WinDbg step through subroutine calls and interrupt-handling routines.

Q (Quit WinDbg)

Syntax

Q

Description

Exits WinDbg. This command is identical to choosing Exit from the File menu.

R (Registers)

Syntax

[*thread*] **R** [*register* [=*value*]]

Parameters

thread
The thread from which the registers are to be read. See Process and Thread Syntax for information on the *thread* syntax.

register
The register to display or modify.

value
The value to assign to *register*.

Description

Displays or modifies registers. If you do not specify a thread, the current thread is used.

If you do not specify a register, all of the registers are displayed. If you specify a register, the command displays the current value of the register. If you specify both a register and a value, the command changes the register to contain the value.

Use the **FR** (Floating-Point Registers) command to view and modify floating-point registers.

REMOTE (Start Remote Server)

Syntax

REMOTE [*pipename* | STOP]

Parameters

pipename
The name that you want to use for the remote server pipe.

STOP
Ends a currently active remote server.

Description

Starts a remote server, which lets you get access to the Command window from another computer on the network. You can then enter Command window commands on the other computer to debug the application remotely.

If you do not specify any parameters to REMOTE, the command displays its connection status and the name of the client.

To connect from another computer, run the following in a cmd shell (the Windows NT command prompt):

remote /c *hostname pipename*

where *hostname* is the computer name of the computer that is running WinDbg and *pipename* is the same as above.

You now have access from the remote computer to the WinDbg Command window.

RT (Register Display Toggle)

Syntax

[*thread*] **RT**

Parameters

thread
The thread from which the registers are to be read. See Process and Thread Syntax for information on the *thread* syntax.

Description

Toggles the register display (WinDbg KD only). The default is off.

S+, S - (Set Source/Assembly Mode)

Syntax

S+
S -

Description

Use these commands to switch between source mode (where the debugger steps line-by-line through the source code) and assembly-language mode (where the debugger steps instruction-by-instruction). The **S+** command switches to Source mode, and **S** - switches to assembly-language mode. The status bar displays whether the current mode is SRC or ASM.

SA, SB, SD, SI, SS, ST, SU, SW (Search Memory)

Syntax

S{**A** | **B** | **D** | **I** | **S** | **T** | **U** | **W**} *range pattern*

Parameters

range
The memory area to search through. See Range Syntax for information on the *range* syntax.

pattern
One or more byte values or ANSI or Unicode characters to search for. Separate multiple values with spaces.

Description

Searches through memory to find a specific byte pattern. If the pattern is found, WinDbg will display the first memory address in *range* where it was found.

When entering numeric values to search for, you can use C-style radix prefixes to override the default radix. Prefix octal constants with a "0o" (for example 0o1776), hexadecimal constants with "0x" (for example 0xF000), and decimal constants with "0t" (for example 0t199).

When entering Unicode values to search for, you can include space (**" "**) characters by enclosing the character string in quotation marks (**"** or **'**). If you enclose the string in double quotation marks, WinDbg will automatically null-terminate the string. Single quotation marks (**'**) will not add a null character. Standard C escape characters (such as **\t**, **\007**, and **\"**) are also allowed.

Command	Definition	Fill
SA	Search ANSI	ANSI (extended ASCII) characters
SB	Search Bytes (**char**)	Byte values
SD	Search Doublewords (**long**)	Doubleword (4-byte) values
SI	Search 8-Byte Reals (**double**)	Floating-point numbers
SS	Search 4-Byte Reals (**float**)	Floating-point numbers
ST	Search 10-Byte Reals (**long double**)	Floating-point numbers
SU	Search Unicode	Unicode characters
SW	Search Words (**short**)	Word values

SEB, SEW (Set Error Break, Set Error Warning)

Syntax

SEB [*level*]
SEW [*level*]

Parameters

level
The error level required to break into the debugger (**SEB**) or display a message (**SEW**). One of the following:

Level	Triggered by
0	Nothing. Disables breaks or warnings.
1	Severe errors only.
2	All errors.
3	All warnings and errors.

Description

The **SEB** command causes the debugger to break after the debuggee causes a system error or warning of the appropriate *level*. The **SEW** command causes the debugger to display a message under these conditions. If you do not specify a level, the commands will default to level 1.

The **SEB** command implies an **SEW** *level* equal to or greater than its own level. For example, **SEB 3** causes the debugger to give warnings at levels 1 - 3.

These errors are triggered by the Windows subsystem. They are known as RIPs from Win16 where they usually signal a fatal program error. They are often caused by passing a bad value to an API function.

SX (Set Exceptions)

Syntax

SX [*exception*]
SXE *exception* [*message*] [**/C***cmdlist1*] [**/C2***cmdlist2*]
SX{**D** | **N**} *exception* [*message*] [**/C2***cmdlist2*]

Parameters

exception
The exception number that the command acts upon, in the current radix. If you do not specify an exception, the **SX** command will display information on all exceptions.

message
Message to display in the Command window when the exception is trapped.

cmdlist1
Semicolon-separated list of WinDbg commands to execute when an exception first occurs. The **/C** option is permitted only with the **SXE** command. Enclose in quotes if *cmdlist1* includes spaces or semicolons.

cmdlist2
Semicolon-separated list of WinDbg commands to execute after an exception has not been handled. Enclose in quotes if *cmdlist2* includes spaces or semicolons.

Description

Controls the behavior of the debugger when trapping exceptions before executing exception-handling code. The debugger always halts before execution returns from the exception handler.

Command	Action
SX	Displays the events that the debugger will halt for.
SXD	Causes the debugger to ignore the specified exception and issue an automatic **GN** command.
SXE	Causes the debugger to halt at the specified exception.
SXN	Causes the debugger to display a message before the exception is passed to the debuggee, and causes the debugger to issue a **GN** command after the message is displayed.

The **/C** option (allowed only with **SXE**) tells WinDbg to execute the specified debugger commands on the first chance (before the exception is passed to the debuggee). The **/C2** option tells WinDbg to execute the specified debugger commands on the second chance (if the exception is not handled by WinDbg or the application). See your documentation on structured exception handling for more information.

When the debugger stops due to an exception, only the **GN** (Go Exception Not Handled) and **GH** (Go Exception Handled) commands can be used to continue execution. The **SXD** and **SXN** commands automatically call **GN**.

T (Trace)

Syntax

[*thread*] **T** [*count*]

Parameters

thread
The thread to trace through. See Process and Thread Syntax for information on the *thread* syntax.

count
The number of instructions or source lines (depending on the Source/Assembly mode) to step through before stopping.

Description

Executes the instruction or source line at the instruction pointer and displays the resulting values of all registers and flags.

Use the **P** (Program Step) command to have WinDbg execute subroutine calls or interrupts without returning control to the debugger.

U (Unassemble)

Syntax

U [[*range*]

Parameters

range
The memory area that contains the instructions to unassemble. See Range Syntax. You can use the standard range syntax or *startaddress* **l** *line* to dump *line* instructions starting at *startaddress*.

Description

Displays the instructions of the program being debugged. If you do not specify *range*, the debugger displays the instructions generated from the first eight lines of code at the current address.

The 80286 protected-mode mnemonics cannot be displayed.

X (Examine Symbols)

Syntax

X [*options*] [{ [*procedure*]**,** [*module*]**,** [*executable*] }] [*pattern*]

Parameters

options

Specifies the scope of the symbol search. These options cannot be separated by spaces and must immediately follow the **X** (for example, **XEM**). Use the following letters to specify scope:

Option	Description
C	Search current class.
E	Search entire executable or DLL, except for current module.
F	Search current function.
G	Search all global symbols.
L	Search current code block (lexical scope).
M	Search current module.
P	Search all public symbols.

procedure, module, executable

The area to search. For example,

```
{Function1, srcfile.c, target.dll}
```

will search Function1 from SCRFILE.C in TARGET.DLL. This is identical to the breakpoint context operator.

pattern

A pattern to search for. The **?** and ***** wildcards are supported.

Description

Displays the symbols in all contexts that match *pattern*. If you do not specify any options, all except public symbols are searched. If you do not specify a pattern, all symbols will be displayed. The search will be case-sensitive if the Ignore Case option is not checked in the Debug command from the Options menu.

Z (Unfreeze Thread)

Syntax

[*thread*] **Z**

Parameters

thread
The thread or threads to be unfrozen. See Process and Thread Syntax for information on the *thread* syntax.

Description

Unfreezes the specified thread. If you do not specify the thread, this command unfreezes the thread that caused the most recent exception, if any. You can freeze threads with the **F** (Freeze) command.

Creating Help Files

Microsoft Windows Help provides online Help for users working with a Windows-based application. Windows Help provides a practical way to present information about your application in a format users can access easily.

This section introduces the tools you can use to develop Windows Help files and to incorporate Help in Windows-hosted applications.

About Windows Help Files

Windows Help files can display information by using the following elements:

- Text in multiple fonts, sizes, and colors
- Bitmaps and metafiles with up to 16 colors
- Segmented-graphics bitmaps with embedded hot spots
- Cross-reference jumps for links to additional information
- Pop-up windows to present text and graphics
- Secondary windows to present information without the full menus and buttons of Windows Help
- Keywords to help users find the information they need

You create Help files by creating topic and graphics files and a Help project file. A topic file contains the text for the Help topic and contains the Help statements and macros that define the format of the text and the position of graphics in each topic. The graphics files contain the bitmaps and metafiles you want to display in the topics. The project file contains a description of how to build the Help file.

You use the Microsoft Help Compiler to build the final Help file. Combining the topic, graphics, and project files, the compiler creates a single Help file (with the filename extension .HLP) that you can open and view by using Windows Help.

Creating Topic Files

A topic file contains the text for the Help file, as well as the statements and macros that define the format of the text and the position of the graphics. Every topic file consists of one or more topics. A topic is any distinct unit of information, such as a contents screen, a conceptual description, a set of instructions, a keyboard table, a glossary definition, a list of jumps, a picture, and so on.

Windows Help displays only one topic at a time, but a user can view any topic in a Help file by using a link to the topic or searching for keywords associated with the topic.

You create topic files directly by using a text editor and inserting Help statements. You can create them indirectly by using a word processor that generates rich-text format (RTF) files. The Help statements are an extended subset of the RTF statements, which provide a wide variety of formatting capabilities.

Declaring Character Set, Fonts, and Colors

When you create a topic file, you must ensure that the entire contents of the file are enclosed in braces (**{ }**). The first statement in the file must be the **\rtf** statement; it immediately follows the first opening brace. You should follow the **\rtf** statement with a **\ansi** statement (or a similar statement) that specifies the character set used in the file. The following example shows the general form for a topic file:

```
{\rtf1\ansi
     .
     .
     .
}
```

You must declare the names of the fonts you use in the file by using a **\fonttbl** statement. The **\fonttbl** statement, enclosed in braces, contains a list of font and family names and specifies a unique number for each font. You use these numbers with **\f** statements later in the file to set specific fonts. The following **\fonttbl** statement assigns font numbers 0, 1, and 2 to the TrueType® fonts Times New Roman, Courier New, and Arial, respectively:

```
{\fonttbl
\f0\froman Times New Roman;
\f1\fdecor Courier New;
\f2\fswiss Arial;}
```

You should also use the **\deff** statement to set the default font for the file. Windows Help uses this default font if no other font is specified. The following example sets the default font number to zero, corresponding to the Times New Roman font specified in the previous **\fonttbl** statement:

```
\deff0
```

If you use specific text colors or choose not to rely on the default text colors set by Windows, you must define your colors by using a **\colortbl** statement. The **\colortbl** statement, enclosed in braces, defines each color by specifying the amount of each primary color (red, green, and blue) used in it. The statement implicitly numbers the colors consecutively starting from zero. You use these color numbers with **\cf** statements later in the file to set the color. The following example creates four colors (black, red, green, and blue):

```
{\colortbl
\red0\green0\blue0;
\red255\green0\blue0;
\red0\green128\blue0;
\red0\green0\blue255;}
```

Although it is not shown here, you can put a semicolon immediately after the **\colortbl** statement to define the default color as 0.

Defining Individual Topics

Each topic starts with one or more **\footnote** statements and ends with a **\page** statement. All text and graphics specified between these statements belong to the topic.

Every topic must have a context string. Windows Help uses the context string to locate the topic when the user requests to view it. You assign a context string to a topic by using the **\footnote** statement and the number sign (**#**) footnote character. Context strings can consist of letters, digits, and the underscore character (**_**). To prevent conflicts, each context string in a Help file must be unique.

You can also assign a title to the topic by using the **\footnote** statement and the dollar sign (**$**) footnote character. Windows Help uses the title to identify the topic in the History and Search dialog boxes. You must provide a title if you assign keywords to the topic.

The following example defines a small topic having the context string "topic1" and the title "My Topic":

```
#{\footnote topic1}
${\footnote My Topic}
This is my first topic.
\par
\page
```

In general, you use the **\par** statement to mark the end of each paragraph. In this example, the **\par** statement marks the end of the only paragraph in the topic.

You can add a macro to a topic by using the **\footnote** statement and the exclamation point (**!**) as the footnote character. For example, the following **\footnote** statement adds the **CopyTopic** macro to the topic:

```
!{\footnote CopyTopic()}
```

Windows Help executes the macro each time it displays the topic.

The total size of text and graphics data stored in a paragraph must not exceed 64K. (Bitmaps included by using the **bmc**, **bml**, and **bmr** statements do not contribute to this total.)

Setting Font Size and Name

You can set the font name and size by using the **\f** and **\fs** statements. The name is set by using a font number specified in the **\fonttbl** statement. The size of the font is specified in half-points. The following example sets the text to 10-point Times New Roman (if the **\fonttbl** statement matches the example in Declaring Character Set, Fonts, and Colors):

```
\f0\fs20
```

Once you set the font name and size, the settings apply to all subsequent text up to the next **\plain** statement or until you change the name or size by using the **\f** or **\fs** statement again. The **\plain** statement resets the name and font to the defaults. The default font name is as set by the **\deff** statement; the default font size is 12 points.

Setting Space Before and After Paragraphs

You can set the amount of space before and after each paragraph by using the **\sb** and **\sa** statements. These statements let you control the amount of space that appears between paragraphs. You specify the space in twips. (A *twip* is 1/1440 inch, or 1/20 of a printer's point). The following example sets the space before a paragraph to 360 twips:

```
\sb360
This paragraph has 360 twips space immediately before it.
\par
This paragraph also has 360 twips before it.
\par
```

Once you set the space before or after a paragraph, the spacing applies to all subsequent paragraphs up to the next **\pard** statement or until you change the spacing by using the **\sa** and **\sb** statements again. The **\pard** statement restores the default spacing.

Setting the Left and Right Indents

When Windows Help displays its window, it automatically creates left and right margins and wraps text to fit within these margins. The margins are positioned slightly within the left and right edges of the window to prevent text in the topic from being clipped by the window.

You can override these margins by setting the left and right indents for a paragraph. The **\li** and **\ri** statements set an indent to a position relative to the corresponding left and right margins. For example, the following paragraph is indented 1 inch (1440 twips) from the left margin:

```
\li1440
This paragraph is indented 1 inch.
\par \pard
This paragraph is not indented.
```

Once indents are set, they apply to all subsequent paragraphs up to the next **\pard** statement. Note that the **\pard** statement must follow the **\par** statement that ends the paragraph to be indented.

You can set an indent for the first line in a paragraph by using the **\fi** statement. This allows you to create paragraphs with hanging indents. It is also useful for creating two-column lists.

Setting Tab Stops

You can set tab stops by using the **\tx** statement. You can use one or more **\tx** statements, each setting a specific position in twips relative to the left margin. Once you have set tab stops, you can use the **\tab** statement to align subsequent text with the next tab. The tab settings remain active until you use the **\pard** statement. The following example creates a two-column list by using a tab stop and paragraph indenting:

```
\fi-1440\li1440\tx1440
left
\tab
right
\par
left
\tab
right
\par
\pard
```

Breaking Lines

Ordinarily, Windows Help wraps all lines in a paragraph, fitting as many words on a line as will fit between the current left and right indents. You can force Windows Help to break a line at a given place by using the **\line** statement. You can control wrapping by using the **\keep** and **\pard** statements.

The following example uses the **\keep** statement to turn off word wrapping for three short lines and uses the **\pard** statement to restore the default properties:

```
\keep
3 pairs black socks\line
5 pairs blue socks\line
2 pairs brown socks\line
\par
\pard
```

The following example uses the **\keep** and **\pard** statements to create three nonwrapping paragraphs:

```
\keep
3 pairs black socks
\par
5 pairs blue socks
\par
2 pairs brown socks
\par
\pard
```

Creating Links and Pop-up Topics

Windows Help displays only one topic at a time. To enable users to view other topics, you must create hot spots that link your topics to other topics. You create a hot spot by using the **\strike**, **\ul**, or **\uldb** statement and a corresponding **\v** statement. When you create a link, you provide the text for the hot spot and the context string for the topic that is to be jumped to or displayed. The following example creates a hot spot named Glossary and establishes a link from the hot spot to the topic having the context string "glo1":

```
You can find a list of terms used in this
help file in the {\uldb Glossary}{\v glo1}.
```

When Windows Help displays the topic with this hot spot, it places a line under the word *Glossary* and colors the word green. The context string is not shown, but if the user clicks on the hot spot, Windows Help jumps to and displays the corresponding topic.

The **\strike** and **\uldb** statements are used to create jumps to other topics. The **\ul** statement creates a link to a pop-up topic. Windows Help displays pop-up topics in a pop-up window and leaves the current topic in the main window.

You can also associate a Help macro with a hot spot in a topic. For example, the following **\uldb** and **\v** statements create a hot spot for the **ExecProgram** macro:

```
{\uldb Clock}{\v !ExecProgram("clock.exe", 1)}
```

Windows Help executes the macro whenever the user chooses the hot spot. Windows Help continues displaying the topic while it executes the macro, unless the macro causes a jump to another topic.

Creating a Keyword List

You can also enable users to find and view topics by assigning keywords to the topics. You assign a keyword by using the **\footnote** statement and the letter K as the footnote character. Windows Help collects all keywords in a Help file and displays them in its Search dialog box. Using this dialog box, a user can select a keyword and view the Help topics associated with it. The following example assigns the keyword "Sample Topics" to the current topic:

```
#{\footnote topic1}
${\footnote My Topic}
K{\footnote Sample Topics}
This is my first topic.
\par
\page
```

If a keyword begins with the letter K, you must place an extra space before the word. Multiple keywords for a topic are separated by semicolons.

A keyword can be assigned to any number of topics. When the user selects the keyword in the Search dialog box, Windows Help displays all topics associated with the keyword. The user then picks the one to view.

You can also create alternative keywords for a Help file for use with the **WinHelp** function.

Creating Browse Sequences

You can enable users to browse through a sequence of Help topics by creating a browse sequence and adding browse buttons to your Help file. A browse sequence typically consists of two or more related topics that are intended to be read sequentially. You create a browse sequence by using the **\footnote** statement and the plus-sign (**+**) footnote character to assign a sequence identifier. The following example assigns a sequence identifier to the topic titled "A Topic":

```
#{\footnote topic5}
${\footnote A Topic}
+{\footnote shorttopics}
This is one topic in a browse sequence.
\par
\page
```

Windows Help adds topics with sequence identifiers to the browse sequence and determines the order of topics in the sequence by sorting the identifiers alphabetically. If two topics have the same identifier, Windows Help assumes that the topic that was compiled first is to be displayed first.

Windows Help uses the sequence only if the browse buttons have been enabled. You can enable the buttons by placing the following statements in the Help project file:

```
[CONFIG]
BrowseButtons()
```

For more information about the project file, see Creating Help Project Files.

You can create more than one browse sequence in a Help file by using sequence numbers with sequence identifiers. The sequence number consists of a colon (**:**) followed by an integer. Windows Help combines all topics having the same sequence identifier (but different sequence numbers) into a single browse sequence and determines the order of the topics by sorting them alphabetically. To ensure that numerals are sorted correctly, they should have the same number of digits. For example, the numerals 1 through 10 should be 01 through 10.

```
#{\footnote topic10}
${\footnote Alpha Topic #3}
+{\footnote alpha:3}
This topic is part of the alpha browse sequence.
\par
\page
```

Using Graphics Files

You can add bitmaps and metafiles to your Help files by using the **bml**, **bmc**, and **bmr** statements. These statements take the name of a graphics file and insert the corresponding bitmap or metafile into the Help file at the specified position.

Windows Help requires graphics files to be in one of the following formats:

- Windows bitmap (.BMP)
- Placeable Windows metafile (.WMF)
- Multiple-resolution bitmap (.MRB)
- Segmented-graphics bitmap (.SHG)

Multiple-resolution bitmaps can be created by using the Microsoft Multiple-Resolution Bitmap Compiler (MRBC). Segmented-graphics bitmaps can be created by using Microsoft Windows Hotspot Editor. Only 16-color and monochrome bitmaps may be used. Windows Help does not support 256-color bitmaps.

Although the **\pict** statement can also be used to add bitmaps and metafiles to a Help file, the bitmap or metafile data must be inserted into the topic file rather than specified as a separate file.

Inserting a Bitmap in Text

You can insert a bitmap into a paragraph as if it were a character by using the **bmc** statement. The statement aligns the bottom of the bitmap with the base line of the current line of text and places the left edge of the bitmap at the next character position. The following example inserts a bitmap into a line of text:

```
Press the \{bmc enter.bmp\} key to complete the task and return to
the main window.
```

Since the bitmap is treated as text, any paragraph properties assigned to the paragraph also apply to the bitmap. Windows Help places text following the bitmap on the same base line at the next available character position.

In general, bitmaps inserted as characters should be clipped to the smallest possible size. Any extra white space at the top or bottom of the bitmap image affects the alignment of the bitmap with the text and may affect the spacing between lines.

You must not specify negative line spacing for paragraphs that contain **bmc** statements. Doing so might cause the bitmap to appear on top of the paragraph.

Wrapping Text Around a Bitmap

You can place a bitmap at the left or right margin of the Help window and have subsequent text wrap around the bitmap by using the **bml** or **bmr** statement. The **bml** statement inserts a bitmap at the left margin; **bmr** inserts it at the right.

If you want text to wrap around a bitmap, you must place the **bml** or **bmr** statement at the beginning of a paragraph. Windows Help aligns the start of the paragraph with the top of the bitmap and wraps around the left or right edge of the bitmap. The following example places the bitmap at the left margin and subsequent text wraps around its right edge:

```
\{bml mybitmap.bmp\}
The text in this paragraph wraps around
the right edge of the bitmap.
\par
```

If you place a **bml** or **bmr** statement at the end of a paragraph, Windows Help places the bitmap under the paragraph instead of wrapping the text around the bitmap. If you do not want text to wrap around a bitmap, place **\par** statements immediately before and after the **bml** or **bmr** statement.

Using a Bitmap as a Hot Spot

You can use bitmaps as hot spots. This enables you to create graphics, such as icons or buttons, and use them as "jumps" to particular topics or as hot spots for macros. The following example uses the bitmap in the MYBUTTON.BMP file to create a link. When the user clicks the bitmap, Windows Help jumps to the topic identified by the context string "topic15":

```
{\strike \{bml mybutton.bmp\}}{\v topic15}
```

You can also divide a single bitmap into several hot spots and assign a different link or macro to each hot spot. Such bitmaps, called segmented-graphics bitmaps, are created by using Hotspot Editor. For example, if you have a bitmap of a dialog box, you can assign links to each of the control windows in the dialog box, enabling the user to click a control window and view information about it. Segmented-graphics bitmaps already contain the context strings needed for the links; only a **bml** or **bmr** statement is needed to insert the bitmap. The **\strike** and **\v** statements must not be used.

```
\{bml mydialog.shg\}
```

Using a Bitmap on Different Displays

A multiple-resolution bitmap is a single bitmap file that contains one or more bitmaps that have been marked for use with specific displays, such as the CGA, EGA, VGA, or 8514 displays. You use multiple-resolution bitmaps to avoid problems associated with displaying bitmaps designed for a single type of display. Single-resolution bitmaps can have the following problems:

- Appear too big or too small on displays having different resolutions
- Appear stretched or compressed on displays with different aspect ratios
- Lack colors or use unintended colors on displays with different color capabilities

You create multiple-resolution bitmaps by using MRBC. The compiler, an MS-DOS program, has the following command-line syntax:

mrbc [**/s**] *filename*

The *filename* parameter specifies the name of a Windows bitmap file. Typically, you specify several filenames, one for each type of display. Wildcards can be used. The compiler uses the filename of the first bitmap file as the name of the output file but gives the output file the filename extension .MRB. The following example combines the bitmap files MYBUTTON.EGA, MYBUTTON.VGA, and MYBUTTON.854 into the multiple-resolution bitmap file MYBUTTON.MRB:

```
mrbc mybutton.ega mybutton.vga mybutton.854
```

In this example, the compiler checks the **biXPelsPerMeter** and **biYPelsPerMeter** members of the **BITMAPINFOHEADER** structure in each bitmap file to determine the display type for the bitmap. If these members are set to zero, the compiler prompts for the display type with a message such as the following:

```
Please enter the monitor type for the bitmap mybutton.ega:
```

You must enter at least the first character of one of the following display-type names: CGA, EGA, VGA, or 8514. The compiler sets the display type you specify, but it does not check that the type is valid. For example, if you specify VGA for an EGA bitmap, the compiler marks it as a VGA bitmap. The result may be undesirable.

The **/s** option, specifying silent mode, speeds up compilation if the names of the bitmap files conform to the MRBC filename convention. If you use the **/s** option, the compiler uses the first character of the filename extension to determine the display type for the bitmap, as described in the following list:

Letter	Meaning
C	CGA bitmap
E	EGA bitmap
V	VGA bitmap
8	8514 bitmap

If the filename extension starts with any other character, MRBC assumes a VGA bitmap. The following example creates the multiple-resolution bitmap file MYBUTTON.MRB, containing bitmaps for EGA, VGA, and 8514 displays:

```
mrbc /s mybutton.ega mybutton.vga mybutton.854
```

The compiler never writes over existing multiple-resolution bitmap files. If the output file already exists, the compiler displays an error message.

You insert multiple-resolution bitmaps into your Help file by using the same statements as for Windows bitmaps. For example, the following **bmc** statement inserts the bitmaps from the MYBUTTON.MRB file:

```
Click the \{bmc mybutton.mrb\} button to complete the task and
return to the main window.
```

Before displaying a multiple-resolution bitmap, Windows Help checks the display type for the computer and then selects the bitmap that has the closest matching resolution, aspect ratio, and color capabilities. Windows Help never displays more than one bitmap from a multiple-resolution bitmap file.

Creating Help Project Files

This section describes the format and contents of the Help project file (.HPJ) used to build the Help file. The project file contains all the information the Microsoft Help Compiler needs to combine topic files and other elements into a Help file.

Project File Sections

Every project file consists of one or more sections. Each section has a section name, enclosed in brackets ([]), that defines the purpose and format of statements and options in the section. Following are the sections used in project files:

Section	Description
[OPTIONS]	Specifies options that control the build process. This section is optional. If this section is used, it should be the first section listed in the project file, so that the options will apply during the entire build process.
[FILES]	Specifies topic files to be included in the build. This section is required.
[BUILDTAGS]	Specifies valid build tags. This section is optional.
[CONFIG]	Specifies Help macros that define nonstandard menus, buttons, and macros used in the Help file. This section is required if the Help file uses any of these features. This section is new for Windows 3.1.
[BITMAPS]	Specifies bitmap files to be included in the build. This section is not required if the project file lists a path for bitmap files by using the **BMROOT** or **ROOT** option.
[MAP]	Associates context strings with context numbers. This section is optional.
[ALIAS]	Assigns one or more context strings to the same topic. This section is optional.
[WINDOWS]	Defines the characteristics of the primary Help window and the secondary-window types used in the Help file. This section is required if the Help file uses secondary windows. This section is new for Windows 3.1.
[BAGGAGE]	Lists files that are to be placed within the Help file (which contains its own file system). This section is optional.

Every project file requires a [FILES] section. This section names the topic files. Most project files also have an [OPTIONS] section that specifies how to build the Help file. A very useful option in the [OPTIONS] section is the **COMPRESS** option, which specifies whether the Help file should be compressed or uncompressed. Compressing a Help file reduces its size considerably and saves valuable disk space.

The following example creates a compressed Help file from two topic files, MAIN.RTF and MENUS.RTF:

```
[OPTIONS]
COMPRESS=TRUE

[FILES]
MAIN.RTF
MENUS.RTF
```

Using Macros in Project Files

You can add macros to the [CONFIG] section of a project file. Since Windows Help executes the macros when it first opens the Help file, macros that create menus, menu items, and buttons are typically placed in this section. If there is more than one macro listed in the [CONFIG] section, Windows Help executes them in the order in which they are listed.

You can create new menu items and buttons for Windows Help by using such macros as **CreateButton** and **InsertMenu**. These macros define other Help macros and associate them with the menu items and buttons. Windows Help executes these macros when the user chooses a corresponding menu item or button. Macros that create Help buttons, menus, or menu items remain in effect until the user quits Windows Help or opens a new Help file.

You can extend the capabilities of Windows Help by developing your own dynamic-link libraries (DLLs) and defining Help macros that call functions in the libraries. To define Help macros that call DLL functions, you must register each function and its corresponding library by using the **RegisterRoutine** macro in the [CONFIG] section of the project file.

Sample Project File

The following example is a sample project file for the Cardfile application. Comments, marked by a beginning semicolon (**;**), indicate the purpose of each section in the file:

```
; Options used to define the Help title bar and icon
[OPTIONS]
ROOT=C:\HELP
BMROOT=C:\HELP\ART
CONTENTS=cont_idx_card
TITLE=Cardfile Help
ICON=CARDHLP.ICO
COMPRESS=OFF
WARNING=3
REPORT=ON
ERRORLOG=CARD.BUG

; Files used to build Cardfile Help
[FILES]
RTFTXT\COMMANDS.RTF
RTFTXT\HOWTO.RTF
RTFTXT\KEYS.RTF
RTFTXT\GLOSSARY.RTF

; Button macros and Using Help file
[CONFIG]
CreateButton("btn_up", "&Up", "JumpContents('HOME.HLP')")
BrowseButtons()
SetHelpOnFile("APPHELP.HLP")

; Secondary-window characteristics
[WINDOWS]
picture = "Samples", (123,123,256,256), 0, (0,255,255), (255,0,0)
```

Using Help in a Windows Application

Windows applications can offer help to their users by using the **WinHelp** function to start Windows Help and display topics in the application's Help file. The **WinHelp** function gives a Windows application complete access to the Help file, as well as to the menus and commands of Windows Help. Many applications use **WinHelp** to implement context-sensitive Help. Context-sensitive Help enables users to view topics about specific windows, menus, menu items, and control windows by selecting the item with the keyboard or the mouse. For example, a user can learn about the Open command on the File menu by selecting the command (using the direction keys) and pressing the F1 key.

Choosing Help from the Help Menu

Every application should provide a Help menu to allow the user to open the Help file with either the keyboard or the mouse. The Help menu should contain at least one Contents menu item that, when chosen, displays the contents or the main topic in the Help file. To support the Help menu, the application's main window procedure should check for the Contents menu item and call the **WinHelp** function, as in the following example:

```
case WM_COMMAND:
    switch (wParam) {
    case IDM_HELP_CONTENTS:
        WinHelp(hwnd, "myhelp.hlp", HELP_CONTENTS, 0L);
        return 0L;
        .
        .
        .

    }
    break;
```

You can add other menu items to the Help menu for topics containing general information about the application. For example, if your Help file contains a topic that describes how to use the keyboard, you can place a Keyboard menu item on the Help menu. To support additional menu items, your application must specify either the context string or the context identifier for the corresponding topic when it calls the **WinHelp** function. The following example uses a Help macro to specify the context string IDM_HELP_KEYBOARD for the Keyboard topic:

```
case IDM_HELP_KEYBOARD:
    WinHelp(hwnd, "myhelp.hlp", HELP_COMMAND,
            (LPSTR)"JumpID(\"myhelp.hlp\",\"IDM_HELP_KEYBOARD\")");
    return 0L;
```

A better way to display a topic is to use a context identifier. To do this, the Help file must assign a unique number to the corresponding context string, in the [MAP] section of the project file. For example, the following section assigns the number 101 to the context string IDM_HELP_KEYBOARD:

```
[MAP]
IDM_HELP_KEYBOARD   101
```

An application can display the Keyboard topic by specifying the context identifier in the call to the **WinHelp** function, as in the following example:

```
#define IDM_HELP_KEYBOARD 101

WinHelp(hwnd, "myhelp.hlp", HELP_CONTEXT, (DWORD)IDM_HELP_KEYBOARD);
```

To make maintenance of an application easier, most programmers place their defined constants (such as IDM_HELP_KEYBOARD in the previous example) in a single header file. As long as the names of the defined constants in the header file are identical to the context strings in the Help file, you can include the header file in the [MAP] section to assign context identifiers, as shown in the following example:

```
[MAP]
#include <myhelp.h>
```

If a Help file contains two or more Contents topics, the application can assign one as the default by using the context identifier and the HELP_SETCONTENTS value in a call to the **WinHelp** function.

Choosing Help with the Keyboard

An application can enable the user to choose a Help topic with the keyboard by intercepting the F1 key. Intercepting this key lets the user select a menu, menu item, dialog box, message box, or control window and view Help for it with a single keystroke.

To intercept the F1 key, the application must install a message-filter procedure by using the **SetWindowsHook** function. This allows the application to examine all keystrokes for the application, regardless of which window has the input focus. If the filter procedure detects the F1 key, it posts a WM_F1DOWN message (application-defined) to the application's main window procedure. The procedure then determines which Help topic to display.

The filter procedure should have the following form:

```
int FAR PASCAL FilterFunc(nCode, wParam, lParam)
int nCode;
WORD wParam;
DWORD lParam;
{
    LPMSG lpmsg = (LPMSG)lParam;

    if ((nCode == MSGF_DIALOGBOX || nCode == MSGF_MENU) &&
            lpmsg->message == WM_KEYDOWN && lpmsg->wParam == VK_F1) {
        PostMessage(hWnd, WM_F1DOWN, nCode, 0L);
    }

    DefHookProc(nCode, wParam, lParam, &lpFilterFunc);

    return 0;
}
```

The application should install the filter procedure after creating the main window, as shown in the following example:

```
lpProcInstance = MakeProcInstance(FilterFunc, hInstance);
if (lpProcInstance == NULL)
    return FALSE;

lpFilterFunc = SetWindowsHook(WH_MSGFILTER, lpProcInstance);
```

Like all callback functions, the filter procedure must be exported by the application.

The filter procedure sends a WM_F1DOWN message only when the F1 key is pressed in a dialog box, message box, or menu. Many applications also display the Contents topic if no menu, dialog box, or message box is selected when the user presses the F1 key. In this case, the application should define the F1 key as an accelerator key that starts Help.

To create an accelerator key, the application's resource-definition file must define an accelerator table, as follows:

```
1 ACCELERATORS
BEGIN
    VK_F1, IDM_HELP_CONTENTS, VIRTKEY
END
```

To support the accelerator key, the application must load the accelerator table by using the **LoadAccelerators** function and translate the accelerator keys in the main message loop by using the **TranslateAccelerator** function.

In addition to installing the filter procedure, the application must keep track of which menu, menu item, dialog box, or message box is currently selected. In other words, when the user selects an item, the application must set a global variable indicating the current context. For dialog and message boxes, the application should set the global variable immediately before calling the **DialogBox** or **MessageBox** function. For menus and menu items, the application should set the variable whenever it receives a WM_MENUSELECT message. As long as identifiers for all menu items and controls in an application are unique, an application can use code similar to the following example to monitor menu selections:

```
case WM_MENUSELECT:
    /*
     * Set dwCurrentHelpId to the Help ID of the menu item that is
     * currently selected.
     */

    if (HIWORD(lParam) == 0)             /* no menu selected    */
        dwCurrentHelpId = ID_NONE;
    else if (lParam & MF_POPUP) {        /* pop-up selected     */
        if ((HMENU)wParam == hMenuFile)
            dwCurrentHelpId = ID_FILE;
        else if ((HMENU)wParam == hMenuEdit)
            dwCurrentHelpId = ID_EDIT;
        else if ((HMENU)wParam == hMenuHelp)
            dwCurrentHelpId = ID_HELP;
        else
            dwCurrentHelpId = ID_SYSTEM;
    }

    else                                 /* menu item selected */
        dwCurrentHelpId = wParam;

    break;
```

In this example, the *hMenuFile*, *hMenuEdit*, and *hMenuHelp* parameters must previously have been set to specify the corresponding menu handles. An application can use the **GetMenu** and **GetSubMenu** functions to retrieve these handles.

When the main window procedure finally receives a WM_F1DOWN message, it should use the current value of the global variable to display a Help topic. The application can also provide Help for individual controls in a dialog box by determining which control has the focus at this point, as shown in the following example:

```
case WM_F1DOWN:
    /*
     * If there is a current Help context, display it.
     */

    if (dwCurrentHelpId != ID_NONE) {
        DWORD dwHelp = dwCurrentHelpId;

        /*
         * Check for context-sensitive Help for individual dialog
         * box controls.
         */

        if (wParam == MSGF_DIALOGBOX) {
            WORD wID = GetWindowWord(GetFocus(), GWW_ID);
            if (wID != IDOK && wID != IDCANCEL)
                dwHelp = (DWORD) wID;
        }

        WinHelp(hWnd, szHelpFileName, HELP_CONTEXT, dwHelp);

        /*
         * This call is used to remove the highlighting from the
         * System menu, if necessary.
         */

        DrawMenuBar(hWnd);
    }

    break;
```

When the application ends, it must remove the filter procedure by using the **UnhookWindowsHook** function and free the procedure instance for the function by using the **FreeProcInstance** function.

Choosing Help with the Mouse

An application can enable the user to choose a Help topic with the mouse by intercepting mouse input messages and calling the **WinHelp** function. To distinguish requests to view Help from regular mouse input, the user must press the SHIFT+F1 key combination. In such cases, the application sets a global variable when the user presses the key combination and changes the cursor shape to a question-mark pointer to indicate that the mouse can be used to choose a Help topic.

To detect the SHIFT+F1 key combination, an application checks for the VK_F1 virtual-key value in each WM_KEYDOWN message sent to its main window procedure. It also checks for the VK_ESCAPE virtual-key code. The user presses the ESC key to quit Help and restore the mouse to its regular function. The following example checks for these keys:

```
case WM_KEYDOWN:
    if (wParam == VK_F1) {

        /* If Shift-F1, turn Help mode on and set Help cursor. */

        if (GetKeyState(VK_SHIFT)) {
            bHelp = TRUE;
            SetCursor(hHelpCursor);
            return DefWindowProc(hwnd, message, wParam, lParam);
        }

        /* If F1 without shift, call Help main index topic. */

        else {
            WinHelp(hwnd,"myhelp.hlp",HELP_CONTENTS,0L);
        }
    }
    else if (wParam == VK_ESCAPE && bHelp) {

        /* Escape during Help mode: turn Help mode off. */

        bHelp = FALSE;
        SetCursor((HCURSOR) GetClassWord(hWnd, GCW_HCURSOR));
    }

    break;
```

Until the user clicks the mouse or presses the ESC key, the application responds to WM_SETCURSOR messages by resetting the cursor to the arrow and question-mark combination.

```
case WM_SETCURSOR:
    /*
     * In Help mode, it is necessary to reset the cursor in response
     * to every WM_SETCURSOR message. Otherwise, by default, Windows
     * will reset the cursor to that of the window class.
     */

    if (bHelp) {
        SetCursor(hHelpCursor);
        break;
    }

    return (DefWindowProc(hwnd, message, wParam, lParam));

case WM_INITMENU:
    if (bHelp) {
        SetCursor(hHelpCursor);
    }

    return (TRUE);
```

If the user clicks the mouse button in a nonclient area of the application window while in Help mode, the application receives a WM_NCLBUTTONDOWN message. By examining the *wParam* value of this message, the application can determine which context identifier to pass to **WinHelp**.

```
case WM_NCLBUTTONDOWN:
    /*
     * If in Help mode (Shift+F1), display context-sensitive
     * Help for nonclient area.
     */
```

```
    if (bHelp) {
        dwHelpContextId =
            (wParam == HTCAPTION) ?(DWORD) HELPID_TITLE_BAR:
            (wParam == HTSIZE) ? (DWORD) HELPID_SIZE_BOX:
            (wParam == HTREDUCE) ? (DWORD) HELPID_MINIMIZE_ICON:
            (wParam == HTZOOM) ? (DWORD) HELPID_MAXIMIZE_ICON:
            (wParam == HTSYSMENU) ?(DWORD) HELPID_SYSTEM_MENU:
            (wParam == HTBOTTOM) ? (DWORD) HELPID_SIZING_BORDER:
            (wParam == HTBOTTOMLEFT) ? (DWORD) HELPID_SIZING_BORDER:
            (wParam == HTBOTTOMRIGHT) ?(DWORD) HELPID_SIZING_BORDER:
            (wParam == HTTOP) ?(DWORD) HELPID_SIZING_BORDER:
            (wParam == HTLEFT) ?(DWORD) HELPID_SIZING_BORDER:
            (wParam == HTRIGHT) ?(DWORD) HELPID_SIZING_BORDER:
            (wParam == HTTOPLEFT) ?(DWORD) HELPID_SIZING_BORDER:
            (wParam == HTTOPRIGHT) ? (DWORD) HELPID_SIZING_BORDER:
            (DWORD) 0L;

        if (!((BOOL) dwHelpContextId))
            return DefWindowProc(hwnd, message, wParam, lParam);
        bHelp = FALSE;
        WinHelp(hWnd, szHelpFileName, HELP_CONTEXT, dwHelpContextId);
        break;
    }

    return (DefWindowProc(hWnd, message, wParam, lParam));
```

If the user clicks a menu item while in Help mode, the application intercepts the WM_COMMAND message and sends the Help request:

```
case WM_COMMAND:

    /* In Help mode (Shift-F1)? */

    if (bHelp) {
        bHelp = FALSE;
        WinHelp(hWnd, szHelpFileName, HELP_CONTEXT, (DWORD)wParam);
        return NULL;
    }
```

Searching for Help with Keywords

An application can enable the user to search for Help topics based on full or partial keywords. This method is similar to employing the Search dialog box in Windows Help to find useful topics. The following example searches for the keyword "Keyboard" and displays the corresponding topic, if found:

```
WinHelp(hwnd, "myhelp.hlp", HELP_KEY, "Keyboard");
```

If the topic is not found, Windows Help displays an error message. If more than one topic has the same keyword, Windows Help displays only the first topic.

An application can give the user more options in a search by specifying partial keywords. When a partial keyword is given, Windows Help usually displays the Search dialog box to allow the user to continue the search or return to the application. However, if there is an exact match and no other topic exists with the given keyword, Windows Help displays the topic. The following example opens the Search dialog box and selects the first keyword in the list starting with the letters *Ke*:

```
WinHelp(hwnd, "myhelp.hlp", HELP_PARTIALKEY, "Ke");
```

When the HELP_KEY and HELP_PARTIALKEY values are specified in the **WinHelp** function, Windows Help searches the K keyword table. This table contains keywords generated by using the letter K with **\footnote** statements in the topic file. An application can search alternative keyword tables by specifying the HELP_MULTIKEY value in the **WinHelp** function. In this case, the application must specify the footnote character and the full keyword in a **MULTIKEYHELP** structure, as follows:

```
HGLOBAL hglblmkh;
MULTIKEYHELP FAR* mkh;
PSTR pszKeyword = "Frame";
UINT cb;

cb = sizeof(MULTIKEYHELP) + lstrlen(pszKeyword);

hglblmkh = GlobalAlloc(GHND, (DWORD) cb);
if (hglblmkh == NULL)
    break;
mkh = (MULTIKEYHELP FAR*) GlobalLock(hglblmkh);

mkh->mkSize    = cb;
mkh->mkKeylist = 'L';
lstrcpy(mkh->szKeyphrase, pszKeyword);

WinHelp(hwnd, "myhelp.hlp", HELP_MULTIKEY, (DWORD) mkh);

GlobalUnlock(hglblmkh);
GlobalFree(hglblmkh);
```

If the keyword is not found, Windows Help displays an error message. If more than one topic has the keyword, Windows Help displays only the first topic. (For a full description of the **MULTIKEYHELP** structure, see the *Microsoft Programmer's Reference*.)

Applications cannot use alternative keyword tables unless the **MULTIKEY** option is specified in the [OPTIONS] section of the project file.

Displaying Help in a Secondary Window

An application can display Help topics in secondary windows instead of in the Windows Help main window. Secondary windows are useful whenever the user does not need the full capabilities of Windows Help. The Windows Help menus and buttons are not available in secondary windows.

To display Help in a secondary window, the application specifies the name of the secondary window along with the name of the Help file. The following example displays the Help topic having the context identifier IDM_FILE_SAVE in the secondary window named "wnd_menu":

```
WinHelp(hwnd, "myhelp.hlp>wnd_menu", HELP_CONTEXT, IDM_FILE_SAVE);
```

The name and characteristics of the secondary window must be defined in the [WINDOWS] section of the project file, as in the following example:

```
[WINDOWS]
wnd_menu = "Menus", (128, 128, 256, 256), 0
```

Windows Help displays the secondary window with the initial size and position specified in the [WINDOWS] section. However, an application can set a new size and position by specifying the HELP_SETWINPOS value in the **WinHelp** function. In this case, the application sets the members in a **HELPWININFO** structure to specify the window size and position. The following examples sets the secondary window wnd_menu to a new size and position:

```
HANDLE hhwi;
LPHELPWININFO lphwi;
WORD wSize;
char *szWndName = "wnd_menu";

wSize = sizeof(HELPWININFO) + lstrlen(szWndName);
hhwi = GlobalAlloc(GHND, wSize);
lphwi = (LPHELPWININFO)GlobalLock(hhwi);

lphwi->wStructSize = wSize;
lphwi->x           = 256;
lphwi->y           = 256;
lphwi->dx          = 767;
lphwi->dy          = 512;
lphwi->wMax        = 0;
lstrcpy(lphwi->rgchMember, szWndName);

WinHelp(hwnd, "myhelp.hlp", HELP_SETWINPOS, lphwi);

GlobalUnlock(hhwi);
GlobalFree(hhwi);
```

Canceling Help

Windows Help requires an application to explicitly cancel Help so that Windows Help can free any resources it used to keep track of the application and its Help files. The application can do this at any time.

An application cancels Windows Help by calling the **WinHelp** function and specifying the HELP_QUIT value, as shown in the following example:

```
WinHelp(hwnd, "myhelp.hlp", HELP_QUIT, NULL);
```

If the application has made any calls to the **WinHelp** function, it must cancel Help before it closes its main window (for example, in response to the WM_DESTROY message in the main window procedure). An application needs to call **WinHelp** only once to cancel Help, no matter how many Help files it has opened. Windows Help remains running until all applications or dynamic-link libraries that have called the **WinHelp** function have canceled Help.

Windows Help Statements and Macros

This section describes the syntax and purpose of statements and macros used in topic and project files for the Microsoft Windows Help application. The Windows Help statements define the format and placement of text and graphics in the Help file. The Windows Help macros define actions to take while the Help file is being viewed, such as creating custom buttons and carrying out menu commands.

Help Statement Syntax

Windows Help statements are an extended subset of tokens defined by the rich-text-format (RTF) standard. The statements specify character and paragraph properties, such as font, color, spacing, and alignment for text in the Help file.

The Help statements are presented to the Microsoft Help Compiler in topic files, which are specified in the [FILES] section of a project file. A topic file consists of statements, groups, and unformatted text. Each statement consists of a backslash (\) followed by a statement name. For example, the following line demonstrates usage of the **\tab** statement:

```
left column\tab right column
```

Statements must be separated from subsequent text or statement parameters by a delimiter. A delimiter can be one of the following:

- A space.
- A digit or minus sign, which indicates that a numeric parameter follows. The subsequent digit sequence is then delimited by a space or character other than a letter or digit.
- Any character other than a letter or digit.

When a space is used as a delimiter, the Microsoft Help Compiler discards it. If any other character is used, the compiler processes it as text or the start of another statement. For example, if a backslash is used as a delimiter, the compiler interprets it as the beginning of the next statement.

A group consists of Help statements and text enclosed in braces (**{}**). Formatting specified within a group affects only the text within that group. Text within a group inherits any formatting of the text preceding the group.

Unformatted text consists of any combination of 7-bit ASCII characters. Although characters whose values are greater than 127 are not permitted in topic files, the **\'** statement can be used to insert them in the final Help file. The Microsoft Help Compiler treats spaces as part of the text, but it discards carriage return and linefeed characters.

Although the Microsoft Help Compiler supports many RTF tokens, it does not support them all. The compiler ignores any RTF statement that is not explicitly defined in this section. Furthermore, the compiler may interpret an RTF token differently than it is specified by the standard. For example, the standard specifies that the **\uldb** statement indicates a double underline, but the Microsoft Help Compiler uses this statement to indicate a hot spot.

Help Macro Syntax

Windows Help macros specify actions that Windows Help takes when it loads Help or displays a topic. (Help macros may also be executed when the user selects a hotspot or when the user clicks on a designated segmented-graphic.) A Help macro consists of a macro name and parameters enclosed in parentheses.

Macro names specify the action to take, such as creating buttons or inserting menu items. The names are not sensitive to case, so any combination of uppercase and lowercase letters may be used.

Macro parameters specify the files, buttons, menus, or topics on which to carry out the action. The parameters must be enclosed in parentheses and separated by spaces. Parameters in many macros must also be enclosed in quotation marks. This is especially true if the parameter contains space characters. The valid quotation characters are the matching double quotation marks (**" "**) and the opening and closing single quotation marks (**' '**). If a quotation character is needed as part of a parameter rather than to enclose a parameter, the parameter should be enclosed in single quotation marks. When you use single quotation marks in this case, you can omit the backslash escape character for the double quotation marks, as shown in the following:

```
'command "string as parameter"'
```

Macros can be used as parameters in other macros. In most cases, embedded macros must be enclosed in quotation marks. If the embedded macro also has quoted parameters, the quotation character that is used must be different than the quotation characters enclosing the macro. The following example shows the correct way to use nested quotation marks:

```
CreateButton("time_btn", "&Time", "ExecProgram('clock', 0)")
```

A Help macro and all of its parameters must not exceed 512 characters.

Help macros can be combined into macro strings by separating the macros with semicolons (**;**). The Microsoft Help Compiler processes the macro string as a unit and executes the individual macros sequentially.

Help Project File Reference

This section describes the different sections and options in a Help project file and gives examples of their use. The entries are in alphabetic order.

[ALIAS]

The [ALIAS] section assigns one or more context strings to the same topic alias. This section is optional.

Syntax

context_string **=** *alias*
. . .

Parameters

context_string
Specifies the context string that identifies a particular topic. This context string may be used in a hotspot or in the [MAP] section to refer to a particular topic.

alias
Specifies the alternative string or alias name. This string is used in the **\footnote** statement. An alias string has the same form and follows the same conventions as the topic context string. That is, it is not case sensitive and may contain the alphabetic characters A through Z, the numeric characters 0 through 9, and the period and underscore characters.

Remarks

Because context strings must be unique for each topic and cannot be used for any other topic in the Help project, the [ALIAS] section provides a way to delete or combine Help topics without recoding your files. For example, if you create a topic that replaces information in three other topics, you could manually search through your files for invalid cross-references to the deleted topics. The easier approach, however, would be to use the [ALIAS] section to assign the name of the new topic to the deleted topics.

The [ALIAS] section can also be used when your application has multiple context identifiers for one Help topic. This situation occurs in context-sensitive Help.

Alias names can be used in a [MAP] section, but only if the [ALIAS] section precedes the [MAP] section.

Example

The following example creates several aliases:

```
[ALIAS]
sm_key=key_shrtcuts
cc_key=key_shrtcuts
st_key=key_shrtcuts     ; combined into Keyboard Shortcuts topic
clskey=us_dlog_bxs
maakey=us_dlog_bxs      ; covered in Using Dialog Boxes topic.
chk_key=dlogprts
drp_key=dlogprts
lst_key=dlogprts
opt_key=dlogprts
tbx_key=dlogprts        ; combined into Parts of Dialog Box topic.
frmtxt=edittxt
wrptxt=edittxt
seltxt=edittxt          ; covered in Editing Text topic.
```

See Also

\footnote, [MAP]

[BAGGAGE]

The [BAGGAGE] section lists files (typically multimedia elements) that the Microsoft Help Compiler stores within the Help file's internal file system. Windows Help can access data files stored in the Help file more efficiently than it can access files in the normal file system, since it doesn't have to read the file allocation table from CD-ROM.

Syntax

filename
. . .

Parameter

filename
Specifies the full path of a file. If a file cannot be found, the compiler reports an error.

Remarks

A maximum of 1000 files can be stored as baggage files.

If a file is listed in the [BAGGAGE] section, you must use or write a dynamic-link library that uses Windows Help to read these files from the Help file.

See Also

ROOT

[BITMAPS]

The [BITMAPS] section specifies the names and locations of the bitmap files specified in the **bmc**, **bml**, and **bmr** statements.

Syntax

filename
. . .

Parameter

filename
Specifies the full path of a bitmap file. If a file cannot be found, the compiler reports an error.

Remarks

For Windows 3.1, the [BITMAPS] section is not required if the bitmaps are located in the Help project directory or if the path containing the bitmaps is listed in the **BMROOT** or **ROOT** option. If the project file does not include either of these options, each bitmap filename must be listed in the [BITMAPS] section of the project file.

Example

The following example specifies three bitmap files:

```
[BITMAPS]
BMP01.BMP
BMP02.BMP
BMP03.BMP
```

See Also

bmc, **bml**, **bmr**, **BMROOT**, **ROOT**

BMROOT

The **BMROOT** option specifies the directory containing the bitmap files specified in the **bmc**, **bml**, and **bmr** statements.

Syntax

BMROOT = *path*[**,** *path*]

Parameter

path
Specifies a drive and full path.

Remarks

If the project file has a **BMROOT** option, you do not need to list the bitmap files in the [BITMAPS] section.

If the project file does not have a **BMROOT** option, the Help compiler looks for bitmaps in the directories specified by the **ROOT** option. If the project file does not have a **ROOT** option or if the **ROOT** option does not specify the directory containing the bitmap files, the filename for each bitmap must be specified in the [BITMAPS] section.

Example

The following example specifies that bitmaps are in the \HELP\BMP directory on drive C and the \GRAPHICS\ART directory on drive D:

```
[OPTIONS]
BMROOT=C:\HELP\BMP, D:\GRAPHICS\ART
```

See Also

[BITMAPS], **bmc**, **bml**, **bmr**, [OPTIONS], **ROOT**

BUILD

The **BUILD** option specifies which topics containing build tags are included in a build. The **BUILD** option does not apply to topics that do not contain build tags.

A topic contains a build tag if it contains a build-tag **\footnote** statement. Topics without build tags are always compiled, regardless of the current build expression.

Syntax

BUILD = *expression*

Parameter

expression
Specifies the build expression. This parameter consists of a combination of build tags (specified in the [BUILDTAGS] section) and the following operators:

Operator	Description
~	Applies the NOT operator to a single tag. The Help compiler compiles a topic only if the tag is *not* present. This operator has the highest precedence; the compiler applies it before any other operator.
&	Combines two tags by using the AND operator. The Help compiler compiles a topic only if it contains both tags. The compiler applies this operator only after the ~ operator has been applied.
|	Combines two tags by using the OR operator. The Help compiler compiles a topic if it has at least one tag. This operator has the lowest precedence; the compiler applies it only after all other operators have been applied.
( )	Parentheses may be used to override operator precedence. Expressions enclosed in parentheses are always evaluated first.

Remarks

Only one **BUILD** option can be given per project file.

The Help compiler evaluates all build expressions from left to right, using the specified precedence rules.

Example

The following examples assume that the [BUILDTAGS] section in the project file defines the build tags DEMO, MASTER, and TEST_BUILD. Although the following examples show several **BUILD** options on consecutive lines, only one **BUILD** option per project file is allowed.

```
BUILD = DEMO             ; compile topics that have the DEMO tag
BUILD = DEMO & MASTER    ; compile topics with both DEMO and MASTER
BUILD = DEMO | MASTER    ; compile topics with either DEMO or MASTER
BUILD = ~DEMO            ; compile topics that do not have DEMO
BUILD = (DEMO | MASTER) & TEST_BUILD
                         ; compile topics that have TEST_BUILD and
                         ; either DEMO or MASTER
```

See Also

\footnote, [BUILDTAGS], [OPTIONS]

[BUILDTAGS]

The [BUILDTAGS] section defines the build tags for the Help file. The Help compiler uses these tags to determine which topics to include when building the Help file.

This section is used in conjunction with the build-tag **\footnote** statements. These **\footnote** statements associate a build tag with a given topic. If the build tag is also defined in the [BUILDTAGS] section, the Help compiler compiles the topic; otherwise, it ignores the topic.

Syntax

tag

. . .

Parameter

tag

Specifies a build tag consisting of any combination of characters except spaces. The Help compiler strips any space characters from the tag. Also, the compiler treats uppercase and lowercase characters as the same characters (that is, it is case-insensitive).

Remarks

The [BUILDTAGS] section is optional. If given, it can contain up to 30 build tags.

Example

The following example shows the form of the [BUILDTAGS] section in a sample project file:

```
[BUILDTAGS]
DEMO            ; topics to include in demo build
MASTER          ; topics to include in master help file
DEBUGBUILD      ; topics to include in debugging build
TESTBUILD       ; topics to include in a mini-build for testing
```

See Also

\footnote, **BUILD**

CITATION

The **CITATION** option places a custom citation in the About dialog box of Windows Help. Windows Help displays the citation immediately below the Microsoft copyright notice.

Syntax

CITATION = *citation*

Parameter

citation
Specifies the citation. The notice can be any combination of characters; its length must be in the range 35 through 75 characters.

See Also

COPYRIGHT, [OPTIONS]

COMPRESS

The **COMPRESS** option specifies the level of compression to be used when building the Help file. Compression levels indicate either no compression, medium compression (approximately 40%), or high compression (approximately 50%).

Syntax

COMPRESS = *compression-level*

Parameter

compression-level
Specifies the level of compression. This parameter can be one of the following values:

Value	Meaning
0	No compression
1	High compression
FALSE	No compression
HIGH	High compression
MEDIUM	Medium compression
NO	No compression
TRUE	High compression
YES	High compression

Remarks

Depending on the degree of compression requested, the build uses either block compression or a combination of block and key-phrase compression. Block compression compresses the topic data into predefined units known as blocks. Key-phrase compression combines repeated phrases found within the source file(s). The compiler creates a phrase-table file with the .PH extension if one does not already exist. If the compiler finds a file with the .PH extension, it uses that file for the current compilation. Because the .PH file speeds up the compression process when little text has changed since the last compilation, you might want to keep the phrase file if you compile the same Help file several times with compression. However, you will get maximum compression if you delete the .PH file before starting each build.

See Also

[OPTIONS]

[CONFIG]

The [CONFIG] section contains one or more macros that carry out actions, such as enabling browse buttons and registering dynamic-link library (DLL) functions. Windows Help executes the macros when it opens the Help file.

Syntax

macro
. . .

Parameter

macro
Specifies a Windows Help macro. For more information about these macros, see the *Microsoft Windows Programmer's Reference*, *Volume 4*.

Remarks

The [CONFIG] section may include any number of lines. Each line of the [CONFIG] section may be up to 254 characters long.

Example

The following example registers a DLL, creates a button, enables the browse buttons, and sets the name of the Help file containing information about how to use Help:

```
[CONFIG]
RegisterRoutine("bmp","HDisplayBmp","USSS")
RegisterRoutine("bmp","CopyBmp", "v=USS")
CreateButton("btn_up", "&Up", "JumpContents('HOME.HLP')")
BrowseButtons()
SetHelpOnFile("APPHELP.HLP")
```

CONTENTS

The **CONTENTS** option identifies the context string of the highest-level or Contents topic. This topic is usually a table of contents or index within the Help file. Windows Help displays the Contents topic whenever the user clicks the Contents button.

Syntax

CONTENTS = *context-string*

Parameter

context-string
Specifies the context string of a topic in the Help file. The string can be any combination of characters, except spaces, and must also be specified in a context-string **\footnote** statement in some topic in the Help file.

Remarks

If the [OPTIONS] section does not include a **CONTENTS** option, the compiler assumes that the Contents topic is the first topic encountered in the first listed topic file in the [FILES] section of the project file.

The **CONTENTS** option is equivalent to the **INDEX** option that was available in Windows version 3.0.

Example

The following example sets the topic containing the context string "main_contents" as the Contents topic:

```
CONTENTS=main_contents
```

See Also

\footnote, [FILES], [OPTIONS]

COPYRIGHT

The **COPYRIGHT** option places a custom copyright notice in the About dialog box of Windows Help. Windows Help displays the notice immediately below the Microsoft copyright notice.

Syntax

COPYRIGHT = *copyright-notice*

Parameter

copyright-notice
Specifies the copyright notice. The notice can be any combination of characters; its length must be in the range 35 through 75 characters.

Remarks

The copyright notice is also appended to topics that are copied to the clipboard, unless it is replaced by using the **CITATION** option.

See Also

CITATION, [OPTIONS]

ERRORLOG

The **ERRORLOG** option directs the Help compiler to write all error messages to the specified file. The compiler also displays the error messages on the screen.

Syntax

ERRORLOG = *error-filename*

Parameter

error-filename
Specifies the name of the file to receive the error messages. This parameter can be a full or partial path if the error file should be written to a directory other than the project root directory.

Example

The following example writes all errors during the build to the HLPBUGS.TXT file in the Help project root directory.

```
ERRORLOG=HLPBUGS.TXT
```

See Also

[OPTIONS]

[FILES]

The [FILES] section lists all topic files used to build the Help file. Every project file requires a [FILES] section.

Syntax

filename
    . . .

Parameter

filename
Specifies the full or partial path of a topic file. If a partial path is given, the Help compiler uses the directories specified by the **ROOT** option to construct a full path. If a file cannot be found, the compiler reports an error.

Remarks

The **#include** directive can also be used in the [FILES] section to specify the topic files indirectly by designating a file that contains a list of the topic files.

Example

The following example specifies four topic files:

```
[FILES]
rtftxt\COMMANDS.RTF
rtftxt\HOWTO.RTF
rtftxt\KEYS.RTF
rtftxt\GLOSSARY.RTF
```

The following example uses the **#include** directive to specify the topic files indirectly. In this case, the file RTFFILES.H must be in the project directory (the Help compiler does not use the INCLUDE environment variable to search for files).

```
[FILES]
#include <rtffiles.h>
```

See Also

#include, **ROOT**

FORCEFONT

The **FORCEFONT** option forces the specified font to be substituted for all requested fonts. The option is used to create Help files that can be viewed on systems that do not have all fonts available.

Syntax

FORCEFONT = *fontname*

Parameter

fontname

Specifies the name of an available font. Font names must be spelled the same as they are in the Fonts dialog box in Control Panel. Font names cannot exceed 20 characters. If an invalid font name is given, the Help compiler uses the MS Sans Serif font as the default.

See Also

[OPTIONS]

ICON

The **ICON** option identifies the icon to display when the user minimizes Windows Help.

Syntax

ICON = *icon-file*

Parameter

icon-file
Specifies the name of the icon file. This file must have the standard Windows icon-file format.

See Also

[OPTIONS]

LANGUAGE

The **LANGUAGE** option sets the sorting order for keywords in the Search dialog box.

Syntax

LANGUAGE = *language-name*

Parameter

language-name
Specifies the language on which to base sorting. This parameter can be **scandinavian**, which sets the sorting order to the Scandinavian-language order.

Remarks

The default sorting order is the English-language order.

Microsoft Windows Help version 3.1 supports only English and Scandinavian sorting.

See Also

[OPTIONS]

[MAP]

The [MAP] section associates context strings (or aliases) with context numbers for context-sensitive Help. The context number corresponds to a value the parent application passes to Windows Help in order to display a particular topic. This section is optional.

Syntax

context-string context-number
. . .

Parameter

context-string
Specifies the context string of a topic in the Help file. The string can be any combination of characters, except spaces, and must also be specified in a context-string **\footnote** statement in some topic in the Help file.

context-number
Specifies the context number to associate with the context string. The number can be in either decimal or standard C hexadecimal format. Only one context number may be assigned to a context string or alias. Assigning the same number to more than one context string generates a compiler error. At least one space must separate the context number from the context string.

Remarks

You can define the context strings listed in the [MAP] section either in a Help topic or in the [ALIAS] section. The compiler generates a warning message if a context string appearing in the [MAP] section is not defined in any of the topic files or in the [ALIAS] section.

If you use an alias name, the [ALIAS] section must precede the [MAP] section in the Help project file.

The [MAP] section supports two additional statements for specifying context strings and their associated context numbers. The first statement has the following form:

#define *context-string context-number*

The *context-string* and *context-number* parameters are as described in the Parameters section.

The second statement has the following form:

#include "*filename***"**

The *filename* parameter, which can be enclosed in either double quotation marks or angle brackets (**<>**), specifies the name of a file containing one or more **#define** statements. The file may contain additional **#include** statements as well, but files may not be nested in this way more than five deep.

Example

The following example assigns hexadecimal context numbers to the context strings:

```
[MAP]
Edit_Window     0x0001
Control_Menu    0x0002
Maximize_Icon   0x0003
Minimize_Icon   0x0004
Split_Bar       0x0005
Scroll_Bar      0x0006
Title_Bar       0x0007
Window_Border   0x0008
```

See Also

#define, **#include**, [ALIAS]

MAPFONTSIZE

The **MAPFONTSIZE** option maps font sizes specified in topic files to different sizes when they are displayed in the Help window. This option is especially useful if there is a significant size difference between the authoring display and the intended user display.

Syntax

MAPFONTSIZE = *m*:*p*

Parameter

m
Specifies the size of the source font. This parameter is either a single point size or a range of point sizes. A range of point sizes consists of the low and high point sizes separated by a hyphen (**-**). If a range is specified, all fonts in the range are changed to the size specified by the *p* parameter.

p
Specifies the size of the font for the Help file.

Remarks

Although the [OPTIONS] section can contain up to five font ranges, only one font size or range is allowed with each **MAPFONTSIZE** statement. If more than one **MAPFONTSIZE** statement is included, the source font size or range specified in subsequent statements cannot overlap previous mappings.

Example

The following examples illustrate the use of the **MAPFONTSIZE** option:

```
MAPFONTSIZE=8:12      ; display all 8-pt. fonts as 12-pt.
MAPFONTSIZE=12-24:16  ; display fonts from 12 to 24 pts. as 16 pts.
```

See Also

[OPTIONS]

MULTIKEY

The **MULTIKEY** option specifies the footnote character to use for an alternative keyword table. This option is intended to be used in conjunction with topic files that contain **\footnote** statements for alternative keywords.

Syntax

MULTIKEY = *footnote-character*

Parameter

footnote-character
Specifies the case-sensitive letter to be used for the keyword footnote.

Remarks

Since keyword footnotes are case-sensitive, you should limit your keyword-table footnotes to one case, usually uppercase. If an uppercase letter is specified, the compiler will not include footnotes with the lowercase form of the same letter in the keyword table.

You may use any alphanumeric character for a keyword table except *K* and *k*, which are reserved for Help's standard keyword table. There is an absolute limit of five keyword tables, including the standard table. However, depending upon system configuration and the structure of your Help system, a practical limit of only two or three tables may be more realistic. If the compiler cannot create an additional keyword table, the additional table is ignored in the build.

Example

The following example illustrates how to enable the letter *L* for a keyword-table footnote:

```
MULTIKEY=L
```

See Also

\footnote, [OPTIONS]

OLDKEYPHRASE

The **OLDKEYPHRASE** option specifies whether an existing key-phrase file should be used to build the Help file.

Syntax

OLDKEYPHRASE = *onoff*

Parameters

onoff

Specifies whether the existing file should be used. This parameter can be one of the following values:

Value	Meaning
0	Recreate the file
1	Use the existing file
FALSE	Recreate the file
NO	Recreate the file
OFF	Recreate the file
ON	Use the existing file
TRUE	Use the existing file
YES	Use the existing file

See Also

[OPTIONS]

OPTCDROM

The **OPTCDROM** option optimizes a Help file for display on CD-ROM by aligning topic files on predefined block boundaries.

Syntax

OPTCDROM = *yesvalue*

Parameter

yesvalue
Specifies that the file should be optimized for CD-ROM. This parameter can be any of the values: **YES**, **TRUE**, **1**, or **ON**.

See Also

[OPTIONS]

[OPTIONS]

The [OPTIONS] section includes options that control how a Help file is built and what feedback the build process displays. If this section is included in the project file, it should be the first section listed, so that the options will apply during the entire build process.

Syntax

option

. . .

Parameters

option

Specifies one of the following project-file options:

Option	Description
BMROOT	Specifies the directory containing the bitmap files named in the bmc, bml, and bmr statements in topic files.
BUILD	Specifies which topics to include in the build.
CITATION	Specifies a string that is appended to topics that are copied from Windows Help instead of the COPYRIGHT string.
COMPRESS	Specifies the type of compression to use during the build.
CONTENTS	Specifies the context string of the Contents topic for a Help file.
COPYRIGHT	Adds a unique copyright message for the Help file to the About dialog box.
ERRORLOG	Puts compilation errors in a file during the build.
FORCEFONT	Forces all authored fonts in the topic files to appear in a different font when displayed in the Help window.
ICON	Specifies the icon file to be displayed when the Help window is minimized.
LANGUAGE	Specifies a different sorting order for Help files authored in a Scandinavian language.
MAPFONTSIZE	Maps a font size in the topic file to a different font size in the compiled Help file.
MULTIKEY	Specifies an alternative keyword table to use for mapping topics.
OLDKEYPHRASE	Specifies whether the compiler should use the existing key-phrase table or create a new one during the build.
OPTCDROM	Optimizes the Help file for CD-ROM use.
REPORT	Controls the display of messages during the build process.
ROOT	Specifies the directories containing the topic and data files listed in the project file.
TITLE	Specifies the text displayed in the title bar of the Help window when the file is open.
WARNING	Specifies the level of error-message reporting the

compiler is to display during the build.

Remarks

These options can appear in any order within the [OPTIONS] section. The [OPTIONS] section is not required.

REPORT

The **REPORT** option displays messages on the screen during the build. These messages indicate when the Help compiler is performing the different phases of the build, including compiling the file, resolving jumps, and verifying browse sequences.

Syntax

REPORT = ON

See Also

[OPTIONS], **WARNING**

ROOT

The **ROOT** option specifies the directories where the Help compiler looks for files listed in the project file.

Syntax

ROOT = *pathname*[**,** *pathname*]

Parameter

pathname
Specifies either a drive and full path or a relative path from the project directory. If the project file has a **ROOT** option, all relative paths in the project file refer to one of these paths. If the project file does not have a **ROOT** option, all paths are relative to the directory containing the project file.

Remarks

If the project file does not have a **BMROOT** option, the compiler looks in the directories specified in the **ROOT** option to find bitmaps positioned by using the **bmc**, **bml**, and **bmr** statements. If none of these directories contains these bitmaps, the bitmap filenames must be listed in the [BITMAPS] section of the project file.

Example

The following example specifies that the project root directory is C:\WINHELP\HELPDIR and is found on drive C:

```
[OPTIONS]
ROOT=C:\WINHELP\HELPDIR
```

Given this root directory, if the [FILES] section contains the entry TOPICS\FILE.RTF, the full path for the topic file is C:\WINHELP\HELPDIR\TOPICS\FILE.RTF.

See Also

bmc, **bml**, **bmr**, [BITMAPS], **BMROOT**, [OPTIONS], [FILES]

TITLE

The **TITLE** option sets the title for the Help file. Windows Help displays the title in its title bar whenever it displays the Help file.

Syntax

TITLE = *titlename*

Parameter

titlename
Specifies the title displayed in the Windows Help title bar. The title must not exceed 50 characters.

Remarks

If no title is specified by using the **TITLE** option, Windows Help displays the title Windows Help in the title bar.

Example

The following example sets the Help-file title to ABC Help.

```
[OPTIONS]
TITLE=ABC Help
```

See Also

[OPTIONS]

WARNING

The **WARNING** option specifies the amount of debugging information the Help compiler is to report.

Syntax

WARNING = *level*

Parameter

level

Specifies the warning level. This parameter may be one of the following values:

Value	Meaning
1	Report only the most severe errors.
2	Report an intermediate number of errors.
3	Report all errors and warnings.

Example

The following example specifies an intermediate level of error reporting:

```
[OPTIONS]
WARNING=2
```

See Also

[OPTIONS], **REPORT**

[WINDOWS]

The [WINDOWS] section defines the size, location, and colors for the primary Help window and any secondary-window types used in a Help file.

The secondary windows defined in this section are intended to be used with Windows applications that specify secondary windows when calling the **WinHelp** function.

Syntax

type = **"***caption***", (***x***, *y*, *width*, *height*), *sizing*, (*clientRGB*), (*nonscrollRGB*), (*fTop*)**
. . .

Parameters

type
Specifies the type of window that uses the defined attributes. For the primary Help window, this parameter is **main**. For a secondary window, this parameter may be any unique name of up to 8 characters. Any jumps that display a topic in a secondary window give this type name as part of the jump.

caption
Specifies the title for a secondary window. Windows Help places the title in the title bar of the window. To set the title for the primary Help window, use the **TITLE** option in the [OPTIONS] section.

x
Specifies the x-coordinate, in help units, of the window's upper-left corner. Windows Help always assumes the screen is 1024 help units wide, regardless of resolution. For example, if the x-coordinate is 512, the left edge of the Help window is in the middle of the screen.

y
Specifies the y-coordinate, in help units, of the window's upper-left corner. Windows Help always assumes the screen is 1024 help units high, regardless of resolution. For example, if the x-coordinate is 512, the top edge of the Help window is in the middle of the screen.

width
Specifies the default width, in help units, for a secondary window.

height
Specifies the default height, in help units, for a secondary window.

sizing
Specifies the relative size of a secondary window when Windows Help first opens the window. This parameter can be one of the following values:

Value	Meaning
0	Set the window to the size specified by the *x*, *y*, *width*, and *height* parameters.
1	Maximize the window; ignore the *x*, *y*, *width*, and *height* parameters.

clientRGB
Specifies the background color of the window. This parameter is an RGB color value consisting of three 8-bit hexadecimal numbers enclosed in parentheses and separated by commas. If this parameter is not given, Windows Help uses the default window color specified by Control Panel.

nonscrollRGB
Specifies the background color of the nonscrolling region (if any) in the Help window. This parameter is an RGB color value consisting of three 8-bit hexadecimal numbers enclosed in parentheses and separated by commas. If this parameter is not given, Windows Help uses the default window color specified by Control Panel.

fTop

Specifies whether the secondary window is displayed on top of all other windows. When this parameter is 1, the window is displayed over all windows that do not also use this attribute. Otherwise, it should be zero. This parameter is optional.

Example

The following example defines two windows, the main window and a secondary window named "picture". The main-window definition sets the background color of nonscrolling regions in the main Help window to (128, 0, 128) but leaves several other values empty (for which Windows Help will supply its own default values). The secondary-window definition sets the caption to "Samples" and sets the width and height of the window to about one-quarter of the width and height of the screen. The background colors for the window and nonscrolling region are (0, 255, 255) and (255, 0, 0), respectively. The *sizing* parameter for both the main and secondary windows is zero.

```
[WINDOWS]
main=,,,,, 0,,,, (128, 0, 128)
picture = "Samples", (123,123,256,256), 0, (0,255,255), (255,0,0)
```

See Also

[OPTIONS], **TITLE**, **WinHelp**

Help Macro Reference

This section lists the Microsoft Windows Help macros in alphabetic order.

About

The **About** macro displays Windows Help's About dialog box.

Syntax

About()

Parameters

This macro does not take any parameters.

Remarks

Use of this macro in secondary windows is not recommended.

AddAccelerator

The **AddAccelerator** macro assigns a Help macro to an accelerator key (or key combination) so that the macro is carried out when the user presses the accelerator key(s).

Syntax

AddAccelerator(*key***,** *shift-state***, "***macro***")**

Parameters

key

Specifies the Windows virtual-key value. For a list of virtual-key codes, see the *Microsoft Windows Programmer's Reference*, *Volume 3*.

shift-state

Specifies the combination of ALT, SHIFT, and CTRL keys to be used with the accelerator. This parameter may be one of the following values:

Value	Meaning
0	None
1	SHIFT
2	CTRL
3	SHIFT+CTRL
4	ALT
5	ALT+SHIFT
6	ALT+CTRL
7	SHIFT+ALT+CTRL

macro

Specifies the Help macro or macro string executed when the user presses the accelerator key(s). The macro must appear in quotation marks ("). Multiple macros in a string must be separated by semicolons (;).

Remarks

The **AddAccelerator** macro can be abbreviated as **AA**.

Example

The following macro executes the Windows Clock program when the user presses ALT+SHIFT+CONTROL+F4:

```
AddAccelerator(0x73, 7, "ExecProgram('clock.exe', 1)")
```

See Also

RemoveAccelerator

Annotate

The **Annotate** macro displays the Annotation dialog box from the Edit menu.

Syntax

Annotate()

Parameters

This macro does not take any parameters.

Remarks

Use of this macro in secondary windows is not recommended.

AppendItem

The **AppendItem** macro appends a menu item to the end of a Windows Help menu.

Syntax

AppendItem("*menu-id***", "***item-id***", "***item-name***", "***macro***")**

Parameters

menu-id

Specifies the name of a standard Windows Help menu or the name used in the **InsertMenu** macro used to create the menu. This name must appear in quotation marks. The new item is appended to this menu. For a standard menu, this parameter can be one of the following:

Name	Menu
MNU_FILE	File
MNU_EDIT	Edit
MNU_BOOKMARK	Bookmark menu
MNU_HELP	Help

item-id

Specifies the name that Windows Help uses internally to identify the menu item. This name must appear in quotation marks (**"**). This name is used by the **DisableItem** or **DeleteItem** macros.

item-name

Specifies the name that Windows Help displays on the menu for the item. This name must appear in quotation marks (**"**). Within the quotation marks, place an ampersand (**&**) before the character used for the macro's accelerator key.

macro

Specifies one or more macros that are to be executed when the user chooses the menu item. The macro must appear in quotation marks. Multiple macros in a string must be separated by semicolons (**;**).

Remarks

Windows Help ignores this macro if it is executed in a secondary window.

If the keyboard accelerator conflicts with other menu access keys, Windows Help displays the error message "Unable to add item" and ignores the macro.

Example

The following macro appends a menu item labeled "Tools" to a pop-up menu that has an identifier "IDM_TLS". Choosing the menu item causes a jump to a topic with the context string "tpc1" in the TLS.HLP file:

```
AppendItem("IDM_BKS", "IDM_TLS", "&Tools", "JI('tls.hlp', 'tpc1')")
```

See Also

DeleteItem, **DisableItem**, **InsertMenu**

Back

The **Back** macro displays the previous topic in the history list. The history list is a list of the last 40 topics the user has displayed since starting Windows Help.

Syntax

Back()

Parameters

This macro does not take any parameters.

Remarks

Windows Help ignores this macro if it is executed in a secondary window.

If the **Back** macro is executed when the Back list is empty, Windows Help takes no action.

BookmarkDefine

The **BookmarkDefine** macro displays the Define dialog from the Bookmark menu.

Syntax

BookmarkDefine()

Parameters

This macro does not take any parameters.

Remarks

Use of this macro in secondary windows is not recommended.

If the **BookmarkDefine** macro is executed from a pop-up window, the bookmark is attached to the topic that invoked the pop-up window.

BookmarkMore

The **BookmarkMore** macro displays the More dialog from the Bookmark menu. The More command appears on the Bookmark menu if the menu lists more than nine bookmarks.

Syntax

BookmarkMore()

Parameters

This macro does not take any parameters.

Remarks

Use of the macro in secondary windows is not recommended.

BrowseButtons

The **BrowseButtons** macro adds browse buttons to the button bar.

Syntax

BrowseButtons()

Parameters

This macro does not take any parameters.

Remarks

Windows Help ignores this macro if it is executed from a secondary window.

If the **BrowseButtons** macro is used with one or more **CreateButton** macros in the [CONFIG] section of the project file, the order of the browse buttons on the Windows Help button bar is determined by the order of the **BrowseButtons** macro in relation to the other macros listed in the [CONFIG] section.

Example

The following macros in the project file cause the Clock button to appear immediately before the two browse buttons on the button bar:

```
[CONFIG]
CreateButton("&Clock", "ExecProgram('clock', 0)")
BrowseButtons()
```

See Also

[CONFIG], **CreateButton**

ChangeButtonBinding

The **ChangeButtonBinding** macro assigns a Help macro to a Help button.

Syntax

ChangeButtonBinding("*button-id***", "***button-macro***")**

Parameters

button-id

Specifies the identifier assigned to the button by the **CreateButton** macro or, for a standard Help button, one of the following predefined button identifiers:

ID	Description
BTN_CONTENTS	Contents
BTN_SEARCH	Search
BTN_BACK	Back
BTN_HISTORY	History
BTN_PREVIOUS	Browse previous
BTN_NEXT	Browse next

The button identifier must be enclosed in quotation marks (**"**).

button-macro

Specifies the Help macro executed when the user selects the button. The macro must be enclosed in quotation marks (**"**).

Remarks

Windows Help ignores this macro if it is executed in a secondary window.

The **ChangeButtonBinding** macro can be abbreviated as **CBB**.

Example

In the following macro, "conts" is the context string for the table of contents in the DICT.HLP file:

```
ChangeButtonBinding("btn_contents", "JumpId('dict.hlp', 'conts')")
```

See Also

CreateButton

ChangeItemBinding

The **ChangeItemBinding** macro assigns a Help macro to an item previously added to a Windows Help menu using the **AppendItem** macro.

Syntax

ChangeItemBinding("*item-id***", "***item-macro***")**

Parameters

item-id
Identifies the menu item appended by the **AppendItem** macro. The item identifier must be enclosed in quotation marks (**"**).

item-macro
Specifies the Help macro to execute when the user selects the item. The macro must be enclosed in quotation marks (**"**).

Remarks

Windows Help ignores this macro if it is executed in a secondary window.

The **ChangeItemBinding** macro can be abbreviated as **CIB**.

Example

The following macro changes the menu item identified by "time_item" so that it displays the Windows clock:

```
ChangeItemBinding("time_item", "ExecProgram('clock', 0)")
```

See Also

AppendItem

CheckItem

The **CheckItem** macro places a check mark beside a menu item.

Syntax

CheckItem("*item-id***")**

Parameters

item-id
Identifies the menu item to check. The item identifier must be enclosed in quotation marks (**"**).

Remarks

The **CheckItem** macro can be abbreviated as **CI**.

See Also

UncheckItem

CloseWindow

The **CloseWindow** macro closes either a secondary window or the main Help window.

Syntax

CloseWindow("*window-name***")**

Parameters

window-name
Specifies the name of the window to close. The name "main" is reserved for the main Help window. For secondary windows, the window name is defined in the [WINDOWS] section of the project file. This name must be enclosed in quotation marks (**"**).

Example

The following macro closes the secondary window named "keys":

```
CloseWindow("keys")
```

See Also

[WINDOWS]

Contents

The **Contents** macro displays the Contents topic in the current Help file. The Contents topic is defined by the **CONTENTS** option in the [OPTIONS] section of the project file. If the project file does not have a **CONTENTS** option, the Contents topic is the first topic of the first topic file specified in the project file.

Syntax

Contents()

See Also

CONTENTS, [OPTIONS]

CopyDialog

The **CopyDialog** macro displays the Copy dialog from the Edit menu.

Syntax

CopyDialog()

Remarks

Use of this macro in secondary windows is not recommended.

CopyTopic

The **CopyTopic** macro copies all the text in the currently displayed topic to the Clipboard.

Syntax

CopyTopic()

Remarks

Use of the macro in secondary windows is not recommended.

CreateButton

The **CreateButton** macro adds a new button to the button bar.

Syntax

CreateButton("*button-id***", "***name***", "***macro***")**

Parameters

button-id
Specifies the name that WinHelp uses internally to identify the button. This name must appear in quotation marks (**"**). Use this name in the **DisableButton** or **DestroyButton** macro if you want to remove or disable the button or in the **ChangeButtonBinding** if you want to change the Help macro that the button executes in certain topics.

name
Specifies the text that appears on the button. To make a letter in this text the mnemonic for the button, place an ampersand (**&**) before that letter. The button name is case sensitive and can have up to 29 characters in it − any additional characters are ignored.

macro
Specifies the Help macro or macro string executed when the user clicks on the button. Multiple macros in a macro string must be separated by semicolons.

Remarks

Windows Help allows a maximum of 16 custom buttons. It allows a total of 22 buttons, including the standard Browse buttons, on the button bar.

If the **BrowseButtons** macro is used with one or more **CreateButton** macros in the project file, the buttons appear in the same order on the button bar as the macros appear in the project file.

Windows Help ignores this macro if it is executed in a secondary window.

The **CreateButton** macro can be abbreviated as **CB**.

Example

The following macro creates a new button labeled "Ideas" that jumps to the topic with the context string "dir" in the IDEAS.HLP file when clicked:

```
CreateButton("btn_ideas", "&Ideas", "JumpId('ideas.hlp', 'dir')")
```

See Also

BrowseButtons, **ChangeButtonBinding**, **DestroyButton**, **DisableButton**

DeleteItem

The **DeleteItem** macro removes a menu item that was added by using the **AppendItem** macro.

Syntax

DeleteItem("*item-id***")**

Parameters

item-id
Specifies the item identifier used in the **AppendItem** macro. The item identifier must be enclosed in quotation marks (**"**).

Remarks

Windows Help ignores this macro if it is executed in a secondary window.

Example

The following macro removes the menu item "Tools" appended in the example for the **AppendItem** macro:

```
DeleteItem("IDM_TOOLS")
```

See Also

AppendItem

DeleteMark

The **DeleteMark** macro removes a text marker added with the **SaveMark** macro.

Syntax

DeleteMark("*marker-text***")**

Parameters

marker-text
Specifies the text marker previously added by the **SaveMark** macro. The marker text must be enclosed in quotation marks (**"**).

Remarks

If the marker does not exist when the **DeleteMark** macro is executed, Windows Help displays a "Topic not found" error message.

Example

The following macro removes the marker "Managing Memory" from a Help file:

```
DeleteMark("Managing Memory")
```

See Also

SaveMark

DisableButton

The **DisableButton** macro grays out a button added with the **CreateButton** macro. This button cannot be used in the topic until an **EnableButton** macro is executed.

Syntax

DisableButton("*button-id***")**

Parameters

button-id
Specifies the identifier assigned to the button by the **CreateButton** macro. The button identifier must be enclosed in quotation marks (**"**).

Remarks

Windows Help ignores this macro if it is executed in a secondary window.

The **DisableButton** macro can be abbreviated as **DB**.

See Also

CreateButton, **EnableButton**

DisableItem

The **DisableItem** macro grays out a menu item added with the **AppendItem** macro. The menu item cannot be used in the topic until an **EnableItem** macro is executed.

Syntax

DisableItem("*item-id***")**

Parameters

item-id
Identifies a menu item previously appended with the **AppendItem** macro. The item identifier must be enclosed in quotation marks (**"**).

Remarks

Windows Help ignores this macro if it is executed in a secondary window.

The **DisableItem** macro can be abbreviated as **DI**.

See Also

AppendItem, **EnableItem**

DestroyButton

The **DestroyButton** macro removes a button added with the **CreateButton** macro.

Syntax

DestroyButton("*button-id***")**

Parameters

button-id
Identifies a button previously created by the **CreateButton** macro. The button identifier must be enclosed in quotation marks (**"**).

Remarks

The button identifier cannot be an identifier for one of the standard Help buttons. For a list of those identifiers, see the **ChangeButtonBinding** macro.

Windows Help ignores this macro if it is executed in a secondary window.

See Also

CreateButton, **ChangeButtonBinding**

EnableButton

The **EnableButton** macro re-enables a button disabled with the **DisableButton** macro.

Syntax

EnableButton("*button-id***")**

Parameters

button-id
Specifies the identifier assigned to the button by the **CreateButton** macro. The button identifier must be enclosed in quotation marks (**"**).

Remarks

Windows Help ignores this macro if it is executed in a secondary window.

The **EnableButton** macro can be abbreviated as **EB**.

See Also

CreateButton, **DisableButton**

EnableItem

The **EnableItem** macro re-enables a menu item disabled with the **DisableItem** macro.

Syntax

EnableItem("*item-id***")**

Parameters

item-id
Specifies the identifier assigned to the menu item by the **AppendItem** macro. The item identifier must be enclosed in quotation marks (**"**).

Remarks

Windows Help ignores this macro if it is executed in a secondary window.

The **EnableItem** macro can be abbreviated as **EI**.

See Also

AppendItem, **DisableItem**

ExecProgram

The **ExecProgram** macro executes a Windows application.

Syntax

ExecProgram("*command-line***", *display-state***)**

Parameters

command-line
Specifies the command line for the application to be executed. The command line must be enclosed in quotation marks (**"**). Windows Help searches for this application in the current directory, followed by the Windows directory, the user's path, and the directory of the currently viewed Help file.

display-state
Specifies a value indicating how the application is shown when executed. It may be one of the following values:

Value	Meaning
0	Normal
1	Minimized
2	Maximized

Remarks

The **ExecProgram** macro can be abbreviated as **EP**.

The backslash (\) character should not be used to escape double quotation-mark characters in macros. Instead, you can enclose the command line in single quotation marks (**'**) and omit the backslash for the double quotation marks, as shown in the following:

```
'command "string as parameter"'
```

Note that the first single quotation mark must be an open quote and the last single quotation mark must be a close quote.

Example

The following example executes the Clock application. The application is minimized when it starts:

```
ExecProgram('clock.exe', 1)
```

Exit

The **Exit** macro exits the Windows Help application. It has the same effect as selecting Exit from the File menu.

Syntax

Exit()

Parameters

This macro does not take any parameters.

FileOpen

The **FileOpen** macro displays the Open dialog box from the File menu.

Syntax

FileOpen()

Parameters

This macro does not take any parameters.

Remarks

Use of the macro in secondary windows is not recommended.

FocusWindow

The **FocusWindow** macro changes the focus to the specified window, either the main Help window or a secondary window.

Syntax

FocusWindow("*window-name***")**

Parameters

window-name

Specifies the name of the window to receive the focus. The name "main" is reserved for the main Help window. For secondary windows, the window name is defined in the [WINDOWS] section of the project file. This name must be enclosed in quotation marks (**"**).

Remarks

This macro is ignored if the specified window does not exist.

Example

The following macro changes the focus to the secondary window "keys":

```
FocusWindow("keys")
```

See Also

[WINDOWS]

GoToMark

The **GoToMark** macro jumps to a marker set with the **SaveMark** macro.

Syntax

GoToMark("*marker-text***")**

Parameters

marker-text
Specifies a text marker previously defined by using the **SaveMark** macro.

Example

The following macro jumps to the marker "Managing Memory":

```
GoToMark("Managing Memory")
```

See Also

SaveMark

HelpOn

The **HelpOn** macro displays the Help file for the Windows Help application. The macro carries out the same action as choosing the How to Use Help command on the Help menu.

Syntax

HelpOn()

Parameters

This macro does not take any parameters.

HelpOnTop

The **HelpOnTop** macro toggles the on-top state of Windows Help. It is equivalent to checking or unchecking the Always On Top command in the Help menu.

Syntax

HelpOnTop()

Parameters

This macro does not take any parameters.

Remarks

Windows Help does not provide a macro to check the current state of the Always On Top command. It is up to the user to determine whether the macro should be used to change the state of the command.

History

The **History** macro displays the history list, which shows the last 40 topics the user has viewed since opening a Help file in Windows Help. It has the same effect as choosing the History button.

Syntax

History()

Parameters

This macro does not take any parameters.

Remarks

Windows Help ignores this macro if it is executed in a secondary window.

IfThen

The **IfThen** macro executes a Help macro if a given marker exists. It uses the **IsMark** macro to make the test.

Syntax

IfThen(IsMark("*marker-text***"), "***macro***")**

Parameters

marker-text
Specifies a text marker previously created by using the **SaveMark** macro. The marker must be enclosed in quotation marks (**"**).

macro
Specifies a Help macro or macro string to be executed if the marker exists. Multiple macros in a macro string must be separated by semicolons (**;**).

Example

The following macro jumps to the topic with context string "man_mem" if a marker named "Managing Memory" has been set by the **SaveMark** macro:

```
IfThen(IsMark("Managing Memory"), "JI('trb.hlp', 'man_mem')")
```

See Also

IsMark, **SaveMark**

IfThenElse

The **IfThenElse** macro executes one of two Help macros depending on whether or not a marker exists. It uses the **IsMark** macro to make the test.

Syntax

IfThenElse(IsMark("*marker-text***"), "***macro1***", "***macro2***")**

Parameters

marker-text
Specifies a text marker previously created by using the **IsMark** macro. The marker must be enclosed in quotation marks (**"**).

macro1
Specifies a Help macro or macro string to be executed if the marker exits. Multiple macros in either macro string must be separated by semicolons (**;**).

macro2
Specifies a Help macro or macro string to be executed if the marker does not exist. Multiple macros in either macro string must be separated by semicolons.

Example

The following macro jumps to the topic with context string "mem" if a marker named "Memory" has been set by the **SaveMark** macro. If the marker does not exist, it jumps to the next topic in the browse sequence.

```
IfThenElse(IsMark("Memory"), "JI('trb.hlp', 'mem')", "Next()")
```

See Also

IsMark, **SaveMark**

InsertItem

The **InsertItem** macro inserts a menu item at a given position on an existing menu. The menu can be either one you create with the **InsertMenu** macro or one of the standard Windows Help menus.

Syntax

InsertItem("*menu-id***", "***item-id***", "***item-name***", "***macro***",** *position***)**

Parameters

menu-id

Identifies either a standard Windows Help menu or a menu previously created by using the **InsertMenu** macro. For a standard menu, this parameter can be one of the following:

Name	Menu
MNU_FILE	File
MNU_EDIT	Edit
MNU_BOOKMARK	Bookmark menu
MNU_HELP	Help

For other menus, this parameter must be the name used with the **InsertMenu** macro. In all cases, the menu identifier must be enclosed in quotation marks. The new item is inserted into this menu.

item-id

Specifies the name that Windows Help uses internally to identify the menu item. The item identifier must be enclosed in quotation marks (**"**).

item-name

Specifies the name Windows Help displays in the menu for the item. This name is case-sensitive and must be enclosed in quotation marks. An ampersand (**&**) before a character in the name identifies it as the item's keyboard access key.

macro

Specifies a Help macro or macro string to be executed when the user chooses the menu item. The macro must be enclosed in quotation marks. Multiple macros in a string must be separated by semicolons (**;**).

position

Specifies the position of the menu item in the menu. It must be an integer value. Position 0 is the first or topmost position in the menu.

Remarks

The *item-id* parameter can be used in a subsequent **DisableItem** or **DeleteItem** macro to remove or disable the item or to change the operations that the item performs in certain topics.

Windows Help ignores this macro if it is executed in a secondary window.

The specified keyboard access keys must be unique. If a key conflicts with other menu access keys, Windows Help displays the error message "Unable to add item" and ignores the macro.

Example

The following macro inserts a menu item labeled "Tools" as the third item on a menu that has an identifier "MNU_BKS". Selecting the menu item causes a jump to a topic with the context string "tls1" in the TLS.HLP file:

```
InsertItem("mnu_bks", "m_tls", "&Tools", "JI('tls.hlp', 'tls1')", 3)
```

See Also

DeleteItem, DisableItem, InsertMenu

InsertMenu

The **InsertMenu** macro inserts a new menu in the Windows Help menu bar.

Syntax

InsertMenu("*menu-id***", "***menu-name***",** *menu-position***)**

Parameters

menu-id
Specifies the name that Windows Help uses internally to identify the menu. The menu identifier must be enclosed in quotation marks (**"**). This identifier can be used in the **AppendItem** macro to add macros to the menu.

menu-name
Specifies the name that Windows Help displays on the menu bar. This name must be enclosed in quotation marks. An ampersand (**&**) before a character in the name identifies it as the menu's keyboard access key.

menu-position
Specifies the position on the menu bar of the new menu name. This parameter must be an integer number. Positions are numbered from left to right, with position 0 the left-most menu.

Remarks

Windows Help ignores this macro if it is executed in a secondary window.

Example

The following macro adds a menu named "Utilities" to the Windows Help application. The label "Utilities" appears as the fourth item on the Windows Help menu bar. The user presses ALT+U to open the menu.

```
InsertMenu("IDM_UTIL", "&Utilities", 3)
```

See Also

AppendItem

IsMark

The **IsMark** macro tests whether or not a marker set by the **SaveMark** macro exists. It is used as a parameter to the conditional macros **IfThen** and **IfThenElse**. The **IsMark** macro returns nonzero if the mark exists or zero if it does not.

Syntax

IsMark("*marker-text***")**

Parameters

marker-text
Specifies a text marker previous created using the **SaveMark** macro.

Remarks

The **Not** macro can be used to reverse the results of the **IsMark** macro.

Example

The following macro jumps to the topic with the context string "man_mem" if a marker named "Managing Memory" has been set by the **SaveMark** macro:

```
IfThen(IsMark("Managing Memory"), "JI('trb.hlp', 'man_mem')")
```

See Also

IfThen, **IfThenElse**, **SaveMark**

JumpContents

The **JumpContents** macro jumps to the Contents topic of a specified file in the Help file. The Contents topic is indicated by the **CONTENTS** option entry in the [OPTIONS] section of the project file. If the **CONTENTS** option is not specified, Windows Help jumps to the first topic in the Help file.

Syntax

JumpContents("*filename***")**

Parameters

filename

Specifies the name of the destination file for the jump. The filename must be enclosed in quotation marks (**"**). If Windows Help cannot find this file, it displays an error message and does not perform the jump.

Remarks

Windows Help ignores this macro if it is executed in a secondary window.

Example

The following macro jumps to the Contents topic of the PROGMAN.HLP file:

```
JumpContents("PROGMAN.HLP")
```

See Also

[OPTIONS], **CONTENTS**

JumpContext

Syntax

JumpContext("*filename***", *context-number*)**

Parameters

filename
Specifies the name of the destination file for the jump. The filename must be enclosed in quotation marks (**"**). If Windows Help cannot find this file, it displays an error message and does not perform the jump.

context-number
Specifies the context number of the topic in the destination file. The context number must be defined in the [MAP] section of the project file. If the context number is not valid, Windows Help jumps to the Contents topic or to the first topic in the file instead and displays an error message.

Remarks

The **JumpContext** macro can be abbreviated as **JC**.

Example

The following macro jumps to the topic mapped to the context number 801 in the PROGMAN.HLP file:

```
JumpContext("PROGMAN.HLP", 801)
```

See Also

[MAP]

JumpHelpOn

The **JumpHelpOn** macro jumps to the Contents topic of the How to Use Help file. The How To Use Help file is either the default WINHELP.HLP file shipped with Windows 3.1 or the Help file designated by the **SetHelpOnFile** macro in the [CONFIG] section of the project file.

Syntax

JumpHelpOn()

Parameters

This macro does not take any parameters.

Remarks

If Windows Help cannot find the specified Help file, it displays an error message and does not perform the jump.

Example

The following macro jumps to the Contents topic of the designated How to Use Help file:

```
JumpHelpOn()
```

See Also

[CONFIG], **SetHelpOnFile**

JumpId

The **JumpId** macro jumps to the topic with the specified context string in the Help file.

Syntax

JumpId("*filename***", "***context-string***")**

Parameters

filename
Specifies the name of the Help file containing the context string. The filename must be enclosed in quotation marks (**"**). If Windows Help does not find this file, it displays an error message and does not perform the jump.

context-string
Context string of the topic in the destination file. The context string must be enclosed in quotation marks. If the context string does not exist, Windows Help jumps to the Contents topic for that file instead.

Remarks

The **JumpId** macro may be abbreviated as **JI**.

Example

The following macro jumps to a topic with "second_topic" as its context string in the SECOND.HLP file:

```
JI("second.hlp", "second_topic")
```

JumpKeyword

The **JumpKeyword** macro loads the indicated Help file, searches through the K keyword table, and displays the first topic containing the index keyword specified in the macro.

Syntax

JumpKeyword("*filename***", "***keyword***")**

Parameters

filename
Specifies the name of the Help file containing the desired keyword table. The filename must be enclosed in quotation marks (**"**). If this file does not exist, Windows Help displays an error message and does not perform the jump.

keyword
Specifies the keyword that the macro searches for. The keyword must be enclosed in quotation marks. If Windows Help finds more than one match, it displays the first matched topic. If it does not find any matches, it displays a "Not a keyword" message and displays the Contents topic of the destination file instead.

Remarks

The **JumpKeyword** macro can be abbreviated as **JK**.

Example

The following macro displays the first topic that has "hands" as an index keyword in the CLOCK.HLP file:

```
JumpKeyword("clock.hlp", "hands")
```

Next

The **Next** macro displays the next topic in the browse sequence for the Help file.

Syntax

Next()

Parameters

This macro does not take any parameters.

Remarks

If the currently displayed topic is the last topic of a browse sequence, this macro does nothing.

Windows Help ignores this macro if it is executed in a secondary window.

Not

The **Not** macro reverses the result (nonzero or zero) returned by the **IsMark** macro. It is used along with the **IsMark** macro as a parameter to the conditional macros **IfThen** and **IfThenElse**.

Syntax

Not(IsMark("*marker-text***"))**

Parameters

marker-text
Specifies a text marker previously created by using the **SaveMark** macro. The marker text must be enclosed in quotation marks (**"**).

Example

The following macro jumps to the topic with the context string "mem1" if a marker named "Memory" has not been set by the **SaveMark** macro:

```
IfThen(Not(IsMark("Memory")), "JI('trb.hlp', 'mem1')")
```

See Also

IfThen, **IfThenElse**, **IsMark**, **SaveMark**

PopupContext

The **PopupContext** macro displays in a pop-up window the topic identified by a specific context number.

Syntax

PopupContext("*filename***",** *context-number***)**

Parameters

filename
Specifies the name of the file that contains the topic to be displayed. The filename must be enclosed in quotation marks (**"**). If Windows Help cannot find this file, it displays an error message.

context number
Specifies the context number of the topic to be displayed. The context number must be specified in the [MAP] section of the project file. If the context number is not valid, Windows Help displays the Contents topic or the first topic in the file instead.

Remarks

The **PopupContext** macro can be abbreviated as **PC**.

Example

The following macro displays in a pop-up window the topic mapped to the context number 801 in the PROGMAN.HLP file:

```
PopupContext("progman.hlp", 801)
```

See Also

[MAP]

PopupId

The **PopupId** macro displays a topic from a specified file in a pop-up window.

Syntax

PopupId("*filename***", "***context-string***")**

Parameters

filename
Specifies the name of the file containing the pop-up window topic. The filename must be enclosed in quotation marks (**"**). If this file does not exist, Windows Help displays a warning.

context-string
Specifies the context string of the topic in the destination file. If the requested context string does not exist, Windows Help displays the Contents topic or the first topic in the file.

Remarks

The **PopupId** macro can be abbreviated as **PI**.

Example

The following macro displays a pop-up window with context string "second_topic" from the SECOND.HLP file:

```
PopupId("second.hlp", "second_topic")
```

PositionWindow

The **PositionWindow** macro sets the size and position of a window.

Syntax

PositionWindow(*x***,** *y***,** *width***,** *height***,** *state***, "***name***")**

Parameters

x
Specifies the x-coordinate, in help units, of the upper-left corner of the window. Windows Help always assumes the screen (regardless of resolution) is 1024 help units wide. For example, if the x-coordinate is 512, the left edge of the Help window is in the middle of the screen.

y
Specifies the y-coordinate, in help units, of the upper-left corner of the window. Windows Help always assumes the screen (regardless of resolution) is 1024 help units high. For example, if the y-coordinate is 512, the top edge of the Help window is in the middle of the screen.

width
Specifies the default width, in help units, of the window.

height
Specifies the default height, in help units, of the window.

state
Specifies how the window is sized. This parameter can be one of the following values:

Value	Meaning
0	Normal size
1	Maximized

If the parameter is 1, Windows Help ignores the *x*, *y*, *width*, and *height* parameters.

name
Specifies the name of the window to position. The name "main" is reserved for the main Help window. For secondary windows, the window name must be defined in the [WINDOWS] section of the project file. This name must be enclosed in quotation marks (**"**).

Remarks

If the window to be positioned does not exist, Windows Help ignores the macro.

The **PositionWindow** macro can be abbreviated as **PW**.

Example

The following macro positions the secondary window "Samples" in the upper-left corner (100, 100) with a width and height of 500 (in help units):

```
PositionWindow(100, 100, 500, 500, 0, "Samples")
```

See Also

[WINDOWS]

Prev

The **Prev** macro displays the previous topic in the browse sequence for the Help file. If the currently displayed topic is the first topic of a browse sequence, this macro does nothing.

Syntax

Prev()

Parameters

This macro does not take any parameters.

Remarks

Windows Help ignores this macro if it is executed in a secondary window.

Print

The **Print** macro sends the currently displayed topic to the printer. It should be used only to print topics in windows other than the main Help window (for example, topics in a secondary window).

Syntax

Print()

Parameters

This macro does not take any parameters.

PrinterSetup

The **PrinterSetup** macro displays the Printer Setup dialog box from the File menu.

Syntax

PrinterSetup()

Parameters

This macro does not take any parameters.

Remarks

Use of the macro in secondary windows is not recommended.

RegisterRoutine

RegisterRoutine("*DLL-name***", "***function-name***", "***format-spec***")**

The **RegisterRoutine** macro registers a function within a dynamic-link library (DLL). Registered functions can be used in macro footnotes in topic files or in the [CONFIG] section of the project file, the same as standard Help macros.

Syntax

Parameters

DLL-name
Specifies the filename of the DLL. The filename must be enclosed in quotation marks (**"**). If Windows Help cannot find the library, it displays an error message.

function-name
Specifies the name of the function to execute in the designated DLL.

format-spec
Specifies a string indicating the formats of parameters passed to the function. The format string must be enclosed in quotation marks. Characters in the string represent C parameter types:

Character	Description
u	**unsigned short** (WORD)
U	**unsigned long** (DWORD)
i	**short int**
I	**int**
s	**near char *** (PSTR)
S	**far char *** (LPSTR)
v	**void**

If the function is used as a Help macro, Windows Help makes sure the macro parameters match the parameter types given in this macro.

Remarks

The **RegisterRoutine** macro can be abbreviated as **RR**.

Example

The following call registers a routine named PlayAudio in a DLL, MMLIB.DLL. PlayAudio takes arguments of the **far char ***, **int**, and **unsigned long** types:

```
RegisterRoutine("MMLIB", "PlayAudio", "SIU")
```

See Also

[CONFIG]

RemoveAccelerator

The **RemoveAccelerator** macro removes the assignment of a Help macro to an accelerator key (or key combination). These assignments are made by using the **AddAccelerator** macro.

Syntax

RemoveAccelerator(*key***,** *shift-state***)**

Parameters

key

Specifies the Windows virtual-key value. For a list of Virtual-Key Codes, see the *Microsoft Windows Programmer's Reference*, *Volume 3*.

shift-state

Specifies the combination of ALT, SHIFT, and CTRL keys that were used with the accelerator. This parameter may be one of the following values:

Value	Meaning
0	None
1	SHIFT
2	CTRL
3	SHIFT+CTRL
4	ALT
5	ALT+SHIFT
6	ALT+CTRL
7	SHIFT+ALT+CTRL

Remarks

The **RemoveAccelerator** macro can be abbreviated as **RA**. No error occurs when this macro is used with an accelerator for which a macro was not defined.

Example

The following macro disassociates a macro from the ALT+SHIFT+CONTROL+F4 key combination:

```
RemoveAccelerator(0x73, 7)
```

See Also

AddAccelerator

SaveMark

The **SaveMark** macro saves the location of the currently displayed topic and file and associates a text marker with that location. The **GotoMark** macro can then be used to jump to this location.

Syntax

SaveMark("*marker-text***")**

Parameters

marker-text
Specifies the text marker to be used to identify the topic location. This text must be enclosed in quotation marks (**"**), and it must be unique. If the same text is used for more than one marker, the most recently entered marker is used.

Remarks

A text marker can be used with the **GotoMark**, **DeleteMark**, **IfThen**, and **IfThenElse** macros.

If the user exits Windows Help, all text markers are deleted.

Example

The following macro saves the marker "Managing Memory" in the current topic:

```
SaveMark("Managing Memory")
```

See Also

DeleteMark, **GotoMark**, **IfThen**, **IfThenElse**

Search

The **Search** macro displays the dialog for the Search button, which allows users to search for topics using keywords defined by the K footnote character.

Syntax

Search()

Parameters

This macro does not take any parameters.

Remarks

Windows Help ignores this macro if it is executed in a secondary window.

SetContents

The **SetContents** macro designates a specific topic as the Contents topic in the specified Help file.

Syntax

SetContents("*filename***",** *context-number***)**

Parameters

filename
Specifies the name of the Help file that contains the Contents topic. The filename must be enclosed in quotation marks (**"**). If Windows Help cannot find this file, it displays an error message and does not perform the jump.

context number
Specifies the context number of the topic in the specified file. The context number must be defined in the [MAP] section of the project file. If the context number is not valid, Windows Help displays an error message.

Example

The following example sets the topic mapped to the context number 801 in the PROGMAN.HLP file as the Contents topic. After executing this macro, clicking the Contents button will cause a jump to the topic specified by the *context-number* parameter:

```
SetContents("PROGMAN.HLP", 801)
```

See Also

[MAP]

SetHelpOnFile

Syntax

SetHelpOnFile("*filename***")**

Parameters

filename

Specifies the name of the replacement How to Use Help file. The filename must be enclosed in quotation marks (**"**). If Windows Help cannot find this file, it displays an error message.

Remarks

If this macro appears in a topic in the Help file, the replacement file is set after execution of the macro. If this macro appears in the [CONFIG] section of the project file, the replacement file is set when the Help file is opened.

Example

The following macro sets the Using Help file to MYHELP.HLP:

```
SetHelpOnFile("myhelp.hlp")
```

See Also

[CONFIG]

UncheckItem

The **UncheckItem** macro removes the check mark from a menu item.

Syntax

UncheckItem("*item-id***")**

Parameters

item-id
Identifies the menu item to uncheck. The item identifier must be enclosed in quotation marks.

Remarks

The **UncheckItem** macro can be abbreviated **UI**.

See Also

CheckItem

Help Statement Reference

This section lists the RTF keywords recognized by the Help compiler and the special Help statements in alphabetic order. Help-specific commands that are not RTF keywords are marked "Help only." The following table summarizes each RTF keyword and Help statement:

RTF Keyword	Description
\'	Inserts character by value
\ansi	Specifies ANSI character set
\b	Starts bold text
\bin	Specifies picture (bitmap or metafile) data
bmc	Displays picture in text (Help only)
bmcwd	Displays picture in current line of text (Help only)
bml	Displays picture at left margin (Help only)
bmlwd	Displays picture at left margin (Help only)
bmr	Displays picture at right margin (Help only)
bmrwd	Displays picture at right margin (Help only)
\box	Draws box
\brdrb	Draws bottom border
\brdrbar	Draws vertical bar
\brdrdb	Sets double-lined borders
\brdrdot	Sets dotted border
\brdrl	Draws left border
\brdrr	Draws right border
\brdrs	Sets standard borders
\brdrt	Draws top border
\brdrth	Sets thick borders
\cell	Marks end of table cell
\cellx	Sets the position of cell's right edge
\cf	Sets foreground color
\colortbl	Creates color table
\deff	Sets default font
emc	Allows DLL to paint window in text (Help only)
eml	Allows DLL to paint window at left margin (Help only)
emr	Allows DLL to paint window at right margin (Help only)
\fi	Sets first-line indent
\fldrslt	Result of field
\f	Sets font
\fonttbl	Creates font table
\footnote	Defines topic-specific information
\fs	Sets font size
\i	Starts italic text
\intbl	Marks paragraph as in table
\keep	Makes text non-wrapping
\keepn	Creates nonscrolling region

\li	Sets left indent
\line	Breaks current line
\mac	Sets Apple® Macintosh® character set
\page	Ends current topic
\par	Marks end of paragraph
\pard	Restores default paragraph properties
\pc	Sets PC character set
\pichgoal	Specifies desired picture height
\pich	Specifies picture height
\picscalex	Specifies horizontal scaling value
\picscaley	Specifies vertical scaling value
\pict	Creates picture
\picwgoal	Specifies desired picture width
\picw	Specifies picture width
\plain	Restores default character properties
\qc	Centers text
\ql	Aligns text left
\qr	Aligns text right
\ri	Sets right indent
\row	Marks end of table row
\rtf	Specifies RTF version
\sa	Sets spacing after paragraph
\sb	Sets space before
\scaps	Starts small capitals
\sect	Marks end of section and paragraph
\sl	Sets spacing between lines
\strike	Creates hotspot
\tab	Inserts tab character
\tqc	Tabs and centers text
\tqr	Tabs and aligns text right
\trgaph	Sets space between text columns in table
\trleft	Sets left margin for first cell
\trowd	Sets table defaults
\trqc	Sets relative column widths
\trql	Left-aligns table row
\tx	Sets tab stop
\ul	Creates link to a pop-up topic
\uldb	Creates hot spot
\v	Creates link to a topic
\wbitmap	Specifies Windows bitmap and its type
\wbmbitspixel	Specifies number of bits per pixel
\wbmplanes	Specifies number of planes
\wbmwidthbytes	Specifies bitmap width in bytes
\windows	Sets Windows character set

\wmetafile	Specifies Windows metafile

\'

The **\'** statement converts the specified hexadecimal number into a character value and inserts the value into the Help file. The appearance of the character when displayed depends on the character set specified for the Help file.

Syntax

\'*hh*

Parameter

hh
Specifies a two-digit hexadecimal value.

Remarks

Since the Microsoft Help Compiler does not accept character values greater than 127, the **\'** statement is the only way to insert such character values into the Help file.

Example

The following example inserts a trademark in a Help file that uses the **\ansi** statement to set the character set:

```
ABC\'99 is a trademark of the ABC Product Corporation.
```

See Also

\ansi, **\mac**, **\pc**

\ansi

The **\ansi** statement sets the American National Standards Institute (ANSI) character set. The Windows character set is essentially equivalent to the ANSI character set.

See Also

\mac, **\pc**

\b

The **\b** statement starts bold text. The statement applies to all subsequent text up to the next **\plain** or **\b0** statement.

Remarks

No **\plain** or **\b0** statement is required if the **\b** statement and subsequent text are enclosed in braces. Braces limit the scope of a character property statement to just the enclosed text.

The **\b0** statement was first supported in the Microsoft Help Compiler version 3.1

Example

The following example sets "Note" to bold:

```
{\b Note}  Setting the Auto option frees novice users from determining their
system configurations.
```

See Also

\i, **\plain**, **\scaps**

\bin

The **\bin** statement indicates the start of binary picture data. The Help compiler interprets subsequent bytes in the file as binary data. This statement is used in conjunction with the **\pict** statement.

Syntax

\bin*n*

Parameter

n
Specifies the number of bytes of binary data following the statement.

Remarks

A single space character must separate the **\bin** statement from subsequent bytes. The Microsoft Help Compiler assumes that all subsequent bytes, including linefeed and carriage return characters, are binary data. These bytes can have any value in the range 0 through 255. For this reason, the **\bin** statement is typically used in program-generated files only.

If the **\bin** statement is not given with a **\pict** statement, the default picture data format is hexadecimal.

See Also

\pict

bmc

The **bmc** statement displays a specified bitmap or metafile in the current line of text. The statement positions the bitmap or metafile as if it were the next character in the line, aligning it on the base line and applying the current paragraph properties.

Syntax

\{bmc *filename***\}**

Parameter

filename
Specifies the name of a file containing a Windows bitmap, a placeable Windows metafile, a multiresolution bitmap, or a segmented-graphics bitmap.

Remarks

Since the **bmc** statement is not a standard RTF statement, the Microsoft Help Compiler relies on the opening and closing braces, including the backslashes (\), to distinguish the statement from regular text.

If a file containing a metafile is specified, the file must contain a placeable Windows metafile; the Microsoft Help Compiler will not accept standard Windows metafiles. Furthermore, Windows Help sets the MM_ANISOTROPIC mode prior to displaying the metafile, so the placeable Windows metafile must either set the window origin and extents or set some other mapping mode.

Example

The following example inserts a bitmap representing a keyboard key in a paragraph:

```
\par
Press the \{bmc escape.bmp\} key to return to the main window.
\par
```

See Also

bmr, **bml**, **\wbitmap**

bmcwd

The **bmcwd** statement displays a specified bitmap or metafile in the current line of text. The statement positions the bitmap or metafile as if it were the next character in the line, aligning it on the base line and applying the current paragraph properties.

The **bmcwd** statement is intended to be used for small bitmaps or metafiles that appear only once in a Help file.

Syntax

\{bmcwd *filename***\}**

Parameter

filename
Specifies the name of a file containing a Windows bitmap, a placeable Windows metafile, a multiresolution bitmap, or a segmented-graphics bitmap.

Remarks

Since the **bmcwd** statement is not a standard RTF statement, the Help compiler relies on the opening and closing braces, including the backslashes (\), to distinguish the statement from regular text or RTF codes.

If a file containing a metafile is specified, the file must contain a placeable Windows metafile; the Help compiler will not accept standard Windows metafiles. Furthermore, Windows Help sets the MM_ANISOTROPIC mode prior to displaying the metafile, so the placeable Windows metafile must either set the window origin and extents or set some other mapping mode.

See Also

bmr, **bml**, **bmc**, **\wbitmap**

bml

The **bml** statement displays a specified bitmap or metafile at the left margin of the Help window. The first line of subsequent text aligns with the upper-right corner of the image and subsequent lines wrap along the right edge of the image.

Syntax

\{bml *filename***\}**

Parameter

filename
Specifies the name of a file containing a Windows bitmap, a placeable Windows metafile, a multiresolution bitmap, or a segmented-graphics bitmap.

Remarks

Since the **bml** statement is not a standard RTF statement, the Microsoft Help Compiler relies on the opening and closing braces, including the backslashes (\), to distinguish the statement from regular text or RTF codes.

If a file containing a metafile is specified, the file must contain a placeable Windows metafile; the Microsoft Help Compiler will not accept standard Windows metafiles. Furthermore, Windows Help sets the MM_ANISOTROPIC mode prior to displaying the metafile, so the placeable Windows metafile must either set the window origin and extents or set some other mapping mode.

Example

The following example places a bitmap at the left margin. The subsequent paragraph wraps around the bitmap:

```
\par
\{bml roadmap.bmp\}
The map at the left shows the easiest route to the school.
Although many people use Highway 125, there are fewer stops
and less traffic if you use Ames Road.
```

See Also

bmc, **bmr**, **\wbitmap**

bmlwd

The **bmlwd** statement displays a specified bitmap or metafile at the left margin of the Help window. The first line of subsequent text aligns with the upper-right corner of the image and subsequent lines wrap along the right edge of the image.

The **bmlwd** statement is intended to be used for small bitmaps or metafiles that appear only once in a Help file.

Syntax

\{bmlwd *filename***\}**

Parameter

filename
Specifies the name of a file containing a Windows bitmap, a placeable Windows metafile, a multiresolution bitmap, or a segmented-graphics bitmap.

Remarks

Since the **bmlwd** statement is not a standard RTF statement, the Windows Help compiler relies on the opening and closing braces, including the backslashes (\), to distinguish the statement from regular text or RTF codes.

If a file containing a metafile is specified, the file must contain a placeable Windows metafile; the Help compiler will not accept standard Windows metafiles. Furthermore, Windows Help sets the MM_ANISOTROPIC mode prior to displaying the metafile, so the placeable Windows metafile must either set the window origin and extents or set some other mapping mode.

See Also

bmc, **bml**, **bmr**, **\wbitmap**

bmr

The **bmr** statement displays a specified bitmap or metafile at the right margin of the Help window. The first line of subsequent text aligns with the upper-left corner of the image and subsequent lines wrap along the left edge of the image.

Syntax

\{bmr *filename***\}**

Parameter

filename
Specifies the name of a file containing a Windows bitmap, a placeable Windows metafile, a multiresolution bitmap, or a segmented-graphics bitmap.

Remarks

Since the **bmr** statement is not a standard RTF statement, the Microsoft Help Compiler relies on the opening and closing braces, including the backslashes (\), to distinguish the statement from regular text or RTF codes.

If a file containing a metafile is specified, the file must contain a placeable Windows metafile; the Help compiler will not accept standard Windows metafiles. Furthermore, Windows Help sets the MM_ANISOTROPIC mode prior to displaying the metafile, so the placeable Windows metafile must either set the window origin and extents or set some other mapping mode.

Example

The following example places a bitmap at the right margin. The subsequent paragraph wraps around the bitmap:

```
\par
\{bmr roadmap.bmp\}
The map at the right shows the easiest route to the school.
Although many people use Highway 125, there are fewer stops
and less traffic if you use Ames Road.
```

See Also

bmc, **bml**, **\wbitmap**

bmrwd

The **bmrwd** statement displays a specified bitmap or metafile at the right margin of the Help window. The first line of subsequent text aligns with the upper-left corner of the image and subsequent lines wrap along the left edge of the image.

The **bmrwd** statement is intended to be used for small bitmaps or metafiles that appear only once in a Help file.

Syntax

\{bmrwd *filename***\}**

Parameter

filename
Specifies the name of a file containing a Windows bitmap, a placeable Windows metafile, a multiresolution bitmap, or a segmented-graphics bitmap.

Remarks

Since the **bmrwd** statement is not a standard RTF statement, the Windows Help compiler relies on the opening and closing braces, including the backslashes (\), to distinguish the statement from regular text or RTF codes.

If a file containing a metafile is specified, the file must contain a placeable Windows metafile; the Help compiler will not accept standard Windows metafiles. Furthermore, Windows Help sets the MM_ANISOTROPIC mode prior to displaying the metafile, so the placeable Windows metafile must either set the window origin and extents or set some other mapping mode.

See Also

bmc, **bml**, **bmr**, **\wbitmap**

\box

The **\box** statement draws a box around the current paragraph or picture. The statement applies to all subsequent paragraphs or pictures up to the next **\pard** statement.

Remarks

For paragraphs, Windows Help uses the height of the paragraph, excluding space before or after the paragraph, as the height of the box. For pictures (as defined by **\pict** statements), Windows Help uses the specified height of the picture as the height of the box. For both paragraphs and pictures, the width of the box is equal to the space between the left and right indents.

Windows Help draws the box using the current border style.

Example

The following example draws a box around the paragraph:

```
\par \box
{\b Note}  Setting the Auto option frees novice users from
determining their system configurations.
\par \pard
```

See Also

\brdrb, **\brdrl**, **\brdrr**, **\brdrt**, **\pard**, **\pict**

\brdrb

The **\brdrb** statement draws a border below the current paragraph or picture. The statement applies to all subsequent paragraphs or pictures up to the next **\pard** statement.

Remarks

Windows Help draws the border using the current border style.

See Also

\box, **\brdrbar**, **\brdrl**, **\brdrr**, **\brdrt**, **\pard**

\brdrbar

The **\brdrbar** statement draws a vertical bar to the left of the current paragraph or picture. The statement applies to all subsequent paragraphs or pictures up to the next **\pard** statement.

Remarks

Windows Help draws the border using the current border style.

In a print-based document, the **\brdrbar** statement draws the bar on the right side of paragraphs on odd-numbered pages, but on the left side of paragraphs on even-numbered pages.

See Also

\box, **\brdrl**, **\brdrb**, **\brdrr**, **\brdrt**, **\pard**

\brdrdb

The **\brdrdb** statement selects a double line for drawing borders. The selection applies to all subsequent paragraphs or pictures up to the next **\pard** statement.

See Also

\brdrdot, **\brdrs**, **\brdrth**, **\pard**

\brdrdot

The Help compiler ignores this statement.

See Also

\brdrs, **\brdrth**, **\brdrdb**, **\pard**

\brdrl

The **\brdrl** statement draws a border to the left of the current paragraph or picture. The statement applies to all subsequent paragraphs or pictures up to the next **\pard** statement.

Remarks

Windows Help draws the border using the current border style.

See Also

\box, **\brdrb**, **\brdrbar**, **\brdrr**, **\brdrt**, **\pard**

\brdrr

The **\brdrr** statement draws a border to the right of the current paragraph or picture. The statement applies to all subsequent paragraphs or pictures up to the next **\pard** statement.

Remarks

Windows Help draws the border using the current border style.

See Also

\box, **\brdrb**, **\brdrbar**, **\brdrl**, **\brdrt**, **\pard**

\brdrs

The **\brdrs** statement selects a standard-width line for drawing borders. The selection applies to all subsequent paragraphs or pictures up to the next **\pard** statement.

Remarks

The Help compiler ignores this statement.

See Also

\brdrdb, **\brdrdot**, **\brdrth**, **\pard**

\brdrt

The **\brdrt** statement draws a border above the current paragraph or picture. The statement applies to all subsequent paragraphs or pictures up to the next **\pard** statement.

Remarks

Windows Help draws the border using the current border style.

See Also

\box, **\brdrb**, **\brdrbar**, **\brdrl**, **\brdrr**, **\pard**

\brdrth

The **\brdrth** statement selects a thick line for drawing borders. The selection applies to all subsequent paragraphs or pictures up to the next **\pard** statement.

See Also

\brdrdb, **\brdrdot**, **\brdrs**, **\pard**

\cb

The Help compiler ignores this statement.

Syntax

\cb*n*

Parameter

n
Specifies the color number to set as the background color. The number must be an integer number in the range 1 to the maximum number of colors specified in the color table for the Help file. If an invalid color number is specified, Windows Help uses the default background color.

Remarks

The default background color is the window background color set by Control Panel.

See Also

\cf, **\colortbl**

\cell

The **\cell** statement marks the end of a cell in a table. A cell consists of all paragraphs from a preceding **\intbl** or **\cell** statement to the ending **\cell** statement. Windows Help formats and displays these paragraphs using the left and right margins of the cell and any current paragraph properties.

Remarks

This statement was first supported in the Microsoft Help Compiler version 3.1.

Example

The following example creates a two-column table. The second column contains three separate paragraphs, each having different paragraph properties:

```
\cellx2880\cellx5760
\intbl
Alignment\cell
\ql
Left-aligned
\par
\qc
Centered
\par
\qr
Right-aligned\cell
\row \pard
```

See Also

\cellx, **\intbl**, **\row**, **\trgaph**, **\trleft**, **\trowd**

\cellx

The **\cellx** statement sets the absolute position of the right edge of a table cell. One **\cellx** statement must be given for each cell in the table. The first **\cellx** statement applies to the left-most cell, the last to the right-most cell. For each **\cellx** statement, the specified position applies to the corresponding cell in each subsequent row of the table up to the next **\trowd** statement.

Syntax

\cellx*n*

Parameter

n

Specifies the position of the cell's right edge, in twips. The position is relative to the left edge of the Help window. It is not affected by the current indents.

Remarks

A table consists of a grid of cells in columns and rows. Each cell has an explicitly defined right edge; the position of a cell's left edge is the same as the position of the right edge of the adjacent cell. For the left-most cell in a row, the left edge position is equal to the Help window's left margin position. Each cell has a left and right margin between which Windows Help aligns and wraps text. By default, the margin positions are equal to the left and right edges. The **\trgaph** and **\trleft** statements can be used to set different margins for all cells in a row.

This statement was first supported in the Microsoft Help Compiler version 3.1.

Example

The following example creates a three-column table having two rows. The positions of the right edges of the three cells are 2, 4, and 6 inches, respectively:

```
\cellx2880\cellx5760\cellx8640
\intbl
Row 1 Cell 1\cell
Row 1 Cell 2\cell
Row 1 Cell 3\cell
\row
\intbl
Row 2 Cell 1\cell
Row 2 Cell 2\cell
Row 2 Cell 3\cell
\row \pard
```

See Also

\cell, **\intbl**, **\row**, **\trgaph**, **\trleft**, **\trowd**

\cf

The **\cf** statement sets the foreground color. The new color applies to all subsequent text up to the next **\plain** or **\cf** statement.

Syntax

\cf*n*

Parameter

n

Specifies the color number to set as foreground. The number must be an integer number in the range 1 to the maximum number of colors specified in the color table for the Help file. If an invalid color number is specified, Windows Help uses the default foreground color.

Remarks

No **\plain** or **\cf** statement is required if the **\cf** statement and subsequent text are enclosed in braces. Braces limit the scope of a character property statement to the enclosed text only.

If the **\cf** statement is not given, the default foreground color is the text color set by Control Panel.

Example

The following example displays green text:

```
{\colortbl;\red0\green255\blue0;}
{\cf1 This text is green.}
```

See Also

\colortbl, **\plain**

\chftn

The Microsoft Help Compiler ignores this statement.

Syntax

\chftn*n*

Parameter

n
Specifies the footnote reference character.

See Also

\footnote

\clmgf

The Microsoft Help Compiler ignores this statement.

Remarks

All cells between the **\clmgf** statement and a subsequent **\clmrg** statement are combined into a single cell. The left edge of the new cell is the same as that of the leftmost cell to be merged; the right edge is the same as that of the rightmost cell.

See Also

\clmrg

\clmrg

The Microsoft Help Compiler ignores this statement.

Remarks

All cells between the **\clmgf** statement and a subsequent **\clmrg** statement are combined into a single cell. The left edge of the new cell is the same as that of the leftmost cell to be merged; the right edge is the same as that of the rightmost cell.

See Also

\clmgf

\colortbl

The **\colortbl** statement creates a color table for the Help file. The color table consists of one or more color definitions. Each color definition consists of one **\red**, **\green**, and **\blue** statement specifying the amount of primary color to use to generate the final color. Each color definition must end with a semicolon (**;**).

Syntax

{\colortbl
\red*redval***\green***greenval***\blue***blueval***;**
. . .
}

Parameters

redval
Specifies the intensity of red in the color. It must be an integer in the range 0 through 255.

greenval
Specifies the intensity of green in the color. It must be an integer in the range 0 through 255.

blueval
Specifies the intensity of blue in the color. It must be an integer in the range 0 through 255.

Remarks

Color definitions are implicitly numbered starting at zero. A color definition's implicit number can be used in the **\cf** statement to set the foreground color.

The default colors are the window-text and window-background colors set by Control Panel. To override the default colors, both a **\colortbl** statement and a **\cf** statement must be given.

Example

The following example creates a color table containing two color definitions. The first color definition is empty (only the semicolon is given), so color number 0 always represents the default color. The second definition specifies green; color number 1 can be used to display green text:

```
{\colortbl;\red0\green255\blue0;}
```

See Also

\cf

\deff

The **\deff** statement sets the default font number. Windows Help uses the number to set the default font whenever a **\plain** statement is given or an invalid font number is given in a **\f** statement.

Syntax

\deff*n*

Parameter

n

Specifies the number of the font to be used as the default font. This parameter must be a valid font number as specified by the **\fonttbl** statement for the Help file.

Remarks

If the **\deff** statement is not given, the default font number is zero.

See Also

\f, **\fonttbl**, **\plain**

emc

The **emc** statement allows an external dynamic-link library to paint a window that is embedded in a Help topic. This statement displays the window in the current line of text. The statement positions the window as if it were the next character in the line, aligning it on the base line and applying the current paragraph properties.

Syntax

\{emc *module*, *class*, *data* [, *dx*, *dy*]**\}**

Parameters

module
Specifies the name of the dynamic-link library that paints the embedded window.

class
Specifies the name of the registered window class for the embedded window.

data
Specifies a string that is passed to the embedded window in its WM_CREATE message.

dx
Specifies the suggested width of the embedded window. This parameter is optional.

dy
Specifies the suggested height of the embedded window. This parameter is optional.

Remarks

Since the **emc** statement is not a standard RTF statement, the Microsoft Help Compiler relies on the opening and closing braces, including the backslashes (\), to distinguish the statement from regular text or RTF codes.

See Also

bmr, **bml**, **bmc**, **eml**, **emr**, **\wbitmap**, WM_CREATE

eml

The **eml** statement allows an external dynamic-link library to paint a window that is embedded at the left margin in a Help topic. The first line of subsequent text aligns with the upper-right corner of the window and subsequent lines wrap along the right edge of the window.

Syntax

\{eml *module*, *class*, *data* [, *dx*, *dy*]**\}**

Parameters

module
Specifies the name of the dynamic-link library that paints the embedded window.

class
Specifies the name of the registered window class for the embedded window.

data
Specifies a string that is passed to the embedded window in its WM_CREATE message.

dx
Specifies the suggested width of the embedded window. This parameter is optional.

dy
Specifies the suggested height of the embedded window. This parameter is optional.

Remarks

Since the **eml** statement is not a standard RTF statement, the Microsoft Help Compiler relies on the opening and closing braces, including the backslashes (\), to distinguish the statement from regular text or RTF codes.

See Also

bmr, **bml**, **bmc**, **emc**, **emr**, **\wbitmap**, WM_CREATE

emr

The **emr** statement allows an external dynamic-link library to paint a window that is embedded at the right margin in a Help topic. The first line of subsequent text aligns with the upper-left corner of the window and subsequent lines wrap along the left edge of the window.

Syntax

\{emr *module*, *class*, *data* [, *dx*, *dy*]**\}**

Parameters

module
Specifies the name of the dynamic-link library that paints the embedded window.

class
Specifies the name of the registered window class for the embedded window.

data
Specifies a string that is passed to the embedded window in its WM_CREATE message.

dx
Specifies the suggested width of the embedded window. This parameter is optional.

dy
Specifies the suggested height of the embedded window. This parameter is optional.

Remarks

Since the **emr** statement is not a standard RTF statement, the Microsoft Help Compiler relies on the opening and closing braces, including the backslashes (\), to distinguish the statement from regular text or RTF codes.

See Also

bmr, **bml**, **bmc**, **emc**, **eml**, **\wbitmap**, WM_CREATE

\f

The **\f** statement sets the font. The new font applies to all subsequent text up to the next **\plain** or **\f** statement.

Syntax

\f*n*

Parameter

n
Specifies the font number. This parameter must be one of the integer font numbers defined in the font table for the Help file.

Remarks

The **\f** statement does not set the point size of the font; use the **\fs** statement instead.

No **\plain** or **\f** statement is required if the **\f** statement and subsequent text are enclosed in braces. Braces limit the scope of a character property statement to just the enclosed text.

If the **\f** statement is not given, the default font is defined by the **\deff** statement (or is zero if no **\deff** statement is given).

Example

The following example uses the Arial font to display text:

```
{\fonttbl {\f0\fswiss Arial;}}
{\f0
This text illustrates the Arial font.}
\par
```

See Also

\deff, **\fonttbl**, **\fs**, **\plain**

\fi

The **\fi** statement sets the first-line indent for the paragraph. The new indent applies to the first line of each subsequent paragraph up to the next **\pard** statement or **\fi** statement. The first-line indent is always relative to the current left indent.

Syntax

\fi*n*

Parameter

n
Specifies the indent, in twips. This parameter can be either a positive or negative number.

Remarks

If the **\fi** statement is not given, the first-line indent is zero by default.

Example

The following example uses the first-line indent and a tab stop to make a numbered list:

```
\tx360\li360\fi-360
1
\tab
Insert the disk in drive A.
\par
2
\tab
Type a:setup and press the ENTER key.
\par
3
\tab
Follow the instructions on the screen.
\par \pard
```

See Also

\li, **\pard**

\field

The Microsoft Help Compiler ignores this statement and all related field statements except the **\fldrslt** statement.

See Also

\fldrslt

\fldrslt

The **\fldrslt** statement specifies the most recently calculated result of a field. The Microsoft Help Compiler interprets the result as text and formats it using the current character and paragraph properties.

Remarks

The Help compiler ignores all field statements except the **\fldrslt** statement. Any text associated with other field statements is ignored.

\fonttbl

The **\fonttbl** statement creates a font table for the Help file. The font table consists of one or more font definitions. Each definition consists of a font number, a font family, and a font name.

Syntax

{\fonttbl
    {\f*n**family font-name***;}**
        . . .
}

Parameter

n
Specifies the font number. This parameter must be an integer. This number can be used in subsequent **\f** statements to set the current font to the specified font. In the font table, font numbers should start at zero and increase by one for each new font definition.

family
Specifies the font family. This parameter must be one of the following:

Value	Meaning
fnil	Unknown or default fonts (default)
froman	Roman, proportionally spaced serif fonts (for example, Times New Roman and Palatino)
fswiss	Swiss, proportionally spaced sans serif fonts (for example, Arial)
fmodern	Fixed-pitch serif and sans serif fonts (for example, Courier and Lucida Sans Typewriter)
fscript	Script fonts (for example, Cursive and Lucida Handwriting)
fdecor	Decorative fonts (for example, Old English and Bodoni Open)
ftech	Technical, symbol, and mathematical fonts (for example, Symbol)

font-name
Specifies the name of the font. This parameter should specify an available Windows font.

Remarks

If a font with the specified name is not available, Windows Help chooses a font from the specified family. If no font from the given family exists, Windows Help chooses a font having the same character set as specified for the Help file.

The **\deff** statement sets the default font number for the Help file. The default font is set whenever the **\pard** statement is given.

See Also

\deff, **\f**, **\fs**, **\pard**

\footnote

The **\footnote** statement defines topic-specific information, such as the topic's build tags, context string, title, browse number, keywords, and execution macros. Every topic must have a context string, at least, to give the user access to the topic through links.

Syntax

{*n***}{\footnote {***n***}** *text***}**

Parameter

n

Specifies the footnote character. It can be one of the following:

Value	Meaning
*	Specifies a build tag. The Microsoft Help Compiler uses build tags to determine whether it should include the topic in the Help file. The *text* parameter can be any combination of characters but must not contain spaces. Uppercase and lowercase characters are treated as equivalent characters (case-insensitive). If a topic has build-tag statements, they must be the first statements in the topic. The Microsoft Help Compiler checks a topic for build tags if the project file specifies a build expression using the **BUILD** option.
#	Specifies a context string. The *text* parameter can be any combination of letters and digits but must not contain spaces. Uppercase and lowercase characters are treated as equivalent characters (case-insensitive). The context string can be used with the **\v** statement in other topics to create links to this topic.
$	Specifies a topic title. Windows Help uses the topic title to identify the topic in the Search and History dialog boxes. The *text* parameter can be any combination of characters including spaces.
+	Specifies the browse-sequence identifier. Windows Help adds topics having an identifier to the browse sequence and allows users to view the topics by using the browse buttons. The *text* parameter can be a combination of letters and digits. Windows Help determines the order of topics in the browse sequence by sorting the identifier alphabetically. If two topics have the same identifier, Windows Help assumes that the topic that was compiled first is to be displayed first. Windows Help uses the browse sequence identifier only if the browse buttons have been enabled by using the **BrowseButtons** macro.
K	Specifies a keyword. Windows Help displays all keywords in the Help file in the Search dialog box and allows a user to choose a topic to view by choosing a keyword. The *text* parameter can be any combination of characters including spaces. If the first character is the letter K, it must be preceded with an extra space or a semicolon. More than one keyword can be given by separating the keywords with semicolons (;). A topic cannot contain keywords unless it also has a topic title.

!
: Specifies a Help macro. Windows Help executes the macro when the topic is displayed. The *text* parameter can be any Help macro.

If *n* is any letter (other than K), the footnote specifies an alternative keyword. Windows applications can search for topics having alternative keywords by using the HELP_MULTIKEY command with the **WinHelp** function.

text
: Specifies the build tag, context string, topic title, browse-sequence number, keyword, or macro associated with the footnote. This parameter depends on the footnote type as specified by the *n* parameter.

Remarks

Repetition of the footnote character, *n*, in the syntax is deliberate.

A topic can have more than one build-tag, context-string, keyword, and Help-macro statement, but must not have more than one topic-title or browse-sequence-number statement.

In print-based documents, the **\footnote** statement creates a footnote. The footnote is anchored to the character immediately preceding the **\footnote** statement.

The characters in a context string must be alphanumeric and can include underscore characters (**_**) and periods (**.**).

The browse sequence string consists of a major sequence string and a minor sequence string, delimited by a colon:

{+}{\footnote {+} *major*:*minor***}**

This syntax specifies disjoint sets of ordered browse sequences. The major sequence string determines which browse sequence a topic belongs to, while the minor sequence string determines its position. Minor sequence strings are sorted alphabetically, not numerically; to use numbers, they should be preceded with zeros so that they are all the same length. All topics with browse sequence strings that omit the major sequence string are placed on the same browse sequence.

A topic cannot have more than one build tag footnote. If a topic has a build tag footnote, it must be the first thing in that topic. The title, browse sequence, and macro must be in the first paragraph. Context strings and keywords may appear anywhere; if placed in the middle of a topic, jumps to that context string or keyword will bring you to the middle of that topic.

Example

The following example defines a topic titled Short Topic. The context string topic1 can be used to create links to this topic. The keywords example topic and short topic appear in the Search dialog box and can be used to choose the topic for viewing:

```
${\footnote Short Topic}
#{\footnote topic1}
K{\footnote example topic;short topic}
This topic has a title, context string, and two keywords.
\par
\page
```

See Also

\v

\fs

The **\fs** statement sets the size of the font. The new font size applies to all subsequent text up to the next **\plain** or **\fs** statement.

Syntax

\fs*n*

Parameter

n
Specifies the size of the font, in half points.

Remarks

The **\fs** statement does not set the font face; use the **\f** statement instead.

No **\plain** or **\fs** statement is required if the **\fs** statement and subsequent text are enclosed in braces. Braces limit the scope of a character property statement to just the enclosed text.

If the **\fs** statement is not given, the default font size is 24.

Example

The following example sets the size of the font to 10 points:

```
{\fs20 This line is in 10 point type.}
\par
```

See Also

\plain, \f

\i

The **\i** statement starts italic text. The statement applies to all subsequent text up to the next **\plain** or **\i0** statement.

Remarks

No **\plain** or **\i0** statement is required if the **\i** statement and subsequent text are enclosed in braces. Braces limit the scope of a character property statement to just the enclosed text.

Example

The following example sets "not" to italic:

```
You must {\i not} save the file without first setting the
Auto option.
```

See Also

\b, **\plain**, **\scaps**

\intbl

The **\intbl** statement marks subsequent paragraphs as part of a table. The statement applies to all subsequent paragraphs up to the next **\row** statement.

Remarks

This statement was first supported in Microsoft Help Compiler version 3.1.

Example

The following example creates a three-column table having two rows:

```
\cellx1440\cellx2880\cellx4320
\intbl
Row 1 Column 1\cell
Row 1 Column 2\cell
Row 1 Column 3\cell \row
\intbl
Row 2 Column 1\cell
Row 2 Column 2\cell
Row 2 Column 3\cell \row \pard
```

See Also

\cell, **\cellx**, **\row**, **\trgaph**, **\trleft**, **\trowd**

\keep

The **\keep** statement prevents Windows Help from wrapping text to fit the Help window. The statement applies to all subsequent paragraphs up to the next **\pard** statement.

Remarks

If the text in a paragraph exceeds the width of the Help window, Help displays a horizontal scroll bar.

In print-based documents, the **\keep** statement keeps paragraphs intact.

See Also

\keepn, **\line**

\keepn

The **\keepn** statement creates a nonscrolling region at the top of the Help window for the given topic. The **\keepn** statement applies to all subsequent paragraphs up to the next **\pard** statement. All paragraphs with this paragraph property are placed in the nonscrolling region.

Remarks

If a **\keepn** statement is used in a topic, it must be applied to the first paragraph in the topic (and subsequent paragraphs as needed). The Help compiler displays an error message and does not create a nonscrolling region if paragraphs are given before the **\keepn** statement. Only one nonscrolling region per topic is allowed.

Windows Help formats, aligns, and wraps text in the nonscrolling region just as it does in the rest of the topic. It separates the nonscrolling region from the rest of the Help window with a horizontal bar. Windows Help sets the height of the nonscrolling region so that all paragraphs in the region can be viewed if the Help window is large enough. If the window is smaller than the nonscrolling region, the user will be unable to view the rest of the topic. For this reason, the nonscrolling region is typically reserved for a single line of text specifying the name or title of the topic.

In print-based documents, the **\keepn** statement keeps the subsequent paragraph with the paragraph that follows it.

See Also

\keep, **\page**

\li

The **\li** statement sets the left indent for the paragraph. The indent applies to all subsequent paragraphs up to the next **\pard** or **\li** statement.

Syntax

\li*n*

Parameter

n
Specifies the indent, in twips. The value can be either positive or negative.

Remarks

If the **\li** statement is not given, the left indent is zero by default. Windows Help automatically provides a small left margin so that if no indent is specified the text does not start immediately at the left edge of the Help window.

Specifying a negative left indent moves the starting point for a line of text to the left of the default left margin. If the negative indent is large enough, the start of the text may be clipped by the left edge of the Help window.

Example

The following example uses the left indent and a tab stop to make a bulleted list. In this example, font number 0 is assumed to be the Symbol font:

```
Use the Auto command to:
\par
\tx360\li360\fi-360
{\f0\'B7}
\tab
Save files automatically
\par
{\f0\'B7}
\tab
Prevent overwriting existing files
\par
{\f0\'B7}
\tab
Create automatic backup files
\par \pard
```

See Also

\fi, **\pard**, **\ri**

\line

The **\line** statement breaks the current line without ending the paragraph. Subsequent text starts on the next line and is aligned and indented according to the current paragraph properties.

See Also

\par

\mac

The **\mac** statement sets the Apple Macintosh character set.

See Also

\ansi, **\pc**

\page

The **\page** statement marks the end of a topic.

Remarks

In a print-based document, the **\page** statement creates a page break.

Example

The following example shows a complete topic:

```
${\footnote Short Topic}
#{\footnote short_topic}
Most topics in a topic file consist of topic-title and
context-string statements followed by the topic text. Every
topic ends with a {\b \\page} statement.
\par
\page
```

See Also

\par

\par

The **\par** statement marks the end of a paragraph. The statement ends the current line of text and moves the current position to the left margin and down by the current line-spacing and space-after-paragraph values.

Remarks

The first line of text after a **\par**, \page, or \sect statement marks the start of a paragraph. When a paragraph starts, the current position is moved down by the current space-before-paragraph value. Subsequent text is formatted using the current text alignment, line spacing, and left, right, and first-line indents.

Example

The following example has three paragraphs:

```
\ql
This paragraph is left-aligned.
\par \pard
\qc
This paragraph is centered.
\par \pard
\qr
This paragraph is right-aligned.
\par
```

See Also

\line, \page, \pard, \sect

\pard

The **\pard** statement restores all paragraph properties to default values.

Remarks

If the **\pard** statement appears anywhere before the end of a paragraph (that is, before the **\par** statement), the default properties apply to the entire paragraph.

The default paragraph properties are as follows:

Property	Default
Alignment	Left-aligned
First-line indent	0
Left indent	0
Right indent	0
Space before	0
Space after	0
Line spacing	Tallest character
Tab stops	None
Borders	None
Border style	Single-width

See Also

\par

\pc

The **\pc** statement sets the OEM character set (also known as code page 437).

See Also

\ansi, **\mac**

\pca

The Help compiler ignores this statement.

See Also

\ansi, **\mac**, **\pc**

\pich

The **\pich** statement specifies the height of the picture. This statement must be used in conjunction with a **\pict** statement.

Syntax

\pich*n*

Parameter

n

Specifies the height of the picture, in twips or pixels, depending on the picture type. If the picture is a metafile, the width is in twips; otherwise, the width is, in pixels.

See Also

\pict, **\picw**

\pichgoal

The **\pichgoal** statement specifies the desired height of a picture. If necessary, Windows Help stretches or compresses the picture to match the requested height. This statement must be used in conjunction with a **\pict** statement.

Syntax

\pichgoal*n*

Parameter

n
Specifies the desired height, in twips.

Remarks

The **\pichgoal** statement is not supported for metafiles. Applications should use the **\pich** statement, instead.

See Also

\pich, **\pict**, **\picwgoal**

\picscalex

The **\picscalex** statement specifies the horizontal scaling value. This statement must be used in conjunction with a **\pict** statement.

Syntax

\picscalex*n*

Parameter

n

Specifies the scaling value as a percentage. If this value is greater than 100, the bitmap or metafile is enlarged.

Remarks

If the **\picscalex** statement is not given, the default scaling value is 100.

See Also

\pict, **\picscaley**

\picscaley

The **\picscaley** statement specifies the vertical scaling value. This statement must be used in conjunction with a **\pict** statement.

Syntax

\picscaley*n*

Parameter

n

Specifies the scaling value as a percentage. If this value is greater than 100, the bitmap or metafile is enlarged.

Remarks

If the **\picscaley** statement is not given, the default scaling value is 100.

See Also

\pict, **\picscalex**

\pict

The **\pict** statement creates a picture. A picture consists of hexadecimal or binary data representing a bitmap or metafile.

Syntax

\pict *picture-statements picture-data*

Parameter

picture-statements

Specifies one or more statements defining the type of picture, the dimensions of the picture, and the format of the picture data. It can be a combination of the following statements:

Statement	Description
\wbitmap	Specifies a Windows bitmap.
\wmetafile	Specifies a Windows metafile.
\picw*n*	Specifies the picture width.
\pich*n*	Specifies the picture height.
\picwgoal*n*	Specifies the desired picture width.
\pichgoal*n*	Specifies the desired picture height.
\picscalex*n*	Specifies the horizontal scaling value.
\picscaley*n*	Specifies the vertical scaling value.
\wbmbitspixel*n*	Specifies the number of bits per pixel.
\wbmplanes*n*	Specifies the number of planes.
\wbmwidthbytes*n*	Specifies the bitmap width, in bytes.
\bin*n*	Specifies binary picture data and its length in bytes.

picture-data

Specifies hexadecimal or binary data representing the picture. The picture data follows the last picture statement.

Remarks

If a data format is not specified, the default format is hexadecimal.

See Also

\bin, **\pich**, **\pichgoal**, **\picscalex**, **\picscaley**, **\picw**, **\picwgoal**, **\wbitmap**, **\wbmbitspixel**, **\wbmplanes**, **\wbmwidthbytes**, **\wmetafile**

\picw

The **\picw** statement specifies the width of the picture. This statement must be used in conjunction with a **\pict** statement.

Syntax

\picw*n*

Parameters

n

Specifies the width of the picture, in twips or pixels, depending on the picture type. If the picture is a metafile, the width is in twips; otherwise, the width is in pixels.

See Also

\pict, **\pich**

\picwgoal

The **\picwgoal** statement specifies the desired width of the picture, in twips. If necessary, Windows Help stretches or compresses the picture to match the requested height. This statement must be used in conjunction with a **\pict** statement.

Syntax

\picwgoal*n*

Parameter

n
Specifies the desired width, in twips.

Remarks

The **\picwgoal** statement is not supported for metafiles. Applications should use the **\picw** statement, instead.

See Also

\pict, **\picw**, **\pichgoal**

\plain

The **\plain** statement restores the character properties to default values.

Remarks

The default character properties are as follows:

Property	Default
Bold	Off
Italic	Off
Small caps	Off
Font	0
Font size	24

See Also

\b, **\i**, **\scaps**, **\f**, **\fs**

\qc

The **\qc** statement centers text between the current left and right indents. The statement applies to subsequent paragraphs up to the next **\pard** statement or text-alignment statement.

Remarks

If a **\ql**, **\qr**, or **\qc** statement is not given, the text is left-aligned by default.

See Also

\pard, **\ql**, **\qr**

\qj

The Microsoft Help Compiler ignores this statement.

Remarks

If a **\ql**, **\qr**, **\qc**, or **\qj** statement is not given, the text is left-aligned by default.

See Also

\qc, **\ql**, **\qr**, **\pard**

\ql

The **\ql** statement aligns text along the left indent. The statement applies to subsequent paragraphs up to the next **\pard** statement or text-alignment statement.

Remarks

If a **\ql**, **\qr**, or **\qc** statement is not given, the text is left-aligned by default.

See Also

\pard, **\qc**, **\qr**

\qr

The **\qr** statement aligns text along the right indent. The statement applies to subsequent paragraphs up to the next **\pard** statement or text-alignment statement.

Remarks

If a **\ql**, **\qr**, or **\qc** statement is not given, the text is left-aligned by default.

See Also

\pard, **\qc**, **\ql**

\ri

The **\ri** statement sets the right indent for the paragraph. The indent applies to all subsequent paragraphs up to the next **\pard** or **\ri** statement.

Syntax

\ri*n*

Parameter

n
Specifies the right indent, in twips. It can be a positive or negative value.

Remarks

If the **\ri** statement is not given, the right indent is zero by default. Windows Help automatically provides a small right margin so that when no right indent is specified, the text does not end abruptly at the right edge of the Help window.

Windows Help never displays less than one word for each line in a paragraph even if the right indent is greater than the width of the window.

Example

In the following example, the right and left indents are set to one inch and the subsequent text is centered between the indents:

```
\li1440\ri1440\qc
Microsoft Windows Help\line
Sample File\line
```

See Also

\li, **\pard**

\row

The **\row** statement marks the end of a table row. The statement ends the current row and begins a new row by moving down past the end of the longest cell in the row. The next **\cell** statement specifies the text of the leftmost cell in the next row.

Remarks

This statement was first supported in the Microsoft Help Compiler version 3.1.

Example

The following example creates a table having four rows and two columns:

```
\cellx2880\cellx5760
\intbl
Row 1, Column 1\cell
Row 1, Column 2\cell \row
\intbl
Row 2, Column 1\cell
Row 2, Column 2\cell \row
\intbl
Row 3, Column 1\cell
Row 3, Column 2\cell \row
\intbl
Row 4, Column 1\cell
Row 4, Column 2\cell \row
\par \pard
```

See Also

\cell, **\cellx**, **\intbl**

\rtf

The **\rtf** statement identifies the file as a rich-text format (RTF) file and specifies the version of the RTF standard used.

Syntax

\rtf*n*

Parameter

n
Specifies the version of the RTF standard used. For the Microsoft Help Compiler version 3.1, this parameter must be 1.

Remarks

The **\rtf** statement must follow the first open brace in the Help file. A statement specifying the character set for the file must also follow the **\rtf** statement.

See Also

\ansi

\sa

The **\sa** statement sets the amount of vertical spacing after a paragraph. The vertical space applies to all subsequent paragraphs up to the next **\pard** or **\sa** statement.

Syntax

\sa*n*

Parameter

n
Specifies the amount of vertical spacing, in twips.

Remarks

If the **\sa** statement is not given, the vertical spacing after a paragraph is zero by default.

See Also

\sb, **\pard**

\sb

The **\sb** statement sets the amount of vertical spacing before the paragraph. The vertical space applies to all subsequent paragraphs up to the next **\pard** statement or **\sb** statement.

Syntax

\sb*n*

Parameter

n
Specifies the amount of vertical spacing, in twips.

Remarks

If the **\sb** statement is not given, the vertical spacing before the paragraph is zero by default.

See Also

\sa, **\pard**

\scaps

The **\scaps** statement starts small-capital text. The statement converts all subsequent lowercase letters to uppercase before displaying the text. This statement applies to all subsequent text up to the next **\plain** or **\scaps0** statement.

Remarks

The **\scaps** statement does not affect uppercase letters.

No **\plain** or **\scaps0** statement is required if the **\scaps** statement and subsequent text are enclosed in braces. Braces limit the scope of a character property statement to just the enclosed text.

The **\scaps** statement does not reduce the point size of the text. To reduce point size, the **\fs** statement must be used.

Example

The following example displays the key name ENTER in small capitals:

```
Press the {\scaps enter} key to complete the action.
```

See Also

\fs, **\plain**

\sect

The **\sect** statement marks the end of a section and paragraph.

See Also

\par

\sl

The **\sl** statement sets the amount of vertical space between lines in a paragraph. The vertical space applies to all subsequent paragraphs up to the next **\pard** or **\sl** statement.

Syntax

\sl*n*

Parameter

n

Specifies the amount of vertical spacing, in twips. If this parameter is a positive value, Windows Help uses this value if it is greater than the tallest character. Otherwise, Windows Help uses the height of the tallest character as the line spacing. If this parameter is a negative value, Windows Help uses the absolute value of the number even if the tallest character is taller.

Remarks

If the **\sl** statement is not given, Windows Help automatically sets the line spacing by using the tallest character in the line.

See Also

\pard

\strike

The **\strike** statement creates a hot spot. The statement is used in conjunction with a **\v** statement to create a link to another topic. When the user chooses a hot spot, Windows Help displays the associated topic in the Help window.

The **\strike** statement applies to all subsequent text up to the next **\plain** or **\strike0** statement.

Remarks

No **\plain** or **\strike0** statement is required if the **\strike** statement and subsequent text are enclosed in braces. Braces limit the scope of a character property statement to just the enclosed text.

The **\strike** statement creates the same type of hot spot as the **\uldb** statement.

In print-based documents, or whenever it is not followed by **\v**, the **\strike** statement creates strikeout text.

Example

The following example creates a hot spot for a topic. When displayed, the hot-spot text, "Hot Spot," is green and has a solid line under it:

```
{\strike Hot Spot}{\v Topic}
```

See Also

\ul, **\uldb**, **\v**

\tab

The **\tab** statement inserts a tab character (ASCII character code 9).

Remarks

The tab character (ASCII character code 9) has the same effect as the **\tab** statement.

See Also

\tqc, **\tqr**, **\tx**

\tb

The Microsoft Windows Help Compiler ignores this statement.

See Also

\tab, **\tqc**, **\tqr**, **\tx**

\tqc

The **\tqc** statement is used with the **\tx** statement to create a tab stop where text is centered. For example, the following statement creates a centered tab stop at 2880 twips:

```
\tqc\tx2880
```

See Also

\tab, **\tqr**, **\tx**

\tqr

The **\tqr** statement is used with the **\tx** statement to create a tab stop where text right-justified. For example, the following statement creates a right-justified tab stop at 2880 twips:

```
\tqr\tx2880
```

See Also

\tab, **\tqc**, **\tx**

\trgaph

The **\trgaph** statement specifies the amount of space between text in adjacent cells in a table. For each cell in the table, Windows Help uses the space to calculate the cell's left and right margins. It then uses the margins to align and wrap the text in the cell. Windows Help applies the same margin widths to each cell ensuring that paragraphs in adjacent cells have the specified space between them.

The **\trgaph** statement applies to cells in all subsequent rows of a table up to the next **\trowd** statement.

Syntax

\trgaph*n*

Parameter

n

Specifies the space, in twips, between text in adjacent cells. If this parameter exceeds the actual width of the cell, the left and right margins are assumed to be at the same position in the cell.

Remarks

The width of the left margin in the first cell is always equal to the space specified by this statement. The **\trleft** statement is typically used to move the left margin to a position similar to the left margins in all other cells.

This statement was first supported in the Microsoft Help Compiler version 3.1.

Example

The following example creates a three-column table with one-quarter inch space between the text in the columns:

```
\trgaph360 \cellx1440\cellx2880\cellx4320
\intbl
Row 1 Column 1\cell
Row 1 Column 2\cell
Row 1 Column 3\cell \row
\intbl
Row 2 Column 1\cell
Row 2 Column 2\cell
Row 2 Column 3\cell \row \pard
```

See Also

\cell, **\cellx**, **\intbl**, **\row**, **\trleft**, **\trowd**

\trleft

The **\trleft** statement sets the position of the left margin for the first (leftmost) cell in a row of a table. This statement applies to the first cell in all subsequent rows of the table up to the next **\trowd** statement.

Syntax

\trleft*n*

Parameter

n
Specifies the relative position, in twips, of the left margin. This parameter can be a positive or negative number. The final position of the left margin is the sum of the current position and this value.

Remarks

This statement was first supported in the Microsoft Help Compiler version 3.1.

Example

The following example creates a three-column table with one-quarter inch space between the text in the columns. The left margin in the first cell is flush with the left margin of the Help window:

```
\trgaph360\trleft-360 \cellx1440\cellx2880\cellx4320
\intbl
Row 1 Column 1\cell
Row 1 Column 2\cell
Row 1 Column 3\cell \row
\intbl
Row 2 Column 1\cell
Row 2 Column 2\cell
Row 2 Column 3\cell \row \pard
```

See Also

\cell, **\cellx**, **\intbl**, **\row**, **\trgaph**, **\trowd**

\trowd

The **\trowd** statement sets default margins and cell positions for subsequent rows in a table.

Remarks

This statement was first supported in the Microsoft Help Compiler version 3.1.

See Also

\cell, **\cellx**, **\intbl**, **\row**, **\trgaph**, **\trleft**

\trqc

The **\trqc** statement directs Windows Help to dynamically adjust the width of table columns to fit in the current window.

Remarks

In a print-based document, the **\trqc** statement centers a table row with respect to its containing column.

Windows Help will not resize a table to smaller than the widths specified in the **\trqc** statement. Therefore, the table should be created in the smallest size in which it would ever be displayed. All columns in the table are sized proportionally.

This statement was first supported in the Microsoft Help Compiler version 3.1.

See Also

\trowd, **\trql**

\trql

The **\trql** statement aligns the text in each cell of a table row to the left.

Remarks

This statement was first supported in the Microsoft Help Compiler version 3.1.

See Also

\trowd, **\trqc**

\tx

The **\tx** statement sets the position of a tab stop. The position is relative to the left margin of the Help window. A tab stop applies to all subsequent paragraphs up the next **\pard** statement.

Syntax

\tx*n*

Parameter

n
Specifies the tab stop position, in twips.

Remarks

If the **\tx** statement is not given, tab stops are set at every one-half inch by default.

See Also

\pard, **\tab**, **\tqc**, **\tqr**

\ul

The **\ul** statement creates a link to a pop-up topic. The statement is used in conjunction with a **\v** statement to create a link to another topic. When the user chooses the link, Windows Help displays the associated topic in a pop-up window.

The **\ul** statement applies to all subsequent text up to the next **\plain** or **\ul0** statement.

Remarks

No **\plain** or **\ul0** statement is required if the **\ul** statement and subsequent text are enclosed in braces. Braces limit the scope of a character property statement to just the enclosed text.

In print-based documents, or whenever it is not followed by **\v**, the **\ul** statement creates a continuous underline.

Example

The following example creates a pop-up link for a topic. When displayed, the link text, "Popup Link," is green and has a dotted line under it:

```
{\ul Popup Link}{\v PopupTopic}
```

See Also

\plain, **\strike**, **\uldb**, **\v**

\uldb

The **\uldb** statement creates a hot spot. This statement is used in conjunction with a **\v** statement to create a link to another topic. When the user chooses a hot spot, Windows Help displays the associated topic in the Help window.

The **\uldb** statement applies to all subsequent text up to the next **\plain** or **\uldb0** statement.

Remarks

No **\plain** or **\uldb0** statement is required if the **\uldb** statement and subsequent text are enclosed in braces. Braces limit the scope of a character property statement to just the enclosed text.

The **\uldb** statement creates the same type of hot spot as the **\strike** statement.

Example

The following example creates a hot spot for a topic. When displayed, the hot-spot text, "Hot Spot", is green and has a solid line under it:

```
{\uldb Hot Spot}{\v Topic}
```

See Also

\plain, **\strike**, **\ul**, **\v**

\v

The **\v** statement creates a link to the topic having the specified context string. The **\v** statement is used in conjunction with the **\strike**, **\ul**, and **\uldb** statements to create hot spots and links to topics.

Syntax

{\v *context-string***}**

Parameter

context-string
Specifies the context string of a topic in the Help file. The string can be any combination of characters, except spaces, and must also be specified in a context-string **\footnote** statement in some topic in the Help file.

Remarks

If the context string is preceded by a percent sign (**%**), Windows Help displays the associated hot spot or link without applying the standard underline and color. If the context string is preceded by an asterisk (*****), Windows Help displays the associated hot spot or link with an underline but without applying the standard color.

In print-based documents, the **\v** statement creates hidden text.

For links or hot spots, the syntax of the **\v** statement is as follows:

[**%**|*****] *context* [**>***secondary-window*] [**@***filename*]

In this syntax, *secondary-window* is the name of the secondary window to jump to. When the secondary window is not specified, the jump is to the same window as the current Help topic is using. To jump to the main window, specify "main" for this parameter. This parameter may not be used with pop-up windows.

The *filename* parameter specifies a jump to a topic in a different Help file.

For a macro hotspot, the syntax of the **\v** statement is as follows:

[**%**|*****] **!** *macro* [**;***macro*][**;**]

Example

The following example creates a hot spot for the topic having the context string "Topic". Windows Help applies an underline and the color green the text "Hot Spot" when it displays the topic:

```
{\uldb Hot Spot}{\v Topic}
```

See Also

\footnote, **\strike**, **\ul**, **\uldb**

\wbitmap

The **\wbitmap** statement sets the picture type to Windows bitmap. This statement must be used in conjunction with a **\pict** statement.

Syntax

\wbitmap*n*

Parameter

n
Specifies the bitmap type. This parameter is zero for a logical bitmap.

Remarks

The **\wbitmap** statement is optional; if a **\wmetafile** statement is not specified, the picture is assumed to be a Windows bitmap.

Example

The following example creates a 32-by-8 pixel monochrome bitmap:

```
{\pict \wbitmap0\wbmbitspixel1\wbmplanes1\wbmwidthbytes4\picw32\pich8
3FFFFFFC
F3FFFFCF
FF3FFCFF
FFF3CFFF
FFFC3FFF
FFCFF3FF
FCFFFF3F
CFFFFFF3
}
```

See Also

bmc, **bml**, **bmr**, **\pict**, **\wmetafile**

\wbmbitspixel

The **\wbmbitspixel** statement specifies the number of consecutive bits in the bitmap data that represent a single pixel. This statement must be used in conjunction with the **\pict** statement.

Syntax

\wbmbitspixel*n*

Parameter

n
Specifies the number of bits per pixel.

Remarks

If the **\wbmbitspixel** statement is not given, the default bits per pixel value is 1.

See Also

\pict, **\wbitmap**, **\wbmplanes**

\wbmplanes

The **\wbmplanes** statement specifies the number of color planes in the bitmap data. This statement must be used in conjunction with a **\pict** statement.

Syntax

\wbmplanes*n*

Parameter

n
Specifies the number of bitmap planes.

Remarks

If the **\wbmplanes** statement is not given, the default number of planes is 1.

See Also

\pict, **\wbitmap**, **\wbmbitspixel**

\wbmwidthbytes

The **\wbmwidthbytes** statement specifies the number of bytes in each scan line of the bitmap data. This statement must be used in conjunction with the **\pict** statement.

Syntax

\wbmwidthbytes*n*

Parameter

n
Specifies the width of the bitmap, in bytes.

See Also

\pict, **\wbitmap**

\windows

The **\windows** statement sets the Windows character set.

Remarks

If no **\windows**, **\pc**, or **\pca** statement is given in the Help file, the Windows character set is used by default.

See Also

\ansi, **\pc**, **\pca**

\wmetafile

The **\wmetafile** statement sets the picture type to a Windows metafile. This statement must be used in conjunction with the **\pict** statement.

Syntax

\wmetafile*n*

Parameter

n
Specifies the metafile type. This parameter must be 8.

Remarks

Windows Help expects the hexadecimal data associated with the picture to represent a valid Windows metafile. By default, Windows Help sets the MM_ANISOTROPIC mapping mode prior to displaying the metafile. To ensure that the picture is displayed correctly, the metafile data must either set the window origin and extents by using the **SetWindowOrg** and **SetWindowExt** records or set another mapping mode by using the **SetMapMode** record.

Example

The following example creates a picture using a metafile:

```
{{\pict\wmetafile8\picw2880\pich2880
0100090000034f0000000200090000000000
050000000b0200000000050000000c026400
6400090000001d066200ff00640064000000
000008000000fa0200000200000000000000
040000002d0100000500000140200000000
0500000013026400640005000000014020000
64000500000013026400000008000000fa02
0000000000000000000040000002d010100
04000000f00100000300000000004e0dff00
870020000050000020000000000000000000}
\par }
```

See Also

SetMapMode, **SetWindowExt**, **SetWindowOrg**, **bmc**, **bml**, **bmr**, **\pict**, **\wbitmap**

Help Compiler Error Messages

This section lists the error messages displayed by the Microsoft Help Compiler when it encounters errors in building a Help file. Whenever possible, the compiler displays the name of the file that contains the error, as well as the number used to identify the specific line of the project file or the topic that produced the error. Since topics are not actually numbered, the topic number given with an error message refers to that topic's sequential position in the topic file.

Interpreting Error Messages

The Microsoft Help Compiler displays either warning or fatal-error messages. A warning message indicates a problem during compilation that was not severe enough to prevent the Help file from being created. Microsoft Windows Help should be able to open the file but may encounter problems when displaying some topics. A fatal error indicates a problem that prevents the compiler from creating a Help file. The compiler always reports fatal errors, regardless of the current warning level or reporting option.

While the Microsoft Help Compiler processes the project file, it ignores lines that contain errors and attempts to continue. This means that errors encountered early in the file may result in many more errors being reported as the compiler continues.

When the Microsoft Help Compiler processes topic files, it reports any errors it encounters, and if the errors are not fatal, compilation continues. A single error in a topic file may result in more than one error message being displayed by the compiler. For example, a typographic mistake in a topic's context string will cause an error to be reported every time the compiler encounters a reference to the correct topic identifier.

Error Message Categories

Error-message numbers have four digits; the first one or two of those digits identify the message category. The message-number prefixes and the categories they identify are defined as follows:

Prefix	Error
1	Problems with files used to build the Help file
2	Problems with the project file
30 - 31	Problems with build tags or build-tag expressions
35 - 36	Problems with Help macros
40 - 41	Problems with context strings
42 - 45	Problems with footnotes
46 - 47	Problems with the topic file
5	Other problems

File Errors

The following messages result from problems with files used to build a Help file. A description is given for messages that are not self-explanatory.

Number	File error message
1019	**Project file extension cannot be .HLP or .PH.**
1030	**File name exceeds limit of 259 characters.** The combined length of the path and filename must not be more than the MS-DOS limit of 259 characters.
1079	**Out of file handles.** The compiler does not have enough available file handles to continue. If possible, increase the **FILES** setting in the CONFIG.SYS file.
1100	**Cannot open file** *filename***: permission denied.** Requested files must have at least read privilege to be opened.
1150	**Cannot overwrite file** *filename*. Files with the read-only attribute cannot be overwritten.
1170	**File** *filename* **is a directory.** A directory in the project directory has the same name as the requested Help file.
1190	**Cannot use reserved DOS file name** *filename*. Do not use reserved MS-DOS filenames, such as COM1, LPT2, or PRN, when specifying topic or other data files.
1230	**File** *filename* **not found.** The specified file could not be found or is unreadable.
1292	**File** *filename* **is not a valid bitmap.** The specified bitmap file could not be found or is not in a recognizable bitmap format.
1319	**Disk full.**
1513	**Bitmap name** *filename* **duplicated.** The [BITMAPS] section contains duplicate bitmap names. The compiler uses the first occurrence of the name.
1536	**Not enough memory to compress bitmap** *filename*. The specified bitmaps cannot be compressed due to insufficient memory.

Project-File Errors

The following messages result from errors in the Help project file (with the .HPJ filename extension) used to build a Help file. A description is given for messages that are not self-explanatory.

Number	Project-file error message
2010	**Include statements nested more than 5 deep.** The **#include** statement on the specified line has exceeded the maximum of five include levels.
2030	**Comment starting at line** *linenumber* **of file** *filename* **unclosed at end of file.** The compiler has unexpectedly come to the end of the project file. There may be an open comment in the project file or in an include file.
2050	**Invalid #include syntax.** The **#include** statement requires a filename.
2091	**Bracket missing from section heading [***sectionname***].**
2111	**Section heading missing.** The section heading on the specified line is not complete. This error is also reported if the first entry in the project file is not a section heading.
2131	**Invalid OPTIONS syntax: 'option=value' expected.**
2141	**Invalid ALIAS syntax: 'context=context' expected.**
2151	**Incomplete line in [***sectionname***] section**
2171	**Unrecognized text.**
2191	**Section heading [***sectionname***] unrecognized.**
2214	**Line in .HPJ file exceeds length limit of 2047 characters.**
2273	**[OPTIONS] should precede [FILES] and [BITMAPS] for all options to take effect.** The [OPTIONS] section should be the first section in the project file. Also, if the **ERRORLOG** option is used, that option should be the first line in the [OPTIONS] section.
2291	**Section** *sectionname* **previously defined.** The compiler ignores the lines under the duplicated section and continues from the next valid section heading.
2305	**No valid files in [FILES] section.** The file section is empty or contains only invalid files.
2322	**Context string** *context_name* **cannot be used as alias string.** A context string that has been assigned an alias cannot be used later as an alias for another context string. That is, you cannot map **a=b** and then **c=a** in the [ALIAS] section. The compiler ignores the attempted reassignment.
2331	**Context number already used in [MAP] section.** The context number on the specified line in the project file was previously mapped to a different context string.
2341	**Invalid or missing context string.** The specified line is missing a context string before an equal sign.

2351 **Invalid context identification number.**

The context number on the specified line is empty or contains invalid characters.

2362 **Context string** *context_name* **already assigned an alias.**

A context string can have only one alias. That is, you cannot map **a=b** and then **a=c** in the [ALIAS] section. The specified context string has already been assigned an alias in the [ALIAS] section. The compiler ignores the attempted reassignment.

2372 **Alias string** *aliasname* **already assigned.**

You cannot alias an alias. That is, an alias string cannot, in turn, be assigned another alias. You cannot map **a=b** and then **b=c** in the [ALIAS] section. The compiler ignores the attempted reassignment.

2391 **Limit of 6 window definitions exceeded.**

The maximum number of window definitions is one main-window definition and five secondary-window definitions.

2401 **Window maximization state must be 0 or 1.**

The *sizing* parameter in a window definition must be either zero or 1.

2411 **Invalid syntax in window color.**

A window color in a window definition consists of three decimal numbers enclosed in parentheses and separated by commas.

2421 **Invalid window position.**

The window position in a window definition consists of four decimal numbers enclosed in parentheses and separated by commas.

2431 **Missing quote in window caption.**

The window caption in a window definition must be enclosed in quotation marks.

2441 **Window name** *windowname* **is too long.**

The window name exceeds the maximum length of 8 characters.

2451 **Window position value out of range 01023.**

One or more of the window-position coordinates exceed the maximum limit of 1023.

2461 **Window name missing.**

A window definition in the project file is missing the window name.

2471 **Invalid syntax in [WINDOWS] section.**

2481 **Secondary-window position required.**

A window definition for a secondary window must specify the four window-position parameters.

2491 **Duplicate window name** *windowname*.

Window names must be unique.

2501 **Window caption** *windowcaption* **exceeds limit of 50 characters.**

2511 **Unrecognized option** *optionname* **in [OPTIONS] section.**

2532 **Option** *optionname* **previously defined.**

The compiler ignores the attempted redefinition.

2550 **Invalid path** *pathname* **in** *optionname* **option.**

The compiler cannot find the path specified by the **ROOT** or **BMROOT** option. The compiler uses the current working directory.

2570 **Path in** *optionname* **option exceeds** *number of* **characters.**

The specified root path exceeds the maximum limit for MS-DOS. The compiler ignores the path and uses the current working directory.

2591 **Invalid MAPFONTSIZE option.**

The font range syntax used is invalid. A font range consists of a low and high point size, separated by a hyphen (-).

2612 **Maximum of 5 font ranges exceeded.**

The compiler ignores additional ranges.

2632 **Current font range overlaps previously defined range.**

The compiler ignores the second mapping.

2651 **Font name exceeds limit of 20 characters.**

2672 **Unrecognized font name** *fontname* **in FORCEFONT option.**

The compiler ignores the font name and uses the default Helvetica font.

2691 **Invalid MULTIKEY syntax.**

The **MULTIKEY** option must specify a single capital letter other than the letter *K*.

2711 **Maximum of 5 keyword tables exceeded.**

The compiler ignores the additional tables.

2732 **Character already used.**

A character used for indicating the keyword table was previously used. The compiler ignores the line.

2752 **Characters 'K' and 'k' cannot be used.**

These characters are reserved for Help's standard keyword table.

2771 **REPORT option must be 'ON' or 'OFF'.**

2811 **OLDKEYPHRASE option must be 'ON' or 'OFF'.**

2832 **COMPRESS option must be 'OFF', 'MEDIUM' or 'HIGH'.**

2842 **OPTCDROM option must be 'TRUE' or 'FALSE'.**

2852 **Invalid TITLE option.**

The **TITLE** option defines a string that is empty or contains more than 32 characters.

2872 **Invalid LANGUAGE option.**

You have specified an ordering that is not supported by the compiler. The compiler uses English sorting order.

2893 **Warning option must be 1, 2, or 3.**

The compiler uses full reporting (level 3).

2911 **Invalid icon file** *filename*.

The compiler cannot find the icon file specified in the **ICON** option, or the file is not a valid icon file.

2932 **Copyright string exceeds limit of 50 characters.**

The maximum length of the copyright string in the About box is limited to 50 characters.

3011 **Maximum of 32 build tags exceeded.**

The compiler ignores the additional tags.

3031 **Build tag length exceeds 32 characters.**

The compiler ignores the build tag.

3051 **Build tag** *tagname* **contains invalid characters.**

Build tags can contain only alphanumeric characters or the underscore (_) character.

3076 **[BUILDTAGS] section missing.**

The **BUILD** option declared a conditional build, but there is no [BUILDTAGS] section in the project file. The compiler includes all topics in the build.

3096 **Build expression too complex.**

The build expression has too many expressions (**~**, **|**, or **&**) or is too deeply nested.

3116 **Invalid build expression.**

The syntax used in the build expression on the specified line contains one or more logical or syntax errors.

3133 **Duplicate build tag in [BUILDTAGS] section.**

3152 **Build tag** *tagname* **not defined in [BUILDTAGS] section.**

The specified build tag has been assigned to a topic but not declared in the project file. The compiler ignores the tag for the topic.

3178 **Build expression missing from project file.**

The topics have build tags, but there is no **build** expression in the project file. The compiler includes all topics in the build.

Macro Errors

The following messages result from errors in the use of Help macros in footnotes, hot spots, and the [CONFIG] section of the Help project file. A description is given for messages that are not self-explanatory.

Number	Macro error message
3511	**Macro** *macrostring* **exceeds limit of 254 characters.**
3532	**Undefined function in macro** *macroname***.** The specified macro is not on the list of macros supported by the compiler, nor is it specified in the **RegisterRoutine** macro. The compiler passes the macro to the Help file, however.
3552	**Undefined variable in macro** *macroname***.**
3571	**Wrong number of parameters to function in macro** *macroname***.**
3591	**Syntax error in macro** *macroname***.**
3611	**Function parameter type mismatch in macro** *macroname***.** There is a type mismatch (string or numeric) in the function call.
3631	**Bad macro prototype.** The prototype string passed to the **RegisterRoutine** macro is invalid.
3652	**Empty macro string.** The ! footnote or a hidden text starting with "!" does not contain a macro.
3672	**Macro** *macroname* **nested too deeply.** Macro strings that contain macro strings as parameters may not nest more than three deep.

Context-String Errors

The following messages are caused by problems with context-string footnotes or context strings specified in jumps or in Help project-file options. A description is given for messages that are not self-explanatory.

Number	Context-string error messages
4011	**Context string** *contextname* **already used.** The specified context string was previously assigned to another topic. The compiler ignores the latter string and the topic has no identifier.
4031	**Invalid context string** *contextname***.** The context string footnote contains non-alphanumeric characters or is empty. The compiler does not assign the topic an identifier.
4056	**Unresolved context string specified in CONTENTS option.** The Contents topic defined in the project file could not be found. The compiler uses the first topic in the build as the Contents topic.
4072	**Context string exceeds limit of 255 characters.** The compiler ignores the context string.
4098	**Context string(s) in [MAP] section not defined in any topic.** The compiler cannot find a context string listed in the [MAP] section in any of the topics in the build.
4113	**Unresolved jump or popup** *contextname***.** The specified topic contains a context string that identifies a nonexistent topic.
4131	**Hash conflict between** *contextname* **and** *contextname***.** The hash algorithm has generated the same hash value for both of the listed context strings. Change one of the context strings and recompile.
4151	**Invalid secondary window name** *windowname***.** The window name for the secondary window is "main" or another disallowed member name.
4171	**Cannot use secondary window with popup.** The hidden text defining the pop-up identifier contains a secondary-window name.
4196	**Jumps and lookups not verified.** Due to low memory conditions, the build continues without verifying the validity of jumps and popups. (The reference to "lookups" in the error message is incorrect.)
4211	**Footnote text exceeds limit of 1023 characters.** Footnote text cannot exceed the limit of 1023 characters. The compiler ignores the footnote.
4231	**Footnote text missing.** The specified topic contains a footnote that has no characters.
4251	**Browse sequence not in first paragraph.** The browse-sequence footnote is not in the first paragraph of the topic. The compiler ignores the browse sequence.

4272 **Empty browse sequence string.**

The browse-sequence footnote for the specified topic contains no sequence characters.

4292 **Missing sequence number.**

A browse-sequence number ends in a colon (**:**) for the specified topic. Remove the colon or enter a "minor" sequence number and then recompile.

4312 **Browse sequence already defined.**

A browse-sequence footnote already exists for the specified topic. The compiler ignores the latter sequence.

4331 **Title not in first paragraph.**

The title footnote (**$**) is not in the first paragraph of the topic. The topic will not have a topic title string.

4352 **Empty title string.**

The title footnote for the specified topic contains no characters. The compiler does not assign the topic a title.

4372 **Title defined more than once.**

There is more than one title footnote in the specified topic. The compiler uses the first title string.

4393 **Title exceeds limit of 128 characters.**

The compiler ignores the additional characters.

4412 **Keyword string exceeds limit of 255 characters.**

4433 **Empty keyword string.**

There are no characters in the keyword footnote.

4452 **Keyword(s) defined without title.**

The topic has a keyword assigned to it, but no title.

4471 **Build tag footnote not at beginning of topic.**

The build-tag footnote marker, if used, must be the first character in the topic.

4492 **Build tag exceeds limit of 32 characters.**

The compiler ignores the tag for the topic.

4551 **Entry macro not in first paragraph.**

The ! footnote (for executing a macro) is not in the first paragraph of the topic. The compiler ignores the macro.

Topic-File Errors

The following messages result from problems in rich-text format (RTF) formatting in one or more topic files. A description is given for messages that are not self-explanatory.

Number	Topic-file error message
4616	**File** *filename* **is not a valid RTF topic file.**
4639	**Error in file** *filename* **at byte offset 0x***offset***.** The specified file contains unrecognized RTF at that byte offset.
4649	**File** *filename* **contains more than 32767 topics.**
4652	**Table formatting too complex.** The compiler encountered a table with borders, shading, or right justification.
4662	**Side by side paragraph formatting not supported.** The side-by-side paragraph formatting is not supported in Microsoft Windows Help 3.1.
4671	**Table contains more than 32 columns.**
4680	**Font** *fontname* **in file** *filename* **not in RTF font table.** The compiler uses the default system font.
4692	**Unrecognized graphic format.** The compiler supports only Windows bitmaps, Windows metafiles, segmented graphics, and multi-resolution graphics. The compiler ignores the graphic.
4733	**Hidden page break.** A page break is a part of the hidden text. A page break formatted as hidden text will not separate two topics.
4753	**Hidden paragraph.** A paragraph marker is part of the hidden text. The compiler ignores the paragraph marker.
4763	**Hidden carriage return.** A carriage return is part of the hidden text. The compiler ignores the carriage return.
4774	**Paragraph exceeds limit of 64K.** A single paragraph has more than 64K of text or 64K of graphics. (This limit does not include graphics stored separately from the data, using the **bmc**, **bml**, or **bmr** statements.)
4792	**Non-scrolling region defined after scrolling region.** A **\keepn** statement precedes a paragraph that is not the first paragraph in the topic. The compiler ignores the statement; the paragraph is treated as regular text and is part of the regular topic text.
4813	**Non-scrolling region crosses page boundary.** A **\pard** statement must appear before the **\page** statement in a topic containing a **\keepn** statement.

Miscellaneous Errors

The following messages are caused by conditions such as operating-system file errors or out-of-memory conditions. A description is given for messages that are not self-explanatory.

Number	Error message
5035	**File** *filename* **not created.** There are no topics to compile or the build expression is false for all topics. The compiler does not create a Help file.
5059	**Not enough memory to build Help file.** To free memory, unload any unneeded applications, device drivers, and memory-resident programs.
5075	**Help Compiler corrupted. Please reinstall HC.EXE.** Virus-checking code has detected a corruption in the compiler. Reinstall the compiler.
5098	**Using old key-phrase table.** Maximum compression can result only by deleting the .PH file before each recompilation of the Help topics or by setting the **OLDKEYPHRASE** option to "0".
5115	**Write failed.** Write-to-disk operation failed. Contact Microsoft Product Support Services.
5139	**Aborted by user.** Compilation was terminated when the user pressed CTRL+C.

Using Symbolic Debuggers

The Microsoft Windows NT Software Development Kit (SDK) includes three symbolic debuggers:

- NTSD, the NT Symbolic Debugger with which you can debug user-mode programs.
- KD, the Kernel Debugger with which you can debug kernel-mode device drivers. KD provides assembly language debugging and is intended for system developers and device driver authors. NTSD and KD share much of their syntax and support many identical commands.
- WinDbg, the Windows-based source-level debugger with which you can debug both user-mode programs and kernel-mode drivers.

In most cases, you can use WinDbg to debug application programs, user-mode drivers, and kernel-mode drivers. You may have to use NTSD and KD to debug drivers that initialize before you can run WinDbg or that interfere with WinDbg itself. KD is a kernel-mode remote debugger that runs on a separate host computer. KD provides assembly language debugging and is intended for system developers and device driver authors. NTSD and KD share much of their syntax and support many identical commands. This section explains how to use NTSD and KD. Introducing WinDbg explains how to use WinDbg.

NTSD supports multiple-thread and multiprocess debugging of user-mode applications and DLLs. NTSD is extensible, and can read and write both paged and nonpaged memory.

Using NTSD, you can display and execute program code, set breakpoints, and examine and change values in memory. Because NTSD can access memory locations through addresses, global symbols, or line numbers, you can refer to data and instructions by name rather than by address, making it easy to locate and debug specific sections of code. You can debug C programs at source-file level as well as at machine level. You can display source statements, disassembled machine code, or a combination of both.

KD supports debugging of kernel-mode code. It also supports multiprocessor debugging. Because it is deficient in setting breakpoints in user-mode code and cannot be used to examine or deposit paged-out memory, KD is not well suited to debugging user-mode programs. KD does not provide support for threads.

KD runs on a host computer and communicates through a null-modem serial cable with the target computer running NT. The host computer must be running Windows NT.

The NT Symbolic Debugger (NTSD)

This section describes operation of the NT Symbolic Debugger. Summaries of NTSD commands are included.

The following topics are covered:

- TOOLS.INI
- NTSD Command Line
- Using NTSD

For a complete reference of NTSD commands, see NTSD and KD Reference.

TOOLS.INI

On startup, NTSD searches for an [NTSD] header in the TOOLS.INI file and extracts initialization information from the entries under the header. The environment variable INIT must point to the directory containing the TOOLS.INI file.

NTSD also attempts on startup to take commands from the file NTSD.INI in the current directory or in the path specified in the TOOLS.INI file.

The TOOLS.INI entries for the [NTSD] section are as follows:

$u0-$u9
: Assign values to user-defined pseudo registers.

DebugChildren
: True or false. If true, NTSD debugs the specified application as well as any children that it may spawn.

DebugOutput
: True or false. If true, NTSD sends output and receives input through a terminal.

IniFile
: Specifies the name of the script file NTSD takes commands from at startup.

LazyLoad
: True or false. If true, NTSD performs lazy symbol loading; that is, symbols are not loaded until needed and NTSD starts faster.

StopFirst
: True or false. If true, NTSD stops on the first breakpoint it encounters.

StopOnProcessExit
: True or false. If true, NTSD stops when it receives a process termination notification.

sxd, sxe
: Set exception handling for the following events:

Code	Description
3c	Child App Termination (break on GUI application exit)
av	Access Violation
cc	CTRL+C
ct	Create Thread
et	Exit Thread
ep	Exit Process
ld	Load DLL

: The **sxe** command enables the break status for an exception; the **sxd** command disables the break status for an exception.

VerboseOutput
: True or false. If true, NTSD will display detailed information on symbol handling, event notification, and other run-time occurrences.

An example TOOLS.INI file follows:

```
[ntsd]
sxe: 3c
sxe: cc
$u0: VeryLongName
VerboseOutput:true
```

NTSD Command Line

The NTSD command line uses the following syntax:

NTSD [*options*] *imagefile*

where *imagefile* is the name of the image to be debugged and *options* is one of the following:

Option	Description
-2	Opens a new window for debugging character mode applications.
-d	Redirects output to the debugging terminal.
-g	Causes execution past the first breakpoint automatically.
-G	Causes NTSD to exit immediately when the child terminates.
-o	Enables debugging of multiple processes. The default is for one process directly spawned by the debugger.
-p *process-id*	Specifies debugging of the process identified by *process-id*.
-v	Produces verbose output.

Using NTSD

Before using NTSD, follow these steps:

1. On *x*86 platforms, compile your program using the CL386 option **/Zd**. This generates debugger symbol information for your .EXE file.
2. On *x*86 platforms, generate your object file using CVTOMF with the **-g** option.
3. Link your program using LINK with the **/DEBUG:full** option.

Note The MSTOOLS directory includes examples of these options.

Starting NTSD

To start debugging your program with NTSD, enter the following:

```
ntsd yourexename.exe
```

This starts NTSD in the current console window and loads the application specified by yourexename.exe. From the NTSD prompt, set any initial breakpoints with the bp command, and type **g** (**go**) to start the application. The **g** command directs NTSD to pass control to the program. Execution begins at the current code address.

Note NTSD cannot access symbols if the .EXE or .DLL being debugged is on drive C and the system was restarted from a drive other than drive C.

If the application hits a breakpoint or has an access violation, you will be returned to the NTSD prompt. It is often handy to run NTSD on the kernel debugging computer. This allows you to continue to watch the NTSD screen while the application is running; you can break into NTSD with CTRL+C. To start NTSD with its display redirected to KD, enter

```
ntsd -d appname
```

To break into NTSD, use the minus key (below PRINT SCREEN) on an 84-key keyboard; on a 101/102-key keyboard, use F12.

On invocation of NTSD, you can use the **-g** option to go past the first breakpoint automatically. For example, the command line

```
ntsd -g ls
```

causes NTSD to pass control to LS, which executes until it terminates or until an exception causes NTSD to take control.

Using Basic NTSD Commands

These are the most important commands for getting started debugging your application. NTSD and KD Reference contains the complete list of NTSD commands.

Command	Meaning
?	Display command summary
bp	Set breakpoint, for example "bp main"
g	Go – continue execution
ln	List near symbols
p	Single step
r	Display register contents, set register
s+	Set source display mode
u	"Unassemble" (disassemble)
v	View source lines

Canceling NTSD Commands

To cancel the current command, use SYS RQ on the target keyboard (ALT-SYS RQ on enhanced keyboards) for MIPS platforms; for *x*86 platforms, use CTRL+C. This action is most useful for halting commands that generate long output listings on the display.

If NTSD is running, program execution is stopped. Entering CTRL+C will not affect commands that pass execution control to the application being debugged.

Quitting NTSD

You can terminate NTSD at any time by using the **q** command to return to the operating system prompt.

Basic Debugging

Follow this sample procedure to get started debugging your application:

1. Type "ntsd *yourappname*" to start NTSD.
2. Set source display mode using the **s+** command.
3. Set a breakpoint at "main" or "winmain" using the **bp** command.
4. Go to this breakpoint using the **g** command.
5. Unassemble code from this point using the **u** command.
6. Type "v .23 l10" to view 16 (10 hex) source lines starting at line 23.
7. Set more breakpoints as necessary. If the virtual address corresponding to the next line number you want to stop at is 41000a, type "bp 41000a".
8. Use the **p** command to single step or the **g** command to run until the next breakpoint is encountered.

The Kernel Debugger (KD)

This section describes basic procedures for using KD, the kernel debugger. The following topics are included:

- Setting Up KD
- Using KD
- Quitting KD

Setting Up KD

Preparing a Cable

To use KD to debug kernel-mode executables from a remote location, you must connect a null-modem serial cable to the serial ports of the host computer running KD and the NT computer running the executables. The following two tables list these connections:

KD Serial Cable Connections

KD Serial port	Connected to	NT Serial port
Transmit		Receive
Receive		Transmit
Ground		Ground

KD 9-pin and 25-pin D-subminiature Cable Connections

Pin	Connected to	Pin
2		3
3		2
7		7

Connecting KD

First connect the Windows NT computer and KD computer with a standard null-modem serial cable. On the Windows NT (target) computer, connect the cable to the highest-numbered serial port. For example, on a computer with two serial ports, connect it to COM2. On the debugger (host) computer, connect it to COM1. See the instructions below if you need to use a different port on the debugger computer.

{ewl msdncd, EWGraphic, group10552 0 /a "SDK.BMP"} To connect KD

1. Install a debug version of Windows NT on the debugger computer.
2. For *x*86 platforms: copy \NT\MSTOOLS\BIN\I386KD.EXE from the development computer to \NT\BIN on the debugger computer; alternatively, copy \DEBUG\MSTOOLS\BIN\I386\I386KD.EXE from the CD-ROM to \NT\BIN on the debugger computer.

   For MIPS platforms: copy \NT\MSTOOLS\BIN\MIPSKD.EXE from the development computer to \NT\BIN on the debugger computer; alternatively, copy \DEBUG\MSTOOLS\BIN\MIPS\MIPSKD.EXE from the CD-ROM to \NT\BIN on the debugger computer.
3. Create a batch file beginning with these two lines:

   ```
   SET _NT_DEBUG_PORT=COM1
   SET _NT_DEBUG_LOG_FILE_APPEND=C:\DEBUG.LOG
   ```

   The last line in this batch file is platform dependent. For *x*86 platforms add

   ```
   i386kd
   ```

   and for MIPS platforms add

   ```
   mipskd
   ```

4. Run the batch file to execute KD. The KD debugging session is logged in the file specified by the _NT_DEBUG_LOG_FILE_APPEND variable. To connect the debugging cable to a port other than COM1 (on the debugging computer), use the _NT_DEBUG_PORT variable to set the desired COM port.

Setting Environment Variables

Set environment variables on the debugger (host) computer as follows:

Variable	Meaning
_nt_debug_port=*portname*	Name of the debugger computer port through which the debugger computer and the target computer are connected (for example, _NT_DEBUG_PORT=COM1).
_nt_symbol_path=*pathname*	Name of the root of a directory tree containing image files containing symbols (for example, if you set _NT_SYMBOL_PATH=C:\TESTROOT on the host, and you run \NT\BINARY\TEST1.EXE on the target computer, KD will try to read symbols from C:\TESTROOT\NT\BINARY\TEST1.EXE on the host computer.). If this variable is not provided, X: is the default path.
_nt_debug_log_file_append=*filename*	Name of the log file to which KD appends output.
_nt_debug_log_file_open=*filename*	Name of the log file to which KD sends output.

Starting KD

Note Remember to start KD before starting the Windows NT computer.

To start KD on *x*86 platforms, enter the following command line at the command prompt:

i386kd [**-?**|**-h**|**-H**] [**-b**[**-n**] [**-v**] *exename*

where *exename* is the pathname of the host computer's kernel image file for loading symbols.

To start KD on MIPS platforms, enter the following command line at the command prompt:

mipskd [**-?**] [**-h**] [**-l** *image-file*] [**-n**] [**-t**] [**-v**]

To start KD on PowerPC platforms, enter the following command line at the command prompt:

ppckd

A list of options and their meanings follows:

Option	Meaning
-?	Displays help
-b	Causes initial break in the kernel – *x*86 only
-h	Displays help – *x*86 only
-l *image-file*	Load symbols from *image-file* immediately – MIPS only
-n	No lazy symbol loading
-t	Time out between serial port character I/O – MIPS only
-v	Generates verbose messages for loads, deferred loads, and unloads

If your debugger computer is running NT, entering "i386kd -?" (on an *x*86 platform), "mipskd -?" (on a MIPS platform), or "ppckd -?" (on a PowerPC platform) at the command prompt gives information on environment variables and control keys. At the debugger prompt, typing "?" gives help for built-in commands, Entering "!?" gives help for extension commands, which start with a "!".

Using KD

The following list gives suggestions for effective use of KD and summarizes the most common commands:

Command	Description
ENTER	Repeats the last command.
bp	Sets a breakpoint. Breakpoints cannot be set in user space (addresses below 80000000). Breakpoints in user space transfer control to NTSD if it is debugging the thread that encountered the breakpoint. Breakpoints in system space transfer control to KD.
CTRL+R	Synchronizes with the target computer. Use this command if the debugger computer isn't responding.
CTRL+D	Dumps the MIPSKD state.
CTRL+C	Interrupts commands that generate long output listings on the display (for *x*86 platforms only).

Note KD cannot dump floating-point registers, but it can dump floating-point variables.

Canceling Commands

To cancel KD commands use the following commands:

- SYS RQ on the target keyboard (ALT-SYS RQ on enhanced keyboards)
- SYS RQ on the debugger (host) keyboard (ALT-SYS RQ on enhanced keyboards)
- CTRL+C on the target keyboard (for x86 platforms only)
- CTRL+C on the debugger (host) keyboard

Quitting KD

Use the **q** command to save your log file and quit KD.

Try CTRL+R followed by ENTER on the debugger keyboard if the q command fails to work, and then try using the q command again (to save your log file and quit). Otherwise, press CTRL+B followed by ENTER to locally abort the current session and return to the local command prompt. The log file is not saved when using CTRL+B to quit KD. Press CTRL+B to quit if both the host and target computers are hung.

NTSD and KD Debugging Topics

This section discusses selected aspects of using NTSD and KD.

DLL Load Notification

Symbols are loaded from the DLL itself (if allowed) and are at the location specified by the pathname for the DLL.

Setting Breakpoints with Symbols

You can use the **bp** (breakpoint set) command and symbols to set breakpoints in your application code. The **bp** command uses symbolic names to compute the breakpoint address. For example, for a character mode application, the command

```
bp main
```

sets a breakpoint at the application's **main** function. For a Windows-based application, the command

```
bp winmain
```

sets a breakpoint at the application's **winmain** function.

On MIPS platforms, the breakpoint is automatically set after the prolog.

The "KB" stack trace uses the stack frame to determine which function is executing. Since the stack frame is set up by the first two instructions, you should step a minimum of two assembly-language instructions into a procedure to get an accurate stack trace.

A stack trace after an access violation in a window procedure may be uninformative, since the window procedure is called via a function pointer. If you place a breakpoint in the offending function, you will get a better stack trace.

KD supports virtual breakpoints. If an application has not been loaded, KD sets a virtual breakpoint. A virtual breakpoint has no effect on execution until the application is actually loaded. Once an application is loaded, KD computes the actual coded addresses of all virtual breakpoints and enables them.

Debugging Multiple Processors

KD supports multiple processor computer debugging. Processors are numbered 0 through *n*.

If the current processor is processor 0 (that is, if it is processor 0 that currently causes KD to be active), you can look at the other noncurrent processors (processors 1 through *n*). However, you cannot change anything in the noncurrent processors; you can only look at their state. For example, you can run **k** and **r** commands on other processors, using "1r" to dump the registers of processor 1, and "2k" to do a stack trace on processor 2, etc. But running "1r eax=8080808" gives a syntax error because changing the state of noncurrent processors is not allowed.

The KD commands **BA** (data breakpoint, not implemented on MIPS or PowerPC) and **BP** (breakpoint) apply to all processors of a multiple processor computer. For example, if the current processor is 63 and you type

```
bp SomeAddress
```

to put a breakpoint at SomeAddress, any processor (not just processor 63) executing at that address will cause a breakpoint trap.

Displaying Variables

Use the **d** command to display an application's global variables. These commands can use a variable's symbol as an argument and compute the variable's address.

For example, the **d** command to dump a word

```
dw hwInstance
```

causes NTSD or KD to display the contents of the variable hwInstance, which might be:

```
00000235 0000 0000 0000 0000 0000 0000
```

To display an application's local variables on *x*86 platforms, first compile the application with the **/Zi** compiler option. Use the "dot" command (**.**) to display the contents of local variables.

Using Pseudoregisters

NTSD supports the use and definition of pseudoregisters. Four of these are automatically set to specific debug information:

- $ea is set to the last effective address displayed
- $exp is set to the last expression evaluated
- $ra is set to the return address currently on the stack
- $p is set to the value printed by the last **dd**, **dw**, or **db** command

The pseudoregisters $u0 through $u9 are user-defined and can be used as macros. Pseudoregisters can be used to eliminate typing of long or complex symbol names:

```
0:000> r $u0=usersrv!NameTooLongToWantToType
0:000> dw $u0+8
```

Here is another example using a pseudoregister to simplify typing of commands:

```
0:000> r $u5='dd esp 14;g'
0:000> bp Api1 $u5; bp Api2 $u5
```

The preceding expands to

```
bp Api1 'dd esp 14;g'
```

Note Recursive definitions such as the following are not supported:

```
0:000> r $u0=abc + $u1
0:000> r $u1=def + $u0
```

You can predefine pseudoregisters in your TOOLS.INI file by adding **$u** fields to your [NTSD] entry:

```
[NTSD]
$u0:_ntdll!_RtlRaiseException
$u1:"dd esp 14;g"
$u9:$u0 + $u7
```

Extending NTSD

A debugger extension is an entry point in a DLL called by NTSD. Debugger extensions are called when NTSD sees the following command:

![*module-name***.**] *extension-name* [*extension-arguments*]

If the module name is specified, it is loaded into NTSD using a call to LoadLibrary(module-name). Otherwise, the default extension module is loaded using LoadLibrary("ntsdexts").

After NTSD has loaded the extension library, NTSD calls **GetProcAddress** to locate the extension name in the extension module. The extension name is case sensitive and must be entered exactly as it appears in the extension module's .DEF file. If the extension address is found, the extension is called.

The header file NTSDEXTS.H specifies the extension protocol.

NTSD and KD Reference

This section includes arguments and expressions used with NTSD and KD commands, as well as predefined names used as register and register-flag names. Following these descriptions is an alphabetic list of all NTSD and KD commands.

Note Several commands and command options listed in Commands and Command Arguments and in Command Reference are unique to either NTSD or KD. NTSD does not support multiple processor debugging. KD does not support threads or multiprocess debugging.

Commands and Command Arguments

Any combination of uppercase and lowercase letters may be used in commands and arguments. If a command uses two or more parameters, separate them with a single comma (**,**) or one or more spaces.

Command arguments can be numbers, symbols, line numbers, or expressions. The arguments specify addresses or values used by commands. The arguments and their meanings are listed below:

address
The relative or symbolic address of a variable or function. Flat addresses are prefixed with a percent sign (**%**), when necessary, to distinguish them from addresses in other modes.

byte
A value argument representing a byte value. It must be within the range 0 to 255.

command-string
One or more commands. Multiple commands must be separated by a semicolon (**;**).

expression
A combination of arguments and operators representing a single value or address.

filename
The name of a file or a device. The filename must follow MS-DOS filename conventions.

id
A decimal number representing a breakpoint. The number must be within the range 0 to 31.

d-list
One or more unique decimal numbers representing a list of breakpoint identifiers. The numbers must be within the range 0 to 31. Multiple numbers must be separated by spaces. The wildcard character (*****) can be used to specify all breakpoints.

list
One or more value arguments. The values must be within the range 0 to $2^{(32)} - 1$. Multiple values must be separated by spaces. A list can also be specified as a list of ASCII values. The list can contain any combination of characters and must be enclosed in either single (**'**) or double quotation marks (**"**). If the enclosing mark appears within the list, it must be given twice.

range
A range of addresses. Address ranges have two forms: a starting- and ending-address pair and a start address and object count. The first form is two address arguments, the first specifying the starting address and the second specifying the ending address. The second form consists of an address argument, the letter L, and a value argument. The address specifies the starting address; the value specifies the number of objects to be examined or displayed. The size of an object depends on the command. If a command requires a range but only a start address is given in the command, the command assumes the range to have a default object count as follows:

Size	Object count
ASCII	384
byte	128
word	64
double	32

This default count does not apply to commands that require a range followed immediately by a value or an address argument.

register
The name of a CPU register or a floating point register. The register set is processor dependent. All registers of interest to the application programmer are displayed. A name preceded by an at sign (**@**) always indicates a register name. If no at sign is present, a name is tested to be first a variable,

then a hex number, and finally a register name.

symbol

The address of a variable, function, or segment. A symbol consists of one or more characters, but always begins with a letter, underscore (**_**), question mark (**?**), at sign (**@**), or dollar sign (**$**). Any combination of uppercase and lowercase letters can be used; NTSD and KD symbols are not case sensitive. For some commands, the wildcard character (*****) can be used as part of a symbol to match any combination of characters; you can use a question mark (**?**) to match any single character.

process-id

A process identifier. A vertical bar (**|**) precedes the process identifier, which is one of the following:

Process identifier	Description
.	Current process
*****	All processes
ddd	Decimal number of process ordinal

Processes are assigned ordinals as they are created.

Process specifiers appear as command prefixes. The current process is the process that caused the debug exception. It defines the memory space and the set of threads used. That process remains the current process until a new process is specified by a **|s** command.

Commands that are process-id specific are prefixed by a vertical bar (**|**) and can be followed by a process number or special name as follows:

Process option	Description
.	Current process
#	Process causing present exception
ddd	Decimal number of process ordinal

*prefix***!***symbol*

A symbol name may be prefixed by a DLL name. An exclamation mark (**!**) separates the DLL name from the symbol.

threadspec

A thread specifier. A tilde (**~**) precedes the thread specifier, which is one of the following:

Thread specifier	Description
.	Current thread
#	Thread causing present exception
*****	All threads in the process
ddd	Decimal number of the thread ordinal

Thread specifiers appear as command prefixes. Note that not all wildcards are available in all commands using thread specifiers.

value

Specifies an integer number in octal, decimal, or hexadecimal format.

Symbol Name Resolution

NTSD and KD attempt to match a symbol name in the following order:

1. Exact case match (*sym*)
2. Exact case match with leading underscore (*_sym*)
3. Mixed-case match (*Sym* or *SYM*)
4. Mixed-case match with leading underscore (*_Sym* or *_SYM*)

See also the rules for resolving register, hex, and DLL names listed in the preceding section.

Address Modes and Segment Support

On *x*86 platforms, NTSD and KD support two addressing modes. These modes are distinguished by their prefixes:

Prefix	Name	Mode prompt	Address types
%	flat (also 16:32)	KD>	32-bit addresses (also 16-bit selectors pointing to 32-bit segments – *x*86 only)
&	virtual 86	VMKD>	Real-mode addresses – *x*86 only.

If you access memory through an addressing mode that is not the current default, you can use the address mode prefixes to override the current address mode.

NTSD and KD support only the flat addressing mode on MIPS platforms.

Address Arguments

Address arguments specify the location of variables and functions.

The following table explains the syntax and meaning of the various addresses used in NTSD and KD.

Syntax	Meaning
offset	Absolute 32-bit address in virtual memory space.
&[[segment:]] offset	Real address − *x*86 only.
%segment: [[offset]]	Flat 32-bit address − *x*86 only.
%[[offset]]	Flat 32-bit address.
name[[+|-]] offset	Flat 32-bit address; *name* can be any symbol. The *offset* specifies the offset in bytes. The address can be specified as a positive (+) or negative (-) offset.

Line-Number Arguments

If source line information exists in the .EXE or .DLL file, the debugger can display a specified source line. Line number syntax is:

[*module* **!**[*filename*[**.***expr*]]]

Symbols are as follows:

Symbol	Meaning
module	The name of an executable .EXE or .DLL file. The special name *nt* indicates the kernel. If *module* is not given, the module executing as the current process is the default.
filename	The name of a source file, with no path or extension. If *filename* is not given, the default source file (if available) is that of the module currently executing.
expr	A value in the range from 1 to the highest source-line number in the specified source file. The default radix is 10 regardless of the radix given in the n command.

Line number arguments are translated to the code address designated by the first source line in what may be a multiline source statement. If a line number does not correspond to a source statement, an error occurs. The **v** command is an exception to this rule: it causes the debugger to display the specified source line, which may be a non-executable line such as a comment or blank line.

Expressions

An expression is a combination of arguments and operators that evaluates to an 8-, 16-, or 32-bit value. Expressions can be used as values in any command.

An expression can combine any symbol, number, or address with any of the unary and binary operators in the tables in Unary Operators and Binary Operators. All unary operators have the highest precedence; the precedence of binary operators is indicated by their position in the table.

Unary Operators

Unary address operators assume DS as the default segment for addresses. Expressions are evaluated in order of operator precedence. If adjacent operators have equal precedence, the expression is evaluated from left to right. Use parentheses to override this order.

For symbols with the same character sequence as reserved words (for example, register names and operators), prefix the symbol name with an at sign (**@**) to force symbol access.

The unary operators are listed below:

Operator	**Meaning**
+	Unary plus
-	Unary minus
not	one's complement
hi	High 16-bits
low	Low 16-bits
by	Low-order byte from given address
wo	Low-order word from given address
dw	Double-word from given address
poi	Pointer from given address (same as **dw**)

Binary Operators

The binary operators are listed in the following table in their order of precedence. Operators with higher precedence appear first.

Operator	Meaning
*	Multiplication
/	Integer division
mod (%)	Modulus
+	Addition
-	Subtraction
and	Bitwise Boolean AND
xor	Bitwise Boolean exclusive OR
or	Bitwise Boolean OR

Command Reference

This section describes the NTSD commands. The commands are summarized in NTSD and KD Debugging Quick Reference.

Enter (Repeat Last Command)

Syntax

ENTER

Repeats the last command typed.

| (Process Status)

Syntax

|[**.** | **#** | ***** | *process-id*]

Prints the status of all processes in the system. If a process is specified, then status for the given process only is printed.

~ (Thread Status)

Syntax

~[**.** | **#** | ***** | ddd]

Prints status for threads in the current process. If a thread is specified then status is printed for the given thread only.

(Search for Disassembly Pattern)

Syntax

*pattern address-expression*

Displays the first line of assembly that contains the specified pattern after address.

For example,

```
# strlen main
```

causes the debugger to look for the first reference to the function strlen after the entry point to main.

? (Display Help)

Syntax

?

Displays a list of all NTSD or KD commands and operators.

? (Display Expression)

Syntax

? *expression*

Displays the value of *expression*. The display includes a full address, a 16-bit hexadecimal value, a full 32-bit hexadecimal value, a decimal value (enclosed in parentheses), and a string value (enclosed in double quotation marks). The expression parameter can include any combination of numbers, symbols, addresses, and operators.

$< (Redirect Input)

Syntax

$< *filename*

Redirects input to the specified file until the end of file is reached. When end of file is reached, input is reset to the console.

. (Display Local Variables)

Syntax

.

Displays local variables for the current function. This command is not implemented on MIPS or PowerPC.

a (Assemble)

Syntax

a [*address*]

Assembles instruction mnemonics and places the resulting instruction codes into memory at *address*. If no address is given, the assembly starts at the address given by the current value of the instruction pointer. To assemble a new instruction, type the desired mnemonic and press ENTER. To terminate assembly, press ENTER only.

ba, ~ba (Breakpoint Address)

Syntax

~[[*threadspec*] **ba** [*io*] *option size address* [*value*] [*cmd-string*]

Sets an address breakpoint at a given address. If *threadspec* is given, the breakpoints are set for the specified threads. If your program accesses memory at this address, the debugger stops execution and displays the current values of all registers and flags.

The *option* parameter specifies the type of breakpoint:

Option	Action
E (executable)	Breakpoint trap when the CPU fetches an instruction from the given *address*.
R (read only)	Breakpoint trap when the CPU reads or writes a byte, word, or double-word at the given address.
W (read/write)	Breakpoint trap when the CPU writes a byte, word, or double-word at the given address.

The *io* parameter is an optional value in the range 0-31.

The *address* parameter can specify any valid address. Offsets only are supported.

The *size* parameter is the size in bytes. It can be 1, 2, or 4.

The optional *value* parameter specifies the number of times the breakpoint is to be ignored before being taken. It can be any 16-bit value.

The *cmd-string* parameter specifies an optional list of commands to be executed each time the breakpoint is taken. Each command in the list can include parameters and is separated from the next command by a semicolon (**;**).

The **bc**, **bd**, **be**, and **bl** commands can all be used on these breakpoints.

This command is not implemented on MIPS or PowerPC.

bc (Breakpoint Clear)

Syntax

bc [*id-list*]

Permanently removes one or more breakpoints. If *id-list* is given, the command removes the breakpoints named in the list. The *id-list* can be any combination of integer values from 0 to 31. If the *id-list* is a wildcard character (*****), the command removes all breakpoints.

bd (Breakpoint Disable)

Syntax

bd [*id-list*]

Disables, but does not delete, one or more breakpoints. If *id-list* is given, the command disables the breakpoints named in the list. The *id-list* can be any combination of integer values from 0 to 31. If the *id-list* is a wildcard character (*), the command disables all breakpoints.

be (Breakpoint Enable)

Syntax

be [*id-list*]

Restores one or more breakpoints that were temporarily disabled by a **bd** command. If *id-list* is given, the command enables the breakpoints named in the list. The *id-list* can be any combination of integer values from 0 to 9. If the *id-list* is a wildcard character (*), the command enables all breakpoints.

bl (Breakpoint List)

Syntax

bl

Lists current information about all breakpoints. The command displays the breakpoint number, the enabled status, the address of the breakpoint, the number of passes remaining, and the initial number of passes (in parentheses). The enabled status can be enabled (e) or disabled (d).

bp (Breakpoint Set)

Syntax

~ [**.** | **#** | ddd | *****] [*threadspec*] **bp** [*id*] *address* [*value*] [*command-string*]

Creates a "sticky" breakpoint at the given address. Sticky breakpoints stop execution and display the current values of all registers and flags. Sticky breakpoints remain in the program until they are removed by the **bc** command or temporarily disabled by the **bd** command.

If *threadspec* is given, breakpoints are set on the specified threads. For example, the **~*bp** command sets breakpoints on all threads; **~#bp** sets a breakpoint on the thread causing the current exception; and **~123bp** sets a breakpoint on thread 123. The **~bp** and **~.bp** commands both set a breakpoint on the current thread.

The debugger supports up to 32 breakpoints, which are numbered 0 through 31. The optional *id* parameter specifies the number of the breakpoint to be created. If *id* is not given, the first available breakpoint number is used.

The *address* parameter can be any valid instruction address (it must be the first byte of an instruction).

The optional *value* parameter specifies the number of times the breakpoint is to be ignored before being taken. It can be any 16-bit value.

The optional *command-string* parameter specifies a list of commands to be executed each time the breakpoint is taken. This command-string is executed only if the breakpoint is from a g command (and not from a **t** or **p** command). Debugger commands in the list can include parameters; the commands are separated by semicolons (**;**).

c (Compare)

Syntax

c [*range*] *address*

Compares the bytes in the memory locations specified by *range* with the corresponding bytes in the memory locations beginning at *address*. If all corresponding bytes match, the debugger displays its prompt and waits for the next command. If one or more pairs of corresponding bytes do not match, the debugger displays each pair of mismatched bytes.

d (Dump)

Syntax

d [*range*]

Displays the contents of memory in *range*. The command displays data in the same format as the most recent dump command (**d**, **da**, **db**, **dd**, **dg**, **dh**, **dl**, **dq**, **dt**, or **dw**).

If no range is given and no previous dump command has been used, the debugger displays bytes starting at the current instruction pointer.

da (Dump ASCII)

Syntax

da [*range*]

Displays the ASCII characters in the given *range*. Each line displays up to 48 characters. The display continues until the first null byte or until all characters in *range* have been displayed. Nonprintable characters, such as carriage returns and line feeds, are displayed as periods (**.**).

db (Dump Bytes)

Syntax

db [*range*]

Displays the hexadecimal and ASCII values of the bytes in the given *range*. Each display line shows the address of the first byte in the line, followed by up to 16 hexadecimal byte values. The byte values are immediately followed by the corresponding ASCII values. The eighth and ninth hexadecimal values are separated by a hyphen (**-**). Nonprintable ASCII values are displayed as periods (**.**).

dd (Dump Double-Words)

Syntax

dd [*range*]

Displays the hexadecimal values of the double-words (4-byte values) in the given *range*. Each display line shows the address of the first double-word in the line and up to four hexadecimal double-word values.

dw (Dump Words)

Syntax

dw [*range*]

Displays the hexadecimal values of the words (2-byte values) in the given *range*. Each display line shows the address of the first word in the line and up to eight hexadecimal word values.

e (Enter)

Syntax

e *address* [*list*]

Enters one or more values into memory. The size of the value entered depends on the most recently used **Enter** command (**e**, **ea**, **eb**, **ed**, **el**, **es**, **et**, or **ew**). The default is **eb** (bytes). If no list is given, the command displays the value at *address* and prompts for a new value. If *list* is given, the debugger replaces the value at *address* and at each subsequent address until all values in the list have been used.

ea (Enter ASCII)

Syntax

ea *address* [*list*]

Enters an ASCII string into memory. If *list* is not given, the debugger displays the byte at *address* and prompts for a replacement. If *list* is given, the debugger replaces the bytes at *address*, then displays the next byte and prompts for a replacement.

eb (Enter Bytes)

Syntax

eb *address* [*list*]

Enters one or more byte values into memory. If *list* is given, the debugger replaces the byte at *address* and at each subsequent address until all values in the list have been used. If *list* is not given, the debugger displays the byte at *address* and prompts for a new value. To skip to the next byte, enter a new value or press SPACE. To move back to the previous byte, type a hyphen (**-**). To exit from the command, press ENTER.

ed (Enter Double-Words)

Syntax

ed *address* [*list*]

Enters a double-word value into memory. If *list* is not given, the debugger displays the double-word at *address* and prompts for a replacement. If *list* is given, the debugger replaces the double-word at *address*, then displays the next double-word and prompts for a replacement. Double-words must be entered as two words separated by a colon (**:**).

ew (Enter Words)

Syntax

ew *address* [*list*]

Enters a word value into memory. If *list* is not given, the debugger displays the word at *address* and prompts for a replacement. If *list* is given, the debugger replaces the word at *address*, then displays the next word and prompts for a replacement.

f (Fill)

Syntax

f *range list*

Fills the addresses in *range* with the values in *list*. If *range* specifies more bytes than the number of values in the list, the list is repeated until all bytes in the range are filled. If *list* has more values than the number of bytes in the range, the debugger ignores the extra values.

~f (Freeze Thread)

Syntax

~ [**.** | **#** | ***** | ddd] **f**

Freezes the current thread. The **~*f** command freezes all threads; **~#f** freezes the thread causing the current exception; and **~123f**, for example, freezes thread 123.

g (Go)

Syntax

[*threadspec*] **g** [**=** *startaddress*] [*breakaddress*] **...**

Passes execution control to the program at *startaddress*. Execution continues to the end of the program or until *breakaddress* is encountered. The program also stops at any breakpoints set using the bp command.

If *startaddress* is not given, the debugger passes execution to the address specified by the current value of the instruction pointer. If *breakaddress* is given, it must specify an instruction address (that is, the address must contain the first byte of an instruction). Up to 10 break addresses, in any order, can be given at one time.

If **~g** is given then the go command is executed with the specified thread unfrozen, and all others frozen. For example, if the command **~123g**, **~#g**, or **~*g** is given, the specified threads are unfrozen and all others are frozen.

gh, gn (Go for Exceptions)

Syntax

~[*threadspec*] **gh** [**=** *startaddress*] [*breakaddress*] **...**

~[*threadspec*] **gn** [**=** *startaddress*] [*breakaddress*] **...**

Control exception handling. These commands can be used after the first break to an exception. The **gh** command treats the exception as handled; **gn** treats the exception as not handled. See the **sx** command for more details.

ib, iw, id (Input from Port)

Syntax

i [**b** | **w** | **d**] *address*

Inputs and displays a byte, word, or double word (**b**, **w**, or **d**) from the port at the location given by *address* (KD only).

.inputfile (Read from Command File)

Syntax

.inputfile *filename*

Reads commands from *filename* until the end of file is reached or another .inputfile command is executed. Input files are not recursive and cannot be nested: a .inputfile command in a log file causes an immediate jump to the new log file with no return.

j (Conditional Execution)

Syntax

j *expression* ['']*command1*['] **;** ['']*command2*['']

Allows conditional execution of commands. If *expression* evaluates to a nonzero value, the string *command1* is executed, else the string *command2* is executed. The optional single quotes are used to group multiple commands (separated by semicolons) into a single command string. This command can be combined with the bp command to create conditional breakpoints:

```
bp _foo "j (eax>0x1) 'dd app!bar;g;'u;|' "
```

~k (Backtrace Stack)

Syntax

~ [*threadspec*] **k** [*value*]

[*processor-num*] **k** [*value*]

Displays the current stack frame. Each line shows the name of a procedure, its arguments, and the address of the statement that called it. If *value* is given, the command displays that many stack frames.

Using the **k** command at the beginning of a function (before the function prolog has been executed) will give incorrect results. The debugger uses the frame register to compute the current backtrace, and this register is not correctly set for a function until its prolog has been executed.

If a thread specifier is given, a backtrace for the given thread is printed.

With the kernel debugger, the process number can be specified to indicate the stack frame for a given process. Default is 1.

ln (List Near)

Syntax

ln *address*

Lists the nearest symbols before and after *address*.

.logappend (Append to Log File)

Syntax

.logappend [*file*]

Appends history information to the current log file. The default filenames are NTSD.LOG and KD.LOG, for NTSD.EXE and SDK.EXE, respectively.

.logclose (Close Log File)

Syntax

.logclose

Closes the current log file.

.logopen (Open Log File)

Syntax

.logopen [*file*]

Opens the log file and deletes any existing file. The default filenames are NTSD.LOG and KD.LOG for NTSD.EXE and SDK.EXE, respectively.

Any current log file is closed.

m (Move)

Syntax

m *range address*

Moves the block of memory specified by *range* to the location starting at *address*. All moves are guaranteed to be performed without data loss.

ob, ow, od (Output to Port)

Syntax

o [**b** | **w** | **d**] *address value*

Outputs a byte, word, or double word (b, w, or d) *value* to the port at *address* (KD only).

p (Program Step)

Syntax

p [**r**] [**=** *startaddress*] [*repeat*]

Executes an instruction, then displays the current values of all registers and flags.

The **r** flag toggles display of register values.

If *startaddress* is given, the debugger starts execution at the given address. Otherwise, the debugger starts execution at the instruction pointed to by the instruction pointer.

If *repeat* is given, the debugger executes *repeat* number of instructions before stopping. The debugger automatically executes and returns from any call instructions or software interrupts it encounters, leaving execution control at the next instruction after the call or interrupt.

With **s -** and **s&**, the **p** command steps one machine instruction; with **s+**, the **p** command steps one source line.

~p (Program Thread Step)

Syntax

~ [**.** | **#** | ***** | ddd] **p** [**r**] [**=** *startaddress*] [*repeat*]

Steps execution with indicated threads thawed; all others are frozen. For example, **~*p** steps with all threads thawed; **~#p** steps with only the current exception thread unfrozen; and **~123p**, steps with thread 123 thawed and all others frozen. The **~p** and **~.p** commands both step execution with the current thread unfrozen.

q (Quit)

Syntax

q

Terminates NTSD execution and returns control to the operating system.

r, rf (Register Display)

Syntax

~ [**.** | **#** | ddd | *****] **r** [*register*] [= [*value*]]

~ [**.** | **#** | ddd | *****] **rf** [*floatregister*] [= [*value*]]

Display the contents of CPU registers and allow their contents to be changed to new values.

If *register* is not specified, the debugger displays all registers, all flags, and the instruction at the address pointed to by the instruction pointer. If *register* is specified, the debugger displays the current value of the register and prompts for a new value. If both *register* and *value* are specified, the debugger changes the register contents to the specified value.

With the kernel debugger, a processor number can be specified to indicate the registers for a given processor. The default is 0.

With the user-mode debugger, a thread specifier can be specified to display registers for a given thread.

The **rf** command displays floating-point registers. Note that floating-point registers may be operated on in the same way as other CPU registers.

rt (Register Display Toggle)

Syntax

[*threadspec*] **rt**

Toggles the register display (KD only). The default is off.

With the kernel debugger, a processor number can be specified to indicate the registers for a given processor. The default is 0.

.reboot (Restart Target Computer)

Syntax

.reboot

Restarts the target computer (KD only, not implemented on MIPS or PowerPC). This command exits the debugger, so that open symbol files are closed.

!reload

Syntax

!reload

Loads symbols for all drivers that have been loaded on the target system. This command is useful in the event of a system crash which may result in the loss of symbols for the target computer being debugged. To load the correct symbols, invoke KD with an explicit path to the kernel, for example:

```
i386kd c:\nt\oskrnl.exe
```

Then you can type

```
!reload
```

and the debugger will load the symbols for the drivers that the target system has already loaded.

s (Search)

Syntax

s *range list*

Searches the given *range* of memory locations for the byte values given in *list*. The debugger displays the address of each byte found.

ss (Set Suffix)

Syntax

ss [**n | a | w**]

Outputs the current state of the suffix value. Suffix values may be:

Suffix	**Meaning**
n	No suffix
a	ASCII
w	Wide characters

For the a and w suffix values, any symbol that is undefined will have the suffix value appended to its name, and the resulting symbol is searched for in the symbol table.

sx, sxe, sxd (Set Exceptions)

Syntax

sx [**e** | **d**] [*eventcode* | *exceptnum*]

Control the behavior of the debugger when exceptions occur (NTSD only).

The **sx** command causes the debugger to display its break status for all events, including exceptions.

Followed by one of the two-letter event codes listed below or by an exception number, the **sx** command causes the debugger to print its break status for the given event. Non-exception events are identified by a two-letter combination:

Event code	Meaning
av	Access Violation
cc	CTRL+C
ct	Create Thread
et	Exit Thread
ep	Exit Process
ld	Load DLL

The **sx** command also can be used to set the debugger's behavior on an event. The **sxe** command enables the break status for an exception; the **sxd** command disables the break status of an exception.

On a break on an event, only the **g** command (not **t** or **p**) is allowed to continue execution.

On a first-time break to an exception, special versions of the **go** command can be used: **gh** treats the exception as handled, and **gu** treats the exception as unhandled. A normal **g** command uses the default disposition.

Second time exceptions always break into the debugger.

s+, s&, s - (Set Source Display Mode)

Syntax

s+
s&
s -

Set the display mode for commands that display instruction code:

Command	Action
s -	Disassemble and display the instruction code in memory
s&	Display the source line and the disassembled code
s+	Display the source line corresponding to the instruction to be displayed

Source lines are identified by *module.lineno*, where *module* is the source filename (without a path or extension) and *lineno* is the line number in the source file.

|s (Set Process)

Syntax

| [**.** | **#** | *process-id*] **s**

Sets the current process number. If *process-id* is not given, information for the current process is printed.

~s (Set Thread)

Syntax

~ [**.** | **#** | ddd] s

Sets the current thread number. If no thread number is given, information for the current thread is printed.

t (Trace)

Syntax

t [**r**] [**=** *startaddress*] [*repeat*]

Executes an instruction, then displays the current values of all registers and flags.

The **r** flag toggles display of register values.

If *startaddress* is given, the debugger starts execution at the given address. Otherwise, it starts execution at the instruction pointed to by the instruction pointer.

If *repeat* is given, the debugger continues to execute *repeat* instructions before stopping.

In source-only mode (**s+**), **t** operates directly on source lines. The **t** command can be used to trace instructions in ROM.

The **t** command with a repetition count outputs instructions for all repetitions.

~t (Trace Thread)

Syntax

~ [**.** | **#** | ***** | ddd] **p** [**t**] [**=** *startaddress*] [*repeat*]

Traces with indicated threads thawed; all others are frozen. For example, **~*t** traces with all threads thawed; **~#t** traces with only the current exception thread thawed; and **~123t**, traces with thread 123 thawed, and all others frozen. The **~t** and **~.t** commands both trace with the current thread thawed.

u (Unassemble)

Syntax

u [*range*]

Displays the instructions and/or statements of the program being debugged. The **s+**, **s -** , and **s&** commands set the display format.

If *range* is given, the debugger displays instructions generated from code within the given range. Otherwise, the debugger displays the instructions generated from the first eight lines of code at the current address.

The 80286 protected-mode mnemonics cannot be displayed.

~u (Unfreeze)

~ [**.** | **#** | ***** | ddd] **u**

Unfreezes the current thread. The **~*u** command unfreezes all threads; **~#u** unfreezes the thread causing the current exception; and **~123u**, for example, unfreezes thread 123.

v (View Source Lines)

Syntax

v *range*

Displays source lines beginning at the specified *range*. The symbol file must contain line-number information.

x (Examine Symbols)

Syntax

x *symbol*

Displays all symbols matching *symbol*. The pattern match is case sensitive. The ? character matches any character and the ***** character matches zero or more characters.

The **x *** form of the command displays module information. The **x *!*** command displays module and line number information, when available, for all modules. The first asterisk is a wildcard for module names and the second is a wildcard for line number expressions.

NTSD and KD Debugging Quick Reference

The following table lists the command syntax for NTSD and KD debugging commands:

NTSD and KD Debugging Commands

Syntax	Meaning
ENTER	Repeat last command
| [. | # | * | *process-id*]	Process status
~ [. | # | * | ddd]	Thread status
# *pattern address=expression*	Search for dissassembly pattern
?	Display debugger help
? *expression*	Compute and display *expression*
$< *filename*	Redirect input from *filename*
.	Display local variables – not implemented on MIPS or PowerPC
a [*address*]	Assemble instruction mnemonics
~ [*threadspec*] **ba** [*io*] *option size address* [*value*] [*cmd-string*]	Set address breakpoint (KD only – not implemented on MIPS or PowerPC)
bc [*id-list*]	Clear breakpoint(s)
bd [*id-list*]	Disable breakpoint(s)
be [*id-list*]	Enable breakpoint(s)
bl	List breakpoint(s)
~ [. | # | ddd | *] [*threadspec*] **bp** [*id*] *address* [*value*] [*command-string*]	Set breakpoints
c [*range*] *address*	Compare
d [*range*]	Dump memory using previous type
da [*range*]	Dump memory ASCII
db [*range*]	Dump memory bytes
dd [*range*]	Dump memory double-words
dw [*range*]	Dump memory words
e *address* [*list*]	Enter using previous type
ea *address* [*list*]	Enter ASCII
eb *address* [*list*]	Enter bytes
ed *address* [*list*]	Enter double-words
ew *address* [*list*]	Enter words
f *range list*	Fill range
~ [. | # | * | ddd] f	Freeze thread
g, **gh**, and **gn**	Go commands
i [b | w | d] *address*	Input from port at *address* (KD only)
.inputfile *filename*	Read from command file
j *expression* [']*command1*['] **;** [']*command2*[']	Conditional execution
~ [*threadspec*] **k** [*value*]	Backtrace stack
ln *address*	Lists first symbols before and after

	address
.logappend [*file*]	Append to the current log file
.logclose	Close the current log file
.logopen [*file*]	Open a log file
m *range address*	Move
o [b | w | d] *address value*	Output *value* to port at *address* (KD only)
[**.** | **#** | ***** | ddd] **p** [r] [**=** *startaddress*] [*repeat*]	Execute an instruction, then display registers and flags
~ [*threadspec*] **p** [= *startaddress*] [*value*]	Trace program instruction or call
q	Quit
[*threadspec*] **r** [t] [*register*] [= [*value*]]	Register set (**rt** toggles full register display – KD only)
[*threadspec*] **rf** [*floatregister*] [= [*value*]]	Register set command for floating-point values
.reboot	Restart target computer (KD only – not implemented on MIPS or PowerPC)
!reload	Load symbols for all the drivers that have been loaded on the target system.
s *range* **list**	Search *range* for bytes in *list*
ss [n | a | w]	Set suffix to n for none, a for ASCII, or w for wide. With no argument, prints current suffix setting.
sx [e | d] [*eventcode* | *exceptnum*]	Set exceptions (NTSD only)
s+	Set source display only
s&	Set mixed assembler and source display mode
s -	Set assembler display only
| [. | # | *process-id*] s	Set process
~ [. | # | ddd] s	Set thread
t [r] [= *startaddress*] [*repeat*]	Trace program instruction
~ [. | # | * | ddd] **p** [t] [= *startaddress*] [*repeat*]	Trace thread
u [*range*]	Display unassembled instructions
~ [. | # | * | ddd] **u**	Unfreeze thread
v *range*	View source lines
x *symbol*	Examine symbol(s)

Debugging Programs

The WinDbg debugger lets you analyze the internal workings of your program by using the debugger to set breakpoints and examine variables. In this section, you use the GENERIC program to learn debugging strategies and techniques within WinDbg.

WinDbg supports debugging of EXEs and DLLs for Windows NT. This section describes how to debug an .EXE project. If you want to debug a .DLL project, you must create an appropriate .EXE shell program to call the DLL. Load the calling program from the command line or with the Open command from the Program Menu, then run the EXE with the Go command.

Using Debugging Information Windows

WinDbg displays information in a series of windows that you can view as you debug a program. You activate these windows with commands from the Window menu.

These are the windows and information they show:

Window	Information
Watch	Monitor expressions set with the Watch Expression command.
Locals	Values of local variables within the function currently being stepped through.
Registers	Current contents of the CPU registers.
Disassembly	Assembly-language instructions being debugged.
Command	Allows you to enter debugger commands with the keyboard. These commands are described in Command Window Interface.
Floating-Point Registers	Current contents of floating-point registers and stack.
Memory	Current contents of memory. Display format controlled by Memory Options dialog box.
Calls	Displays the current call stack and lets you jump to any stack frame displayed.

You can size and minimize these windows during debugging so that you can see various types of information at one time.

Using the WinDbg Debugger

WinDbg lets you set breakpoints in the source code; view variables, memory, and registers; and control program execution.

Preparing a Debug Version of a Program

Before a program can be debugged, you must include debugging information with the project.

{ewl msdncd, EWGraphic, group10547 0 /a "SDK.BMP"} To build the GENERIC project for debugging

1. Open a command-line session with the Windows NT Program Manager.
2. Move to the directory containing the GENERIC example program.
3. Build the executable by typing

   ```
   NMAKE /A
   ```

4. Load the executable into WinDbg by typing

   ```
   WINDBG GENERIC.EXE
   ```

Note If there are errors during the build, make sure that the build tools are in your path and that the INCLUDE and LIB environment variables point to the directories containing the system .H and .LIB files respectively.

Setting and Removing Breakpoints

Breakpoints are useful when you have a general idea of where a bug occurs in a program. WinDbg runs until it reaches a breakpoint, then stops. You can then step to the next line of code or trace through a function until you find the problem.

{ewl msdncd, EWGraphic, group10547 1 /a "SDK.BMP"} To set a breakpoint in the GENERIC program

1. Move the insertion point to the first line of the **About** function in the source code.
2. From the Debug menu, choose Breakpoints. The Breakpoints dialog box appears. (See the figure below.)
3. From the Break list, select at Location.
4. Choose Add.
5. Choose OK.

{ewc msdncd, EWGraphic, group10547 2 /a "SDK.bmp"}

6. Choose Go from the Run menu. The GENERIC program begins to run.
7. While the program is running, choose About Generic from its Help menu. The program stops and the breakpoint line is highlighted in the source window.

{ewl msdncd, EWGraphic, group10547 3 /a "SDK.BMP"} To remove a breakpoint

1. From the Debug menu, choose Breakpoints. The Breakpoints dialog box appears. (See the figure above.)
2. Select the breakpoint to remove from the Breakpoints list.
3. Choose Delete.
4. Choose OK.

The Breakpoints dialog box offers a variety of options for setting breakpoints. You can set breakpoints based on Windows messages, memory location, and the values in an expression. See the WinDbg online help for details.

Controlling Program Execution

Once a breakpoint is reached and the program stops, you can control execution with commands on the Run menu. The following list shows the commands and their actions.

Command	Action
Restart	Resets execution to the first line of the program. It reloads the program into memory and discards the current values of all variables.
Stop Debugging	Terminates the debugging session and returns to a normal editing session.
Go	Executes code from the current statement until a breakpoint or the end of the program is reached.
Continue to Cursor	Executes the program as far as the line that currently has the cursor. This is equivalent to setting a temporary breakpoint at the cursor location.
Trace Into	Steps into a function when it is called and steps through all of the instructions in the function.
Step Over	Single-steps through instructions in the program. If this command is used when you reach a function call, the function is executed without stepping through the function instructions.
Set Thread	Selects thread to monitor with the debugger. This determines which thread receives debugger commands and is viewed by the display windows. Also freezes and thaws individual threads.
Set Process	Selects process to monitor with the debugger. This determines which process receives debugger commands and is viewed by the display windows.
Source/Asm Mode	Toggles between source and assembly modes.

Using Watch Expressions

The Watch Expressions command shows a variable's name and value in the Watch window. It updates the window whenever the variable's value changes. This feature is useful during debugging to check the value of a variable. You can enter expressions as well as variables.

{ewl msdncd, EWGraphic, group10547 4 /a "SDK.BMP"} To add a watch variable

1. In the GENERIC program, select the *message* variable in the **About** function.
2. From the Debug menu, choose Watch Expression. The selected *message* variable appears in the Watch Expression text box.
3. Choose Add.
4. Choose OK. The Watch window opens, and it includes the *message* variable.
5. Resize the Watch window so it is visible with the source window displayed.
6. With the breakpoint in the GENERIC program still in place, choose Go from the Run menu. The *message* variable is updated to show its new value.

{ewl msdncd, EWGraphic, group10547 5 /a "SDK.BMP"} To remove the *message* variable from the Watch Expression list

1. From the Debug menu, choose Watch Expression.
2. Select Message from the Watch Expression list.
3. Choose Delete.
4. Choose Close.

Using Quickwatch

Quickwatch gives you a fast way to view variables and expressions. Unlike Watch Expressions, which are stored in the Watch window, the values of Quickwatch variables and expressions appear only in the Quickwatch dialog box (see the figure below.) This is useful in exploratory debugging where you are checking a number of variables that are suspect.

{ewl msdncd, EWGraphic, group10547 6 /a "SDK.BMP"} To use Quickwatch

1. With the breakpoint in the About procedure, run the GENERIC program, open the Help menu of the Generic Sample Application, and select the About Generic command.
2. Move the insertion point in the source file to the *message* variable in the About function.
3. From the Debug menu, choose Quickwatch. You can also use the Quickwatch button on the Toolbar.
4. Choose Cancel to close the Quickwatch dialog box.

{ewc msdncd, EWGraphic, group10547 7 /a "SDK.bmp"}

The Quickwatch dialog box also lists commands that you can use to evaluate expressions, add variables and expressions to the Watch Expression list, contract and expand structures, and modify values.

Using the Command Window

With the Command window, you can use the keyboard instead of the mouse to operate the debugger using CodeView-style commands.

See the WinDbg online help or Command Window Interface for a detailed list of the commands available in the Command window.

{ewl msdncd, EWGraphic, group10547 8 /a "SDK.BMP"} To debug a program from the Command Window

1. Open the Command window by choosing Command from the Window menu.
2. Reset the program counter by choosing Restart from the Run Menu.
3. Type **bl** in the command window to list breakpoints. You should see the breakpoint in the About procedure.
4. Type **g** to start program execution. After the program begins, open the Help menu of the Generic Sample Application, and select the About Generic command.
5. Type **dc** to display the assembly language instructions following the breakpoint.

Postmortem Debugging with WinDbg

When a program crashes (for example, after a general-protection fault), Windows NT automatically invokes a debugger that is specified in the registry. To make WinDbg be that debugger, use REGEDT32.EXE to edit the following registry key:

HKEY_LOCAL_MACHINE
SOFTWARE
Microsoft
Windows NT
CurrentVersion
AeDebug

Add or edit the Debugger value entry (REG_SZ). Give it the following string value:

```
windbg -p %ld -e %ld
```

You can also use the Attach command from the Run menu to attach the debugger to a process that is already running.

Remote Debugging

Remote debugging lets you debug a program from another computer. You run WinDbg on one computer (the *host*) and install the program you are debugging on another (the *target*). You run a remote monitor program (WinDbgRm) on the target computer to control the program you are debugging. WinDbgRm communicates with WinDbg across a serial cable or across the network.

Remote debugging isolates WinDbg from the program being debugged so that errors in the program do not affect the debugger, and the debugger does not affect the target system. If the program crashes the remote system, your development system continues to run.

The remote monitor demands fewer system resources, and after starting and loading the program to be debugged, it does not use the file system. Therefore, the monitor has no effect on the resources that can change your program's behavior.

You can debug large programs or programs that destabilize the operating system. You can also debug programs on smaller systems that cannot support the full debugger. Some bugs that you cannot reproduce while running under the full debugger can appear under the remote monitor. Remote debugging also allows the debugging of Win32 programs on Win32s™, since WinDbg will not run on Win32s.

The process of debugging a program on a remote computer is almost the same as for local debugging. The only difference is in how you start the session. The following sections describe the hardware and files required for remote debugging and how to configure the debugger components on the host and target computers.

Requirements

Remote debugging requires two computers, the host running Microsoft Windows NT and the target running under Windows NT or Win32s. Serial debugging requires a serial port connecting the host and target computers (this is the only connection allowed when debugging Win32s applications). Debugging through a named pipe requires that both computers be connected to a common network that is supported by Windows NT. Network debugging is faster and does not require a special cable between the two systems.

To remotely debug using the serial ports, connect the host and target computers with a null-modem cable plugged into the serial ports on the two computers. A null modem is a serial cable that connects the transmitting line at each end to the receiving line at the opposite end. You cannot use an extension cable with "straight-through" connections.

Most computer stores stock or can assemble a null-modem cable for you with the correct wiring and the appropriate connectors for your host and target computers.

Starting a Remote Debugging Session

After the WinDbg components are in their locations and properly configured, you can begin a remote debugging session.

{ewl msdncd, EWGraphic, group10547 9 /a "SDK.BMP"} To start a remote debugging session

1. If you are debugging across a serial cable, connect the cable to both the target and host.
2. On the target computer:
   - Install your program and its DLLs. You do not need to have the source files on the target.
   - Start WinDbgRm.
   - WinDbgRm starts minimized as an icon. Its default is to communicate with WinDbg across the network, using the named pipe "windbg". If you want to change how the host and target communicate, double-click on the WinDbgRm icon and select the Transport DLL command from the Options menu. Choose the appropriate transport layer in the Known Transport Layers list box.
3. On the host computer:
   - Start WinDbg from the Windows NT command prompt:

     `start windbg` *`yourprog`*`.exe`

   - Use the Debugger DLLs command from the Options menu to change the transport layer to match the one used by WinDbgRm on the target. If you are using named pipes, use the Change button to modify the transport layer parameters so that the first parameter is the name of the target computer.
   - Select the Execution Model and Expression Evaluator to reflect the target computer.
   - Use the User DLLs command from the Options menu to specify the symbol search path to your program's symbols; use the Debug command from the Options menu to specify the source search path to your source files.
   - Select Go from the Run menu.

If you can start your program on the target before attaching the debugger to it, then you can change step 3 as follows:

On the host computer:

- Start WinDbg.
- Use the Debugger DLLs command from the Options menu to change the transport layer to match the one used by WinDbgRm on the target. If you are using named pipes, use the Change button to modify the transport layer parameters so that the first parameter is the name of the target computer.
- Select the Execution Model and Expression Evaluator to reflect the target computer.
- Use the User DLLs command from the Options menu to specify the symbol search path to your program's symbols; use the Debug command from the Options menu to specify the source search path to your source files.
- Select the Attach command from the Run menu.
- Select your program from the task list that the Attach command displays.

To connect to a computer with an IPC password, use the Windows NT NET USE command to manually connect to the remote IPC$ share.

Use the .disconnect command to disconnect the host from the target. Use the .attach command to reestablish the connection. Another computer can run WinDbg and use .attach to connect to the target and debug your program. If you are still connected and another computer tries to connect to the target, you will receive a popup notifying you of that fact. You have 20 seconds to cancel the pop-up, then you will be disconnected and the other computer will be connected as the host of the remote debugging

session.

If problems occur, check connections and make sure that WinDbg and WinDbgRm are using the same communications methods. If your system has trouble maintaining the communications link between the host and target computers, when communicating through a serial port, reduce the transmission baud rate.

Creating Dialog Boxes

The Dialog Editor automates the process of creating dialog boxes. You can see your dialogs as you design them without writing and compiling your own resource scripts.

The Dialog Editor works much like a paint program. You design dialogs interactively, placing and moving graphical elements right on the screen. When you complete your design, the Dialog Editor automatically creates the files you need to incorporate the dialogs in your program.

This section explains how to create dialog boxes and include them in an application. You can use the Dialog Editor's online Help to get more information on specific features.

Note A dialog box that has not been compiled and exists only in resource script form is referred to as a "dialog."

Using the Dialog Editor

The following topics describe how to design, test, and include dialogs in your application.

Note The Dialog Editor requires a mouse or equivalent pointing device.

Creating a New Dialog

{ewl msdncd, EWGraphic, group10548 0 /a "SDK.BMP"} To create your application's dialog

1. Start the Dialog Editor from its group icon in the Program Manager.
2. From the File menu, choose New.

The Properties Bar, the rectangular area immediately below the Dialog Editor's Menu Bar, displays the screen coordinates of the currently selected dialog or control. When a dialog is selected, as it is after you first choose New from the File menu, there are also three edit fields on the Properties Bar:

- Dlg. Sym., for your symbolic dialog name
- Caption, for your dialog title
- A dialog ID field

The small solid rectangles around the dialog border are called *resize handles.* They show that the dialog is currently selected.

You can use the mouse to resize the dialog with the resize handles, and you can move the dialog with the arrow keys or the mouse. To move the dialog with the mouse, click and drag its title bar or border. The coordinates in the fields on the left side of the Properties Bar change as you move the dialog.

At times it may be useful to design the dialog in the graphical context in which it will be used. For example, you may wish to see how the dialog box looks against the backdrop of the application (the way users of your application will see it).

{ewl msdncd, EWGraphic, group10548 1 /a "SDK.BMP"} To design dialogs using your application as a backdrop

1. Start your application.
2. Start the Dialog Editor.
3. Move the dialog you are designing outside the Dialog Editor window.
4. Reduce the Dialog Editor window in size and move it to a corner of the display.

You can design the dialog box on the screen where it will eventually appear.

Adding a Control

The Toolbox on the right side of the screen contains icons of the mouse pointer and all supplied controls. The mouse pointer, when pressed, indicates that no control has been selected. You can place any of the other controls in a dialog, and you can also create and include your own custom controls.

{ewl msdncd, EWGraphic, group10548 2 /a "SDK.BMP"} To add a control to the dialog

1. In the Toolbox, click one of the control icons.
2. Click in the dialog.

You can use the mouse to resize a control with its resize handle. You can move a control with the cursor keys when it is selected or by dragging it with the mouse.

When a control is selected or created, the Properties Bar displays information about it. The Symbol edit field replaces the Dlg. Sym. edit field, and the Text edit field replaces the Caption edit field. The numeric control ID edit field appears for controls just as for dialogs.

You can associate a symbolic name with the control's ID by typing the name in the Symbol edit field. The Dialog Editor assigns each new control in a dialog an ID number. It assigns these numbers sequentially, beginning with the number following the dialog ID. Even if you assigned it a zero-terminated string and deleted the integer ID.

Symbolic Names and ID Numbers

You select the new dialog by clicking anywhere outside a control in the dialog window. The Properties Bar then displays the dialog's default title, "Dialog Title," and the default ID, which is 100 in this case.

Each dialog box in an application needs a unique integer ID. You use the ID in your application source code to tell Windows which dialog box to call, and Windows uses the ID to search the resource file for the dialog box.

You can associate a symbolic name with the ID by typing the name in the Dlg. Sym. edit field. It's easier to refer to a dialog by a name than by a number. Alternatively you can name a dialog with a zero-terminated string, although this is not as efficient.

{ewl msdncd, EWGraphic, group10548 3 /a "SDK.BMP"} To enter a zero-terminated string as a dialog designator

1. Type the string in the Dlg. Sym. edit field.
2. Immediately tab to the field on the right (the Dialog ID edit field).
3. Delete the integer ID.
4. Press ENTER.

You can give your dialog a title by typing text in the Caption edit field. You can use the TAB key to move the insertion point. From the Dlg. Sym. edit field, the insertion point first tabs to the Dlg. Sym. ID edit field and then to the Caption edit field. You can also reposition the insertion point by clicking the mouse pointer in the edit field you want to use.

When you select or create a control, the Symbol edit field replaces the Dlg. Sym. edit field. You can associate a symbolic name with the control's ID by typing the name in the Symbol edit field, or by choosing it from the list of available symbols (if you have any).

The IDOK, IDCANCEL, and UNUSED Symbols

Because most dialog boxes have OK and Cancel buttons, Windows provides defined symbols for these elements.

You use the defined symbol IDOK for an OK button in your dialog. When the user presses ENTER, Windows sends your dialog procedure a message as if the OK button had been pressed. IDOK has an ID value of 1.

You use IDCANCEL for a Cancel button in your dialog. When the user presses ESC, it's as if the Cancel button had been pressed. IDCANCEL has an ID value of 2.

Some controls don't need a unique identifier. For example, you won't usually need to access a group box frame, which simply frames other controls. You can assign all such controls the symbol "(Unused)," which has an ID value of -1.

You can use the Symbols command on the Edit menu to display and edit the current symbols. The Symbols dialog box displays a list box of symbols and corresponding IDs. You can add, change, and delete symbols in the Symbols dialog box.

When you remove a control from a dialog, the symbol and ID for that control remain defined but not in use. You can display unused symbols by checking the Show Unused Symbols Only check box in the Symbols dialog box. You can then delete these unused symbols if you like.

The Text field on the Properties bar contains the button's default text, "Push."

{ewl msdncd, EWGraphic, group10548 4 /a "SDK.BMP"} To assign other text to the button

1. Type the text for the push button in the Text edit field.
2. Press ENTER.

For example, you might assign the symbol IDOK to your first push button and give it the text OK, and then press ENTER.

You can use the Edit menu Size to Text command to make the width of a selected control better fit its text. However, you cannot size a push button smaller than a default-sized push button. A push button always automatically maintains a margin around its text. The following controls can be sized to their text:

- Check box
- Push button
- Radio button
- Text
- Custom controls written to support this feature

{ewl msdncd, EWGraphic, group10548 5 /a "SDK.BMP"} To size a control to its text

1. Select the control.
2. From the Edit menu choose the Size to Text command, or press F7.

You can size a single control to fit its text, or you can select a group of controls and size them all at once. If a group of controls is selected, only those controls that support sizing to fit their text will be changed.

Adding More Controls

Dialog boxes often include more than one control – push buttons, list boxes, and check boxes are common. This section shows you how to add controls and how to edit and organize these controls.

Duplicating a Control

You will probably want to have more than one push button in your dialog.

{ewl msdncd, EWGraphic, group10548 6 /a "SDK.BMP"} To add another push button

1. Hold down the CTRL key.
2. Drag the existing push button to a new location and drop it.

You might give this second button the symbol IDCANCEL and the label Cancel.

You can repeat this process to create more push buttons. Or you can use the CTRL key when you select controls from the Toolbox. If you hold down the CTRL key when you click a Toolbox button, the Toolbox button stays down. You can then click repeatedly in the dialog and place a number of identical controls. When you have all the controls you need of this kind, click the top control in the Toolbox, or press ESC when the Toolbox has the focus.

Adding a List Box

Now add a list box to the dialog, clicking its icon in the Toolbox (fourth row, second column) and dropping it into the dialog box. A list box is a box of text strings with a scrollbar on its right side.

The list box appears only as an outline at this stage. You'll see more of the list box in a moment, when you test the dialog.

Adding Custom Controls

When a standard control – such as a list box or edit field – doesn't meet your needs, you can create a custom control that performs a specific task. For example, the buttons on a toolbar can be custom controls.

The code that defines a custom control resides in either a DLL or, occasionally, in the .EXE file that calls the dialog box. For information about creating custom controls, see the *Microsoft Windows 3 Developer's Workshop*, by Richard Wilton.

The Dialog Editor supports custom-control DLLs and temporary custom controls, as described in the following table:

Custom Control	Description
DLL	A dynamic-link library (DLL) that defines one or more custom control classes.
Temporary	A place holder for a custom control defined elsewhere. The Dialog Editor creates a gray rectangle to indicate the control's location and size. You can set the temporary control's class name, size, and other attributes to match the ones used by the final custom control.

Once opened by the Dialog Editor, custom-control DLLs remain available between Dialog Editor sessions. Temporary custom controls are not saved between sessions.

When the Dialog Editor opens a .DLG file that contains a custom control, it searches its list of custom-control DLLs to find a match. If a match is found (the controls have the same class name), the Dialog Editor uses the information in the DLL to display the custom control. If a match is not found, the Dialog Editor creates a temporary custom control with the given class name.

The Dialog Editor replaces any temporary custom control during an editing session when you open a DLL custom control with the same name.

{ewl msdncd, EWGraphic, group10548 7 /a "SDK.BMP"} To add a new temporary custom control

1. From the File menu, choose New Custom. The New Temporary Custom Control dialog box appears.
2. Type a name for the custom control in the Class Name edit field.
3. Optionally, change the other Control Defaults.
4. Choose OK.

The Dialog Editor adds the temporary control to the list of available custom controls. To use the temporary control, click the custom control tool. You can use the following method to add either a temporary custom control or a custom control DLL to a dialog box:

{ewl msdncd, EWGraphic, group10548 8 /a "SDK.BMP"} To add a custom control

1. Click the Custom Control (bottom right) tool on the Toolbox.

   If you have more than one custom control installed (either temporary or from DLLs), the Select Custom Control dialog box appears. If you only have one custom control, skip to Step 6.
2. From the Available Controls list box, select the custom control you want to add to the dialog box.
3. Choose OK.
4. Move the cursor into the dialog. A rectangle with the "move" icon inside replaces the cursor.
5. Press and hold the mouse button to see an outline of the control. The coordinates for the location of the control are displayed in the Properties Bar.

   You cannot move the outlined control outside of the dialog once you have pressed the mouse

button.

6. Drop the control by releasing the mouse button. The custom control is centered over the point where you released the mouse button.

Editing Groups of Controls

You'll often find it useful to edit an entire group of controls. You can drag, cut, copy, paste, align, and size a group of controls.

First, enlarge the OK button by selecting it and then dragging one of its corner handles with the mouse. Now select all three buttons by dragging open a rectangle with the mouse. To drag a rectangle open, place the mouse pointer beside a control, then press and hold the left mouse button, and drag the mouse. In this way you can select all the controls that are within or partially within the rectangle boundaries.

Another way to select several controls at a time is to hold down the SHIFT key and click on each control. This is useful for making a group out of a number of scattered controls. You can also toggle the selection state of each control without affecting other controls by clicking on that control while holding down the SHIFT key.

The Anchor Control

In a selected group of controls, one of the controls has solid resize handles, while the others have outlined handles. The button with the solid handles is called the *anchor control.* The anchor serves as a guide for various group alignment and sizing operations. You can change the current anchor in a selected group by clicking any control that is part of the group. Now make the third (unnamed) push button the anchor by clicking it.

Sizing and Aligning a Group

{ewl msdncd, EWGraphic, group10548 9 /a "SDK.BMP"}

To make all the selected buttons the same size as the anchor button

1. From the Arrange menu, choose the Same Size command.
2. Click the top option to make the buttons the same width.
3. Choose the Same Size command again.
4. Click the bottom option to make the buttons the same height.

All the buttons become the same size as the anchor button.

The two Push Button command options on the Arrange menu control the alignment of push buttons. One centers buttons across the bottom of the dialog box, and the other arranges them down the right side. Use the proper command option to arrange the buttons down the right side.

Ordering Controls

You've now built a dialog with three push buttons and a list box. At this stage in the design, you'll often refine the order of controls, that is, the order in which the user will move through them with the TAB key.

When the user of your dialog box presses the TAB key, the focus moves through the controls in order, stopping at each one for which a tab stop is set. Some controls, such as a frame, have no need for a focus. Controls that can use the focus have a tab stop set by default. Each control in your dialog already has a tab stop.

Changing Control Tabbing Order

To see the current tabbing order of your controls, choose the Order/Group command from the Arrange menu. This brings up the Order/Group dialog box. All controls in your new dialog are listed in the list box in their current tabbing order, which is the order in which you created them.

If this dialog box were opened in an application, by default the focus would begin with the first control you created, the OK button. The user would probably prefer that the focus begin at the list box.

{ewl msdncd, EWGraphic, group10548 10 /a "SDK.BMP"} To change the tabbing order by moving the list box control to the start of the list

1. Select the list box control from the list in the Order/Group dialog box by clicking its title once with the mouse.
2. Move the mouse pointer to the start of the list. The mouse pointer changes shape at a valid insertion point.
3. Click to reposition the list box.

Making Groups

You can also use the Order/Group command to group controls in your dialog box. This allows the user to cycle the focus through the members of a group using the arrow keys after moving to a group member with the TAB key.

To be grouped together, controls must be contiguous in the Order/Group list box. Hence you should define groups only after you have the tab order set as you want it.

{ewl msdncd, EWGraphic, group10548 11 /a "SDK.BMP"} To make and order a group of controls

1. From the Arrange menu, choose Order/Group. This brings up the Order/Group dialog box.
2. In the control list box, click the first control you want in a particular group.
3. Drag the mouse up or down to select the other controls in the group.
4. Choose Make Group.

Groups are shown by lines separating the control names in the list box.

If you have more than one logical group of radio buttons within a dialog, you should group them here so that Windows will handle their logic properly.

Changing Control Size and Placement

You can also use other commands on the Arrange menu to size and place groups of controls. Using commands on the Arrange menu you can:

- Align controls along an edge (Align command)
- Distribute controls evenly with respect to an anchor control (Even Spacing command)
- Set controls to the same size (Same Size command)
- Distribute push buttons evenly along the bottom or side of the dialog (Push Buttons command)
- Change grid and spacing values for editing controls (Settings command)

You might want to experiment with these commands in the dialog you have just created. Click the Arranging icon in Help for more information about these commands.

Testing Your Dialog

To see how the controls in your dialog work, choose the Test Mode command from the Options menu. Test Mode creates a functioning dialog box for testing. The list box becomes operational with a scrollbar and lines of sample text.

The Dialog Editor inserts lines of sample text. You cannot insert lines of your own text directly into the Dialog Editor. You can scroll the sample text, select it, and see how many lines and characters the box can display, and you can test the changes you make using the Order/Group command.

You can also use Test Mode to investigate the various style options you can apply to controls and to the dialog box itself (see Dialog and Control Styles).

To quit Test mode, choose the Test Mode command from the Options menu, or press ALT+F4.

Dialog and Control Styles

Behind each control and the dialog itself are some associated attributes called *styles.* Styles govern everything from the way a dialog or control appears to the way it behaves. So far, you've just used the default styles.

Double-click the list box to open its Styles dialog box. You can also open the Styles dialog box for the currently selected dialog box or control by choosing the Styles command from the Edit menu.

As you can see from the Styles dialog, you have many choices for dialog styles. Test Mode gives you a hands-on way of exploring style. For example, turn on the list box's Extended Selection style, and then close its Styles dialog box. Return to Test Mode. A default list box lets you select only one item from its list, but now you can select a range of items from the list box by dragging down the list with the mouse.

Designing Multiple Dialogs for an Application

You've now designed one dialog, but your application may need many. You can work on more than one dialog at a time in the Dialog Editor.

Your first dialog was automatically started for you when you selected New from the File menu. To begin work on a second dialog, select New Dialog from the Edit menu. Your first dialog still exists within the editor. Use the Select Dialog command from the Edit menu to switch between the dialogs.

You can create as many dialogs as you need for your application. Each dialog is automatically numbered in sequence (100, 200, and so on). You can use the Restore Dialog command to restore a dialog to the state it was in before you selected it for editing. This allows you to abandon your most recent changes.

You can continue creating more dialogs for your application using the New Dialog command. When you are finished creating the new dialog, you can save it with your other dialogs by choosing the same base file name. This is how you keep all the dialogs together for your application. The Dialog Editor uses the same group of files to keep track of all dialogs that you save together. See the following section for a description of the files the Dialog Editor writes when it saves a group of dialogs.

You can save each dialog you create separately. But if you save them together, the Dialog Editor can avoid conflicts when it assigns ID values, and it can also alert you to conflicts when you choose symbolic names and IDs. Also, for each dialog you save separately, you must place three include statements in your project source code. For the typical application, this might amount to about 100 include statements. When you save an application's dialogs together, however, you need only three include statements for your entire project.

Note If you choose New from the File menu and then save with the same base file name as you did before, your previously-saved dialogs are overwritten.

Including Dialogs in an Application

When you save a dialog or dialogs, the editor normally creates three files:

- The .RES (resource) file, a resource file dedicated to the Dialog Editor. This .RES file is read only by the Dialog Editor and is not the same as the .RES file that is produced by the Resource Compiler and linked into your application.
- The .DLG (dialog) script, a text file containing **DIALOG** and **CONTROL** statements that the Resource Compiler interprets.
- The .H (include) file, a text file containing **#define** statements associated with the symbolic names of controls in dialogs. (If you haven't defined any symbolic names in your dialogs, the editor won't create an .H file.)

The .DLG and .H files are used to integrate the current dialogs into your application.

By default, the editor uses the same base name for all three files. Thus, if you use the editor's suggested default name of DIALOGS, the editor writes DIALOGS.RES, DIALOGS.DLG, and DIALOGS.H.

{ewl msdncd, EWGraphic, group10548 12 /a "SDK.BMP"} To include dialogs in your application

1. Use **#include** statements to include DIALOGS.H and DIALOGS.DLG in your application's resource script (.RC) file. Place the DIALOGS.H statement before the DIALOGS.DLG statement.

   ```
   #include "dialogs.h"
   #include "dialogs.dlg"
   ```

2. Include DIALOGS.H in your source code.
3. Write a dialog procedure for each dialog to initialize controls and process their messages. Place each dialog procedure name in the module definition file.
4. Compile your source code.
5. Use the Resource Compiler to compile the .RC file into the application's .RES file.
6. Link the .RES file into your application's .EXE file.

The DIALOGS.RES File

The DIALOGS.RES and DIALOGS.H files are created by the Dialog Editor. Together they are an archive of all the dialog information. The .RES file contains all the dialog information except the symbolic names you've defined, which are in the .H file.

Don't confuse the .RES file written by the Dialog Editor with the .RES file produced by the Resource Compiler that is linked into your application. The .RES file that the Dialog Editor writes is just an archive for the Dialog Editor's use.

Note You must avoid name conflicts between the two .RES files. Failing to avoid name conflicts will cause unpredictable results.

The DIALOGS.DLG File

When you save a dialog, one of the files the Dialog Editor writes is the .DLG file. This text file contains the information in the .RES file expressed as resource script statements. Each dialog is described in a sequence of statements called a "template."

When you create a dialog in the editor, it has a unique ID number, which becomes a handle to the template. When you want to create an instance of the dialog in your application, you pass this handle to Windows in a call to the **DialogBox** function. Windows does the rest. Here's a typical .DLG script file containing two simple dialogs:

```
100 DIALOG 6, 16, 148, 84
STYLE DS_MODALFRAME | WS_VISIBLE | WS_CAPTION | NOT WS_BORDER | WS_POPUP
CAPTION "Invert"
BEGIN
    PUSHBUTTON      "Push This", 101, 54, 30, 40, 14,
END

200 DIALOG 6, 16, 148, 84
STYLE DS_MODALFRAME | WS_VISIBLE | WS_CAPTION | NOT WS_BORDER | WS_POPUP
CAPTION "Reverse"
BEGIN
    PUSHBUTTON      "Push That", 101, 43, 30, 40, 14,
END
```

The handles to the two dialogs are 100 and 200. You can also define a corresponding symbolic name for a dialog in the editor. It's easier to use a name than a number in your code.

The Dialog Editor cannot read the DIALOGS.DLG file. The data is saved and read from the .RES file.

The DIALOGS.H File

If you define symbolic names in your dialogs, the Dialog Editor writes them to a .H file.

When you create a dialog control, it gets an ID number. You can also assign a symbolic name to the control, and can refer to the control by name or number in your code. The .H file associates symbolic names with corresponding ID numbers.

A typical .H file looks like this:

```
#define MYOPTION1  101
#define MYOPTION2  102
```

You can then refer to controls in this way:

```
BOOL FAR PASCAL FirstDlgProc(hDlg, wMessage, wParam, lParam)
 .
 .
 .
case WM_COMMAND:
     switch (LOWORD(wParam))
     {
        case IDOK :
           /* Respond to Ok button here */
                     break;
        case IDCANCEL :
           /* Respond to Cancel button here */
                     break;
        case MYOPTION1:
           /* Respond to MyOption1 button here */
                     break;
        case MYOPTION2:
           /* Respond to MyOption2 button here */
                     break;

 .
 .
 .
```

Remember that IDOK and IDCANCEL are defined by Windows and should be used for the OK and Cancel buttons in your dialog box.

Modifying a Dialog

Because they are text files, you may be tempted to modify the .DLG file and the .H file directly. However, these two files are interdependent, and if you make a change to one, you risk creating a conflict with the other. Also, changes you make in the .DLG file won't become a permanent part of a dialog because the .DLG file is only written by the Dialog Editor, not read. The best way to modify a dialog is to load it into the editor and make the changes there.

Using the Editor for Existing Projects

If you begin using the Dialog Editor in an existing project, you may initially find the structure of the project awkward. The project's dialogs may have been written by hand, and the resulting resource script statements may be in an .RC file along with all the other resources for the application. The project's dialog symbols may be in an .H file along with numerous other of the application's symbols. If you don't restructure the project, you'll have to merge files by hand each time you create a dialog.

The easiest way to proceed is to read the application's .RES file with the Dialog Editor by loading it into the editor and at the same time retrieve the dialog symbols by opening the appropriate .H file. You can then save all the dialog information in the Dialog Editor's own dedicated .RES, .DLG, and .H files.

If you save the dialogs under the name DIALOGS, you can then use DIALOGS.DLG as a guide for removing the appropriate resource script statements from the .RC file and substituting a DIALOGS.DLG include statement. In the same way, remove the dialog symbols from the original .H file and include DIALOGS.H in your application and .RC file.

You could continue to let the dialog symbols reside in the original .H file by using the Set Include command on the File menu to associate that file with the dialogs. However, it will be easier for you if you take the time to put the dialog symbols in their own file. Then if you later want to archive the dialogs, examine the dialog files, or move the dialogs to another environment, all the information will be readily available to you.

Using the Editor

Use the WinDbg integrated text editor to edit source files. Writing text with the WinDbg editor is similar to using most other editors. For example, to start a new line, press ENTER; to leave a blank line between lines, press ENTER twice; If you make a mistake while typing, press BACKSPACE to delete the error.

This section provides information on how to edit source files. Most of the procedures, such as file and text handling and moving around in a file, follow basic Windows editing conventions.

Managing Source Files

This section explains how to create, save, open, and close a file. To familiarize yourself with the steps, you can use the sample program below. You can also use this program to try the other editing features described in this section.

```
/* first sample program */

#include <stdio.h>

main()
{
     printf("Windows NT console I/O is easy!\n");
}
```

Opening and Closing Source Files

{ewl msdncd, EWGraphic, group10549 0 /a "SDK.BMP"} To create a new source file

1. Choose New from the File menu.
2. Type your program into the new window.

New files are labeled "UNTITLED *n*" until they are saved.

Note Before you can save or close a window it must be active. To make a window active, either click on the window or select the window name or number from the Window menu.

{ewl msdncd, EWGraphic, group10549 1 /a "SDK.BMP"} To open an existing source file

1. From the File menu, choose Open.
2. If the file is on a network share, click the Network button to select the share.
3. Choose a drive from the Drives list box. The default drive is the current drive.
4. From the Directories list box, double-click a directory where the file is stored (or down a directory path to the directory where the file is stored.)
5. Choose a file type in the List Files of Type box. Only files with the chosen extension(s) are displayed in the File Name box.
6. Double-click a filename, or click a filename and choose OK.

The names of the four most recently opened files are displayed at the end of the File menu. To open one of these files, choose its name from the menu.

Because the editor works with any ASCII file, you can open and edit text files created with other editors.

When a source file is opened, its name is added to the Window menu. If you want to see more than one view of your source file, use the New Window command on the Window menu to open another view. A file can be opened only once; if you try to open a file that is already open, the file's window will activate, but the file will not be reloaded. You can, however, open multiple windows showing the same source file, so that you can, for instance, compare two different sections of the file. If more than one copy of the file is displayed, the window is titled *filename*:*x*, where *x* is a unique window number.

{ewl msdncd, EWGraphic, group10549 2 /a "SDK.BMP"} To save a source file

1. Activate the source window.
2. From the File menu, choose Save.
3. If your file is unnamed, WinDbg displays the Save As dialog box. Enter the file's name. If you are using the sample program, name the file FIRST.C.
4. Select the drive and the directory to save the file to by choosing items from the Drives and Directories list boxes.
5. Choose OK.

If the file has already been named, the Save command saves any changes without displaying the Save As dialog box.

The title bar will display an asterisk (*****) when the source file has been modified.

{ewl msdncd, EWGraphic, group10549 3 /a "SDK.BMP"} To save a source file under a different name

1. Activate the source window.
2. From the File menu, choose Save As.
3. Type a new name for the file.

4. Choose OK.

This is useful for maintaining revised copies of a source file while keeping the original unchanged. You can also use the Save As command to name and save a newly created file.

{ewl msdncd, EWGraphic, group10549 4 /a "SDK.BMP"} To save all open source files that are new or have been changed

- Choose Save All from the File menu.

{ewl msdncd, EWGraphic, group10549 5 /a "SDK.BMP"} To close a source file

1. Activate the source window.
2. From the File menu, choose Close.

If you are using the sample program, use one of these techniques to close the FIRST.C source window.

If you create a new source file or modify an existing file and try to close it before saving, a message appears asking if you want to save the changes before closing.

{ewl msdncd, EWGraphic, group10549 6 /a "SDK.BMP"} To activate a source window

- Click the title bar.

  - Or -

- Select it from the Window menu.

  - Or -

- Use the shortcut key listed on the title bar (ALT + # for windows 0 - 9, ALT + CTRL + # for windows 10 - 19).

  - Or -

- Use CTRL + F6 to cycle through all active windows.

{ewl msdncd, EWGraphic, group10549 7 /a "SDK.BMP"} To close a source window

1. Activate the source window.
2. Double-click the document Control-menu box.

   - or -

   Press CTRL + F4.

Merging Source Files

The Merge command inserts another file at the insertion point. You can also use the command to replace selected text within the active source window with the contents of another file.

{ewl msdncd, EWGraphic, group10549 8 /a "SDK.BMP"}

To merge source files

1. Move the insertion point where you want to merge text, or select the text you want to replace.
2. From the File menu, choose Merge. The Merge dialog box appears.
3. Select the drive and directory where the source file to be inserted is stored. The default is the current drive and directory.
4. Select the type of file to display.
5. Type a filename or select the name from the list of files.
6. Choose OK to merge the specified file at the insertion point.

As an alternative you can open a file with the editor and copy and paste text to another file.

Moving Around Source Files

This section discusses some special techniques for moving around source files in WinDbg.

{ewl msdncd, EWGraphic, group10549 9 /a "SDK.BMP"} To move to a specific line within the source window

1. From the View menu, choose Goto Line. The Line dialog box appears.
2. Enter the line number you wish to move to.
3. Choose OK.

{ewl msdncd, EWGraphic, group10549 10 /a "SDK.BMP"} To set a tag in the source window

1. Move the insertion point to the line where you want to set a tag.
2. From the View menu, choose Toggle Tag. The tagged line is highlighted.

You can tag frequently accessed lines in your source file and use menu commands to move quickly to them. Tags can be cleared when you no longer need them.

{ewl msdncd, EWGraphic, group10549 11 /a "SDK.BMP"} To clear a tag

1. Move the insertion point to the tagged line.
2. From the View menu, choose Toggle Tag. The tag and the highlight are removed.

To remove all tags in a window, choose Clear All Tags from the View menu.

{ewl msdncd, EWGraphic, group10549 12 /a "SDK.BMP"} To move to the next tag after the insertion point

- From the View menu, choose Next Tag, or press F2.

{ewl msdncd, EWGraphic, group10549 13 /a "SDK.BMP"} To move to the previous tag before the insertion point

- From the View menu, choose Previous Tag, or press SHIFT + F2.

Using the Keyboard Commands

The editor includes a number of special keystrokes for editing and moving around in a source file. These are in addition to the familiar ARROW keys, SPACEBAR, and ENTER.

To move to	Press
Word to left	CTRL+LEFT ARROW
Word to right	CTRL+RIGHT ARROW
Beginning of current line	HOME
End of current line	END
Beginning of file	CTRL+HOME
End of file	CTRL+END

See online Help for the complete set of WinDbg keyboard commands.

Controlling the Source Window

The editor features a number of options that affect the source window. These include the ability to set tabs, highlight language syntax within the window, and make the file read-only.

Setting Tabs

You can set tab stops in a source file and save them as tabs or spaces when you save the file.

{ewl msdncd, EWGraphic, group10549 14 /a "SDK.BMP"} To set tabs

1. Choose Environment from the Options menu. The Environment dialog box appears.
2. Enter the number of spaces to be used as a tab stop in the Tab Stops text box.
3. Choose the Keep Tabs option to treat tabs as a single tab character when the source file is saved.
   - or -
   Choose Insert Spaces to convert tabs to the number of spaces shown in the Tab Stops box.
4. Choose OK.

Highlighting Language Syntax

WinDbg highlights language keywords, identifiers, comments, and strings in different colors. This feature is useful when learning a language or when viewing lengthy and complex source files. For example, if you are editing a file with a .C extension, all C keywords are highlighted.

Making a File Read-Only

The Read Only command makes the active source window read-only. When you choose this command, the window cannot be altered.

This command is useful when you are viewing a program and don't want to accidentally make any changes to the window.

{ewl msdncd, EWGraphic, group10549 15 /a "SDK.BMP"}

To make a source window read-only

1. Activate the window.
2. From the Edit menu, choose Read Only.

To deactivate the command, choose it again.

A check mark next to the Read Only menu item and the word READ in the status bar at the bottom of the screen indicate that the source file is read-only.

Finding and Replacing

The WinDbg editor offers advanced find and replace capabilities. You can search for literal text or use regular expressions to find words or characters in the source window.

The Find command searches for specified text. The Replace command finds the text and replaces it with other text.

Finding Text

{ewl msdncd, EWGraphic, group10549 16 /a "SDK.BMP"} To find a character, word, or group of characters in the active source window

1. Position the insertion point where you want to start the search.
2. From the Edit menu, choose Find. The Find dialog box appears.
3. In the Find What box, type the text you want to find, or choose from the 16 previous instances of text searched for, which are listed in the drop-down box.
4. Choose any of the Find options, as described below.
5. Choose OK.

When you choose the Find command, the editor uses the location of the insertion point to select a default search string. If the insertion point is inside a word, that word is displayed as the search text in the Find What box.

If the insertion point is between words, the word to the right is displayed. If there is no word to the right, the word to the left is displayed. If that is not possible, nothing is displayed.

The Find dialog box has these choices for locating text:

Command	Description
Match Whole Word Only	Searches for complete occurrences of the text. A word contains only the characters a - z and A - Z, the numbers 0 - 9, and the # or _ (underscore) characters. If you don't select this option, the editor finds embedded occurrences; for example, "main" in "remainder".
Match Case	Searches for text that matches the capitalization of the search text.
Regular Expression	Finds text using patterns. See the WinDbg help file for details on regular expressions.
Direction	The Up option searches from the insertion point to the beginning of the file. The Down option searches from the insertion point to the end of the file.
Tag All	Searches for text and marks each line that contains the text. Use the Tag commands from the View menu to move to each marked line.

If the text is located, the source window scrolls to the text and highlights it. Choose Find Next from the abbreviated Find dialog box to move to the next occurrence of the text.

Replacing Text

{ewl msdncd, EWGraphic, group10549 17 /a "SDK.BMP"} To find and replace text

1. From the Edit menu, choose Replace. The Replace dialog box appears.
2. In the Find What box, type the search text or regular expression, or choose from the 16 previous instances of text searched for, which are listed in the drop-down box.
3. In the Replace With box, type the replacement text, or choose from the 16 previous instances of text searched for, which are listed in the drop-down box.
4. Set any change options you need. These options are the same as those in the Find dialog box (see the previous section, "Finding Text").
5. Choose OK.

For information on default search strings, see the previous section, "Finding Text."

{ewl msdncd, EWGraphic, group10549 18 /a "SDK.BMP"} To repeat the last search or search and replace

- Choose Find Next in the abbreviated Find or Replace dialog box. The abbreviated dialog box remains active until the search reaches the starting location (after wrapping) or until you choose Cancel.

Introducing WinDbg

The WinDbg debugger is a powerful, graphically based tool that allows you to debug programs and edit source code under Windows NT. Its user interface is similar to Microsoft QuickC for Windows 1.0, but the debugger has been significantly enhanced to allow more flexible debugging and to provide full support for the NT operating system.

You can use WinDbg to debug kernel-mode drivers.

This introduction explains how to install WinDbg and briefly introduces the WinDbg debugger. It describes starting and quitting WinDbg, using WinDbg windows, and customizing WinDbg.

Installing WinDbg

The WinDbg files are copied with other development files from the NT SDK CD-ROM. By default, the files are held in the MSTOOLS\BIN directory. You can use the SDK SETUP program to install WinDbg.

Starting and Quitting WinDbg

After you have installed WinDbg, start it by double-clicking its icon on the Program Manager. You can also start WinDbg from the Windows NT command line by typing

windbg [-a] [-g] [-h] [-i] [-k[*platform port speed*]] [-l[*text*]] [-m] [-p *id* [-e *event*]] [-s[*pipe*]] [-v] [-w *name*] [-y *path*] [-z *crashfile*] [*filename*[.*ext*] [*arguments*]]

The following table describes the command-line options.

WinDbg Command-Line Options

Option	Description
-a	Ignore all bad symbols (but still print warning message).
-g	Go now; start executing the process.
-h	Causes child processes to inherit access to WinDbg's handles.
-i	Ignore workspace; like running without any registry data.
-k [*platform port speed*]	Run as a kernel debugger with the specified options: *platform* is the target computer type (i386, mips, alpha) *port* is the com port (com1 ... com*n*) *speed* is the com port speed (9600, 19200, 57600, ...)
-l [*text*]	Sets the window title for WinDbg.
-m	Start WinDbg minimized.
-p *id*	Attach to the process with the given id.
-e *event*	Signal an event after process is attached. Used only for post-mortem debugging (AeDebug).
-s [*pipe*]	Start a remote.EXE server, using the named pipe.
-v	Verbose option; WinDbg prints module load and unload messages.
-w *name*	Load the named workspace.
-y *path*	Search for symbols along the specified path. You can specify multiple paths by separating them with semicolons.
-z *crashfile*	Debug the specified crash dump file.
filename[.*ext*]	Program to debug or file to edit. If no extension is specified, WinDbg assumes an .EXE extension. If the file is not an .EXE or .COM file, WinDbg will try to load it as a text file.
arguments	Arguments to program being debugged.

To quit WinDbg, choose Exit from the File menu. This returns you to the Windows NT Program Manager.

WinDbg Windows

WinDbg uses windows to present and enter information. These windows are like standard windows in other applications for Windows.

The Toolbar

The Toolbar appears beneath the menu bar (see the figure below). It provides shortcuts for menu selections.

{ewc msdncd, EWGraphic, group10550 0 /a "SDKb.bmp"}

You can hide or display the Toolbar with the Toolbar command on the View menu. When this command is checked, the Toolbar appears. When it is turned off (unchecked), the Toolbar is hidden. You turn the command on or off by selecting it.

To issue debug commands, click one of the icons on the Toolbar:

Icon	Action
{ewc ï¿½}	Starts or continues execution. Execution proceeds until the program reaches a breakpoint or the program ends. All threads and processes are unfrozen. Equivalent to the Go command from the Run menu.
{ewc ï¿½}	Stops the program and its threads, allowing you to regain control of the debugger. Equivalent to the Halt command from the Run menu.
{ewc ï¿½}	Sets a breakpoint at the current line.
{ewc ï¿½}	Displays the Quickwatch debugging dialog box. Equivalent to the Quickwatch command from the Debug menu.
{ewc ï¿½}	Steps into a function when it is called and steps through all of the instructions in the function. Equivalent to the Trace Into command from the Run menu.
{ewc ï¿½}	Single-steps through instructions in the program. If this command is used when you reach a function call, the function is executed without stepping through the function instructions. Equivalent to the Step Over command from the Run menu.
{ewc ï¿½}	Switches between source and assembly mode. Equivalent to switching modes with the Toggle Source/Asm Mode command from the Run menu.
{ewc ï¿½}	Switches between source and assembly mode. Equivalent to switching modes with the Toggle Source/Asm Mode command from the Run menu.
{ewc ï¿½}	Opens the Options dialog, if any, for the current window.

When a button cannot be used (for example, a trace command while the debuggee is running), the button is grayed.

Font selection is not available from the toolbar. You can change the typeface of the current source file by choosing Font from the Options menu.

The Status Bar

A status bar (see the figure below) appears at the bottom of the main WinDbg window. The status bar provides information about WinDbg and the active source window, such as line and column position and editor mode. When debugging, the status bar will also give the WinDbg process ID (PID) and thread ID (TID) of the code being debugged.

The status bar displays the following information:

Contents	Description
Message box	Displays messages from the environment.
SRC/ASM	Indicates whether WinDbg is in source or assembly mode.
pid	Shows the ID number of the process being debugged.
tid	Shows the ID number of the thread being debugged.
^Q	Indicates a WordStar keystroke sequence.
OVR	Indicates overtype mode.
READ	Indicates read-only status.
CAPS	Indicates that CAPS LOCK is on.
NUM	Indicates that NUM LOCK is on.
Line	Displays the line number at the insertion point in the current window.
Column	Displays the column number at the insertion point in the current window.

{ewc msdncd, EWGraphic, group10550 10 /a "SDKa.bmp"}

You can hide or display the status bar with the Status Bar command on the View menu. When this command is checked, the status bar appears. When it is turned off (unchecked), the status bar is hidden. Turn the command on or off by selecting it.

Customizing WinDbg

Many parts of WinDbg can be customized to suit your programming needs. This section discusses how you can modify the appearance and functionality of WinDbg.

Using the Program Registry

To start debugging a program from within WinDbg, you must first add it to the program registry (or use the **.ATTACH** command). The WinDbg program registry is a list of programs that WinDbg can debug. Each program in the registry has one or more workspaces associated with it. A workspace is a set of WinDbg defaults that can be associated with a program.

{ewl msdncd, EWGraphic, group10550 11 /a "SDK.BMP"} To add a program to the registry

1. Choose Open from the Program menu.
   The Program Open dialog will appear with a list of all of the programs in the registry.
2. Choose New.
   The common Open dialog will appear.
3. Open the executable file that you want to debug, and choose OK.
   The dialog boxes will close. WinDbg is now ready to begin debugging.

{ewl msdncd, EWGraphic, group10550 12 /a "SDK.BMP"} To load a program already in the registry

1. Choose Open from the Program menu.
   The Program Open dialog will appear with a list of all of the programs in the registry.
2. Select the program to debug from the Programs list.
3. Select the workspace to use from the Workspaces list.
   For more information on workspaces, see the following section, "Using Workspaces."
4. Choose OK.

{ewl msdncd, EWGraphic, group10550 13 /a "SDK.BMP"} To delete a program

1. Choose Delete from the Program menu.
2. In the Program box, select the program that you want to delete.
3. Choose Delete. The selected program will disappear.
4. Choose OK.

Using Workspaces

With workspaces, you can save and restore the state of WinDbg between debugging sessions and customize WinDbg for different debugging situations. Workspaces are saved in the Windows NT registry for each user.

When you debug a program for the first time or start WinDbg without loading a debuggee, you start with the Common Workspace. The Common Workspace is different from other workspaces because it is not associated with a particular program, but is available at all times. You can use the Common Workspace as a starting point for custom workspaces that are saved with each program.

You can change the Common Workspace to reflect your preferences for WinDbg.

{ewl msdncd, EWGraphic, group10550 14 /a "SDK.BMP"} To change the Common Workspace

1. Configure WinDbg windows and options.
2. Choose Save Common from the Program menu.

Note Since the Common Workspace does not correspond to a specific program, breakpoints, watch expressions, and other program-specific information are not saved.

{ewl msdncd, EWGraphic, group10550 15 /a "SDK.BMP"} To create a new workspace

1. Load a debuggee using Open from the Program menu.
2. Configure WinDbg windows, breakpoints, and options.
3. Choose Save As from the Program menu.
4. Enter a name for the workspace.

   If you check the Make Default box, the workspace will be saved with the given name, and it will be made the default workspace for the program when the Program Open Dialog box is next used.
5. Choose OK.

{ewl msdncd, EWGraphic, group10550 16 /a "SDK.BMP"} To delete a workspace

1. Choose Delete from the Program menu.
2. In the Program box, select the program that has the workspace that you want to delete.
3. In the Workspace box, select the workspaces that you want to delete.
4. Choose Delete. The selected workspaces will disappear.
5. Choose OK.

Changing Display Colors

Using different colors for various language elements such as functions and variables gives you immediate visual cues about the structure of your source code. You can change the default colors of these elements as well as the color of other text in WinDbg, such as reserved words, breakpoints, and information windows.

{ewl msdncd, EWGraphic, group10550 17 /a "SDK.BMP"} To change the colors in the source window

1. From the Options menu, choose Color. The Color dialog box appears.
2. From the Items list select the item you want to change. For this example, select Source Window.
3. Select the color square from the palette to represent the foreground color. Press Set Foreground. The new color combination is shown in the Items box.
4. Select the color square to represent the background color from the palette shown. Press Set Background. The new color combination is shown in the Items box.
5. Choose OK to apply the change to the source window. You can change several items in the list before choosing OK.

{ewl msdncd, EWGraphic, group10550 18 /a "SDK.BMP"} To change the source window back to its original colors

1. From the Options menu, choose Color.
2. From the Items list, select Source Window.
3. Choose Set Default.
4. Choose OK.

{ewl msdncd, EWGraphic, group10550 19 /a "SDK.BMP"} To change all windows back to their default colors

1. From the Options menu, choose Color.
2. Choose Select All.
3. Choose Set Default.
4. Choose OK.

Setting Font Type and Size

You can specify which font type and size appear in WinDbg source windows. You can choose any font type and size found within the Windows NT system.

{ewl msdncd, EWGraphic, group10550 20 /a "SDK.BMP"} To set the font type and size

1. From the Options menu, choose Font. The Font dialog box appears.
2. Select the font, font style, and size from the combo boxes. The words "sample text" will change to the font you selected.
3. Choose OK.

The Make Default box in the Font dialog box sets the selected font and size as the default.

Text within the source window can be only one font and size. Multiple fonts cannot be displayed in the same source window.

Compiling Messages

The Microsoft Windows Message Compiler (MC) is a tool for the Microsoft Windows NT operating system. This section describes how to create a message text file, and how to compile it for inclusion in a resource script.

About the Message Compiler

The Message Compiler (MC.EXE) converts message text files into binary files suitable for inclusion in a resource script (.RC file). The Resource Compiler is then used to place the messages into a resource for inclusion in an application or DLL.

Applications that perform event logging typically use an independent resource-only DLL that contains the messages, rather than carry the messages in the application image.

Using the Message Compiler

The Message Compiler utility has the following command-line syntax:

Syntax

MC [**-v**] [**-w**] [**-s**] [**-d**] [**-h** *dir*] [**-e** *extension*] [**-r** *dir*] *filename*[**.MC**]...

Parameters

-v
Generates verbose output to **stderr**.

-w
Generates a warning message whenever an insert escape sequence is seen that is a superset of the type supported by the OS/2 MKMSGF utility. These are any escape sequences other than **%0** and **%***n*. This option is useful for converting MKMSGF message files to MC format.

-s
Adds an extra line to the beginning of each message that is the symbolic name associated with the message identifier.

-d
Ouputs **Severity** and **Facility** constants in decimal. Sets the initial output radix for messages to decimal.

-h *dirs*
Specifies the target directory of the generated include file. The include-file name is the base name of the .MC file with a .H extension.

-e *extension*
Specifies the extension for the header file, which can be from one to three characters. The default is .H.

-r *dir*
Specifies the target directory of the generated Resource Compiler script (.RC file). The script file name is the base name of the .MC file with a .RC extension.

filename[**.MC**]
Specifies one or more input message files that is compiled into one or more binary resource files, one for each language specified in the Message Compiler source files.

The Message Compiler reads the source file and generates a C/C++ include file containing definitions for the symbolic names. For each **LanguageId** statement, MC generates a binary file containing a message table resource. It also generates a single RC script file that contains the appropriate Resource Compiler statements to include each binary output file as a resource with the appropriate symbolic name and language type.

Message Compiler Source Files

Message Compiler source files (default extension .MC) are converted into binary resource files by the Message Compiler (MC.EXE). The binary resources are then passed to the Resource Compiler which puts them in the resource table for an application or DLL. For applications performing event logging, the messages are typically placed in a DLL that contains nothing but the message table. This DLL is registered by the application as the source of message text for the events that it logs. The application then uses the event logging functions or the **FormatMessage** function to retrieve and use the message text.

Messages are defined using ASCII text in a text file. The Message Compiler source-format supports multiple versions of the same message text, one for each national language supported by your application. The Message Compiler automatically assigns numbers to each message, and generates a C/C++ include file for use by the application to access a message using a symbolic constant. The purpose of the message text file is to define all of the messages needed by an application, in a format that makes it easy to support multiple languages with the same image file.

Message Source File General Syntax

The general syntax for lines in the message source file is:

keyword=*value*

Spaces around the equal sign are ignored, and the value is delimited by white space (including line breaks) from the next *keyword*=*value* pair. Case is ignored when comparing against keyword names. The *value* portion can be a numeric integer constant using C/C++ syntax; a symbol name that follows the rules for C/C++ identifiers; or a file name that follows the rules for the base name of a file with the FAT file system (eight characters or less, with no periods).

Message Source File Comments

Comment lines are allowed in the source file. A semicolon (;) begins a comment that ends at the end of the line. A comment by itself on a line is copied to the generated include file with the semicolon converted to the C++ line-comment delimiter (//).

Header Section

The overall structure of a message text file consists of a header which defines names and language identifiers for use by the message definitions in the body of the file. The header contains zero or more of the following statements:

MessageIdTypedef = [*type*]
SeverityNames = (*name***=***number*[**:***name*]**)**
FacilityNames = (*name***=***number*[**:***name*]**)**
LanguageNames = (*name***=***number***:***filename***)**
OutputBase = {*number*}

These keywords have the following meaning:

MessageIdTypedef = *type*
Gives a **typedef** name that is used in a type cast for each message code in the generated include file. Each message code appears in the include file with the format:

#define *name* **(**(*type*) **0x***nnnnnnnn***)**

The default value for *type* is empty, and no type cast is generated. It is the programmer's responsibility to specify a typedef statement in the application source code to define the type. The type used in the **typedef** must be large enough to accommodate the entire 32-bit message code.

SeverityNames = (*name***=***number*[**:***name*]**)**
Defines the set of names that are allowed as the value of the **Severity** keyword in the message definition. The set is delimited by left and right parentheses. Associated with each severity name is a number that, when shifted left by 30, gives the bit pattern to logical-OR with the **Facility** value and **MessageId** value to form the full 32-bit message code.

The default value of this keyword is:

```
SeverityNames=(
    Success=0x0
    Informational=0x1
    Warning=0x2
    Error=0x3
    )
```

Severity values occupy the high two bits of a 32-bit message code. Any severity value that does not fit in two bits is an error. The severity codes can be given symbolic names by following each value with **:***name*

FacilityNames = (*name***=***number*[**:***name*]**)**
Defines the set of names that are allowed as the value of the **Facility** keyword in the message definition. The set is delimited by left and right parentheses. Associated with each facility name is a number that, when shifted left by 16 bits, gives the bit pattern to logical-OR with the **Severity** value and **MessageId** value to form the full 32-bit message code.

The default value of this keyword is:

```
FacilityNames=(
    System=0x0FF
    Application=0xFFF
    )
```

Facility codes occupy the low-order 12 bits of the high-order 16 bits of a 32-bit message code. Any facility code that does not fit in 12 bits is an error. This allows for 4096 facility codes. The first 256 codes are reserved for use by the system software. The facility codes can be given symbolic names by following each value with **:***name*

LanguageNames = (*name***=***number***:***filename***)**
Defines the set of names that are allowed as the value of the **Language** keyword in the message

definition. The set is delimited by left and right parentheses. Associated with each language name are a number and a file name that are used to name the generated resource file that contains the messages for that language. The number corresponds to the language identifier to use in the resource table. The number is separated from the file name by a colon.

The initial value of **LanguageNames** is:

```
LanguageNames=(English=1:MSG00001)
```

Any new names in the source file which don't override the built-in names are added to the list of valid languages. This allows an application to support private languages with descriptive names.

OutputBase = {*number*}

Sets the output radix for the message constants output to the C/C++ include file. It does not set the radix for the **Severity** and **Facility** constants; these default to HEX, but can be output in decimal by using the **-d** switch. If present, **OutputBase** overrides the **-d** switch for message constants in the include file.

The legal values for *number* are 10 and 16.

You can use **OutputBase** in both the header section and the message definition section of the input file. You can change **OutputBase** as often as you like.

Message Definitions

Following the header section is the body of the Message Compiler source file. The body consists of zero or more message definitions. Each message definition begins with one or more of the following statements.

MessageId = [*number*|**+***number*]
Severity = *severity_name*
Facility = *facility_name*
SymbolicName = *name*
OutputBase = {*number*}

The **MessageId** statement marks the beginning of the message definition. A **MessageID** statement is required for each message, although the value is optional. If no value is specified, the value used is the previous value for the facility plus one. If the value is specified as *+number* then the value used is the previous value for the facility, plus the number after the plus sign. Otherwise, if a numeric value is given, that value is used. Any **MessageId** value that does not fit in 16 bits is an error.

The **Severity** and **Facility** statements are optional. These statements specify additional bits to OR into the final 32-bit message code. If not specified they default to the value last specified for a message definition. The initial values prior to processing the first message definition are:

```
Severity=Success
Facility=Application
```

The value associated with **Severity** and **Facility** must match one of the names given in the **FacilityNames** and **SeverityNames** statements in the header section.

The **SymbolicName** statement allows you to associate a C/C++ symbolic constant with the final 32-bit message code.

The constant definition in the generated include file has the format:

//
// message text
//

#define *name* **(**(*type*) **0x***nnnnnnnn***)**

The comment before the definition is a copy of the message text for the first language specified in the message definition. The *name* is the value given in the **SymbolicName** statement. The *type* is the type name specified in the **MessageIdTypedef** statement. If no type was specified, the cast is not generated.

The following comment appears in each header file to explain the bit fields in the 32-bit message:

```
//  Values are 32 bit values laid out as follows:
//
//   3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1
//   1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
//  +---+-+-+-----------------------+-------------------------------+
//  |Sev|C|R|     Facility          |               Code            |
//  +---+-+-+-----------------------+-------------------------------+
//
```

The meanings of the fields in the message code are:

Sev
The severity code:

Bits	**Meaning**

00	Success
01	Informational
10	Warning
11	Error

C
The Customer code flag.

R
A reserved bit.

Facility
The facility code.

Code
The status code for the facility.

The **OutputBase** statement lets you set the output radix for the message constants output to the C/C++ include file. It does not set the radix for the **Severity** and **Facility** constants; these default to HEX, but can be output in decimal by using the **-d** switch. If present, **OutputBase** overrides the **-d** switch for message constants in the include file.

The legal values for *number* are 10 and 16

You can use **OutputBase** in both the header section and the message definition section of the input file. You can change **OutputBase** as often as you like.

Language-Specific Message Text Definitions

After the message definition statements, you specify one or more message text definitions.

Syntax

Language=*language_name*
messagetext
.

Each message begins with a **Language** statement that identifies the binary output file for this message. The first line of the message text begins with the next line. The message text is terminated by a line containing a single period at the beginning of the line, immediately followed by a new line. No spaces are allowed around the terminating period. Within the message, blank lines and white space are preserved as part of the message.

You can specify several escape sequences for formatting the message when the message text is used by the application or an event viewer. The percent sign character (**%**) begins all escape sequences.

%0
Terminates a message text line without a trailing newline. This can be used to build up long lines or to terminate the message without a trailing newline, which is useful for prompt messages.

%*n*[**!***printf-format-specifier***!**]
Identifies an insert. Each insert refers to a parameter used in a call to the **FormatMessage** function. **FormatMessage** returns an error if the message text specifies an insert that was not passed to **FormatMessage**.

The value of *n* can be between 1 and 99. The **printf** format specifier must be enclosed in exclamation marks. It is optional and defaults to !s! if not specified.

The **printf** format specifier can contain the ***** specifier for either the precision or width components. When specified, they consume inserts numbered *n*+1 and *n*+2 for their values at run time. MC prints a warning message if these inserts are specified elsewhere in the message text.

Any character following a percent sign other than a digit is formatted in the output message without the percent sign.

You can specify the following additional escape sequences:

%%
Generates a single percent sign in the formatted message text.

%
Generates a hard line break when it occurs at the end of a line. Useful when **FormatMessage** is supplying normal line breaks so the message fits in a certain width.

%r
Generates a hard carriage return, without a trailing newline character.

%b
Generates a space character in the formatted message text. This can be used to ensure there are the appropriate number of trailing spaces in a message text line.

%.
Generates a single period in the formatted message text. This can be used to get a period at the beginning of a line without terminating the message definition.

%!
Generates a single exclamation point in the formatted message text. This can be used to specify an exclamation point immediately after an insert.

Using PView

The PView process viewer lets you examine and modify many characteristics of processes and threads running on your system.

PView can help you answer questions like these:

- How much memory does the program allocate at various points in its execution, and how much memory is being paged out?
- Which processes and threads are using the most CPU time?
- How does the program run at different system priorities?
- What happens if a thread or process stops responding to DDE, OLE, or pipe I/O?
- What percentage of time is being spent running API calls?

Warning PView lets you modify the status of processes running on your system. As a result, by using PView, you can stop processes and potentially halt the entire system. Make sure that you have saved edited files before running PView.

PView Main Dialog

The main PView dialog box consists of several boxes containing information on active processes and threads, and controls to change their behavior (see the following figure).

{ewc msdncd, EWGraphic, group10553 0 /a "SDK.bmp"}

The following buttons control PView actions:

Exit
: Closes PView.

Memory Details
: Opens the Memory Details dialog box. See Memory Details Dialog for more information.

Kill Process
: Removes the highlighted process from the system. This is different from choosing Close from the system menu, because the process is not informed of the shutdown (with WM_DESTROY) before it is stopped.

Refresh
: Updates information in the main PView dialog box and the Memory Details dialog box.

Connect
: View information about the computer specified in the Computer text box. The Computer box should contain the network name of the computer that you wish to view. Your ability to connect to a remote system may be affected by security on the target computer.

Process Selection

The Process Selection list box displays information on the accessible processes running on the system. From this list, you can select a process to use for future actions. All other information and control areas in PView reflect the process chosen in this box.

Note Since Windows NT is a secure operating system, you may not be able to view or alter attributes of some programs running on the system. See your NT *User's Guide* for more information on NT security.

The fields in the Process Selection box are as follows:

Process
: Name of the process on this line. Usually an .EXE filename.

CPU Time
: Amount of CPU time that this process has used.

Privileged
: Percentage of the CPU time that was spent executing privileged code (code in the NT Executive).

User
: Percentage of the CPU time that was spent executing user code. This time includes time running protected subsystem code.

Process Memory Used

The Process Memory Used box displays information on the memory usage of the process selected in the Process Selection box:

Working Set
: The average amount of physical memory used by the process. The longer a process has been running, the more accurate this value is.

Heap Usage
: The current total heap being used by the process. Heap space is taken by dynamically allocated data, including memory reserved by **malloc**, **new**, **LocalAlloc**, **HeapAlloc**, **VirtualAlloc**, and **GlobalAlloc**.

Priority

The Priority buttons let you change the base priority of the process highlighted in the Process Selection box. This priority determines the activity of all threads of the selected process:

Very High
: Maximum priority. CPU time is split between this and other Very High priority processes. Lower priority processes will execute only when all Very High priority processes are blocked.

Normal
: The standard-priority group, also known as "foreground." Most applications run with normal priority.

Idle
: The lowest-priority group, also known as "background." Processes with this priority will execute only when the system has no higher-priority processes that need CPU time. Screen savers run at this priority.

Thread Selection

The Thread Selection list box displays statistics for threads of the process selected in the Process Selection box and lets you select a thread for further operations:

Threads
: The thread ID number. This is the handle returned by **CreateThread**.

CPU Time
: The amount of time that this instance of the thread has been running.

% Privileged
: The percentage of the CPU time that was spent executing privileged code (code in the NT Executive).

% User
: The percentage of the CPU time that was spent executing user code. This time includes time running protected subsystem code.

Thread Information

The Thread Selection box displays execution information about the thread selected in the Thread Selection box:

User PC Value
: The value of the instruction pointer for this thread.

Start Address
: The address of the entry point of this thread. This information is useful for debugging.

Context Switches
: Number of times that this thread has received CPU attention.

Dynamic Priority
: The current dynamic thread priority. This number is determined by many factors, including user activity.

Thread Priority

The Thread Priority box shows you the base priority of the thread selected in the Thread Selection box. This is not an absolute priority, but is a range of priorities that can be selected by the operating system for the selected thread:

Highest
: The highest priority level allowed by the process priority.

Above Normal
: Slightly elevated priority.

Normal
: The standard priority level for the given process priority.

Below Normal
: Reduced priority.

Idle
: No CPU time will be spent on this thread unless all other threads are blocked.

Memory Details Dialog

The Memory Details dialog box (see the following figure) gives information on the process selected by the Process Selection box in the main PView dialog box. To update the information contained in this dialog box, return to the main dialog box and click on the Refresh button.

{ewc msdncd, EWGraphic, group10553 1 /a "SDK.bmp"}

This dialog box consists of the following buttons:

OK
: Returns to the main PView dialog box.

Process
: The name and process ID of the process selected in the Process Selection box of the main dialog.

User Address Space for
: Displays the statistics for specific .EXE or .DLL files or "Total Image Commit," which displays statistics for all components of the currently selected process. These statistics are the following:

  Inaccessible
  : Address space that cannot be accessed. This includes memory reserved by **VirtualAlloc**.

  Read Only
  : Read-only data and code.

  Writeable
  : Total data address space that can be written to.

  Writeable (Not Written)
  : Data address space that can be written to, but has not been.

  Executable
  : Code in selected EXEs and DLLs.

Virtual Memory Counts
: Displays the following statistics on Virtual Memory usage:

  Working Set
  : Average amount of virtual memory used by the process. The longer a process has been running, the more accurate this value is.

  Peak Working Set
  : Maximum value attained by the Working Set described above.

  Private Pages
  : Number of pages marked as private.

  Virtual Size
  : Current size of virtual memory for this process.

  Peak Virtual Size
  : Maximum size of virtual memory for this process.

  Fault Count
  : Number of page faults. Each page fault represents an attempt to access memory at an address that was not in physical memory.

  Paged
  : Number of pages currently in the swap file.

  Peak Paged
  : Maximum number of pages currently in the swap file.

  Non-Paged
  : Number of pages that have not been moved to the swap file.

Compiling Resources

The Microsoft Windows Resource Compiler (RC) is a tool for the Microsoft Windows NT operating system. This section describes how to create a resource-definition (script) file, and how to compile your application's resources and add them to the application's executable file.

Including Resources in an Application

To include resources in your 32-bit Windows application, do the following:

1. Create individual resource files for cursors, icons, bitmaps, dialog boxes, and fonts. To do this, you can use Microsoft Image Editor and Dialog Editor (IMAGEDIT.EXE and DLGEDIT.EXE) and Microsoft Windows Font Editor (FONTEDIT.EXE).
2. Create a resource-definition file (*script*) that describes all the resources used by the application.
3. Compile the script into a resource (.RES) file with RC.
4. Link the compiled resource files into the application's executable file.

Unlike the 16-bit versions of the Windows development system, you do not use RC to include compiled resources into the executable file or to mark the file as a Windows application. The linker recognizes the compiled resource files and links them to the executable file.

Creating a Resource-Definition File

After creating individual resource files for your application's icon, cursor, font, bitmap, and dialog-box resources, you create a resource-definition file, or *script*. A script file is a text file with the extension .RC.

The script lists every resource in your application and describes some types of resources in great detail. For a resource that exists in a separate file, such as an icon or cursor, the script names the resource and the file that contains it. For some resources, such as a menu, the entire definition of the resource exists within the script.

A script file can contain the following types of information:

- Preprocessing directives, which instruct RC to perform actions on the script before compiling it. Directives can also assign values to names.
- Statements, which name and describe resources.

The following sections describe directives and statements you can use in a script. For detailed descriptions and syntax for each statement, see Resource-Definition Statements.

Preprocessing Directives List

The following directives can be used as needed in the script to instruct RC to perform actions or to assign values to names:

Directive	Description
#define	Defines a specified name by assigning it a given value.
#elif	Marks an optional clause of a conditional-compilation block.
#else	Marks the last optional clause of a conditional-compilation block.
#endif	Marks the end of a conditional-compilation block.
#if	Conditionally compiles the script if a specified expression is true.
#ifdef	Conditionally compiles the script if a specified name is defined.
#ifndef	Conditionally compiles the script if a specified name is not defined.
#include	Copies the contents of a file into the resource-definition file.
#undef	Removes the definition of the specified name.

The syntax and semantics for the RC preprocessor are the same as for a C compiler. For more information on preprocessing in RC, see Preprocessing Reference.

Single-Line Statements

A single-line statement can begin with any of the following keywords:

Keyword	Description
BITMAP	Defines a bitmap by naming it and specifying the name of the file that contains it. (To use a particular bitmap, the application requests it by name.)
CURSOR	Defines a cursor by naming it and specifying the name of the file that contains it. (To use a particular cursor, the application requests it by name.)
FONT	Specifies the name of a file that contains a font.
ICON	Defines an icon by naming it and specifying the name of the file that contains it. (To use a particular icon, the application requests it by name.)
LANGUAGE	Sets the language for all resources up to the next **LANGUAGE** statement or to the end of the file. When the **LANGUAGE** statement appears before the **BEGIN** in an **ACCELERATORS**, **DIALOG**, **MENU**, **RCDATA**, or **STRINGTABLE** resource definition, the specified language applies only to that resource.
MESSAGETABLE	Defines a message table by naming it and specifying the name of the file that contains it. The file is a binary resource file generated by the Message Compiler.

Multiline Statements

A multiline statement can begin with any of the following keywords:

Keyword	Description
ACCELERATORS	Defines menu accelerator keys.
DIALOG	Defines a template that an application can use to create dialog boxes.
MENU	Defines the appearance and function of a menu.
RCDATA	Defines data resources. Data resources let you include binary data in the executable file.
STRINGTABLE	Defines string resources. String resources are Unicode strings that can be loaded from the executable file.

Each of these multiline statements allows optional statements before the **BEGIN** ... **END** block that defines the resource. You can specify zero or more of the following statements:

Statement	Description
CHARACTERISTICS *dword*	User-defined information about the resource that can be used by tools that read and write resource files. The value appears in the compiled resource file. However, it is not stored in the executable file and is not used by Windows.
LANGUAGE *language*, *sublanguage*	Specifies the language for the resource. The parameters are constants from WINNLS.H.
VERSION *dword*	User-defined version number for the resource that can be used by tools that read and write resource files. The value appears in the compiled resource file. However, it is not stored in the executable file and is not used by Windows.

Sample Resource-Definition File

The following example shows a script file that defines the resources for an application named Shapes:

```
#include "SHAPES.H"

ShapesCursor  CURSOR  SHAPES.CUR
ShapesIcon    ICON    SHAPES.ICO

ShapesMenu    MENU
    BEGIN
        POPUP "&Shape"
            BEGIN
                MENUITEM "&Clear", ID_CLEAR
                MENUITEM "&Rectangle", ID_RECT
                MENUITEM "&Triangle", ID_TRIANGLE
                MENUITEM "&Star", ID_STAR
                MENUITEM "&Ellipse", ID_ELLIPSE
            END
    END
```

The **CURSOR** statement names the application's cursor resource ShapesCursor and specifies the cursor file SHAPES.CUR, which contains the image for that cursor.

The **ICON** statement names the application's icon resource ShapesIcon and specifies the icon file SHAPES.ICO, which contains the image for that icon.

The **MENU** statement defines an application menu named ShapesMenu, a pop-up menu with five menu items.

The menu definition, enclosed by the **BEGIN** and **END** keywords, specifies each menu item and the menu identifier that is returned when the user selects that item. For example, the first item on the menu, Clear, returns the menu identifier ID_CLEAR when the user selects it. The menu identifiers are defined in the application header file, SHAPES.H.

Using RC (The RC Command Line)

To start RC, use the **RC** command. The following line shows RC command-line syntax:

RC [*options*] *script-file*

The **RC** command's *options* parameter can include one or more of the following options:

/?
Displays a list of **RC** command-line options.

/d
Defines a symbol for the preprocessor that you can test with the **#ifdef** directive.

/fo*resname*
Uses *resname* for the name of the .RES file.

/h
Displays a list of **RC** command-line options.

/i
Searches the specified directory before searching the directories specified by the INCLUDE environment variable.

/l*codepage*
Specifies default language for compilation. For example, -l409 is equivalent to including the following statement at the top of the resource script file:

```
LANGUAGE LANG_ENGLISH,SUBLANG_ENGLISH_US
```

/r
Ignored. Provided for compatibility with existing makefiles.

/v
Displays messages that report on the progress of the compiler.

/x
Prevents RC from checking the INCLUDE environment variable when searching for header files or resource files.

Options are not case sensitive, and a hyphen (**-**) can be used in place of a slash mark (**/**). You can combine single-letter options if they do not require any additional parameters. For example, the following two commands are equivalent:

```
RC /V /X SAMPLE.RC
rc -vx sample.rc
```

The *script-file* parameter specifies the name of the resource-definition file that contains the names, types, filenames, and descriptions of the resources to be compiled.

Defining Names for the Preprocessor

You can specify conditional compilation in a script, based on whether a name is defined on the RC command line with the **/d** option, or in the file or an include file with the **#define** directive.

For example, suppose your application has a pop-up menu, the Debug menu, that should appear only with debugging versions of the application. When you compile the application for normal use, the Debug menu is not included. The following example shows the statements that can be added to the resource-definition file to define the Debug menu:

```
MainMenu MENU
BEGIN
  . . .
#ifdef DEBUG
        POPUP "&Debug"
        BEGIN
                MENUITEM "&Memory usage", ID_MEMORY
                MENUITEM "&Walk data heap", ID_WALK_HEAP
        END
#endif
END
```

When compiling resources for a debugging version of the application, you could include the Debug menu by using the following RC command:

```
rc -d DEBUG myapp.rc
```

To compile resources for a normal version of the application—one that does not include the Debug menu—you could use the following RC command:

```
rc myapp.rc
```

Renaming the Compiled Resource File

By default, when compiling resources, RC names the compiled resource (.RES) file with the base name of the .RC file and places it in the same directory as the .RC file. CVTRES must then be invoked to convert the .RES file to a binary resource (.RBJ) format which can be understood by the linker. The following example compiles MYAPP.RC and creates a compiled resource file named MYAPP.RES in the same directory as MYAPP.RC:

```
rc myapp.rc
```

The **/fo** option gives the resulting .RES file a name that differs from the name of the corresponding .RC file. For example, to name the resulting .RES file NEWFILE.RES, use the following command:

```
rc -fo newfile.res myapp.rc
```

The **/fo** option can also place the .RES file in a different directory. For example, the following command places the compiled resource file MYAPP.RES in the directory C:\SOURCE\RESOURCE:

```
rc -fo c:\source\resource\myapp.res myapp.rc
```

Searching for Files

By default, RC searches for header files and resource files (such as icon and cursor files) first in the current directory and then in the directories specified by the INCLUDE environment variable. (The PATH environment variable has no effect on which directories RC searches.)

Adding a Directory to Search

You can use the **/i** option to add a directory to the list of directories RC searches. The compiler then searches the directories in the following order:

1. The current directory
2. The directory or directories you specify by using the **/i** option, in the order in which they appear on the RC command line
3. The list of directories specified by the INCLUDE environment variable, in the order in which the variable lists them, unless you specify the **/x** option

The following example compiles the resource-definition file MYAPP.RC:

```
rc /i c:\source\stuff /i d:\resources myapp.rc
```

When compiling the script MYAPP.RC, RC searches for header files and resource files first in the current directory, then in C:\SOURCE\STUFF and D:\RESOURCES, and then in the directories specified by the INCLUDE environment variable.

Suppressing the INCLUDE Environment Variable

You can prevent RC from using the INCLUDE environment variable when determining the directories to search. To do so, use the **/x** option. The compiler then searches for files only in the current directory and in any directories you specify by using the **/i** option.

The following example compiles the script file MYAPP.RC:

```
rc /x /i c:\source\stuff myapp.rc
```

When compiling the script MYAPP.RC, RC searches for header files and resource files first in the current directory and then in C:\SOURCE\STUFF. It does not search the directories specified by the INCLUDE environment variable.

Displaying Progress Messages

By default, RC compiles quietly. It does not display messages that report on its progress. You can, however, specify that RC is to display these messages. To do so, use the **/v** option.

The following example causes RC to report on its progress as it compiles the resource-definition file SAMPLE.RC and creates the compiled resource file SAMPLE.RES:

```
rc /v sample.rc
```

Resource-Definition Statements

This section describes the statements that define the resources that the Resource Compiler puts in the resource (.RES) file. Once a resource is linked to the executable file, the application can load the resource as it is needed at run time. All resource statements associate an identifying name or number with a given resource.

Common Statement Parameters

This section lists parameters in common among the resource or control statements. Occasionally, a certain statement will use a parameter differently, or may ignore a parameter. The statement-specific variation is described with the statement in the alphabetical reference.

Common Control Parameters

The general syntax for a control definition, and the meaning of each parameter is as follows:

control [*text*,] *id*, *x*, *y*, *width*, *height* [, *style* [, *extended-style*]]

Horizontal dialog units are 1/4 of the dialog base width unit. Vertical units are 1/8 of the dialog base height unit. The current dialog base units are computed from the height and width of the current system font. The **GetDialogBaseUnits** function returns the dialog base units in pixels. The coordinates are relative to the origin of the dialog box.

Parameters

control
Keyword that indicates the type of control being defined, such as **PUSHBUTTON** or **CHECKBOX**.

text
Specifies text that is displayed with the control. The text is positioned within the control's specified dimensions or adjacent to the control.

This parameter must contain zero or more characters enclosed in double quotation marks (**"**). Strings are automatically null-terminated and converted to Unicode in the resulting resource file, except for strings specified in *raw-data* statements (*raw-data* can be specified in **RCDATA** and user-defined resources.) To specify a Unicode string in *raw-data,* explicitly qualify the string as a wide-character string by using the **L** prefix.

By default, the characters listed between the double quotation marks are ANSI characters, and escape sequences are interpreted as byte escape sequences. If the string is preceded by the **L** prefix, the string is a wide-character string and escape sequences are interpreted as 2-byte escape sequences that specify Unicode characters. If a double quotation mark is required in the text, you must include the double quotation mark twice.

An ampersand (**&**) character in the text indicates that the following character is used as a mnemonic character for the control. When the control is displayed, the ampersand is not shown, but the mnemonic character is underlined. The user can choose the control by pressing the key corresponding to the underlined mnemonic character. To use the ampersand as a character in a string, insert two ampersands (**&&**).

id
Specifies the control identifier. This value must be a 16-bit unsigned integer in the range 0 through 65,535 or a simple arithmetic expression that evaluates to a value in that range.

x
Specifies the x-coordinate of the left side of the control relative to the left side of the dialog box. This value must be a 16-bit unsigned integer in the range 0 through 65,535. The coordinate is in dialog units and is relative to the origin of the dialog box, window, or control containing the specified control.

y
Specifies the y-coordinate of the top side of the control relative to the top of the dialog box. This value must be a 16-bit unsigned integer in the range 0 through 65,535. The coordinate is in dialog units relative to the origin of the dialog box, window, or control containing the specified control.

width
Specifies the width of the control. This value must be a 16-bit unsigned integer in the range 1 through 65,535. The width is in 1/4-character units.

height
Specifies the height of the control. This value must be a 16-bit unsigned integer in the range 1 through 65,535. The height is in 1/8-character units.

style
Specifies the control styles. Use the bitwise OR (|) operator to combine styles.

extended-style

Specifies extended (WS_EX_*xxx*) styles. You must specify a *style* to specify an *extended-style*. See also **EXSTYLE**.

Common Resource Attributes

All resource-definition statements include a *load-mem* option that specifies the loading and memory characteristics of the resource. These attributes are divided into two groups: load attributes and memory attributes. The only attribute that is used by Win32 is the **DISCARDABLE** attribute. The remaining attribute specifiers are allowed in the script for compatibility with existing scripts, but are ignored.

Load Attributes

The load attributes specify when the resource is to be loaded. The load parameter must be one of the following:

PRELOAD
Ignored. In 16-bit Windows, the resource is loaded with the executable file.

LOADONCALL
Ignored. In 16-bit Windows, the resource is loaded when called.

Memory Attributes

The memory attributes specify whether the resource is fixed or movable, whether it is discardable, and whether it is pure. The memory parameter can be one or more of the following:

FIXED
Ignored. In 16-bit Windows, the resource remains at a fixed memory location.

MOVEABLE
Ignored. In 16-bit Windows, the resource can be moved if necessary in order to compact memory.

DISCARDABLE
Resource can be discarded if no longer needed.

PURE
Ignored. Accepted for compatibility with existing resource scripts.

IMPURE
Ignored. Accepted for compatibility with existing resource scripts.

The default is **DISCARDABLE** for cursor, icon, and font resources.

Statement Reference

This section lists the resource and control statements in alphabetic order. For information on the preprocessing directives, see Preprocessing Reference.

ACCELERATORS Resource

The **ACCELERATORS** statement defines one or more accelerators for an application. An accelerator is a keystroke defined by the application to give the user a quick way to perform a task. The **TranslateAccelerator** function is used to translate accelerator messages from the application queue into WM_COMMAND or WM_SYSCOMMAND messages.

Syntax

acctablename **ACCELERATORS**
[*optional-statements*]
BEGIN
event, *idvalue*, [*type*] [*options*]
. . .
END

Parameters

acctablename
Specifies either a unique name or a 16-bit unsigned integer value that identifies the resource.

optional-statements
Zero or more of the following statements:

Statement	Description
CHARACTERISTICS *dword*	User-defined information about a resource that can be used by tools that read and write resource files.
LANGUAGE *language*, *sublanguage*	Specifies the language for the resource. The parameters are constants from WINNLS.H.
VERSION *dword*	User-defined version number for the resource that can be used by tools that read and write resource files.

event
Specifies the keystroke to be used as an accelerator. It can be any one of the following character types:

"*char*"
A single character enclosed in double quotation marks (**"**). The character can be preceded by a caret (^), meaning that the character is a control character.

character
An integer value representing a character. The *type* parameter must be **ASCII**.

virtual-key character
An integer value representing a virtual key. The virtual key for alphanumeric keys can be specified by placing the uppercase letter or number in double quotation marks (for example, "9" or "C"). The *type* parameter must be **VIRTKEY**.

idvalue
Specifies a 16-bit unsigned integer value that identifies the accelerator.

type
Required only when the *event* parameter is a *character* or a *virtual-key character*. The *type* parameter specifies either **ASCII** or **VIRTKEY**; the integer value of *event* is interpreted accordingly. When **VIRTKEY** is specified and *event* contains a string, *event* must be uppercase.

options
Specifies the options that define the accelerator. This parameter can be one or more of the following

values:

NOINVERT
Specifies that no top-level menu item is highlighted when the accelerator is used. This is useful when defining accelerators for actions such as scrolling that do not correspond to a menu item. If **NOINVERT** is omitted, a top-level menu item will be highlighted (if possible) when the accelerator is used.

ALT
Causes the accelerator to be activated only if the ALT key is down.

SHIFT
Causes the accelerator to be activated only if the SHIFT key is down.

CONTROL
Defines the character as a control character (the accelerator is only activated if the CONTROL key is down). This has the same effect as using a caret (^) before the accelerator character in the *event* parameter.

The **ALT**, **SHIFT**, and **CONTROL** options apply only to virtual keys.

Example

The following example demonstrates the use of accelerator keys:

```
1 ACCELERATORS
BEGIN
  "^C",  IDDCLEAR         ; control C
  "K",   IDDCLEAR         ; shift K
  "k",   IDDELLIPSE, ALT  ; alt k
  98,    IDDRECT, ASCII   ; b
  66,    IDDSTAR, ASCII   ; B (shift b)
  "g",   IDDRECT          ; g
  "G",   IDDSTAR          ; G (shift G)
  VK_F1, IDDCLEAR, VIRTKEY                ; F1
  VK_F1, IDDSTAR, CONTROL, VIRTKEY        ; control F1
  VK_F1, IDDELLIPSE, SHIFT, VIRTKEY       ; shift F1
  VK_F1, IDDRECT, ALT, VIRTKEY            ; alt F1
  VK_F2, IDDCLEAR, ALT, SHIFT, VIRTKEY    ; alt shift F2
  VK_F2, IDDSTAR, CONTROL, SHIFT, VIRTKEY ; ctrl shift F2
  VK_F2, IDDRECT, ALT, CONTROL, VIRTKEY   ; alt control F2
END
```

See Also

TranslateAccelerator
Multiline Statements
CHARACTERISTICS, **DIALOG**, **LANGUAGE**, **MENU**, **RCDATA**, **STRINGTABLE**, **VERSION**

AUTO3STATE Control

The **AUTO3STATE** statement creates an automatic 3-state check box. The control is an open box with the given text positioned to the right of the box. When chosen, the box automatically advances between three states: checked, unchecked, and disabled (grayed). The control sends a message to its parent whenever the user chooses the control.

Syntax

AUTO3STATE *text*, *id*, *x*, *y*, *width*, *height* [, *style* [, *extended-style*]]

Parameters

style

Specifies styles for the control, which can be a combination of the BS_AUTO3STATE style and the following styles: WS_TABSTOP, WS_DISABLED, and WS_GROUP.

The default style for this control is BS_AUTO3STATE and WS_TABSTOP.

For more information on the *text*, *id*, *x*, *y*, *width*, *height*, *style*, and *extended-style* parameters, see Common Statement Parameters.

See Also

AUTOCHECKBOX, **CHECKBOX**, **CONTROL**, **STATE3**

AUTOCHECKBOX Control

The **AUTOCHECKBOX** statement creates an automatic check box control. The control is a small rectangle (check box) that has the specified text displayed next to it (typically, to the right). When the user chooses the control, the control highlights the rectangle and sends a message to its parent window. The **AUTOCHECKBOX** statement, which can only be used in the body of a **DIALOG** statement, defines the text, identifier, dimensions, and attributes of the control.

Syntax

AUTOCHECKBOX *text*, *id*, *x*, *y*, *width*, *height* [, *style* [, *extended-style*]]

Parameters

style

Specifies the styles of the control. This value can be a combination of the button class style BS_AUTOCHECKBOX and the WS_TABSTOP and WS_GROUP styles.

If you do not specify a style, the default style is BS_AUTOCHECKBOX and WS_TABSTOP.

For more information on the *text*, *id*, *x*, *y*, *width*, *height*, *style*, and *extended-style* parameters, see Common Statement Parameters.

See Also

AUTO3STATE, **CHECKBOX**, **CONTROL**, **STATE3**

AUTORADIOBUTTON Control

The **AUTORADIOBUTTON** control statement specifies an automatic radio button control. This control automatically performs mutual exclusion with the other **AUTORADIOBUTTON** controls in the same group. When the button is chosen, the application is notified with BN_CLICKED.

Syntax

AUTORADIOBUTTON *text*, *id*, *x*, *y*, *width*, *height* [, *style* [, *extended-style*]]

Parameters

text
The specified text appears next to the radio button.

style
Specifies styles for the automatic radio button, which can be a combination of BUTTON-class styles and the following styles: WS_TABSTOP, WS_DISABLED, and WS_GROUP.

The default style for **AUTORADIOBUTTON** is BS_AUTORADIOBUTTON and WS_TABSTOP.

For more information on the *text*, *id*, *x*, *y*, *width*, *height*, *style*, and *extended-style* parameters, see Common Statement Parameters.

See Also

CONTROL, **RADIOBUTTON**

BITMAP Resource

The **BITMAP** resource-definition statement specifies a bitmap that an application uses in its screen display or as an item in a menu or control.

Syntax

nameID **BITMAP** [*load-mem*] *filename*

Parameter

nameID
Specifies either a unique name or a 16-bit unsigned integer value identifying the resource.

load-mem
Specifies loading and memory attributes for the resource. For more information, see Common Resource Attributes.

filename
Specifies the name of the file that contains the resource. The name must be a valid filename; it must be a full path if the file is not in the current working directory. The path can either be a quoted or nonquoted string.

Example

The following example specifies two bitmap resources:

```
disk1   BITMAP disk.bmp
12      BITMAP PRELOAD diskette.bmp
```

See Also

LoadBitmap

CAPTION Statement

The **CAPTION** statement defines the title for a dialog box. The title appears in the box's caption bar (if it has one).

The default caption is empty.

Syntax

CAPTION "*captiontext***"**

Parameter

captiontext
Specifies a character string enclosed in double quotation marks (**"**).

Example

The following example demonstrates the use of the **CAPTION** statement:

```
CAPTION "Error!"
```

CHARACTERISTICS Statement

The **CHARACTERISTICS** statement allows the developer to specify information about a resource that can be used by tools that read and write resource-definition files. The specified *dword* value appears with the resource in the compiled .RES file. However, the value is not stored in the executable file and has no significance to Windows.

The **CHARACTERISTICS** statement appears before the **BEGIN** in an **ACCELERATORS**, **DIALOG**, **MENU**, **RCDATA**, or **STRINGTABLE** resource definition. The specified value applies only to that resource.

Syntax

CHARACTERISTICS *dword*

Parameter

dword
A user-defined doubleword value.

See Also

Multiline Statements, **ACCELERATORS**, **DIALOG**, **LANGUAGE**, **MENU**, **RCDATA**, **STRINGTABLE**

CHECKBOX Control

The **CHECKBOX** statement creates a check box control. The control is a small rectangle (check box) that has the specified text displayed next to it (typically, to the right). When the user selects the control, the control highlights the rectangle and sends a message to its parent window. The **CHECKBOX** statement, which can only be used in a **DIALOG** statement, defines the text, identifier, dimensions, and attributes of the control.

Syntax

CHECKBOX *text*, *id*, *x*, *y*, *width*, *height* [, *style* [, *extended-style*]]

Parameters

text
Specifies text that is displayed to the right of the control.

style
Specifies the control styles. This value can be a combination of the button class style BS_CHECKBOX and the WS_TABSTOP and WS_GROUP styles.

If you do not specify a style, the default style is BS_CHECKBOX and WS_TABSTOP.

For more information on the *text*, *id*, *x*, *y*, *width*, *height*, *style*, and *extended-style* parameters, see Common Statement Parameters.

Example

This example creates a check-box control that is labeled "Italic":

```
CHECKBOX "Italic", 3, 10, 10, 40, 10
```

See Also

AUTOCHECKBOX, **AUTO3STATE**, **GetDialogBaseUnits**, **STATE3**

CLASS Statement

The **CLASS** statement defines the class of the dialog box. The **CLASS** statement appears in the optional section before a **DIALOG** statement's **BEGIN** keyword. If no class is given, the Windows standard dialog class is used.

Syntax

CLASS *class*

Parameters

class
Specifies a 16-bit unsigned integer or a string, enclosed in double quotation marks (**"**), that identifies the class of the dialog box. If the window procedure for the class does not process a message sent to it, it must call the **DefDlgProc** function to ensure that all messages are handled properly for the dialog box. A private class can use **DefDlgProc** as the default window procedure. The class must be registered with the **cbWndExtra** member of the **WNDCLASS** structure set to DLGWINDOWEXTRA.

Remarks

The **CLASS** statement should only be used with special cases, because it overrides the normal processing of a dialog box. The **CLASS** statement converts a dialog box to a window of the specified class; depending on the class, this could give undesirable results. Do not use the redefined control-class names with this statement.

Example

The following example demonstrates the use of the **CLASS** statement:

```
CLASS "myclass"
```

See Also

DefDlgProc, **DIALOG**

COMBOBOX Control

The **COMBOBOX** statement creates a combination box control (a combo box). A combo box consists of either a static text box or an edit box combined with a list box. The list box can be displayed at all times or pulled down by the user. If the combo box contains a static text box, the text box always displays the selection (if any) in the list box portion of the combo box. If it uses an edit box, the user can type in the desired selection; the list box highlights the first item (if any) that matches what the user has entered in the edit box. The user can then select the item highlighted in the list box to complete the choice. In addition, the combo box can be owner-drawn and of fixed or variable height.

Syntax

COMBOBOX *text*, *id*, *x*, *y*, *width*, *height* [, *style* [, *extended-style*]]

Parameters

style

Specifies the control styles. This value can be a combination of the COMBOBOX class styles and any of the following styles: WS_TABSTOP, WS_GROUP, WS_VSCROLL, and WS_DISABLED.

If you do not specify a style, the default style is CBS_SIMPLE and WS_TABSTOP.

For more information on the *text*, *id*, *x*, *y*, *width*, *height*, *style*, and *extended-style* parameters, see Common Statement Parameters.

Example

This example creates a combo-box control with a vertical scroll bar:

```
COMBOBOX 777, 10, 10, 50, 54, CBS_SIMPLE | WS_VSCROLL | WS_TABSTOP
```

CONTROL: General Control

This statement defines a user-defined control window.

Syntax

CONTROL *text*, *id*, *class*, *style*, *x*, *y*, *width*, *height* [, *extended-style*]

Parameters

class
Specifies a redefined name, character string, or a 16-bit unsigned integer value that defines the class. This can be any one of the control classes; for a list of the control classes, see the first list following this description. If the value is a redefined name supplied by the application, it must be a string enclosed in double quotation marks (**"**).

style
Specifies a redefined name or integer value that specifies the style of the given control. The exact meaning of *style* depends on the *class* value. The sections following this description show the control classes and corresponding styles.

For more information on the *text*, *id*, *x*, *y*, *width*, *height*, *style*, and *extended-style* parameters, see Common Statement Parameters.

The six possible control classes are described in the following sections.

The Button Control Class

A button control is a small rectangular child window that represents a "button" that the user can turn on or off by clicking it with the mouse. Button controls can be used alone or in groups, and can either be labeled or appear without text. Button controls typically change appearance when the user clicks them.

A button can have only one of the following styles, with the exception of BS_LEFTTEXT, which can be combined with check boxes and radio buttons:

BS_3STATE
: Creates a button that is the same as a check box, except that the box can be grayed (dimmed) as well as checked. The grayed state is used to show that the state of the check box is not determined.

BS_AUTO3STATE
: Creates a button that is the same as a three-state check box, except that the box changes its state when the user selects it. The state cycles through checked, grayed, and normal.

BS_AUTOCHECKBOX
: Creates a button that is the same as a check box, except that an X appears in the check box when the user selects the box; the X disappears (is cleared) the next time the user selects the box.

BS_AUTORADIOBUTTON
: Creates a button that is the same as a radio button, except that when the user selects it, the button automatically highlights itself and clears (removes the selection from) any other buttons in the same group.

BS_CHECKBOX
: Creates a small square that has text displayed to its right (unless this style is combined with the BS_LEFTTEXT style).

BS_DEFPUSHBUTTON
: Creates a button that has a heavy black border. The user can select this button by pressing the ENTER key. This style is useful for enabling the user to quickly select the most likely option (the default option).

BS_GROUPBOX
: Creates a rectangle in which other controls can be grouped. Any text associated with this style is displayed in the rectangle's upper-left corner.

BS_LEFTTEXT
: Places text on the left side of the radio button or check box when combined with a radio button or check box style.

BS_OWNERDRAW
: Creates an owner-drawn button. The owner window receives a WM_MEASUREITEM message when the button is created, and it receives a WM_DRAWITEM message when a visual aspect of the button has changed. The BS_OWNERDRAW style cannot be combined with any other button styles.

BS_PUSHBUTTON
: Creates a push button that posts a WM_COMMAND message to the owner window when the user selects the button.

BS_RADIOBUTTON
: Creates a small circle that has text displayed to its right (unless this style is combined with the BS_LEFTTEXT style). Radio buttons are usually used in groups of related but mutually exclusive choices.

The Combobox Control Class

Combo-box controls consist of a selection field similar to an edit control plus a list box. The list box may be displayed at all times or may be dropped down when the user selects a "pop box" next to the selection field.

Depending on the style of the combo box, the user can or cannot edit the contents of the selection field. If the list box is visible, typing characters into the selection box will cause the first list-box entry that matches the characters typed to be highlighted. Conversely, selecting an item in the list box displays the selected text in the selection field. Combo-box control styles are described below.

CBS_SIMPLE
: Displays the list box at all times. The current selection in the list box is displayed in the edit control.

CBS_DROPDOWN
: Similar to CBS_SIMPLE except that the list box is not displayed unless the user selects an icon next to the selection field.

CBS_DROPDOWNLIST
: Similar to CBS_DROPDOWN except that the edit control is replaced by a static text item which displays the current selection in the list box.

CBS_OWNERDRAWFIXED
: Specifies a fixed-height owner-draw combo box. The owner of the list box is responsible for drawing its contents; the items in the list box are all the same height.

CBS_OWNERDRAWVARIABLE
: Specifies a variable-height owner-draw combo box. The owner of the list box is responsible for drawing its contents; the items in the list box can have different heights.

CBS_AUTOHSCROLL
: Scrolls the text in the edit control to the right when the user types a character at the end of the line. If this style is not set, only text that fits within the rectangular boundary is allowed.

CBS_SORT
: Sorts strings entered into the list box.

CBS_HASSTRINGS
: Specifies an owner-draw combo box that contains items consisting of strings. The combo box maintains the memory and pointers for the strings so that the application can use the LB_GETTEXT message to retrieve the text for a particular item.

CBS_OEMCONVERT
: Converts text entered in the combo box edit control from the ANSI character set to the OEM character set and then back to ANSI. This ensures proper character conversion when the application calls the **CharToOem** function to convert an ANSI string in the combo box to OEM characters. This style is most useful for combo boxes that contain filenames and applies only to combo boxes created with the CBS_SIMPLE or CBS_DROPDOWN styles.

The Edit Control Class

An edit control is a rectangular child window in which the user can enter text from the keyboard. The user selects the control, and gives it the input focus, by clicking the mouse inside it or pressing the TAB key. The user can enter text when the control displays a flashing insertion point. The mouse can be used to move the cursor and select characters to be replaced, or to position the cursor for inserting characters. The BACKSPACE key can be used to delete characters.

Edit controls use the fixed-pitch font and display Unicode characters. They expand tab characters into as many space characters as are required to move the cursor to the next tab stop. Tab stops are assumed to be at every eighth character position. Edit control styles are described below.

ES_LEFT
Justifies the text to the left.

ES_CENTER
Centers the text. This style is valid in multiline edit controls only.

ES_RIGHT
Justifies the text to the right. This style is valid in multiline edit controls only.

ES_LOWERCASE
Converts all characters to lowercase as they are typed into the edit control.

ES_UPPERCASE
Converts all characters to uppercase as they are typed into the edit control.

ES_PASSWORD
Displays all characters as an asterisk (*) as they are typed into the edit control. An application can use the EM_SETPASSWORDCHAR message to change the character that is displayed.

ES_MULTILINE
Multiple-line edit control. (The default is single-line.) Shows as many lines of text as possible. The following four styles specify options for horizontal and vertical scrolling.

ES_AUTOVSCROLL specified
Shows as many lines as possible and scrolls vertically when the user presses ENTER. (This is actually the carriage-return character, which the edit control expands to a carriage-return - linefeed combination.)

ES_AUTOVSCROLL not specified
Shows as many lines as possible and beeps if the user presses ENTER when no more lines can be displayed.

ES_AUTOHSCROLL specified
Scrolls horizontally when the insertion point goes past the right edge of the control. To start a new line, press ENTER.

ES_AUTOHSCROLL not specified
Wraps words to the beginning of the next line when necessary. A new line is also started if the user presses ENTER. If the window size changes, the word-wrap position changes, and the text is redisplayed.

Multiple-line edit controls can have scroll bars. An edit control with scroll bars processes its own scroll-bar messages. Edit controls without scroll bars scroll as described above and process any scroll messages sent by the parent window.

ES_AUTOVSCROLL
Scrolls text automatically up one page when the user presses ENTER on the last line.

ES_AUTOHSCROLL
Scrolls text automatically to the right by 10 characters when the user types a character at the end of the line. When the user presses ENTER, the control scrolls all text back to position 0.

ES_NOHIDESEL
Overrides the default action, in which an edit control hides the selection when the control loses the

input focus. Inverts the selection instead.

ES_OEMCONVERT

Converts text entered in the edit control from the ANSI character set to the OEM character set and then back to ANSI. This ensures proper character conversion when the application calls the **CharToOem** function to convert an ANSI string in the edit control to OEM characters. This style is most useful for edit controls that contain filenames.

The Listbox Control Class

Listbox controls consist of a list of character strings. The control is used whenever an application needs to present a list of names, such as filenames, that the user can view and select. The user can select a string by pointing to the string with the mouse and clicking a mouse button. When a string is selected, it is highlighted and a notification message is passed to the parent window. A scroll bar can be used with a listbox control to scroll lists that are too long or too wide for the control window. Listbox control styles are described below.

LBS_STANDARD
: Strings in the list box are sorted alphabetically and the parent window receives an input message whenever the user clicks or double-clicks a string. The list box contains borders on all sides.

LBS_DISABLENOSCROLL
: Shows a disabled vertical scroll bar for the list box when the box does not contain enough items to scroll. If this style is not specified, the scroll bar is hidden when the list box does not contain enough items.

LBS_EXTENDEDSEL
: The user can select multiple items using the mouse with the SHIFT and/or the CONTROL key or special key combinations.

LBS_HASSTRINGS
: An owner-draw list box contains items consisting of strings. The list box maintains the memory and pointers for the strings so the application can use the LB_GETTEXT message to retrieve the text for a particular item.

LBS_NOTIFY
: The parent receives an input message whenever the user clicks or double-clicks a string.

LBS_MULTIPLESEL
: The string selection is toggled each time the user clicks or double-clicks the string. Any number of strings can be selected.

LBS_MULTICOLUMN
: The list box contains multiple columns. The list box can be scrolled horizontally. The LB_SETCOLUMNWIDTH message sets the width of the columns.

LBS_NOINTEGRALHEIGHT
: The size of the list box is exactly the size specified by the application when it created the list box. Normally, Windows sizes a list box so that the list box does not display partial items.

LBS_SORT
: The strings in the list box are sorted alphabetically.

LBS_NOREDRAW
: The list-box display is not updated when changes are made. This style can be changed at any time by sending a WM_SETREDRAW message.

LBS_OWNERDRAWFIXED
: The owner of the list box is responsible for drawing its contents; the items in the list box are all the same height.

LBS_OWNERDRAWVARIABLE
: The owner of the list box is responsible for drawing its contents; the items in the list box are variable in height.

LBS_USETABSTOPS
: The list box is able to recognize and expand tab characters when drawing its strings. The default tab positions are set at every 32 dialog units. (A dialog unit is a horizontal or vertical distance. One horizontal dialog unit is equal to 1/4 of the current dialog base width unit. The dialog base units are computed from the height and width of the current system font. The **GetDialogBaseUnits** function returns the size of the dialog base units in pixels.)

LBS_WANTKEYBOARDINPUT

The owner of the list box receives WM_VKEYTOITEM or WM_CHARTOITEM messages whenever the user presses a key while the list box has input focus. This allows an application to perform special processing on the keyboard input.

The Scroll-Bar Control Class

A scroll-bar control is a rectangle that contains a scroll thumb and has direction arrows at both ends. The scroll bar sends a notification message to its parent whenever the user clicks the mouse in the control. The parent is responsible for updating the thumb position, if necessary. Scroll-bar controls have the same appearance and function as the scroll bars used in ordinary windows. But unlike scroll bars, scroll-bar controls can be positioned anywhere within a window and used whenever needed to provide scrolling input for a window. Scroll-bar control styles are described below.

SBS_VERT
: Vertical scroll bar. If neither SBS_RIGHTALIGN nor SBS_LEFTALIGN is specified, the scroll bar has the height, width, and position given in the **CreateWindow** function.

SBS_RIGHTALIGN
: Used with SBS_VERT. The right edge of the scroll bar is aligned with the right edge of the rectangle specified by the *x*, *y*, *width*, and *height* values given in the **CreateWindow** function. The scroll bar has the default width for system scroll bars.

SBS_LEFTALIGN
: Used with SBS_VERT. The left edge of the scroll bar is aligned with the left edge of the rectangle specified by the *x*, *y*, *width*, and *height* values given in the **CreateWindow** function. The scroll bar has the default width for system scroll bars.

SBS_HORZ
: Horizontal scroll bar. If neither SBS_BOTTOMALIGN nor SBS_TOPALIGN is specified, the scroll bar has the height, width, and position given in the **CreateWindow** function.

SBS_TOPALIGN
: Used with SBS_HORZ. The top edge of the scroll bar is aligned with the top edge of the rectangle specified by the *x*, *y*, *width*, and *height* values given in the **CreateWindow** function. The scroll bar has the default height for system scroll bars.

SBS_BOTTOMALIGN
: Used with SBS_HORZ. The bottom edge of the scroll bar is aligned with the bottom edge of the rectangle specified by the *x*, *y*, *width*, and *height* values given in the **CreateWindow** function. The scroll bar has the default height for system scroll bars.

SBS_SIZEBOX
: Size box. If neither SBS_SIZEBOXBOTTOMRIGHTALIGN nor SBS_SIZEBOXTOPLEFTALIGN is specified, the size box has the height, width, and position given in the **CreateWindow** function.

SBS_SIZEBOXTOPLEFTALIGN
: Used with SBS_SIZEBOX. The top-left corner of the size box is aligned with the top-left corner of the rectangle specified by the *x*, *y*, *width*, and *height* values given in the **CreateWindow** function. The size box has the default size for system size boxes.

SBS_SIZEBOXBOTTOMRIGHTALIGN
: Used with SBS_SIZEBOX. The bottom-right corner of the size box is aligned with the bottom-right corner of the rectangle specified by the *x*, *y*, *width*, and *height* values given in the **CreateWindow** function. The size box has the default size for system size boxes.

The Static Control Class

Static controls are simple text fields, boxes, and rectangles that can be used to label, box, or separate other controls. Static controls take no input and provide no output. Static control styles are described below.

SS_LEFT
: A simple rectangle displaying the given text flush left. The text is formatted before it is displayed. Words that would extend past the end of a line are automatically wrapped to the beginning of the next line.

SS_CENTER
: A simple rectangle displaying the given text centered. The text is formatted before it is displayed. Words that would extend past the end of a line are automatically wrapped to the beginning of the next line.

SS_RIGHT
: A simple rectangle displaying the given text flush right. The text is formatted before it is displayed. Words that would extend past the end of a line are automatically wrapped to the beginning of the next line.

SS_LEFTNOWORDWRAP
: A simple rectangle displaying the given text flush left. Tabs are expanded, but words are not wrapped. Text that extends past the end of a line is clipped.

SS_SIMPLE
: A simple rectangle with a single line of text flush left. The line of text cannot be shortened or altered in any way. (The control's parent window or dialog box must not process the WM_CTLCOLOR message.)

SS_NOPREFIX
: Removes any ampersand (**&**) characters and underlines the next character in the string. Unless this style is specified, Windows will interpret any ampersand characters in the control's text to be accelerator prefix characters. If a static control is to contain text where this feature is not wanted, SS_NOPREFIX may be added. This static-control style may be included with any of the defined static controls.

  You can combine SS_NOPREFIX with other styles by using the bitwise OR (**|**) operator. This is most often used when filenames or other strings that may contain an ampersand need to be displayed in a static control in a dialog box.

SS_ICON
: An icon displayed in the dialog box. The given text is the name of an icon (not a filename) defined elsewhere in the resource file. For the **ICON** statement, the *width* and *height* parameters in the **CreateWindow** function are ignored; the icon automatically sizes itself.

SS_BLACKRECT
: A rectangle filled with the color used to draw window frames. This color is black in the default Windows color scheme.

SS_GRAYRECT
: A rectangle filled with the color used to fill the screen background. This color is gray in the default Windows color scheme.

SS_WHITERECT
: A rectangle filled with the color used to fill window backgrounds. This color is white in the default Windows color scheme.

SS_BLACKFRAME
: Box with a frame drawn with the same color as window frames. This color is black in the default Windows color scheme.

SS_GRAYFRAME
: Box with a frame drawn with the same color as the screen background (desktop). This color is gray

in the default Windows color scheme.

SS_WHITEFRAME
Box with a frame drawn with the same color as window backgrounds. This color is white in the default Windows color scheme.

SS_USERITEM
User-defined item.

CTEXT Control

The **CTEXT** statement creates a centered-text control. The control is a simple rectangle displaying the given text centered in the rectangle. The text is formatted before it is displayed. Words that would extend past the end of a line are automatically wrapped to the beginning of the next line. The **CTEXT** statement, which you can use only in a **DIALOG** statement, defines the text, identifier, dimensions, and attributes of the control.

Syntax

CTEXT *text*, *id*, *x*, *y*, *width*, *height* [, *style* [, *extended-style*]]

Parameters

text
Specifies text that is centered in the rectangular area of the control.

style
Specifies the control styles. This value can be any combination of the following styles: SS_CENTER, WS_TABSTOP, and WS_GROUP.

If you do not specify a style, the default style is SS_CENTER and WS_GROUP.

For more information on the *text*, *id*, *x*, *y*, *width*, *height*, *style*, and *extended-style* parameters, see Common Statement Parameters.

Example

This example creates a centered-text control that is labeled "filename":

```
CTEXT "filename", 101, 10, 10, 100, 100
```

See Also

CONTROL, **DIALOG**, **LTEXT**, **RTEXT**

CURSOR Resource

The **CURSOR** statement specifies a bitmap that defines the shape of the cursor on the display screen.

Syntax

nameID **CURSOR** [*load-mem*] *filename*

Parameters

nameID
Specifies either a unique name or a 16-bit unsigned integer identifying the resource.

load-mem
Specifies loading and memory attributes for the resource. For more information, see Common Resource Attributes.

filename
Specifies the name of the file that contains the resource. The name must be a valid filename; it must be a full path if the file is not in the current working directory. The path can either be a quoted or nonquoted string.

Remarks

Icon and cursor resources can contain more than one image. If the resource is marked with the **PRELOAD** option, Windows loads all images in the resource when the application executes.

Example

The following example specifies two cursor resources; one by name (cursor1) and the other by number (2):

```
cursor1 CURSOR bullseye.cur
2       CURSOR "d:\\cursor\\arrow.cur"
```

DEFPUSHBUTTON Control

The **DEFPUSHBUTTON** statement creates a default push-button control. The control is a small rectangle with a bold outline that represents the default response for the user. The given text is displayed inside the button. The control highlights the button in the usual way when the user clicks the mouse in it and sends a message to its parent window.

Syntax

DEFPUSHBUTTON *text*, *id*, *x*, *y*, *width*, *height* [, *style* [, *extended-style*]]

Parameters

text
Specifies text that is centered in the rectangular area of the control.

style
Specifies the control styles. This value can be a combination of the following styles: BS_DEFPUSHBUTTON, WS_TABSTOP, WS_GROUP, and WS_DISABLED.

If you do not specify a style, the default style is BS_DEFPUSHBUTTON and WS_TABSTOP.

For more information on the *text*, *id*, *x*, *y*, *width*, *height*, *style*, and *extended-style* parameters, see Common Statement Parameters.

Example

This example creates a default push-button control that is labeled "Cancel":

```
DEFPUSHBUTTON "Cancel", 101, 10, 10, 24, 50
```

See Also

PUSHBUTTON, **RADIOBUTTON**

DIALOG Resource

The **DIALOG** statement defines a window that an application can use to create dialog boxes. The statement defines the position and dimensions of the dialog box on the screen as well as the dialog box style.

Syntax

nameID **DIALOG** [*load-mem*] *x*, *y*, *width*, *height*
[*optional-statements*]
BEGIN
control-statement
. . .
END

Parameters

nameID
Identifies the dialog box. This is either a unique name or a unique 16-bit unsigned integer value in the range 1 to 65,535.

load-mem
Specifies loading and memory attributes for the resource. For more information, see Common Resource Attributes.

option-statements
Specifies options for the dialog box. This can be zero or more of the following statements:

CAPTION "*text***"**
Specifies the caption of the dialog box if it has a title bar. See **CAPTION** for more information.

CHARACTERISTICS *dword*
Specifies a user-defined double-word value for use by resource tools. This value is not used by Windows. For more information, see **CHARACTERISTICS**.

CLASS *class*
Specifies a 16-bit unsigned integer or a string, enclosed in double quotation marks (**"**), that identifies the class of the dialog box. See **CLASS** for more information.

LANGUAGE *language,sublanguage*
Specifies the language of the dialog box. See **LANGUAGE** for more information.

STYLE *styles*
Specifies the styles of the dialog box. See **STYLE** for more information.

EXSTYLE=*extended-styles*
Specifies the extended styles of the dialog box. See **EXSTYLE** for more information.

VERSION *dword*
Specifies a user-defined doubleword value. This statement is intended for use by additional resource tools and is not used by Windows. For more information, see **VERSION**.

For more information on the *x*, *y*, *width*, and *height* parameters, see Common Statement Parameters.

Remarks

The **GetDialogBaseUnits** function returns the dialog base units in pixels. The exact meaning of the coordinates depends on the style defined by the **STYLE** option statement. For child-style dialog boxes, the coordinates are relative to the origin of the parent window, unless the dialog box has the style DS_ABSALIGN; in that case, the coordinates are relative to the origin of the display screen.

Do not use the WS_CHILD style with a modal dialog box. The **DialogBox** function always disables the parent/owner of the newly created dialog box. When a parent window is disabled, its child windows are implicitly disabled. Since the parent window of the child-style dialog box is disabled, the child-style

dialog box is too.

If a dialog box has the DS_ABSALIGN style, the dialog coordinates for its upper-left corner are relative to the screen origin instead of to the upper-left corner of the parent window. You would typically use this style when you wanted the dialog box to start in a specific part of the display no matter where the parent window may be on the screen.

The name **DIALOG** can also be used as the class-name parameter to the **CreateWindow** function to create a window with dialog box attributes.

Example

The following demonstrates the usage of the **DIALOG** statement:

```
#include <windows.h>

ErrorDialog DIALOG  10, 10, 300, 110
STYLE WS_POPUP|WS_BORDER
CAPTION "Error!"
BEGIN
    CTEXT "Select One:", 1, 10, 10, 280, 12
    PUSHBUTTON "&Retry", 2, 75, 30, 60, 12
    PUSHBUTTON "&Abort", 3, 75, 50, 60, 12
    PUSHBUTTON "&Ignore", 4, 75, 80, 60, 12
END
```

See Also

CONTROL, **CreateDialog**, **CreateWindow**, **DialogBox**, **DIALOGEX**, **GetDialogBaseUnits**, Multiline Statements, **ACCELERATORS**, **CHARACTERISTICS**, **LANGUAGE**, **MENU**, **RCDATA**, **STRINGTABLE**, **VERSION**

DIALOGEX Resource

The **DIALOGEX** resource is an extension of the **DIALOG** resource. In addition to the functionality offered by **DIALOG**, **DIALOGEX** allows for the following:

- Help IDs on the dialog itself as well as on controls within the dialog.
- Use of the **EXSTYLE** statement for the dialog itself as well as on controls within the dialog.
- Font weight and italic settings for the font to be used in the dialog.
- Control-specific data for controls within the dialog.
- Use of the BEDIT, IEDIT, and HEDIT predefined system class names.

Syntax

nameID **DIALOGEX** [*load-mem*] *x*, *y*, *width*, *height* [, *helpID*]
[*optional-statements*]
BEGIN
control-statement
. . .
END

Parameters

nameID
Identifies the dialog box. This is either a unique name or a unique 16-bit unsigned integer value in the range 1 to 65,535.

load-mem
Specifies loading and memory attributes for the resource. For more information, see Common Resource Attributes.

x
Specifies the location on the screen of the left side of the dialog, in dialog units.

y
Specifies the location on the screen of the top of the dialog, in dialog units.

width
Specifies the width of the dialog, in dialog units.

height
Specifies the height of the dialog, in dialog units.

helpID
Specifies a numeric expression indicating the ID used to identify the dialog during WM_HELP processing.

optional-statements
Specifies options for the dialog box. This can be zero or more of the following statements:

CAPTION *"text"*
Specifies the caption of the dialog box if it has a title bar. See **CAPTION** for more information.

CHARACTERISTICS *DWORD*
Specifies a user-defined DWORD value for use by resource tools. This value is not used by Windows. For more information see **CHARACTERISTICS**.

CLASS *class*
Specifies a 16-bit unsigned integer or a string, enclosed in double quotation marks (**"**), that identifies the class of the dialog box. See **CLASS** for more information.

EXSTYLE=*extended-styles*
Specifies the extended styles of the dialog box. See **EXSTYLE** for more information.

FONT *pointsize, typeface, weight, italic*

pointsize
Specifies the size, in points, of the font.

typeface
Specifies the name of the typeface. This name must be identical to the name defined in the [FONTS] section of WIN.INI. This parameter must be enclosed in double quotation marks (**"**).

weight
Specifies a numeric expression for the font weight (explicit FW_* values defined in WINGDI.H can be used by adding an include to the RC file: **#include** "WINGDI.H")

italic
Indicates whether the font should be italic or not. Specify either TRUE or FALSE for the *italic* value.

LANGUAGE *language, sublanguage*
Specifies the language of the dialog box. See **LANGUAGE** for more information.

MENU *menuname*
Specifies the menu to use. This value is either the name of the menu or the integer identifier.

STYLE *styles*
Specifies the styles of the dialog box. See **STYLE** for more information.

VERSION *DWORD*
Specifies a user-defined **DWORD** value. This statement is intended for use by additional resource tools and is not used by Windows. For more information, see **VERSION**.

The **DIALOGEX** body is marked with a **BEGIN** statement at the beginning of the body and an **END** statement at the end of the body. The body is made up of any number of control statements. There are four families of control statements: generic, static, button, and edit. While each of these families uses a different syntax for defining specific features of its controls, they all share a common syntax for defining the position, size, extended styles, help identification number, and control-specific data. That common syntax is:

(controlType), *x*, *y*, *cx*, *cy* **[,** **[***exStyle***]** **[,** *helpID***]**
[BEGIN
data-element-1 **[,**
data-element-2 **[,**
...]]
END]

(controlType)
Family-specific definition – described below.

x
Specifies the location in the dialog of the left side of the control, in dialog units.

y
Specifies the location in the dialog of the top of the control, in dialog units.

cx
Specifies the width of the control, in dialog units.

cy
Specifies the height of the control, in dialog units.

exStyle
Specifies any number of extended window styles (explicit WS_EX_* style values defined in winuser.h can be used by adding an include to the RC file: #include "winuser.h")

helpID
Specifies a numeric expression indicating the ID used to identify the control during WM_HELP processing.

controlData
Marked by a nested **BEGIN** and **END**, specifies the control-specific data for the control. When a

dialog is created, and a control in that dialog which has control-specific data is created, a pointer to that data is passed into the control's window procedure via the *lParam* to the WM_CREATE message for that control.

Following is the family-specific syntax of the (*controlType*):

Generic control:

CONTROL *controlText*, *id*, *className*, *style*

controlText
Specifies the window text for the control. For more information, see Common Statement Parameters.

id
Specifies the control identifier. For more information, see Common Statement Parameters.

className
Specifies the name of the class. This may be either a string enclosed in double quotation marks (**"**) or one of the following predefined system classes: BUTTON, STATIC, EDIT, LISTBOX, SCROLLBAR, or COMBOBOX.

style
Specifies the window styles (explicit WS_*, BS_*, SS_*, ES_*, LBS_*, SBS_*, and CBS_* style values defined in WINUSER.H can be used by adding an include to the RC file: **#include** "WINUSER.H")

Static control:

staticClass *controlText*, *id*

staticClass {LTEXT | RTEXT | CTEXT}

controlText
Specifies the window text for the control. For more information, see "Common Statement Parameters".

id
Specifies the control identifier. For more information, see "Common Statement Parameters".

Button control:

buttonClass *controlText*, *id*

buttonClass {AUTO3STATE | AUTOCHECKBOX | AUTORADIOBUTTON | CHECKBOX | PUSHBOX | PUSHBUTTON | RADIOBUTTON | STATE3 | USERBUTTON}

controlText
Specifies the window text for the control. For more information, see "Common Statement Parameters".

id
Specifies the control identifier. For more information, see "Common Statement Parameters".

Edit control:

editClass *id*

editClass {EDITTEXT | BEDIT | HEDIT | IEDIT}

id
Specifies the control identifier. For more information, see "Common Statement Parameters".

Arithmetic and Boolean Operations

Valid operations that may be contained in any of the numeric expressions in the statements of **DIALOGEX** are:

- Add ('+')
- Subtract ('-')
- Unary minus ('-')
- Unary NOT ('~')
- AND ('&')
- OR ('|')

See Also

CONTROL, **CreateDialog**, **CreateWindow**, **DialogBox**, **DIALOGEX**, **GetDialogBaseUnits**, Multiline Statements, **ACCELERATORS**, **CHARACTERISTICS**, **LANGUAGE**, **MENU**, **RCDATA**, **STRINGTABLE**, **VERSION**

EDITTEXT Control

The **EDITTEXT** statement defines an EDIT control belonging to the EDIT class. It creates a rectangular region in which the user can type and edit text. The control displays a cursor when the user clicks the mouse in it. The user can then use the keyboard to enter text or edit the existing text. Editing keys include the BACKSPACE and DELETE keys. The user can also use the mouse to select characters to be deleted or to select the place to insert new characters.

Syntax

EDITTEXT *text*, *id*, *x*, *y*, *width*, *height* [, *style* [, *extended-style*]]

Parameters

style

Specifies the control styles. This value can be a combination of the edit class styles and the following styles: WS_TABSTOP, WS_GROUP, WS_VSCROLL, WS_HSCROLL, and WS_DISABLED.

If you do not specify a style, the default style is ES_LEFT, WS_BORDER, and WS_TABSTOP.

For more information on the *text*, *id*, *x*, *y*, *width*, *height*, *style*, and *extended-style* parameters, see Common Statement Parameters.

Example

The following example demonstrates the use of the **EDITTEXT** statement:

```
EDITTEXT  3, 10, 10, 100, 10
```

EXSTYLE Statement

The EXSTYLE command allows you to designate a dialog box with the WS_EX_*style* styles. In a resource definition, the **EXSTYLE** statement is placed with the optional statements before the **BEGIN** keyword.

Syntax

EXSTYLE *extended-style*

Parameter

extended-style
The WS_EX_*style* style for the dialog box or control.

Remarks

It can also be placed with the *load-mem* parameters in a **DIALOG** statement as follows:

name **DIALOG** [*load-mem*] **EXSTYLE=***extended-style x*, *y*, ...

For controls, extended styles are specified after the *style* option in any control definition, as follows:

control text, *id*, *x*, *y*, *width*, *height*, [*style*[, *extended-style*]]

See also

ACCELERATORS, **CONTROL**, **DIALOG**, **MENU**, **POPUP**, **RCDATA**, **STRINGTABLE**, User-Defined Resource

FONT Resource

The **FONT** resource-definition statement specifies a file that contains a font.

For a font resource, *nameID* must be a number; it cannot be a name.

Syntax

nameID **FONT** [*load-mem*] *filename*

Parameters

nameID
Specifies either a unique name or a 16-bit unsigned integer value identifying the resource.

load-mem
Specifies loading and memory attributes for the resource. For more information, see Common Resource Attributes.

filename
Specifies the name of the file that contains the resource. The name must be a valid filename; it must be a full path if the file is not in the current working directory. The path can either be a quoted or nonquoted string.

Example

The following example specifies a single font resource:

```
5 FONT   CMROMAN.FNT
```

FONT Statement

The **FONT** statement defines the font with which Windows will draw text in the dialog box. The font must have been previously loaded, either from the WIN.INI file or by calling the **LoadResource** function.

Syntax

FONT *pointsize*, *typeface*

Parameters

pointsize
Specifies the size, in points, of the font.

typeface
Specifies the name of the typeface. This name must be identical to the name defined in the [FONTS] section of WIN.INI. This parameter must be enclosed in double quotes (**"**).

Example

The following example demonstrates the use of the **FONT** statement:

```
FONT 12, "MS Sans Serif"
```

See Also

DIALOG, **LoadResource**

GROUPBOX Control

The **GROUPBOX** statement creates a group box control. The control is a rectangle that groups other controls together. The controls are grouped by drawing a border around them and displaying the given text in the upper-left corner. The **GROUPBOX** statement, which you can use only in a **DIALOG** statement, defines the text, identifier, dimensions, and attributes of a control window.

When the style contains WS_TABSTOP or the text specifies an accelerator, tabbing or pressing the accelerator key moves the focus to the first control within the group.

Syntax

GROUPBOX *text*, *id*, *x*, *y*, *width*, *height* [, *style* [, *extended-style*]]

Parameters

style

Specifies the control styles. This value can be a combination of the button class style BS_GROUPBOX and the WS_TABSTOP and WS_DISABLED styles.

If you do not specify a style, the default style is BS_GROUPBOX.

For more information on the *text*, *id*, *x*, *y*, *width*, *height*, *style*, and *extended-style* parameters, see Common Statement Parameters.

Example

This example creates a group-box control that is labeled "Options":

```
GROUPBOX "Options", 101, 10, 10, 100, 100
```

See Also

DIALOG

ICON Resource

The **ICON** resource-definition statement specifies a bitmap that defines the shape of the icon to be used for a given application.

Syntax

nameID **ICON** [*load-mem*] *filename*

Parameters

nameID
Specifies either a unique name or a 16-bit unsigned integer value identifying the resource.

load-mem
Specifies loading and memory attributes for the resource. For more information, see Common Resource Attributes.

filename
Specifies the name of the file that contains the resource. The name must be a valid filename; it must be a full path if the file is not in the current working directory. The path can either be a quoted or nonquoted string.

Remarks

Icon and cursor resources can contain more than one image. If the resource is marked as **PRELOAD**, Windows loads all images in the resource when the application executes.

Example

The following example specifies two icon resources:

```
desk1   ICON desk.ico
11      ICON DISCARDABLE custom.ico
```

ICON Control

The **ICON** statement creates an icon control. This control is an icon displayed in a dialog box. The **ICON** control statement, which you can use only in a **DIALOG** statement, defines the icon-resource identifier, icon-control identifier, position, and attributes of a control.

Syntax

ICON *text*, *id*, *x*, *y*, [*width*, *height*, *style* [, *extended-style*]]

Parameters

text
Specifies the name of an icon (not a filename) defined elsewhere in the resource file.

width
This value is ignored and should be set to zero.

height
This value is ignored and should be set to zero.

style
Specifies the control style. This parameter is optional. The only value that can be specified is the SS_ICON style. This is the default style whether this parameter is specified or not.

For more information on the *text*, *id*, *x*, *y*, *width*, *height*, *style*, and *extended-style* parameters, see Common Statement Parameters.

Example

This example creates an icon control whose icon identifier is 901 and whose name is "myicon":

```
ICON "myicon" 901, 30, 30
```

See Also

DIALOG, **ICON**

LANGUAGE Statement

The **LANGUAGE** statement sets the language for all resources up to the next **LANGUAGE** statement or to the end of the file. When the **LANGUAGE** statement appears before the **BEGIN** in an **ACCELERATORS**, **DIALOG**, **MENU**, **RCDATA**, or **STRINGTABLE** resource definition, the specified language applies only to that resource.

Syntax

LANGUAGE *language*, *sublanguage*

Parameters

language
Language identifier. Must be one of the constants from WINNLS.H

sublanguage
Sublanguage identifier. Must be one of the constants from WINNLS.H

See Also

Multiline Statements, **ACCELERATORS**, **CHARACTERISTICS**, **DIALOG**, **MENU**, **RCDATA**, **STRINGTABLE**, **VERSION**

LISTBOX Control

The **LISTBOX** statement creates commonly used controls for a dialog box or window. The control is a rectangle containing a list of strings (such as filenames) from which the user can select. The **LISTBOX** statement, which can only be used in a **DIALOG** or **WINDOW** statement, defines the identifier, dimensions, and attributes of a control window.

Syntax

LISTBOX *text*, *id*, *x*, *y*, *width*, *height* [, *style* [, *extended-style*]]

Parameters

style

Specifies the control styles. This value can be a combination of the list-box class styles and any of the following styles: WS_BORDER and WS_VSCROLL.

If you do not specify a style, the default style is LBS_NOTIFY and WS_BORDER.

For more information on the *text*, *id*, *x*, *y*, *width*, *height*, *style*, and *extended-style* parameters, see Common Statement Parameters.

Example

This example creates a list-box control whose identifier is 101:

```
LISTBOX 101, 10, 10, 100, 100
```

See Also

COMBOBOX, **DIALOG**

LTEXT Control

The **LTEXT** statement creates a left-aligned text control. The control is a simple rectangle displaying the given text left-aligned in the rectangle. The text is formatted before it is displayed. Words that would extend past the end of a line are automatically wrapped to the beginning of the next line. The **LTEXT** statement, which can be used only in a **DIALOG** statement, defines the text, identifier, dimensions, and attributes of the control.

Syntax

LTEXT *text*, *id*, *x*, *y*, *width*, *height* [, *style* [, *extended-style*]]

Parameters

style

Specifies the control styles. This value can be any combination of the BS_RADIOBUTTON style and the following styles: SS_LEFT, WS_TABSTOP, and WS_GROUP.

If you do not specify a style, the default style is SS_LEFT and WS_GROUP.

For more information on the *text*, *id*, *x*, *y*, *width*, *height*, *style*, and *extended-style* parameters, see Common Statement Parameters.

Example

This example creates a left-aligned text control that is labeled "Filename":

```
LTEXT "Filename", 101, 10, 10, 100, 100
```

See Also

CONTROL, **CTEXT**, **DIALOG**, **RTEXT**

MENU Resource

The **MENU** statement defines the contents of a menu resource. A menu resource is a collection of information that defines the appearance and function of an application menu. A menu is a special input tool that lets a user select commands from a list of command names.

Syntax

menuID **MENU** [*load-mem*]
[*optional-statements*]
BEGIN
item-definitions
. . .
END

Parameters

menuID
Identifies the menu. This value is either a unique string or a unique 16-bit unsigned integer value in the range of 1 to 65,535.

load-mem
Specifies loading and memory attributes for the resource. For more information, see Common Resource Attributes.

optional-statements
Zero or more of the following statements:

Statement	Description
CHARACTERISTICS *dword*	User-defined information about a resource that can be used by tools that read and write resource files.
LANGUAGE *language*, *sublanguage*	Specifies the language for the resource. The parameters are constants from WINNLS.H.
VERSION *dword*	User-defined version number for the resource that can be used by tools that read and write resource files.

Example

Following is an example of a complete **MENU** statement:

```
sample MENU
BEGIN
     MENUITEM "&Soup", 100
     MENUITEM "S&alad", 101
     POPUP "&Entree"
     BEGIN
          MENUITEM "&Fish", 200
          MENUITEM "&Chicken", 201, CHECKED
          POPUP "&Beef"
          BEGIN
               MENUITEM "&Steak", 301
               MENUITEM "&Prime Rib", 302
          END
     END
     MENUITEM "&Dessert", 103
END
```

See Also

MENUEX, **MENUITEM**, **POPUP**, Multiline Statements, **ACCELERATORS**, **CHARACTERISTICS**, **DIALOG**, **LANGUAGE**, **RCDATA**, **STRINGTABLE**, **VERSION**

MENU Statement

The **MENU** statement defines the dialog box's menu. If no statement is given, the dialog box has no menu.

Syntax

MENU *menuname*

Parameter

menuname
Specifies the menu to use. This value is either the name of the menu or the integer identifier of the menu.

Example

The following example demonstrates the use of the **MENU** dialog statement:

```
MENU errmenu
```

MENUEX Resource

The **MENUEX** resource is an extension of the **MENU** resource. In addition to the functionality offered by **MENU**, **MENUEX** allows for the following:

- Help IDs on popup menus.
- IDs on popup menus.
- Use of the menu type and state flags created for Windows 95 (MFT_* type flags and MFS_* state flags).

Syntax

menuID **MENUEX**
  BEGIN
    [{[**MENUITEM** *itemText* [, [*id*] [, [*type*] [, *state*]]]] |
    [**POPUP** *itemText* [, [*id*] [, [*type*] [, [*state*] [, *helpID*]]]]
    BEGIN
      popupBody
    END]} ...]
  END

Parameters

MENUITEM statement
  Defines a normal menu item
  itemText
    Specifies the string containing the text for the menu item. For more information, see **MENUITEM**.
  id
    Specifies a numeric expression indicating the ID of the menu item.
  type
    Specifies a numeric expression indicating the type of the menu item (explicit MFT_* type values defined in WINUSER.H can be used by adding an include to the RC file: **#include** "WINUSER.H")
  state
    Specifies a numeric expression indicating the state of the menu item (explicit MFS_* state values defined in WINUSER.H can be used by adding an include to the RC file: **#include** "WINUSER.H")

POPUP statement
  Defines a menu item which has another menu (a submenu) associated with it.
  itemText
    Specifies a string containing the text for the menu item.
  id
    Specifies a numeric expression indicating the ID of the menu item.
  type
    Specifies a numeric expression indicating the type of the menu item (explicit MFT_* type values defined in WINUSER.H can be used by adding an include to the RC file: **#include** "WINUSER.H").
  state
    Specifies a numeric expression indicating the state of the menu item (explicit MFS_* state values defined in WINUSER.H can be used by adding an include to the RC file: **#include** "WINUSER.H").
  helpID
    Specifies a numeric expression indicating the ID used to identify the menu during WM_HELP processing.

popupBody
Contain any combination of the **MENUITEM** and **POPUP** statements described above.

Remarks

Valid arithmetic and Boolean operations that may be contained in any of the numeric expressions in the statements of **MENUEX** are:

- Add ('+')
- Subtract ('-')
- Unary minus ('-')
- Unary NOT ('~')
- AND ('&')
- OR ('|')

See Also

MENU, **MENUITEM**, **POPUP**, Multiline Statements, **ACCELERATORS**, **CHARACTERISTICS**, **DIALOG**, **LANGUAGE**, **RCDATA**, **STRINGTABLE**, **VERSION**

MENUITEM Statement

The **MENUITEM** statement defines a menu item.

Syntax

MENUITEM *text*, *result*, [*optionlist*]
MENUITEM SEPARATOR

Parameters

text
Specifies the name of the menu item.

The string can contain the escape characters **\t** and **\a**. The **\t** character inserts a tab in the string and is used to align text in columns. Tab characters should be used only in pop-up menus, not in menu bars. (For information on pop-up menus, see the **POPUP** statement.) The **\a** character aligns all text that follows it flush right to the menu bar or pop-up menu.

result
Specifies the result generated when the user selects the menu item. This parameter takes an integer value. Menu-item results are always integers; when the user clicks the menu-item name, the result is sent to the window that owns the menu.

optionlist
Specifies the appearance of the menu item. This optional parameter takes one or more redefined menu options, separated by commas or spaces. The menu options are as follows:

CHECKED
Item has a check mark next to it.

GRAYED
Item name is initially inactive and appears on the menu in gray or a lightened shade of the menu-text color.

HELP
Identifies a help item.

INACTIVE
Item name is displayed but it cannot be selected.

MENUBARBREAK
Same as MF_MENUBREAK except that for pop-up menus, it separates the new column from the old column with a vertical line.

MENUBREAK
Places the menu item on a new line for static menu-bar items. For pop-up menus, it places the menu item in a new column with no dividing line between the columns.

The **INACTIVE** and **GRAYED** options cannot be used together.

SEPARATOR
The **MENUITEM SEPARATOR** form of the **MENUITEM** statement creates an inactive menu item that serves as a dividing bar between two active menu items in a pop-up menu.

Example

The following example demonstrates the use of the **MENUITEM** and **MENUITEM SEPARATOR** statements:

```
MENUITEM "&Roman", 206, CHECKED, GRAYED
MENUITEM SEPARATOR
MENUITEM "&Blackletter", 301
```

See Also

MENU, **POPUP**

MESSAGETABLE Resource

The **MESSAGETABLE** resource-definition statement defines the ID and file of an application's message table resource. Message tables are special string resources used in event logging and with the **FormatMessage** function. The file contains a binary message table generated by the Message Compiler. The Message Compiler also generates a resource script file that contains the **MESSAGETABLE** statements you need to include the message table resources in the compiled resource file. Use the **#include** directive to include this resource script into your main resource script.

Syntax

nameID **MESSAGETABLE** *filename*

Parameters

nameID
Specifies either a unique name or a 16-bit unsigned integer value identifying the resource.

filename
Specifies the name of the file that contains the resource. The name must be a valid filename; it must be a full path if the file is not in the current working directory. The path can either be a quoted or nonquoted string.

See Also

Compiling Messages, **STRINGTABLE**

POPUP Resource

The **POPUP** statement marks the beginning of the definition of a pop-up menu. A pop-up menu (which is also known as a drop-down menu) is a special menu item that displays a sublist of menu items when it is selected.

Syntax

POPUP text, [*optionlist*]

BEGIN
item-definitions
.
.
.
END

Parameters

text
Specifies the name of the pop-up menu. This string must be enclosed in double quotation marks (**"**).

optionlist
Specifies one or more redefined menu options that specify the appearance of the menu item. The menu options follow:

CHECKED
Item has a check mark next to it. This option is not valid for a top-level pop-up menu.

GRAYED
Item name is initially inactive and appears on the menu in gray or a lightened shade of the menu-text color.

INACTIVE
Item name is displayed but it cannot be selected.

MENUBARBREAK
Same as MF_MENUBREAK except that for pop-up menus, it separates the new column from the old column with a vertical line.

MENUBREAK
Places the menu item on a new line for static menu-bar items. For pop-up menus, it places the menu item in a new column with no dividing line between the columns.

The **INACTIVE** and **GRAYED** options cannot be used together.

Example

The following example demonstrates the use of the **POPUP** statement:

```
chem MENU
BEGIN
    POPUP "&Elements"
    BEGIN
         MENUITEM "&Oxygen", 200
         MENUITEM "&Carbon", 201, CHECKED
         MENUITEM "&Hydrogen", 202
         MENUITEM SEPARATOR
         MENUITEM "&Sulfur", 203
         MENUITEM "Ch&lorine", 204
    END
```

```
    POPUP "&Compounds"
    BEGIN
         POPUP "&Sugars"
         BEGIN
            MENUITEM "&Glucose", 301
            MENUITEM "&Sucrose", 302, CHECKED
            MENUITEM "&Lactose", 303, MENUBREAK
            MENUITEM "&Fructose", 304
         END
         POPUP "&Acids"
         BEGIN
              "&Hydrochloric", 401
              "&Sulfuric", 402
         END
    END
END
```

See Also

MENU, **MENUITEM**

PUSHBOX Control

The **PUSHBOX** statement creates a push-box control, which is identical to a **PUSHBUTTON**, except that it does not display a button face or frame; only the text appears.

Syntax

PUSHBOX *text*, *id*, *x*, *y*, *width*, *height* [, *style* [, *extended-style*]]

Parameters

style

Specifies styles for the pushbox, which can be a combination of the BS_PUSHBOX style and the following styles: WS_TABSTOP, WS_DISABLED, and WS_GROUP.

The default style is BS_PUSHBOX and WS_TABSTOP.

For more information on the *text*, *id*, *x*, *y*, *width*, *height*, *style*, and *extended-style* parameters, see Common Statement Parameters.

See Also

PUSHBUTTON

PUSHBUTTON Control

The **PUSHBUTTON** statement creates a push-button control. The control is a round-cornered rectangle containing the given text. The text is centered in the control. The control sends a message to its parent whenever the user chooses the control.

Syntax

PUSHBUTTON *text*, *id*, *x*, *y*, *width*, *height* [, *style* [, *extended-style*]]

Parameters

style

Specifies styles for the pushbutton, which can be a combination of the BS_PUSHBUTTON style and the following styles: WS_TABSTOP, WS_DISABLED, and WS_GROUP.

The default style is BS_PUSHBUTTON and WS_TABSTOP.

For more information on the *text*, *id*, *x*, *y*, *width*, *height*, *style*, and *extended-style* parameters, see Common Statement Parameters.

Example

The following example demonstrates the use of the **PUSHBUTTON** statement:

```
PUSHBUTTON "ON", 7, 10, 10, 20, 10
```

See Also

GetDialogBaseUnits

RADIOBUTTON Control

The **RADIOBUTTON** statement creates a radio-button control. The control is a small circle that has the given text displayed next to it, typically to its right. The control highlights the circle and sends a message to its parent window when the user selects the button. The control removes the highlight and sends a message when the button is next selected.

Syntax

RADIOBUTTON *text*, *id*, *x*, *y*, *width*, *height* [, *style* [, *extended-style*]]

Parameters

style

Specifies styles for the radio button, which can be a combination of BUTTON-class styles and the following styles: WS_TABSTOP, WS_DISABLED, and WS_GROUP.

The default style for **RADIOBUTTON** is BS_RADIOBUTTON and WS_TABSTOP.

For more information on the *text*, *id*, *x*, *y*, *width*, *height*, *style*, and *extended-style* parameters, see Common Statement Parameters.

Example

The following example demonstrates the use of the **RADIOBUTTON** statement:

```
RADIOBUTTON "Italic", 100, 10, 10, 40, 10
```

See Also

GetDialogBaseUnits

RCDATA Resource

The **RCDATA** statement defines a raw data resource for an application. Raw data resources permit the inclusion of binary data directly in the executable file.

Syntax

nameID **RCDATA** [*load-mem*]
[*optional-statements*]
BEGIN
raw-data
. . .
END

Parameters

nameID
Specifies either a unique name or a 16-bit unsigned integer value that identifies the resource.

load-mem
Specifies loading and memory attributes for the resource. For more information, see Common Resource Attributes.

optional-statements
Zero or more of the following statements:

Statement	Description
CHARACTERISTICS *dword*	User-defined information about a resource that can be used by tools that read and write resource files.
LANGUAGE *language*, *sublanguage*	Specifies the language for the resource. The parameters are constants from WINNLS.H.
VERSION *dword*	User-defined version number for the resource that can be used by tools that read and write resource files.

raw-data
Specifies raw data consisting of one or more integers or strings of characters. Integers can be specified in decimal, octal, or hexadecimal format. RC does not automatically append a terminating null character to a string. The string is a sequence of the specified ANSI (byte) characters unless explicitly qualified as a wide-character string with the **L** prefix. Strings in all resources other than **RCDATA** and user-defined resources are Unicode strings.

The block of data begins on a **DWORD** boundary and RC performs no padding or alignment of data within the *raw-data* block. It is the programmer's responsibility to ensure the proper alignment of data within the block.

Example

The following example demonstrates the use of the **RCDATA** statement:

```
resname RCDATA
BEGIN
   L"Here is a Unicode string\0", /* A Unicode string. Note: explicitly
                                     null-terminated */
   1024,                  /* int        */
   0x029a,                /* hex int    */
   0o733,                 /* octal int  */
   "\07"                  /* octal byte */
END
```

See Also

Multiline Statements, **ACCELERATORS**, **CHARACTERISTICS**, **DIALOG**, **LANGUAGE**, **MENU**, **STRINGTABLE**, **VERSION**

RTEXT Control

The **RTEXT** statement creates a right-aligned text control. The control is a simple rectangle displaying the given text right-aligned in the rectangle. The text is formatted before it is displayed. Words that would extend past the end of a line are automatically wrapped to the beginning of the next line.

Syntax

RTEXT *text*, *id*, *x*, *y*, *width*, *height* [, *style* [, *extended-style*]]

Parameters

style

Specifies styles for the text control, which can be any combination of the following: WS_TABSTOP and WS_GROUP.

The default style for **RTEXT** is SS_RIGHT and WS_GROUP.

For more information on the *text*, *id*, *x*, *y*, *width*, *height*, *style*, and *extended-style* parameters, see Common Statement Parameters.

Example

The following example demonstrates the use of the **RTEXT** statement:

```
RTEXT "Number of Messages", 4, 30, 50, 100, 10
```

See Also

CONTROL, **CTEXT**, **DIALOG**, **LTEXT**

SCROLLBAR Control

The **SCROLLBAR** statement creates a scroll-bar control. The control is a rectangle that contains a scroll box and has direction arrows at both ends. The scroll-bar control sends a notification message to its parent whenever the user clicks the mouse in the control. The parent is responsible for updating the scroll-box position. Scroll-bar controls can be positioned anywhere in a window and used whenever needed to provide scrolling input.

Syntax

SCROLLBAR *text*, *id*, *x*, *y*, *width*, *height* [, *style* [, *extended-style*]]

Parameters

style

Specifies a combination (or none) of the following styles: WS_TABSTOP, WS_GROUP, and WS_DISABLED.

In addition to these styles, the *style* parameter may contain a combination (or none) of the SCROLLBAR-class styles. The default style for **SCROLLBAR** is SBS_HORZ.

For more information on the *text*, *id*, *x*, *y*, *width*, *height*, *style*, and *extended-style* parameters, see Common Statement Parameters.

Example

The following example demonstrates the use of the **SCROLLBAR** statement:

```
SCROLLBAR 999, 25, 30, 10, 100
```

STATE3 Control

The **STATE3** statement defines a three-state check box control. The control is identical to a **CHECKBOX**, except that it has three states: checked, unchecked, and disabled (grayed).

Syntax

STATE3 *text*, *id*, *x*, *y*, *width*, *height* [, *style* [, *extended-style*]]

Parameters

text
Specifies text that is displayed to the right of the control.

style
Specifies the control styles. This value can be a combination of the button class style BS_3STATE and the WS_TABSTOP and WS_GROUP styles.

If you do not specify a style, the default style is BS_3STATE and WS_TABSTOP.

For more information on the *text*, *id*, *x*, *y*, *width*, *height*, *style*, and *extended-style* parameters, see Common Statement Parameters.

See Also

AUTOCHECKBOX, **CHECKBOX**, **CONTROL**

STRINGTABLE Resource

The **STRINGTABLE** statement defines one or more string resources for an application. String resources are simply null-terminated Unicode strings that can be loaded when needed from the executable file, using the **LoadString** function.

Syntax

STRINGTABLE [*load-mem*]
[*optional-statements*]
BEGIN
stringID string
. . .
END

Parameter

load-mem
Specifies loading and memory attributes for the resource. For more information, see Common Resource Attributes.

optional-statements
Zero or more of the following statements:

Statement	Description
CHARACTERISTICS *dword*	User-defined information about a resource that can be used by tools that read and write resource files.
LANGUAGE *language*, *sublanguage*	Specifies the language for the resource. The parameters are constants from WINNLS.H.
VERSION *dword*	User-defined version number for the resource that can be used by tools that read and write resource files.

stringID
Specifies an unsigned 16-bit integer that identifies the resource.

string
Specifies one or more strings, enclosed in double quotation marks. The string must be no longer than 255 characters and must occupy a single line in the source file. To add a carriage return to the string, use this character sequence: **\012**. For example, "Line one\012Line two" would define a string that would be displayed as follows:

```
Line one
Line two
```

Remarks

Grouping strings in separate sections allows all related strings to be read in at one time and discarded together. When possible, an application should make the table movable and discardable. RC allocates 16 strings per section and uses the identifier value to determine which section is to contain the string. Strings with the same upper-12 bits in their identifiers are placed in the same section.

Example

The following example demonstrates the use of the **STRINGTABLE** statement:

```
#define IDS_HELLO    1
#define IDS_GOODBYE  2

STRINGTABLE
BEGIN
    IDS_HELLO,   "Hello"
    IDS_GOODBYE, "Goodbye"
END
```

See Also

LoadString, Multiline Statements, **ACCELERATORS**, **CHARACTERISTICS**, **DIALOG**, **LANGUAGE**, **MENU**, **RCDATA**, **VERSION**

STYLE Statement

The **STYLE** statement defines the window style of the dialog box. The window style specifies whether the box is a pop-up or a child window. The default style has the following attributes: WS_POPUP, WS_BORDER, and WS_SYSMENU.

Syntax

STYLE *style*

Parameter

style

Specifies the window style. This parameter takes an integer value or redefined name. The following lists the redefined styles:

DS_LOCALEDIT

Specifies that edit controls in the dialog box will use memory in the application's data section. By default, all edit controls in dialog boxes use memory outside the application's data section. This feature can be suppressed by adding the DS_LOCALEDIT flag to the **STYLE** command for the dialog box. If this flag is not used, EM_GETHANDLE and EM_SETHANDLE messages must not be used since the storage for the control is not in the application's data section. This feature does not affect edit controls created outside of dialog boxes.

DS_MODALFRAME

Creates a dialog box with a modal dialog box frame that can be combined with a title bar and System menu by specifying the WS_CAPTION and WS_SYSMENU styles.

DS_NOIDLEMSG

Suppresses WM_ENTERIDLE messages that Windows would otherwise send to the owner of the dialog box while the dialog box is displayed.

DS_SYSMODAL

Creates a system-modal dialog box.

WS_BORDER

Creates a window that has a border.

WS_CAPTION

Creates a window that has a title bar (implies the WS_BORDER style).

WS_CHILD

Creates a child window. It cannot be used with the WS_POPUP style.

WS_CHILDWINDOW

Creates a child window that has the WS_CHILD style.

WS_CLIPCHILDREN

Excludes the area occupied by child windows when drawing within the parent window. Used when creating the parent window.

WS_CLIPSIBLINGS

Clips child windows relative to each other; that is, when a particular child window receives a WM_PAINT message, this style clips all other top-level child windows out of the region of the child window to be updated. (If the WS_CLIPSIBLINGS style is not given and child windows overlap, it is possible, when drawing in the client area of a child window, to draw in the client area of a neighboring child window.) For use with the WS_CHILD style only.

WS_DISABLED

Creates a window that is initially disabled.

WS_DLGFRAME

Creates a window with a modal dialog box frame but no title.

WS_GROUP

Specifies the first control of a group of controls in which the user can move from one control to the

next by using the arrow keys. All controls defined with the WS_GROUP style after the first control belong to the same group. The next control with the WS_GROUP style ends the style group and starts the next group (that is, one group ends where the next begins). This style is valid only for controls.

WS_HSCROLL
Creates a window that has a horizontal scroll bar.

WS_ICONIC
Creates a window that is initially iconic. For use with the WS_OVERLAPPED style only.

WS_MAXIMIZE
Creates a window of maximum size.

WS_MAXIMIZEBOX
Creates a window that has a Maximize box.

WS_MINIMIZE
Creates a window of minimum size.

WS_MINIMIZEBOX
Creates a window that has a Minimize box.

WS_OVERLAPPED
Creates an overlapped window. An overlapped window has a caption and a border.

WS_OVERLAPPEDWINDOW
Creates an overlapped window having the WS_OVERLAPPED, WS_CAPTION, WS_SYSMENU, WS_THICKFRAME, WS_MINIMIZEBOX, and WS_MAXIMIZEBOX styles.

WS_POPUP
Creates a pop-up window. It cannot be used with the WS_CHILD style.

WS_POPUPWINDOW
Creates a pop-up window that has the WS_POPUP, WS_BORDER, and WS_SYSMENU styles. The WS_CAPTION style must be combined with the WS_POPUPWINDOW style to make the System menu visible.

WS_SIZEBOX
Creates a window that has a size box. Used only for windows with a title bar or with vertical and horizontal scroll bars.

WS_SYSMENU
Creates a window that has a System-menu box in its title bar. Used only for windows with title bars. If used with a child window, this style creates a Close box instead of a System-menu box.

WS_TABSTOP
Specifies one of any number of controls through which the user can move by using the TAB key. The TAB key moves the user to the next control specified by the WS_TABSTOP style. This style is valid only for controls.

WS_THICKFRAME
Creates a window with a thick frame that can be used to size the window.

WS_VISIBLE
Creates a window that is initially visible. This applies to overlapping and pop-up windows. For overlapping windows, the *y* parameter is used as a parameter for the **ShowWindow** function.

WS_VSCROLL
Creates a window that has a vertical scroll bar.

Remarks

If the redefined names are used, you must include WINDOWS.H.

User-Defined Resource

A user-defined resource statement specifies a resource that contains application-specific data. The data can have any format and can be defined either as the content of a given file (if the *filename* parameter is given) or as a series of numbers and strings (if the *raw-data* block is given).

Syntax

nameID typeID [*load-mem*] *filename*

The *filename* specifies the name of a file containing the binary data of the resource. The contents of the file are included as the resource. RC does not interpret the binary data in any way. It is the programmer's responsibility to ensure that the data is properly aligned for the target computer architecture.

A user-defined resource can also be defined completely in the resource script using the syntax:

nameID typeID [*load-mem*]
BEGIN
raw-data
END

Parameters

nameID
Specifies either a unique name or a 16-bit unsigned integer that identifies the resource.

typeID
Specifies either a unique name or a 16-bit unsigned integer that identifies the resource type. If a number is given, it must be greater than 255. The numbers 1 through 255 are reserved for existing and future redefined resource types.

load-mem
Specifies loading and memory attributes for the resource. For more information, see Common Resource Attributes.

filename
Specifies the name of the file that contains the resource data. The parameter must be a valid filename; it must be a full path if the file is not in the current working directory.

raw-data
Specifies raw data consisting of one or more integers or strings of characters. Integers can be specified in decimal, octal, or hexadecimal format. RC does not automatically append a terminating null character to a string. The string is a sequence of the specified ANSI (byte) characters unless explicitly qualified as a wide-character string with the L prefix. Strings in all resources other than **RCDATA** and user-defined resources are Unicode strings.

The block of data begins on a **DWORD** boundary, and RC performs no padding or alignment of data within the *raw-data* block. It is the programmer's responsibility to ensure the proper alignment of data within the block.

Example

The following example shows several user-defined statements:

```
array   MYRES   data.res
14      300     custom.res
18 MYRES2
BEGIN
  "Here is a data string\0", /* A string. Note: explicitly
                                null-terminated */
   1024,                     /* int        */
   0x029a,                   /* hex int    */
   0o733,                    /* octal int  */
   "\07"                     /* octal byte */
END
```

VERSION Statement

The **VERSION** statement allows the developer to specify version information about a resource that can be used by tools that read and write resource files. The specified *dword* value appears with the resource in the compiled .RES file. However, the value is not stored in the executable file and has no significance to Windows.

The **VERSION** statement appears before the **BEGIN** in an **ACCELERATORS**, **DIALOG**, **MENU**, **RCDATA**, or **STRINGTABLE** resource definition. The specified value applies only to that resource.

Syntax

VERSION *dword*

Parameter

dword
A user-defined doubleword value.

See Also

Multiline Statements, **ACCELERATORS**, **CHARACTERISTICS**, **DIALOG**, **LANGUAGE**, **MENU**, **RCDATA**, **STRINGTABLE**

VERSIONINFO Resource

The **VERSIONINFO** statement creates a version-information resource. The resource contains such information about the file as its version number, its intended operating system, and its original filename. The resource is intended to be used with the File Installation library functions.

Syntax

versionID **VERSIONINFO** *fixed-info*
BEGIN
block-statement
. . .
END

Parameters

versionID
Specifies the version-information resource identifier. This value must be 1.

fixed-info
Specifies the version information, such as the file version and the intended operating system. This parameter consists of the following statements:

FILEVERSION *version*
Specifies the binary version number for the file. The *version* consists of two 32-bit integers, defined by four 16-bit integers. For example, "FILEVERSION 3,10,0,61" is translated into two doublewords: 0x0003000a and 0x0000003d, in that order. Therefore, if *version* is defined by the doublewords dw1 and dw2, they need to appear in the **FILEVERSION** statement as follows: **HIWORD**(dw1), **LOWORD**(dw1), **HIWORD**(dw2), **LOWORD**(dw2).

PRODUCTVERSION *version*
Specifies the binary version number for the product with which the file is distributed. The *version* parameter is two 32-bit integers, defined by four 16-bit integers. For more information about *version*, see the **FILEVERSION** description.

FILEFLAGSMASK *fileflagsmask*
Specifies which bits in the **FILEFLAGS** statement are valid. If a bit is set, the corresponding bit in **FILEFLAGS** is valid.

FILEFLAGS *fileflags*
Specifies the Boolean attributes of the file. The *fileflags* parameter must be the combination of all the file flags that are valid at compile time. For Windows 3.1, this value is 0x3f.

FILEOS *fileos*
Specifies the operating system for which this file was designed. The *fileos* parameter can be one of the operating system values given in the Comments section.

FILETYPE *filetype*
Specifies the general type of file. The *filetype* parameter can be one of the file type values listed in the Comments section.

FILESUBTYPE *subtype*
Specifies the function of the file. The *subtype* parameter is zero unless the *type* parameter in the **FILETYPE** statement is **VFT_DRV**, **VFT_FONT**, or **VFT_VXD**. For a list of file subtype values, see the Comments section.

block-statement
Specifies one or more version-information blocks. A block can contain string information or variable information.

Remarks

To use the constants specified with the **VERSIONINFO** statement, the WINVER.H file (included in

WINDOWS.H) must be included in the resource-definition file.

The following list describes the parameters used in the **VERSIONINFO** statement:

fileflags

Specifies a combination of the following values:

VS_FF_DEBUG
File contains debugging information or is compiled with debugging features enabled.

VS_FF_INFOINFERRED
File contains a dynamically created version-information resource. Some of the blocks for the resource may be empty or incorrect. This value is not intended to be used in version-information resources created by using the **VERSIONINFO** statement.

VS_FF_PATCHED
File has been modified and is not identical to the original shipping file of the same version number.

VS_FF_PRERELEASE
File is a development version, not a commercially released product.

VS_FF_PRIVATEBUILD
File was not built using standard release procedures. If this value is given, the **StringFileInfo** block must contain a **PrivateBuild** string.

VS_FF_SPECIALBUILD
File was built by the original company using standard release procedures but is a variation of the standard file of the same version number. If this value is given, the **StringFileInfo** block must contain a **SpecialBuild** string.

fileos

Specifies one of the following values:

Value	Description
VOS_UNKNOWN	Operating system for which the file was designed is unknown to Windows.
VOS_DOS	File was designed for MS-DOS.
VOS_NT	File was designed for Windows NT.
VOS_WINDOWS16	File was designed for Windows version 3.0 or later.
VOS_WINDOWS32	File was designed for 32-bit Windows.
VOS_DOS_WINDOWS16	File was designed for Windows version 3.0 or later running with MS-DOS.
VOS_DOS_WINDOWS32	File was designed for 32-bit Windows running with MS-DOS.
VOS_NT_WINDOWS32	File was designed for 32-bit Windows running with Windows NT.

The values 0x00002L, 0x00003L, 0x20000L and 0x30000L are reserved.

filetype

Specifies one of the following values:

Value	Description
VFT_UNKNOWN	File type is unknown to Windows.
VFT_APP	File contains an application.
VFT_DLL	File contains a dynamic-link library (DLL).
VFT_DRV	File contains a device driver. If the **dwFileType** member is **VFT_DRV**, the **dwFileSubtype** member

	contains a more specific description of the driver.
VFT_FONT	File contains a font. If the **dwFileType** member is **VFT_FONT**, the **dwFileSubtype** member contains a more specific description of the font.
VFT_VXD	File contains a virtual device.
VFT_STATIC_LIB	File contains a static-link library.

All other values are reserved for use by Microsoft.

subtype

Specifies additional information about the file type.

If the **FILETYPE** statement specifies **VFT_DRV**, this parameter can be one of the following values:

Value	Description
VFT2_UNKNOWN	Driver type is unknown to Windows.
VFT2_DRV_COMM	File contains a communications driver.
VFT2_DRV_PRINTER	File contains a printer driver.
VFT2_DRV_KEYBOARD	File contains a keyboard driver.
VFT2_DRV_LANGUAGE	File contains a language driver.
VFT2_DRV_DISPLAY	File contains a display driver.
VFT2_DRV_MOUSE	File contains a mouse driver.
VFT2_DRV_NETWORK	File contains a network driver.
VFT2_DRV_SYSTEM	File contains a system driver.
VFT2_DRV_INSTALLABLE	File contains an installable driver.
VFT2_DRV_SOUND	File contains a sound driver.

If the **FILETYPE** statement specifies **VFT_FONT**, this parameter can be one of the following values:

Value	Description
VFT2_UNKNOWN	Font type is unknown to Windows.
VFT2_FONT_RASTER	File contains a raster font.
VFT2_FONT_VECTOR	File contains a vector font.
VFT2_FONT_TRUETYPE	File contains a TrueType font.

If the **FILETYPE** statement specifies **VFT_VXD**, this parameter must be the virtual-device identifier included in the virtual-device control block.

All *subtype* values not listed here are reserved for use by Microsoft.

langID

Specifies one of the following language codes:

Code	Language	Code	Language
0x0401	Arabic	0x0415	Polish
0x0402	Bulgarian	0x0416	Brazilian Portuguese
0x0403	Catalan	0x0417	Rhaeto-Romanic
0x0404	Traditional Chinese	0x0418	Romanian
0x0405	Czech	0x0419	Russian
0x0406	Danish	0x041A	Croato-Serbian (Latin)
0x0407	German	0x041B	Slovak
0x0408	Greek	0x041C	Albanian
0x0409	U.S. English	0x041D	Swedish

0x040A	Castilian Spanish	0x041E	Thai
0x040B	Finnish	0x041F	Turkish
0x040C	French	0x0420	Urdu
0x040D	Hebrew	0x0421	Bahasa
0x040E	Hungarian	0x0804	Simplified Chinese
0x040F	Icelandic	0x0807	Swiss German
0x0410	Italian	0x0809	U.K. English
0x0411	Japanese	0x080A	Mexican Spanish
0x0412	Korean	0x080C	Belgian French
0x0413	Dutch	0x0C0C	Canadian French
0x0414	Norwegian - Bokml	0x100C	Swiss French
0x0810	Swiss Italian	0x0816	Portuguese
0x0813	Belgian Dutch	0x081A	Serbo-Croatian (Cyrillic)
0x0814	Norwegian - Nynorsk		

charsetID

Specifies one of the following character-set identifiers:

Identifier	Character Set
0	7-bit ASCII
932	Windows, Japan (Shift - JIS X-0208)
949	Windows, Korea (Shift - KSC 5601)
950	Windows, Taiwan (GB5)
1200	Unicode
1250	Windows, Latin-2 (Eastern European)
1251	Windows, Cyrillic
1252	Windows, Multilingual
1253	Windows, Greek
1254	Windows, Turkish
1255	Windows, Hebrew
1256	Windows, Arabic

string-name

Specifies one of the following redefined names:

Comments

Specifies additional information that should be displayed for diagnostic purposes.

CompanyName

Specifies the company that produced the file—for example, "Microsoft Corporation" or "Standard Microsystems Corporation, Inc." This string is required.

FileDescription

Specifies a file description to be presented to users. This string may be displayed in a list box when the user is choosing files to install—for example, "Keyboard Driver for AT-Style Keyboards" or "Microsoft Word for Windows". This string is required.

FileVersion

Specifies the version number of the file—for example, "3.10" or "5.00.RC2". This string is required.

InternalName

Specifies the internal name of the file, if one exists—for example, a module name if the file is a dynamic-link library. If the file has no internal name, this string should be the original filename,

without extension. This string is required.

LegalCopyright
Specifies all copyright notices that apply to the file. This should include the full text of all notices, legal symbols, copyright dates, and so on—for example, "Copyright© Microsoft Corporation 1990-1992". This string is optional.

LegalTrademarks
Specifies all trademarks and registered trademarks that apply to the file. This should include the full text of all notices, legal symbols, trademark numbers, and so on—for example, "Windows™ is a trademark of Microsoft® Corporation". This string is optional.

OriginalFilename
Specifies the original name of the file, not including a path. This information enables an application to determine whether a file has been renamed by a user. The format of the name depends on the file system for which the file was created. This string is required.

PrivateBuild
Specifies information about a private version of the file—for example, "Built by TESTER1 on \TESTBED". This string should be present only if the **VS_FF_PRIVATEBUILD** flag is set in the **dwFileFlags** member of the **VS_FIXEDFILEINFO** structure of the root block.

ProductName
Specifies the name of the product with which the file is distributed—for example, "Microsoft Windows". This string is required.

ProductVersion
Specifies the version of the product with which the file is distributed—for example, "3.10" or "5.00.RC2". This string is required.

SpecialBuild
Specifies how this version of the file differs from the standard version—for example, "Private build for TESTER1 solving mouse problems on M250 and M250E computers". This string should be present only if the **VS_FF_SPECIALBUILD** flag is set in the **dwFileFlags** member of the **VS_FIXEDFILEINFO** structure in the root block.

A string information block has the following form:

```
BLOCK "StringFileInfo"
  BEGIN
    BLOCK "lang-charset"
    BEGIN
      VALUE "string-name", "value"
          . . .
    END
  END
```

Following are the parameters in the **StringFileInfo** block:

lang-charset
Specifies a language and character-set identifier pair. It is a hexadecimal string consisting of the concatenation of the language and character-set identifiers listed earlier in this section.

string-name
Specifies the name of a value in the block and can be one of the redefined names listed earlier in this section.

value
Specifies, as a character string, the value of the corresponding string name. More than one **VALUE** statement can be given.

A variable information block has the following form:

```
BLOCK "VarFileInfo"
```

BEGIN
VALUE "Translation",
langID, *charsetID*
. . .
END

Following are the parameters in the variable information block:

langID
Specifies one of the language identifiers listed earlier in this section.

charsetID
Specifies one of the character-set identifiers listed earlier in this section. More than one identifier pair can be given, but each pair must be separated from the preceding pair with a comma.

Preprocessing Reference

The syntax and semantics for the RC preprocessor are the same as for the preprocessor in a C compiler. Single-line comments begin with two forward slashes (**//**) and run to the end of the line. Block comments begin with an opening delimiter (**/***) and run to a closing delimiter (***/**). Comments do not nest.

You use the **#define** directive to define symbols for your resource identifiers in a header file. You include this header both in the resource script and your application source code. Similarly, the values for attributes and styles are defined by including the C header files for Windows.

You can also define macros. The standard C preprocessing operators (#, ##) can be used in a macro definition. For detailed information on preprocessing and defining macros, see your C compiler documentation.

The rest of this section describes each of the preprocessing directives in alphabetical order.

#define

The **#define** directive assigns the given value to the specified name. All subsequent occurrences of the name are replaced by the value.

Syntax

#define *name value*

Parameters

name
Specifies the name to be defined. This value is any combination of letters, digits, and punctuation.

value
Specifies any integer, character string, or line of text.

Example

This example assigns values to the names "NONZERO" and "USERCLASS":

```
#define     NONZERO     1
#define     USERCLASS   "MyControlClass"
```

See Also

#ifdef, **#ifndef**, **#undef**

#elif

The **#elif** directive marks an optional clause of a conditional-compilation block defined by a **#ifdef**, **#ifndef**, or **#if** directive. The directive controls conditional compilation of the resource file by checking the specified constant expression. If the constant expression is nonzero, **#elif** directs the compiler to continue processing statements up to the next **#endif**, **#else**, or **#elif** directive and then skip to the statement after **#endif**. If the constant expression is zero, **#elif** directs the compiler to skip to the next **#endif**, **#else**, or **#elif** directive. You can use any number of **#elif** directives in a conditional block.

Syntax

#elif *constant-expression*

Parameter

constant-expression
Specifies the expression to be checked. This value is a defined name, an integer constant, or an expression consisting of names, integers, and arithmetic and relational operators.

Example

In this example, **#elif** directs the compiler to process the second **BITMAP** statement only if the value assigned to the name "Version" is less than 7. The **#elif** directive itself is processed only if Version is greater than or equal to 3.

```
#if Version < 3
BITMAP 1 errbox.bmp
#elif Version < 7
BITMAP 1 userbox.bmp
#endif
```

See Also

#else, **#endif**, **#if**, **#ifdef**, **#ifndef**

#else

The **#else** directive marks an optional clause of a conditional-compilation block defined by a **#ifdef**, **#ifndef**, or **#if** directive. The **#else** directive must be the last directive before the **#endif** directive.

This directive has no arguments.

Syntax

#else

Example

This example compiles the second **BITMAP** statement only if the name "DEBUG" is not defined:

```
#ifdef DEBUG
    BITMAP 1 errbox.bmp
#else
    BITMAP 1 userbox.bmp
#endif
```

See Also

#elif, **#endif**, **#if**, **#ifdef**, **#ifndef**

#endif

The **#endif** directive marks the end of a conditional-compilation block defined by a **#ifdef** directive. One **#endif** is required for each **#if**, **#ifdef**, or **#ifndef** directive.

Syntax

#endif

This directive has no arguments.

See Also

#elif, **#else**, **#if**, **#ifdef**, **#ifndef**

#if

The **#if** directive controls conditional compilation of the resource file by checking the specified constant expression. If the constant expression is nonzero, **#if** directs the compiler to continue processing statements up to the next **#endif**, **#else**, or **#elif** directive and then skip to the statement after the **#endif** directive. If the constant expression is zero, **#if** directs the compiler to skip to the next **#endif**, **#else**, or **#elif** directive.

Syntax

#if *constant-expression*

Parameter

constant-expression
Specifies the expression to be checked. This value is a defined name, an integer constant, or an expression consisting of names, integers, and arithmetic and relational operators.

Example

This example compiles the **BITMAP** statement only if the value assigned to the name "Version" is less than 3:

```
#if Version < 3
BITMAP 1 errbox.bmp
#endif
```

See Also

#elif, **#else**, **#endif**, **#ifdef**, **#ifndef**

#ifdef

The **#ifdef** directive controls conditional compilation of the resource file by checking the specified name. If the name has been defined by using a **#define** directive or by using the **/d** command-line option with the Resource Compiler, **#ifdef** directs the compiler to continue with the statement immediately after the **#ifdef** directive. If the name has not been defined, **#ifdef** directs the compiler to skip all statements up to the next **#endif** directive.

Syntax

#ifdef *name*

Parameter

name
Specifies the name to be checked by the directive.

Example

This example compiles the **BITMAP** statement only if the name "Debug" is defined:

```
#ifdef Debug
BITMAP 1 errbox.bmp
#endif
```

See Also

#define, **#endif**, **#if**, **#ifndef**, **#undef**

#ifndef

The **#ifndef** directive controls conditional compilation of the resource file by checking the specified name. If the name has not been defined or if its definition has been removed by using the **#undef** directive, **#ifndef** directs the compiler to continue processing statements up to the next **#endif**, **#else**, or **#elif** directive and then skip to the statement after the **#endif** directive. If the name is defined, **#ifndef** directs the compiler to skip to the next **#endif**, **#else**, or **#elif** directive.

Syntax

#ifndef *name*

Parameter

name
Specifies the name to be checked by the directive.

Example

This example compiles the **BITMAP** statement only if the name "Optimize" is not defined:

```
#ifndef Optimize
BITMAP 1 errbox.bmp
#endif
```

See Also

#elif, **#else**, **#endif**, **#if**, **#ifdef**, **#undef**

#include

The **#include** directive causes Resource Compiler to process the file specified in the *filename* parameter. This file should be a header file that defines the constants used in the resource-definition file.

Syntax

#include *filename*

Parameter

filename
Specifies the name of the file to be included. If the file is in the current directory, the string must be enclosed in double quotation marks; if the file is in the directory specified by the INCLUDE environment variable, the string must be enclosed in less-than and greater-than characters (**<>**). You must give a full path enclosed in double quotation marks (**"**) if the file is not in the current directory or in the directory specified by the INCLUDE environment variable.

Example

This example processes the header files WINDOWS.H and HEADERS\MYDEFS.H while compiling the resource-definition file:

```
#include <windows.h>
#include "headers\mydefs.h"
```

See Also

#define

#undef

The **#undef** directive removes the current definition of the specified name. All subsequent occurrences of the name are processed without replacement.

Syntax

#undef *name*

Parameter

name
Specifies the name to be removed. This value is any combination of letters, digits, and punctuation.

Example

This example removes the definitions for the names "nonzero" and "USERCLASS":

```
#undef     nonzero
#undef     USERCLASS
```

See Also

#define

RC Diagnostic Messages

This section contains descriptions of diagnostic messages produced by the Microsoft Resource Compiler (RC) for Windows NT. Many of these messages appear when RC is not able to compile resources properly. The descriptions clarify the cause of each message. The messages are listed in alphabetic order.

A capital V in parentheses (V) at the beginning of a message description indicates that the message is displayed only if RC is run with the **/v** (verbose) option. These messages are generally informative and do not necessarily indicate errors.

A

Accelerator Type required (ASCII or VIRTKEY)
The *type* parameter in the **ACCELERATORS** statement must contain either the **ASCII** or **VIRTKEY** value.

B

BEGIN expected in Accelerator Table
An **ACCELERATORS** statement was not followed by the **BEGIN** keyword.

BEGIN expected in Dialog
A **DIALOG** statement was not followed by the **BEGIN** keyword.

BEGIN expected in menu
A **MENU** statement was not followed by the **BEGIN** keyword.

BEGIN expected in RCData
An **RCDATA** statement was not followed by the **BEGIN** keyword.

BEGIN expected in String Table
A **STRINGTABLE** statement was not followed by the **BEGIN** keyword.

BEGIN expected in VERSIONINFO resource
A **VERSIONINFO** statement was not followed by the **BEGIN** keyword.

Bitmap file *resource-file* **is not in** *version-number* **format.**
Use Microsoft Image Editor (IMAGEDIT.EXE) to convert old resource files to the current format.

C

Cannot Re-use String Constants
You are using the same value twice in a **STRINGTABLE** statement. Make sure that you have not mixed overlapping decimal and hexadecimal values.

Control Character out of range [A - Z]
A control character in the **ACCELERATORS** statement is invalid. The character following the caret (^) must be in the range A through Z.

Could not find RCPP.EXE
The preprocessor (RCPP.EXE) must be in the current directory or in a directory specified in the PATH environment variable.

Could not open *in-file-name*
Microsoft Windows Resource Compiler (RC) could not open the specified file. Make sure that the file exists and that you typed the filename correctly.

Couldn't open *resource-name*
Microsoft Windows Resource Compiler (RC) could not open the specified file. Make sure that the file exists and that you typed the filename correctly.

Creating *resource-name*
(V) Microsoft Windows Resource Compiler (RC) is creating a new binary resource (.RES) file.

E

Empty menus not allowed
An **END** keyword appears before any menu items are defined in the **MENU** statement. Empty menus are not permitted by Microsoft Windows Resource Compiler (RC). Make sure that you do not have any opening quotation marks within the **MENU** statement.

END expected in Dialog
The **END** keyword must appear at the end of a **DIALOG** statement. Make sure that there are no opening quotation marks left from the preceding statement.

END expected in menu
The **END** keyword must appear at the end of a **MENU** statement. Make sure that there are no mismatched **BEGIN** and **END** statements.

Error Creating *resource-name*
Microsoft Windows Resource Compiler (RC) could not create the specified binary resource (.RES) file. Make sure that it is not being created on a read-only drive. Use the **/v** option to find out whether the file is being created.

Expected Comma in Accelerator Table
Microsoft Windows Resource Compiler (RC) requires a comma between the *event* and *idvalue* parameters in the **ACCELERATORS** statement.

Expected control class name
The *class* parameter of a **CONTROL** statement in the **DIALOG** statement must be one of the following control types: **BUTTON**, **COMBOBOX**, **EDIT**, **LISTBOX**, **SCROLLBAR**, **STATIC**, or user-defined. Make sure that the class is spelled correctly.

Expected font face name
The *typeface* parameter of the **FONT** statement in the **DIALOG** statement must be a character string enclosed in double quotation marks (**"**). This parameter specifies the name of a font.

Expected ID value for Menuitem
The **MENU** statement must contain a **MENUITEM** statement, which has either an integer or a symbolic constant in the *MenuID* parameter.

Expected Menu String
Each **MENUITEM** and **POPUP** statement must contain a *text* parameter. This parameter is a string enclosed in double quotation marks (**"**) that specifies the name of the menu item or pop-up menu. A **MENUITEM SEPARATOR** statement requires no quoted string.

Expected numeric command value
Microsoft Windows Resource Compiler (RC) was expecting a numeric *idvalue* parameter in the **ACCELERATORS** statement. Make sure that you have used a **#define** constant to specify the value and that the constant used is spelled correctly.

Expected numeric constant in string table
A numeric constant, defined in a **#define** statement, must immediately follow the **BEGIN** keyword in a **STRINGTABLE** statement.

Expected numeric point size
The *pointsize* parameter of the **FONT** statement in the **DIALOG** statement must be an integer point-size value.

Expected Numerical Dialog constant
A **DIALOG** statement requires integer values for the *x, y, width*, and *height* parameters. Make sure that these values, which are included after the **DIALOG** keyword, are not negative.

Expected String in STRINGTABLE
A string is expected after each numeric *stringid* parameter in a **STRINGTABLE** statement.

Expected String or Constant Accelerator command
Microsoft Windows Resource Compiler (RC) was not able to determine which key was being set up for the accelerator. The *event* parameter in the **ACCELERATORS** statement might be invalid.

Expected VALUE, BLOCK, or END keyword.
The **VERSIONINFO** structure requires a **VALUE**, **BLOCK**, or **END** keyword.

Expecting number for ID
A number is expected for the *id* parameter of a **CONTROL** statement in the **DIALOG** statement. Make sure that you have a number or a **#define** statement for the control identifier.

Expecting quoted string for key
The key string following the **BLOCK** or **VALUE** keyword should be enclosed in double quotation marks.

Expecting quoted string in dialog class
The *class* parameter of the **CLASS** statement in the **DIALOG** statement must be an integer or a string enclosed in double quotation marks (**"**).

Expecting quoted string in dialog title
The *captiontext* parameter of the **CAPTION** statement in the **DIALOG** statement must be a character string, enclosed in double quotation marks (**"**).

F

File not found: *filename*
The file specified in the **rc** command was not found. Make sure that the file has not been moved to another directory and that the filename or path is typed correctly.

Font names must be ordinals
The *pointsize* parameter in the **FONT** statement must be an integer, not a string.

I

Insufficient memory to spawn RCPP.EXE
There wasn't enough memory to run the preprocessor (RCPP.EXE).

Invalid Accelerator
An *event* parameter in the **ACCELERATORS** statement was not recognized or was more than two characters long.

Invalid Accelerator Type (ASCII or VIRTKEY)
The *type* parameter in the **ACCELERATORS** statement must contain either the **ASCII** or **VIRTKEY** value.

Invalid control character
A control character in the **ACCELERATORS** statement is invalid. A valid control character consists of a caret (^) followed by a single letter.

Invalid Control type
The **CONTROL** statement in a **DIALOG** statement must be one of the following: **AUTO3STATE**, **AUTOCHECKBOX**, **AUTORADIOBUTTON**, **CHECKBOX**, **COMBOBOX**, **CONTROL**, **CTEXT**, **DEFPUSHBUTTON**, **EDITTEXT**, **GROUPBOX**, **ICON**, **LISTBOX**, **LTEXT**, **PUSHBOX**, **PUSHBUTTON**, **RADIOBUTTON**, **RTEXT**, **SCROLLBAR**, **STATE3**.

Invalid directive in preprocessed RC file
The specified filename has an embedded newline character.

Invalid .EXE file
The executable (.EXE) file is invalid. Make sure that the linker created it correctly and that the file exists.

Invalid switch, *option*
An option used was invalid. For a list of the command-line options, use the **rc /?** command.

Invalid switch, too many -D switches
Too many **-d** options were specified. To define a large number of symbols, use the **#include** directive to include a file of definitions.

Invalid type
The resource type was not among the types defined in the include file.

Invalid usage. Use RC -? for Help
Make sure that you have at least one filename to work with. For a list of the command-line options, use the **rc -?** command.

I/O error reading file
Read failed. Since this is a generic routine, no specific filename is supplied.

I/O error seeking in file
Seeking in file failed. Since this is a generic routine, no specific filename is supplied.

I/O error writing file
Write failed. Since this is a generic routine, no specific filename is supplied.

N

No resource binary filename specified.
The **/fo** option was used, but no binary resource (.RES) file was specified.

O

Old DIB in *resource-name*. **Pass it through IMAGEDIT.**
The resource file specified is not compatible with Win32. Make sure you have read and saved this file using the latest version of Microsoft Image Editor (IMAGEDIT.EXE).

Out of heap memory
There was not enough memory.

Out of memory, needed *n* **bytes**
Microsoft Windows Resource Compiler (RC) was not able to allocate the specified amount of memory.

R

Redefinition of Button Type
The specified button styles conflict with or modify the style specified by the control keyword. This is a warning only and may be ignored.

The following common method of declaring an automatic radio button generates this warning:

```
RADIOBUTTON "&Italic", MYRB_ITALIC, 10, 10, 30, 12, BS_AUTORADIOBUTTON
```

To avoid the warning, use the control statement appropriate for the type of button that you want to define. In this example, the corrected statement is:

```
AUTORADIOBUTTON "&Italic", MYRB_ITALIC, 10, 10, 30, 12
```

RC: Invalid switch: *option*
An option used was invalid. For a list of the command-line options, use the **rc /?** command.

RC: RCPP.ERR not found
The RCPP.ERR file must be in the current directory or in a directory specified in the PATH environment variable.

RC terminated by user
A CTRL+C key combination was pressed, exiting Microsoft Windows Resource Compiler (RC).

RC terminating after preprocessor errors
For information about preprocessor errors, see the documentation for the preprocessor.

RCPP.EXE is not a valid executable
The preprocessor (RCPP.EXE) may have been altered. Try copying the file again from the Microsoft Windows Software Development Kit (SDK) disks.

Resource file *resource-name* **is not in** *version-number* **format.**
Make sure your icons and cursors have been read and saved using the latest version of Microsoft Image Editor (IMAGEDIT.EXE).

T

Text string or ordinal expected in Control
The *text* parameter of a **CONTROL** statement in the **DIALOG** statement must be either a text string or an ordinal reference to the type of control that is expected. If using an ordinal, make sure that you have a **#define** statement for the control.

U

Unable to create *destination*
Microsoft Windows Resource Compiler (RC) was not able to create the destination file. Make sure that there is enough disk space.

Unbalanced Parentheses
Make sure that you have closed every opening parenthesis in the **DIALOG** statement.

Unexpected value in RCData
The values for the *raw-data* parameter in the **RCDATA** statement must be integers or strings, separated by commas. Make sure that you did not leave out a comma or a quotation mark around a string.

Unexpected value in value data
A statement contained information with a format or size different from the expected value for that parameter.

Unknown DIB header format
The device-independent bitmap (DIB) header is not a **BITMAPCOREHEADER** or **BITMAPINFOHEADER** structure.

Unknown error spawning RCPP.EXE
For an unknown reason, the preprocessor (RCPP.EXE) has not started. Try copying the file again from the distribution disks.

Unknown Menu SubType
The *item-definitions* parameter of the **MENU** statement can contain only **MENUITEM** and **POPUP** statements.

Unrecognized VERSIONINFO field; BEGIN or comma expected
The format of the information following a **VERSIONINFO** statement is incorrect.

V

Version WORDs separated by commas expected
Values in an information block for a **VERSIONINFO** statement should be separated by commas.

W

Warning: ASCII character not equivalent to virtual key code
An invalid virtual-key code exists in the **ACCELERATORS** statement. The ASCII values for some characters such as *****, **^**, or **&** are not equivalent to the virtual-key codes for the corresponding keys. In the case of the asterisk (*****), the virtual-key code is equivalent to the ASCII value for 8, the numeric character on the same key. Therefore, the statement **VIRTKEY '*'** is invalid.

Warning: SHIFT or CONTROL used without VIRTKEY
The **ALT**, **SHIFT**, and **CONTROL** options apply only to virtual keys in the **ACCELERATORS** statement. Make sure that the **VIRTKEY** option is used with one of these other options.

Writing resource *type image* **lang:***language* **size:***size*
(V) Microsoft Windows Resource Compiler (RC) is writing the specified resource. The *type* is the resource type. It may be a number or a string. The *image* can be a number or a string. The *language* is the national language of the resource. The *size* is the decimal size of the resource in bytes.

Monitoring Messages and Data

Microsoft Windows NT Spy (SPY.EXE) and DDESpy (DDESPY.EXE) are monitoring tools for the Microsoft Windows NT operating system. Spy makes it possible for you to monitor messages sent to one or more windows and to examine the values of message parameters. DDESpy lets you monitor dynamic data exchange (DDE) activity in the Microsoft Windows NT operating system. This section describes the use of both Spy and DDESpy.

Spy

This section describes how to use the Spy, Edit, Options, and Start!/Stop! menus to specify how Spy is to operate.

The Spy Menu

Use the Spy menu to select the window you want Spy to monitor. The Spy menu contains the following commands:

Select Window

Specifies the window that Spy is to monitor. By default, all windows are selected. When you choose the Window command, Spy displays the Spy Window dialog box. This dialog box displays information about the window in which the cursor is located. As you move the cursor from window to window, the following information is updated:

Item	Description
Window	Handle of the window.
Class	Window class.
Parent	Handle of the parent window and the name of the program that created the parent window.
Rect	Upper-right and lower-left coordinates of the window and the window size in screen coordinates.
Style	Style bits of the window in which the cursor is located, the principal style of the window, and an identifier if the window is a child window. The principal style can be WS_POPUP, WS_ICONIC, WS_OVERLAPPED, or WS_CHILD.
All Windows	Specifies that Spy is to display messages received by all windows.

About

Provides information about the version of Spy you are using.

Exit

Closes Spy.

The Edit Menu

You can use the Edit menu to Cut, Copy, and Clear information from the Spy window to the clipboard.

Command	Description
Cut	Cuts the selected text to the clipboard.
Copy	Copies the selected text to the clipboard.
Clear	Clears the selected text from the Spy window.

The Options Menu

The Options menu displays a dialog box in which you make selections about the following:

- Which message types Spy is to monitor
- Which font Spy should use
- Which output device Spy is to send messages to

You make your selections from items displayed under Messages, Font, and Output.

Selecting Message Types

Under Messages, you can select any of the following message types you want Spy to monitor:

Message	Description
DDE	Dynamic data exchange (DDE) messages, such as WM_DDE_REQUEST
Clipboard	Clipboard messages, such as WM_RENDERFORMAT
Mouse	Mouse messages, such as WM_MOUSEMOVE and WM_SETCURSOR
Non Client	Windows nonclient messages, such as WM_NCDESTROY and WM_NCHITTEST
Keyboard	Keyboard messages, such as WM_KEYDOWN and WM_ACTIVATE
Button Control	Button control messages, such as BN_CLICKED and BM_GETCHECK
Combo Box Control	Combo Box control messages, such as CB_ADDSTRING and CB_LIMITTEXT
Editfield Control	Editfield control messages, such as EM_GETSEL and EM_UNDO
Listbox Control	Listbox control messages, such as LB_GETSEL and LB_GETCARETINDEX
Static Control	Static control messages, such as STM_GETICON and WM_CTLCOLORSTATIC
WM_USER	Application-defined WM_USER messages
Unknown	Messages other than those explicitly listed

By default, Spy monitors all messages.

Selecting the Font

Under Font, you can select which font you would like Spy to use in its window. This dialog box allows you to choose the font name, font style and point size, and displays a sample of the font you have chosen.

Selecting the Output Device

Under Output, you can select which of the following output devices you want Spy to send messages to:

Device	Description
Window	Spy displays messages in the Spy window. You can specify how many messages Spy stores in its buffer. By default, Spy stores up to 250 lines of messages, which you can view by scrolling through the Spy window. You can also change the maximum number of lines that can be stored in the buffer.
Com1	Spy sends messages to the COM1 port.
File	Spy sends messages to the specified file. The default output file is SPY.LOG.

The Start!/Stop! Menu

After using the Options and Spy menus to make your selections, start Spy by clicking the Start! menu. To stop monitoring messages, click the Stop! menu.

DDESpy

This section describes how to use DDESpy. For more information about dynamic data exchange, see the *Microsoft Windows 32-Bit API Programmer's Reference*.

DDESpy is a typical DDE monitoring application. Because DDE is a cooperative activity, DDE monitoring applications must follow certain guidelines for your Windows system to operate properly while they are in use. In particular, DDE monitoring applications should not perform DDE server or client communications – problems may arise when the monitoring application intercepts its own communications.

The Output Menu

DDESpy can display DDE information in a window or on your debugging terminal or can save the displayed information in a file for later use.

You use the Output menu to select where DDESpy is to send output. If you choose the File command, you must specify the name of an output file. After you have chosen the File command once, DDESpy prompts you for an output filename every time you restart the application.

From the Output menu, you can choose to send your output to either a debug terminal or to the DDESpy window. You can also choose the Clear Screen command to clear the display window. Use the Mark command to add text to the display as a marker – for example, before a DDE event to make it easier to find the event in the output file.

The Monitor Menu

You use the Monitor menu to specify one or more types of DDE information that DDESpy is to display. The following information can be displayed:

- String handle data
- Posted DDE messages
- Callbacks
- Errors
- Filters

The Dynamic Data Exchange protocol passes information by using shared memory. The contents of the shared memory depend on the type of DDE transaction. Several structures have been defined to allow applications using DDE to access the information in shared memory. DDESpy displays the contents of the appropriate structure for the DDE activity being monitored.

You can use the Filters option to choose the type of DDE messages and types of DDE callbacks to monitor.

Monitoring String-Handle Data

The DDE protocol uses the **MONHSZSTRUCT** structure to pass string-handle data. DDESpy displays the following information from this structure:

- Task (application instance)
- Time, in milliseconds, since you started Windows
- Activity type (create, destroy, or increment)
- String handle
- String contents

The following example shows a typical DDESpy display of string-handle data:

```
Task:0x94f, Time:519700, String Handle Created: c4a4(this is a test)
Task:0x94f, Time:526126, String Handle Created: c4aa(another test)
```

Monitoring Posted DDE Messages

The DDE protocol uses the **MONMSGSTRUCT** structure to post DDE messages. DDESpy displays the following information from this structure:

- Task
- Time
- Handle of receiving window
- Transaction type (sent or posted)
- Message type
- Handle of sending application
- Other message-specific information

The following example shows a typical DDESpy display of DDE message activity:

```
Task:0x8df Time:642402 hwndTo=0x38dc Message(Sent)=Initiate:
       hwndFrom=9224, App=0xc35d("Server")
       Topic=*
Task:0x94f Time:642457 hwndTo=0x2408 Message(Sent)=Ack:
       hwndFrom=9396, App=0xc35d("Server")status=c35d(fAck fBusy )
       Topic=Item=0xc361("System")
```

Monitoring Callbacks

The DDE protocol uses the **MONCBSTRUCT** structure to pass information to application callback functions. DDESpy displays the following information from this structure:

- Task
- Time
- Transaction type
- Exchanged-data format, if any
- Conversation handle
- String handles and their referenced strings
- Transaction-specific data

The following example shows a typical DDESpy display of callback activity:

```
Task:0x8df Time:2882628 Callback:
       Type=Advstart, fmt=0x1("CF_TEXT"), hConv=0xc24b4,
       hsz1=0xc361("System") hsz2=0xc4df("xxcall"), hData=0x0,
       lData1=0x83f0000, lData2=0x0
       return=0x0
```

Monitoring Errors

When an error occurs during a DDE transaction, the DDE protocol places the error value and associated information in a **MONERRSTRUCT** structure. DDESpy uses this structure to display the following information about the error:

- Task (the handle of the application that caused the error)
- Time
- Error value and name

Tracking Options

DDESpy can also display information about aspects of DDE communication in your Windows system:

- String handles
- Conversations
- Links
- Services

You can use the Track menu to specify which DDE activity DDESpy is to track. When you choose a command from the Track menu, DDESpy creates a separate window for the display of information in conjunction with the DDE functions. For each window created, DDESpy updates the displayed information as DDE activity occurs. Events that occurred prior to creation of the tracking window are not displayed in the tracking window.

DDESpy can sort the displayed information in the tracking window. If you select the heading for a particular column in the tracking window, DDESpy will sort the displayed information based on the column you selected. This can be useful if you are searching for a particular event or handle.

Tracking String Handles

Windows maintains a system-wide string table containing the string handles that applications use in DDE transactions. To display the system string table so that the string, the string handle, and the string usage count are shown, choose the String Handles command from the Track menu.

Tracking Conversations

To see a display of all active DDE conversations in your Windows system, choose the Conversations command from the Track menu. The Conversations window shows the service name, the current topic, and the server and client handles for each active conversation.

Tracking Links

To see a display of all active DDE advise loops, choose the Links command from the Track menu. The Links window shows the server name, topic, item format, transaction type, client handle, and server handle for every active advise loop in your Windows system.

Tracking Services

Server applications use the **DdeNameService** function to register with the DDE protocol. When the DDE protocol receives the **DdeNameService** function call, it adds the server name and an instance-specific name to a list of registered servers. To see a list of registered servers, choose the Services command from the Track menu.

The About Menu

Provides information about the version of DDESpy you are using.

Creating a Setup Program

When you install a Microsoft software program on your computer, you may be using Setup and its supporting functions to decompress and copy the program files onto your hard disk. Setup is a tool that you can use to create programs that will install your application on a user's computer. Microsoft is providing the Setup toolkit as part of the SDK so that you can take advantage of its many automated procedures when you create an installation program for your own product.

The Basic Components of Setup

The Setup toolkit includes the following basic components:

- Sample dialog box templates and procedures (DIALOGS.DLG, DIALOGS.RES, and DIALOGS.C), which you modify using the Windows Dialog Editor to create the dialog and message boxes you need for your installation.
- Five .DLL files that contain useful routines for detecting the hardware and software environment, managing dialog and message boxes, copying files, modifying .INI files, and performing other program management functions. These procedures are described in Setup Procedures. you will use them to create your Setup program.
- Disk Layout Utilities, DISKLAYT.EXE and DISKLAYT2.EXE, which you use to create the installation disks you will ship with your product.

Use these components as described in the next section to create your own Setup program. Using Setup for your product installations will ensure that the process is safe and efficient, and that the installation meets Windows programming standards.

Steps for Creating a Setup Program

Use the following procedure to create a Setup program for your product. The remainder of this section discusses each step in more detail.

{ewl msdncd, EWGraphic, group10538 0 /a "SDK.BMP"}

To create a Setup installation program

1. Identify the files that will be installed for your product.
2. Design the directory structure for those files.
3. Identify all user-defined parameters.
4. Design the dialog boxes you will need for the installation.
5. Create the code that will install your product's files.
6. Create the images for the installation disks using the Disk Layout Utilities.
7. Test your installation.

Step 1: Identify the Files That Will Be Installed for Your Product

Before you start modifying sample code and dialog box templates, it's a good idea to make a list of the files you will need to install. For each file that you will install, answer the following questions:

1. Is this file unique for your product, is it a shared resource, or is it a system resource? For example, a shared resource could be a language dictionary used by more than one product for your company. A system resource could be a TrueType font. If the file is a shared file or a system file, you will want your code to check to see if it already exists and whether it is currently in use before copying it onto the user's hard disk.
2. Can the user decide whether to install this file? For example, is the installation of tutorial files optional? If so, you'll need to design a dialog box that asks the user to choose the files to install.
3. If an older version of the file already exists, should you overwrite it or rename it? Or, if you want to delete it, is the older version of the file under a different filename? If so, you will want your code to remove it, as well as install its newer version.

Beside each filename on your list, make notations indicating the answers to these questions. These notations will help you later when you design dialog boxes or set the properties for each file with the Disk Layout Utilities.

Step 2: Design the Directory Structure for the Installable Files

Take the time now to sketch out the directory structure for your product by organizing the installable files into categories that make sense. For example, you might put all computer-based training files in one subdirectory and all font files in another. You may need to place some files in the Windows directory or in one of its subdirectories. On the other hand, one directory (with no subdirectories) may suffice for a product that has only a few files.

Step 3: Identify All User-Defined Parameters

Identify the dialog and message boxes you will need for your Setup program. What input does the user provide during installation? For example, will you store the user's name, the company name, and the product serial number in a file? Can the user decide which directory to use for installation? Can the user decide not to install some of the product files? You should also note whether you will allow network installations and, if so, how the installation process will differ when installing to a network drive.

Are there any issues that you need to communicate to the user? For example, do you need a message box to notify the user that you will be updating or deleting existing files? If so, note these as well.

Step 4: Design Dialog Boxes

Make a rough sketch of each dialog box and identify the controls (buttons, check boxes, and list boxes) that will be needed. This will help you choose the most appropriate template to modify. Then use the Windows Dialog Editor to customize the templates. You may also need to modify the dialog box procedures (in DIALOGS.C) to process the user's responses. For more information about this process, see Designing Dialog Boxes.

Step 5: Create Your Installation Program

Starting from the sample program provided in the toolkit, you will create your own Setup program. For more information, see Building a Setup Program. For descriptions of the functions used in these files, see Setup Procedures.

Step 6: Create the Images for the Installation Disks

Gather all of your product files and installation program files into the directory structure that you sketched out earlier. Then use the Disk Layout Utilities to define each file's properties (such as whether it can be put on a writable disk) and to create the images for the installation disks. The Disk Layout Utilities automatically create the .INF file as part of this process.

You can use the Disk Layout Utilities throughout your software development project, until you create your master disks. Each time you release another version of your product, use the Disk Layout Utilities to update the .INF file and disk images.

For more information about the Disk Layout Utilities, see Using the Disk Layout Utilities.

Step 7: Test Your Installation Program

Once you've created your own Setup installation program, test it thoroughly. Test your code under a variety of situations and computer configurations. Check the results by verifying that the files were copied to their appropriate directories and that system files were updated correctly. When you are satisfied that the installation is correct, create your master disks by copying the disk images onto media that you've chosen to deliver your product.

Designing Dialog Boxes

The Setup toolkit includes dialog box templates that you can customize to meet your installation's specific needs. You can use the Microsoft Windows Dialog Editor (DLGEDIT.EXE) to edit the templates.

The Dialog Editor is a tool that lets you design and test a dialog box on the display screen instead of defining dialog statements in a resource script. Using the Dialog Editor, you can add, modify, and delete controls in a dialog box. The Dialog Editor saves the changes you make as resource script statements. You then compile these statements into a binary resource file that is linked to your Setup application's executable file of dialog procedures.

The Setup toolkit provides the following files that contain sample dialog box templates and procedures:

- DIALOGS.DLG, which contains dialog box templates and is used to create MSCUISTF.DLL. See the table below for descriptions of these templates. DIALOGS.DLG is updated automatically when you use the Dialog Editor to read the companion file that the editor creates, DIALOGS.RES.
- DIALOGS.RC, which contains the resource statements for the bitmaps and the icons that are used in the DIALOGS.DLG and DIALOGS.H sample files.
- DLGPROCS.C, which contains the C code for sample dialog box procedures associated with each template and is used to create MSCUISTF.DLL. You modify this file to update the procedures for the dialog box templates that you edited using the Dialog Editor. You can also add new dialog procedures to this file.
- DIALOGS.H, which contains the dialog control identification number definitions and is used to create MSCUISTF.DLL. This file is updated automatically when you use the Dialog Editor.
- CUI.H, which is the header file for the Setup toolkit dialog box C functions.
- MAKEFILE, which you can use to compile the preceding files.

Use these files with the Dialog Editor, the C compiler, and the linker to create your own dialog boxes.

To customize the dialog box templates for your installation program, follow these steps

1. Use the Dialog Editor to modify DIALOGS.RES. For each dialog box that you need, choose a template that closely resembles the design of the dialog box and modify it as necessary.
2. When you save your changes, the Dialog Editor updates the script statements in DIALOGS.DLG and the constants in DIALOGS.H.
3. If necessary, edit DLGPROCS.C to update the dialog box procedures for the templates that you modified.

   The Setup toolkit provides a set of functions that you can use in the dialog procedures in addition to the standard Windows functions. These Setup functions are written in C and are described in detail in the following section.

Note DLGPROCS.C uses the Symbol Table, a temporary storage area in memory, to transfer information between dialog box procedures and Setup. The comments embedded in the code for each dialog box procedure identify the symbols the procedure uses. For more information about the Symbol Table, see Building a Setup Program.

Note Two constants are defined in CUI.H that can directly affect dialog box procedures: STF_REINITDIALOG and STF_ACTIVATEAPP. If your dialog box procedure has called the **UIStartDlg** function for a dialog box that is already on top of the dialog box stack (that is, to update the contents of the dialog box), Windows returns the STF_REINITDIALOG constant to let you know. If the user has switched to another application during your installation, Windows returns the STF_ACTIVATEAPP constant to let you know that control has returned to your Setup program.

3. Compile the dialog box procedures with MAKEFILE to create the .DLL file with your changes.

MAKEFILE compiles the dialog procedures and dialog resources into MSCUISTF.DLL. You then use the name of the .DLL file, the resource identification numbers of the dialog boxes, the help description resource identification numbers, and the names of the associated dialog box procedures as parameters for the **UIStartDlg** procedure, which is called from the installation program.

Note All dialog boxes must have a style of WS_CHILD in order to run properly.

The following table provides a quick guide to the dialog box procedures provided in DLGPROCS.C and their associated templates provided in DLGPROCS.DLG. Each procedure in DIALOGS.C is preceded with comments indicating what the procedure does and what symbols it uses. You can also preview each template using the Dialog Editor to open DIALOGS.RES.

Dialog Box Procedures

Procedure name	Used for	Associated template(s)
FCheckDlgProc	Dialog boxes with one to ten check boxes.	CHECK
FCustInstDlgProc	Dialog boxes with one to ten check boxes. Each check box can have an associated push button. This procedure also supports a push button that displays the current installation path and allows the user to change it.	CUSTINST
FEditDlgProc	Dialog boxes that contain one edit control.	DESTPATH
FHelpDlgProc	Dialog boxes that contain help messages.	APPHELP
FInfoDlgProc	Dialog boxes that present information to the user. The user must respond.	WELCOME
FInfo0DlgProc	This procedure is the same as FInfoDlgProc but does not support an Exit button.	BADPATH EXITFAILURE EXITQUIT EXITSUCCESS TOOBIG
FListDlgProc	Dialog boxes that contain one single-selection list box.	SINGLELIST
FModelessDlgProc	Dialog boxes that present information to the user during lengthy operations. The user does not have to respond.	MODELESS
FMultiDlgProc	Dialog boxes with one	EXTENDEDLIST

	multiple-selection list box.	MULTILIST
FQuitDlgProc	Dialog boxes that let the user either quit or resume the installation process.	ASKQUIT
FRadioDlgProc	Dialog boxes with a single group of one to ten radio buttons.	OPTIONS

For information about the **UIStartDlg** procedure, see Setup Procedures. For information about using the Dialog Editor, refer to "Designing Dialog Boxes: The Dialog Editor" in *Microsoft Windows Programming Utilities.*

Dialog Box Functions

AsciiToInt

int AsciiToInt(LPSTR *sz*);

The **AsciiToInt** function converts an ASCII string representing a positive value into an integer.

Argument

sz
A non-null zero-terminated string to convert.

Return Value

Returns the integer that is represented by *sz*.

Comments

The integer to convert must be both positive and less than 100.

See Also

IntToAscii

AssertSzUs

void AssertSzUs(LPSTR, unsigned *fValue***);**

The **AssertSzUs** function asserts whether a boolean expression is true when the DEBUG compiler flag is defined. When DEBUG is not defined, the function is ignored.

Argument

fValue
Specifies the Boolean value that you want to assert.

Return Value

This function has no return value.

Comments

If the asserted value is true, the **Assert** function simply returns. If the asserted value is false, the program displays a message box containing the source filename and the line number of the failed assertion. You must click OK to continue.

CbGetListItem

int CbGetListItem(LPSTR *szListSymbol*, **unsigned** n, **LPSTR** *szListItem*, **unsigned** *cbMax*);

The **CbGetListItem** function copies the specified list item into the provided buffer as a zero-terminated string.

Arguments

szListSymbol
Specifies the name of the symbol whose associated value is the list you want.

n
Specifies the index number (one-based) for the list item you want to copy into the buffer.

szListItem
Specifies the buffer for the copy of the list item.

cbMax
Specifies the size of the buffer.

Return Value

If the function is successful, the return value is the length (in bytes) of the full string of the specified list item. If *szListSymbol* or *n* doesn't exist, the return value is zero and the empty string is placed in the buffer.

Comments

If you specify a buffer that is smaller than the length of the symbol value, the **CbGetListItem** function will copy in as many characters as will fit (including a trailing zero). However, the return value will be the full length of the string.

See Also

UsGetListLength, **FAddListItem**, **FReplaceListItem**. For information about setting symbol values in the Symbol Table, see **FSetSymbolValue**.

CbGetSymbolValue

int CbGetSymbolValue(LPSTR *szSymbol***, LPSTR** *szValue***, unsigned** *Length***);**

The **CbGetSymbolValue** function copies the specified value from the symbol-value pair in the Symbol Table into a buffer.

Arguments

szSymbol
Specifies the name of the symbol whose value you want to copy into the buffer.

szValue
Specifies a buffer for the value associated with the symbol.

Length
Specifies the length, in bytes, of the buffer.

Return Value

If the function is successful in copying the value into the buffer, the return value is the length of the value string (excluding the terminating zero). If the symbol does not exist or is an empty string, the return value is zero.

Comments

If you specify a buffer length that is smaller than the length of the value, the function will copy in as many characters as will fit (including a trailing zero). However, the return value will be the full length of the specified value.

See Also

FSetSymbolValue, **FRemoveSymbol**

DoMsgBox

int DoMsgBox(LPSTR *lpText*, **LPSTR** *lpCaption*, **unsigned** *wType*);

The **DoMsgBox** function launches a Windows message box of the style specified by *wType*. The return value is the identification for the user's response, such as IDOK.

Arguments

lpText
Specifies the text you want to appear in the message box.

lpCaption
Specifies the caption for the message box.

wType
Specifies the contents of the message box. *wType* can be a combination of values.

Return Value

The return value is the value of the button control that the user selected (such as IDOK). If there is not enough memory to create the message box, the return value is zero.

Comments

This function is similar to the Windows **MessageBox** function. The valid message box values and control values are the same as for the **MessageBox** function.

See Also

For more information on the **MessageBox** function, see the Microsoft Windows Programmer´s Reference.

FAddListItem

BOOL FAddListItem(LPSTR *szListSymbol*, **LPSTR** *szListItem*);

The **FAddListItem** function adds the specified item to the end of the list of items associated with the symbol in the Symbol Table.

Arguments

szListSymbol
Points to a zero-terminated string that identifies the symbol.

szListItem
Points to a zero-terminated string that identifies the item you want to add to the list associated with *szSym*.

Return Value

If the function is successful in adding the item, the return value is **fTrue** (one). Otherwise, the return value is **fFalse** (zero).

Comments

You can initialize an empty list by setting its associated symbol value to "" with the **FSetSymbolValue** function. You can then add values to the list using **FAddListItem**.

See Also

FReplaceListItem, **CbGetListItem**, **UsGetListLength**

FCloseHelp

BOOL FCloseHelp();

The **FCloseHelp** function closes the currently open help dialog box, if one exists.

Return Value

The return value is **fTrue** (one) if the help dialog box is successfully closed. Otherwise, it is **fFalse** (zero).

See Also

HdlgShowHelp

FHandleOOM

BOOL FHandleOOM();

The **FHandleOOM** function displays a message box when an "Out Of Memory" error occurs and waits for a user response. This function lets the user switch out of the current application and free up some memory by closing other applications.

Return Value

If the user presses the RETRY button, the return value is **fTrue** (one). If the user presses the ABORT button, the return value is **fFalse** (zero).

FRemoveSymbol

BOOL FRemoveSymbol(LPSTR *szSym***);**

The **FRemoveSymbol** function removes a symbol and its associated value from the Symbol Table.

Argument

szSym
Specifies the name of the symbol you want to remove.

Return Value

If the function is successful, the return value is **fTrue** (one). Otherwise, the return value is **fFalse** (zero).

See Also

FSetSymbolValue

FReplaceListItem

BOOL FReplaceListItem(LPSTR *szListSymbol***, unsigned** *n***, LPSTR** *szListItem***);**

The **FReplaceListItem** function replaces the specified item in the list of items associated with the symbol in the Symbol Table.

Arguments

szListSymbol
Specifies the name of the symbol. *szSym* must be a zero-terminated string.

n
Specifies the index number (one-based) of the item you want to replace.

szListItem
Identifies the item you want to use to replace the existing item. *szItem* must be a zero-terminated string.

Return Value

If the function successfully replaces the item, the return value is **fTrue** (one). If the index is invalid or the application is out of memory, the return value is **fFalse** (zero).

See Also

FAddListItem, **CbGetListItem**, **UsGetListLength**

FSetSymbolValue

int FSetSymbolValue(LPSTR *szSymbol***, LPSTR** *szValue***);**

The **FSetSymbolValue** function inserts a new symbol-value pair into the Symbol Table. If the symbol already exists, the function replaces the symbol's associated value.

Arguments

szSymbol
Specifies the name of the symbol you want to create or whose associated value you want to replace.

szValue
Specifies the value you want to add or replace. If *szValue* is NULL, an empty string is added or used to replace the existing value.

Return Value

If the function is successful, the return value is **fTrue** (one). If the application is out of memory, the return value is **fFalse** (zero).

See Also

CbGetSymbolValue, **FRemoveSymbol**

HdlgShowHelp

HWND HdlgShowHelp();

The **HdlgShowHelp** function displays the help dialog box for the dialog box that is currently on the top of the dialog box stack.

Return Value

The return value is the handle to the help dialog. If the help dialog does not exist and cannot be created, the return value is NULL.

See Also

FCloseHelp

IntToAscii

int IntToAscii(int *i,* **LPSTR** *sz***);**

The **IntToAscii** function converts a positive integer (< 100) into a string that represents its value.

Arguments

i
The integer to convert.

sz
The buffer to hold the converted string (at least 3 bytes).

Return Value

Returns the ASCII value of the provided integer (*sz*).

Comments

The integer to convert must be both positive and less than 100.

See Also

AsciiToInt

ReactivateSetupScript

void ReactivateSetupScript();

The **ReactivateSetupScript** function returns control to the Setup program.

Return Value

This function has no return value.

Comments

This function is the vehicle for returning from a dialog box procedure to the Setup program.

SzDlgEvent

LPSTR SzDlgEvent(WORD *wParam***);**

The **SzDlgEvent** function gets the string values for the following WM_COMMAND events:

IDC_B, IDC_C, IDC_X, and IDCANCEL.

Argument

wParam
Event parameter value.

Return Value

Returns a pointer to string value constant. If the event is unknown, **SzDlgEvent** returns NULL.

See Also

SzLastChar

SzLastChar

LPSTR SzLastChar(LPSTR *sz*);

The **SzLastChar** function finds the last character in a string.

Argument

sz
Specifies the non-null zero-terminated string to search.

Return Value

Returns NULL for an empty string. If the string is not empty, **SzLastChar** returns a non-Null string pointer to the last valid character in *sz*.

See Also

SzDlgEvent

UsGetListLength

int UsGetListLength(LPSTR *szSymbol***);**

The **UsGetListLength** function determines the number of items in the list associated with the specified symbol.

Argument

szSymbol
Specifies the name of the symbol. *szSym* must be a zero-terminated string.

Return Value

The return value is the number of items in the list associated with the symbol.

See Also

CbGetListItem, **FReplaceListItem**

Building a Setup Program

Once you've identified the list of files you need to install and used the Windows Dialog Editor to design dialog boxes and message boxes, you're ready to create an C program that contains the commands and procedure calls that Setup uses to install your product on the user's system.

Start by reviewing the files that make up the Setup toolkit. The following table lists the name and purpose of each file in the toolkit.

Setup Toolkit Files

Filename	Description
SETUP.EXE (required)	The program that you write to copy your files to the user's hard disk.
Setup .DLL files: (required)	These .DLL files contain the code for the Setup procedures that you call from your installation program. You should list them in the SETUP.LST file and include them on the first installation disk.
MSCOMSTF.DLL	The common library, which contains supporting routines for the other .DLL files.
MSDETSTF.DLL	The detection library, which contains procedures that return information about the user's system, such as the version of Windows.
MSINSSTF.DLL	The install library, which contains procedures that install files on the user's hard disk.
MSUILSTF.DLL	The user interface library, which contains procedures that manipulate the user interface, such as displaying a dialog box or deleting a dialog box from the dialog stack.
MSSHLSTF.DLL	The shell library, which contains the routines that manage the frame window.
README.TXT	A text file that contains information about Setup that could not be included in this documentation. You should read this file before creating an installation program. Do not include this file on your installation disks.

The rest of this section explains the process of creating an installation program and defining which files you need to include on the first installation disk.

Modifying the Sample Installation Program

At this point, you have defined the list of files you will be installing and designed the dialog boxes you will need. Your focus has been on identifying the choices you will ask the user to make – for example, determining the directory in which you will install your product.

To create the installation program, your focus must now shift slightly. When you create the code that handles the dialog boxes and installs the product files, you must pay attention to designing a safe and efficient installation process. Therefore, before you start modifying the Setup sample program, answer these questions:

- How much disk space wnill the installation use? Is it a significant amount? If so, you will need to ensure that the user's hard disk has enough disk space available. Also, if your product includes optional files that the user has chosen, you may want to inform the user how much disk space each optional file will use.
- Does your product have minimum hardware and system software requirements? If so, you may want to check the version of your operating system, or check for the existence of, for example, a math coprocessor. If the user's system falls below your minimum requirements, you will want to display a warning message.
- Will you be installing shareable files? For example, does your product require a spelling dictionary that the user already may have installed with another product? If so, you will need to take some special steps to handle these files.
- Will you be installing any system files? If so, you may need to use special Setup functions to exit Windows and update files that would be in use while Windows is running.
- What other parts of your installation have associated risks? Do you need to post warnings of any kind for the user? Do you need to check the validity of paths and filenames the user enters?

The answers to these questions provide the information you need to design your installation code. You should design the installation program just as you would any other program: Identify the types of routines you will need, determine the logical order in which those routines should occur, and then draw a flowchart of the installation process.

Using the Symbol Table

Whether your installation is simple or complex, you will probably use the Symbol Table to store values. The Symbol Table is a temporary storage area in memory that contains a table of text symbols and their associated text values. Setup uses the Symbol Table to store information such as directory names and data that is passed between the installation program and the .DLL files.

Setup automatically creates and sets three symbols that you can use:

- STF_SRCDIR, which is the source directory
- STF_CWDDIR, which is the current working directory or the temporary directory for Setup
- STF_SRCINFPATH, which is the path for the .INF file (usually empty, unless you or the user supplied it as part of the SETUP.EXE command line)

The procedures in MSCUISTF.DLL (the customized dialog box routines) also use the Symbol Table to store the user's responses.You'll see other uses of the Symbol Table interspersed throughout the sample code. Use the Symbol Table to pass data between your installation program and the .DLL files.

Note To conserve memory, clear all Symbol Table strings after you use them.

Installing Shared Files or System Files

If you are planning to install shared files or system files, you must handle that part of the installation with extra care. This section describes some of the issues involved with installing shared files or system files and how Setup handles them.

Shared Files

A shared file is a file that may be used by more than one application on the user's system. For example, your company may have two products that use the same spelling dictionary. If the user has already installed one of the products, that dictionary file may already be installed. Furthermore, if the user has that product running during the installation process, the dictionary file may be in use, in which case it can't be updated.

To handle this problem, the **CopyFilesInCopyList** procedure checks each file listed in your .INF file with the SHARED attribute to see if the file is in use. If so, Setup displays an error message. The user can fix the problem by:

- Switching out of Setup, closing the other application, switching back into Setup, and then choosing the Retry button.
- Exiting Setup and rerunning the installation after closing the other application.
- Ignoring the message. If the file is marked as vital, Setup will display another error message. If the file is not vital, Setup skips it but continues to copy other files in the list.

System Files

A system file is a file that may be in use by Windows when Windows is running. The **CopyFilesInCopyList** procedure checks each file listed in your .INF file with the SYSTEM attribute to see if the file is in use. If so, the procedure copies the file to a temporary location (the restart directory) and adds a command to the _MSSETUP.BAT file.

Toward the end of the installation, your program should call the **RestartListEmpty** function. If the function returns zero, there are system files that must be updated. You should inform the user about this and then use the **ExitExecRestart** function to shut down Windows, update the files, and restart Windows.

For more information, see **CopyFilesInCopyList**, **RestartListEmpty**, and **ExitExecRestart**, and Setup Procedures.

Using the Disk Layout Utilities

The Disk Layout Utilities automate tedious, error-prone tasks by taking your project files and creating efficiently organized disk images for your product installation. As part of this process, the Disk Layout Utilities also creates the .INF file.

Typically, you will use the Disk Layout Utilities in the following manner:

- As soon as you have a distributable product release, even if it is planned for internal release within your company, use the Disk Layout Utilities to create the .INF file and the disk images.
- For each subsequent release, run the Disk Layout Utilities to update the directory of disk images for any files you may have added or changed.

After the first time you use the Disk Layout Utilities to create a software release, the program remembers which files have already been included and tells you if you have added any new files. The Disk Layout Utilities also update disk images and compressed files only when they change.

This chapter describes the disk layout process and explains the use of the Dsklayt and Dsklayt2 programs to create a layout file, disk images, and the .INF file.

Understanding the Disk Layout Process

The Disk Layout Utilities consist of two parts:

- A Windows-based program (Dsklayt) that you use to specify the properties for all files that will go into your product release
- An Win32-based program (Dsklayt2) that creates the disk images and the .INF file for the installation

You use Dsklayt to create a layout file containing file specifications. Dsklayt2 then uses the directives in the layout file to create the disk images and the .INF file. You can use Dsklayt2 as part of your product build process.

{ewl msdncd, EWGraphic, group10542 0 /a "SDK.BMP"} To run Dsklayt

1. In File Manager, choose Run from the File menu.
2. In the Command Line box, type \MSTOOLS\MSSETUP\DISKLAY\DSKLAYT, and then click OK.
   The main window for Dsklayt appears with an open dialog box.
3. In the Microsoft Disk Layout Utilities dialog box, click either Open Layout to open an existing layout file or New Layout to create a new layout file.
4. If you are opening an existing layout file, specify the name of the file and click OK. If you are creating a new layout file, specify the directory where your product files are stored and click OK.

The files from the source directory appear in the list box on the left side of the main window. You can then select one or more files from the list box and specify their properties.

Using the Disk Layout Utilities Commands

Dsklayt has a main window and three menus – File, Options, and Help. This section describes the contents of the main window and each of the commands on the menus.

Dsklayt Main Window

You can use the options in the main window to set most of the file specifications for your installation. To set specifications for one file or for a group of files, you simply select the file(s) you want to affect from the list box and specify the options you want the file(s) to have.

The Dsklayt main window options are divided roughly into two types: *layout time* options, which affect how the files are stored on the installation disks, and *install time* options, which affect how the files are copied onto the user's hard disk. Each option is described below.

Source Directory
Displays the name of the top-level directory for your product files.

List box
Displays all the files in the source directory and its subdirectories. Choose files from this list to set their attributes. You can choose:

- A single file, by clicking on it.
- A contiguous range of files, by clicking on the first file, holding down the SHIFT key, and then clicking on the last file.
- A discontiguous range of files, by holding down the CTRL key and clicking on each file.

Just Show New
Checking this box displays only the files that are new or have been updated since you last created the layout file. Use this option when you are doing successive product releases and only need to add specifications for the new or changed source files.

Layout Time Options:

File Destination
Determines the type of disk on which the selected file can be placed. Choose one of five options:

1. **Any Diskette:** Indicates that the selected file can go on any diskette in the installation set. This option is the default.
2. **Writable Diskette:** Indicates that the selected file must go on a disk that Setup can write to.
3. **Read-Only Diskette:** Indicates that the selected file must go on a write-protected disk. For example, you may want to store uncompressed binary files (.EXE files) that might be targets of viruses on a read-only diskette.
4. **Setup Diskette (#1):** Indicates that the selected file must go on the first disk in the installation disk set. For example, you would choose this option for your Setup program.
5. **Do Not Lay Out File:** Indicates that the selected file should not be placed on an installation disk. Use this option for files that reside in your project directories but are not part of the product installation, such as source code management files.

File Attributes
Marks a file in the layout file as having one or more attributes. Check one or more of the following:

- **System File:** The file is a system file, such as WINHELP.EXE or GDI.EXE.
- **Shared File:** The selected file may be shared by one or more applications, such as a common code library that ships with all of your company products.
- **Vital File:** The installation will fail unless the selected file is installed successfully. This option is the default.

Set File Date
Specifies the date stamp used for the file when it is copied into a disk image directory by Dsklayt2 and when Setup copies the file onto the user's hard disk. Choose one of the following options:

- **Source Date:** Uses the date of the installable file.
- **Other:** Uses the date you specify in the adjacent text box (in the format YYYY-MM-DD). Use this option when you want the date on all installed files to be a significant date, such as the product

release date.

Compress

Determines whether the selected file should be compressed by Dsklayt2.

Check For Version

Tells Dsklayt2 to check the source file for the existence of a version resource. If the version resource exists, Dsklayt2 puts this information into the .INF file. Otherwise, Dsklayt2 issues a warning and leaves this portion of the file description blank in the .INF file.

Reference Key

Specifies a unique reference for the selected file. Use this option when you want the Setup program to determine whether to install the selected file based on information available at the time of the installation. For example, the type of monitor on the user's system could affect the files you install for your product. You can also use this option when you want to display reference keys rather than filenames in the dialog boxes displayed by the installation, because the keys are more descriptive than the filenames.

Put In Section

Specifies a unique .INF section name for the selected file. Use this field when you want installation files organized by categories rather than all listed in the default "Files" section of the .INF file.

Install-Time Options:

Overwrite

Specifies what should happen if the selected file already exists on the user's hard disk. Choose one of the following options:

- **Always:** The installed file will always overwrite any existing version of the file.
- **Never:** The installed file will never overwrite an existing version of the file.
- **Older:** The installed file will overwrite an existing version of the file only if the existing version is older. Setup will look for version information; if none exists, it will use the file dates to determine which file is older.
- **Unprotected:** The installed file will overwrite an existing version of the file only if the existing version has a file attribute of "Write."

Decompress

Indicates that Setup should check to see if the source file is compressed and, if so, decompress it before copying it onto the user's hard disk. You should leave this option checked in most cases, even if the source file is not compressed.

Mark as Read Only

Indicates that you want the file to have a file attribute of "Read Only" when it is copied onto the user's hard disk.

Rename Copied File

Indicates that you want to rename the file to the filename you supply in the adjacent text box when it is copied onto the user's hard disk.

Backup Existing File

Indicates that you want to back up an existing version of the file to the filename you supply in the adjacent text box before copying the source file. If you type an asterisk (*), Setup will back up the file to the same name with a .BAK extension.

File menu

New

Creates a new, untitled layout file.

Open

Displays a dialog box that you can use to open an existing layout file so that you can update it. Dsklayt checks for new product files and notifies you if there are any.

Save

Saves any changes you have made to the currently open layout file.

Save As

Displays a dialog box that you can use to save the current layout file under a name you specify.

Exit

Exits Dsklayt and returns you to the most recently active window. If you have made changes but did not save them, Dsklayt prompts you to save.

Options menu

Disk Labels

Displays a dialog box that lets you add, delete, or modify the disk labels for your installation disks.

To insert a new label, select the label in the list box that you want to follow the new label, type the new label name in the text box, and then click Add. To delete a label, select it from the list box and then click Delete. To modify a label, select it from the list box, click Delete, type the correct text in the text box, and then click Add.

You can use generic labels, such as "Disk 1," until you see how the files are organized on the disks. You can then rename the labels to reflect the content of the disks.

You can specify which disk label goes on the writable disk by selecting the label in the drop-down list box at the bottom of the dialog box.

Note The order in which you add disk labels is the order in which Dsklayt2 will apply them to the disk images.

Remove Files List

Displays a dialog box in which you can create one or more lists of files that you want Setup to remove from the user's hard disk during product installation. Use this list to remove obsolete files or older versions of files whose names differ from the newer versions.

To add a file to a removal list, type its filename in the Filename box, type the name of the .INF section in the .INF Section box (if necessary), and then click Add. If you don't type a section name in the .INF Section box, Dsklayt2 adds the name of the file to the "Files" section in the .INF file.

To delete a file from a removal list, select it from the list box and click Delete. To modify a filename in a removal list, select it from the list box, click Delete, type the correct filename in the Filename box, and then click Add.

Help menu

About

Displays a dialog box that provides copyright and version information for Dsklayt.

Using the Dsklayt2 Program

After saving your specifications in a layout file, use the Dsklayt2 program to generate disk images and the .INF file for your product installation. Dsklayt2 reads the directives in the layout file and creates a directory of disk images that you can copy onto disks using the Diskcopy command.

The Dsklayt2 program has the following command line syntax:

Dsklayt2 *[drive:][path]layout_filename [[drive:][path]INF_filename]* *[options]*

Parameter	Description
drive:path for layout_filename	The drive letter and path of the layout file.
layout_filename	The name of the layout file that Dsklayt2 should read to create the disk images. This argument is required.
drive:path for .INF_filename	The drive letter and path of the .INF file.
INF_filename	The name of the .INF file that you want Dsklayt2 to create. If you do not specify a filename, Dsklayt2 uses SETUP.INF.
Options:	Specifies the type of disk Dsklayt2 should target when creating the disk images. For *n,* you can specify: • A standard size ("360," "720," "12," or "144"), "N" for Network • O *n/m* for Other, where *n/m* is the bytes per cluster and the cluster per disk, respectively. The default is 1.2 MB.
/k{*n*}	
/f	Specifies that Dsklayt2 should overwrite the existing .INF file, if necessary.
/w{*n*}	Specifies the writable disk. For *n,* specify a disk number. This field overrides the specification in the layout file. If you omit this option, Dsklayt2 makes the last disk in the installation the writable disk. If you specify the option without a disk number, Dsklayt2 assumes that all disks are read-only.
/d*{destdir}*	Specifies the destination directory for the disk images. Dsklayt2 creates a directory for each disk image, named DISK 1, DISK 2, and so on. If your product will be installed from a network drive (that is, if the /k option specifies a network device), all files are placed at the top-level destination directory. If you omit this option, Dsklayt2 will not create any files in the destination directory.
/c*{compdir}*	Specifies the directory where Dsklayt2 can put compressed versions of the files. If you specify the same directory each time you run Dsklayt2, the program adds or updates only those files that are new or have changed. If you omit this option, Dsklayt2 creates a COMP subdirectory in the parent of the

	source directory. If the source directory is the root directory, Dsklayt2 displays an error message and aborts.
/z*{compcmd}*	Specifies a compression utility that Dsklayt2 can use to compress files. For *compcmd,* specify the operating system command that will execute the compression utility. Dsklayt2 will call this command with two arguments: the source directory and the destination directory. If this option is not specified, Dsklayt2 looks for COMPRESS.EXE in the path. (To specify internal compression only use */zi*.)

Setup Procedures

This chapter describes the functions and subroutines that you can call in your installation program. They are listed in alphabetical order.

Note All functions and subroutines are declared in the Setup .C files. The descriptions in this chapter show the calling syntax for each procedure.

To:	Use these procedures:
Manipulate what the user sees on the screen	DoMsgBox InitSetupToolkit RestoreCursor SetBeepingMode SetBitmap SetCopyGaugePosition SetSilentMode SetTitle ShowWaitCursor UIPop UIPopAll UIStartDlg
Manipulate a list associated with a symbol in the Symbol Table	AddListItem GetListItem GetListLength GetSymbolValue MakeListFromSectionKeys RemoveSymbol ReplaceListItem SetSymbolValue
Modify the contents of the global list of installable files (the copy list)	AddSectionFilesToCopyList AddSectionKeyFileToCopyList AddSpecialFileToCopyList ClearCopyList CopyFilesInCopyList DumpCopyList GetCopyListCost
Control aspects of the copy list installation	SetCopyMode SetDecompMode
Manipulate billboard dialog boxes and the global billboard list	AddBlankToBillboardList AddToBillboardList ClearBillboardList
Manipulate a file on the user's system	BackupFile CopyAFile DoesFileExist FindFileInTree FindFileUsingFileOpen FindTargetOnEnvVar GetDateOfFile GetSizeOfFile GetVersionNthField GetVersionOfFile HandleSharedFile InStr

	IsFileWritable PrependToPath RemoveFile RenameFile RightTrim StampResource SzCatStr SzCat2Str SzCat3Str
Manipulate a directory on the user's system	CreateDir RemoveDir DoesDirExist GetWindowsDir IsDirWritable
Update an .INI file	CreateIniKeyValue CreateSysIniKeyValue DoesIniKeyExist DoesIniSectionExist FValidIniFile GetIniKeyString GetNthFieldFromIniString RemoveIniKey RemoveIniSection
Create a Program Manager group and item for your product	CreateProgmanGroup CreateProgmanItem ShowProgmanGroup
Add information to or get information from the Registration Database	CreateRegKey CreateRegKeyValue DeleteRegKey DeleteAllSubkeys DoesRegKeyExist GetRegKeyValue SetRegKeyValue
Install system resources (that may be in use while Windows is running)	ExitExecRestart RestartListEmpty SearchForLocationForSharedFile SetRestartDir
Create a record of what occurred during an installation	CloseLogFile OpenLogFile SetAbout WriteToLogFile
Query the user's environment	FValidDrive GetConfigLastDrive GetConfigNumBuffers GetConfigNumFiles GetConfigRamdriveSize GetConfigSmartdrvSize GetDOSMajorVersion GetDOSMinorVersion GetEnvVariableValue GetFreeSpaceForDrive GetLocalHardDrivesList GetNetworkDrivesList

	GetNumWinApps GetParallelPortsList GetProcessorType GetRemovableDrivesList GetScreenHeight GetScreenWidth GetSerialPortsList GetTotalSpaceForDrive GetTypeFaceNameFromTTF GetValidDrivesList GetWindowsMajorVersion GetWindowsMinorVersion GetWindowsMode GetWindowsSysDir Has87MathChip HasMonochromeDisplay HasMouseInstalled IsDriveLocalHard IsDriveNetwork IsDriveRemovable IsDriverInConfig IsDriveValid IsWindowsShared
Parse a date field	GetDayFromDate GetHourFromDate GetMinuteFromDate GetMonthFromDate GetSecondFromDate GetYearFromDate
Retrieve information about the topmost frame window	HinstFrame HwndFrame
Read or manipulate information from the .INF file	FValidInfSect GetSectionKeyDate GetSectionKeyFilename GetSectionKeySize GetSectionKeyVersion MakeListFromSectionDate MakelistFromSectionFilename MakeListFromSectionKeys MakeListFromSectionSize MakeListFromSectionVersion ReadInfFile RemoveSymbol ReplaceListItem
Set and remove environment variable values.	SetEnvVariable RemoveEnvVariable
Uninstall an installation.	FStoreUninstallData FUninstall

AddBlankToBillboardList

VOID AddBlankToBillboardList(LONG *lTicks***);**

AddBlankToBillboardList adds a hidden dialog box to the global billboard list. The hidden dialog box destroys the previous billboard dialog box and delays the display of the next billboard dialog box.

Argument

lTicks
Defines the amount of time you want to delay the display of the next billboard dialog box. The unit is arbitrary, relative to a total number of units.

Comments

Use **AddBlankToBillboardList** prior to calling **CopyFilesInCopyList**.

AddListItem

int AddListItem(LPSTR *szListSymbol*, **LPSTR** *szListItem);*

AddListItem adds a new item to the end of the list associated with the specified symbol name.

Arguments

szListSymbol
Specifies the name of the symbol in the Symbol Table to which the list is associated.

szListItem
Specifies the item that you want to add to the list.

Comments

If *szListSymbol* is previously undefined, a new list with the specified item is created and associated with the symbol name. You can create a new, empty list by using **SetSymbolValue** and specifying the value as " ".

AddSectionFilesToCopyList

VOID AddSectionFilesToCopyList(LPSTR *szSect*, **LPSTR** *szSrc*, **LPSTR** *szDest*);

AddSectionFilesToCopyList adds all file descriptions from the specified section of the .INF file to the global list of installable files (the copy list).

Arguments

szSect
Specifies the name of the section in the .INF file that contains the files you want to add to the copy list.

szSrc
Specifies the full path of the directory where the files currently reside. Typically, you use the value associated with the symbol STF_SRCDIR for *szSrc.*

szDest
Specifies the full path of the directory to which the files will be copied.

Comments

You must call **ReadInfFile** before using **AddSectionFilesToCopyList**.

AddSectionKeyFileToCopyList

VOID AddSectionKeyFileToCopyList(LPSTR *szSect*, **LPSTR** *szKey*, **LPSTR** *szSrc*, **LPSTR** *szDest*);

AddSectionKeyFileToCopyList adds a file description identified by the reference key from the .INF file to the global list of installable files (the copy list).

Arguments

szSect
Specifies the name of the section in the .INF file that contains the file you want to add to the copy list.

szKey
Specifies the reference key for the file you want to add to the copy list.

szSrc
Specifies the full path of the directory where the file currently resides. Typically, you use the value associated with the symbol STF_SRCDIR for *szSrc$*.

szDest
Specifies the full path of the directory to which the file will be copied.

Comments

You must call **ReadInfFile** before using **AddSectionKeyFileToCopyList**.

AddSpecialFileToCopyList

VOID AddSpecialFileToCopyList(LPSTR *szSect*, **LPSTR** *szKey*, **LPSTR** *szSrc*, **LPSTR** *szDest*);

AddSpecialFileToCopyList adds the file description of a special file, such as a shared resource, from the .INF file to the global list of installable files (the copy list).

Arguments

szSect
Specifies the name of the section in the .INF file that contains the file you want to add to the copy list.

szKey
Specifies the reference key for the file you want to add to the copy list.

szSrc
Specifies the full path of the directory where the file currently resides. Typically, you use the symbol STF_SRCDIR for *szSrc$*.

szDest
Specifies the full path of the file to be copied.

Comments

You must call **ReadInfFile** before using **AddSpecialFileToCopyList**.

AddToBillboardList

VOID AddToBillboardList(LPSTR *szDll*, **INT** *idDlg*, **LPSTR** *szProc*, **LONG** *lTicks*);

AddToBillboardList adds a billboard dialog box to the end of the global billboard list. The dialog box will be displayed during the next **CopyFilesInCopyList** call.

Arguments

szDll
Specifies the name of the .DLL file that contains the dialog box resource and procedure.

idDlg
Specifies the dialog box resource identification number.

szProc
Specifies the name of the dialog box procedure.

lTicks
Defines the amount of time you want the billboard dialog box to display. The unit is arbitrary, relative to the total number of units specified at the time the files are copied onto the user's hard disk or network drive.

BackupFile

VOID BackupFile(LPSTR *szFullPath*, **LPSTR** *szBackup*);

BackupFile backs up the specified file by renaming it.

Arguments

szFullPath
Specifies the full path and name of the file you want to create a copy of.

szBackup
Specifies the filename of the copy.

Comments

The copy is placed in the same directory as the original file(as specified by *szFullPath*). **BackupFile** is identical to **RenameFile**.

ClearBillboardList

VOID ClearBillboardList(VOID);

ClearBillboardList deletes all dialog boxes from the global billboard list.

ClearCopyList

VOID ClearCopyList(VOID);

ClearCopyList removes all file entries from the global list of installable files (or copy list).

CloseLogFile

VOID CloseLogFile(VOID);

CloseLogFile closes the currently open log file.

CopyAFile

VOID CopyAFile(LPSTR *szFullPathSrc*, **LPSTR** *szFullPathDst*, **INT** *cmo*, **INT** *fAppend*);

CopyAFile copies the specified file from its source directory to its destination directory.

Arguments

szFullPathSrc
Specifies the full path of the file you want to copy.

szFullPathDst
Specifies the full path of the destination directory for the file.

cmo
Specifies one or more command option flags. You can use one or more of the following for *cmo* (by adding them together): *cmoDecompress, cmoTimeStamp, cmoReadOnly, cmoOverwrite, cmoNone,* or *cmoAll .*

fAppend
Specifies whether you want any existing file to be appended to. A value of one indicates that you want to append to an existing file; zero indicates that you want to remove the existing file before copying the new file.

CopyFilesInCopyList

VOID CopyFilesInCopyList(VOID);

CopyFilesInCopyList sorts the file descriptions in the global list of installable files (the copy list) and then copies them from their source directory to their destination directory.

Comments

The files are sorted by their source disk identification number to minimize the number of times the user has to insert disks during the installation. The source and destination directories are specified in the functions that add the files to the copy list.

CreateDir

VOID CreateDir(LPSTR *szDir*, **INT** *cmo*);

CreateDir creates a directory with the specified path and name.

Arguments

szDir

Specifies the complete path and name of the directory you want to create (starting with the disk drive letter and backslash).

cmo

Specifies a command option flag. You can use *cmoVital* or *cmoNone*. If the directory already exists, the subroutine does nothing.

CreateIniKeyValue

VOID CreateIniKeyValue(LPSTR *szFile*, **LPSTR** *szSect*, **LPSTR** *szKey*, **LPSTR** *szValue*, **INT** *cmo*);

CreateIniKeyValue creates a symbol and an associated value in the designated section of the .INI file.

Arguments

szFile
Specifies the name of the .INI file in which you want to create the symbol and value. If the file you specify is WIN.INI, you do not have to provide the full path. If the file you specify does not exist, it is created.

szSect
Specifies the section name in which you want to create the symbol and value. *szSect* must be a non-empty string.

szKey
Defines the name of the symbol you want to create. If *szKey* already exists, **CreateIniKeyValue** will fail unless you specify *cmoOverwrite* for the command option flag.

szValue
Defines the value that will be associated with the symbol.

cmo
Specifies the command option flag. You can use *cmoVital*, *cmoNone*, *cmoAll*, or *cmoOverwrite*.

CreateProgmanGroup

VOID CreateProgmanGroup(LPSTR *szGroup*, **LPSTR** *szPath*, **INT** *cmo*);

CreateProgmanGroup creates a new Program Manager group with the specified name.

Arguments

szGroup
Specifies the name of the Program Manager group you want to create. This name will be displayed in the group window title bar (or below the icon when the group window is minimized).

szPath
Specifies the name and path for the Program goup that you want to create. If you provide an empty string for *szPath* (the suggested method), a default file is created.

cmo
Specifies the command option flag. You can use *cmoVital* or *cmoNone*.

Comments

Typically, you should use an empty string for *szPath.*

CreateProgmanItem

VOID CreateProgmanItem(LPSTR *szGroup*, **LPSTR** *szItem*, **LPSTR** *szCmd*, **LPSTR** *szOther*, **INT** *cmo*);

CreateProgmanItem creates a new item in the specified Program Manager group.

Arguments

szGroup
Specifies the name of the Program Manager group in which you want to create the item. If *szGroup* does not exist, **CreateProgmanItem** does nothing.

szItem
Specifies the description that will be displayed below the item.

szCmd
Specifies the path and executable filename for the new item.

szOther
Specifies an optional icon file, icon resource index, *x* and *y* icon positions for the new item, and the working directory, separated by commas. If you provide an empty string for *szOther*, defaults are used. However, if you need to specify one of the latter options in the string, you must specify the preceding ones.

cmo
Specifies the command option flag. You can use *cmoVital* or *cmoOverwrite*. If you use *cmoOverwrite* and the user is running Windows version 3.1, Setup will replace an existing Program Manager item.

See Also

Command Option Flags, *Guide to Programming.*

CreateRegKey

void CreateRegKey(LPSTR *szKey***);**

CreateRegKey creates a Registration Database key that is a subkey of HKEY_CLASSES_ROOT.

Argument

szKey
Specifies the name of the key you want to create. This key will have no associated value.

CreateRegKeyValue

void CreateRegKeyValue(LPSTR *szKey*, **LPSTR** *szValue*);

CreateRegKeyValue creates a Registration Database key that is a subkey of HKEY_CLASSES_ROOT and associates a value with the key.

Arguments

szKey
Specifies the name of the key you want to create.

szValue
Specifies the value you want to associate with the key.

CreateSysIniKeyValue

VOID CreateSysIniKeyValue(LPSTR *szFile*, **LPSTR** *szSect*, **LPSTR** *szKey*, **LPSTR** *szValue*, **INT** *cmo*);

CreateSysIniKeyValue adds the specified symbol and its associated value to the .INI file.

Arguments

szFile
Specifies the full path of the .INI file. **CreateSysIniKeyValue** should not be used to modify WIN.INI.

szSect
Specifies the name of the section in which you want to add the symbol-value pair. *szSect$* must be a non-empty string.

szKey
Specifies the name of the symbol you want to add. This string does not have to be unique.

szValue
Defines the value you want to associate with the symbol.

cmo
Specifies the command option flag. You can use *cmoVital* or *cmoNone*.

Comments

Use **CreateSysIniKeyValue**, rather than **CreateIniKeyValue**, to add symbol-value pairs with non-unique keys to the .INI file. For example, you could add a line such as

DEV = *.VGA
where DEV = is likely to occur several times in the file.

DeleteAllSubKeys

BOOL DeleteAllSubKeys(HKEY *hKey*, **LPCTSTR** *szKey*);

DeleteAllSubkeys deletes a subkey and all of its subkeys.

Argument

hKey
Specifies the handle of an open key.

szKey
Address of the name of the subkey to delete.

DeleteRegKey

void DeleteRegKey(LPSTR *szKey***);**

DeleteRegKey removes the specified Registration Database key, its associated values, and subkeys.

Argument

szKey
Specifies the name of the key you want to remove.

DoesDirExist

BOOL DoesDirExist(LPSTR *szDir***);**

DoesDirExist determines if the specified directory exists.

Argument

szDir
Specifies the name of the directory.

Return Value

If the directory exists, the return value is one. Otherwise, the return value is zero.

DoesFileExist

BOOL DoesFileExist(LPSTR *szFile*, **int** *mode*);

DoesFileExist determines if the specified file exists, can be read from, can be written to, or all of these states.

Arguments

szFile
Specifies the name of the file you want to inquire about.

mode
Specifies the file exist mode (femExists, femRead, femWrite, femReadWrite) you want to inquire about.

Return Value

If the answer is yes, the return value is one. Otherwise, the return value is zero.

DoesIniKeyExist

BOOL DoesIniKeyExist(LPSTR *szFile***, LPSTR** *szSect***, LPSTR** *szKey***);**

DoesIniKeyExist determines if the specified key exists in the specified section of the .INI file.

Arguments

szFile
Specifies the name of the .INI file. If you specify WIN.INI, you do not have to provide the full path.

szSect
Specifies the section of the file. *szSect* must be a non-empty string.

szKey
Specifies the key (or symbol) you are looking for.

Return Value

If the key exists, the return value is one. Otherwise, the return value is zero.

DoesIniSectionExist

BOOL DoesIniSectionExist(LPSTR *szFile*, **LPSTR** *szSect*);

DoesIniSectionExist determines if the specified section exists in the .INI file.

Arguments

szFile
Specifies the name of the .INI file. If you specify WIN.INI, you do not have to provide the full path.

szSect
Specifies the section of the file. *szSect* must be a non-empty string.

Return Value

If the section exists, the return value is one. Otherwise, the return value is zero.

DoesRegKeyExist

int DoesRegKeyExist(LPSTR *szKey***);**

DoesRegKeyExist checks for the existence of the specified key in the Registration Database.

Argument

szKey
Specifies the name of the key. This key is assumed to be a subkey of HKEYS_CLASSES_ROOT.

Return Value

If the key exists, the return value is one. Otherwise, the return value is zero.

DoesSharedFileNeedCopying

BOOL DoesSharedFileNeedCopying(VOID);

DoMsgBox

int DoMsgBox(LPSTR *lpText***, LPSTR** *szCaption***, INT** *wType***);**

DoMsgBox launches a Windows message box containing the specified caption and text.

Arguments

szText
Specifies the text you want to appear in the message box.

szCaption
Specifies the caption for the message box.

wType
Specifies the contents of the message box. *wType* can be a combination of values.

Return Value

The return value is the value of the button control that the user selected (such as IDOK). If there is not enough memory to create the message box, the return value is zero.

Comments

DoMsgBox is similar to the Windows **MessageBox** function. The valid message box values and control values are the same as for **MessageBox**.

See Also

For more information on **MessageBox**, message box values, and control values, see the *Programmer's Reference.*

DumpCopyList

VOID DumpCopyList(LPSTR *szFile***);**

DumpCopyList prints the contents of the global list of installable files (the copy list) to the specified file.

Argument

szFile
Specifies the path and name of the file to which you want the list to be copied.

Comments

Use **DumpCopyList** when debugging your installation files.

EndSetupToolkit

VOID EndSetupToolkit(VOID);

ExitExecRestart

int ExitExecRestart(VOID);

ExitExecRestart installs system resources that may be in use by Windows (and therefore can't be overwritten).

Return Value

If the function succeeds, it doesn't return. The return value is true (one) if the restart list is empty or the user is running Windows version 3.0 (see comments below). The return value is false (zero) if there are write errors, the restart fails, Windows can't exit, or the function can't find _MSSETUP.EXE.

Comments

This function returns a value but can fail for one of three reasons: if the restart list is empty, if the user is running Windows version 3.0 (which does not support the function), or if an error occurs (such as Windows not exiting because of open MS-DOS boxes or applications). Therefore, you should use the **RestartListEmpty** function to determine the contents of the restart list before calling this function. You should also check the current version of Windows. If the user is running Windows version 3.0, you will need to provide a message box explaining that the user must exit Windows and run the batch file to update the shared resources. In addition, you should warn the user to close all MS-DOS boxes and applications.

ExitExecRestart exits Windows and executes _MSSETUP.EXE to read any commands that have been placed in _MSSETUP.BAT. (Commands are placed in _MSSETUP.BAT when a file in the copy list is specified as a system resource, Setup determines that it is a newer version, and it is currently in use.) **ExitExecRestart** will then delete _MSSETUP.EXE and _MSSETUP.BAT before restarting Windows. You must call **SetRestartDir** before using **ExitExecRestart**.

FindFileInTree

LPSTR FindFileInTree(LPSTR *szFile*, **LPSTR** *szDir*, **LPSTR** *szBuf*, **int** *cbBuf*);

FindFileInTree searches for the specified file in a directory and its subdirectories.

Arguments

szFile
Specifies the name of the file you are trying to locate.

szDir
Specifies the top-level directory of the tree structure you want to search.

szBuf
Points to the destination buffer that will receive the path string.

cbBuf
Length, in bytes, of the destination buffer.

Return Value

The return value is the full path of the first instance of the file. If the file isn't found, the return value is an empty string.

FindFileUsingFileOpen

LPSTR FindFileUsingFileOpen(LPSTR *szFile*, **LPSTR** *szBuf*, **INT** *cbBuf*);

FindFileUsingFileOpen locates a file by using the Windows **FileOpen** function.

Arguments

szFile
Specifies the name of the file you want to find.

szBuf
Points to the destination buffer that will receive the path string.

cbBuf
Length, in bytes, of the destination buffer.

Return Value

The return value is the full path of the file. If the file isn't found, the return value is an empty string.

FindTargetOnEnvVar

LPSTR FindTargetOnEnvVar(LPSTR *szFile*, **LPSTR** *szEnvVar*, **LPSTR** *szBuf*, **int** *cbBuf*);

FindTargetOnEnvVar searches for the specified file in directories based on the designated environment variable (such as PATH).

Arguments

szFile
Specifies the name or partial path of the file you are trying to locate.

szEnvVar
Specifies the environment variable for the search.

szBuf
Points to the destination buffer that will receive the string.

cbBuf
Length, in bytes, of the destination buffer.

Return Value

The return value is the full path of the specified file. If the file isn't found, the return value is an empty string.

FStoreUninstallData

BOOL FStoreUninstallData(LPSTR *szUninstallDataFile***);**

FStoreUninstallData stores information on the current installation. It stores the files, directories, environment variables, and Program Manager items and groups that were installed. This information is required for a subsequent call to **FUninstall**.

Argument

szUninstallDataFile
Specifies the name of the file in which to store the installation information.

Return Value

If the function succeeds, it returns **fTrue**.

If the function fails, it returns **fFalse**.

See Also

FUninstall

FUninstall

BOOL FUninstall(LPSTR *szUninstallDataFile*, **LPSTR** *szNewPath*);

FUninstall removes the files, directories, environment variables. and Program Manager items and groups created during installation.

Arguments

szUninstallDataFile
Specifies the name of the file that contains the installation data. This file was previously created by **FStoreUninstallData**, which should be run at the end of the initial setup.

szNewPath
Specifies a directory to place files that were found in the installation directory structure, but which were not part of the installation. If *szNewPath* is NULL, then these files will be left in the installation directory.

Return Value

If the function succeeds, it returns **fTrue**.

If the function fails, it returns **fFalse**.

See Also

FStoreUninstallData

FValidDrive

int FValidDrive(LPSTR *szDrive***);**

FValidDrive determines whether or not a specified pathname begins with a valid drive letter. For example, if the the first character is between a and z and the next character is either a NULL or a colon (:), the pathname is valid.

Arguments

szDrive
The specified pathname.

FValidInfSect

int FValidInfSect(LPSTR *szSect***);**

FValidInfSect determines if the specified .INF section name is valid.

Arguments

szSect
Specifies the .INF section name.

Return Value

Returns a 0 if the section name is invalid and a 1 if the section name is valid.

FValidIniFile

int FValidIniFile(LPSTR *szFile***);**

int FValidIniFile determines if the path to an .INI file is valid.

Arguments

szFile
Specifies the full path to the .INI file.

Return Value

Returns TRUE if the path is either valid or *szfile* equals WIN.INI.

GetConfigLastDrive

LPSTR GetConfigLastDrive(LPSTR *szBuf*, **int** *cbBuf*);

GetConfigLastDrive determines the LASTDRIVE set in the CONFIG.SYS file.

Arguments

szBuf
Points to the destination buffer that will receive the string.

cbBuf
Length, in bytes, of the destination buffer.

Return Value

The return value is the string for LASTDRIVE.

GetConfigNumBuffers

int GetConfigNumBuffers(VOID);

GetConfigNumBuffers determines the number of BUFFERS set in the CONFIG.SYS file.

Return Value

The return value is the number of BUFFERS.

GetConfigNumFiles

int GetConfigNumFiles(VOID);

GetConfigNumFiles determines the number of FILES set in the CONFIG.SYS file.

Return Value

The return value is the number of FILES.

GetCopyListCost

LONG GetCopyListCost(LPSTR *szExtraList*, **LPSTR** *szCostList*, **LPSTR** *szNeedList*);

GetCopyListCost examines the files listed in the global list of installable files (the copy list) and determines the amount of disk space needed to copy, back up, and overwrite files. The values retrieved are associated with symbols in the Symbol Table.

Arguments

szExtraList
Symbol whose associated value identifies the extra, or incidental, disk space needed on each disk drive to update files such as .INI files. The symbol value is a list of as many as 26 integers. Missing values are assumed to be zero.

szCostList
Symbol whose associated value is set to a list of 26 numbers, each of which identifies the cost per disk drive to copy and update files. A positive number is the amount of free space that will be used by new or larger files. A negative number is the amount of space that will be freed by removing files or replacing existing files with smaller versions.

szNeedList
Symbol whose associated value is set to a list of 26 numbers, each of which identifies the additional space needed per disk drive. Each entry in *szCostList* that is not zero has a corresponding entry in this list that is the value in *szCostList* minus the current free space on that disk drive. A positive number indicates that the new files will not fit (and by how much). A negative number indicates how much free space will be left after the new files are copied.

Return Value

The return value is the total additional free disk space needed. This value is the sum of the positive numbers in *szNeedList.* If there is enough free disk space on the appropriate disk drives for **CopyFilesInCopyList** to succeed, the return value is zero.

GetDateOfFile

LPSTR GetDateOfFile(LPSTR *szFile*, **LPSTR** *szBuf*, **int** *cbBuf*);

GetDateOfFile determines the file date of the specified file.

Arguments

szFile
Specifies the full path of the file.

szBuf
Points to the destination buffer that will receive the date string.

cbBuf
Length, in bytes, of the destination buffer.

Return Value

The return value is the date in YYYY-MM-DD-HH-MM-SS format. If *szFile* does not exist, or if it has an invalid date, the return value will be 1980-01-01-00-00-00.

GetDayFromDate

int GetDayFromDate(LPSTR *szDate***);**

GetDayFromDate retrieves the day field from the return value for **GetDateOfFile**.

Argument

szDate
Specifies the date in YYYY-MM-DD-HH-MM-SS format. This value is obtained from the **GetDateOfFile** function.

Return Value

The return value is an integer with a valid range from 1 through 31.

GetDOSMajorVersion

int GetDOSMajorVersion(void);

GetDOSMajorVersion determines the major version number of the currently installed MS-DOS.

Return Value

The return value is the major version number of MS-DOS.

GetDOSMinorVersion

int GetDOSMinorVersion(void);

GetDOSMinorVersion determines the minor version number of the currently installed MS-DOS.

Return Value

The return value is the minor version number of MS-DOS.

GetEnvVariableValue

LPSTR GetEnvVariableValue(LPSTR *szEnvVar*, **LPSTR** *szBuf*, **int** *cbBuf*);

GetEnvVariableValue determines the associated value for the specified environment variable.

Arguments

szEnvVar
Specifies the environment variable name.

szBuf
Points to the destination buffer that will receive the string.

cbBuf
Length, in bytes, of the destination buffer.

Return Value

The return value is the string associated with the environment variable. If there is no associated value, the string is empty.

GetFreeSpaceForDrive

LONG GetFreeSpaceForDrive(LPSTR *szDrive***);**

GetFreeSpaceForDrive determines the amount of free space available on the specified disk drive.

Argument

szDrive
Specifies a string identifying the disk drive letter (A through Z).

Return Value

The return value is the amount of free disk space. If *szDrive* is not a valid disk drive, the return value is zero.

GetHourFromDate

int GetHourFromDate(LPSTR *szDate***);**

GetHourFromDate retrieves the hour field from the return value for **GetDateOfFile** .

Argument

szDate
Specifies the date in YYYY-MM-DD-HH-MM-SS format. This value is obtained from **GetDateOfFile**.

Return Value

The return value is an integer with a valid range from 0 through 23.

GetIniKeyString

LPSTR GetIniKeyString(LPSTR *szFile*, **LPSTR** *szSect*, **LPSTR** *szKey*, **LPSTR** *szBuf*, **int** *cbBuf*);

GetIniKeyString searches the .INI file for the specified key string.

Arguments

szFile
Specifies the .INI file that you want to search. If you specify WIN.INI, you do not have to provide the full path.

szSect
Specifies the section of the file to be searched. *szSect* must be a non-empty string.

szKey
Specifies the key or symbol you want to search for.

szBuf
Points to the destination buffer that will receive the string.

cbBuf
Length, in bytes, of the destination buffer.

Return Value

If the key is found, the return value is the full string associated with the key. Otherwise, the return value is an empty string.

GetListItem

LPSTR GetListItem(LPSTR *szSymbol*, **INT** *n*, **LPSTR** *szBuf*, **INT** *cbBuf*);

GetListItem retrieves a single item from the list associated with the specified symbol name.

Arguments

szSymbol
Specifies the name of the symbol in the Symbol Table with which the list is associated.

n
Specifies the index (one-based) of the item you want to retrieve.

szBuf
Points to the destination buffer that will receive the string.

cbBuf
Length, in bytes, of the destination buffer.

Return Value

The return value is the item from the list.

Comments

If the string is empty, the symbol or item does not exist.

GetListLength

int GetListLength(LPSTR *szSymbol*);

GetListLength determines the number of items in the list associated with the specified symbol name.

Argument

szSymbol
Specifies the name of the symbol in the Symbol Table with which the list is associated.

Return Value

The return value is the number of items in the list.

GetLocalHardDrivesList

void GetLocalHardDrivesList(LPSTR *szSymbol***);**

GetLocalHardDrivesList sets the specified symbol to a list of all local hard drives (that is, "A", "B", and so on).

Argument

szSymbol
Specifies the name of the symbol to associate with the list.

GetMinuteFromDate

int GetMinuteFromDate(LPSTR *szDate***);**

GetMinuteFromDate retrieves the minute field from the return value for **GetDateOfFile**.

Argument

szDate
Specifies the date in YYYY-MM-DD-HH-MM-SS format. This value is obtained from the **GetDateOfFile** function.

Return Value

The return value is an integer with a valid range from 0 through 59.

GetMonthFromDate

int GetMonthFromDate(LPSTR *szDate***);**

GetMonthFromDate retrieves the month field from the return value for **GetDateOfFile**.

Argument

szDate
Specifies the date in YYYY-MM-DD-HH-MM-SS format. This value is obtained from **GetDateOfFile**.

Return Value

The return value is an integer with a valid range from 1 through 12.

GetNetworkDrivesList

void GetNetworkDrivesList(LPSTR *szSymbol***);**

GetNetworkDrivesList sets the specified symbol to a list of all network drives (that is, "A", "B", and so on).

Argument

szSymbol
Specifies the name of the symbol to associate with the list.

GetNthFieldFromIniString

LPSTR GetNthFieldFromIniString(LPSTR *szLine*, **INT** *iField*, **LPSTR** *szBuf*, **INT** *cbBuf*);

GetNthFieldFromIniString extracts the specified comma-separated field from the specified string.

Arguments

szLine
Specifies the string from which you want to extract the field.

iField
Specifies the index (one-based) of the field that you want to extract.

szBuf
Points to the destination buffer that will receive the string.

cbBuf
Length, in bytes, of the destination buffer.

Return Value

The return value is the field. If the requested field doesn't exist or is invalid, the return value is an empty string.

GetNumWinApps

int GetNumWinApps(void);

GetNumWinApps determines the number of unique instances of Windows applications currently running in the system.

Return Value

The return value is the number of applications.

GetParallelPortsList

void GetParallelPortsList(LPSTR *szSymbol***);**

GetParallelPortsList sets the specified symbol to a list of all parallel ports (that is, "LPT1","LPT2",_).

Argument

szSymbol
Specifies the name of the symbol to associate with the list.

GetProcessorType

int GetProcessorType(void);

GetProcessorType determines the type of processor being run on the user's system.

Return Value

The return value is either zero (for 8086), one (for 80186), two (for 80286), three (for 80386), or four (for 80486).

GetRegKeyValue

LPSTR GetRegKeyValue(LPSTR *szKey***, LPSTR** *szBuf***, int** *cbBuf***);**

GetRegKeyValue determines the value associated with the specified Registration Database key.

Argument

szKey
Specifies the name of the key whose value you want to retrieve. Because all keys are assumed to be subkeys of HKEYS_CLASSES_ROOT, you only have to specify the name of the key.

szBuf
Points to the destination buffer that will receive the string.

cbBuf
Length, in bytes, of the destination buffer.

Return Value

The return value is the value associated with *szKey* in the Registration Database.

GetRemovableDrivesList

void GetRemovableDrivesList(LPSTR *szSymbol***);**

GetRemovableDrivesList sets the specified symbol to a list of all removable drives (that is, "A", "B", and so on).

Argument

szSymbol
Specifies the name of the symbol to associate with the list.

GetScreenHeight

int GetScreenHeight(void);

GetScreenHeight determines the height of the screen.

Return Value

The return value is the height of the screen (in pixels).

GetScreenWidth

int GetScreenWidth(void);

GetScreenWidth determines the width of the screen.

Return Value

The return value is the width of the screen (in pixels).

GetSecondFromDate

int GetSecondFromDate(LPSTR *szDate***);**

GetSecondFromDate retrieves the seconds field from the return value for **GetDateOfFile**.

Argument

szDate
Specifies the date in YYYY-MM-DD-HH-MM-SS format. This value is obtained from the **GetDateOfFile** function.

Return Value

The return value is an integer with a valid range from 0 through 59.

GetSectionKeyDate

LPSTR GetSectionKeyDate(LPSTR *szSect*, **LPSTR** *szKey*, **LPSTR** *szBuf*, **INT** *cbBuf*);

GetSectionKeyDate retrieves the date from a file description in the .INF file.

Arguments

szSect
Specifies the name of the section where the file description resides.

szKey
Specifies the name of the key associated with the file description.

szBuf
Points to the destination buffer that will receive the string.

cbBuf
Length, in bytes, of the destination buffer.

Return Value

The return value is the date. This string can be parsed by **GetDayFromDate**, **GetMonthFromDate**, and **GetYearFromDate**.

GetSectionKeyFilename

LPSTR GetSectionKeyFilename(LPSTR *szSect*, **LPSTR** *szKey*, **LPSTR** *szBuf*, **INT** *cbBuf*);

GetSectionKeyFilename retrieves the filename from a file description in the .INF file.

Arguments

szSect
Specifies the name of the section where the file description resides.

szKey
Specifies the name of the key associated with the file description.

szBuf
Points to the destination buffer that will receive the string.

cbBuf
Length, in bytes, of the destination buffer.

Return Value

The return value is the filename.

GetSectionKeySize

LONG GetSectionKeySize(LPSTR *szSect*, **LPSTR** *szKey*);

GetSectionKeySize retrieves the file size from a file description in the .INF file.

Arguments

szSect
Specifies the name of the section where the file description resides.

szKey
Specifies the name of the key associated with the file description.

Return Value

The return value is the file size in bytes.

GetSectionKeyVersion

LPSTR GetSectionKeyVersion(LPSTR *szSect*, **LPSTR** *szKey*, **LPSTR** *szBuf*, **INT** *cbBuf*);

GetSectionKeyVersion retrieves the version number from a file description in the .INF file.

Arguments

szSect
Specifies the name of the section where the file description resides.

szKey
Specifies the name of the key associated with the file description.

szBuf
Points to the destination buffer that will receive the string.

cbBuf
Length, in bytes, of the destination buffer.

Return Value

The return value is the version number. This string can be parsed by **GetVersionNthField**.

GetSerialPortsList

void GetSerialPortsList(LPSTR *szSymbol***);**

GetSerialPortsList sets the specified symbol to a list of all serial ports (that is, "COM1","COM2", and so on).

Argument

szSymbol
Specifies the name of the symbol to associate with the list.

GetSizeOfFile

LONG GetSizeOfFile(LPSTR *szFile***);**

GetSizeOfFile determines the size of the specified file.

Argument

szFile
Specifies the path and name of the file.

Return Value

The return value is the size of the file in bytes. If *szFile* is invalid or does not exist, the return value is zero.

GetSymbolValue

LPSTR GetSymbolValue(LPSTR *szSymbol*, **LPSTR** *szBuf*, **INT** *cbBuf*);

GetSymbolValue finds the value associated with a symbol in the Symbol Table.

Arguments

szSymbol
Specifies the name of the symbol to be found in the Symbol Table.

szBuf
Points to the destination buffer that will receive the string.

cbBuf
Length, in bytes, of the destination buffer.

Return Value

The return value is the value associated with *szSymbol* in the Symbol Table. If there is no value associated with *szSymbol*, the return value is an empty string.

Comments

Use this function to retrieve information stored in the Symbol Table by other .DLL file procedures.

GetTotalSpaceForDrive

LONG GetTotalSpaceForDrive(LPSTR *szDrive***);**

GetTotalSpaceForDrive determines the total amount of disk space for the specified disk drive.

Argument

szDrive
Specifies a string identifying the disk drive letter (A through Z).

Return Value

The return value is the total capacity in bytes of the disk drive. If *szDrive* is not a valid disk drive, the return value is zero.

GetTypeFaceNameFromTTF

int GetTypeFaceNameFromTTF(LPSTR *szFile***, LPSTR** *szBuf***, int** *cbBuf***);**

GetTypeFaceNameFromTTF extracts the typeface name from a TrueType font file.

Arguments

szFile
Specifies the name of the TrueType font file.

szBuff
Specifies the buffer you are providing for the storage of the typeface name.

cbBuff
Specifies the size of the buffer you are providing.

Return Value

If the file you specified is not a TrueType font file, the return value is zero. Otherwise, the return value is the actual length of the typeface name.

Comments

Check to ensure that the return value is not greater than the size of *szBuff.* If the return value is greater, the typeface name has been truncated.

GetValidDrivesList

void GetValidDrivesList(LPSTR *szSymbol***);**

GetValidDrivesList sets the specified symbol to a list of all valid disk drives (that is, "A", "B", and so on).

Argument

szSymbol
Specifies the name of the symbol to associate with the list.

GetVersionNthField

LONG GetVersionNthField(LPSTR *szVersion*, **int** *nField*);

GetVersionNthField extracts the specified field from the return value for **GetVersionOfFile**.

Arguments

szVersion
Specifies the string returned from **GetVersionOfFile**.

nField
Specifies the number of the field you want to extract (1 through 4 from the version string).

Return Value

The return value is the integer extracted from *szVersion*.

GetVersionOfFile

LPSTR GetVersionOfFile(LPSTR *szFile***, LPSTR** *szBuf***, int** *cbBuf***);**

GetVersionOfFile determines the version of the specified file.

Argument

szFile
Specifies the path and name of the file.

szBuff
Specifies the buffer you are providing for the storage of the typeface name.

cbBuff
Specifies the size of the buffer you are providing.

Return Value

The return value is a string in the format *N.N.N.N*, where each *N* is an integer.

GetWindowsDir

LPSTR GetWindowsDir(LPSTR *szBuf*, **INT** *cbBuf*);

GetWindowsDir determines the name and path of the Windows directory.

Arguments

szBuff
Specifies the buffer you are providing for the storage of the typeface name.

cbBuff
Specifies the size of the buffer you are providing.

Return Value

The return value is the path name terminated with a backslash; for example, "C:\WINDOWS\".

GetWindowsMajorVersion

int GetWindowsMajorVersion(void):

GetWindowsMajorVersion determines the major version number for the currently installed Windows software.

Return Value

The return value is the Windows major version number.

GetWindowsMinorVersion

int GetWindowsMinorVersion(void);

GetWindowsMinorVersion determines the minor version number for the currently installed Windows software.

Return Value

The return value is the Windows minor version number.

GetWindowsMode

int GetWindowsMode(void);

GetWindowsMode determines the current mode of Windows.

Return Value

The return value is zero for Real mode, one for Standard mode, or two for Enhanced mode.

GetWindowsSysDir

LPSTR GetWindowsSysDir(LPSTR *szBuf*, **INT** *cbBuf*);

GetWindowsSysDir determines the name and path of the Windows system directory.

Arguments

szBuff
Specifies the buffer you are providing for the storage of the typeface name.

cbBuff
Specifies the size of the buffer you are providing.

Return Value

The return value is the path which ends with a backslash.

GetYearFromDate

int GetYearFromDate(LPSTR *szDate***);**

GetYearFromDate retrieves the year field from the return value for **GetDateOfFile**.

Argument

szDate
Specifies the date in YYYY-MM-DD-HH-MM-SS format. This value is obtained from **GetDateOfFile**.

Return Value

The return value is an integer with a valid range from 1980 through 2099.

HandleSharedFile

LPSTR HandleSharedFile(LPSTR *szInfSection*, **LPSTR** *szInfKey*, **LPSTR** *szSubDir*, **LPSTR** *szRegDbKey*, **LPSTR** *szWinIniSect*, **LPSTR** *szWinIniKey*, **int** *iWinIniField*, **LPSTR** *szBuf*, **int** *cbBuf*);

HandleSharedFile checks a shared file already exists, based on *szWinIniKey*. If the file does not exist, it is added to WIN.INI.

Arguments

szInfSection
Specifies the .INF Section that contains the shared application description line.

szInfKey
Specifies the .INF reference key for the shared application description line.

szSubDir
Specifies the shared application WIN.INI key (such as PROOF, TEXTCONV, or MSDRAW).

szRegDbKey
Specifies the Registration Database key that might contain the full path of an existing copy of the shared application.

szWinIniSect
Specifies the WIN.INI section that might contain the full path of an existing copy of the shared application (such as "MS Proofing Tools").

szWinIniKey
Specifies the WIN.INI key which might reference the full path of an existing copy of the shared application (such as "Spelling 1033,0").

iWinIniField
An index of the path in *szWinIniKey* (such as "1" for speller, "2" for dictionary).

szBuf
Specifies the destination buffer for the full path of the shared application.

cbBuf
The size of *szBuf*.

Has87MathChip

BOOL Has87MathChip(void);

Has87MathChip checks for an 87 math coprocessor on the user's system.

Return Value

If an 87 math coprocessor exists, the return value is one. Otherwise, the return value is zero.

HasMonochromeDisplay

BOOL HasMonochromeDisplay(void);

HasMonochromeDisplay checks for a monochrome display on the user's system.

Return Value

If the monochrome display exists, the return value is one. Otherwise, the return value is zero.

HasMouseInstalled

BOOL HasMouseInstalled(void);

HasMouseInstalled determines if a mouse is installed on the user's system.

Return Value

If a mouse exists, the return value is one. Otherwise, the return value is zero.

HinstFrame

handle HinstFrame(void);

HinstFrame retrieves the instance handle for the Setup program.

Return Value

The return value is the instance handle.

HwndFrame

int HwndFrame(void);

HwndFrame provides the handle for the Setup application frame window (the main window).

Return Value

The return value is the handle of the frame window (the main window).

Comments

Use the return value from this function for the *hwnd* argument in procedures that require it.

InitSetupToolkit

int InitSetupToolkit(LPSTR *szCmdLine***);**

InitSetupToolkit creates the Setup Toolkit window.

Argument

szCmdLine
Specifies the command line passed in to Setup, which must be called at the beginning of every Setup application.

InStr

int InStr(INT *cch*, **LPSTR** *sz1*, **LPSTR** *sz2*);

InStr determines if *sz2* is contained in *sz1*, starting at the character *cch*.

cch
Position to compare from, must be 1 or greater.

sz1
Source string.

sz2
String to search for in *sz1*.

IsDirWritable

int IsDirWritable(LPSTR *szDir***);**

IsDirWritable determines if the specified directory is writable (so that Setup can create a file in it).

Argument

szDir
Specifies the name of the directory.

Return Value

If the directory is writable, the return value is one. Otherwise, the return value is zero.

IsDriveLocalHard

BOOL IsDriveLocalHard(LPSTR *szDrive***);**

IsDriveLocalHard determines whether the specified disk drive is a local hard disk.

Argument

szDrive
Specifies a string identifying the disk drive letter (A through Z).

Return Value

If the disk drive is a local hard disk, the return value is one. Otherwise, the return value is zero.

IsDriveNetwork

BOOL IsDriveNetwork(LPSTR *szDrive***);**

IsDriveNetwork determines if the specified disk drive is a network drive.

Argument

szDrive
Specifies a string identifying the disk drive letter (A through Z).

Return Value

If the disk drive is a network drive, the return value is one. Otherwise, the return value is zero.

IsDriveRemovable

BOOL IsDriveRemovable(LPSTR *szDrive***);**

IsDriveRemovable determines if the specified disk drive is a removable disk drive.

Argument

szDrive
Specifies a string identifying the disk drive letter (A through Z).

Return Value

If the disk drive is removable, the return value is one. Otherwise, the return value is zero.

IsDriverInConfig

BOOL IsDriverInConfig(LPSTR *szDrv***);**

IsDriverInConfig determines if the specified device driver is in the CONFIG.SYS file.

Argument

szDrv
Specifies the name of the device driver you want to find.

Return Value

If the device driver statement is found, the return value is one. Otherwise, the return value is zero.

IsDriveValid

BOOL IsDriveValid(LPSTR *szDrive***);**

IsDriveValid determines if the specified disk drive exists.

Argument

szDrive
Specifies a string identifying the disk drive letter (A through Z).

Return Value

If the specified disk drive is valid, the return value is one. Otherwise, the return value is zero.

IsFileWritable

bool IsFileWritable(LPSTR *szFile***);**

IsFileWritable determines whether the specified file is writable.

Argument

szFile
Specifies the full path and name of the file you want to write to.

Return Value

If the file is writable, the return value is one. Otherwise, the return value is zero.

IsWindowsShared

bool IsWindowsShared(void);

IsWindowsShared determines if Windows is shared by comparing the Windows and system directories.

Return Value

If Windows is shared, the return value is one. Otherwise, the return value is zero.

MakeListFromSectionDate

VOID MakeListFromSectionDate(LPSTR *szSym***, LPSTR** *szSect***);**

MakeListFromSectionDate creates a string with all of the dates from the specified .INF section.

Arguments

szSym
Specifies the buffer to receive the list of dates.

szSect
Specifies the section name.

MakeListFromSectionFilename

VOID MakeListFromSectionFilename(LPSTR *szSym*, **LPSTR** *szSect*);

MakeListFromSectionFilename creates a string with all of the filenames from the specified .INF section.

Arguments

szSym
Specifies the buffer to receive the list of filenames.

szSect
Specifies the section name.

MakeListFromSectionKeys

VOID MakeListFromSectionKeys(LPSTR *szSymbol*, **LPSTR** *szSect*);

MakeListFromSectionKeys creates a list of the reference key values found in a section of the currently open .INF file and associates the list with the specified symbol name.

Arguments

szSymbol
Specifies the symbol name of the list in the Symbol Table.

szSect
Specifies the name for the section in the .INF file that contains the reference key values.

Comments

Use this function to create a list that can be used in dialog list boxes.

MakeListFromSectionSize

VOID MakeListFromSectionSize(LPSTR *szSym*, **LPSTR** *szSect*);

MakeListFromSectionSize creates a string with all of the sizes from the specified .INF section.

Arguments

szSym
Specifies the buffer to receive the list of sizes.

szSect
Specifies the section name.

MakeListFromSectionVersion

VOID MakeListFromSectionVersion(LPSTR *szSym*, **LPSTR** *szSect*);

MakeListFromSectionVersion creates a string with all of the version numbers from the specified .INF section.

Arguments

szSym
Specifies the buffer to receive the list of versions.

szSect
Specifies the section name.

OpenLogFile

VOID OpenLogFile(LPSTR *szFile*, **INT** *fAppend*);

OpenLogFile opens a log file that the Setup program or your script can use for writing status information.

Argument

szFile
Specifies the path and name of the log file. If this file does not exist, it is created.

fAppend
Specifies whether to add information to the log file or overwrite the existing contents of the file. Specify one to add information and zero to overwrite information.

PrependToPath

VOID PrependToPath(LPSTR *szSrc*, **LPSTR** *szDst*, **LPSTR** *szDir*, **INT** *cmo*);

PrependToPath modifies the specified AUTOEXEC.BAT to add a specified directory to the head of the PATH environment variable.

Arguments

szSrc
Specifies the full path to file to modify. If NULL or empty (" "), just create a new, one line *szDst* file.

szDst
Specifies the valid full path for the resulting modified file.

szDir
Specifies the valid path to add.

cmo
Specifies the command option flag. You can use : cmoOverwrite, which overwrites any existing *szDst* file, or cmoVital, which calls the Vital command handler if an error occurs.

Comments

The directory is added to the head of the first PATH statement of the file and to every following PATH statement that doesn't contain a %PATH% reference. **PrependToPath** does not check for resulting file line length or duplicate directory nodes in the path and dynamically allocates a buffer for the whole file.

Returns

Returns **fTrue** if the function is successful and **fFalse** if an error occurs.

ReadInfFile

VOID ReadInfFile(LPSTR *szFile***);**

ReadInfFile opens and reads the contents of the specified .INF file.

Arguments

szFile
Specifies the path and name of the .INF file.

Comments

The Setup program should call **ReadInfFile** to read an .INF file before attempting to call any of the procedures that access .INF file information.

RemoveDir

VOID RemoveDir(LPSTR *szDir*, **INT** *cmo*);

RemoveDir deletes the specified directory.

Arguments

szDir
Specifies the path and the name of the directory to be deleted.

cmo
Specifies the command option flag. You can use either *cmoVital* or *cmoNone* for the command option flag.

RemoveEnvVariable

BOOL RemoveEnvVariable(LPSTR *szName*, **LPSTR** *szValue***);**

RemoveEnvVariable removes the specified value from the specified environment variable. Under Windows NT, the environment variables affected are the ones specified in the registry. Under Windows 95, autoexec.bat is modified to remove the values.

Arguments

szName
Specifies the name of the environment variable.

szValue
Specifies the value that you want to remove from the environment variable.

Return Value

If the function succeeds, it returns **fTrue**.

If the function fails, it returns **fFalse**.

See Also

SetEnvVariable

RemoveFile

VOID RemoveFile(LPSTR *szFullPathSrc*, **INT** *cmo*);

RemoveFile deletes the specified file.

Arguments

szFullPathSrc
Specifies the full path of the file you want to delete.

cmo
Specifies the command option flag. You can use *cmoVital, cmoForce,* or *cmoNone*.

RemoveIniKey

VOID RemoveIniKey(LPSTR *szFile*, **LPSTR** *szSect*, **LPSTR** *szKey*, **INT** *cmo*);

RemoveIniKey deletes a symbol from the specified .INI file.

Arguments

szFile
Specifies the name of the .INI file. If the .INI file is WIN.INI, you do not have to provide the full path.

szSect
Specifies the name of the section where the symbol to be deleted exists. *szSect$* must be a non-empty string.

szKey
Specifies the name of the symbol you want to delete.

cmo
Specifies the command option flag. You can use *cmoVital* or *cmoNone*.

RemoveIniSection

VOID RemoveIniSection(LPSTR *szFile*, **LPSTR** *szSect*, **INT** *cmo*);

RemoveIniSection deletes the specified section from the designated .INI file.

Arguments

szFile
Specifies the name of the .INI file. If the .INI file is WIN.INI, you do not have to provide the full path.

szSect
Specifies the name of the section you want to delete. *szSect$* must be a non-empty string.

cmo
Specifies the command option flag. You can use *cmoVital* or *cmoNone.*

RemoveSymbol

VOID RemoveSymbol(LPSTR *szSym***);**

RemoveSymbol deletes a symbol from the Symbol Table.

Argument

szSym
Specifies the name of the symbol in the Symbol Table.

Comments

Use **RemoveSymbol** to free up space occupied by unused symbols and their associated values in the Symbol Table.

RenameFile

VOID RenameFile(LPSTR *szFullPath*, **LPSTR** *szBackup*);

The **RenameFile** renames the specified file.

Arguments

szFullPath
Specifies the full path and name of the file you want to rename.

szBackup
Specifies the new name for the file.

Comments

The renamed file is placed in the same directory as the original file (as specified by *szFullPath*). **RenameFile** is the same as **BackupFile**.

ReplaceListItem

VOID ReplaceListItem(LPSTR *szSymbol*, **INT** *n*, **LPSTR** *szItem*);

ReplaceListItem replaces an item in the list associated with the specified symbol.

Arguments

szSymbol
Specifies the name of the symbol whose associated value is the list.

n
Specifies the index (one-based) of the item to be replaced.

szItem
Specifies the new item.

RestartListEmpty

bool RestartListEmpty(void);

RestartListEmpty determines if any files have been added to _MSSETUP.BAT that need to be copied or deleted by **ExitExecRestart** when Windows is restarted.

Return Value

If the restart list is empty, the return value is one. Otherwise, the return value is zero.

Comments

Setup adds files to _MSSETUP.BAT as a result of reading the .INF file and finding files identified as a system resource (that is, the file description contains a SYSTEM flag). If the file is newer than the file on the user's system, Setup adds its name to _MSSETUP.BAT.

RestoreCursor

VOID RestoreCursor(int *hPrev***);**

The **RestoreCursor** restores the previous cursor state.

Argument

hPrev

Specifies the identification number of the previous cursor state. Use the return value from **ShowWaitCursor** for this parameter.

RightTrim

VOID RightTrim(LPSTR *sz***);**

RightTrim removes any trailing spaces from a string.

Argument

sz
Specifies the string to be trimmed.

SearchForLocationForSharedFile

LPSTR SearchForLocationForSharedFile (LPSTR *szRegDbKey*, **LPSTR** *szWinIniSect*, **LPSTR** *szWinIniKey*, **LPSTR** *iWinIniField*, **LPSTR** *szDefault*, **LPSTR** *szVersion*, **LPSTR** *szBuf*, **int** *cbBuf*);

SearchForLocationForSharedFile uses information from the Registration Database and WIN.INI to determine where the specified shared file should be installed and whether the file will actually be copied.

Arguments

szRegDbKey
Specifies the Registration Database key that might have an associated value that is the path of an existing copy of the file. You can tell the function to ignore the Registration Database in its search by specifying an empty string for this argument.

szWinIniSect
Specifies the section in the WIN.INI file that might have an entry that is the path of an existing copy of the file. You can tell the function to ignore WIN.INI in its search by specifying an empty string for this argument.

szWinIniKey
Specifies the key for the WIN.INI file that will contain a path to an existing copy of this shared file.

iWinIniField
Specifies the index (one-based) of the field in the WIN.INI line entry that contains a path to an existing copy of this shared file.

szDefault
Specifies the default path of the file if no existing copy can be found.

szVersion
Specifies the version for the new copy of the file as a string of 1 to 4 integers separated by periods; for example, 3.1.0.16.

szBuf
Specifies the source buffer.

cbBuf
Size of the source buffer.

Return Value

The return value is the full path for installing the shared file. This function also sets the global variable **SharedFileNeedsCopying** to one (if the file should be copied) or zero (if the file shouldn't be copied).

Comments

To determine a location for the shared file, the function first uses the Registration Database path. If this file doesn't exist, the function will retrieve only the filename from the Registration Database and perform a **FileOpen** function to try to determine the path. If either of these methods works and the file found is writable or newer, the function returns that path. If these methods fail, the function uses the path from the WIN.INI field. If that file doesn't exist in the WIN.INI field, the function uses the path from WIN.INI and searches by using a **FileOpen** function. If these two methods don't succeed, *szDefault* is used.

After determining where the file should be installed, the function then determines if the file will be copied later when **CopyFilesInCopyList** is called. **SearchForLocationForSharedFile** then sets the value of the global variable *SharedFileNeedsCopying* accordingly.

SetAbout

VOID SetAbout(LPSTR *szAbout1*, **LPSTR** *szAbout2*);

SetAbout adds the specified strings to Setup's About dialog box.

Arguments

szAbout1
Specifies the first string you want to add to the dialog box, typically the product name.

szAbout2
Specifies the second string you want to add to the dialog box, typically the product version, the date, or the copyright notice.

SetBeepingMode

int SetBeepingMode(int *mode***);**

SetBeepingMode allows the Setup script to specify whether error messages, requests for diskettes, and similar messages will be accompanied by a beep.

Argument

mode
Specifies the beeping mode value. If *mode* is one, beeping mode is on and the message will generate beeps. If *mode* is zero, beeping mode is off.

Return Value

The return value is the value of the previous beeping mode.

SetBitmap

VOID SetBitmap(LPSTR *szDll*, **int** *Bitmap*);

SetBitmap defines the logo bitmap used in the background of the frame (or main) window.

Arguments

szDll
Specifies the name of the .DLL file that contains the bitmap resource.

Bitmap
Specifies the identification number of the bitmap resource.

Comments

It is best to use a plain white bitmap. **SetBitmap** automatically adds a shadow down the right side to give the impression that the bitmap is above the plane of the background.

SetCopyGaugePosition

VOID SetCopyGaugePosition(int *x*, **int** *y*);

SetCopyGaugePosition specifies the display position of the Copy Gauge dialog box during subsequent **CopyFilesInCopyList** calls.

Arguments

x

Specifies the *x* coordinate in dialog units relative to the upper-left corner of the client window frame. A value of 1 precisely centers the dialog box horizontally.

y

Specifies the *y* coordinate in dialog units relative to the upper-left corner of the client window frame. A value of 1 centers the dialog box vertically one-third of the distance from the top margin.

Comments

By default, the Copy Gauge dialog box is centered over the client window frame. However, the default position can interfere with the display of billboard dialog boxes, depending on how the user has resized the window, the size of the monitor, and so on. Use **SetCopyGaugePosition** to control the position of the Copy Gauge dialog box.

SetCopyMode

int SetCopyMode(int *fMode***);**

SetCopyMode specifies whether the files from the copy list will actually be copied.

Argument

fMode

Specifies whether copy mode is on (one) or off (zero). If the copy mode is off, the files will not be copied when the script calls **CopyFilesInCopyList**.

Return Value

The return value is the prior value for the copy mode.

Comments

Use this function to streamline testing of your Setup program.

SetEnvVariable

BOOL SetEnvVariable(LPSTR *szName*, **LPSTR** *szValue*, **CMO** *cmo*);

SetEnvVariable sets the specified environment variable to a certain value. The environment variable can also be elongated by specifying the command option flag *fAppend* or *fPrepend*. Under Windows NT, the environment variables affected are the ones specified in the registry. Under Windows 95, autoexec.bat is modified to set the values.

Arguments

szName
Specifies the name of the environment variable.

szValue
Specifies the new value that you want to associate with the environment variable.

cmo
Specifies the command option flag. You can use *fOverwrite*, *fAppend*, or *fPrepend*.

Return Value

If the function succeeds, it returns **fTrue**.

If the function fails, it returns **fFalse**.

See Also

RemoveEnvVariable

SetDecompMode

int SetDecompMode(int *fMode***);**

SetDecompMode specifies whether compressed files will be decompressed when they are copied.

Argument

fMode

Specifies whether decompression is on (one) or off (zero). When decompression is off, **CopyFilesInCopyList** copies compressed files byte for byte without decompressing them.

Return Value

The return value is the prior value for decompression.

Comments

You can use **SetDecompMode** to install compressed files on a network drive.

SetRegKeyValue

void SetRegKeyValue(LPSTR *szKey*, **LPSTR** *szValue*);

SetRegKeyValue replaces the value associated with the specified Registration Database key with the specified value.

Arguments

szKey
Specifies the name of the key. All keys are assumed to be subkeys of HKEY_CLASSES_ROOT; therefore, you only have to specify the name of the key.

szValue
Specifies the new value you want to associate with the key.

SetRestartDir

VOID SetRestartDir(LPSTR *szDir***);**

SetRestartDir establishes the restart directory where _MSSETUP.EXE and _MSSETUP.BAT (used to restart the system) will be located.

Argument

szDir
Specifies the name of the directory. If *szDir* does not exist, it is created.

SetSilentMode

int SetSilentMode(INT *mode***);**

SetSilentMode allows the Setup script to determine whether or not the copy gauge, error messages, billboards, and so on will be displayed.

Argument

mode

Specifies the silent mode value. If *mode* is one, silent mode is on and the Setup procedures cannot display message-related dialog boxes. If *mode* is zero, silent mode is off.

Return Value

The return value is the value of the previous silent mode.

SetSymbolValue

VOID SetSymbolValue(LPSTR *szSymbol*, **LPSTR** *szValue*);

SetSymbolValue associates a value with a symbol in the Symbol Table.

Arguments

szSymbol
Specifies the name of the symbol in the Symbol Table.

szValue
Specifies the value you want to associate with the symbol.

Comments

Use **SetSymbolValue** to store information that can be shared among .DLLs and the Symbol Table.

SetTitle

VOID SetTitle(LPSTR *sz***);**

SetTitle defines the title used in the frame (or main) window.

Argument

sz
Defines the title used in the frame window.

ShowProgmanGroup

VOID ShowProgmanGroup(LPSTR *szGroup*, **INT** *Cmd*, **INT** *cmo*);

ShowProgmanGroup minimizes, maximizes, or restores the window of the specified Program Manager group.

Arguments

szGroup
Specifies the name of the Program Manager group.

Cmd
Specifies the operation you want to perform on the group window. To activate and display the group window, use one; to activate and display the group window icon, use two; and to activate and display the group window maximized, use three.

cmo
Specifies the command option flag. You can use *cmoVital* or *cmoNone*.

Comments

Creating a new item in a Program Manager group causes the Program Manager to become active and its window to come to the front. You can use **ShowProgmanGroup** to determine what state the group window will be in when it comes to the front.

ShowWaitCursor

int ShowWaitCursor(void);

ShowWaitCursor loads and displays the wait cursor.

Return Value

The return value is the previous cursor state.

Comments

Use the return value of this function for the *hPrev* parameter in **RestoreCursor**.

StampResource

VOID StampResource(LPSTR *szSect*, **LPSTR** *szKey*, **LPSTR** *szDst*, **int** *wResType*, **int** *wResId*, **LPSTR** *szData*, **int** *cbData*);

StampResource modifies the first *cbData* bytes of a file resource with the specified data.

Arguments

szSect
Specifies the section of the .INF file that contains the description line of the file to be modified.

szKey
Specifies the reference key to the description line of the file to be modified.

szDst
Specifies the destination directory where the file to be modified resides.

wResType
Specifies the resource identification type.

wResId
Specifies the resource identification number.

szData
Defines the data that will be used to modify the resource.

cbData
Specifies how many bytes of data will be replaced with *szData.*

SzCatStr

LPSTR SzCatStr(LPSTR *sz1*, **LPSTR** *sz2*);

SzCatStr appends sz2 to sz1 and returns the concatenated string.

Argument

sz1
Specifies the string to which sz2 is added.

sz2
Specifies the string to append to *sz1*.

Return Value

SzCatStr returns the concatenated string.

SzCat2Str

LPSTR SzCat2Str(LPSTR *sz1*, **LPSTR** *sz2*, **LPSTR** *sz3*);

SzCat2Str appends *sz2* and *sz3* to *sz1* so that *sz1* becomes *sz1*+*sz2*+*sz3*.

Arguments

sz1
Specifies the original string to which the others are appended.

sz2
Specifies the first string to be appended to *sz1*.

sz3
Specifies the string to be appended to *sz1*+*sz2*.

Return Value

SzCat2Str returns the concatenated string *sz1*+*sz2*+*sz3*.

SzCat3Str

LPSTR SzCat3Str(LPSTR *sz1*, **LPSTR** *sz2*, **LPSTR** *sz3*, **LPSTR** *sz4*);

SzCat3Str appends *sz2*, *sz3*, and *sz4* to *sz1* so that *sz1* becomes *sz1+sz2+sz3+sz4*.

Arguments

sz1
Specifies the original string to which the others are appended.

sz2
Specifies the first string to be appended to *sz1*.

sz3
Specifies the string to be appended to *sz1+sz2*.

sz4
Specifies the string to be appended to *sz1+sz2+sz3*.

Return Value

SzCat2Str returns the concatenated string *sz1+sz2+sz3+sz4*.

UIPop

void UIPop(int *n*);

UIPop destroys dialog boxes and removes them from the top of the dialog stack in memory.

Argument

n
Identifies the number of dialog boxes you want to destroy and remove from the dialog stack.

Comments

If *n* is greater than the number of dialog boxes in the dialog stack, the subroutine will destroy all the dialog boxes in the dialog stack.

UIPopAll

VOID UIPopAll(void);

UIPopAll destroys all dialog boxes and removes them from the dialog box stack in memory.

UIStartDlg

LPSTR UIStartDlg(LPSTR *szDll*, **int** *Dlg*, **LPSTR** *szDlgProc*, **int** *HelpDlg*, **LPSTR** *szHelpProc*, **LPSTR** *szBuf*, **int** *cbBuf*);

UIStartDlg launches a dialog box and adds it to the top of the dialog stack in memory.

Arguments

szDll
Specifies the name of the .DLL file that contains the dialog box template resources and procedures.

Dlg
Specifies the identification number of the dialog box template resource in the .DLL file.

szDlgProc
Specifies the name of the dialog box procedure exported in the .DLL file.

HelpDlg
Specifies the identification number of the associated help dialog box template resource in the .DLL file.

szHelpProc
Specifies the name of the associated help dialog box procedure exported in the .DLL file.

szBuf
Points to the destination buffer that will receive the string.

cbBuf
Length, in bytes, of the destination buffer.

Return Value

UIStartDlg returns a string that is the value associated with the DLGEVENT symbol at the time the dialog ends**.**

Comments

If the dialog box is modeless, **UIStartDlg** returns immediately; if the dialog box is modal, it returns after a user action.

WriteToLogFile

VOID WriteToLogFile(LPSTR *szStr***);**

WriteToLogFile writes the specified string to the log file.

Argument

szStr

Defines the information you want to write to the log file. The string is written and terminated with a new-line character.

Comments

If the log file is not open, **WriteToLogFile** does nothing.

INF File Format

This appendix provides information about the format of the .INF file. You do not need to create the .INF file yourself; the Disk Layout Tools will do it for you. Refer to the information below when you are using the Disk Layout Tools to define the properties of installable files.

The Setup toolkit contains a single sample .INF file: SAMPLE.INF. Look at this file as an example of a typical .INF file.

An .INF file will have at least three sections:

- Source Media Descriptions, which describes each of the disks in the installation set
- Files, which lists each of the files that will be installed by Setup
- Default File Settings, which describes the defaults Setup uses to install a file

The Files section can be split into as many sections as you want; you specify these section names using the Disk Layout Tools. You can also specify one or more sections for files that you want to remove during the installation. Throughout the .INF file, you will see lines that begin with a semicolon at the left margin. These are comment lines.

Source Media Descriptions

This section of the .INF file contains one line for each of the disks you use to install your product. Source Media Description lines must be indented and contain four quoted strings, separated by commas:

- The disk identification number, which is a unique integer between 1 and 999
- The disk label, which you create using the Disk Layout Tools
- The tag filename, which is the name of a file that resides on the diskette
- The relative path for SETUP.EXE. This fourth string exists only if the installable files reside on a network disk drive

The following is the Source Media Descriptions line for the first installation disk. Its tag file is SETUP.EXE:

"1","My Disk Label 1","SETUP.EXE,""

Files

Each line in this section of the .INF file has one of two formats:

1. The first format begins at the left margin with the identification number for the disk on which the file resides, followed by the filename and a list of nineteen file properties, separated by commas (see the table below). These entries are not enclosed in quotation marks. The following line, from SAMPLE1.INF, is a typical example of this format:

   1, bldcui\dialogs.res,,,,1992-01 30,,,,,,,ROOT,,, 13839,,6,,,
2. The second format begins at the left margin with a reference key enclosed in quotation marks, followed by an equal sign and an unquoted disk identification number. (A disk with that identification number must have a description line in the Source Media Descriptions section.) The reference key and disk identification number are followed by the filename and the list of file properties. The following line, from SAMPLE3.INF, describes a shared file with the reference key CustDict (for customer dictionary) :

   "CustDict" = 1, custom.dic, ,,,1992-01-13,,, OLDER,,,,,, SHARED, 69632, ,,, 0.2.0.2,

The following table lists the file properties that can be included in a file description line:

File Properties

Property	Possible values	Meaning
Append	<empty>	Don't append to an existing file, overwrite it instead.
	valid filename	Append to the specified file. **Note**: You cannot append the installable file to an existing file if you have specified Rename, Root, or Backup.
Backup	<empty>	Use the default setting.
	*	Back up the file to the same filename with a .BAK extension.
	valid filename	The filename you want to use for the backup copy.
Copy	COPY	Copy the file onto the user's hard disk or network server.
	!COPY	Don't copy the file.
Date	<empty>	Use the default setting.
	YYYY-MM-DD	Specify a date with a range from 1980-01-01 through 2099-12-31.
Decompress	<empty>	Use the default setting.
	DECOMPRESS	Decompress the file.
	!DECOMPRESS	Don't decompress the file.
Destination	<empty>	Use the destination directory specified in the script (for example, in the AddSectionFilesToCopyList subroutine).
	full path of a valid directory	Override the specified destination directory with this one.
Overwrite	<empty>	Use the default setting.
	ALWAYS NEVER OLDER UNPROTECTED	For information about these values, see Using the Disk Layout Utilities.
ReadOnly	<empty>	Use the default setting.

	READONLY	Set read-only attributes on the file after it has been installed.
	!READONLY	Don't set read-only attributes.
Remove	<empty>	Copy the file using other properties.
	REMOVE	Remove the file from the user's hard disk or network server.
	!REMOVE	Copy the file using other properties.
Rename	<empty>	Use the source filename when the file is copied.
	valid filename	Use the specified filename when the file is copied. **Note**: You cannot specify a filename for this property if you have also specified Root or Append.
Root	<empty>	Use the default setting.
	ROOT	Strip any subdirectories from the filename when the file is copied.
	!ROOT	Don't strip subdirectories from the filename when the file is copied. **Note***:* You cannot specify Root or Append if you have specified that the file be renamed when it is copied.
SetTimeStamp	<empty>	Use the default setting.
	SETTIME	Use the DATE property.
	!SETTIME	Use the current system time.
Shared	SHARED	Treat the file as if it is a shared file that is, an existing version of it may be in use during installation.
	<empty> or !SHARED	Do not treat the file as if it is a shared file.
Size	integer	Size of the file in bytes (uncompressed).
System	SYSTEM	This is a system file; therefore, it will be replaced during system restart.
	<empty> or !SYSTEM	Treat the file as if it is not a system file.
TimeToCopy	<empty>	Use the default setting.
	integer	Increment the progress indicator by integers. This is an arbitrary number of time units relative to the other file description lines.
Reserved	<empty>	This is a reserved field and must be empty.
Version	<empty>	There is no version resource in source file.
	1 to 4 integers separated by periods	The version of the file. For information about this format, see the Microsoft Windows SDK documentation.
Vital	<empty>	Use the default setting.
	VITAL	The installation will fail if this file cannot be successfully installed.
	!VITAL	The installation will not fail if this file cannot be successfully installed.

Default File Settings

This section contains default values that Setup uses if a file description line in the .INF file has no entry for the field. These lines always begin at the left margin with a quoted symbol name, followed by an equal sign and another quoted string that is the value associated with the symbol. The value is required but may be an empty string.

The following line is an example of a default setting from SAMPLE1.INF:

"STF_COPY" = "YES"

The following table lists the default settings.

Default File Settings

Attribute/Symbol	Possible values or format	Default
STF_BACKUP	"*" = create a .BAK file or " " = don't create one	" "
STF_COPY	"YES" = copy or " " = don't copy	"YES"
STF_DECOMPRESS	"YES" = decompress or " " = don't decompress	" "
STF_OVERWRITE	"ALWAYS" = always overwrite an existing version of the file, "NEVER" = never overwrite an existing version of the file, "OLDER" = overwrite the existing version of the file if it is older, or "UNPROTECTED" = overwrite the existing version of the file if it is not write-protected	"ALWAYS"
STF_READONLY	"YES" = when the file is copied, set the read-only file attribute or " " = do not set the read-only file attribute	" "
STF_ROOT	"YES" = when copying the file, strip any subdirectories from the path of the file in the .INF file description or " " = use the entire path for the file in the .INF file decription	" "
STF_SETTIME	"YES" = use the contents of the Date file property in the .INF file description to set the creation date of the copied file or " " = leave the creation date as is (the time at which the file was copied onto the user's hard disk)	"YES"
STF_TIME	non-negative integer = use this number to calculate the display of the copy gauge or "" = use the value entered for the Size file property in the .INF file decription	value of Size
STF_VITAL	"YES" = this is a vital file that must be successfuly copied to the user's hard disk or the installation will fail; or	" " = the file is not vital to the success of the installation

"YES

Command Option Flags

This appendix provides a list of the command option flags that you can specify as parameters for many of the Setup script procedures. Use the list below to determine which command option flag to use for a function or subroutine. You can use combinations of command option flags with logical operators (such as AND or OR). For more information about Setup procedures and which ones use command option flags, see Setup Procedures.

Command Option Flags

Name	Meaning
cmoVital	This function must be successfully completed or the installation will fail and Setup will be terminated.
cmoDecompress	The file should be decompressed as it is copied. If this option is not specified, the file will be copied byte by byte, whether or not it is compressed.
cmoTimeStamp	Set the timestamp on the file after it has been copied.
cmoReadOnly	Set a read-only attribute on the file after it has been copied.
cmoBackup	Back up any existing version of the file (using the same filename with a .BAK extension) before the source file is copied onto the user's hard disk or network server.
cmoForce	Force the removal of a file from the user's hard disk or network server, even if it has a file attribute of read-only.
cmoOverwrite	Overwrite any existing version of the file on the user's hard disk or network server. Or, overwrite the entry in the .INI file.
cmoAppend	Append the file to the existing file on the user's hard disk or network server rather than replacing the existing file. Or, append the value to the existing value in the .INI file.
cmoPrepend	Prepend the value to the existing .INI value rather than replacing the value.
cmoNone	No command option flag is specified.
cmoAll	All command option flags are specified.

Legal Notice

Olivetti is a registered trademark of Ing. C. Olivetti.

PostScript is a registered trademark of Adobe Systems, Inc.

R4000 is a trademark of MIPS Computer Systems, Inc.

Roland is a registered trademark of Roland Corporation.

SCSI is a registered trademark of Security Control Systems, Inc.

Epson is a registered trademark of Seiko Epson Corporation, Inc.

Silicon Graphics is a registered trademark and OpenGL is a trademark of Silicon Graphics, Inc.

Stacker is a registered trademark of STAC Electronics.

Tandy is a registered trademark of Tandy Corporation.

Unicode is a registered trademark of Unicode, Incorporated.

UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company, Ltd.

VAX is a trademark of Digital Equipment Corporation

Yamaha is a registered trademark of Yamaha Corporation of America.

Paintbrush is a trademark of Wordstar Atlanta Technology Center.

Microsoft Win32 Developer's Reference

You have requested information from the **Microsoft Win32 Developer's Reference**. One or more of these help files is not available on your system.