The Windows NT Command Shell

By About the Author

Archived: 2026-04-05 22:33:21 UTC

By Tim Hill

Chapter 2 from *Windows NT Shell Scripting*, published by MacMillan Technical Publishing

- **Command Shell Basics** The command shell is introduced, along with the basic command syntax. The difference between CMD.EXE and COMMAND.COM is explained.
- **Starting a Command Shell** Learn how to start, stop and nest command shells. Command line switches accepted by the command shell are also detailed.
- **Command Line Editing** Find out about the various command text-editing features, including less well-known features such as command completion and the command history.
- **DOSKEY and Command Macros** The features available through the DOSKEY command are described here, including command macros.
- **Launching Applications from the Shell** This section provides complete details of how the shell launches applications, including how they are located and how file associations are used. The role of the PATH and PATHEXT variables is explored, as well as the use of the START command.
- **Controlling Script Output** Various commands allow control over script output, such as ECHO and TITLE.
- **Command Redirection** Find out how to capture command output, redirect command input, and send the output of one command to the input of another.
- **Running Multiple Commands** Complex command lines execute multiple commands; learn about multi-line commands and the use of parentheses to group commands together.
- **Using Command Filters** The filter commands, such as MORE and SORT, are described here.
- **The Windows NT Command Scheduler** This section describes the Windows NT command scheduler, the Schedule Service and the AT command.

The previous chapter introduced the Windows NT console window. By far the most common use for a console window is to execute the Windows NT command shell. The command shell defines the Windows NT scripting language and is responsible for processing scripts, as well as commands typed at the keyboard.

The command shell is a console application. As explained in chapter 1, console applications are Windows NT applications that interact via a console window, rather than via GUI elements such as windows and dialog boxes. When you start a command shell, Windows NT creates a console window for that shell. All commands that are run from within a shell (including other command shells) share the same console window for output. The only exception to this is the START command, which can be used to create additional console windows.

To start a default command shell in a console window, click the Start button, select the Programs item, and then select the Command Prompt command. The section "Starting a Command Shell" details additional ways to start a command shell.

By default, command shells run in Interactive mode. In this mode, the shell displays a prompt and then waits for keyboard input. When a command line is entered, it is immediately interpreted and then executed. After execution completes, the shell displays another prompt, and the whole sequence begins again. This continues until the EXIT command ends the command shell session. Figure 2.1 shows this basic command execution sequence.

If a command entered is the name of a script file, the command shell switches to Script mode, and begins reading, interpreting, and executing commands from the specified script file. Figure 2.2 shows the command execution sequence while the shell is in script mode. Execution of the script ends when the shell reaches the end of the script file. At this point, the shell returns to interactive mode, displays another command prompt, and waits for keyboard input.

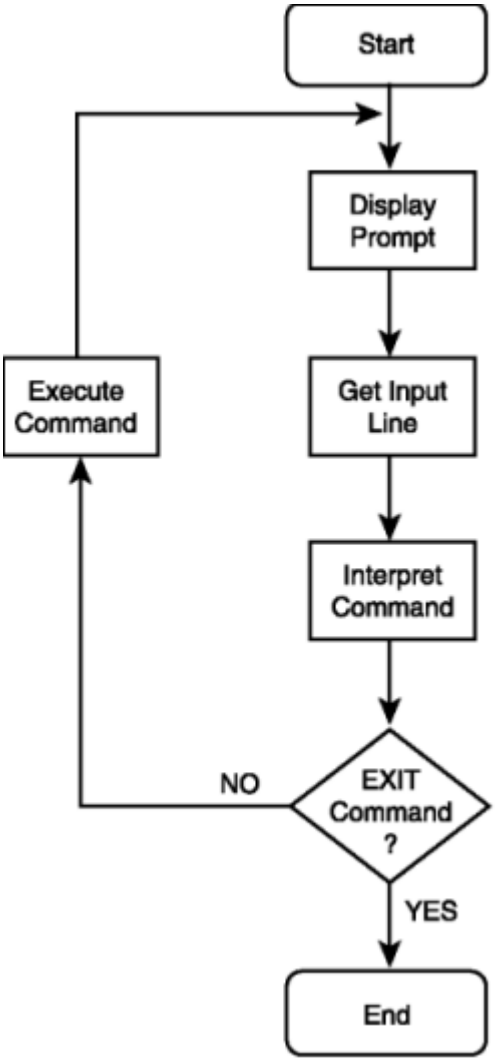

Figure 2.1: Interactive mode shell execution sequence

The Command Shell Prompt

The command shell displays the shell prompt when it is ready to accept a command in interactive mode. The default shell prompt is the current drive and path name, followed by a > character. When the command shell is in script mode, prompts are only displayed if command echo is enabled. The ECHO command, described in the section "Controlling Script Output," controls command echo.

The PROMPT command changes the command shell prompt. Follow the PROMPT command with the text of the new shell prompt. For example:

```
1. C:\>prompt ???
2. ???
```

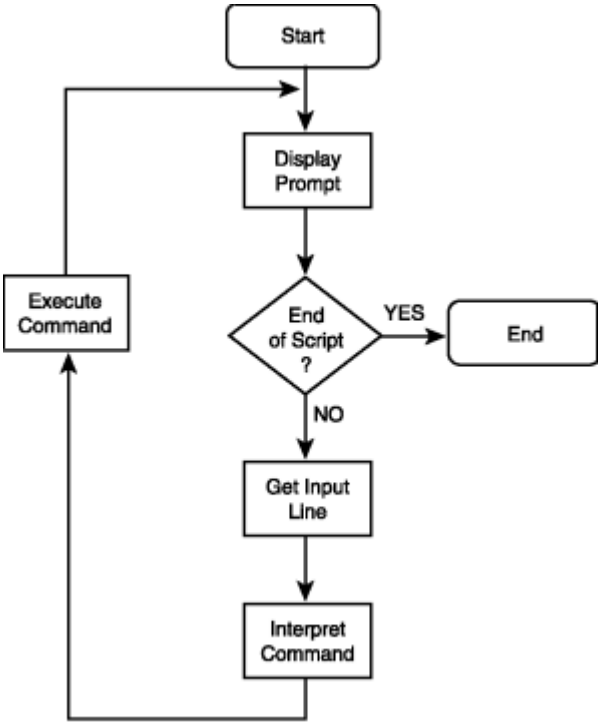

Figure 2.2: Script mode shell execution sequence

This example changes the command prompt to three question marks. Notice that the shell uses this prompt on line 2 in the example. To revert to the default prompt, enter a PROMPT command without any prompt text. For example:

```
1. ???prompt
2. C:\>
```

The prompt is restored to the default.

The command prompt text can contain special characters used as placeholders for additional information. For example, $T in the command text is replaced with the current time when the prompt is displayed, and $G is replaced with the > character. The following example shows this:

```
1. C:\>prompt $T$G
2. 14:23:50.35>
```

Using $G to represent the > character might seem unnecessary, as this character can simply be typed. However, this does not work, because the command shell reserves certain characters for special purposes. These reserved shell characters all have $c equivalents, so that they can be used in command prompts. For example, $A is used for an ampersand, $L for a < character, and so on. Table 2.1 shows the complete set of special characters recognized in a command shell prompt.

Table 2.1 Special Characters in a Shell Prompt

Character	Description
$A	Ampersand character.
$B	Pipe (|) character.
$C	Left parenthesis.
$D	Current date.
$E	Escape code (ASCII 27).
$F	Right parenthesis.
$G	Greater than character.
$H	Backspace character.
$L	Less than character.
$N	Current drive letter.
$P	Current drive letter and directory path.
$Q	Equal sign.
$S	Space.
$T	Current time.

Character	Description
$V	Windows NT version number.
$_	New line.
$$	Dollar sign.
$+	A series of "+" signs, corresponding to the number of pushed directories on the PUSHD stack. See the PUSHD command in Part III.
$M	The remote name (UNC name) for the current drive.

The $+ special character works in conjunction with the PUSHD and POPD commands (see Part III). These commands manage a push-down stack of directories and drives, and the $+ special character displays a sequence of + characters in the command prompt, one for each level in the push-down stack.

If the current drive is a network drive, the $M special character displays the remote UNC name of this drive in the form \\server\share. If the current drive is a local drive, then $M does not display anything.

The current prompt text is stored in the PROMPT environment variable. Changing the prompt changes the value of this variable and vice versa. Thus, these two commands have the same effect:

```
1. C:\>prompt [$p]
2. C:\>set PROMPT=[$p]
```

Simple Command Syntax

As described in chapter 1, simple shell commands consist of a command name followed by any required arguments. The command name and arguments (if any) are separated by a space. A command is always entered in response to a shell prompt. Figure 2.3 shows a simple shell command.

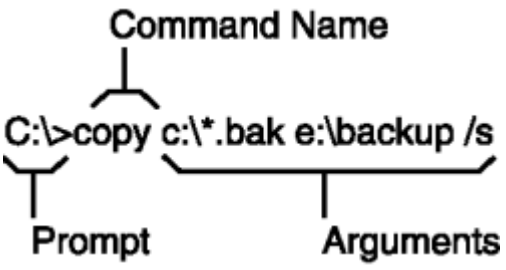

Figure 2.3: Simple command syntax

In Figure 2.3, the command name describes the action to be performed, while the arguments provide additional information needed to carry out this action. The syntax of the command arguments is specific to each command. However, there are a number of well-established conventions for command argument syntax. These are only conventions, however, and each individual command is free to interpret the supplied arguments however it chooses:

- First, multiple arguments are normally separated from one another by spaces. In Figure 2.3, the command has three arguments, c:*.bak, e:\backup, and /s. Occasionally, other characters are used as argument separators. For example, the COPY command can use + characters to separate multiple filenames.
- Second, any argument that contains spaces or begins or ends with spaces must be enclosed in double quotes. This is particularly important when using long file and directory names, which frequently contain one or more spaces. If a double-quoted argument itself contains a double quote character, the double quote must be doubled. For example, enter "Quoted" Argument as """Quoted"" Argument".
- Third, command switches always begin with a slash / character. A switch is an argument that modifies the operation of the command in some way. Occasionally, switches begin with a + or - character. Some switches are global, and affect the command regardless of their position in the argument list. Other switches are local, and affect specific arguments (such as the one immediately preceding the switch).
- Fourth, all reserved shell characters not in double quotes must be escaped. These characters have special meaning to the Windows NT command shell. The reserved shell characters are:

  & | ( ) < > ^

  To pass reserved shell characters as part of an argument for a command, either the entire argument must be enclosed in double quotes, or the reserved character must be escaped. Prefix a reserved character with a carat (^) character to escape it. For example, the following command example will not work as expected, because < and > are reserved shell characters:

```
C:&gt;echo <dir>
The syntax of the command is incorrect.
```

  Instead, escape the two reserved characters, as follows:

```
C:&gt;echo ^<dir^>
<dir>
```

  Typically, the reserved shell characters are not used in commands, so collisions that require the use of escapes are rare. They do occur, however. For example, the popular PKZIP program supports a -& switch

to enable disk spanning. To use this switch correctly under Windows NT, -^& must be typed.

Tip The carat character is itself a reserved shell character. Thus, to type a carat character as part of a command argument, type two carats instead. Escaping is necessary only when the normal shell interpretation of reserved characters must be bypassed.

- Finally, the maximum allowed length of a shell command appears to be undocumented by Microsoft. Simple testing shows that the Windows NT command shell allows very long commands—in excess of 4,000 characters. Practically speaking, there is no significant upper limit to the length of a command.

Be aware that a command shell is not an MS-DOS command prompt, even though it shares the same icon. The Windows NT command shell is a full 32-bit Windows NT console application that resides in the CMD.EXE executable file. The MS-DOS command prompt is a 16-bit DOS application that resides in the COMMAND.COM executable file. Because COMMAND.COM is a 16-bit DOS executable, Windows NT executes this shell within a Windows NT virtual DOS machine (VDM). COMMAND.COM is supplied primarily for compatibility with MS-DOS.

Surprisingly, however, both the Windows NT and the MS-DOS shells have almost identical features. Here is a sample IF command entered into a Windows NT command shell, followed by the command output:

```
1. C:\>if /i a==A echo MATCH
2. MATCH
```

The IF command compares the letter "a" to the letter "A" and echoes MATCH if they compare. The /I switch compares the two letters without regard to letter case. Therefore, not surprisingly, the command echoes MATCH.

Here is the same IF command entered into an MS-DOS 16-bit COMMAND.COM shell (running on Windows NT on the same computer):

```
1. C:\>if /i a==A echo MATCH
2. MATCH
```

The output of both commands is identical. This is surprising, as the /I switch is a new feature of the Windows NT command shell (CMD.EXE) which is not understood by COMMAND.COM when running under actual MS-DOS.

This behavior reveals a quite subtle feature of Windows NT that is very important. The 16-bit MS-DOS shell (COMMAND.COM) that ships with Windows NT is specially designed for Windows NT. When a command is entered for execution by this shell, it does not actually execute it. Instead, it packages the command text and sends it to a 32-bit CMD.EXE command shell for execution. Because all commands are actually executed by CMD.EXE (the Windows NT command shell), the 16-bit shell inherits all the features and facilities of the full Windows NT shell.

You can see COMMAND.COM automatically execute a 32-bit CMD.EXE shell by using the Windows NT Task Manager application. Follow this procedure:

1. Right-click in an empty area in the taskbar. From the popup window select the Task Manager command to start Task Manager.
2. Click the Processes tab to display the list of running Windows NT processes.
3. Click the Start button in the taskbar and select the Run command.
4. In the Open box, type COMMAND. Then click OK to start a 16-bit command prompt.
5. Now examine the list of processes in the Task Manager window. You should see at least one NTVDM process. This is the Windows NT VDM, which NT starts to allow COMMAND.COM to execute.
6. In the COMMAND.COM window, enter any command which generates lengthy output (DIR /S is a good choice). When the output begins scrolling, press Ctrl+S to pause the command.
7. Switch to Task Manager and examine the list of processes again. Notice that a new CMD.EXE (command shell) is running. Figure 2.4 shows a typical Task Manager window.
8. Switch to the COMMAND.COM window and press Ctrl+S again. Wait for the command to complete.
9. Switch to Task Manager again. Notice that the CMD.EXE shell has disappeared from the Processes list.
10. To close the COMMAND.COM window enter an EXIT command.

This simple experiment shows that every command entered for execution, regardless of the shell used, is ultimately executed by CMD.EXE, the Windows NT command shell.

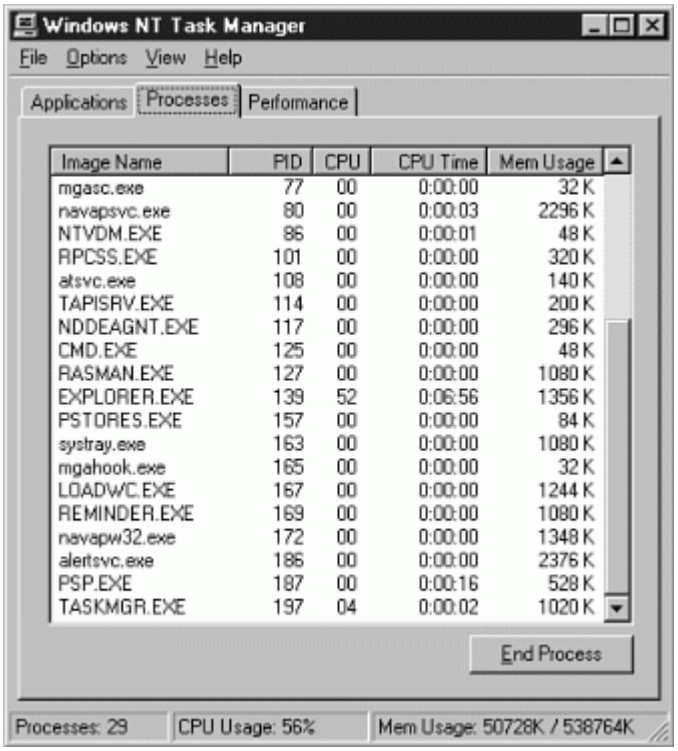

Figure 2.4: Task Manager showing automatic CMD.EXE execution

You can start a Windows NT command shell in a number of different ways:

- Select the Programs item in the Start menu, and then select the Command Prompt command.
- Select the Run command in the Start menu, enter CMD in the Open box and click OK.
- Enter a CMD command in an existing command shell.
- Enter a START command in an existing command shell, and specify CMD as the command to execute.
- Double-click a .BAT or .CMD script file in an Explorer window.

All these methods except the last start a command shell in Interactive mode. The last method starts the command shell in Script mode.

Like most other commands, the command shell CMD.EXE accepts several switches that control various shell options. Table 2.2 shows these switches, which are described in detail in the "Command Reference" in Part III.

Table 2.2 CMD.EXE Switches

Switch	Description
/X	Enables command extensions (default).
/Y	Disables command extensions.
/A	Command output to files or pipes will be ANSI (default).
/U	Command output to files or pipes will be Unicode.
/T	Sets foreground and background window colors.
/C	Executes command specified and then terminate shell.
/K	Executes command specified and then prompts for additional commands.

To specify shell switches, place them after the CMD command on the command line. It is not possible to directly enter shell switches if the shell is started from the Start menu Command Prompt command, or by double-clicking

a .BAT or .CMD script file (the first and last methods in the preceding list).

Without any switches, a command shell starts in Interactive mode with command extensions enabled. Command extensions are certain features added to the command shell since Windows NT was first released. In general, these extensions are backward compatible, but they can occasionally cause older Windows NT and MS-DOS scripts to fail. In this case, the /Y switch disables the extensions and, in effect, runs an MS-DOS compatible command shell.

The /C and /K switches directly execute a command. The command to execute is specified following the switch. For example:

```
1. C:\>cmd /c echo Run this...
2. Run this...
```

All command line arguments following the /C or /K switch describe the command to execute. The /C switch executes the specified command, and the command shell then terminates. The /K switch executes the specified command, and the shell then enters Interactive mode. The /K switch is particularly useful to set up a command shell to a predetermined state before the first command prompt is displayed.

The /C and /K switches both accept any valid shell command, including the name of a script file to execute. In this case, the shell enters script mode and executes the specified script. This is the method the Windows NT Explorer uses to start a script file when it is opened. For example, if you double-click on the file SCRIPT.BAT, Explorer actually executes this command:

```
cmd /c script.bat
```

In Interactive mode, the EXIT command terminates a command shell. If a console window was created for the command shell, the console window closes. If the command shell was invoked from within another program (including another command shell), that program regains control of the console window.

Tip Closing the console window also terminates a command shell. This method is valid only for the 32-bit CMD.EXE shell. The 16-bit COMMAND.COM shell should not be terminated in this manner.

The EXIT command can also be used in script mode. The command should be used with care, however, as the shell terminates immediately, effectively aborting script execution. Scripts should normally end by executing to the end of the script file, or by the GOTO :EOF command (see Chapter 4).

If a script file is running in a shell that was started using the /C switch, the command shell terminates when the script reaches the end of the file.

A command shell can be started from within another command shell, commonly referred to as nesting a shell. This can be done either with the START command or by entering a CMD command directly at the command prompt. Nesting shells is useful if a command or script must be executed using different options to the current shell. For example, command extensions can be disabled in the nested command shell, or the command redirection mode changed. For example:

```
1. c:\>prompt [$p]
2.
3. [c:\]cmd /y
4. Microsoft(R) Windows NT(TM)
5. (C) Copyright 1985-1996 Microsoft Corp.
6.
7. [c:\]prompt
8. c:\>exit
9. [c:\]
```

The first PROMPT command changes the prompt. A new command shell is then invoked with the /Y switch (to disable command extensions). Notice that the new command shell inherits the prompt from the previous shell. The second PROMPT command returns the prompt to the default, as can be seen in the subsequent shell prompt (C:\>). The EXIT command exits the nested command shell, reverting to the previous (original) shell. Notice that the prompt reverts to the modified form, showing that the second command shell has indeed terminated.

The inheritance of the current command prompt by nested command shells is not a special feature of the shell. It occurs because the current command prompt is stored in an environment variable, and all commands executed from a shell inherit the current shell environment. The environment and inheritance is described fully in Chapter 3.

Nested command shells are useful when used with the /C switch. Suppose the script SXT.BAT was designed to run correctly only if command extensions are enabled. The following command executes this script correctly regardless of the state of command extensions for the current shell:

```
C:\>cmd /x /c sxt.bat
```

Another use of nested command shells—capturing all script output—is described in the upcoming section, "Controlling Script Output."

When in interactive mode, the command shell provides a rich set of tools to assist in command entry and editing. Most of these tools are applicable to all console applications that use line by line input. The command-line editing tools include:

- Basic character editing, such as Backspace and Delete.
- Template editing, which operates with a copy of the previously entered command.
- Command history editing, which allows quick recall of previously entered commands.
- Command completion, which automatically completes partially entered file and directory names.
- Command macros, which are described in "DOSKEY and Command Macros."

The "Command Line Editing" section in the "Command Reference" describes the full set of command-line editing commands.

Basic character editing includes familiar commands such as Backspace, which erases the character to the left of the cursor, and Delete, which erases the character to the right of the cursor. Use the Left Arrow or Right Arrow keys to move the cursor left or right one character. Use these keys with Ctrl to move the cursor left or right one word. Use Home to move the cursor to the start of the command line, and End to move to the end of the command. Press the Esc key to delete the entire command, and press Enter to execute the command. The cursor does not have to be at the end of the line when Enter is pressed.

By default, the character editor operates in Overwrite mode. New characters typed overwrite characters at the current cursor location. Pressing the Insert key switches to Insert mode. In Insert mode, characters to the right of the cursor are shifted to the right as new characters are entered at the cursor location. Repeatedly pressing Insert toggles back and forth between the two modes. The cursor changes shape to indicate the mode—Insert mode is indicated by a larger, block shaped cursor. The mode resets when the Enter key is pressed.

The default Insert/Overwrite mode can be changed from Overwrite to Insert either by the DOSKEY command or via the Console Window property sheet's Options tab, as described in Chapter 1.

The second editing method, template editing, works in conjunction with a hidden command buffer, the template, which contains a copy of the most recently entered command. Template editing is present for compatibility with MS-DOS, and the newer command history editing commands generally make template editing obsolete.

The template F4 command is useful to quickly delete blocks of characters in a command line. Position the cursor at the first character to delete and press F4. A popup prompt appears. Enter a single character, and all characters in the command line from the cursor up to (but not including) the first instance of the specified character are deleted.

Command history editing provides quick access to a list of recently executed commands, also known as the history buffer. Commands in this buffer can be re-executed, or recalled to the command line and edited as desired.

Each time a shell command executes, the command text is added to the command history buffer, and is then available for later recall. The maximum number of stored commands is set by the DOSKEY command or via the Console Window property sheet's Options tab, as described in Chapter 1.

The F7 key displays the command history buffer. Press F7, and a popup window appears containing the contents of the history buffer, with the oldest commands listed first.

Figure 2.5 shows a console window with a command history popup displayed. Each command is numbered, starting with 1 for the oldest command. The Up Arrow and Down Arrow keys move up and down through the list of commands, while the Page Up or Page Down keys move to the start or end of the command list. The Esc key closes the history buffer popup window without executing a command.

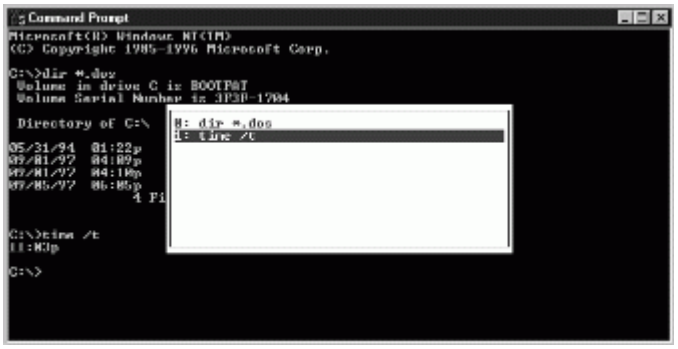

Figure 2.5: The command history buffer popup window

To recall a command from the history buffer and execute it:

1. Press F7 to display the command history popup window.

2. Use the up arrow and down arrow keys in the popup window to highlight the command to execute.

3. Press Enter to execute the command.

When editing a command, use the up and down arrow keys to directly recall commands from the history buffer for editing. This differs from using these keys within the command history popup window. Press the keys repeatedly to cycle through all commands in the command buffer. The Up Arrow key moves back through the command history (starting with the most recently entered command) while the Down Arrow moves forward through the command history. The Page Up key moves to the oldest command in the history buffer, and the Page Down key moves to the newest command in the buffer. Once a command is recalled to the command line, it can be edited as needed and then executed by pressing the Enter key.

The F8 key searches the history buffer. Begin by typing the first few characters of the command that is required into the command line. Press the F8 key, and the first (most recent) command in the history buffer that begins with the characters entered is recalled to the command line. Repeatedly pressing F8 continues to search for additional matching commands in the history buffer.

The F9 key recalls commands from the history buffer by number. Command numbers are displayed in the command history popup, accessible by pressing F7. Press F9, and a popup window appears asking for the command number. Enter the command number, and the specified command is recalled to the command line ready for editing or execution. The F9 key can also be used while the command history buffer popup window is displayed.

Windows NT supports command history editing for any program which reads keyboard input on a line-by-line basis. Each program maintains its own command history, so multiple command shells each have an independent command history buffer.

Command completion automatically completes the typing of long file or directory names. Before this feature can be used, it must be enabled by changing a setting in the Windows NT registry. The value to alter is:

```
HKEY_CURRENT_USER \Software \Microsoft \Command Processor\CompletionChar
```

The default value for this is 0x0, which disables command completion. To enable command completion, set this value to the ASCII code for the key to use as the command completion key. The Tab key, value 0x9, is a good choice since it is not normally used in commands. The value is located in the per-user portion of the registry, and is therefore enabled on a per-user basis.

Once enabled, command completion operates as follows:

1. When entering a file or directory name, type the first few characters of the name and press the chosen command completion key (e.g. the Tab key).
2. The shell searches the current directory for a file or directory name which begins with the specified characters, and replaces the typed characters with the complete file or directory name.
3. Pressing the command completion key again cycles through all additional matching file names.

For example, suppose the current directory (C:\BOOK) contains three files, CHAP01.DOC, CHAP02.DOC and CHAPTER.DOC. Type the following, but do not press Enter:

```
C:\book>type chap
```

Press the command completion key. The command line immediately changes to:

```
C:\book>type "c:\book\chap01.doc"
```

The partial file name is replaced with the first file name found in the directory. The full path name is substituted, and the whole argument is placed in double quotes. This ensures that a valid argument is created even if the file name contains spaces. Press the command completion key again, and the command line changes to:

```
C:\book>type "c:\book\chap02.doc"
```

The command completion character can also be used when no characters of a file or directory name are entered. Type the command up to the file or directory name, making sure that the last character entered is a space. Then press the command completion key. In this case, all file and directory names in the current directory are presented in sequence.

Command completion is a very useful editing tool, particularly when long file and directory names are in use.

The DOSKEY command provides command line control of various advanced shell command editing features. In MS-DOS, the DOSKEY command was a Terminate-and-Stay Resident (TSR) that needed to be loaded before its features were available for use. In Windows NT, these features are built into the shell and are always available; DOSKEY merely provides an interface to control them.

The DOSKEY command actually provides three distinct functions:

- Insert mode control
- command history buffer management
- command macro management

Insert mode control sets the initial Insert/Overwrite edit toggle. For example, this command sets the initial toggle to Insert mode:

```
C:\>doskey /insert
```

This command sets the initial toggle to overwrite mode:

```
C:\>doskey /overstrike
```

The initial state of this toggle is set for a console window using the console window property sheet Options tab, as described in Chapter 1.

The following DOSKEY command clears the command history buffer (described in the previous section "Command History Editing") of all commands:

```
C:\>doskey /reinstall
```

This command optionally sets a new size for the command history buffer. For example:

```
C:\>doskey /reinstall /listsize=100
```

This example clears the command history buffer and sets the number of commands in the buffer to 100. The command history buffer size can also be set for a console window using the Console Window property sheet Options tab, as described in Chapter 1.

The final set of DOSKEY commands provides control of command shell macros. Command macros are shorthand commands for longer, more complex shell commands. Macros can reduce typing or, in scripts, provide simple one-line functions. For example, to define a new macro named LS, which executes a DIR command, enter:

```
C:\>doskey ls=dir
```

The command shell will now accept LS as an alias for the DIR command. For example:

```
1. C:\>ls
2.  Volume in drive C is BOOTFAT
3.  Volume Serial Number is 3F3F-1704
```

```
4.
5.  Directory of C:\
6.
7. 05/31/94  01:22p  54,645 COMMAND.DOS
8. ...etc.
```

Command macros, like all shell commands, are not case sensitive. The simple definition of LS is not a direct replacement for the DIR command, however, because it does not accept arguments. For example:

```
C:\>ls *.dos
```

will not work as expected. The *.DOS argument is ignored, and the entire directory is displayed. To pass arguments to a macro, define the macro as follows:

```
C:\>doskey ls=dir $*
```

The special $* argument acts as a placeholder for all arguments entered on the command line. The LS macro is now an exact replacement for the DIR command. The DOSKEY section in the "Command Reference" in Part III lists additional special arguments that can be used within macro definitions.

Obviously, directly replacing one command by an alias is of limited use. However, macros can replace more complex commands. For example:

```
C:\>doskey ls=dir /od $*
```

This definition of the LS macro always provides directory listings sorted in date order. The /OD switch tells the DIR command to sort files according to date and time.

The following macro definition creates a command that displays the user name of the currently logged-on user:

```
1. C:\>doskey myname=for /f "delims=\ tokens=2" %i in ('whoami') do @echo %i
2. C:\>myname
3. TimHill
```

The MYNAME macro uses an advanced FOR command (described in Chapter 4, "Control Flow, Procedures, and Script Nesting") and the WHOAMI command to extract the user name. Typing MYNAME is far simpler than entering the complex FOR command. Macros are also very useful when combined with the various compound command symbols described in the section "Running Multiple Commands." For example:

```
C:\>doskey dircount=dir $* $B find "<DIR>" /c
```

This example creates a DIRCOUNT macro that counts the number of directories in a specified directory. The $B argument acts as a placeholder for the pipe symbol (|). Alternatively, the pipe symbol can be entered directly by escaping it using a ^ character. Pipes are described in the section "Command Redirection."

Macros can be used with any Windows NT application which accepts line-by-line command input. However, each macro is explicitly defined for a specific application. By default, macros are defined for use by the command shell, CMD.EXE. The /EXENAME switch defines a macro for another application. For example, to define an EXIT macro for use with FTP, enter:

```
C:\>doskey /exename=ftp.exe exit=bye
```

The /MACROS switch lists defined macros for the command shell, for example:

```
1. C:\>doskey /macros
2. ls=dir $*
```

The /EXENAME switch lists macros for a specific application, for example:

```
1. C:\>doskey /macros /exename=ftp.exe
2. exit=bye
```

The /MACROS:ALL switch lists macros for all applications. The macros for each application are listed under the application name, which is placed in brackets.

The /MACROFILE switches reads a set of macro definitions from a file, for example:

```
C:\>doskey /macrofile=macros.mac
```

The format of macros in the macro file exactly matches the output of the DOSKEY command with the /MACROS:ALL switch. Therefore, it is possible to define a set of macros interactively and then use DOSKEY to create a macro file. Later, the macros can be recalled. For example:

```
1. C:\>doskey ls=dir /od $*
2. C:\>doskey /exename=ftp.exe exit=bye
3. ...etc.
4. C:\>doskey /macros:all >macros.mac
```

This example uses command output redirection (described in the section "Command Redirection") to capture the macro definitions to the file MACROS.MAC. Later, the /MACROFILE switch can reload the macros. One convenient way to do this is to use the CMD.EXE /K switch. For example:

```
C:\>cmd /k doskey /macrofile=macros.mac
```

This starts a new shell and pre-loads all the macros in MACROS.MAC. If this command is placed in a shortcut, a new command shell can be started and a set of macros loaded automatically without any typing.

As described at the beginning of this chapter, a basic shell command is composed of a command name followed by zero or more arguments. The command name specifies the action to be performed, and the arguments provide additional data used by the command to perform this action. In order to carry out the action specified by the command, the command shell must decode the command. This section describes the steps taken by the shell to decode each command.

Commands can be broken into two main categories: internal and external. An internal command is one that is built-in to the shell itself. An external command is one that is contained within an executable file on the disk. For example, the COPY command is internal, while the XCOPY command is external (it is contained within the XCOPY.EXE file). Generally, the distinction between internal and external commands is unimportant, except that internal commands have no associated executable file.

The following are all of the Windows NT internal commands:

ASSOC	CALL	CHDIR/CD	CLS
COLOR	COPY	DATE	DIR
DPATH	ECHO	ENDLOCAL	ERASE/DEL
EXIT	FOR	FTYPE	GOTO
IF	MKDIR/MD	MOVE	PATH
PAUSE	POPD	PROMPT	PUSHD
REM	RENAME/REN	RMDIR/RD	SET
SETLOCAL	SHIFT	START	TIME
TITLE	TYPE	VER	

Two environment variables are intimately associated with shell command execution: PATH and PATHEXT.

The PATH environment variable defines the Windows NT search path. The search path is a list of directories that are searched when the command shell attempts to locate an executable file. Separate directories in the path list with semi-colons. For example, a typical path might contain:

```
d:\winnt40\system32;d:\winnt40;d:\ntreskit;c:\bin;c:\dos
```

The PATH command manipulates the PATH environment variable, although the variable can also be directly manipulated via the SET command (see Chapter 3 for a description of the SET command). To set a new system path, follow the PATH command with a new path list. For example:

```
C:\>path c:\bin;c:\scripts;d:\winnt
```

This PATH command tells Windows NT to search the C:\BIN, C:\SCRIPTS, and D:\WINNT directories for executable files.

Enter a PATH command without any arguments to display the current search path. For example:

```
1. C:\>path
2. PATH= d:\winnt40\system32;d:\winnt40;d:\ntreskit;c:\bin;c:\dos
```

One common use of the PATH command is adding a new directory to the search path. To do this, specify the existing search path as part of the new path. For example, to add a new directory, C:\NEWDIR, to the start of the path, use this command:

```
C:\>path c:\newdir;%PATH%
```

Use this command to add the same directory to the end of the path:

```
C:\>path %PATH%;c:\newdir
```

The PATH variable is initialized from the following sources of information:

- The System Environment, which is set via the Control Panel System icon.
- The User Environment, which is set via the Control Panel System icon.
- Any PATH statements in AUTOEXEC.BAT (if parsing is enabled).

At logon time, path information from the sources listed above is concatenated together to form the initial path. After logging on, the PATH command is used to alter the path.

The PATHEXT environment variable defines the list of file extensions checked by Windows NT when searching for an executable file. Like the PATH variable, semi-colons separate individual items in the PATHEXT variable.

The default value of PATHEXT is .COM;.EXE;.BAT;.CMD. The PATHEXT variable is manipulated via the SET command. For example, to add the .PL extension, use the following command:

```
C:\>set PATHEXT=%PATHEXT%;.pl
```

The following section describes how Windows NT and the command shell use the PATH and PATHEXT variables.

When a command is submitted for execution (either by typing or as part of a script), the shell performs the following actions:

1. All parameter and environment variable references are resolved (see chapter 3).
2. Compound commands are split into individual commands and each is then individually processed according to the following steps (see the section "Running Multiple Commands" for details of compound commands). Continuation lines are also processed at this step.
3. The command is split into the command name and any arguments.
4. If the command name does not specify a path, the shell attempts to match the command name against the list of internal shell commands. If a match is found, the internal command executes. Otherwise, the shell continues to step 5.
5. If the command name specifies a path, the shell searches the specified path for an executable file matching the command name. If a match is found, the external command (the executable file) executes. If no match is found, the shell reports an error and command processing completes.
6. If the command name does not specify a path, the shell searches the current directory for an executable file matching the command name. If a match is found, the external command (the executable file) executes. If no match is found, the shell continues to step 7.
7. The shell now searches each directory specified by the PATH environment variable, in the order listed, for an executable file matching the command name. If a match is found, the external command (the executable file) executes. If no match is found, the shell reports an error and command processing completes.

In outline, if the command name does not contain a path, the command shell first checks to see if the command is an internal command, then checks the current directory for a matching executable file, and then checks each directory in the search path. If the command name does contain a path, the shell only checks the specified directory for a matching executable file.

If the command name includes a file extension, the shell searches each directory for the exact file name specified by the command name. If the command name does not include a file extension, the shell adds the extensions listed in the PATHEXT environment variable, one by one, and searches the directory for that file name. Note that the shell tries all possible file extensions in a specific directory before moving on to search the next directory (if there is one).

For example, the following command explicitly specifies the path, command name, and file extension:

```
C:\>c:\bin\edit.exe
```

This command executes the program EDIT.EXE found in the directory C:\BIN. If the program is not found, the shell reports an error.

This example omits the path to EDIT.EXE:

```
C:\>edit.exe
```

To execute this command, the shell searches the current directory and then each directory in the search path until EDIT.EXE is found, or reports an error if the file is not found.

This example omits the path and file extension:

```
C:\>edit
```

To execute this command, the shell searches the current directory and then each directory in the search path. Assuming that the PATHEXT variable contains .COM;.EXE;.BAT;.CMD, each directory is searched for EDIT.COM, EDIT.EXE, EDIT.BAT and EDIT.CMD before the shell moves on to the next search directory.

Once the command shell resolves the command name either to an internal command or an external executable file, it executes the command as follows:

- If the command is internal, the shell executes it directly.
- If the command is a 16-bit or 32-bit Windows GUI executable program, the shell runs the program but does not wait for the command to complete.
- If the command is a 32-bit console application, or a 16-bit MS-DOS application, the shell runs the command in the current console window and waits for the command to complete.
- If the command is a script file (.BAT or .CMD), the shell switches to script mode and begins executing the script.
- If the command is a document or data file name associated with an application, the shell executes the appropriate application. The shell applies the previous rules based upon the type of the application associated with the data or document file. See the following section for more information on file associations.

Notice that the command shell does not wait for GUI applications to complete execution before it continues. This behavior can be modified using the START command, described in the following section.

Windows NT provides a database, in the system registry, which allows files to be associated with a particular application. The primary use of this association is called automatic application launching. This feature is used

extensively in the GUI environment: whenever a data file or document is double-clicked, the associated application automatically launches, and the specified file or document then opens within that application.

Tip File associations can also be edited using Windows NT Explorer (using the View, Options command). However, the commands presented here have a finer degree of control than that provided by Explorer. For example, Explorer provides no easy way to delete an individual association.

File associations are also applicable to the command shell environment. For example, suppose Microsoft Word is installed on a computer. During installation of Word, .DOC files are associated with the Word application. Once this is done, it is possible to launch Word and open a .DOC file from a shell command simply by typing its name. For example:

```
C:\>c:\docs\letter.doc
```

This command launches Word and opens the file C:\DOCS\LETTER.DOC. If the file is in the current directory, the path is not required. For example:

```
C:\docs>letter.doc
```

In fact, the command shell applies the same rules when opening a document or data file as it does when searching for any external command. Thus, in the previous example, the shell looks for LETTER.DOC in the current directory and in all directories specified by the search path. If the C:\DOCS directory is added to the search path, the file LETTER.DOC can be opened in Word from any directory, merely by typing its name. For example:

```
1. C:\docs>path c:\docs;%PATH%
2. C:\docs>cd ..
3. C:\>letter.doc
```

As previously noted, the file association database is maintained in the Windows NT Registry. Therefore, changes made to file associations are persistent—they are retained even after the computer is reset. In addition, the database is maintained in the HKEY_LOCAL_COMPUTER portion of the registry. Thus, changes made to the file association database effect all users of a particular computer.

The file association database works with three items of information:

- file extensions
- file types
- launch commands

A file extension is the familiar file name suffix after the last period in the file name. For example, MYSCRIPT.BAT has a file extension of .BAT. Most file extensions are one, two, or three characters long. A file type is a name for a particular class of file. For example, Windows NT scripts might be assigned a file type of

Windows.Script. A launch command is a prototype command used to launch the associated application. For example, NOTEPAD.EXE %1 is a typical launch command.

When a document or data filename is specified as a command name, the shell uses the file association database to launch the correct application. It does this as follows:

1. The shell extracts the file extension from the specified document file. It then searches the database for a matching file association. If no match is found, the shell reports an error and command processing completes.
2. The shell then obtains from the database the name of the file type associated with the file extension.
3. The shell then searches the database again for the specified file type. If no match is found, the shell reports an error and command processing completes.
4. The shell obtains from the database the launch command associated with the file type.
5. The shell now parses the launch command, replacing any parameters with arguments specified in the original shell command.
6. Finally, the shell executes the parsed command. Typically, this launches the associated application.

This procedure is best understood through an example. Consider this command:

```
C:\docs>letter.doc
```

The shell first extracts the file extension, .DOC, and searches the database for the file type associated with this file extension. Typically, this yields a file type such as Word.Document.8. The shell now uses this file type to search for the launch command. Assume this is WINWORD.EXE %1. The %1 in this command is a formal parameter, which the shell replaces with the document file name (LETTER.DOC). So the final launch command is WINWORD.EXE LETTER.DOC. When this command executes, Word runs, and the LETTER.DOC file is opened.

File types are thus an intermediary between a file extension and a launch command. They exist to allow multiple file extensions to be associated with the same launch command. For example, a paint program might need to associate .BMP, .JPG, .TIF and .TGA files with a launch command. Rather than entering the same launch command in the database four times, the program uses a single file type and launch command, and then associates the file extensions with this type. Any subsequent changes made to the launch command are then automatically applied to all associated file extensions.

The ASSOC, FTYPE and ASSOCIATE commands are used to manipulate the file association database. The ASSOC command connects a file extension with a file type, and the FTYPE command connects a file type with a launch command.

Use the FTYPE command to create or edit a file type and associate it with a launch command. For example:

```
C:\>ftype REXX.File=c:\rexx\rexx.exe "%1"
```

This creates the file type REXX.File and associates the prototype command C:\REXX\REXX.EXE "%1" with the type. Notice the use of double quotes around the %1 parameter. This ensures that the command is handled correctly even if the specified document name contains spaces.

Tip When defining a prototype command, it is advisable to include the full path name of the executable file, unless the directory containing the executable will always be part of the search path.

The FTYPE command can also display the current launch command for a file type. For example:

```
1. C:\>ftype REXX.File
2. REXX.File=REXX "%1"
```

Finally, the command and file type can be deleted. For example:

```
1. c:\>ftype REXX.File=
2. c:\>ftype REXX.File
3. File type 'REXX.File' not found or no open command associated with it.
```

Once a file type and launch command are set up, the ASSOC command associates file extensions with that file type. For example:

```
1. C:\>assoc .rex=REXX.File
2. .rex=REXX.File
```

The current association of a file type is displayed using ASSOC. For example:

```
1. C:\>assoc .rex
2. .rex=REXX.File
```

Finally, the file extension association can be deleted. For example:

```
C:\>assoc .rex=
```

Notice that for both FTYPE and ASSOC, specifying the file type or file extension only displays the current association. Specifying the file type or file extension with a trailing = character deletes the current association.

The ASSOCIATE [RK] command provides a shorthand method to perform an ASSOC and FTYPE in one step. While not as versatile as the ASSOC/FTYPE combination, it is easier to use. ASSOCIATE directly associates a file extension with an application, providing the necessary file type and launch command automatically. For example:

```
C:\>associate .rex c:\rexx\rexx.exe
```

This command directly associates .REX files with the REXX.EXE application. The association can be deleted using the following command:

```
C:\>associate .rex /d
```

More information on the ASSOC, FTYPE and ASSOCIATE commands can be found in the "Command Reference."

By using a combination of the PATH and PATHEXT variables, along with the application association database, complete integration of new command types into the command shell is possible.

For example, suppose a REXX interpreter, REXX.EXE, is installed in the C:\REXX directory. REXX scripts can then be executed using commands such as:

```
C:\>c:\rexx\rexx myrexx.rex arg1 arg2 arg3
```

This executes the REXX interpreter, REXX.EXE, which then interprets and runs the script MYREXX.REX. In this example, ARG1 etc. are arguments passed to the script.

By adding an association to the file association database, the invocation of the REXX interpreter can be made implicit. For example:

```
1. C:\>ftype REXX.File=c:\rexx\rexx.exe "%1" %*
2. C:\>assoc .rex=REXX.File
```

These commands create the needed associations, so that a .REX file launches the REXX.EXE interpreter. Notice that by including the full path name in the prototype command the interpreter executes without adding the C:\REXX directory to the search path. In addition, the %1 parameter is placed in double quotes so that .REX filenames containing spaces are correctly handled. Finally, the special %* parameter represents all additional arguments following the first (in this case ARG1 ARG2 ARG3). Now, the original script command can be simplified to:

```
C:\>myrexx.rex arg1 arg2 arg3
```

By adding the file extension .REX to the list of file extensions specified with PATHEXT, the .REX file extension can also be made implicit. For example:

```
1. C:\>set PATHEXT=%PATHEXT%;.REX
2. C:\>myrexx arg1 arg2 arg3
```

At this point, entering a REXX script command to execute is as convenient as entering any native shell command.

Finally, the MYREXX.REX script can be stored in a central directory, and that directory added to the system search path. For example:

```
1. C:\>mkdir c:\rexxscripts
2. C:\>move myrexx.rex c:\rexxscripts
3. C:\>set PATH=%PATH%;c:\rexxscripts
4. C:\>myrexx arg1 arg2 arg3
```

At this point, the MYREXX.REX script can be executed in any directory, just by typing its name.

Tip File association database changes are persistent. However, changes to environment variables are not. Therefore, the changes to the PATH and PATHEXT variables shown in the prior examples are lost when the system is shutdown, or the current shell terminated. To make these changes persistent, edit the PATH and PATHEXT variables in the Environment tab of the Control Panel System applet.

Previous sections described how the command shell implicitly interprets and executes a basic command. The START command explicitly executes a shell command, and provides additional control over how the command is handled by Windows NT.

The syntax of the START command is:

```
START ["title"] [switches] command-name [args]
```

The first item following the START command is an optional window title, enclosed in double quotes. By default, the START command executes the specified command in a new window. In this case, the title text is used as the window title.

Following the title are zero or more switches that control the operation of the START command. A command-name must then be present, which specifies the command to run. Following the command name are zero or more command arguments, which are passed to the specified command.

Tip All START switches and options must appear before the command-name. Switches and options placed after the command-name are passed, unaltered, to the command being started.

Without any of the optional switches, the START command executes the specified command. It does this as follows:

- If the command is a 16-bit or 32-bit Windows GUI executable program, the START command runs the program but does not wait for the command to complete.
- If the command is a 32-bit console application, or a 16-bit MS-DOS application, the START command runs the command in a new console window but does not wait for the command to complete.

- If the command is a script file (.BAT or .CMD), or an internal command, the START command executes a new command shell (CMD.EXE) in a new console window. The script or internal command is executed using the /K switch. The START command does not wait for the command or script to complete.
- If the command is a document or data file name associated with an application, the START command executes the appropriate application. The START command applies the previous rules based upon the type of the application associated with the data or document file.

Notice that the operation of the START command differs from the default command shell sequence:

- First, the START command never waits for the command to complete.
- Second, all console commands start in a new console window, rather than the current console window.
- Finally, if the command is a script file or internal shell command, the START command executes the command using a new command shell (CMD.EXE) with the /K switch. This means that, after the script of internal command completes, the new command shell does not terminate, but instead enters interactive mode. For example:

```
C:\>start "New Window" dir
```

This command executes a DIR command in a new console window, giving the new window the title New Window. Since DIR is an internal command, the new command shell enters Interactive mode and prompts for additional commands after the DIR command completes.

To over-ride this behavior, explicitly execute a new command shell and use the /C switch. For example:

```
C:\>start "New Window" cmd /c dir
```

In this example, the command-name for the START command is actually CMD. The /C switch and DIR command are arguments to the CMD.EXE command shell. The START command therefore executes the command CMD /C DIR in a new console window. When the DIR command completes, the new command shell terminates, and the new console window closes.

The START command is most useful when one or more of the optional switches are used. All START switches follow the window title (if used) and precede the command name.

The /D switch specifies a new current drive and directory for the command. Without /D, the command inherits the drive and directory of the command shell. Follow the /D switch with the new drive and directory. For example:

```
C:\>start /dc:\book dir
```

The /I switch controls environment inheritance. Normally, all commands executed by the command shell inherit a copy of the current environment (this is described in detail in Chapter 3). The /I switch causes the command to

inherit the environment as it existed when the command shell was first started. Thus, any changes made to the environment within the current command shell are not passed to the executed command.

The /MIN and /MAX switches specify the initial state of the new window created by the START command. These switches apply to all applications and commands, including 16 and 32-bit GUI applications. Without /MIN or /MAX, the START command creates the new window using Windows NT default settings. The /MIN switch creates the new window minimized, that is, as a task bar button. The /MAX switch creates the new window maximized, that is, occupying the entire screen.

The /LOW, /NORMAL, /HIGH and /REALTIME switches set the priority class for the command or application. By default, the START command executes all commands or applications at normal priority. The /LOW switch executes the application at low priority, while the /HIGH switch executes the application at high priority. The /REALTIME switch executes the application at real-time priority.

Troubleshooting Tip Use of the /REALTIME switch is strongly discouraged, as its use can compromise Windows NT stability.

Using /LOW to execute an application at low priority is a very useful way of running a low priority background task. For example:

```
C:\>start "Cleanup" /low /min cmd /c cleanup.bat
```

This command executes the CLEANUP.BAT script at low priority. The /MIN switch is used so that the script appears only as a button on the task bar.

As described above, the START command does not wait for the new command to complete; a new command prompt appears immediately and new commands can be executed at once. When used in a script, the next line in the script executes immediately. The /WAIT switch makes the START command wait for the command to complete before continuing. This switch applies to all commands and applications, including GUI applications.

Finally, the /B switch executes the command without creating a new console window. This switch is application only to internal commands and external console applications—it is ignored if the command specifies a GUI application. Using /B also implies the /WAIT switch.

The command shell provides the following simple commands to control script output:

- The REM command, which is used for script comments.
- The CLS command, which clears the console window.
- The COLOR command, which controls colors used in the console window.
- The TITLE command, which changes the console window title bar text.
- The @ command, which controls command echo on a line-by-line basis.
- The ECHO command, which controls command echo and also displays text.

- The NOW [RK] command, which displays time-stamped text.

The REM (remark) command is the simplest script command because it does nothing. Any text can follow the REM command. REM commands are used for shell comments, and should be used liberally within a script (for example, to clarify complex script commands and document the script logic). The REM command also suppresses the meaning of reserved shell characters within the remark text. For example, the following is a valid remark:

```
C:\>rem This is valid in a REM statement: &, &&, ||, ^
```

Normally, the shell assigns special meanings to &, && etc., but in a remark command these characters are treated as regular text.

Troubleshooting Tip Because the REM command suppresses the normal interpretation of special shell characters, REM commands cannot appear within a multi-line command. For example, the following is invalid:

```
1. if "%X%"=="ABC" (
2.     rem An illegal comment!
3.     goto :EXIT
4. )
```

The shell combines multi-line commands (like the one above) into a single line before executing them. Therefore, the shell "sees" the command like this:

```
if "%X%"=="ABC" ( rem An illegal comment! &  goto :EXIT )
```

The REM command will include all the text on the line, up to and including the closing parenthesis. The GOTO command is not seen as a command at all, but instead as additional comment text.

The CLS command clears the current console window and positions the cursor to the top left of the window. Subsequent command output begins at the top of the window and works down the screen. The screen is cleared using the current window colors. The CLS command is useful when a script needs to present un-cluttered output.

The COLOR command sets the text and background colors for the console window. When a console window is started, it uses the colors set using the Console Window property sheet Colors tab, as described in chapter 1. The COLOR command without any arguments returns the console window to these default colors. With a single argument, the COLOR command sets the text and background colors. The argument must be two characters long. The first character specifies the background color and the second character specifies the text color. Table 2.3 shows the color codes used.

Table 2.3 COLOR Command Color Codes

Code	Color	Code	Color
0	Black	8	Gray
1	Blue	9	Light Blue
2	Green	A	Light Green
3	Aqua	B	Light Aqua
4	Red	C	Light Red
5	Purple	D	Light Purple
6	Yellow	E	Light Yellow
7	White	F	Bright White

For example, this command sets a pleasing white on blue color combination:

```
C:\>color 17
```

The TITLE command changes the title bar of the console window to the specified text. The text in the corresponding button on the task bar is also changed. The TITLE command is useful to show the progress of long or complex scripts. It is superior to ECHO for this purpose as it does not scroll text in the console window, and the text is visible in the task bar even when the console window is minimized. For example:

```
C:\>title Backup Drive C:...
```

The @ command controls command echo on a line-by-line basis. By default, the shell displays each command in a script before it is executed. Prefix a command with an @ character to suppress command echo. For example:

```
@title Welcome to the script!
```

The @ command can be placed within a compound command (described in the upcoming section "Running Multiple Commands") to control echo of individual portions of the command, although the utility of this is questionable. For example, this compound command will not echo when executed as part of a script:

```
@(echo starting...)&(title Script phase 1)
```

However, the command

```
(@echo starting...)&(title Script phase 1)
```

echoes as:

```
C:\>( )&(title Script phase 1)
```

The @ command controls command echo on a command-by-command basis. The ECHO command controls command echo for an entire script. This command disables command echo:

```
C:\>echo off
```

This command enables command echo:

```
C:\>echo on
```

To display the current state of command echo, enter:

```
1. C:\>echo
2. ECHO is on.
```

The ECHO command is typically used at the start of a script to disable command echo for the duration of the script. For example, many scripts begin with this line:

```
@echo off
```

This disables echo for the entire script. The echo of the ECHO command is itself suppressed by using the @ command. (The example above is the first special script line that was shown in the section "Special Script Lines" in chapter 1.) Once echo is disabled (or enabled) in a script, the echo state is maintained within the script and within script procedures and nested scripts. The only exception to this is nested scripts executed via a CMD or START command. These scripts run in a new command shell, which always starts with echo enabled.

Echo can be disabled interactively at a command prompt by entering an ECHO OFF command. In this case, the command prompt is suppressed until echo is enabled again. Typed commands are still echoed during entry,

however.

The current echo state is used when the command shell switches from Interactive mode to script mode to begin executing a script. Thus, if echo is enabled interactively, the script begins with echo enabled, and vice versa. However, the shell remembers the original interactive echo state, and recalls it when the script completes. Therefore, after a script ends, the echo state reverts to the value it had before the script began execution.

The ECHO command is also used to echo arbitrary text to the console window. For example:

```
C:\>echo Hello, world!
```

If command echo is disabled, the command itself is not echoed, but the text specified in the ECHO command is echoed. When used this way, the ECHO command is the Windows NT script equivalent of the PRINT statement found in many languages. Typically, ECHO is used with environment variable substitution, which is described in Chapter 3.

Tip The ECHO command cannot be used to echo an empty line. For example:

```
1. C:\>echo
2. ECHO is on.
```

As can be seen, the ECHO command displays the current state of command echo, not an empty line. The nearest equivalent is to echo something inconsequential, such as a single period. For example:

```
1. C:\>echo .
2. .
```

The NOW [RK] command also displays arbitrary text. However, NOW prefixes the text with the current time and date. This is useful when the time taken to execute a command must be monitored. For example:

```
1. E:\workdir>now Start
2. Wed Oct 21 12:20:07 1997 -- Start
3. E:\workdir>echo Quick command
4. Quick command
5. E:\workdir>now End
6. Wed Oct 21 12:20:18 1997 -- End
7. E:\workdir>
```

Most console applications and commands generate output, and many accept input. This input or output is in the form of a stream of characters (either ANSI or Unicode). Applications generally work with up to three streams, as follows:

- The command input stream is used by the application or command to read input. By default this stream comes from keys typed at the keyboard.

- The command output stream is used by the application or command to display output. By default, this stream is displayed in the console window.
- The command error output stream is used by the application or command to display errors. By default, this stream is displayed in the console window.

The default stream input and output provides normal interactive command operation: input is obtained from the keyboard, and output is displayed in the console window. Note that the distinction between the command output stream and the command error output stream is somewhat arbitrary. An application or command can direct output to whichever stream it wishes. Typically, however, normal output is sent to the command output stream, and errors are sent to the command error output stream.

The command shell provides facilities to change the default stream input and output. These facilities are accessed by placing special command redirection symbols in a command. Table 2.4 shows the command redirection symbols.

Table 2.4 Command Redirection Symbols

Symbol	Description
>file	Redirects command output to the file specified. You can also use a standard device name such as LPT1, CON, PRN or CONOUT$ as the file name. Any preexisting contents of the file are lost.
>>file	Redirects command output to the file specified. If the file already exists, all command output is appended to the end of the file.
<file	Redirects command input from the file specified. You can also use a standard device name such as CON or CONIN$.
2>file	Redirects command error output to the file specified. You can also use a standard device name such as LPT1, CON, PRN or CONOUT$ as the file name. Any preexisting contents of the file are lost.
2>&1	Redirects command error output to the same location as command output. This makes any command output redirection also apply to command error output.

Symbol	Description
cmd1 | cmd2	Pipes the command output of cmd1 to the command input of cmd2. Multiple pipe characters are allowed, creating a chain of commands, each sending output to the next command in the chain.

Command redirection symbols are not visible to the command. The shell processes them before the command is executed and they are not passed as arguments to the command. The <, >, and | symbols are reserved shell characters. If these symbols must be passed as command arguments, instead of being used as redirection symbols, then they must be escaped using the ^ character.

The > redirection symbol redirects command output to the specified file. For example:

```
C:\>dir >c:\dir.txt
```

This example creates a text file C:\DIR.TXT containing the output of the DIR command. The > symbol can be placed anywhere in the command, but is typically placed at the end of the command. A space is permitted between the > symbol and the file name. If the file specified by the redirection symbol already exists, any existing contents are deleted before the command is executed.

Tip Only one command output redirection symbol is allowed per command. It is not possible, for example, to duplicate command output by redirecting the command output to multiple files.

The >> redirection symbol redirects command output to the specified file, but concatenates the output onto the end of the file. For example:

```
C:\>dir >> c:\dir.txt
```

This example adds the output of the DIR command to the end of the file C:\DIR.TXT. If the file specified does not exist, it is created.

The < redirection symbol redirects command input from the specified file. For example:

```
C:\>sort <c:\dir.txt
```

This command sorts the contents of the file C:\DIR.TXT and then displays the result. A space is permitted between the < symbol and the file name. The < symbol is used less frequently than > and >>, since there are few shell commands which accept console input.

The 2> redirection symbol redirects command error output to the specified file. For example:

```
C:\>dir 2>c:\error.txt
```

This example redirects the error output of the DIR command to the file C:\ERROR.TXT. A space is permitted between the 2> symbol and the file name. Notice that this example does not redirect regular command output, so the directory listing is still displayed. Only error output (if any) is captured to the file.

The redirection symbols can be combined in a single command. For example:

```
C:\>sort <c:\dir.txt >c:\sortdir.txt 2>c:\error.txt
```

The 2>&1 redirection symbol redirects command error output to command output. This means that command error output is sent to the same destination as command output. For example:

```
C:\>dir >c:\dir.txt 2>&1
```

This command sends both command output and command error output to the file C:\DIR.TXT.

The | (pipe) redirection symbol sends the command output of cmd1 to the command input of cmd2. For example:

```
C:\>dir | sort
```

This example sends the command output of the DIR command to the command input of the SORT command. The output from the SORT command is then displayed. Alternatively, the SORT output can be sent to another command. For example:

```
C:\>dir | sort | more
```

This example sends the command output of the DIR command to the command input of the SORT command. Then, the command output of the SORT command is sent to the command input of the MORE command. Finally, the command output of the MORE command is displayed.

When the command shell processes a pipe (|) symbol, it actually runs both commands specified simultaneously. The right hand command is suspended until the left hand command begins generating command output. Then, the left hand command wakes up and processes the output. When this output has been processed, the command is again suspended until more input is available. The synchronization of both commands is handled automatically by the command shell and Windows NT.

The pipe symbol is both a redirection symbol and a compound command symbol. Compound commands are discussed in the next section.

Command redirection effects are inherited by nested commands. If a command starts with its command output redirected, and this command then starts additional commands, these commands inherit the same redirection as the parent command. This is frequently used with nested shells to capture all script output to a file. For example:

```
C:\>cmd /c myscript.bat >result.txt
```

This command executes the script MYSCRIPT.BAT in a new shell. Since the new command shell has its command output redirected to the file RESULT.TXT, all commands run by the shell (i.e. those in the script) also have their output redirected.

By default, all command input and output is processed as ANSI (or ASCII) characters. However, if the shell is started with the /U switch (see the "CMD" section in the "Command Reference"), command input and output is processed as Unicode characters. In this case, if a command generates ANSI output, the shell automatically converts ANSI command output to Unicode.

In previous sections, the simple "command and arguments" syntax of a shell command was described. The shell also supports compound commands, where a command line specifies more than one command to execute. Compound commands are indicated by special compound command symbols. Table 2.5 shows the compound command symbols.

Table 2.5 Compound Command Symbols

Symbol	Description
cmd1 & cmd2	Executes command cmd1, then command cmd2. Additional commands can be added using additional ampersand symbols.
cmd1 && cmd2	Executes command cmd1, then executes command cmd2 only if cmd1 completed successfully.
cmd1 || cmd2	Executes command cmd1, then executes command cmd2 only if cmd1 did not complete successfully.
( )	Use parentheses to indicate the nesting of complex multi-command sequences. Also used in IF ... ELSE commands.

The &, |, (, and) symbols are reserved shell characters. If these symbols must be passed as command arguments, instead of being used as compound command symbols, then they must be escaped using the ^ character.

The simplest compound command symbol is &. The & symbol separates multiple commands on a single command line. For example:

```
1. C:\>echo Command 1 & echo Command 2
2. Command 1
3. Command 2
```

This example shows two ECHO commands on a single line. When & is used to separate multiple commands, the commands execute one at a time, starting with the first command. Each command runs to completion before the next command executes.

Any number of commands can be placed on a single line using the & symbol. For example:

```
C:\>dir c:\bin >files.txt & dir c:\dos >>files.txt & type files.txt
```

This example accumulates the results of two DIR commands into the file FILES.TXT and then displays the contents of this file. This example also shows that command redirection symbols can be used with each individual command in a compound command.

The && and || compound command symbols provide conditional command execution. The first (left) command is executed. If the && symbol is used, the second command executes only if the first command completed successfully. If the || symbol is used, the second command executes only if the first command did not complete successfully. A command completes successfully if it returns an exit code of 0 or no exit code at all. Exit codes are discussed in chapter 3. For example:

```
C:\>verify on || echo Verify command failed!!
```

Since the VERIFY command executed successfully, the ECHO command was not executed. However, in this example:

```
1. C:\>verify ox || echo Verify command failed!!
2. An incorrect parameter was
3. entered for the command.
4. Verify command failed!!
```

The ECHO command was executed because the VERIFY command syntax was incorrect, causing the command to exit with a non-zero error code.

Multiple commands can be chained together using additional && and || compound command symbols. For example:

```
C:\>dir && copy a b && echo OK!
```

The COPY command executes only if the DIR command succeeds, and the ECHO command executes only if the COPY command succeeds.

The parentheses symbols (and) are used to resolve command ambiguities and indicate the binding of compound command and redirection symbols. They are also used in the IF command and to specify multi-line commands.

For example, the following command collects two directory listings into the file FILES.TXT:

```
C:\>dir *.exe >files.txt & dir *.com >>files.txt
```

The following command might appear to do the same, but in fact it does not work:

```
C:\>dir *.exe & dir *.com >files.txt
```

This second example fails because the command redirection symbols have a higher precedence than the compound command symbols. The shell interprets this command as follows:

```
1. C:\>dir *.exe
2. C:\>dir *.com >files.txt
```

This displays the result of the first DIR command in the console window—which is not the desired effect.

The second example can be corrected by using parentheses to alter the binding of the various symbols, as follows:

```
C:\>(dir *.exe & dir *.com) >files.txt
```

This compound command executes two DIR commands, one after the other. The output of both commands is redirected into the file FILES.TXT. The parentheses change the binding of the command symbols. Notice that this command is far cleaner than the earlier example, and that the file FILES.TXT is only specified once in the command.

Parentheses can be nested to specify arbitrarily complex compound commands. For example:

```
C:\>((echo command1) & (echo command2)) && (echo command 3)
```

An opening parenthesis can be placed anywhere on a command where a command-name is expected. When a command is enclosed in parentheses, either a closing parenthesis or a compound command symbol marks the end of the command. For example:

```
1. C:\>echo (command)
2. (command)
3. c:\>(echo command)
4. command
```

Notice that the second ECHO command did not echo the closing parenthesis. In this case, the shell treats this as the end of the command, and this parenthesis is not passed as part the ECHO arguments. In the first case, the command itself does not begin with a parenthesis, and so the end of line marks the end of the command. In this case, the entire (command) text is passed to the ECHO command.

Parentheses can also be used to enter multi-line commands. If a command line ends with one or more sets of unbalanced parentheses, the command line is assumed to continue on the next line. If the command was entered interactively, the shell prompts for more input until all parentheses balance. If the command is part of a script, the shell reads additional script lines until all the parentheses balance. For example:

```
1. C:\>(
2. More?echo command1
3. More?echo command2
4. More?)
5. command1
6. command2
```

The first line consists only of an open parenthesis. The shell detects this, and prompts for more input. Next, two ECHO commands are entered. Finally, a closing parenthesis balances the opening parenthesis and the command is complete. The shell then executes the compound command, which executes the two individual ECHO commands.

Individual commands do not span lines in multi-line commands. The end of a physical line always terminates a simple command (either as typed or as entered in a script file). Notice in the preceding example how the end of the physical line terminated each ECHO command.

Compound commands and multi-line commands are particularly useful with IF and FOR statements. These commands are described in Chapter 4. For example:

```
1. if exist *.bak (
2.     echo Deleteing *.BAK files...
3.     del *.bak
4. )
```

The IF command executes the following command if one or more .BAK files exist in the current directory. Parentheses are used to execute a compound multi-line command. In this example, the ECHO command displays a warning, and then the DEL command deletes the files. The indentation here is not mandatory, but does help to indicate the flow of control.

Tip Using a parenthesis, even when not strictly needed, is a useful way to increase script readability, and makes explicit the command execution precedence rules.

The previous sections showed how individual commands can be combined using command redirection and compound command symbols. Although command redirection can be used with any command, it is most effective with commands that are specifically designed as command filters. Generally, a command filter reads command input, permutes, or processes the input in some manner, and then writes the permuted input to its command output. Command filters are typically connected to other commands and each other via the pipe (|) redirection symbol.

Command filters are frequently used in scripts to extract specific information needed by a script. Typically, a FIND command can filter the output of a command, extracting only the line (or lines) which contain the required

information. The script can then further process this filtered data. Many of the sample scripts in Part II use this technique.

Windows NT provides four command filters:

- The MORE command, which is used to paginate command output.
- The SORT command, which can sort command output alpha-numerically.
- The FIND command, which filters lines that contain a specified text string.
- The CLIP [RK] command, which captures command output to the Windows NT clipboard.

The MORE command breaks its command input into pages. A page is the same number of lines as the console window. For example:

```
C:\>dir | more
```

This command displays the current directory, one page at a time. Press the spacebar to advance to the next page. The MORE command supports several switches to control the output format. These are detailed in the MORE section of the "Command Reference."

The SORT command sorts its command input alpha-numerically, using the ASCII collating sequence. For example:

```
C:\>sort <data.txt
```

This command sorts the contents of file DATA.TXT and displays the result. By default, the SORT command sorts in ascending order. The /R switch sorts in reverse (descending) order. Lines are sorted based on the data at the start of each line. Use the +n switch to sort based on data starting at column n on each line. For example:

```
C:\>dir | sort +14
```

This command sorts the output of the DIR command based on data starting at column 14 on each line.

The FIND command filters command input, passing to its command output only those lines which contain a specified string. For example:

```
C:\>dir | find "<DIR>"
```

This command filters only directory lines from the output of the DIR command. The FIND command supports several switches to control the filtering process—these are detailed in the FIND topic of the "Command Reference." One useful switch is /C, which outputs only a count of the lines which match the string, instead of the lines themselves. For example:

```
C:\>dir | find "<DIR>" /c
```

This command counts the number of directories in the current directory.

Tip Windows NT also supports the more powerful FINDSTR command, which provides more sophisticated filtering capabilities. However, FINDSTR works with files as input. FINDSTR is described in the "Command Reference."

The CLIP [RK] command places its command input into the clipboard. For example:

```
C:\>dir | clip
```

This command dumps the output of the DIR command into the clipboard. Once in the clipboard, the text can be pasted into any Windows application.

Windows NT provides a special feature called the Schedule Service that can execute any command or application periodically or at a specified time and date. Scheduled commands can execute even if there is no user logged on to the computer. The Schedule Service is therefore ideal for running periodic maintenance tasks, such as file backups. For example, the REPL.BAT sample script in Part II uses the Schedule Service to provide periodic file replication services.

The Schedule Service is controlled through several shell commands. These commands can control the service either on the local computer or any computer on the network (assuming that the user has sufficient rights to access that computer). The ability to control the Schedule Service remotely is particularly useful for system administrators managing many servers.

The two most common mistakes made when using Windows NT involve the setup and use of the Schedule Service:

- First, the Schedule Service, like all services, logs on by default as the LocalSystem account. Typically, this account has insufficient rights to perform many operations (such as accessing network drives).
- Second, because the Schedule Service operates independently of a user logon, drive mappings and other per-user settings are not necessarily available when a scheduled command executes. For example, the environment available to a command executed by the schedule service is restricted to the system environment set via Control Panel.

Fortunately, both these problems are easily solved. Creating a special account exclusively for the Schedule Service (named, for example, ScheduleService) solves the first problem. This account is then assigned whatever access rights and group memberships are needed to ensure that any scheduled commands are successful. Explicitly mapping all required resources within a scheduled script (such as network drives) solves the second problem. Alternatively, use UNC names directly in commands.

For example, assume a logon script maps drive X: to \\SERVER\COMMON. This command, which operates correctly when used interactively, fails when used with the Schedule Service:

```
copy x:\myfiles\*.bak c:\backups
```

The Schedule Service does not execute the logon script, so drive X: is not available. The first solution to this problem uses the NET command to explicitly map drive X: in the script. For example:

```
1. net use x: \\server\common
2. copy x:\myfiles\*.bak c:\backups
3. net use x: /delete
```

The second solution uses UNC names directly. For example:

```
copy \\server\common\myfiles\*.bak c:\backups
```

Virtually all built-in commands, and most external commands, accept UNC names as well as path names as arguments.

The final stage to prepare the Schedule Service for use is starting the service. Typically, the start-up options for this service are changed to require the service to automatically start whenever Windows NT boots. Both of these operations can be accomplished via the Control Panel Services applet. Figure 2.6 shows the Services applet running. The Schedule Service is named Schedule. Click the Start button to start the service, or the Stop button to stop the service.

Click the Startup button to set service startup options. Figure 2.7 shows the startup options dialog box. Typically, the Startup Type should be Automatic, and the Log On As option should be the account setup specifically for the schedule service.

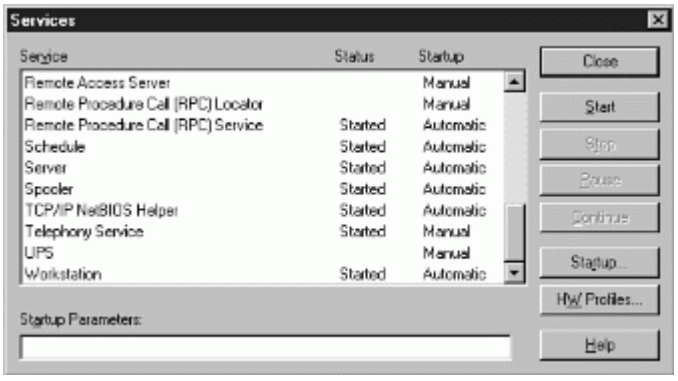

Figure 2.6: The Control Panel Services applet

Once the Schedule Service is running, commands can be scheduled using either the AT command or the SOON [RK] command. Refer to the "Command Reference" for a detailed description of these commands. The Windows NT Resource Kit also provides a GUI interface to the Schedule Service, called WINAT.

Figure 2.7: Schedule Service startup options

By default, all AT commands manage the schedule service on the local computer. Specify the computer name as the first argument to manage the service on another computer. For example, this command lists all scheduled commands on the current computer:

```
1. C:\>at
2. Status  ID  Day  Time  Command Line
3. -------------------------------------------------------------------------------
4. 0  Each  F  8:00 PM  "e:\Tools\NAVNT\NAVWNT" /L
```

This command lists all scheduled commands on the computer CRAFT:

```
C:\>at \\craft
```

Enter an AT command and specify the execution time and the command to execute to schedule a command for execution. For example:

```
C:\>at 23:00:00 /every:monday cmd /c sysbkup.bat
```

This command schedules the command CMD /C SYSBKUP.BAT to execute at 11 p.m. The command is executed every Monday whenever the system clock indicates the time is 11 p.m.

The SOON [RK] command schedules a command to execute a certain number of seconds in the future. For example:

```
C:\>soon 600 "cmd /c sysbkup.bat"
```

This command schedules the command CMD /C SYSBKUP.BAT to execute in 10 minutes (600 seconds) time. SOON can be used to setup a script for periodic execution. Simply place a SOON command as the first line of the script. For example, place this line as the first line in MYSCRIPT.BAT:

```
soon 600 "cmd /c myscript.bat"
```

Then manually schedule MYSCRIPT.BAT to execute once. The first time MYSCRIPT.BAT executes, it reschedules a new copy of itself to run in ten minutes. Thus, every 10 minutes, a new copy of MYSCRIPT.BAT is executed.

Tim Hill is an independent software developer specializing in systems software and operating system architectures.

Click to order