Hamilton C shellä

User Guide and Reference Manual

Release 2.2

October, 1993

Hamilton Laboratories, 13 Old Farm Road, Wayland, MA 01778-3117
Phone 508-358-5715 • FAX 508-358-1113
MCI Mail 389-0321 • Internet 3890321@mcimail.com
BIX hamilton • CompuServe 70034,2025 • Telex 6503890321

Table of Contents

Preface

Thank you for purchasing and using Hamilton C shell. Our goal and guarantee is your satisfaction.

Hamilton C shell is an advanced command processing language for OS/2 and Windows NT. It's a professionally-oriented language for manipulating files, processes and threads and connections between these objects. As a language, it offers what we think of as the human characteristics of language: a useful vocabulary and grammar, a limitless freedom of expression and the ability to describe and relate events in time. Most important, it projects your influence into the future by allowing you to easily describe you want done even if what you want is quite complex and dependent on future events.

Hamilton C shell is a full implementation of the C shell language popular on engineering workstations. It was created specifically for OS/2 protected mode and meticulously ported to Windows NT. Not one of the more than 105,000 lines of code in the current release was created on or ported from anything but OS/2 or NT.

This product complies with accepted standards for the language and with the conventions of OS/2 and NT. Users with previous experience with the standard OS/2, NT or DOS command processors or the original Unix C shell should find enough reasonably familiar language constructs and features to make the product immediately productive.

Douglas A. Hamilton
Wayland, Massachusetts
December 9, 1988
(Last revised October 4, 1993)

IMPORTANT -- READ CAREFULLY BEFORE OPENING. By opening this sealed package, you indicate your acceptance of the following Hamilton Laboratories License Agreement.

Hamilton Laboratories License Agreement

This is a legal agreement between you, the end user, and Hamilton Laboratories. By opening this sealed package, you are agreeing to be bound by the terms of this agreement. If you do not agree to the terms of this agreement, promptly return the unopened package and any accompanying items for a full refund.

HAMILTON LABORATORIES SOFTWARE LICENSE

1. GRANT OF LICENSE. Hamilton Laboratories grants to you the right to use one copy of the enclosed Hamilton Laboratories software program (the ''SOFTWARE'') on a single terminal connected to a single computer (i.e., with a single CPU). You may not network the SOFTWARE or otherwise use it on more than one computer or computer terminal at the same time.

2. COPYRIGHT. The SOFTWARE is owned by Hamilton Laboratories or its suppliers and is protected by United States copyright laws and international treaty provisions. Therefore, you must treat the SOFTWARE like any other copyrighted material (e.g., a book or musical recording) except that you may either (a) make a reasonable number of copies of the SOFTWARE solely for backup purposes or (b) transfer the SOFTWARE to a single hard disk provided the original and any other copies are kept solely for backup or archival purposes. You may not copy the written materials accompanying the software.

3. OTHER RESTRICTIONS. You may not rent or lease the SOFTWARE, but you may transfer the SOFTWARE and accompanying written materials on a permanent basis provided you retain no copies and the recipient agrees to the terms of this Agreement. You may not reverse engineer, decompile or disassemble the SOFTWARE. If SOFTWARE is an update, any transfer must include the update and all prior versions.

4. DUAL MEDIA SOFTWARE. If this SOFTWARE package contains both 3 1/2'' and 5 1/4'' disks, you may use only the disks appropriate for your single-user computer. You may not use the other disks on another computer or loan, rent, lease, or transfer them to another user except as part of the permanent transfer (as provided above) of all SOFTWARE and written materials.

LIMITED WARRANTY

LIMITED WARRANTY. Hamilton Laboratories warrants that the SOFTWARE will perform substantially in accordance with the accompanying written materials for a period of 90 days from the date of purchase. Some states do not allow limitations on the duration of an implied warranty, so the above may not apply to you.

CUSTOMER REMEDIES. Hamilton Laboratories' entire liability and your exclusive remedy shall be, at Hamilton Laboratories' option, either (a) return of the price paid or (b) repair or replacement of the SOFTWARE that does not meet this Limited Warranty and which is returned to Hamilton Laboratories with a copy of your receipt. During the first 90 days from the date of purchase, if you determine that the SOFTWARE is unsatisfactory in any way, you may return it with proof of purchase and a written description of why the SOFTWARE was unsatisfactory for a full refund.

NO OTHER WARRANTIES. Hamilton Laboratories disclaims all other warranties, either express or implied, including, but not limited to implied warranties of merchantability and fitness for a particular purpose, with respect to the SOFTWARE and accompanying written materials. This limited warranty gives you specific legal rights. You may have others, which vary from state to state.

NO LIABILITY FOR CONSEQUENTIAL DAMAGES. In no event shall Hamilton Laboratories or its suppliers be liable for any damages whatsoever (including, without limitation, damages for loss of business profits, business interruption, loss of business information, or other pecuniary loss) arising out of the use of or inability to use this Hamilton Laboratories product, even if Hamilton Laboratories has been advised of the possibility of such damages. Because some states do not allow the exclusion or limitation of liability for consequential or incidental damages, the above limitation may not apply to you.

This Agreement is governed by the laws of the State of Massachusetts.

Should you have any questions concerning this Agreement, or if you wish to contact Hamilton Laboratories for any reason, please write: Hamilton Laboratories Customer Service, 13 Old Farm Road, Wayland, MA 01778-3117.

Hamilton C shellä

Introduction

Hamilton C shell is a language for interactively using OS/2 and Windows NT. Compared to the standard OS/2 and NT command processors, it provides a vocabulary and grammar that allows much more complex activities to be described. Some of its major innovations include

- Command line editing of enormous statements with arrow keys and filename and command completion.
- User-definable function keys.
- Fully recursive grammar. Statements can be arbitrarily nested or piped without concern for statement length or other arbitrary restrictions.
- Procedures and aliases. The vocabulary of the language is meant to be extensible by the user.
- Variables, arrays and expressions. Integer and floating point arithmetic, pattern matching facilities and various file system tests and editing operators provide an expressive grammar.
- Threads and processes. Child threads and processes can be spawned to run commands asynchronously or in the background.
- Command substitution. The output of one command can be stuffed back on the command line as arguments to another.
- History. Past commands can be recalled and edited.
- Advanced filename wildcarding.

This product complies fully with industry-accepted definitions for the C shell language. The user is not asked to learn yet another new proprietary language not available anywhere else. Instead, a tested, proven framework has been adapted with modern compiler technology for OS/2 and NT:

1. A modern top-down parser is used for better language recognition and performance.
2. It’s easier to use. The syntax and grammar has been made flexible and more consistent with other modern high level language conventions.
3. It knows about OS/2 AND NT: HPFS, long filenames, environmental variables, networks, international character sets, how to start PM applications and (under OS/2 2.x) about 32-bit and Virtual DOS machine (VDM) applications.
4. Threads are used extensively to achieve performance and functionality not possible in UNIX.

5. Feedback to the user, especially when reporting errors has been improved.

Who is it Designed For?

Most users of Hamilton C shell are relatively technically oriented computer users. Often, they're software developers. They have a business need for an OS/2 or an NT system.

Peering over their shoulders, they typically have lots of windows open on the screen. Many of the windows are running copies of this shell. Some copies are transient, created to display with little snippets of information needed on the spur of the moment. Other copies of the shell would be used for more long-running projects: for example, getting a **make** working for a major application.

A shell window is like any other application window but with a different paradigm. Instead of data, rows and columns of numbers or lines of text, the object being manipulated is the machine itself.

A good shell tackles a different problem than icons and windows. Instead of the point-and-shoot immediacy of ''do this single thing ***now***,'' a shell offers language and the ability to describe more customized or repetitive actions, e.g., identify a suitable set of files, perform some action against them and filter the results in some interesting way.

Installation Guide

This section outlines how to install the Hamilton C shell on your system. If you are installing the OS/2 version, follow the instructions beginning on this page. To install the Windows NT version of Hamilton C shell, please turn to page 11.

If you encounter problems, consult the ''Common Problems'' section on page 13 or call us for technical support as described on page 21.

Installation on OS/2

The first few steps, copying files from diskette to your hard disk and modifying your `config.sys`, are the same on all releases of OS/2. The remaining steps -- those associated with actually installing Hamilton C shell on your OS/2 desktop -- depend on which release of OS/2 you're running. We suggest a ''vanilla'' installation initially, but later you may want to customize it to your own tastes. For help with that, read the chapter on ''Customizing the Shell,'' beginnning on page 111.

Once you've gained familiarity with both OS/2 and with the C shell, you may want to set up the C shell as the default command processor for OS/2, completely replacing `cmd.exe` as described on page 8. The advantage to be gained (except under the 6.167 Beta and LA builds of 2.0) is that the C shell will then be able to change its own title bar and icon when you run an external command.

System Requirements

Installation requires a 286-, 386- or 486-based ATÒ or PS/2Ò or compatible, running OS/2Ò 1.1 (Presentation Manager) or Microsoft SDK 1.06 or later. Roughly 1.2 MB of disk space is used.

Hamilton C shell and the utilities supplied with it fully support HPFS and long filenames when running under OS/2 1.2 or later. They will work properly in a Presentation Manager text window or full-screen and with networks such as LAN Manager or IBM LAN Server. If you're using OS/2 2.x, it knows how to run 32-bit applications and start up Multiple Virtual DOS machines and run Win3.x applications seamlessly on the desktop. The product is not copy protected.

Basic Installation, Part I (All releases of OS/2)

1. Copy the executables in the `bin` directory into any desired directory on your search `PATH`, so long as it appears ahead of the directory containing the standard

IBM/Microsoft `more.com`. (We supply a dramatically improved `more.exe`, which should take precedence.) If you're creating a new directory, remember to add it to your search `PATH` in `config.sys` and in the `login.csh` file you create next.

2. Edit `config.sys`, adding statements to define whatever directory you choose to designate as your `HOME` directory and to ensure you're configured for a sufficient number of threads. The significance of a home directory is principally that it will be convenient to specify pathnames relative to that directory. The default number of threads is too small if you expect to have lots of windows open. Also, be sure your `PATH` explicitly lists ''`.`'', the current directory.

   You may also want to include definitions for `TABS` and `COLORS`. `more.exe` and some of the other utilities look for `TABS` to see if you want them to display text with tabs expanded out to something other than the default of every 8 characters.

   By default, the C shell displays white characters on a black background. The `COLORS` variable lets you choose something different from this set: `black`, `red`, `green`, `yellow`, `blue`, `magenta` (or `blue red`), `cyan` (or `blue green`) and `white`. Foreground colors may also be `bright`, `dim`, `blink` or `reverse`. The keyword ''`on`'' introduces background colors. (`Blink` only causes true blinking full-screen; in a text window, it just makes the background brighter. Also, `yellow` is a true yellow only if it's `bright`. These are OS/2 limitations not related to the C shell.) For more information on setting screen colors, please refer to the customization chapter or to the `colors.csh` file in the `samples` directory.

   Here's an example of what you might add to `config.sys`:

   ```
   THREADS=255
   SET HOME=D:\DOUG
   SET TABS=3
   SET COLORS=WHITE ON BLUE
   ```

   (Please be sure your `config.sys` file contains only upper-case alphabetics, no lower-case, if you're using OS/2 1.1. Lower-case alphabetics were known to cause random OS/2 system failures in that release of OS/2. This was a known bug in the OS/2 kernel and was not application dependent.)

3. Copy the `login.csh` and `startup.csh` files into ''home'' directory.

   Unless you're convinced that you've set all your environmental variables in your `config.sys` (and that your `PATH` explicitly lists ''`.`'', the current directory), use the `dumpenv` utility to paste a series of `setenv` statements into the `login.csh` file to recreate the environment you've been using with `cmd.exe`:

   ```
   dumpenv >>login.csh
   ```

   (To see what `dumpenv` does, look at the source code in the `samples` directory or simply run `dumpenv` without redirecting the output.)

The `login.csh` and `startup.csh` files can be edited with any ascii editor to customize the shell to your needs. The `login.csh` file has a lot of comments in it which can take the shell a second or more to read; you'll almost certainly want to delete some of them once you've read them so the shell will start up faster. Also, any `setenv` statements that just duplicate what's in your `config.sys` can be discarded.

The remaining steps depend on which release of OS/2 you're running.

Basic Installation, Part II (OS/2 1.1)

4. Add `csh.exe` with the title "`Hamilton C shell`" to the "Start Programs" menu. To do this, pull-down "`Program`" and select "`Add...`" from the menu bar. Fill in:

```
Program title....       Hamilton C shell
Path and file name      ....as appropriate....\csh.exe
Parameters....          -L
```

   The "`-L`" part tells `csh.exe` when it starts up that it's a "login" shell, which means it should look for a `login.csh` file. (Refer to page 183 for additional information on other options.)

5. You will likely want to create a second entry for running full-screen. It's more convenient if you're mostly working with applications that only run full-screen or if you want faster text display, especially scrolling. To do that, from the "Start Programs" menu, pull-down "`Program`" and select "`Copy...`" from the menu bar. In the Copy Programs popup, fill in the following and push the "`Copy`" button:

```
Change Title to:        Hamilton C shell -- Full Screen
```

   Back in the "Start Programs" window, select the new full screen title, pull-down "`Program`" and select "`Change...`". In the Change Program Information popup, push the "`Change`" button. This brings up the How to Run the Program popup; select "`Run the program full-screen`" and "`Enter`".

6. All the material in the `samples` directory is provided for its tutorial value; you may or may not wish to copy it onto your hard disk.

7. Reboot your system before starting Hamilton C shell for the first time. This causes the new statements in `config.sys` to take effect.

Basic Installation, Part II (OS/2 1.2 or 1.3)

4. Add `csh.exe` with the title “`Hamilton C shell`” to the “Group - Main” menu. To do this, pull-down “`Program`” and select “`New...`” from the menu bar. Fill in:

```
Program title:          Hamilton C shell
Path and file name:     ....as appropriate....\csh.exe
Parameters:             -L
```

The “`-L`” part tells `csh.exe` when it starts up that it’s a “login” shell, which means it should look for a `login.csh` file. (Refer to page 183 for additional information on other options.)

5. You will likely want to create a second entry for running full-screen. It’s more convenient if you’re mostly working with applications that only run full-screen or if you want faster text display, especially scrolling. To do that, from the “Group - Main” menu, pull-down “`Program`” and select “`Copy...`” from the menu bar. In the Copy Programs popup, fill in the following and push the “`Copy`” button:

```
Change Title to:        Hamilton C shell -- Full Screen
```

Back in the “Group - Main” window, select the new full screen title, pull-down “`Program`” and select “`Properties...`” . In the Properties popup, push the down arrow next to the “`Program Type:`” box and select “`OS/2 Full Screen`” on the list that will appear and then push the “`Change`” button.

6. All the material in the `samples` directory is provided for its tutorial value; you may or may not wish to copy it onto your hard disk.

7. Reboot your system before starting Hamilton C shell for the first time. This causes the new statements in `config.sys` to take effect.

Basic Installation, Part II (OS/2 2.x)

4. Open the Templates folder and drag a program object to the desktop (or another folder) by pressing and holding the right mouse button as you drag. On the Program page of the ''Program - Settings'' window that will appear, fill in:

   ```
   Path and file name:     ....as appropriate....\csh.exe
   Parameters:             -L
   ```

   The ''`-L`'' part tells `csh.exe` when it starts up that it's a ''login'' shell, which means it should look for a `login.csh` file. (Refer to page 183 for additional information on other options.)

5. On the Window page of the Settings, you will probably want to set

   ```
   Minimized button behavior:  Minimize window to desktop

   Object open behavior:  Create new window
   ```

   Doing this will let you conveniently open up lots of copies of the C shell as needed.

6. On the General page of the Settings, set

   ```
   Title:                  Hamilton C shell
   ```

   Close the Settings by pressing `Alt-F4`.

7. You will likely want to create a second entry for running full-screen. It's more convenient if you're mostly working with applications that only run full-screen or if you want faster text display, especially scrolling. To do that, copy the C shell icon you just created by right-clicking on it and selecting ''`Copy...`'' and then choosing an appropriate destination folder (probably the desktop) for the copy. You can also copy the icon by pressing and holding the Ctrl key while dragging with the right mouse button.

8. Once you've made the copy, right-click on it and select ''`Open`'' and then ''`Settings`''. On the ''`Session`'' page, select ''`OS/2 full screen`''. Then go to the ''`General`'' page and type a new title:

   ```
   Title:                  Hamilton C shell
                           Full Screen
   ```

   Close the Settings window for the copy by pressing `Alt-F4`.

9. All the material in the `samples` directory is provided for its tutorial value; you may or may not wish to copy it onto your hard disk.

10. Reboot your system before starting Hamilton C shell for the first time. This causes the new statements in `config.sys` to take effect.

Installation

Installation as the Default Command Processor

The C shell can also be installed as the default command processor OS/2 protected mode, meaning you specify it, not `cmd.exe` in your `config.sys`. The principal advantage is that when the when the C shell is run as the default command processor, PM allows the C shell to change its own title bar and, under OS/2 1.3 or 2.x (but not the 6.167 Beta or LA builds), its own icon to show what it's running. This can be quite helpful if you have lots of copies of the shell running minimized and would like to know what each one is doing.

The disadvantage is that the installation is slightly messy and it does disable `cmd.exe`'s ability to change its title bar and icon. For these reasons, most users will want to wait until they've gained some familiarity with the C shell and with OS/2 before installing it this way.

To install the C shell as the default command processor, follow the instructions for the basic installation but then make these changes, as appropriate for your system:

Default Command Processor Installation Procedure (OS/2 1.2 or 1.3)

1. Edit the `PROTSHELL` line in your `config.sys`, replacing the pathname and any parameters for `cmd.exe` (remembering what they were) with the pathname for the C shell and a `-L` (login) parameter. The resulting line should look something like this:

```
PROTSHELL=C:\OS2\PMSHELL.EXE C:\OS2\OS2.INI C:\OS2\OS2SYS.INI C:\OS2\BIN\CSH.EXE -L
```

2. Change the pathname you specify for the C shell in Start Programs or Group-Main to `*` (an asterisk). Also, change the parameters line to be either blank (1.1 or 1.2) or (1.3):

```
/K "%*"
```

3. Change the entries (probably named "OS/2 Window" or "OS/2 Full Screen") in Group-Main or Start Programs for `cmd.exe` to fill in the complete pathname for `cmd.exe` instead of an asterisk. Set the parameters to whatever you had specified following the pathname for `cmd.exe` (if anything) in your `config.sys` before changing it in step 1.

4. Change any entries in any of your program groups which invoke `.cmd` scripts to run them via `cmd.exe` explicitly. For example, if you had an entry that specified the program "`c:\myapp\foo.cmd`", change that to:

```
Path and file name:  c:\os2\cmd.exe
Parameters: /C c:\myapp\foo.cmd ...any additional parameters...
```

5. Reboot.

Default Command Processor Installation Procedure (OS/2 2.x)

1. Edit your `config.sys` to set `OS2_SHELL` to point to the C shell, specifying the `-L` (login) option, e.g.,

   ```
   set  OS2_SHELL=c:\hamilton\bin\csh.exe -L
   ```

2. Modify the Settings for the OS/2 Window and OS/2 Full Screen icons to show the full path for `cmd.exe` (e.g., ″`c:\os2\cmd.exe`″) rather than an asterisk on the Program page.

3. Modify the Settings for the Hamilton C shell icons to specify an asterisk pathname (meaning the default shell), deleting any mention of any startup parameters and explicitly specifying the C shell icon rather than the default icon:

   a. Right-click on the icon and open the Settings.

   b. On the Program page, set

      ```
      Path and file name:     *
      Parameters:
      ```

   c. Select ″`Find...`″ next to the icon display.

   d. Select ″`Locate`″ on the Find screen.

   e. Select the ″`Path`″ page on the Locate Folder screen.

   f. Type the pathname of the directory containing the C shell's `csh.ico` icon file. (E.g., ″`c:\hamilton\bin`″.)

   g. Press the ″`OK`″ button on the Locate Folder screen.

   h. Type ″`csh.ico`″ in the Name field on the Find screen.

   i. Press the ″`Find`″ button.

   j. The Find Results screen should appear with the C shell icon highlighted. Press the ″`OK`″ button.

   k. Back in the General Settings screen, you should now see the C shell's icon. Press `Alt-F4` to close the screen.

4. When you reboot, the C shell will be the default shell and it will appear with its correct icon both for starting and when you minimize it.

Installation

Installation on Windows NT

This section describes how to install the Windows NT version of Hamilton C shell.

System Requirements:

Installation requires a 386-, 486- or Pentium-based machine for the Intel x86 version, a MIPS R4000- or R4400-based machine for the MIPS version or a DEC Alpha AXP-based machine for the Alpha version of Hamilton C shell. The machine must be running the final release of Windows NT, build 511 (on Intel or MIPS) or 528 (Alpha) or later. Roughly 2.1 MB of disk space is used on an Intel machine, 3.5MB on a MIPS or 4.3MB on an Alpha.

Basic Installation:

1. Copy the contents of the `bin` and `samples` directories onto your hard disk, putting them anywhere you like. (Notice that the `bin` directory is too big to fit on one diskette; you'll have to merge the two or more diskettes, depending on which system you have.)

2. Copy the `login.csh` and `startup.csh` files into any directory you care to designate as your ''home'' directory. The significance of a home directory is principally that it will be convenient to specify pathnames relative to this directory.

3. Edit the `login.csh` and `startup.csh` files, customizing them to meet your needs. The `login.csh` file has a lot of comments in it which can take the shell a second or more to read each time it starts up; you'll almost certainly want to delete some of these comments once you've read them so the shell will start up faster.

4. Edit the environment variables by opening the Control Panel and then, within that, opening the system icon.

   To define a variable through the Control Panel, type the variable name in the ''Variable:'' fill-in box, the value in the ''Value:'' box and click on the ''Set'' button.

   a. Create or edit your entry for the `PATH` variable, adding the full pathnames for the C shell's `bin` and `samples` directories to the list.

   b. Create an entry for the `HOME` environment variable, setting its value as the full pathname of the directory where you placed `login.csh` and `startup.csh`.

c. You may also want to include definitions for `TABS` and `COLORS`. The shell and all the utilities look for `TABS` to see if you want them to display text with tabs expanded out to something other than the default of every 8 characters.

By default, the C shell displays white characters on a black background. The `COLORS` variable lets you choose a combination from this set: `black`, `red`, `green`, `yellow`, `blue`, `magenta` (or `blue red`), `cyan` (or `blue green`) and `white`. Foreground collows may also be `bright`, `dim`, `blink` or `reverse`. The keyword ''`on`'' introduces background colors. (`Blink` only causes true blinking full-screen; in a text window, it just makes the background brighter. Also, `yellow` is a true yellow only if it's `bright`. These are system limitations not related to the C shell.)

Other color settings you might want to specify now or at some later time through the Control Panel are `MOREPROMPT`, `MOREFILLIN` and `MOREERROR` (for customizing the `more` utility's command line) and `DELETIONS` and `ADDITIONS` (for customizing the `diff` utility).

For more information on setting screen colors, please refer to the the `colors.csh` file in the `samples` directory or to the Customization chapter.

Here's an example of the settings you might specify:

```
HOME=d:\doug
PATH=d:\hamilton\bin;d:\hamilton;samples
COLORS=white on blue
TABS=3
ADDITIONS=bright white on green
DELETIONS=bright white on red
MOREPROMPT=red on white
MOREFILLIN=black
MOREERROR=bright white on red
```

5. Add `csh.exe` with the title ''Hamilton C shell'' to the Program Manager. To do this, pull-down ''File'' and select ''New''. A pop-up will appear asking that you confirm this will be a new Program Item. On the next pop-up, fill in:

```
Description:            Hamilton C shell
Command Line:           ....as appropriate....\csh.exe -L
```

The ''`-L`'' part tells `csh.exe` when it starts up that it's a ''login'' shell, which means it should look for a `login.csh` file.

Common Problems

When I try to start the C shell in a new window, it dies and goes away before I can read its messages.

You've probably made an error on the "`Parameters`" line under OS/2 or in the "Command Line" under NT. Under NT, select the icon for Hamilton C shell and press Alt-Enter to examine the properties.

Under OS/2, you can force the window will to stay up after the shell exits so you can read the message by following the instructions appropriate for your system:

OS/2 1.1: Go to the "How to Run the Program" screen by clicking on the C shell entry in "Start Programs" and pulling down "`Program`" then selecting "`Change...`". Click on the check box beside "`Close the window...`" and press `Enter`.

OS/2 1.2 or 1.3: Click on the C shell entry in "Group - Main", pulling down "`Program`" and selecting "`Properties`". Push the "`Options...`" button and click on the check box next to "`Close window on exit`", removing the `X`.

OS/2 2.x: Right-click on the icon and select "`Open`" followed by "`Settings`." On the Session page, click on the check box next to "`Close window on exit`", removing the check.

The shell doesn't know how to run an external command.

One of the environmental variables, particularly `HOME`, `PATH` or `COMSPEC` is probably set incorrectly. Typical symptoms are that the shell doesn't seem to know how to find an external command or that it doesn't know how to run a `.cmd` file, etc. Another variation might be that it runs the old IBM `more.com` rather than the new `more.exe`. If you experience symptoms like these, first check that these variables are set sensibly.

The other common possibility under OS/2 1.x is that you're using a network and have execute, but not read access to the application you're trying to run. Due to a bug in the OS/2 1.x kernel, the C shell cannot use the kernel's DosQAppType function to determine whether the application should be started full-screen, in a text window or as a PM graphics application. Instead, the C shell is forced to read the application's `.exe` header itself; if it can't read it, it can't run it. The solution is to be sure you have read access.

Common Problems

The shell won't run my new program.

Path hashing can sometimes produce surprising results if you create a newer version in your current directory of a command that already exists in another of your path directories. The shell won't know you've done this; its hash will still only list the older version. To overcome this problem, use either the `rehash` or `unhash` commands.

The shell won't execute commands in the current directory.

Your should add the current directory to the list of directories in the `PATH` variable. `cmd.exe` always checks the current directory before looking in any of the `PATH` directories. Hamilton C shell does not make this assumption; if you want the current directory to be first one checked, you should specify it explicitly as "`.`" at the beginning of the list. For example:

```
setenv PATH = '.;c:\os2;c:\os2\bin'
```

The shell keeps running the old version my shell procedure.

If you define a shell procedure with `proc` in a `.csh` script file and then execute the script, the procedure is compiled into an internal form that's much faster to run and it's kept around in memory to make it run the next time even faster. If you change the text in the script file but don't explicitly throw away the old definition using `unproc`, the C shell won't know it's supposed to recompile.

The shell won't run any cmd.exe internal commands.

Most probably, the shell is unable to find your `startup.csh` file when it starts up. This is the file that should hold the aliases the shell uses to intercept `cmd.exe`'s built-in commands. Check to see that your `HOME` variable is set to the directory where you've placed `startup.csh` and that your `startup.csh` file isn't garbled.

When I start an application from the C shell, it dies immediately.

Under OS/2, if you find that an application dies immediately after starting, check that the `.exe` file is properly marked with its type, i.e., full-screen, PM text windowable or PM graphics. The shell tries to start up an application in accordance with the way it's marked; if it's marked wrong, the application just won't run. Even very recently, a number of PM applications including even `e.exe`, the System Editor, were being shipped unmarked, which by convention is supposed to mean full-screen. To look at or change how an application is marked, use the `markexe.exe` utility. (Type "`markexe -h`" for help.)

Another possibility is that the application has a bug that makes it fail if the maximum file handle count it inherits from its parent process is greater than 20. This problem has been seen in some past releases of the Microsoft linker (discussed below) and of WordPerfect, for example. You can force the C shell *not* to bump the file limit when it starts up using the `-Z` option but this option only works from the Start Programs (1.1) or Group (1.2) menus, not from the command line. (A process always inherits its initial maximum file handle count from its parent; from there, a process can only raise its own limit, never lower it.)

The Microsoft OS/2 linker fails under the C shell even though it works fine under cmd.exe.

Microsoft has determined there was a bug in the version of the C library used to build the `link.exe` distributed with MS C 5.1. The linker can fail if it's run as a child of a process that has a maximum file handle count greater than 20; this is a problem because the C shell sets its maximum to 255. If you're encountering this problem, try patching your `link.exe` file with the `patchlnk.exe` utility. (Type "`patchlnk -h`" for help.)

When I try to run Microsoft's make.exe in the background it hangs.

This is a known problem under OS/2 with make and certain other applications that need to spawn child processes of their own. The OS/2 process initialization and completion logic requests a semaphore in `KBDCALLS.DLL` that's already owned by whatever process in that window is already sleeping in a `KbdCharIn` call. Until another keystroke is pressed, that semaphore is never released and the background processes are never allowed to cleanly exit. This problem has been fixed in OS/2 2.x and through CSD 5050 for OS/2 1.3 with a new `KBDCALLS.DLL`. That DLL for 1.3 is available on request from Hamilton Laboratories and can be downloaded from the listings area in the "hamilton" conference on BIX.

copy or rename *.* doesn't work right.

`copy`, `xcopy`, `rename` and `del` like to do their own wildcard expansion. To make them work sensibly, be sure your `startup.csh` file includes and that you use the aliases and procedure definitions we supply to intercept these commands to turn off shell wildcarding just long enough to run them. These definitions can also serve as a model if you discover other applications that must do their own wildcarding. For more information, refer to the discussion on page 65.

The -! option doesn't work.

The exclamation point is a special character for the shell. The shell lets you pick up text out of previous commands using history references that begin with "`!`" followed by a string that tells what text you're retrieving. To avoid having an exclamation confused as a history reference, be sure the exclamation is at the end of a word, so the next character is a space or a tab.

Common Problems

grep '^foo' doesn't work.

The circumflex has special meaning as the escape character to the C shell, even inside quotes. If you want to pass a literal ''`^`'' to `grep` (or anything else) from the command line, you must type ''`^^`'' unless the immediately preceding character was ''`[`''.

When I list a directory over the network, not everything shows up.

This is a known bug in the OS/2 networking code, not the C shell. The problem occurs if (1) the directory is read over a network, (2) directory entries are being read in blocks (for higher performance) rather than one-at-a-time and (3) the total number of characters in all the filenames in that directory happens to be just right. In all cases observed, adding or deleting any arbitrary entry in the directory makes the problem go away. The bug affects the C shell and its utilities because they use blocked reads; simpler programs like `cmd.exe`'s `DIR` are unaffected because they read one entry at a time.

The bug appears to have been introduced in IBM OS/2 EE CSD WR04098 and Microsoft Lan Manager 2.0, both issued around year-end, 1990. IBM has verified the problem and has developed a fix, which is now shipping as part of OS/2 EE 1.3. If you encounter the problem and you're an IBM customer, you should call 1-800-237-5511 or contact your local IBM representative and ask for a copy of the new `netwksta.sys` file being distributed as APAR IC02287. You can also download this file from the listings area of the ''hamilton'' vendor support conference on Bix or contact us directly and we'll mail you a copy.

In the meantime, this release contains a work-around for disabling the block read feature. If you create an environmental variable, `NETWORKBUG`, and set it equal to 1, directory reads will be done only one-at-a-time, ensuring correct results at all times, albeit with some degradation in performance. You can do this either from the C shell:

```
setenv NETWORKBUG = 1
```

or in your `config.sys`:

```
SET NETWORKBUG=1
```

du, pwd and vol waste time sitting and spinning when they hit a removable drive that's empty.

If you have a removable media device other than `A:` or `B:`, these utilities will normally try to report them. That's probably not you want, at least not usually; you can specify just the set of drives you do want reported using the `DRIVEMASK` environmental variable.

cd /foo doesn't work.

Hamilton C shell tries to serve users coming both UNIX and MS-DOS backgrounds. To do this, the C shell and all the utilities accept command line options to start with either "`-`" (UNIX-style) or "`/`" (DOS-style). It also recognizes filenames typed with either forward or backward slashes. But when you type "`cd /foo`", it guesses wrong and thinks you're trying to give it a command line option that it can't recognize.

If this is really not what you intend, set the `SWITCHCHARS` environmental variable to just the specific characters you want recognized. E.g., you might include this in your `config.sys` to have only "`-`" recognized:

```
set SWITCHCHARS=-
```

I've just installed OS/2 1.2 and suddenly my environment variables don't work.

The auto install program distributed with the fall, 1989 releases of OS/2 1.2 from Microsoft and IBM has a bug. It tries to automatically convert entries on the 1.1 Start Programs menu into corresponding entries on the new 1.2 Group Main menu. If the parameters line for starting a program has text on it (as the C shell's does), the entry is garbled even though it looks correct and causes a garbled environment to be passed to the shell. Editing the entry does not fix the problem. The only solution is to delete the entry and rekey it from scratch.

I can't set my own screen colors.

Yes, you can (finally, in this latest release.) But you cannot do it just by embedding ANSI escape sequences into your prompt since the C shell will immediately reset the colors back to what it thinks they should be. To set your own preferences for screen colors, you must use the `COLORS` environmental variable. See the chapter on customizing the shell or the `colors.csh` script in the `samples` directory for more information.

The C shell's icon won't display in Group-Main.

If you install the C shell as the default command processor by specifying it on the `PROTSHELL` line in config.sys and entering its path as "`*`" in Group-Main, you will see only the default OS/2 icon in Group-Main if you select View Icon. If you start, then minimize the C shell, it will have the correct icon, however. This has been reported to IBM. Their response is that, by design, when the path is an "`*`," the Group code does not attempt to resolve the actual pathname (and whether there's any icon associated with it) until you actually click on the entry to start it. They agree this means you will not see the correct icon in the Group menu but claim this is what they intended and that it's not a bug.

Common Problems

`more` crashes on the OS/2 2.0 Beta and LA Releases.

The dynamic link library supporting 8514 displays in the beta and LA releases from IBM has a bug which causes some VIO applications, including `more`, to crash with a protection violation if they're run in a text window. They work fine full-screen. This problem has been fixed in the GA build.

`more` hangs or exits prematurely on the OS/2 2.0 6.167 and LA releases.

Under the 6.167 and LA releases, the 8514 display driver is completely unusable. It even has problems repainting the screen after a menu has been closed or displaying icons in the templates folder. It even causes `more` to hang the whole system if you have an 8514.

But even using the VGA driver, random problems will be observed due, apparently, to bugs in the keyboard driver. Depending on what's fed to it through a pipe, `more` will occasionally prematurely exit after the first screenful. All these problems have been fixed in the GA release.

The C shell can't change its title bar or icon under the OS/2 2.0 6.167 Beta and LA releases.

This functionality was disabled in the 6.167 Beta and LA releases as part of the work to add the Workplace Shell. This problem has been fixed in the GA release.

Alt-Enter doesn't work to grab commands from the history list under Windows NT and the OS/2 2.0 6.167 Beta Release.

Under Windows NT and OS/2 2.0 6.167, Alt-Enter is gobbled up by the system as a keystroke combination used to signal that an application should be toggled back and forth between the desktop and a full-screen session. Under the these systens, you'll have to type Ctrl-Shift-Enter instead.

The C shell (and lots of other applications) only have default icons under the OS/2 2.0 6.167 Beta and LA Releases.

The Workplace Shell does not support `.ico` files. All icons for text applications must be stored in the extended attributes. The latest builds of the C shell have the icon both in the EA and in an `.ico` file but if you copied the C shell onto your disk with a utility (e.g.., something other than `cp`) that does not support EA's, that information probably got lost.

To put an icon into the extended attributes, use the OS/2 1.3 File Manager, selecting the file, pulling down "`Properties`" and selecting "`Icon...`".

I just installed the C shell as the PROTSHELL and now when I start Commmunications Manager, it dies immediately.

Communications Manager is invoked via a `.cmd` script file. Follow the instructions in step 4 on page 8 to rewrite that entry to start that script explicitly via `cmd.exe`.

I can’t wildcard filenames with $, quoted or escaped characters in them.

Yes, you can (finally, in this latest release.) To do so, just quote or escape the special characters. E.g., to get all the files that begin with $, you might type `^$*` or `’$’*` .

I can’t run the C shell inside an Epsilon editor window.

The Epsilon editor tries to run whatever command processor you use by creating a full-screen session and doing a `KbdRegister` to intercept the `KbdStringIn` API entry so that Epsilon can feed it whatever you type in the editor window. Output (`stdout` and `stderr`) from the child session is redirected over a pipe back to the editor.

There are a couple problems in their approach: (1) They neglected to consider that not all applications use `KbdStringIn`; if `stdin` is attached to a keyboard, the C shell reads a keystroke at a time using `KbdCharIn` and those calls still end up tied to that full-screen session rather than being redirected. (If `stdin` is attached to anything else, it uses `DosRead` calls.) The authors of Epsilon really should have intercepted the whole set of `Kbd` calls, not just one of them. (2) Not all applications write their output to `stdout` or `stderr`; applications like `more`, that use `Vio` output, won’t run properly. Their output appears in that full-screen session, not back in the editor window. Epsilon really should be doing a `VioRegister` to grab the `Vio` output also.

We are working with Lugaru Software (the authors of Epsilon) on a solution that should be available shortly. A partial workaround is to tell Epsilon to use a separate program, which just reads input and pipes it to the C shell. Marty Klos at IBM has written a small C program to do that and placed it in the public domain. A copy is available on request from us or may be downloaded from the listings area of the “hamilton” vendor support conference on BIX.

rm doesn’t remove anything, it just puts everything in a hidden directory.

You’re using the notorious Microsoft `rm` command instead of the Hamilton `rm`. The Microsoft `rm` doesn’t remove anything; it just puts things in a hidden system directory. Hamilton `rm` is actually in `hrm.exe` under Windows NT and should be aliased to `rm` in your `startup.csh` file. Fix that and then, to get rid of all those “deleted” directories:

```
cd \; rm -x ‘ls -1ra +H | dim | grep ’deleted$’‘
```

Common Problems

Product Support

If you encounter problems or would like to make suggestions for a future revision, please contact us by any of the following or by regular mail; we promise a prompt response.

Phone:	`508-358-5715`
FAX:	`508-358-1113`
MCI Mail:	`389-0321`
Telex:	`6503890321`
BIX:	`hamilton`
CompuServe:	`70034,2025`
Internet:	`3890321@mcimail.com`

Also, on Bix, we have a vendor support conference. Do a ''join hamilton'' once you get on or follow the menus into the conference system.

Bug Reports

If you encounter what you believe to be a bug, please try to experiment to see what specific command or command sequence seems to be failing before calling. A problem that's easily reproducible is obviously easier to fix. Built in to Hamilton C shell are a number of consistency checks to trap bugs before they cause damage and to snapshot enough information to help us diagnose and repair the problem. If the shell is actually crashing, look to see if a new entry has been added to the error log, `crash.csh`, in your home directory; that information will be useful.

When you call, we'll try to provide an immediate workaround if there is one. If the problem is serious but straight-forwardly correctable, we can generally offer an interim release at no charge to fix that specific problem. At the very least, we try to schedule it for an upcoming general release.

Future Enhancements

Work continues on additional features and enhancements. As they become available, we want you to have them.

Please return the registration form by mail or FAX. Without that, we often have no way of knowing who you are to send updates to. This is particularly true if your copy was purchased through your company's purchasing department or through a retail distributor. Also, we look forward to your feedback as we strive to improve the product.

Support

User Guide

Getting Started

Starting Hamilton C shell is simple: select it from the Start Programs window or the Program Selector or type “`csh`” as a command to `cmd.exe`. After the initial greeting, you’ll see the first prompt: (The underscore is meant to be the cursor.)

```
Hamilton C shell(tm) Release 2.2
Copyright (c) 1988-1993 by Hamilton Laboratories.  All rights
reserved.
1 D% _
```

This tells you that it will remember what you type as command number 1 and that your current drive is D. The “`%`” is traditional; rather like the “`>`” for DOS. Naturally, you can change your prompt if you want, to be anything you like. For example, to get a prompt that looks like one you might get from `cmd.exe`[†]:

```
1 D% set prompt1 = '[$upper(cwd)] '
[D:\DOUG] _
```

This works by taking the value of the `cwd` (current working directory) variable, turning it to upper case using one of the built-in procedures and pasting left and right brackets around it. The value is recalculated each time a prompt is given, so it always displays an up-to-date value. (Lists of all the built-in variables and procedures are given in later sections.)

To set it back:

```
[D:\DOUG] set prompt1 = '$@ $CDISK% '
3 D% _
```

Basic Statements

Generally speaking, whatever commands you might have typed into `cmd.exe` will still work here. Even an “internal” `cmd.exe` function like `dir` works:

```
3 D% dir

 The volume label in drive D is USER.
 Directory of D:\DOUG\SH\DOCS\SCRIPT\HELLO
```

[†] We introduce this is as the first example with some trepidation: the prompt seems to be the first thing people want to change. But it can also be one of the more daunting projects if you’re getting started. This example is offered more in the spirit of assurance that, with a little experience, the prompt can be set to anything you like.

```
.            <DIR>      2-23-89   2:13p
..           <DIR>      2-23-89   2:13p
HELLO    C          72   2-23-89  12:56p
HELLO    EXE      7731   2-23-89  12:57p
MEMOS        <DIR>      2-23-89   1:46p
     5 File(s)   1581056 bytes free
4 D% _
```

If the command you type refers to a `.cmd` batch file or a `cmd.exe` internal function, Hamilton C shell passes it to a child process running `cmd.exe` for evaluation. (`cmd.exe`'s built-in functions are intercepted with aliases defined in your `startup.csh` file.) Everything else is evaluated directly by Hamilton C shell. For example, if you type the name of an `.exe` file, the appropriate `DosExecPgm( )` or `DosStartSession( )` call to the OS/2 kernel or `CreateProcess( )` call to the NT kernel to start that program will be done directly by Hamilton C shell.

A bit-mapped hash mechanism is used so that when you type the name of a command, the shell zeroes right in on file you mean. It doesn't have to check every path directory for every possible extension. Naturally, if you type a command that doesn't exist, the shell complains:

```
4 D% zork
csh:  Couldn't find an executable file named 'zork'.
```

By being more than merely a ''wrapper'' around an existing command processor, several advantages are created: (1) performance is understandably (and visibly!) much higher and (2) limitations on command line lengths, etc., become the relatively generous limits of OS/2 and NT, rather than the restrictive limits of `cmd.exe`.

Customizing the Screen Colors

The C shell's default screen colors are white characters on a black background. Highlighting and color are used to make some things (special files, etc.) stand out. All the use of color or highlighting is completely customizable. You can choose anything you like. The chapter on customization will go into this in detail, but for now, let's suppose we'd simply like to pick something a little easier on the eyes, like white characters on a blue background:

```
5 D% setenv COLORS = white on blue
```

Command Line Editing

With command line editing, you'll notice immediately how much easier it is do things quickly without a lot of retyping. As you try the examples, notice how the arrow, insert, delete, home, end and other keys can be used to recall previous commands or make changes anywhere on the line.

Command line editing is like having a full-screen editor pasted onto the front end of the shell. Key bindings are intuitive and follow accepted conventions. You can create enormous commands that stretch over screen after screen and move around with the arrow keys, inserting or deleting anywhere. Watch changes ripple almost instantly down even an entire screenful of text. We think you'll find our command line editing superior to *anything* you've seen or used elsewhere.

The basic key assignments are:

<Insert>	Toggle between insert and overstrike modes. (The cursor is thicker when you're inserting.)
<Home> <End>	Beginning/end of command line.
‹ fi	One character left/right.
› fl	Up/down one command in the history list.

Pressing Ctrl with the arrow keys lets you move by words or lines. Pressing Alt instead does word or line deletion. (The convention we follow is that the Alt key is a little "more powerful" than the Ctrl key.)

What you last deleted is kept in a scrap buffer and can be pasted back elsewhere. To paste something from the scrap buffer back into the command line, move the cursor to where you want it done and press:

Ctrl-<Insert>	Paste one word at a time.
Alt-<Insert>	Paste the whole thing.

Command Completion

In addition to backing up through your previous commands one at a time with › and fl, you can also ask the shell to search back through any previous commands you've typed for the last command that either started with or contained the characters in the previous word.

Ctrl-<Enter>	means "look for a command that *started* with ...," and
Alt-<Enter>	(again, a little "stronger") means "look for a command that contained the string *anywhere*." (On NT, it's necessary to type Ctrl-Shift-<Enter> because Alt-<Enter> is grabbed by the system to mean switch to full-screen.)

Repeatedly pressing these keys cycles up through all the matching commands you've previously typed. Command completion uses something called the history mechanism to recall commands you've previously typed. Later, we'll devote a whole chapter to some of the more advanced uses of history.

Filename Completion

Filename completion is another ''creature comfort:'' you type just a fragment of a filename and let the shell fill in the rest. The two variations are using the F key for basic filename completion or the D key if you want all the duplicates listed.

Alt-F or Ctrl-F	Filename completion. Look for a filename that starts with preceding characters. If it matches a single file, fill in the rest of the name. If more than one file matched, show the part that was the same for all, highlighted in green. (Bright red means there were no matches at all.)
Alt-D or Ctrl-D	Duplicate completions. Show any/all matching filenames, one after the other with spaces between.

Filename completion is actually done with wildcarding. Unlike `cmd.exe`, Hamilton C shell does any wildcarding before excuting the command you type. It uses a powerful recursive pattern match algorithm that guarantees sensible matches even if you type a very complex pattern. Wildcarding is the subject of a whole chapter up ahead.

The Tour Begins Here

The following chapters introduce the various facilities Hamilton C shell provides, starting with some of its basic vocabulary: the simple utilities that come with it.

Following discussion shifts to the intrinsic, internal functions provided by the shell itself: i/o redirection, pipes and command substitution; the history mechanism and wildcarding.

Intermediate level discussion follows, describing expressions, variables and aliases and the editing and quoting facilities. The process and thread scheduling mechanism is described, outlining how an activity can be placed in the background.

The tour will then cross the threshold from discussion of individual statements to discussion of structures of statements. Structures for iteration and condition-testing and procedural abstraction will be introduced.

Finally, we'll wrap up with discussion of how to customize the shell together with a section detailing specific compatibility issues between the Hamilton and original Berkeley C shells.

The Utilities

Hamilton C shell comes with a lot of utilities that form some of its vocabulary. They do small, but oft-needed functions, often in a novel, faster or more convenient way than you'd find in ''plain vanilla'' OS/2 or NT. This section provides a quick tour, outlining some of the capabilities and conventions.

ls: List files

`ls` is a somewhat nicer way to list a directory:

```
6 D% ls
memos      hello.c    hello.exe  sysstuff
```

Subdirectories are highlighted (shown here in **`bold`**.) If a file or directory has the system bit set, it's still listed, displayed in green (shown here in *`italic`*.)[†] Normally, `ls` lists everything in lower case for better readability. In long format:

```
7 D% ls -l
D----  Feb 23  13:46          -  memos
---A-  Feb 23  12:56         72  hello.c
---A-  Feb 23  12:57       7731  hello.exe
-S-A-  Feb 23  13:22         15  sysstuff
```

Conventionally, `ls` lists things alphabetically, with directories ahead of files. There might be hidden files or directories, but to see them you have to ask:

```
8 D% ls +H
memos      hello.c    hello.exe  hiding     sysstuff
```

Conventions

To find out how any of the utilities work, just use the `-h` option. For example,

```
9 D% ls -h
```

tells about options for more detailed listings, sorting the list by date or by size, selecting only certain types of files, etc. `ls` is a read-only activity; it ***never*** makes any changes to the file system. Lists are always sorted in memory; its speed and flexibility completely obsolete the old (and dangerous) ''directory sort'' utilities popular on DOS.

[†] All our examples will be given in terms of the default screen colors. But these are easily changed to your own preferences. See the chapter on customization or the `colors.csh` script file in the `samples` directory.

Utilities

The names of the utilities were chosen to be consistent with the names of similar functions on UNIX, where they provided much of the vocabularly of the original UNIX C shell. But changing the name of a utility is a simple matter: just rename the corresponding `.exe` file or, better still, create an alias (discussed later.)

By convention, the utilities expect options to come ahead of any files you specify. Options *are* case-sensitive. We've tried to use mneumonic letters for options (e.g., `h` for help) and to use the same letter to mean the same thing across related utilities; achieving that is simply more feasible with 52, not just 26 characters to choose from.

Our examples generally show options introduced with "`-`", but you could equally well follow the DOS-style convention of using "/" if you prefer. If indeed you want only "`-`" or only "/" interpreted as an option character, this can be set with the `SWITCHCHARS` environmental variable, which can be set either from the C shell or from your `config.sys` file on OS/2 or from the Control Panel on NT. Sadly, it won't have any effect on the standard OS/2 or NT commands like `dir` or `xcopy` or on applications you purchase elsewhere, but it will work on all the commands supplied with the C shell. For example, to have only "`-`" recognized as an option character, you might type this into the C shell:

```
10 D% setenv SWITCHCHARS = -
```

or put this into `config.sys` (rebooting to make it take effect):

```
set SWITCHCHARS=-
```

You can type options in any order (except where one overrides another, in which case the last setting is used) and you group them together or type them separately as you choose. For example, "`ls -L -d -w`" is exactly the same as "`ls -dwL`" and produces a very long format (very detailed) list of the current directory, sorted by date (newest ones last), with sizes of any directories filled in by walking down through the directory tree, adding up all the sizes of all the files found there.

You can always unambiguously end the options with "`--`" in case you have a filename or an argument string that begins with one of option-introducing characters. Also, since the shell does the wildcard expansion, it's a bit more convenient and faster for the utilities to look for any options right at the beginning of what could be a very long list (up to 64 kilobytes) of filenames or other command-line text.

We'll always follow the OS/2 and NT convention of using "\" in filenames in this book and we generally advise that you do too, not so much because the C shell cares but because so much other OS/2 and NT software does. To some fair degree, it's a case of "when in Rome, doing as the Romans do." But if you really do prefer, you can generally use "/" with the C shell and all the utilities. Do remember, however, that if you type a filename starting with "/" to mean the root, you have to be careful that it can't be confused as the start of an option. (This is a good use for the "`--`" option or the `SWITCHCHARS` variable.)

echo

`echo` is a little different than the vanilla OS/2 or NT `echo`. It does only one thing: it prints whatever arguments words you give it; there's no `echo on` or `echo off`-style status reporting function. But it does offer much finer control over what gets printed: you can write binary values, choose not to append a new line and write to stderr instead stdout.

Here's an example where the ANSI escape sequences turning brightness on and off are embedded into a string being echoed. The ANSI escape character is octal 033; binary values or special characters like "`[`" are introduced by the "`^`" shell escape.

```
11 D% echo Have a ^033^[1mnice^033^[0m day.
Have a nice day.
```

(Processing of the `^` escape sequences is done by the shell before any command ever sees it. As a result, you can use escape sequences to construct command line arguments for any command; this feature is introduced here only because it tends to be most often used with `echo`.)

mv, cp and rm: Move, copy and remove

The `mv` (move), `cp` (copy) and `rm` (remove) trio allows files and directories to be treated as simple objects.

`mv` will move either files or directories treating them simply as objects, even across disk partitions. In this example, the two `hello` files are moved into a new directory, illustrating how `mv` understands that if there's a many-to-one relationship, the destination has to be a directory.

```
12 D% mv hello* hello
13 D% ls
hello         memos         sysstuff
14 D% ls hello
hello.c       hello.exe
```

Similarly, `cp` will copy a file or even an entire directory. The copies `cp` produces are always exact logical copies, with correct timestamps[†] and attribute bits and including any hidden or system files.

```
15 D% cp hello newhello
16 D% ls
hello         memos         newhello      sysstuff
17 D% ls -l hello
---A-  Feb 23  12:56         72  hello.c
---A-  Feb 23  12:57       7731  hello.exe
18 D% ls -l newhello
```

[†] Files only under OS/2 1.1. New directories always get the current timestamp unless you're running OS/2 1.2 or later.

```
---A-  Feb 23  12:56         72  hello.c
---A-  Feb 23  12:57       7731  hello.exe
```

`cp` does not consider it an error to copy over an existing file unless the file about to be overwritten has its read-only bit set.

Finally, `rm` can be used to remove a file or even an entire directory. But it does insist that you tell it you really mean it if you ask to remove a directory that's not empty or anything that's marked with the system bit.

```
19 D% rm sysstuff
rm:  Can't remove system file 'systuff' without -S option.
20 D% rm -S sysstuff
21 D% ls
hello         memos         newhello
22 D% rm newhello
rm:  Can't remove non-empty directory 'newhello' without -r
option.
23 D% rm -r newhello
24 D% ls
hello         memos
```

As you can see from these examples, the general style of the utilities is fairly terse. Like the proverbial Vermonter, they don't say anything unless they've got something to say. Even copying or removing a directory happens without fanfare as long as the appropriate ''yes, I really mean it'' options are supplied.

more

`more` is an especially fast browsing filter. There are two ways to use `more`. The first is in a pipeline, the way ''vanilla'' `more` might be used when you suspect the data may be longer than a screenful:

```
25 D% ls -l c:\os2\bin | more
:
:
```

If the output turns out to be less than a screenful, it's as though you'd just typed the `ls` command by itself. In fact, there's not even a noticeable performance penalty. But if it's more than a screenful, `more` switches to an interactive mode where you can use the arrow keys, etc., to browse up and down through the listing.

`more` can also be used for browsing a list of the files you give it on the command line:

```
26 D% more *.c
```

`more` incorporates the Berkeley notion referred to, tongue-in-cheek, as ''more is less[†]'': it's a good paging filter that lets you go forwards and backwards. It also offers a number of different ways of looking at or searching the data including binary, as control characters, line-numbered, etc. Perhaps most important, it's *fast*.

Part of `more`'s speed comes from an internal cache of about 11K characters of text coupled to an indexing structure that it builds on the fly as it reads the input. When you move forward or backward within the cache, screen redraw rates are the limiting factor in performance. Outside of range of the cache, if the input is from a disk file, the indexing structure, technically an ISAM, tells `more` how to seek to the new location.

There is also a ''huge'' version of `more`, called `moreh`, that was compiled in large model and while slightly slower, has the advantage of caching up to about 4M characters. `moreh` can be useful when speed is less important than being able to scroll all the way back through a large amount of text coming through a pipe.

touch

`touch` lets you change the timestamps of individual files or directories[‡] or, using the `-r` (recursive) option, of everything in a whole directory tree.

If the desired timestamp isn't given, `touch` uses the current time. If the filename doesn't exist, it's created as a zero-length file.

```
27 D% ls
hello         memos
28 D% touch zork
29 D% ls
hello         memos         zork
```

chmod

`chmod` lets you set a file's attributes but leaves the timestamp alone. Here is an example, first setting the system bit (making it show up in green), then making it hidden:

```
30 D% chmod +S zork
31 D% ls
hello         memos         zork
32 D% chmod +H zork
33 D% ls
hello         memos
```

[†] The story is now a part of computer folk history: at first, `more` only went forward. Then someone created a filter that went backwards, which he aptly named `less`. When later versions of Berkeley's `more` incorporated this feature, they were heralded by announcements that, finally, ''more was less.''

[‡] On an OS/2 1.1 system, the kernel allows you to change the timestamps only on files, not directories. `touch`'ing a directory does nothing unless you use the `-r` option to recursively `touch` the directory's contents.

Of course, the file is still there and you can continue to manipulate its attributes:

```
34 D% ls -l zork
-SHA-  Feb 23  13:16         0  zork
35 D% ls +a
.              hello          zork
..             memos
36 D% chmod +R zork
37 D% ls -l zork
-SHAR  Feb 23  13:16         0  zork
```

Many users will find that a file's system bit is more useful than they'd thought before. With `chmod`, it's easy to set or clear the bit and setting it doesn't make the file hidden. Quite the contrary, `ls` makes it stands out in green. Also, a file marked ''system'' is a little safer from accidental deletion or overwriting. These are often convenient characteristics to attach a few specific files within a large directory. For example, the author tends to routinely mark `make` files within a C source code directory as ''system'' just so they'll stand out.

du, vol and pwd

`du`, `vol` and `pwd` provide quick snapshots of your disk partitions: `du` tells how much of the partition is used; `vol` displays the label; and `pwd` shows the current directory on each partition.

```
38 D% du
c: 31.904 M Total = 29.465 M Used + 2.439 M ( 7.65%) Free  root
d: 23.920 M Total = 22.438 M Used + 1.482 M ( 6.20%) Free  user
e: 13.957 M Total =  8.520 M Used + 5.438 M (38.96%) Free  misc
39 D% pwd
c:\os2\include
d:\doug\sh\docs
e:\tmp
40 D% vol
c:  Jan 24  22:32:10  1988   root
d:  Nov 27  20:34:58  1988   user
e:  Jan 17  17:12:20  1988   misc
```

A common convention observed by the utilities is that if one entry on a list is more current or special than the others, it's highlighted. `du`, `vol` and `pwd` each highlight the entry describing the current disk.

For the benefit of those who have lots of partitions, some of which they don't want to bother listing all the time, `du`, `vol` and `pwd` look for a `DRIVEMASK` environmental variable which can be used to mask off just the drive you want. This is especially useful for excluding drives that take removable media; if they're empty, they can waste a lot of time trying to read a diskette that's not there.

dirs, pushd, popd and rotd

The shell provides a built-in mechanism for keeping several directories ''handy.'' This mechanism is the directory stack, which always contains a list of fully-qualified directory pathnames with the current directory at the top. You can display the list with the `dirs` command:

```
41 D% dirs
d:\doug\sh\docs
```

Initially the list contains only your current directory. When you push a new directory on the stack with `pushd`, that becomes your new current disk and current directory. `pushd` also reports the resulting stack contents.

```
42 D% pushd c:
c:\os2\include
d:\doug\sh\docs
43 C% pushd e:
e:\tmp
c:\os2\include
d:\doug\sh\docs
```

Calling `pushd` without any arguments just swaps the top two directories:

```
44 E% pushd
c:\os2\include
e:\tmp
d:\doug\sh\docs
```

Popping elements off the stack is done with `popd`, which also reports the resulting stack.

```
45 C% popd
e:\tmp
d:\doug\sh\docs
```

The stack can also be rotated with `rotd`. (We'll push another directory first so we can see that rotation is *upward*, with the top item going to the bottom of the stack.)

```
46 E% pushd \
e:\
e:\tmp
d:\doug\sh\docs
47 E% rotd
e:\tmp
d:\doug\sh\docs
e:\
```

You can pop multiple directory entries at once, but if you ask to pop more than exist, you'll get a message:

```
48 E% popd 5
csh:  The built-in popd command can only accept a integer
argument n, where n > 0 && n < number of elements on the
directory stack.  The default for n is 1.
49 E% popd
d:\doug\sh\docs
e:\
```

fgrep and grep

`fgrep` and `grep` are fast string search utilities. Their names and the regular expression syntax are traditional; it's an accepted standard and we've followed it.

`fgrep` and `grep` are used to scan through long lists of files or filter data coming through a pipe for strings or patterns you specify. They'll quickly report all the matching lines. If you like, you can get more or less detail in the output, e.g., have line numbers shown or just get a total count of all the matches.

`fgrep` and `grep` both have the ability to look for a large number of patterns in parallel (using the `-s` or `-f` options) with almost no discernable performance degredation. They're very fast. Both precompile and optimize their search patterns, use direct kernel api calls for all i/o and use a very high performance buffering structure to allow extremely fast scanning of large amounts of data.

fgrep

`fgrep` is the simpler and slightly faster of the two search utilities. It does a simple string compare between the string you're looking for and the characters on each line. If the search string is found anywhere on the line, it's a match. There are some options for ignoring differences in upper-/lower-case or in the amount of white space (spaces and tabs) between words but but mostly it's quite simple comparison.

Here's an example of using `fgrep` to search a very simple personal phone directory where each record is just a line of text and we'll search it . (Later we'll learn how to package things like this up into aliases or shell procedures so you can call them with just a few keystrokes.)

```
50 D% fgrep -i doctor \phone
Doctor James Gardner    508-999-0000  12 Canton St
Doctor Karen Strickland 508-721-1223  N.E. Medical Offices
```

grep

`grep` looks for special patterns called regular expressions, which are similar to (but slightly different from) filename wildcarding. The grammar is recursive, meaning a regular expression to be matched can be written, in turn, as a nested series of regular expressions:

c	Any ordinary character matches itself.
`\`*c*	Match the literal character *c*.
`^`	Beginning of line.
`$`	End of line.
`.`	Match any single character.
`[...]`	Match any single character in the list.
`[^...]`	Match any single character not in the list.
`\`*n*	Match whatever literal text the *n*'th tagged `\(...\)` expression matched.
r`*`	Match zero or more occurrences of *r*.
r1r2	Match expression *r1* followed by *r2*.
`\(`*r*`\)`	Tagged regular expression. Match the pattern inside the `\(...\)`, and remember the literal text that matched.

At the lowest layer, you give a character or set of characters to be matched anchored, if you want, to match just the beginning or just the end of a line. At the next layer, the “`*`” character lets you match a variable number of repetitions of a pattern.

When you type a regular expression on the command line, keep in mind: (1) Many of the characters have special meaning to the C shell and have to be inside quotes. (2) You have to type two “`^`'s” to get just one because “`^`” is the shell's literal escape character. (3) “`*`” is a *postfix* operator. It operates on the preceding regular expression; by itself, it is *not* a “match zero or more characters” wildcard character as you may be used to with filenames.

Here's an example of searching through all the source code for a large application, looking for all occurrences of lines that begin with “`statement`” followed by a “`y`” somewhere on the line and showing the line numbers of any matches. (The `-s` option tells pushd and popd to work silently.)

```
51 D% pushd -s ~\sh
52 D% grep -n '^^statement.*y' *.c
allocate.c:418:statement_obj *allocate_statement(size, type)
53 D% popd -s
```

sed

`sed` is a stream editor. Just as you might think of using a regular editor to edit a file, deleting or inserting lines, doing search/replace operations, etc., `sed` lets you edit a stream of data: individual lines are read from stdin, edited according to the script you give and written to stdout. A very simple sort of script might be given right on the command line. Here's a simple search/replace:

```
54 D% echo hello world | sed s/world/everybody/
hello everybody
```

`sed` uses the same regular expressions used by `grep`. It's possible to pick up pieces of the input as tagged expressions and move them around. In this example, the two strings on either side of the space are tagged, then swapped around. Quotes are used around the search/replace command so the C shell will treat it as one long literal string to be passed to sed. (Parentheses, spaces and asterisks otherwise have special meaning.) Notice how the ''`*`'' construct, meaning match zero or more occurrences actually matches as many repetitions as possible.

```
55 D% echo hello world | sed 's/\(.*\) \(.*\)/\2 \1/'
world hello
```

For more complex operations, `sed` offers a wide array of operators including even conditional branches and a hold buffer where a string can be saved temporarily from one line to the next. If your script is very long, the `-f` option lets you specify it in a file.

diff

`diff` is an extremely fast and flexible utility for quickly comparing ascii files, looking for differences. In the simplest form, you simply give it two filenames corresponding to the old and new versions and let it go to work, reporting sections that have been deleted or added in a traditional format. For example, as a software developer, I might use it to compare old and new versions of a C program:

```
56 D% diff archive\parse.c parse.c
1493 c 1493
<          d->inline_cnt = src->inline_cnt++;
---
>          d->inline_cnt = ++src->inline_cnt;
```

Each change is reported in terms of the line number or range in the old version, whether it's an addition, change or deletion, the line numbers in the new version and then the affected lines from each file, separated by a line of ''---''.

`diff` supports the traditional options for ignoring differences in upper-/lower-case or in the amount of white space on the line, for recursively comparing entire directory trees of files, etc.

One of `diff`'s most novel features is its ability with the `-!` option to generate a merged listing where text that's deleted is shown in red, new text is shown in green and the rest is displayed normally. This makes it extremely easy to view your changes in context. (To use this option, remember that ''!'' is a special character to the shell; type it at the end of the option list so there'll be a space following.)

head and tail

`head` and `tail` are used to display just the first or last few lines or characters of a file. Normally, they expand any tabs into spaces so you don't need to filter them through `more`.

`tail` is particularly interesting. If all you want is the end of a very large file, `tail` doesn't waste time reading the whole file from start to finish. Instead, it jumps right to the end and reads it backwards! If the file is truly large (on the order of several megabytes) and all you want is a little bit off the end, this is the difference between chugging along for several seconds versus getting an almost instantaneous response.

`tail` also has a `-f` follow option. What that means is that when it gets to the end of file, it enters an endless loop, sleeping for a second, then waking up to see if more has been added. This is particularly useful if, e.g., you have an operation, say a large `make`, active in one window with its output redirected to a file. From another window you can periodically check in on the progress by typing:

```
57 D% tail -f e:\tmp\make.log
:
^C
```

`tail` lets you watch lines get added without consuming much processor resource (since it sleeps in the kernel most of the time) so you can watch a background activity progress without affecting its performance. After you've watched for a while, just type `^C` to interrupt and get out. The interrupt only goes to the tail program; the application off in the background or in another window creating the file is not affected and will go on about its business until you come back once again to check on it.

cut

`cut` is a simple filter for selecting out just certain fields or character positions of each line of input. You choose what characters should be interpreted as the field delimiters and which fields should be copied to the output. For example, if you kept your phone book in `\phone`, you might strip off just the first word from each line to get everyone's first names:

```
58 D% cut -f1 -d' ' \phone
Ed
Helen
Jack
Vickie
:
```

The `-f` option means you want to count by fields, selecting the first field and that the delimiter is a space character. (Notice the quotes around the space.)

Utilities

split

`split` lets you break up a large file into smaller, fixed-size pieces counting either by lines or by characters. Each of the smaller files it creates are numbered, e.g., `chunk.001`, `chunk.002`, `chunk.003`, etc.

One example of where you might use split might be if you had a very large file you wanted to transmit over a modem. If the line dropped suddenly, you wouldn't want to have to start all over on a 2M file. If you split it first into 200K chunks, you'd stand to lose a lot less. Another example might be if you had a truly enormous text file that was just too big to easily edit with your favorite editor. Splitting it up into chunks of only 10K lines each might be a solution.

tabs

`tabs` lets you expand or unexpand tab characters based on a set of tab settings you give it. Tab settings are religious. I like them every 3 spaces but you probably like something else. If you're composing something to be sent as email or posted on a bulletin board, it's probably nice to expand it out before you send it so everyone sees what you see.

tr

`tr` is a another simple filter for translating characters from input to output. For example, you could translate everything from lower to upper case by typing:

```
59 D% tr a-z A-Z
hello world
HELLO WORLD
^Z
```

We typed the first `hello world` and `tr` has just echoed it in upper case. `^Z` is the end-of-file character defined by OS/2 and NT.

`tr` also has a number of options for squeezing out repeated sequences of the same character or editing out just certain characters and even for normalizing the text in a file, ensuring that every line ends with a carriage return/line feed combination. That's handy if you're importing a file from another operating system.

strings

`strings` lets you simply list out all the ASCII strings in an otherwise binary file. Sometimes this can be useful for spelunking around through a file when you're really not sure at all just what's inside it. Various options are available to trimming the output so only strings of a minimum length, etc., will be shown. For example,

```
60 D% strings hello.exe
!This program cannot be run in DOS mode.
:
Hello, world
:
```

Another example might be if you suspected an application was carrying a virus. Naturally, `strings` can't guarantee something's free of any virus, but on the other hand, if you scan it with `strings` and find something like this, obviously you should be careful:

```
61 D% strings a:suspect.exe
:
Aha!  Gotcha!  I just blew away your hard disk!
```

dskread and dskwrite

This pair of utilities can be used to quickly copy, format or mass duplicate diskettes in a single pass. Here's an example using `dskread` to read a whole diskette image onto your hard disk and then write it back out onto a new floppy with `dskwrite`. The `dskwrite -a` option means autoformat, i.e., if the new disk isn't already formatted, format each track as it's written. The `-v` option means read back and verify each write to be sure a good copy was made.

```
62 D% dskread a: >disk.image  # Read the whole diskette
63 D% dskwrite -av a: <disk.image  # Make a new copy
```

To make more copies, you simply keep putting new diskettes in and rerunning the `dskwrite`. Since the entire diskette image, including the boot sector is captured onto the hard disk by `dskread`, it's possible to email a bootable diskette image, even compressing it first with one of the popular compression utilities such as `arc` or `pkzip`.

`dskread` and `dskwrite` can also be used to copy a very large file onto a whole series of diskettes and restore it later as a single file. E.g., to back up your hard disk, you might compress whole directories using one of the popular compression utilities and write the resulting archive file out as 1.2MB (or, as appropriate, 1.4MB) chunks onto separate diskettes. To restore the file, just `cat` or `dskread` the pieces together again. (But if you do try this, be sure the compression utility you use can handle any long filenames or system or hidden files you have lurking in your directories.)

Here's an example of writing a very large `.zip` file to a series of diskettes. The `-c` option means it should continue with as many diskettes as it takes to hold all the data, prompting you for each new diskette when it needs it. The `-d` option causes it to write a dummy file system around the data (so it still looks like a regular OS/2 or NT diskette). The `-n` option takes an argument specifying the name given the file being created on the diskette. In this example, the first diskette will have a file named "`big.001`", the second will have "`big.002`", etc. Once again, the `-av` options mean autoformat and verify each write.

```
64 D% dskwrite -avcd -n big a: < g:big.zip
```

To get it all back, you could simply copy all the pieces individually onto your hard disk and then `cat` them together or use `dskread` to do it for you:

```
65 D% dskread -dc -n big a: > g:big.zip
```

Other Utilities

Other utilities provide means for sleeping for a timed period, counting the number of words in a file and so on. Part of the appeal of Hamilton C shell is that it's relatively easy to continue expanding the vocabularly with simple utilities that may each be only a few hundred lines long.

This has been a fast introduction. Fortunately, you don't have to learn the utilities just from the book. All have on-line information available with `-h`. We encourage you to experiment.

As this is being written, we're still giving thought to additional utilities. If you have favorites you'd like to see included or maybe offered as new products, please contact us.

I/O Redirection and Piping

I/O Redirection

You can redirect or pipe i/o in much the way you might under `cmd.exe`. Here's a simple example redirecting stdout from the word count of the famous ''Hello, world'' program. `cat` just copies from any files you tell it or, by default, from stdin to stdout.

```
66 E% cd hello
67 D% ls
hello.c      hello.exe
68 D% cat hello.c
#include <stdio.h>
main ()
   {
   printf("Hello, world.\n");
   }
69 D% wc hello.c >hello.wc
70 D% cat <hello.wc
        5        8       72  hello.c
```

(`wc` tells us that `hello.c` has 5 lines, containing 8 words, totalling 72 characters.)

If the file you write to with ''>'' exists, it's first truncated to zero length (discarding the old contents); if the file doesn't exist, it's created. With ''<'', it's an error if the file doesn't exist.

Data can be appended to a file with the ''>>'' operator:

```
71 D% echo that^'s all folks >>hello.wc
72 D% cat hello.wc
        5        8       72  hello.c
that's all folks
73 D% _
```

When you append with ''>>'', if the file exists, data is written onto the end; if it doesn't exist, it's created. (The single quote character has special meaning to the shell on the command line; the special meaning is turned off by the shell's escape character,''^''.)

noclobber

Not everyone is comfortable with letting the shell glibly toss away an existing file if you type ''>'' when you meant ''>>'' or lose it somewhere if you mistype an existing filename with ">>". The `noclobber` variable lets you tell the shell you want this to be caught, so you can decide if this was really what you meant.

If you set `noclobber`, you have to type "`>!`" to redirect to an existing file:

```
73 D% set noclobber = 1
74 D% echo trash this file > hello.c
csh:  Couldn't open 'hello.c' as a redirected standard output.
```

Come to think of it, let's *not* overwrite that file.

Similarly if you want to append to something that doesn't already exist:

```
75 D% echo appended data >> newdata
csh:  Couldn't open 'newdata' as a redirected standard output.
76 D% echo appended data >>! newdata
77 D% cat newdata
appended data
78 D% rm newdata
```

Protection Attributes

If a file has any of the special protection attributes, hidden, read-only or system, set, you cannot overwrite it by redirecting i/o to it. Even when you type "`!`", you still can't. Before you can redirect to it, you must clear all these attribute bits.

```
79 D% ls -l zork
-SHAR  Feb 23  13:16         0  zork
80 D% echo new zork data >! zork
csh:  Couldn't open 'zork' as a redirected standard output.
81 D% chmod -R zork
82 D% echo new zork data >! zork
csh:  Couldn't open 'zork' as a redirected standard output.
83 D% chmod -SH zork
84 D% ls -l zork
---A-  Feb 23  13:16         0  zork
85 D% echo new zork data > zork
86 D% _
```

Stdout and Stderr

Redirecting both stdout and stderr together is done by adding an ampersand. For example, using `echo`'s "`-2`" option to deliberately write to stderr and parentheses for a simple grouping:

```
86 D% (echo -2 error; echo standard) > zork
error
```

```
87 D% cat zork
standard
88 D% (echo -2 error; echo standard) >& zork
89 D% cat zork
error
standard
90 D% _
```

Separately redirecting stderr and stdout to different files is a little tricky: first you redirect them both, then redirect stdout by itself. Here's an example running the C compiler with stdout to `log` and stderr going to `errors`.

```
90 D% cl hello.c >& errors > log
```

You can type as many i/o redirections in a row as you like. The shell evaluates them one after another. If you redirect to a new file, then redirect to something else, the effect is just like `touch`'ing the file.

Pipes

Pipes are a way of connecting a series of activities together so that the output of one is read as input to the next. Each of the activities runs asynchronously and concurrently with the others. Data is passed completely in memory and is very fast.

The syntax is similar to i/o redirection in its use of the "&" character. To pipe just stdout, use "`|`" by itself:

```
91 D% ls -L | more
```

To pipe both stdout and stderr together, use "`|&`":

```
92 D% cl hello\hello.c |& more
```

The leftmost part of the pipeline is evaluated directly by the shell's current thread. The successive right parts are evaluated by child threads. (This is so that piping a command that lists status information on the current thread through a filter like `more` operates sensibly.) Each part of the pipeline can be an arbitrarily complex statement, perhaps even run as a separate OS/2 screen group or in a separate NT window.

Pipes are much faster and more responsive than with vanilla OS/2 or NT due to improved buffering and scheduling technology. A long pipeline finishes much faster. Also, when you type `^C` to interrupt, it comes back immediately without a lot of nuisance messages.

Command Substitution

A particularly novel way of piping statements together is to use the output of one as command line arguments of another. This is called command substitution and you indicate it by typing backquotes, `‘...‘`, around a command.

```
93 D% ls +a
.              hello          zork
..             memos
94 D% echo ‘ls +a‘
. hello zork .. memos
95 D% _
```

When command substitution is done, all the extra ‘‘white space’’ (space characters, tabs and newlines) is squeezed out. Also, any ANSI escape sequences that might have turned on highlighting or color, etc., are deleted. You just get the list of words the backquoted command wrote to stdout. In this example, the order of the files is a bit scrambled when the line ends are removed; the `-1` (numeric one) single column option can fix this. (Try it again using `ls +a1` inside the backquotes.)

Command substitution is especially useful anywhere you need to give a list of filenames as arguments to a command. Here’s an example using `ls` to give a detailed listing of the two `more` filters, the old and the new:

```
95 D% whereis more
c:\hamilton\more.exe
c:\os2\more.com
96 D% ls -l ‘whereis more‘
---A-  Mar 20  8:00      20123  c:\hamilton\more.exe
---A-  Oct 26 12:00      31658  c:\os2\more.com
```

(Our more is ‘‘less filling and tastes better.’’)

The string inside the backquotes is passed directly to a child thread for interpretation. If there are any variable substitutions inside the backquotes, they’re done by the child, not the parent. This lets you easily embed `for` loops and other programming constructs into the command substitution.

Inside backquotes, only the backquote character needs to be escaped to avoid having it processed by the parent thread.

Inline Data

A novel variation on i/o redirection is inline data, also called ‘‘here’’ documents: literal text you want the shell to feed a command as stdin. Here’s an example:

```
97 D% cat <<eof
98 D? (this is the inline data)
99 D? eof
(this is the inline data)
100 D% _
```

The “<<” operator is followed by a string the shell is asked to look for to mark the end of the inline data. The end-of-data string can be virtually anything you like, including wildcard characters, dollar signs, etc.; their normal meaning is turned off and they’re treated as ordinary literal characters. Only quote or escape characters have any special meaning, which is to turn off substitutions in the inline text (as we’ll discuss in a moment). Continuation lines as the shell collects the inline data get a different prompt, controlled by the `prompt2` variable. Once the data has been collected in memory, it’s written through a pipe to the command.

One very convenient use of inline data is when you want to quickly search for any one of a number of important words in a large library. E.g., to scan for some specific strings in a set of C files:

```
100 D% fgrep -ns <<xxx ~\sh\*.c
101 D? DosOpen
102 D? DosClose
103 D? DosRead
104 D? DosWrite
105 D? xxx
:  search results
```

In situations where the inline data is being created inside a larger structure, the data is assumed to start on the first line following a break between statements. For example, inside a `for` loop:

```
106 D% for i = 1 to 3 do
107 D? cat <<eof; echo i = $i
108 D? (this is the inline data)
109 D? eof
110 D? end
(this is the inline data)
i = 1
(this is the inline data)
i = 2
(this is the inline data)
i = 3
```

If you want to put several inline i/o redirections on the same line, type the associated inline data sections, each with its own terminating string, in the same left-to-right order in which they appeared.

So far, we’ve just shown examples involving static text. But it’s also possible to ask the shell to do command and variable substitutions on the inline text:

```
111 D% cat << ***
112 D? The ^$home directory is $home.
113 D? Today’s date is ‘date‘.
114 D? ***
The $home directory is d:\doug
Today’s date is Wed   Mar 17 1993   14:40:51.12.
115 D% _
```

Notice that although substitutions and escape characters inside the here document are processed, quotes (both single and double) are not.

[The C shell implements here documents by spawning a child thread to do any substitutions and write the results into a pipe feeding the current thread as it continues to evaluate the statement. If the here document contains references to shared variables, they’ll be evaluated by that other thread. And unless they’re local variables, the values will not be snapshotted when the here document thread is created. If the current thread (or any other thread) continues to make changes to a variable after the here document thread is spawned but before it evaluates the variable, the here document will contain the new, not the old value.]

Command and variable substitution and escape processing inside a here document is turned off if any part of the end-of-data string following the << is quoted (with single, double or backquotes) or escaped:

```
115 D% cat <<^***
116 D? The ^$home directory is $home.
117 D? Today’s date is ‘date‘.
118 D? ***
The ^$home directory is $home
Today’s date is ‘date‘.
119 D% _
```

Inline Data in Scripts

Inline data can be especially useful if you’re writing a script file or passing commands to the shell through a pipe. In either of these cases, the low-level DosReads to the OS/2 kernel (or WriteFiles to the NT kernel) cannot be depended on to stop at the end of a line because pipes and files are considered block-oriented rather than line-oriented like the keyboard. If too many characters are read, there’s no simple way to back up. For this reason, it’s not realistic to write a script where a child process is supposed to inherit stdin pointed into the script file. In a script file, this is ***not*** reliable:

```
:
:
csh
echo hello
exit
:
:
```

The file descriptor the child process inherits will likely not be pointing at the ‘‘`echo hello`’’; when it exits, the parent will likely not find it pointed just past the ‘‘`exit`’’. This type of script ***should*** be written as:

```
:
:
csh <<eof
echo hello
exit
eof
:
:
```

I/O Redirection

The History Mechanism

History

The history mechanism lets you recall and rerun earlier commands. To see what it's remembered, use the `history` command or its usual alias `h`, which might show you something like the following:

```
119 D% history 12
   108  (this is the inline data)
   109  eof
   110  end
   111  cat << ***
   112  The ^$home directory is $home.
   113  Today’s date is ‘date‘.
   114  ***
   115  cat << ^***
   116  The ^$home directory is $home.
   117  Today’s date is ‘date‘.
   118  ***
   119  history 12
120 D% _
```

The history list won't be quite what you typed: it'll be broken into separate words wherever one of the special tokens, &, `|`, `;`, >, <, `(`, `)`, &&, `||`, >> or <<, occurs. Only inline data escapes being broken up into words this way. Each command may be thought of as an array of words, indexed from 0.[†]

To reuse the text or maybe just a few words from a previous command, you type an exclamation point, "`!`", followed by a few characters to identify what you want to reuse. You can do this anywhere and whatever you select is just stuffed back on the command line to be interpreted as whatever the context suggests. For convenience, the exclamation point is not treated as a history reference if it's followed by white space (a space, tab or newline) or by "=", "~" or "`(`".

Retrieving a Whole Command Line

There several ways of picking up a whole command line. You already know how to do it interactively with arrow keys and command completion. You can also use a shorthand notation that can be more convenient if you want to do something a bit more complex. The simplest shorthand is "`!!`", which picks up the text of the immediately preceding command:

[†] Array indices always start with zero.

```
120 D% echo !!
echo history 12
history 12
121 D% !!
echo history 12
history 12
122 D% _
```

The shell first echoes your command showing the effects of the substitutions and then runs it. The other quick ways of referring to a whole command line from history are by the command number,

```
122 D% !96
ls -l  whereis more‘
---A-  Nov 28  16:57   24743  c:\os2\bin\more.exe
---A-  Oct 21   1987   48354  c:\os2\cmds\more.com
```

relative to the immediately preceding command[†],

```
123 D% echo one
one
124 D% echo two
two
125 D% echo three
three
126 D% !-1
echo two
two
127 D% _
```

or by mentioning some of the text to look for. A question mark after the exclamation point means you’ll accept the match anywhere on the line; otherwise it has to be at the start.

```
128 D% !h
history 12
   116  The ^$home directory is $home.
   117  Today’s date is ‘date‘.
   118  ***
   119  history 12
   120  echo history 12
   121  echo history 12
   122  ls -l  ‘whereis more‘
   123  echo one
   124  echo two
   125  echo three
```

[†] In this context, the history list can be thought of as an array starting with the zeroth element being the immediately preceding command line. The negative index captures the notion of counting backwards in time and differentiates the syntax from references by command number. See also the `bsdhistory` variable.

```
126  echo two
127  history 12
```

A search string ends at the first word boundary. This is so it's convenient to type additional text following without having it be confused as part of the search string. For example:

```
128 D% !?one;!?two;!?thr
echo one ; echo two ; echo three
one
two
three
129 D% _
```

Retrieving Individual Words

To pick off individual words of the immediately preceding command, there's some convenient shorthand. "`!*`" gets all the argument words:

```
129 D% echo now is the time
now is the time
130 D% echo Finally, !* to begin
echo Finally, now is the time to begin
Finally, now is the time to begin
131 D% _
```

"`!$`" gets just the last word:

```
131 D% echo the last word was !$.
echo the last word was begin.
the last word was begin.
132 D% _
```

and "`!^`" gets just the first argument word:

```
132 D% echo ===!^=== time is here
echo ===the=== time is here
===the=== time is here
133 D% _
```

Notice that a history substitution can be smashed right up against other literal text.

In the chapter on editing, additional facilities for selecting individual words or doing a search/replace will be introduced.

History Short-Form

Recognizing how frequently one would like to make a simple change to the immediately preceding command to correct a typo, the history mechanism provides a short form for just that purpose. ‘‘%’’ typed as the first character on the command line indicates that a search/replace pair follows:

```
133 D% echo hello world
hello world
134 D% %world%friends%
echo hello friends
hello friends
```

Typing ‘‘%%’’ matches the beginning of the line:

```
135 D% %%echo %
echo echo hello friends
echo hello friends
```

It’s also possible to refer to the search string in the replacement string by using an ampersand. (This example also illustrates that the trailing ‘‘%’’ isn’t required unless you want to explicitly mark the end of the replacement.)

```
136 D% %friends%family, & and neighbors
echo echo hello family, friends and neighbors
echo hello family, friends and neighbors
137 D% _
```

Obviously, that raises the question: how do you put a literal ampersand in the replacement? Simple. Just quote it with ‘‘^’’, the shell escape character.

```
137 D% %and%^&
echo hello family, friends & neighbors
hello family, friends & neighbors
138 D% _
```

Variables

As with any conventional programming language, the shell provides a facility for storing values in variables.

Environmental Variables

Some of the variables are part of the environment, passed along to any child process or screen group. Many of the environment variables will have been created just this way, e.g., set to a value passed along when you start the C shell from Group Main (on OS/2) or the Program Manager (on NT). To list those currently defined, use the `setenv` command. (If you're using Windows NT, mentally edit this example to imagine it says `nt` everywhere you see `os2`.)

```
138 D% setenv
COMSPEC      c:\os2\cmd.exe
COLORS       white on blue
DPATH        c:\os2;c:\os2\system;c:\os2\install;c:\;
HOME         d:\doug
INCLUDE      c:\os2\include
LIB          c:\os2\lib
PATH         .;c:\os2\bin;c:\os2\cmds;c:\os2
PROMPT       $iOS/2 $n$g
PROMPT1      $@ $CDISK%
PROMPT2      $@ $CDISK?
SHELL        c:\os2\bin\csh.exe
TABS         3
TMP          e:\tmp
```

Variable names are case-sensitive on OS/2 but case-*in*sensitive on NT. They can be of arbitrary length. A name must start with an upper- or lower-case alphabetic character or underscore (''_'') or at-sign (''@''); remaining characters may any of these or decimal digits. Many of the environmental variables have specific meanings. For example, the `PATH` variable tells where to look for executable files, etc. Details describing the meaning of each variable are given in the language reference section.

The `setenv` command can also be used to create a new environmental variable or alter or display the value of an existing one:

```
139 D% setenv zork = this is the zork variable
140 D% setenv zork
zork         this is the zork variable
141 D% _
```

If the list of words being assigned to the variable includes any special tokens, it's often useful to use the parenthesized variant of `setenv`. In this example, the ''>'' would have been confused as an i/o redirection if it weren't inside parenthesis. Notice that the parenthesis are stripped off before the assignment is made.

```
141 D% setenv greatest = (At Berkeley, they say 4.3 > V)
142 D% setenv greatest
greatest     At Berkeley, they say 4.3 > V
```

Even though the special meaning is lost, text inside the parenthesis is still broken down into words, as shown in this example:

```
143 D% setenv pdirs = (.;c:\os2\bin;c:\os2\cmds;c:\os2;)
144 D% setenv pdirs
pdirs        . ; c:\os2\bin ; c:\os2\cmds ; c:\os2
```

(To avoid having the text broken up into words, use single or double quotes around the string instead.)

Set Variables

Set variables do not get passed to a child process but are shared among all threads. To get a list of those currently defined, use the `set` command:

```
145 D% set
CDISK        D
argv
cdhome       0
cdisk        d
:
:
path         . c:\os2\bin c:\os2\cmds c:\os2
precision    6
:
:
```

Some of the set variables are linked to the environmental variables: you change one, and the other changes too. For example, `path` contains the same information as `PATH` but, because it's been parsed into individual words, it's often a bit more useful.

On NT, this linkage can pose a bit of a problem. Since the convention on NT is that environmental variables are supposed to be case-insensitive, there clearly is a conflict between, for example, the `PATH` environmental and `path` set variables. The C shell resolves this by making the set, unset and @ statements case-sensitive (so you can still create set variables that differ from environmental variables only by case) but the setenv and unsetenv and $var and other variable references first try case-sensitive, then case insensitive variable lookups.

Many of the set variables are pre-defined by the shell to control various aspects of how errors are handled, etc. In some cases, each thread maintains its own copy. For example, it wouldn't do to insist that all threads must use the same value for the `cwd` (current working directory) variable! The rest of the variables, including any defined by the user, are shared among all threads: if one thread changes a value, all the other threads see the change immediately. As we'll see later, this has some implications when spawning background activities.

In other respects, `set` works just like `setenv`:

```
146 D% set privatezork = this is the private zork variable
147 D% set privatezork
privatezork   this is the private zork variable
148 D% _
```

Once a variable has been created as either a set or an environmental variable, it stays that way: to change it from set to environmental, you must first `unset` (`unsetenv`) it, then redefine it.

Local Variables

We just mentioned that not all the pre-defined set variables are shared. Individual threads get their own private copies of some because to do otherwise wouldn't be sensible. Sometimes you need the same sort of control over the variables you create. You don't want to share a variable with other threads or even with commands outside a very narrow context.

You accomplish this making the variable local, which means it's hidden from outer control blocks or other threads. Local variables are really important, as we'll see later, for recursive procedures or for procedures you want to use from multiple threads. To define a variable as local, use the `local` statement, which accepts a list, separated with commas, of all the variable names you want to be local. When a new local variable is created, its initial value is always null (zero words), even if there was a previous definition. Here you can see how the variable `i` is redefined inside the nested statements but once you exit from the nest, the old value of `i` is again visible:

```
148 D% set i = hello world
149 D% echo $i
hello world
150 D% (local i; echo $i; set i = how are you; echo $i)

how are you
151 D% echo $i
hello world
```

When you spawn a child thread, e.g., as a background activity or as the second or following stage of a pipeline, it gets copies of all your local variables, snapshotted at the time it's spawned. If either the parent or the child later changes to the value of any those local variables, they affect only it's own copy.

Variables

Variable Substitutions

The simplest way to use a variable is in a simple substitution, where a dollar sign is used to indicate that a variable name follows. (Similar to using a ''%'' in `cmd.exe`.) The value is substituted in and the statement is evaluated.

```
152 D% echo $HOME
d:\doug
```

Text surrounding the variable reference is generally just pasted around the value that's substituted in:

```
153 D% echo My home directory is ---$HOME---
My home directory is ---d:\doug---
```

If the surrounding text would be confused as part of the variable name, it's necessary to insulate the variable reference with braces. For example:

```
154 D% echo ${HOME}XXX
d:\dougXXX
```

nonovar

If you try to reference a variable, procedure or an alias and it doesn't exist, it's considered an error unless you set the `nonovar` variable to indicate how you want the situation treated.

```
155 D% echo $nonesuch
csh:  The variable 'nonesuch' is not defined.  To suppress this
error, set nonovar = 1 (pass through) or 2 (discard).
156 D% set nonovar = 1
157 D% !e
echo $nonesuch
$nonesuch
158 D% set nonovar = 2
159 D% !e
echo $nonesuch

160 D% set nonovar = 0
```

How Variables are Stored

Each variable is kept internally as a list (an array) of objects. An individual object can be a (possibly null) character string, a 32-bit integer or a 64-bit floating point value. Generally speaking, it's not necessary to worry too much about how a specific object is represented, though, since the shell automatically does any necessary conversions to allow a value to be used sensibly in any given context.

Notice, however, that even though the value of an environmental variable may be a list, it is always rendered as a simple character string when it's passed to a child process. Here's an example using the ''$#'' notation to ask how many words are in a variable's value:

```
161 D% echo $zork
this is the zork variable
162 D% echo $#zork
5
163 D% csh
Hamilton C shell(tm) Release 2.2
Copyright (c) 1988-1993 by Hamilton Laboratories.  All rights
reserved.
1 D% echo $zork
this is the zork variable
2 D% echo $#zork
1
3 D% exit
164 D% _
```

In this example, `zork` holds five words: ''`this`,'' ''`is`,'' ''`the`,'' ''`zork`,'' and ''`variable`.'' But when we start up a new child process running the shell, the child process sees `zork` as holding only a single word: ''`this is the zork variable`''.

Here's another example where we assign a floating point value to an environmental variable. In the current process, the exact binary floating representation is used. When it's passed to a child process, the value is first converted to a character string, losing some of the precision. This example also introduces the `calc` statement which evaluates an expression and prints the value. In an expression, a variable name is recognized even without a ''$'' to introduce it; in fact, that's the preferable way to do it. If you use a ''$''-style variable substitution, the shell pastes in a character-string representation, again losing precision. Also, the full range of C language expression operators is available.

```
164 D% setenv envVar = 0
165 D% calc ++envVar
1
166 D% calc envVar /= 7
0.142857
167 D% calc envVar*7
1.000000
168 D% csh <<eof
169 D? calc envVar*7
170 D? exit
171 D? eof
Hamilton C shell(tm) Release 2.2
Copyright (c) 1988-1993 by Hamilton Laboratories.  All rights
reserved.
0.999999
172 D% calc $envVar*7
0.999999
```

Variables

Arrays

Variables can hold a list of values indexed as an array. As with the C language, arrays are indexed, counting from zero, with an expression inside ``[...]'' brackets. The expression can be arbitrarily complex, but it must evaluate to an exact integer value. The next example shows how a floating point value can be used, but if it's not precisely an integer, an error results. The precision variable controls the number of digits past the decimal point used when displaying a floating point value; the maximum is 17 digits.

```
173 D% set x = now is the time for all good men
174 D% calc x[5.000000]
all
175 D% calc 2**500
3.27339061e+150
176 D% calc y = log2(!$)/100
calc y = log2 ( 2**500 ) /100
5.000000
177 D% calc x[y]
csh:  Couldn't evaluate the index to the variable 'x'.
178 D% set precision = 30
179 D% calc precision
17
180 D% calc y
5.00000000000000100
181 D% calc x[round(y)]
all
```

An array cannot be ``sparse;'' i.e., before you can create an element 5, element 4 must exist and so on. But it's perfectly legal to keep adding new elements onto the end of array as long as the new element is the zeroth element or only one past whatever's currently the last element. Here's an example using a `for` loop. The @ statement is like `calc` except it doesn't print anything.

```
182 D% unset x
183 D% for i = 0 to 4 do
184 D? @ x[i] = 2*i
185 D? end
186 D% echo $x
0 2 4 6 8
```

nullwords

When you index an array, if the element you specify doesn't exist, that's normally considered an error. For example, `x` contains 5 words; trying to reference a sixth fails:

```
187 D% calc x[5]
csh:  Illegal subscript in variable reference 'x[5]'.  (To
suppress this error, set nullwords = 1.)
```

The `nullwords` variable is used to tell the shell that you'd like to ignore subscripting errors when you try to *evaluate* a variable reference. `nullwords` has no effect over subscripting errors when you try to set the value of a non-existent element. Here are some examples:

```
188 D% set nullwords = 1
189 D% echo -- "--->$x[999]<---"
---><---
190 D% calc x[999] = "does not work"
csh:  Illegal subscript in variable reference 'x[999]'.
```

The use of the ''`--`'' option and of double quotes was important: ''`--`'' told `echo` that it had reached the end of any options, allowing it to print something that began with a minus sign. The quotes were used, in the first case, to turn off recognition of redirection characters ''>'' and ''<'' but still get the variable substitution. The second time, it was to make the string, ''does not work'' a single word. (If you try leaving off the quotes or not using ''`--`,'' you'll see that the error messages are what you'd expect.)

Variables

Wildcarding

The notion of wildcarding is pretty simple: the user gives just a few characters describing the filename he's looking for and system fills in the rest. With ''vanilla'' OS/2 or NT, wildcarding is the responsibility of each application, based on the command-line arguments it's given. Typically, the application designer fulfills this by linking in a library routine which does a simple-minded half-hearted wildcarding.

Hamilton C shell does the wildcarding before invoking the application. The shell's wildcarding includes five components: home directory expansion, wildcarding characters, ranges, alternation and indefinite directories. A powerful recursive match algorithm is employed to guarantee a sensible result no matter how complex the pattern.

Home Directory Expansion

The tilde character, ''~'', is recognized as shorthand for the home directory. In the simplest form, we can use it just by itself:

```
191 D? echo $home
d:\doug
192 D% cd ~
193 D% cd
d:\doug
```

There's also shorthand for children or siblings of the home directory:

```
194 D% cd ~\samples
195 D% cd
d:\doug\samples
196 D% cd ~carol
197 D% cd
d:\carol
```

Wildcard Characters

The wildcard characters, ''*'' and ''?'', provide shorthand for ''match any string'' and ''match any single character,'' respectively.

Suppose the home directory contained the following contents:

```
198 D% cd ~
199 D% ls
```

Wildcarding

```
bcs         mandel      sh          ex.rc       release.csh
bix         mba         testcode    icon.ico    ring.ico
channel.one online      util        login.csh   snapshot.csh
dial        postscpt    word        mail        startup.csh
excel       regressn    backup.csh  os2init.cmd vi.ini
games       resume      brite.csh   popup.txt
icon        samples     class.txt   prime.c
```

The following example shows the use of “?” to match any single character. Wildcard results are always shown alphabetically in lower case. No distinction is made between directories and files.

```
200 D% echo ????
dial icon mail util word
201 D% echo b??
bcs bix
```

The “*” can match zero or more arbitrary characters except “`:`” or “\”; in contrast to DOS-style wildcarding, “*” *can* match “`.`”. If there are ordinary characters in the pattern, they must also be matched.

```
202 D% echo *mp*e*
samples
```

Because the wildcarding is done *before* the command is invoked (without the command even being aware), wildcarding can even be done on a `cd` command:

```
203 D% cd !$
cd *mp*e*
204 D% cd
d:\doug\samples
```

Wildcarding is most emphatically *not* restricted to matches only against a single directory level. Here’s an example that wildcards across all the subdirectories, looking for `.c` files that begin with “`a`”.

```
205 D% cd ..
206 D% echo *\a*.c
samples\args.c sh\allocate.c
```

Wildcarding can even be done against driveletters. For example:

```
207 D% echo *:\*\q*
i:\mail\quotes.doc i:\tmp\query.out j:\doug\quantity.disc
```

When wildcarding against driveletters, the shell restricts the set of drives it will search down to just those specified by the `DRIVEMASK` environment variable. If you don’t specify a `DRIVEMASK`, the default is all drives except the floppies `a:` and `b:`. The search is restricted so you don’t waste time trying to access slow removable media that may not even be ready.

Ranges

Ranges describe a set of characters, any one of which will be matched. It's specified as a list of acceptable characters inside "`[...]`" brackets. The range "`[be]`" means either "`b`" or "`e`"; "`[b-e]`" is shorthand for any character in the sequence "`b`" through "`e`". Within the brackets, any number of hyphenated sequences and single characters can pasted one after the other in any order. For example, "`[a-cu-zgkmp]`" is a perfectly legal range. Here are a couple examples. Notice that ranges can also be used with driveletters.

```
208 D% echo [be]*
backup.csh bcs bix brite.csh ex.rc excel
209 D% echo[d-g]:\[s-t]*
d:\taxes d:\tmp e:\spool e:\startup.cmd e:\temp e:\toolkit.sys
f:\swap f:\tmp f:\toys g:\skip g:\temp g:\tmp
```

An exclusion range is written as a set of characters inside the brackets that starts with a circumflex. It'll match any single character *not* in the range.

```
210 D% echo [^a-t]*
util vi.ini word
```

Alternation

Alternation, specified with "`{...}`" braces, is a shorthand way of specifying that all the combinations of frontparts and backparts should be generated. There isn't any requirement that the filenames constructed actually exist.

```
211 D% echo {zork,gadzooks}.csh
zork.csh gadzooks.csh
212 D% echo {a,b}{c,d}{e,f}
ace acf ade adf bce bcf bde bdf
```

Alternation can be combined arbitrarily with the other wildcard constructs:

```
213 D% echo {[bc],*r}*i*
bix brite.csh brite.csh ring.ico
```

Indefinite Directories

The ellipsis, "`...`", is an indefinite definite directory wildcard. It'll match zero or more arbitrary directory levels -- whatever it takes to make the rest of the wildcard match. To be recognized as a wildcard, the context must indicate it's really a filename, i.e., it must be preceded by "`\`", "`/`", "`~`" or "`:`" or followed by "`\`" or "`/`". For example, to find all the `.inf` files anywhere on the `C:` drive, one might type:

```
214 D% ls c:\...\*.inf
c:\os2\book\cmdref.inf
```

As with all the wildcard constructs, the indefinite directory construct can be used completely arbitrarily. It can even be used several times in the same wildcard. But do notice if you do that, there is a possibility of getting the same file listed more than once:

```
215 D% ls f:\...\a*\...\money*
f:\os2\aldus\art\moneycht.eps    f:\os2\aldus\art\moneycht.eps
```

This can happen if there's more than one possible way to match the same pathname. In this example, the “`a*`” part could matched either “`aldus`” or “`art`” with the first “`...`” matching either “`os2\aldus`” or “`os2`” and the second “`...`” matching either “`art`” or just zero levels.

Match Failures

When you specify a sequence of wildcard patterns and *none* of them match, it's normally treated as an error. In this example, the first command causes an error because there's no file or directory name with a “`z`” in it. The second command executes without error because, out of the *sequence* of patterns, there's at least one match.

```
216 D% echo *z*
csh:  Wildcarding failed to produce any matches.  To suppress
this error, set nonomatch = 1 (pass through) or 2 (discard).
217 D% echo *z* sa*
samples
```

In this context, the fact that alternation caused something to be generated is not the same as a match. In the next example, “`{zork,gadzooks,*z*}.csh`” is the same as “`zork.csh gadzooks.csh *z*.csh`”; only the last element involves any matching, and it fails.

```
218 D% echo {zork,gadzooks,*z*}.csh
csh:  Wildcarding failed to produce any matches.  To suppress
this error, set nonomatch = 1 (pass through) or 2 (discard).
```

The `nonomatch` variable lets you control how a wildcard failure is treated. It works just the way `nonovar` works when you reference to a non-existent variable.

```
219 D% set nonomatch = 1
220 D% echo *z*
*z*
221 D% !s:s/1/2/
set nonomatch = 2
222 D% !e
echo *z*

223 D% !s:s/2/0/
set nonomatch = 0
224 D% !e
echo *z*
```

```
csh:  Wildcarding failed to produce any matches.  To suppress
this error, set nonomatch = 1 (pass through) or 2 (discard).
```

Caution: The copy, xcopy, rename and del commands

Hamilton C shell expands out wildcards before it invokes the application you name. This is not what the copy and xcopy commands expect! Suppose there are two files, `file.a` and `file.b` on your diskette `a:`, that you wanted to copy to your current drive. Under `cmd.exe`, it would be natural to type:

```
[D:\DOUG] xcopy.exe a:*.*
Source files are being read...

A:FILE.A
A:FILE.B

2 file(s) copied.
```

The destination is implicit. `xcopy` understands the wildcarding to mean ''copy everything on drive a: to the current disk and directory.'' That is *not* what would happen under the C shell! Because the wildcard would be expanded first, it would act instead as if you had typed:

```
[D:\DOUG] xcopy.exe a:file.a a:file.b
Source files are being read...

A:FILE.A

1 file(s) copied.
```

Do you see what happens? If wildcarding is done first, the `xcopy` command sees just the two filenames and figures you mean to copy one right over the other. `file.b` is lost! For this reason, the normal `startup.csh` file contains some carefully constructed aliases and procedures to intercept the `copy`, `xcopy`, `rename` and `del` commands:

```
proc safecopy(files)
   cmd /c copy $files; @ nowild = s; unlocal s
end
alias copy   (local s; @ s = nowild; @ nowild = 1; safecopy)

proc safexcopy(files)
   xcopy.exe $files; @ nowild = s; unlocal s
end
alias xcopy  (local s; @ s = nowild; @ nowild = 1; safexcopy)

proc saferename(files)
   cmd /c rename $files; @ nowild = s; unlocal s
end
```

```
alias rename (local s; @ s = nowild; @ nowild = 1; saferename)
alias ren    rename

proc safedel(files)
   cmd /c del $files; @ nowild = s; unlocal s
end
alias del      (local s; @ s = nowild; @ nowild = 1; safedel)
alias erase    del
```

The way this works by saving the current value of `nowild` (which tells whether wildcarding is should be done), turning off wildcarding, invoking the `copy/xcopy/rename/del` command, then restoring the wildcarding state. `s` is a temporary variable that gets discarded after its been used.

Be sure to always invoke `copy`, `xcopy` `rename` and `del` via these aliases. If you encounter other applications that really must do their own wildcarding, use this same technique with them.

Editing

Often, the text returned by a history, variable or command substitution isn't quite what you want. For example, you may want to select only certain words, do a search/replace, or manipulate a filename that's been returned. The editing facilities provide these capabilities.

The examples in this chapter show a sampling of various combinations of editing commands and types of substitutions. There simply isn't room to show all the possibilities nor is there really a need to: with two exceptions ("`:%`"and "`:p`"), any editing command can be applied against any substitution type or against the result of another editing command.

Word Selections

Editing modifiers begin with a colon, "`:`", followed by an editing command. You can select words either by number or symbolically: first, last, all, etc. Here are examples of the "`:*`", "`:$`" and "`:^`" modifiers to select all, last, and first argument words, respectively. Notice that any number of editing modifiers may strung together. For example, "`:*:^`" means the first argument word (word 1) of the sequence formed of all the original argument words: i.e., word 2.

```
225 D% echo Finally, now is the time
Finally, now is the time
226 D% echo !!:$
time
227 D% echo !?Fin:*:^
echo now
now
```

It is also possible to select words by indexing through the array associated with a given substitution. The words are counted from zero.

```
228 D% echo ‘echo now is the time‘:2
the
```

Ranges of words can also be specified with a hyphen between the beginning and ending word indices. In this context, the minus sign indicates a range, *not* subtraction.

```
229 D% set x = now is the time for all good men
230 D% echo $x:1-5
is the time for all
```

When the operand is a "`!?`"-style history substitution, there's a special operator, "`:%`", for selecting the particular word that triggered the match:

```
231 D% echo !?Fin:%
echo Finally
Finally
```

Each of the three symbolic word selectors, ''`^`'', ''`$`'' and ''`%`'', can be used anywhere a decimal word number would be acceptable. For example:

```
232 D% echo !?time:%-$
echo time for all good men
time for all good men
```

Search/Replace Operations

A search/replace operation looks for and replaces a simple character string. (For those familiar with such things, it does not use regular expressions.) If the search string isn't given, the one used last time is used again. If the replace string contains an ampersand, ''&'', that isn't escaped with the circumflex, it's expanded into the search string. Here are a few examples. Notice that putting a ''`g`'' at the start of an operation makes it ''global.''

```
233 D% echo !?now:%-$:s/o/O/
echo nOw is the time for all good men
nOw is the time for all good men
234 D% !!:gs/t/T/
echo nOw is The Time for all good men
nOw is The Time for all good men
235 D% !!:s/nOw is/& really/
echo nOw is The Time for all good men
nOw is really The Time for all good men
```

Pathname Editing

Another set of operators allows filenames to be easily manipulated. As with the search/replace operator, pathname editing is normally applied to only the first operand word; if you want the operation performed on all the words, you must make it global. In this example, the ''`:h`'' (head) operator is used, which returns the name of the directory containing the given file.

```
236 D% echo *\a*.c
samples\args.c sh\allocate.c
237 D% set dotc = !$
set dotc = *\a*.c
238 D% echo $dotc
samples\args.c sh\allocate.c
239 D% echo $dotc:h
samples sh\allocate.c
240 D% echo $dotc:gh
samples sh
```

Specialized Operations

Specialized operations are provided for scanning a character string and breaking it up into words and applying quotes around each word.

The "`:x`" operator for breaking up a string into words is particularly useful for parsing text read with the `getline` pseudo-variable, which always returns the line read as a single character string:

```
241 D% @ data = $<
(I typed -->this<-- in)
242 D% echo $#data $data
1 (I typed -->this<-- in)
243 D% set data = $data:x
244 D% echo $#data $data
10 ( I typed -- > this < -- in )
```

The "`:q`" operator pastes single quote marks around every word in the operand list. As we'll see in the chapter on quoting, this prevents any further wildcard, variable or command substitutions from being done.

```
245 D% echo sa?pl?s *\a*.c
samples samples\args.c sh\allocate.c
246 D% echo !*:q
echo 'sa?pl?s' '*\a*.c'
sa?pl?s *\a*.c
```

History Edits

For history substitutions, the "`:p`" operator can be used to cause the result of the history substitution to be echoed and entered into the history list but for the command not to be executed. This is helpful when you're trying a complicated edit and not sure if the result is going to be what you want.

```
247 D% !?Fin:p
echo Finally
248 D% !!
echo Finally
Finally
249 D% _
```

Editing

Quoting

The shell has several quoting mechanisms for marking a section of a command for special processing. One of them, command substitution, which used the `‘...‘` syntax, was already discussed in the chapter on i/o redirection; that discussion won’t be repeated here.

The other quoting mechanisms focus more simply on the problem of overriding the special meanings that certain characters have.

Double Quotes

Double quotes are of use when you want to pass a character string containing a space or other word separator to an application. Normally, if you called a C program, it would see these words as separate `argv` entries instead a single character string; double quotes prevents this breakup into words. We can demonstrate this using the simple `myecho` program in the `samples` directory which prints out the `argv` list it receives with single quotes around each entry:

```
249 D% cd ~\samples
250 D% myecho hello world
’myecho’ ’hello’ ’world’
arg length = 19 characters
251 D% myecho "hello world"
’myecho’ ’hello world’
arg length = 19 characters
```

Double quotes also turn off the special meaning of the various wildcard characters and the single quote:

```
252 D% echo "* isn’t a wildcard character inside quotes"
* isn’t a wildcard character inside quotes
253 D% echo "~"
~
254 D% _
```

Command, history and variable substitutions inside double quotes are still done:

```
254 D% echo "*** The home directory is $home ***"
*** The home directory is d:\doug ***
255 D% echo "‘echo ~‘"
d:\doug
256 D% echo "myecho.c is ‘wc -l < myecho.c‘ lines long"
myecho.c is 24 lines long
257 D% echo "!?samples"
echo "cd ~\samples"
cd ~\samples
```

Quoting

Single Quotes

Single quotes are a little more brute force to turn off special meanings. Wildcards, variables and command substitutions are all treated as ordinary text. Only history references are recognized inside single quotes.

```
258 D% echo ’*’
*
259 D% echo ’$cwd’
$cwd
260 D% echo ’‘echo hello‘"
‘echo hello‘
261 D% echo ’!?samples’
echo ’echo "cd ~\samples"’
echo "cd ~\samples"
262 D% _
```

It is not necessary to quote an entire word. It’s possible (and often useful) to selectively quote just as much as desired. The quoting characters are processed out just before invoking the command. Example:

```
263 D% echo ’no’w is "t"h’e’ ‘echo time‘
now is the time
```

Shell Escape Character

The shell escape character is normally the circumflex[†], ‘‘^’’. It has two uses: preceding any of the special characters, it turns off that special meaning. When followed by an alphabetic character or hex or octal number, it can be used to enter binary data or characters that couldn’t easily be typed. To get a literal escape character, type two escapes in a row. These specific escape sequences have special meaning:

`^a`	Audible Alert (Bell)	`^r`	Carriage Return
`^b`	BackSpace	`^t`	Tab
`^f`	Form Feed	`^v`	Vertical Tab
`^n`	NewLine	`^^`	Single escapesym

At the very end of a line, the escape has a special meaning: the next line is a continuation line. Inside a quoted string, the ‘‘^’’-newline combination will be replaced with a simple newline; anywhere else, the combination is just turned into a space. The other special case is when it immediately follows ‘‘`[`’’. Since ‘‘`[^...]`’’ is a wildcard exclusion range, the ‘‘^’’ in this case is treated as a literal character so you won’t have to type two of them in a row.

Escape characters work even inside single or double quotes.

[†] The circumflex was chosen as the default shell escape character to be consistent with OS/2 conventions. Choosing the Unix backslash instead would have conflicted badly with OS/2 filename conventions. Nonetheless, the `escapesym` variable does allow the adventuresome to make a different choice.

```
263 D% echo now ^
is the time
now is the time
264 D% echo "now ^
is the time"
now
is the time
265 D% _
```

Quoting just part of a Word

It's possible to combine the quoting mechanisms or use them on just the part of a string you want quoted. For example,

```
265 D% echo '$cwd='$cwd
$cwd=d:\doug
```

Here's another example, searching through a series of `.csh` files, looking for those that are self-loading procedures. For example, we can spot that `whereis.csh` is self-loading because it contains a line like this:

```
whereis $argv
```

To look for occurrences of this sort, we might loop through a list of `.csh` files, `grep`'ing each for the filename (minus the `directory and .csh` extension) followed by white space followed by "`$argv`":

```
266 D% foreach i (~\samples\*.csh)
267 D?    grep $i:b'[ ^t]*$argv' $i
268 D? end
bits $argv
bumpdate $argv
caldate $argv
calendar $argv
:
```

Notice how the "`$i:b`" part is outside the quotes so that the filename can be substituted in and edited to strip off the directory and extension. Conversely, the "`[ ^t]*`" and "`$argv`" portions are inside the quotes to avoid having them confused as a wildcard or variable substitution, respectively.

Wildcarding with Special Characters

If you'd like to wildcard filenames that have literal $'s, ['s, quotes or spaces (under HPFS) or other special characters, you'll have to quote or escape the special characters to turn off their special meanings. For example,

Quoting

```
269 D% ls g:\tmp
$abc          [hello          this name has spaces
270 D% ls ’$’*
$abc
271 D% ls *^ *
this name has spaces
```

Expressions

As we've seen, sometimes a character like ''`*`'' means wildcard and sometimes it means multiply. The meaning of what you type is determined by the context. The shell makes a distinction between words, used as arguments to a command versus an expression context.

In general, expressions are expected wherever the context would seem to suggest that it would be more natural to think of calculating a value as opposed to using wildcarding to produce a list of filenames.

Expressions

The shell's expression grammar is based on that of the C language and provides the full range of arithmetic, logical, bit, indexing and relation-testing and assignment operators. In addition, there are file system tests and pattern matching string compares. To use the shell as simple calculator, use the `calc` statement. This example shows a call to the square root routine, one of the built-in procedures.

```
272 D% calc sqrt(2*pi)
2.506628
```

which writes its result to stdout. If you want to do the calculation silently, use the ''`@`'' variant:[†]

```
273 D% @ r = 12
274 D% @ area = pi * r**2
275 D% calc area
452.389345
```

In addition to the `calc` and `@` statements, other examples where an expression is expected include a variable index inside ''`[...]`'' brackets, in a procedure argument list and, as we'll see, a number of the structured programming constructs such as the `for` statement.

Expression Parsing

All commands are first broken down into words. A word is anything separated by a space or a tab or one of the following special strings: `&` , `|`, `;`, >, <, `(`, `)`, &&, `||`, >> or <<.

After a command line has already been broken up into words, if the context is an expression, it's further broken up into tokens. A token is a variable or procedure name, a character or numeric literal, or one of the expression operators. Spacing between tokens is more-or-less arbitrary: for example, there's certainly no need to put spaces around an arithmetic operator to separate it from a variable name beside it.

[†] The choice of @ is a pun: ''at-sign-ment'' statement.

Expressions

Tokens are separated by any of these characters or character pairs: &, `|`, `^`, `+`, `-`, `*`, `/`, %, `//`, `=`, `!`, `~`, <, >, `(`, `)`, `[`, `]`, `,`, `:`, `;`, `-A`, `-D`, `-H`, `-S`, `-d`, `-e`, `-f`, `-o`, `-w`, `-x`, `-z`, ++, --, `**`, <<, >>, `==`, `!=`, `=~`, `!~`, `+=`, `-=`, `*=`, `/=`, %=, `//=`, `^=` and `**=`. The <=, >=, <<=, >>=, &=, and `|=` are always broken up into separate words before expression parsing begins; for consistency, the parser will accept any of ''*op*='' assignment operators with a space between the ''*op*'' and ''='' parts.[†]

Since the shell knows that any names it encounters in an expression must refer to variables or procedures it's not necessary to use a dollar sign to introduce a variable name. In fact, you'll find that performance is actually a bit better if you do *not* use a dollar sign. The reason is because a $-style variable substitution is evaluated by converting the internal value of the variable to a string and pasting that into the expression where quite often the next step is just to convert it right back again into the integer or floating point value it started out as. Also, if floating point is involved, you may notice some loss of precision. (But don't misunderstand, it is still perfectly legal to use $-style variable and other substitutions in an expression.)

Character literals must appear inside single or double quotes. Numeric literals can be entered in decimal, octal or hex. Octal numbers can contain only the digits `0` through `7` and must begin with `0`. Hex numbers must start with ''`0x`'' and contain only `0` through `f`. (Either upper or lower case is acceptable.)

Expression Operators

Expressions are evaluated according to the relative precedence of each operator in the expression. For example, multiplication is done before addition. The complete precedence hierarchy is shown in tabular form in the language reference.

```
276 D% calc 2 + 3*5
17
```

Some of the operators will be foreign, though we trust, not too difficult to use. The file system tests are unary operators. Each takes the name of a file or directory and tests it for existence, zero-length or some other interesting characteristic. Since the operand is a pathname, the parser temporarily shifts to word mode to read it because word mode is more natural for pathnames. The pathname can include wildcards and should *not* be enclosed in quotes. In the example that follows, ''`-e`'' tests for existence; ''`-D`'' tests whether the name is a directory.

```
277 D% cd ~\samples
278 D% ls
args.c        colors.csh    factor.csh    mcvisa.csh    readme
args.exe      deltaday.csh  finance.csh   myecho.c      ts.csh
bits.csh      dumpenv.c     getprio.c     myecho.exe    viopaste.c
bumpdate.csh  dumpenv.exe   getprio.exe   newfiles.csh  viopaste.exe
caldate.csh   duplicat.csh  julian.csh    rcode.c       weekday.csh
```

[†] The grammar is not perfectly lr(1): proper recognition of the file system tests and the assignment operator requires that parsing decisions in some places have to look ahead two tokens, not just one.

```
calendar.csh  easter.csh    makecpgm.csh  rcode.exe
279 D% echo a*c
args.c
280 D% calc -e a*c
1
281 D% calc -D !$
calc -D a*c
0
```

File System Tests

The value returned from a file system test is always `1` or `0`; there are no restrictions on how the value might be used in further calculations.

```
282 D% calc 1 + (!*) + (-e myecho.c)
calc 1 + ( -D a*c ) + ( -e myecho.c )
2
```

Increment and Decrement Operators

The unary incrementing and decrementing operators are ''++'' and ''`--`''. Pasting one in front of a variable name bumps the variable, then returns the value. Pasting one after the name bumps the variable but returns the prior state.

```
283 D% calc x = 1
1
284 D% calc ++x
2
285 D% calc x++
2
286 D% calc x
3
287 D% calc --x
2
```

Bit Shifting

The ''<<'' and ''>>'' bit shifting operators shift an integer value on the left by the number of bit positions given by the integer value on the right. Bits shifted off the end are lost; values shifted in are always `0`.

```
288 D% calc x << 3
8
289 D% calc x >> 10
0
```

Expressions

Bit Not Operation

The unary "~" operator returns the bit-wise not of an integer operand. As this example shows, integers are 32-bit signed values.

```
290 D% calc ~5
-6
291 D% calc 0xfffffffa
-6
292 D% calc ~!$
calc ~0xfffffffa
5
```

Logical Not

The unary "`!`" operator returns the logical not. If the operand is non-zero, `0` is returned, otherwise `1`. In this example, the parentheses or space after the exclamation are deliberate to avoid having the expression confused as a history reference.

```
293 D% calc !(5.1)
0
294 D% calc ! 0
1
```

Exponentiation

The "`**`" operator is for exponentiation. The left operand is raised to power of the right operand.

```
295 D% calc 2 ** 500
3.27339061e+150
```

Modulo and Integer Division

The "%" operator is for modulo division and returns the remainder.

```
296 D% calc 22 % 7
1
```

A related "`//`" operator does integer division. Where the standard "`/`" operator might return a floating point result, "`//`" gives just the integer part of any division.

```
297 D% calc 8/3
2.666667
298 D% calc 8//3
2
```

Comparison Operators

The ``==`` operator tests for equality; the single ``=`` means assignment. The ``!=``, ``<``, ``<=``, ``>=``, and ``>`` operators are all straight-forward tests of ``not equal,`` ``less than,`` ``less than or equal,`` etc. Comparisons of strings are as easy as of numbers.

```
299 D% calc x = 3
3
300 D% calc x == 5
0
301 D% calc "able" < "baker"
1
```

When the shell is asked to compare two expressions, it first tries to coerce them to numeric values. This is so that, e.g., a string containing ``16`` compares greater than ``2`` even though a simple string compare would give the opposite result.

Pattern Matching Operators

The ``=~``and ``!~`` are the ``pattern matches`` and ``pattern fails`` tests. These are done in pretty much the same way wildcarding is done. On the right is a pattern string possibly containing wildcard characters. It's compared against the string on the left the same way a wildcard expansion would be done except that here, comparisons *are* case-sensitive and where alternation appears, the match succeeds if any of the alternates matches.

```
302 D% calc "Now is" =~ "N*i*"
1
303 D% calc "Now is" !~ "Now is"
0
304 D% calc "Now is" =~ "n*i*"
0
305 D% calc "Now is" =~ "{n,No}*i{s,the}"
1
```

Bitwise And, Xor and Or Operators

The ``&``, ``^`` and ``|`` operators perform bit-wise and, xor and or operations on integer operands.

```
306 D% calc 5 & 4
4
307 D% calc 5 ^ 3
6
308 D% calc 5 | 3
7
```

Logical And and Or

The ''&&'' and ''`||`'' operators perform logical and and or operations:

```
309 D% calc 5 && 4
1
310 D% calc 0 && 4
0
311 D% calc 5 || 3
1
312 D% calc 5 || 0
1
```

The ?: Operator

The ''`?:`'' trinary operator selects between two alternate expressions based on the logical (i.e., true or false) value of the first operand.

```
313 D% calc 0 ? "hello" : "goodby"
goodby
314 D% calc (5 > 3) ? "hit" : "miss"
hit
```

The {...} Operator

The ''{...}'' grouping operator allows you to run a command and evaluate its result as a 1 if it succeeds or a zero if it fails. For example:

```
315 D% calc {echo hello}
hello
1
316 D% calc {cd \nonexistent}
csh:  Couldn’t change the current directory to ’\nonexistent’.
0
```

The *Op*= Operators

Finally, the various ''*op*='' operators apply the *op* to the left and right operands, then assign the result to the left operand.

```
317 D% calc x = 2
2
318 D% calc x **= 500
3.27339061e+150
```

```
319 D% calc x
3.27339061e+150
```

Type Conversions

The shell always tries to evaluate expressions ‘‘sensibly’’ by doing any type conversions that might seem necessary. If an integer calculation results in an overflow, the shell shifts automatically to floating point.

```
320 D% calc 2**30
1073741824
321 D% calc 2**200
1.606938e+060
```

If a character string was given but an integer is needed, the shell tries to do that conversion also. Because these conversions happen automatically, without any fanfare, the following literals all compare equal:

```
27    27.0    033    0x1B    " 27 "    '   0x1b '
```

(Null strings and strings consisting only of white space are considered equal to zero. This is particularly convenient for local variables, which are initially set to null strings.)

The shell does automatic conversions to a character string format when the result is being printed. Numeric results are always shown in decimal. In this example, a procedure, the built-in square root routine, is invoked as a command; the value it returns is converted from floating point to character string and printed.

```
322 D% sqrt 2
1.414213
```

The shell also converts to a character string when you reference an array but use it as if it were not.

```
323 D% set x = Now is the time
324 D% cd ~\samples; myecho $x
'myecho' 'Now' 'is' 'the' 'time'
arg length = 23 characters
325 D% @ y = x
326 D% myecho $y
'myecho' 'Now is the time'
arg length = 23 characters
```

Expressions

Aliases

Aliases are a quick shorthand technique. If you type an alias at the beginning of a command, it's replaced by whatever the alias is defined as. They're intended to be used for relatively simple abbreviations: if any arguments are needed, you have to be able to type them onto the end. (More complex situations will have to wait until we cover procedures.)

To list the aliases currently defined, use the `alias` command:

```
327 D% alias
cdd          cd +c
copy         local s ; @ s = nowild ; @ nowild = 1 ; safecopy
date         dt
del          local s ; @ s = nowild ; @ nowild = 1 ; safedel
dir          cmd /c dir
erase        del
h            history
help         helpmsg
label        cmd /c label
ll           ls -L
md           mkdir
mi           more -i
rd           rmdir
ren          rename
rename       local s ; @ s = nowild ; @ nowild = 1 ; saferename
ro           rotd
start        cmd /c start
type         cat
vol          vl
xcopy        local s ; @ s = nowild ; @ nowild = 1 ; safexcopy
```

Some aliases are used to intercept references to `cmd.exe`'s built-in commands. For example, this is how `dir` is run. Other aliases give simple alternate names to a command, e.g., `rename` for `mv`. Still others are used to customize a command with a useful option. For example, `mi` runs `more` but starts it immediately in interactive mode, which means the screen is cleared first; in a PM text window, this tends to run faster.

To find out how any particular alias is defined, use the `alias` command with only the name you're interested in as an operand.

```
328 D% alias mi
mi           more -i
```

To create a new alias, type the `alias` command followed by the name of alias being created and word list it should be expanded into:

Aliases

```
329 D% alias hello echo hello world
330 D% hello
hello world
```

If you define an alias that refers to itself, either directly or via other aliases, the shell traps the reference rather than allowing it to expand without limit:

```
331 D% alias hello (echo infinite; hello again)
332 D% hello
csh:  A loop in the alias definitions was encountered and
trapped.
```

This raises the question how you might define an alias, say, `ls`, that intercepts references to the `ls` utility without getting into a loop. The answer is that the shell considers it a special case if the first word in the expansion of the alias is the same as its name. Here's an alias that causes `ls` to always display all files:

```
333 D% alias ls ls +a
```

Implementation Details

The alias mechanism is actually part of the parsing mechanism rather than a run-time feature of the C shell. What that means is that the alias expansion is done when the statement is first read, not when it's executed. Here's an example where we attempt to change the definition of an alias inside a loop. Notice that it doesn't have any effect until we exit the loop. That's because the whole loop is being compiled as a block before any part of it is executed.

```
334 D% alias foo echo this is life
335 D% foreach i (hello world)
336 D?    alias foo echo $i
337 D?    foo
338 D? end
this is life
this is life
339 D% foo
world
```

Programming Constructs

This chapter outlines the various structures provided for connecting statements together: describing serial relationships, conditional execution, iteration and how procedures are defined and used.

Serial Execution

As we've seen already, commands typed on successive lines are executed serially, one after the other. Writing several commands on one line with semicolons between them does the same thing.

```
340 D% echo hello; echo world
hello
world
341 D% _
```

Notice that in contrast to `cmd.exe`, the shell doesn't pass the semicolon to the application you invoke. If you really do want to pass a semicolon, e.g., to the linker to indicate the end of the arguments, you have to escape it or put it inside quotes.

A non-zero return code is not normally considered an error: regardless of the return code from any particular command, serial execution continues. We can demonstrate this with the `rcode` utility in the `samples` directory which prints, then exits with the return code value you pass it on the command line. This example also shows how you can retrieve the return code of the last child process by referring to the built-in `status` variable.

```
341 D% cd ~\samples
342 D% rcode 1; rcode 2
1
2
343 D% calc status
2
```

It's also possible to describe a conditional serial relationship. If statements are joined by "&&", the second one is executed only if the return code from the first one is 0, i.e., if the first statement succeeds. If statements are joined by "`||`", the second is executed only if the first one fails, i.e., returns a non-zero return code.

```
344 D% rcode 0 || rcode 1
0
345 D% rcode 1 || rcode 2
1
2
346 D% rcode 0 && rcode 1
0
```

```
1
347 D% rcode 1 && rcode 2
1
```

Statements and Statement Lists

I/O redirectors and statement connectors are recognized according to a precedence. Just as in expressions, where "*" is done before "+", statements are parsed so that some things are done before others. I/O redirection comes before piping which comes before conditional execution which comes before serializing with semicolons. For example:

```
348 D% echo hello; echo world | wc
hello
        1        1        7
```

The shell makes a special distinction between individual statements, no matter how complex, and lists of statements typed on separate lines or separated by semicolons.

Here's an example using the `time` command, which runs a statement and prints out the hours, minutes and seconds it took. `time` expects a single statement as a operand; if you type a semicolon, the `time` command (together with its operand) becomes just one statement in the list.

```
349 D% time echo hello world | wc
        1        2       13
0:00:00.50
350 D% time echo hello; echo world
hello
0:00:00.00
world
```

Parenthesis

There are two ways to group a list of statements together to make them act like a single statement. The simplest way is with parenthesis, which work the way they would in an expression: even if the operator inside the parentheses are of lower precedence, they're done first.

```
351 D% (echo hello; echo world) | wc
        2        2       14
352 D% time (echo hello; echo world)
hello
world
0:00:00.00
```

A parenthesized group gets its own copy of the current directory and disk. This makes it convenient to change directories inside the group and go do something without having to change back afterward.

```
353 D% cd
d:\doug\samples
354 D% (cd ..; cd)
d:\doug
355 D% cd
d:\doug\samples
```

The actual implementation uses the directory stack mechanism: at entry to the group, the current directory is pushed onto the directory stack and at exit, the top entry is popped.

```
356 D% dirs
d:\doug\samples
357 D% ( dirs )
d:\doug\samples
d:\doug\samples
358 D% dirs
d:\doug\samples
```

Control Structures

The more general way of connecting statements together is with control structures, which provide ways of describing conditional or iterative execution or even (with procedures) adding new vocabulary to the language. You can use a control structure anywhere a statement is allowed.

The language is completely recursive: control structures can be nested inside control structures, etc. A statement can be arbitrarily complex. Here's an example timing a statement that turns out to be a `for` loop piped to a `wc` and inside the `for` loop ...

```
359 D% time for i = 1 to 3 do
360 D?         time echo hello world | wc
361 D?      end | wc
        6       12      126
0:00:01.03
```

If Statement

The if statement comes in two forms. The short form is convenient if the choice is only between executing and not executing a single statement, which appears on the same line.

```
362 D% if (5 == 2 + 3) echo yes
yes
```

```
363 D% if (5 == 10) echo really
364 D% _
```

The longer form provides the more traditional if-then-else structure. Indentation is a matter of choice, it's used in these examples merely to improve readability.

```
364 D% if (5 == 10) then
365 D?    echo 5 == 10
366 D? else
367 D?    echo 5 is not 10
368 D? end
5 is not 10
369 D% _
```

Switch Statement

The `switch` statement works by attempting to pattern match the `switch` value against a series of alternative `cases`. The `switch` and `case` values can all be arbitrary expressions. If any pattern match succeeds, execution begins with the next statement following and continues, skipping over any interspersed `case` clauses until either the end of the `switch` block or a `break` statement is reached.

```
369 D% switch ("hello world")
370 D?    case 5:
371 D?       echo hit 5
372 D?    case "h*":
373 D?       echo hit "h*"
374 D?    case "x*":
375 D?       echo hit "x*"
376 D?       break
377 D?    case 43.2:
378 D?       echo hit 43.2
379 D?    default:
380 D?       echo did not hit
381 D? end
hit h*
hit x*
```

The `break` statement used here causes execution to ''break out of'' the innermost control structure. If you're nested several layers deep into control structures and want to break out of a higher level structure you can label the higher level structure and specify that name on the `break` statement.

Foreach Statement

The `foreach` statement is designed for iterating over a series of words. In this example, `i` is iterated over the list of all the files in the `samples` directory. Each one, in turn, is tested to see if it's executable (i.e., has a `.csh`, `.cmd`, `.exe` or `.com` extension.)

```
382 D% cd ~\samples
383 D% ls
args.c       dumpenv.c    finance.csh  myecho.exe   readme
args.exe     dumpenv.exe  makecpgm.csh rcode.c
bits.csh     factor.csh   myecho.c     rcode.exe
384 D% foreach i (*)
385 D?    if (-x $i) echo $i is executable
386 D? end
args.exe is executable
bits.csh is executable
dumpenv.exe is executable
factor.csh is executable
finance.csh is executable
makecpgm.csh is executable
myecho.exe is executable
rcode.exe is executable
```

For Statement

The `for` statement provides more traditional iteration over numerical values. If you specify a range (e.g., "`1 to 3`") but don't specify the increment, 1 is assumed. Although this example shows iteration over integer values, floating point values are equally acceptable.

```
387 D% for i = 1 to 3 do
388 D?    echo $i
389 D? end
1
2
3
```

You can also iterate over a list of ranges or individual values. The `to` and `by` clauses may be specified in either order.

```
390 D% for i = 1, 4, 7, 12, -4 to 6 by 3 do
391 D?    echo $i
392 D? end
1
4
7
12
-4
-1
```

```
2
5
```

While Statement

The `while` statement works in the traditional manner, iterating so long as the while condition is true. This example keeps popping up through the various levels of parent directories until it reaches the root. `fullpath` is one of the built-in procedures; it return the fully-qualified pathname of its argument. Notice that `fullpath` is invoked in three different ways: on line 384, as if it were a command, on 385 in more conventional procedure syntax and on 386, where it's substituted in as if it were a variable.

```
393 D% fullpath .
d:\doug\samples
394 D% while (fullpath(".") !~ "[a-zA-Z]:\")
395 D?    echo $fullpath(".")
396 D?    cd ..
397 D? end
d:\doug\samples
d:\doug
398 D% cd
d:\
```

Repeat Statement

The `repeat` statement has two forms. In the short form, a numeric constant (*not* an expression) specifies the number of times to execute the statement following on the same line.

```
399 D% repeat 4 echo do this again
do this again
do this again
do this again
do this again
```

In the long form, `repeat` provides the more conventional repeat structure, iterating until some exit condition satisfied.

```
400 D% calc i = 1
1
401 D% repeat
402 D?    calc i++
403 D? until (i > 5)
1
2
3
4
5
```

Procedures

Procedures, as in any high-level language, are a convenient way to package together a series of statements as a more convenient operation. Once you've defined a procedure, you can invoke it simply as if it were a new command.

```
404 D% proc hello()
405 D?    echo hello world
406 D? end
407 D% hello
hello world
```

The `proc` statement can also be used to ask what procedures are already defined or what arguments a particular procedure takes:

```
408 D% proc hello
hello        ( )
409 D% proc | mi
abs          ( x )
acos         ( x )
asin         ( x )
:
:
samepath     ( a, b )
sin          ( x )
sinh         ( x )
--- more --- (Press H for Help)
```

You can explicitly discard a definition with `unproc`; otherwise the shell remembers any procedure you tell it until you exit the shell or give it a new definition.

```
410 D% unproc hello
411 D% hello
csh:  Couldn't find an executable file named 'hello'.
```

When you give the shell a procedure definition, the shell compiles it into an internal form so that the next time you refer to it, it'll save the reparsing time and run much faster. As an example, `unproc` the `whereis` procedure to make the shell reload the definition from the `.csh` file and see what that does to the execution time:

```
412 D% unproc whereis
413 D% time whereis ls
f:\os2\bin\ls.exe
0:00:02.15
414 D% !!
time whereis ls
f:\os2\bin\ls.exe
0:00:01.28
```

The namespace for procedures is shared among all the threads: if one thread creates a new procedure, it becomes usable immediately by all the other threads.

Programming Constructs

Arguments

You can write a procedure so it expects arguments, just as you would in any other high level language. Argument names are somewhat like local variables: their initial values are set at entry to a procedure, hiding any previous definition; they go away as soon you exit the procedure code. Here’s a simple example which compares the timestamps on two files.

```
415 D% proc comparedates(a, b)
416 D?    if (‘newer $a $b‘) then
417 D?       echo $a is newer than $b
418 D?    else
419 D?       if (samepath(a, b)) then
420 D?          echo $a and $b are the same file!
421 D?       else
422 D?          echo $a is older than $b
423 D?       end
424 D?    end
425 D? end
426 D% comparedates ‘whereis more‘
c:\os2\bin\more.exe is newer than c:\os2\cmds\more.com
427 D% _
```

When you pass arguments to a procedure on the command line, the individual argument words are paired up, one-by-one, with the argument names you gave. If the shell runs out of names before it runs out of words, the last named argument gets all the remaining words:

```
427 D% proc xx(a, b)
428 D?   echo $#a $a
429 D?   echo $#b $b
430 D? end
431 D% xx now is the time
1 now
3 is the time
```

If you pass arguments to a procedure that doesn’t take any, they’re evaluated but quietly ignored.

If a procedure does take an argument, it always get some value, even if it’s zero words long. So if you want to know if you got passed a value, just count the number of words:

```
432 D% proc xx(a)
433 D?    echo $#a ">>$a<<"
434 D?    if (a == "") echo null argument!
435 D? end
436 D% xx
0 >><<
null argument!
```

In a more serious vein, here's a simple procedure definition I use all the time (I have it in my `startup.csh` file) to implement a real quick and dirty (but very easy to use!) personal phone index:

```
437 D% proc ppi(name)
438 D?    grep -i "$name" h:\phone
439 D? end
440 D% ppi hamilton
Hamilton Laboratories  508-358-5715  Fax: 508-358-1113
```

As you add lines to your `\phone` file, you merely add any interesting search phrases or other tidbits onto the same line with the person's name. Totally free format. Add anything you like and search on anything you like and it's fast.

Return Values

Procedures are also important in expressions, where it's generally useful to think of the procedure as returning a value, just as it might in any other language. The type and value of what you choose to return is arbitrary. Here's a purely mathematical example from `finance.csh` in the `samples` directory:

```
441 D% proc FV_PresentAmount(i, n)
442 D?    # Calculate the multiplier to convert $1 now to a
443 D?    #    future value, given interest rate i
444 D?     return 1/(1 + i/100)**n
445 D? end
446 D% # Calculate the future value of $500 invested
447 D% # for 10 years at 8% interest.
448 D% calc 500*FV_PresentAmount(8, 10)
1079.462499
```

If you call a procedure that returns a value as if it were a command, whatever it returns is printed:

```
449 D% FV_PresentAmount 8 10
2.158925
```

Recursion

A procedure can call other procedures or even itself. When a procedure calls itself, it's called recursion. Typical uses of recursion are in cases where the problem itself is recursive, or self-replicating. For example, here's a procedure to walk down two directory trees A and B that are thought to be related and list any non-hidden files in A that are not in B. (If you set `nonohidden = 1`, it'll compare hidden files also.)

```
450 D% proc comparetrees(a, b)
451 D?    local i, f
452 D?    foreach i ($a\*)
453 D?       @ f = $i:t
454 D?       if (! -e $b\$f) then
455 D?          echo $b\$f is missing
456 D?       else
457 D?          if (-d $i) comparetrees $i $b\$f
458 D?       end
459 D?    end
460 D? end
461 D% comparetrees c:\src\projectx a:\src
```

Notice that `i` and `f` were declared as `local` variables. If the variables were simply `set` variables, one instance of them would be shared by all the levels of recursion. In this particular example, that would still have worked, but only because each level calls the next only *after* anything involving `f` or `i` has been evaluated; it wouldn't matter if `f` or `i` was trampled by the next call. Here's an example where obviously that would *not* be true: a clumsy attempt at a ''post-order'' traversal of a directory tree:

```
462 D% proc traverse(a)   # Don't do it this way
463 D?    foreach i ($a\*)
464 D?      if (-d $i) traverse $i
465 D?      echo $i
466 D?    end
467 D? end
468 D% traverse . | more
```

If you carefully examine the output of this `traverse`, you'll see that subdirectories don't get listed properly: instead of being listed by themselves, the name of their last child is listed twice. For a correct result, try it again with `i` defined as a `local` variable. (Use the <PageUp> key to help you quickly re-enter the lines that stay the same.)

Calling a Procedure

As you may have spotted, there are two ways to invoke a procedure. Sometimes, the arguments are inside parentheses, separated by commas, and sometimes they're not. What's the difference?

The difference is whether the context is an expression or a command. As discussed when we first introduced expressions, the shell always begins to parse statements by first breaking them up into words. That's fine for normal commands, e.g., running an external utility. And it works also when you want to use a procedure as if it were a command, just typing the name of the procedure followed by a list of arguments separated by spaces, e.g.,

```
469 D% proc power(a, b)
470 D?    return a**b
471 D? end
```

```
472 D% power 2 3
8
473 D% _
```

But this style of parsing wouldn't be very suitable in those instances where the point is to do some kind of calculation or expression evaluation. So when the shell encounters something that normally takes an expression, e.g., following the `calc` keyword, or inside the test in an `if` statement, it shifts to a different style of parsing, further breaking up the words into tokens, so that ''`*`'' isn't misunderstood as a wildcard, so we don't need to type spaces around all the operators, so we can type variable names without having to put a ''`$`'' in front of them and so on. All of this is so that the rules for typing an expression can bear some resemblance to those followed by other programming languages like C, Fortran, Pascal, etc.

When we call a procedure from within an expression, all these same arguments still apply. We want it to act pretty much like any other high level languages. We want to be able to pass it arbitrarily complex expressions as arguments. We want to be able to take the value it returns and use that value as a term in still other expressions.

So there's a real problem: to call a procedure from within an expression and pass other expressions as arguments, we need a way of separating one argument from the next (obviously, it can't be just a space as it would be when the procedure is used as if it were a command) and for separating the whole procedure call and its arguments from the rest of the expression. That's why the common high-level language convention of separating arguments by commas and putting parentheses around the whole list is used. Here's an example of what that looks like:

```
473 D% calc 5.5 + power(2, 3)*9
77.500000
```

If you try using a procedure as a command but accidentally type the argument list with parenthesis, it's an error:

```
474 D% power(2, 3)
csh(line 470):  Couldn't evaluate expression operands as
numeric as required by the expression operator.
> in power( "(", "2,", "3", ")" ) defined at line 469
< called from line 474
```

The reason this is an error is because, since this was typed as a command, the shell took the words following the word power as literal arguments. It couldn't tell you meant this as an expression. Let's redefine that procedure, putting some `echo` statements in there so we can see what happened:

```
475 D% proc power(a, b)
476 D?   echo a is $a
477 D?   echo b is $b
478 D?   return a**b
479 D? end
480 D% power(2, 3)
a is (
b is 2, 3 )
```

```
csh(line 478):  Couldn’t evaluate expression operands as
numeric as required by the expression operator.
> in power( "(", "2,", "3", ")" ) defined at line 467
< called from line 480
```

As you can see, the expression “`a**b`” failed to evaluate properly because `a` was set to the first argument word, “(”, and `b` was set to a string concatenation of all the rest of the words. Neither was a number.

If you want to call a procedure and substitute the value back onto the command line even when the context is not an expression, it can be done, however. One way is with command substitution:

```
481 D% echo ‘power 2 3‘
a is 2 b is 3 8
```

This is a bit expensive, though, because the shell will have to create a new thread to run the `power` procedure and set up a pipe to read the result. And as you see, if the procedure also writes to stdout, you’ll pick up that text also, probably unintentially. Another, better way, is to use a dollar sign to introduce the substitution just as if it was a variable substitution:

```
482 D% echo $power(2, 3)
a is 2
b is 3
8
```

Notice that when use the dollar sign-style procedure reference, the rest of the syntax is as if the procedure had been called from within an expression. The arguments do need to be within parenthesis and they do need to be separated by commas. The reason is just the same one as for why a procedure call in an expression has to be done this way: without the parentheses, there’d be no way to tell where the arguments ended. A nice benefit is that in the argument list, we get to use the full expression grammar:

```
483 D% echo $power(2, 3*sin(1/2))
a is 2
b is 1.438277
2.709970
```

Shell Scripts

Scripts are a final way of bundling up a series of statements to be called up and executed as a single command. To create a script, create a file with a `.csh` extension:

```
484 D% cat >trythis.csh
echo hello from trythis
^Z
485 D% trythis
hello from trythis
```

When you tell the shell to run a script, it first creates a new thread to run it. This is partly a holdover from original UNIX language definition, partly a response to a provision in OS/2 and NT for threads, but not a fork mechanism and partly due to a genuine need to inexpensively separate some of the script's environment from that of its caller. (The next chapter has a longer discussion of threads.)

Shell Script Arguments

Arguments to a shell script are passed to it as the `argv` variable. `argv` will be a list of any words that appeared on the command line *following* the name of the shell script. (You can access the name of the script as the `scriptname` variable.) You can access `argv` like any other variable:

```
486 D% cat >tryargv.csh
echo $#argv $argv
^Z
487 D% tryargv hello how are you
4 hello how are you
```

There are also some shorthand forms for getting individual words of argv. `$0` through `$9` is the same as `$argv[0]` through `$argv[9]`. (Remember that unless you have `nullwords` set, subscripting errors will be caught.)

ignorestatus

If you write a script with serially connected statements the only thing that would cause the shell to quit before it gets to the end would be an explicit failure: an application name that couldn't be found, a child process that terminated with a segment fault, or something else of an equally serious nature. Often in a script, that's not what you want: you've written the script with the expectation that everything will work (as you planned) from one step to the next. If something is wrong, you'd like the script to quit as soon as possible, before any damage is done.

The way you do this is by setting `ignorestatus` = 0, which means you do not want to ignore the status codes coming back to this thread from its children. Here's an example in the main thread:

```
488 D% set ignorestatus = 0
489 D% rcode 10
10
csh:  The child process running 'rcode' exited with a non-zero
status = 10.
```

In the main thread, the shell will keep on going and prompt for the next command because interactively that's most sensible. The shell knows to do this because `ignoreerrors` = 1. But in a script, errors cause the shell to quit:

```
490 D% cat >trythis.csh
calc ignoreerrors
set ignorestatus = 0
rcode 10
echo does^'t print
^Z
491 D% trythis
0
10
csh(d:\doug\trythis.csh:line 3): The child process running
'rcode' exited with a non-zero status = 10.
> in d:\doug\trythis.csh
< called from line 491
csh:  The csh script file 'd:\doug\samples\trythis.csh' exited
with a non-zero status = 10.
```

Notice that in this case we got two messages, one from the threads executing the script and one from the main thread, reporting what the script returned. Let's return to the normal mode of ignoring status:

```
492 D% set ignorestatus = 1
```

source statement

The examples so far have shown how a script is normally run somewhat isolated in a separate thread. It is also possible to run a script in your current thread using the source statement. You might want to do this if you wanted to the script to change your current thread's private variables or its current directories or disk. Here's an example to showing how a sourced script runs in the same thread:

```
493 D% cat >trythis.csh
echo argv = $argv, threadid = $threadid
^Z
494 D% echo $threadid
6
495 D% trythis hello world
argv = hello world, threadid = 7
496 D% source trythis hello world
argv = hello world, threadid = 6
498 D% _
```

Notice how the `argv` argument vector is set up the same in either case. Also, notice that the statement number skipped by one. When you `source` a script, the effect is precisely as if you typed those lines in directly to the shell. The lines read by `source` are even entered into the history list:

```
498 D% h 5
    494  echo $threadid
```

```
495  trythis hello world
496  source trythis hello world
497  echo argv = $argv, threadid = $threadid
498  h 5
```

Caution: Labels and Gotos

We haven't mentioned labels and gotos yet but it probably isn't a surprise that the C shell allows them. Indeed:

```
499 D% cat >trythis.csh
goto next
echo this does not print
next: echo this prints
^Z
500 D% trythis
this prints
```

If you want to use gotos to labels, you should be aware that forward references can be little trickier than a more conventional compiled language. The C shell allows you to redefine a label anytime you like. But if you type a `goto` that refers to previously defined label, the shell has no way of knowing that you intend it to redefine it up ahead. You can keep running the last example over and over this way with exactly the same result: because a new thread is started each time with no prior definition of `next`, the shell knows it must be a forward reference. But imagine how repeatedly sourcing this script would fail in an infinite loop:

```
% source trythis
this prints
% source trythis
this prints
this prints
this prints
this prints
this prints
:
```

(Beware of actually trying this: you may find it difficult to interrupt out of it.)

The reason sourcing the script a second time turns into an infinite loop is that the label `next` is already defined after the first run. The second time, when the `goto` is read from the script, the history list would look something like this:

```
source trythis
goto next
echo does not print
next: echo this prints
```

```
source trythis
goto next
```

What particularly gets the shell into a muddle is the way this recurses indefinitely: each time through the loop, it recurses through an another level of sourcing. Ultimately, it runs out of stack space and fails. This is not a nice way to treat the shell!

In general, it's hard to recommend gotos in any programming language nowadays; in a script you intend to run using `source`, they can be particularly nasty.

The shell does automatically age labels and throw them away after a while even if they haven't been redefined. When it discards a label, it also discards any compiled statements it's been holding onto that could have been executed only by a `goto` to that label. The cutoff point where the shell begins to discard labels is set by the `gotowindow` variable. Let's now clean up after ourselves and move along:

```
501 D% rm trythis.csh
```

Interrupts

Normally, when you type `^C`, you interrupt the foreground activity. But what if you were in the midst of a complex script and needed to do some kind of cleanup before you exited? What if you wanted to be sure you had a chance to delete any temporary files you might have littered around?

The solution is the `onintr` statement, which allows you to define the action to be taken when an interrupt is received. It causes whatever's running to be interrupted all the way back up to the block in which the `onintr` routine was defined and for the interrupt routine to be run in that current thread. Within that interrupt routine, you could, for example, remove all your temporary files and `goto` the end of the script or return a special value from a procedure or whatever else might be appropriate.

```
502 D% onintr echo hello
503 D% for i = 1 to 5 do
504 D?    echo $i
505 D?    sleep 1
506 D? end
1
^C
hello
```

Here's another example, returning from a procedure. Note how the value returned (and printed) is the one produced by the `onintr` statement.

```
507 D% proc foobar()
508 D?    onintr return 5
509 D?    for i = 1 to 5 do
510 D?        echo $i
511 D?        sleep 1
```

```
512 D?    end
513 D? return 2
514 D? end
515 D% foobar
1
^C
5
```

When execution leaves the block in which an `onintr` is defined, the previous `onintr` (if any) again takes effect. Note that a null `onintr` routine does not mean that interrupts are ignored, merely that after processing bubbles back up to the level where that `onintr` was defined, that it will continue with the next statement. Notice how, in this example, when the `^C` is received when obviously execution is stuck in the infinite loop inside bar, that the ''`onintr goto xx`'' causes a branch to `xx` in the same block in which the `onintr` was defined, not the `xx` in the block where execution was going on. Also, notice that once both procedures have been exited, we're back to the same `onintr` routine we defined a few statements earlier.

```
516 D% proc foo()
517 D?    onintr goto xx
518 D?    bar
519 D?   xx:  echo this is foo
520 D? end
521 D% proc bar()
522 D?    while (1)   # Deliberately infinite loop
523 D?    end
524 D?   xx:  echo this is bar
525 D? end
526 D% foo
^C
this is foo
527 D% ^C
hello
527 D% _
```

Masking Interrupts

In cases where you'd like to simply turn off interrupts or defer processing them, use the `irqmask` variable. By default, it's set to `0`, meaning interrupts will be accepted immediately. Setting it to `1` means interrupts will be deferred until the mask is cleared again. Setting it to `2` means interrupts will be totally ignored.

`irqmask` is a per-thread variable, meaning each thread can independently decide how it will respond to interrupts. Each new thread always starts out with `irqmask = 0` (interrupts enabled).

Programming Constructs

Scheduling

Foreground Activities

Whenever you type any command, the shell's normal behavior is to start up that child activity and then go to sleep waiting for it complete. This is a foreground activity. If you start something and then decide you want to stop it, type Control-C, which wakes up the shell and causes it to stop the foreground activities.

Under OS/2, Hamilton C shell can tell the difference[†] between full-screen, text-windowable and full Presentation Manager applications and properly spawns new screen groups for applications that can't run in the shell's window. For example, typing the name of the Presentation Manager control panel:

```
527 D% pmcpl
```

will (or at least, should) cause the control panel to appear in a new window.

Similarly, under Windows NT, if you type the name of a graphical application, the C shell will recognize that based on an actual examination of the file itself.

Under both OS/2 and Windows NT, when you start an application that needs a new window, the shell comes right back for the next command.

Also, if the child is a a graphical application and `stdout` or `stderr` is tied to the console window where the C shell's running, the C shell recognizes that that handle won't be inheritable by the child and instead will create a pipe. It'll give the write end to the child and create a background thread in the C shell to read anything coming back over the pipe from the child and copy it to the console window. This means that with the C shell, you can still use ordinary `printf`'s in a OS/2 PM or NT graphical application and not lose any output.

Background Activities

If you want to run something but don't want to wait for it complete, just type an ampersand at the end of the statement:

[†] Actually, under OS/2, the shell depends on the `.exe` file being properly marked with the application type. Unfortunately, not all developers yet know about and obey this rule. For example, in some releases of OS/2, even IBM forgot to mark some applications such as the control panel. Unmarked applications are, by default, started full-screen; a PM application that's started this way dies immediately. If you spot this behavior, you should mark the offending application using the `markexe` utility. (Type "`markexe -h`" for more information.)

```
528 D% wc hello.c >linecnt &
1
529 D% _
```

This creates a child process to run word count in the background, with its output directed to a file. The “`1`” message means that a new background job (job 1) has been spawned to process the command while you continue to work. The job starts as a new thread. If, as in this case, a separate process is needed, that thread will create it with a `DosExecPgm` call to the OS/2 kernel (or `CreateProcess` call to the Windows NT kernel, as appropriate), then exit. Each new thread, process or screen group spawned by a background job will inherit its parent's job number. Every time a new background job is created, the job number is incremented.

The use of i/o redirection in combination with a background activity is not accidental. If it's not redirected, it goes to your terminal, intermixing with the output of any foreground activities. Occasionally, that might be exactly what you want. For example, here's a timer to wake you up in 5 minutes (300 seconds):

```
529 D% (sleep 300; echo ^aWake Up, Sleepyhead.) &
2
:
:
Beep
Wake Up, Sleepyhead.
```

The ampersand works consistently for things that need a new window:

```
530 D% pmcpl &
3
531 D% _
```

A new job starts up and announces itself, then realizes that the control panel has to be run in a separate session. Once it's started the child session, the thread exits and its children are adopted by its parent thread and child is left running as job 3.

Under both OS/2 and Windows NT, background activities are, in a sense, detached: typing Ctrl-C doesn't interrupt them (unless they explicitly ask to be notified.) You can start a large number of background activities and check on their status using the `ps` (process status) command. Here's what you'd see on OS/2; the output under NT would be fairly similar.

```
531 D% ps
Job     ID      Parent      State         Activity
-       t1      -           running       interrupt_handler
-       t2      t1          running       thread_cleanup
-       t3      t1          running       process_cleanup
-       t4      t1          running       screen_cleanup
-       t5      t1          running       keyboard_reader
-       t6      t1          running       main_thread
-       t7      t1          running       console_io
3       s33     t6          running       pmcpl
```

Threads 1 through 4 always run in the background. Each spends most of the time asleep, waking up to do some housekeeping only when an interrupt or the signal that a child activity has completed is received. Thread 5 is dedicated to reading characters from the keyboard on request from other threads. Thread 6 is the foreground command processor. Thread 7 was spawned when the C shell realized that pmcpl is a graphical application and that a pipe would be needed to capture any `stdio` output from the child. Other entries in the `ps` report come and go as you type commands.

If you want to explicitly terminate a background activity, use the `kill` command. But do keep in mind that under NT, there are two ways to kill a process: If it's a console (text window) application, it can be done by sending it a Ctrl-C signal; that's what kill does by default. But if it's a graphical application, it can only be done using the TerminateProcess call, a very brute force way of killing something; any DLL's that were being used by that process will not any notification that the process has died and, thus, will not know to do any cleanup they might normally do.

```
532 D% kill 3
533 D% ps
Job     ID      Parent      State         Activity
-       t1      -           running       interrupt_handler
-       t2      t1          running       thread_cleanup
-       t3      t1          running       process_cleanup
-       t4      t1          running       screen_cleanup
-       t5      t1          running       keyboard_reader
-       t6      t1          running       main_thread
-       t7      t1          running       console_io
```

Variables and Threads

User-defined variables are shared between all active threads unless they're declared as `local`: if one changes a variable's value, the other threads see that change immediately. Because the individual threads run asynchronously, this can cause some surprising results. In this example, the foreground thread spawns new background threads and increments the variable `i` faster than the children can execute. By the time any the children actually start, the loop has finished and every thread sees `i` as having the value 5.

```
534 D% for i = 1 to 3 do
535 D?    echo $i &
536 D? end
4
5
6
537 D% 5
5
5
```

Scheduling

One solution is to use the `eval` statement. `eval` parses the text it's passed at run-time, *after* any variable substitutions have been done. Because the ampersand is inside the quotes, its special meaning isn't detected until run-time.

```
537 D% for i = 1 to 3 do
538 D?    eval "echo $i &"
539 D? end
7
8
9
540 D% 1
2
3
```

A better solution is to make `i` a local variable, since locals are snapshotted and copied when the child is spawned:

```
540 D% local i;  for i = 1 to 3 do
541 D%    echo $i &
542 D% end
10
11
12
543 D% 1
2
3
```

Re-entrancy

Threads also introduce the possibility of re-entrancy. In the next example, we define a procedure for summing all the integers `1` to `n`. Notice that it works fine if it's run by itself, but gives the wrong answers if two threads try to run it simultaneously:

```
543 D% proc sum(n)
544 D?    @ s = 0
545 D?    for i = 1 to n do
546 D?       @ s += i
547 D?    end
548 D? end
549 D% sum 100
5050
550 D% sum 100 &; sum 100
13
6780
551 D% 7177
```

Here also, the solution is simply to include a statement defining `i` and `s` as `local` inside the procedure.

Threads: An advanced discussion

In building Hamilton C shell, a conscious and fundamental decision was made to use threads in many situations where earlier shells might have created separate processes. The result is a dramatically more responsive tool albeit one with some subtle semantic differences from the original.

The UNIX C shell language definition called for individual stages of a pipeline, command substitutions and scripts each to be run in a separate process cloned by *forking* the main process. Using forking, the child inherited all of its parent's state (current directory, open file handles, environmental and set variables, etc.) but any changes it made only affected itself. On a UNIX system with paging hardware and the fork mechanism built into the kernel, it's pretty fast.

But OS/2 and the NT Win32 api's do not have fork†, and trying to recreate precisely this language semantic under OS/2 or NT would have been foolishly expensive, potentially adding several seconds to the startup time each time you invoked a shell script. On the other hand, these systems do offer *threads*. A process can have lots of threads and each one can run along at its own pace. When a thread calls the kernel to do something that takes a long time (e.g., a disk read), it goes to sleep and doesn't wake up until the data's ready. When one thread goes to sleep, the kernel looks around for another that's ready to run. By using threads, it's possible to ensure that if one thing's got to wait, that won't hold up everything else.

Threads turn out to be even faster than a fork (regardless of the hardware), because the amount of state information associated with a thread is so little compared to that of a process. As viewed by the kernel, a thread ''owns'' only a register set, a stack and an instruction pointer. Everything else, memory, current directories, etc., is shared among all the threads in a process. This means creating a thread is very fast, as is switching between threads.

On the other hand, using threads to best advantage imposed some significant design challenges in Hamilton C shell. Certainly, for example, few would consider it acceptable if a script running in the background could >*Boom*< change your foreground current disk! The problem was to create a way for threads to cooperatively share the process resources but without giving away all the performance advantage we'd started with by using threads. Also, some of the elegance of threads is the idea you can keep creating new ones. Each is just like the next: any given thread can run just as complex a program as the next and each can spawn new threads. It would be a shame to lose that recursive characteristic by clumsiness in the language design.

Starting with a clean sheet of paper, our solution was a *highly* multi-threaded architecture. It expects you to start lots of threads: stages in a pipe, background activities, etc. To our knowledge, no other command processor on any system employs this technology. Certainly, all the code in Hamilton C shell is re-entrant: there is a minimum of global, statically-allocated data; the few variables that are global tend to be pointers to the roots of various dynamically-allocated information trees for managing variables, threads,

---

† The decision not to provide a fork semantic under OS/2 was probably forced by the decision that initial releases of OS/2 would run on 286-based machines. Lacking paging hardware, a fork on a 286 would likely have been unacceptably slow, no matter how the software was designed.

processes, file handles and other resources. When the shell creates a new thread, it creates the appropriate records and links them in. Some characteristics given the new thread are inherited from it's parent and some always get set to specific defaults.

Shared variables and other resources are semaphored: before using a resource, a thread requests it; if several resources are needed simultaneously, they're always requested in the same order to avoid deadlocks. Critical resources are held only for short periods. There's no polling anywhere. ''Handle'' mechanisms are used so that, e.g., a thread can decide if its current disk and directories are set up by simply comparing an integer. Path hashing structures are shared with a ''copy on write'' mechanism in case they change directories and need slightly different hash structures. Any thread can do what any other can: compile or execute an arbitrarily complex C shell program or even spawn or pipe child threads.

Given the enormous advantage offered by threads and the unique technology we've developed to exploit them, we expect Hamilton C shell should easily outperform any UNIX shell on comparable hardware.

Order of Evaluation

Finally, to put everything in perspective, here's a summary of roughly the procedure by which the C shell reads, parses and evalutes your commands:

1. The command is read. If `stdin` appears to be a keyboard, the command line editing routines are used to read a keystroke at a time, entering them into the command buffer and doing whatever editing is indicated. Otherwise, the shell simply uses the kernel's `DosRead` function to read small chunks until the end of the statement has been found.

2. History substitution is done. The "`!`" and "`%`"-style history references are expanded.

3. The text is broken up into separate words. Unless it's part of a quoted string, white space (tabs and spaces) separates words. Also, these special strings are interpreted as separate words even if they're run together with other text:

   `&    |    ;    >    <    (    )    &&    ||    >>    <<    >&    >!    >&!`

4. The command is added to the history list. The fact that this is done after the text has been broken up into separate words explains why the commands in the history list will look a bit different than the way you typed them. It's done this way on purpose so that you can refer to individual words in previous commands, e.g., with "`!$`" to get just the last word of the last command.

5. The command is compiled into an internal form using a recursive descent parser, recognizing the language constructs and whether a given portion of a command is really an expression or just a series of words.

   Compilation at this stage is at the level of a whole construct, e.g., a whole `foreach` statement or `proc` definition and everything inside it. That's so that every time through a loop or every time a procedure is run, the shell won't waste time recompiling statements that could have been compiled the first time. Also, aliases are expanded at this stage and some minor optimizations are down, e.g., pre-compiling static patterns appearing in pattern-matching expressions, etc.

6. The internal form is executed. The various quoting and other substitution activities are done, in effect, in this order:

   a. Threads are spawned for separate stages of a pipeline or for background execution. That's to avoid serializing any blocking events as, for example, the shell hits the disk, looking through the search path for executable files, etc. By spawning separate threads, those blocking events can be overlapped.

   b. I/O redirection is performed. If the filename being redirected to/from is actually a wildcard or a command or some kind of substitution, that word will be expanded.

c. Single and double quoted strings are processed. If the quoted string contains any wildcard characters, they're escaped so that they'll appear as literal characters when wildcarding is done but still be unescaped right after that.

   In the compiled internal form, double-quoted strings containing variable or command substitutions are already specially broken up to look, at this stage, like a series double-quoted strings and substitutions concatenated together.

d. Variable and command substitutions are done.

e. Wildcarding is done.

f. Escape characters are processed.

g. The series of words is passed to the command as arguments. (It's at this point, if it's an eval command, that the argument text is passed back through the parser and then to the evaluation logic)

   Commands are searched for in this order:

   (1) User-defined procedures.

   (2) Built-in procedures and commands.

   (3) External commands, searched for in the `PATH` directories in this order within each directory:

   ```
   .csh   .exe   .com   .cmd   .bat
   ```

   (`.bat` files can be run only under OS/2 2.x or Windows NT.)

7. The internal form of each compiled statement is discarded once it's no longer needed, i.e., if there's no way you might invoke that code from a later statement.

   For example, once you define a procedure, it's always accessible; you can call it at any time, so that compiled code is never discarded unless you redefine the procedure or explicitly `unproc` it. But an ordinary statement typed at the command line could be re-run (without re-entering it using history or by retyping it) only if it was part of a larger control structure or if there was a way to `goto` it, meaning there would have to have been a label preceding it.

Customizing the Shell

Depending on your tastes and what you're used to (and whether you come from a DOS or a UNIX background), Hamilton C shell can be customized in a number of ways. This section will outline:

1. Various options you have when installing the shell, including setting it up as the default command processor,
2. How shell starts up, and how it uses the `login.csh` and `startup.csh` files,
3. How to set the environmental variables, screen colors and the prompt strings,
4. How to create aliases you always want available,
5. Customizing the `cd` command and how history references work, and
6. Telling the shell how it should respond to certain special situations.

Installation Options

You have a fair amount of choice about where the various files associated with Hamilton C shell go and also about how it starts up. You won't necessarily want to fool with all this when you first install the shell, but you may want to look at some of this as you get a sense of your own preferences.

The C shell `.exe` and `.csh` files can go anywhere as long as they're on your search path. Most people find it most convenient to keep them in a separate directory. We do suggest it be ahead of the directory containing the standard `more.com` on your search path since we provide a new, vastly improved `more`.

You choose any directory on any drive as your "home" directory. The significance of a home directory is that the shell will make references to it or relative to it very convenient using the "~" wildcard character. Also, this is where the shell will look for your `login.csh` and `startup.csh` files. If `HOME` isn't defined in the environment when the shell starts up, it defaults to the current directory. It's usually most convenient to define the `HOME` variable in your `config.sys` but if you prefer, you can pass it to the shell on the command line using "`-C setenv HOME=`..." (The "`-C`" means what follows is a command.)

Additional customization is usually done with the `login.csh` and `startup.csh` files.

Installing as the Default OS/2 Command Processor

OS/2 gives whatever default command processor you specify on the `PROTSHELL` line of your `config.sys` special privileges to change its own title bar and, under OS/2 1.3 or 2.x, its own icon. This can be quite useful if you have a lot of copies of the shell running minimized and you'd like to know what they're doing.

Under OS/2 1.1 or 1.2, the C shell uses the undocumented `DosSMSetTitle` kernel api entry to change its own title; under 1.3 or 2.x, it uses the new `WinSetTitleAndIcon` api to change both the title and icon. Unfortunately, these api entries are disabled if you're not running in a window whose root process was the default command processor.

To install the C shell as the default command processor, follow the instructions given in the installation section of this manual. In a nutshell, the idea is to replace the reference to `cmd.exe` and any parameters to `cmd.exe` on the `PROTSHELL` or `OS2_SHELL` (under 2.x) line with the full pathname of the C shell plus a ''`-L`'' option to indicate it's a login shell.

Next, fixup the entries for `cmd.exe` and the C shell in the Group-Main or Start Programs menus. To make it a default command processor, the C shell must be listed with an ''*'' (asterisk) for a pathname (on the ''`Path and file name:`'' line if you pull-down ''`Program`'' and select ''`Properties...`' in Group-Main) or else the title bar and icon changes will be disabled. Also, for some reason, if you specify any startup parameters (other than ''`/K "%*"`''), that also disables the title and icon changes. (The ''`/K "%*"`'' option has special significance for the default command processor and is used when OS/2 boots up if it finds a `startup.cmd` file in the root directory; otherwise, it's ignored.)

Conversely, you have to change the asterisk path and any arguments for the `cmd.exe` entries to the actual path for `cmd.exe` and enter whatever arguments (if any) you had been specifying for it on the `PROTSHELL` or `OS2_SHELL` line in your `config.sys`.

login.csh

`login.csh` is read only by copies of `csh.exe` started with the ''`-L`'' option to make them login or root shells. Typically, this is a new window. `startup.csh` is read by each new copy of the shell, even if it's invoked as a child of the C shell.

The main objective of the `login.csh` file is to let you set up those characteristics that are inheritable from parent to child process but which might not be set up if you're starting from the Start Programs menu (OS/2 1.1), a Group menu (OS/2 1.2 or 1.3),from the desktop (OS/2 2.x) or from the Program Manager (Windows NT). In addition, it lets you do a little special customization of these highest-level shells.

The important inheritable characteristics are the environmental variables together with the current directories and current disk settings.

The Environmental Variables

Most users prefer to create the bulk of their environmental variables with `SET` statements in `config.sys` under OS/2 or through the Control Panel under Windows NT (to be inherited by every process on the system) rather than in `login.csh`. It means just one copy of the definitions in one place, so it's easier to make changes. It also runs faster since it's all set up when the shell starts; the shell doesn't have read all those commands. But if you're using an initialization script to set some of your environmentals for `cmd.exe`, you'll want to do that for the C shell also in `login.csh`.

But there are cases where it's more convenient to set up some environmental in your `login.csh`. For example, you might prefer to set your `PATH` statement there. Since the C shell hashes your search path, you'll find it's much faster finding things even with a very long list of path directories. (If you actually had 50 or so directories, the difference in startup time for something in that last directory would be around 6 seconds or so!) So you might find it more natural to have a much longer `PATH` with the C shell than with `cmd.exe`.

Also, if you were going to use an application that needed mixed case characters in some of its environment strings and you were running on OS/2 1.1, you'd want to define them in `login.csh`, not `config.sys`. (Using lower case characters in `config.sys` was unreliable on OS/2 1.1; many users found it caused random system failures, e.g., OS/2 crashing while formatting a series of floppies.)

In addition to `PATH`, the environmental variables used by Hamilton C shell include `HOME`, `PROMPT1` and `PROMPT2`, `TABS`, `CDPATH`, `DRIVEMASK`, `MIXEDCASEDRIVES`, `SWITCHCHARS` and a number of screen color configuration variables.

`PROMPT1` or `PROMPT2` or their aliases `prompt1` and `prompt2` control the prompt strings you'll see for a new command or a continuation line.

`TABS` is used by `more.exe`, `head.exe`, `tail.exe` and other utilities to expand out any tab characters it sees into space characters on the screen. For example, setting `TABS` = 3 means tabstops every 3 character cells.

`CDPATH` is used by `cd` and `pushd` to specify other directories in addition to the current directory in which to search for a subdirectory you're trying to change to. Its format is the same as that as `PATH`: a list of directories separated by colons, but there's no need to list the current directory.

`DRIVEMASK` lets you confine the default list of drives you want searched when you wildcard a driveletter (e.g., ''*:\hamilton'') or reported on by the `du`, `pwd` and `vol` utilities. The value should be a list of upper- or lower-case alphabetic characters or ranges of characters representing the drives you're interested in. For example, setting `DRIVEMASK = cf-h` means you want drives `C:`, `F:`, `G:` and `H:` reported, assuming they exist.

`MIXEDCASEDRIVES` is used by `ls.exe`, by the shell's fullname and wildcarding functions (including filename completion), by the current directory functions (`cd`, `pushd`, `popd` and `dirs`) and by `pwd.exe` to tell which drives you want reported in the actual upper or lower case characters returned by the OS/2 kernel. If you have HPFS or NTFS drives, it's possible to create filenames that have mixed upper and lower characters and you may not want these all routinely shifted to lower case. You set `MIXEDCASEDRIVES` just like `DRIVEMASK`, giving it a list of drives. It's also possible to list UNC names that should be considered mixed case. For example, typing

```
setenv MIXEDCASEDRIVES = [a-z],\\
```

sets all drive `a:` through `z:` and all UNC names on all networked machines as mixed case. Or for example, typing

```
setenv MIXEDCASEDRIVES = \\alpha,\\ps2\d^$
```

sets all the drives on the `alpha` machine and only the `d:` drive on `ps2` as mixed case. You can list as many entries in MIXEDCASEDRIVES, separated by commas, semicolons or spaces, as you wish.

`SWITCHCHARS` is used by the shell and all the utilities supplied with it to indicate what characters you intend as introducing one of the option switches. By default, the C shell tries to satisfy users coming from both DOS, OS/2 or NT and UNIX backgrounds and will accept either "`-`" (DOS-style) or "`/`" (UNIX-style) as introducing an option. Folks coming from a straight UNIX background may find that inconvenient, particularly if they're used to typing filenames using the forward slash; "`cd /foo`" will certainly not do what they want, for example. The solution is to "`setenv SWITCHCHARS = -`", causing only those words that start "`-`" to be recognized as options.

Prompt Strings

Prompt strings are fairly personal matter. This really is beauty in the eye of the beholder only! There are two prompt situations:

1. The main prompt, when the shell expects a new command. Set this with `PROMPT1`.
2. Continuation lines, where the shell is collecting more text before running something. An example would be whatever commands you type inside a `foreach` loop. Set this with `PROMPT2`.

When the shell needs to prompt you, it looks at the appropriate `PROMPT`*x* string and evaluates it as if it were looking at a double quoted string. Any backquoted strings or variable substitutions it finds there are evaluated and whatever results is printed. Wildcards are not replaced nor is the spacing affected. It's quite literally double-quoted: the shell actually pastes double quote characters around the string before passing it to the parser.

If you always wanted the same literal text string displayed, that would be easy but probably not too interesting:

```
543 D% set PROMPT1 = "Hello from CSH!  "
Hello from CSH!  _
```

The difficult part of setting your prompt is remembering that if you want a substitution re-evaluated each time a new prompt is printed, you have to quote the string when you define it to defer the evaluation. Here’s the *wrong*, then the *right* way to create a default IBM-style prompt with your current directory inside rectangular brackets.

```
Hello from CSH!  set PROMPT1 = "[$upper(cwd)] " # Wrong way
[D:\DOUG] cdd c:\      # Notice that the prompt won’t change
[D:\DOUG] set PROMPT1 ’[$upper(cwd)] ’ # Right way
[C:\] _
```

Notice how we’re using the built-in `upper` procedure as if it were a variable substitution with the ‘‘$’’ in front. When the shell sees it’s really a procedure reference, what’s inside the parentheses is evaluated as an expression list. That’s why the `cwd` variable didn’t need its own ‘‘$.’’

There’s really no limit to what you can put inside a prompt string. You can have command substitution and special characters. It can even stretch over multiple lines if you put in carriage return/newline character combinations:

```
[C:\] set PROMPT1 = ’‘date‘^r^n$cwd ’
Mon Oct 23 1989 11:15:15.03
c:\ _
```

The tradeoff to remember in creating your prompt string is that whatever you put there is going to be run every time you need a new prompt. If you make it too complex, it’ll still run, but it could be annoyingly slow. Remember that it takes longer to run external programs than to use shell variables or procedures. If you want something really unusual, try writing a procedure that calculates and returns the string you want.

User-Defined Colors

All of the screen colors used by the C shell and any of the utilities are completely under your control. Here are the variables that control the color settings and default values:

Name	Use	Default
`ADDITIONS`	Lines added found by `diff`.	Bright Green
`COLORS`	Normal screen colors.	White on Black
`DELETIONS`	Lines deleted found by `diff`.	Bright Red
`DIRECTORIES`	Directories listed by `ls`.	Bright
`DUPLICATES`	When filename completion matches more than one name.	Green
`FOREIGNFILES`	Filetypes in a `tar` file that have no counterpart on OS/2 or NT.	Bright Red
`HIGHLIGHT`	Current disk or directory.	Bright
`MATCHFAIL`	When filename or command completion doesn’t match anything.	Bright Red
`MOREEOF`	End or Top of File in `more`.	Green
`MOREERROR`	Unrecognizable command to `more`.	Bright Yellow
`MOREFILLIN`	User response to `more` prompt.	White
`MOREPROMPT`	Prompt line in `more`.	Green

`MORETOPMEM`	Top of Memory message from `more`.	Bright Yellow
`SYSTEMDIRS`	Directories with the System bit on.	Bright Green
`SYSTEMFILES`	Files with the System bit on.	Green

You can choose any color combinations you like from the following set: `black`, `red`, `green`, `yellow`, `blue`, `magenta` (or `blue red`), `cyan` (or `blue green`) and `white`.

Foreground colors may also be `bright`, `dim` (meaning simply, ''not bright''), `blink` or `reverse`. The keyword ''`on`'' introduces background colors. `Blink` only causes true blinking full-screen; in a text window, it just makes the background brighter. Also, `yellow` is a true yellow only if it's `bright`. These are system limitations not related to the C shell.

The names of the colors and the keywords `bright`, `dim`, `blink`, `reverse` and `on` are not case sensitive and may be typed in any combination of upper or lower case characters. The names of the environmental variables themselves must be all in upper case.

If a foreground or background color is left unspecified, that plane is considered transparent and inherits the color underneath it.

You can set the colors either from within the C shell itself by simply typing the appropriate `setenv` statements or by including them in your `login.csh` file. Here's an example color scheme that's a little more colorful than the defaults:

```
setenv   COLORS      =   white on blue
setenv   DELETIONS   =   bright white on red
setenv   ADDITIONS   =   bright white on green
setenv   MOREPROMPT  =   red on white
setenv   MOREFILLIN  =   black
setenv   MOREERROR   =   bright white on red
```

On OS/2, you might also choose to place these settings into your `config.sys` as the corresponding `SET` statements:

```
SET      COLORS=WHITE ON BLUE
SET      DELETIONS=BRIGHT WHITE ON RED
SET      ADDITIONS=BRIGHT WHITE ON GREEN
SET      MOREPROMPT=RED ON WHITE
SET      MOREFILLIN=BLACK
SET      MOREERROR=BRIGHT WHITE ON RED
```

(Notice that if you choose to use `SET` statements in `config.sys`, you should be sure not to leave any space around the equal signs. Also, unless you're running OS/2 1.2 or later, type everything in upper case.)

Or, on Windows NT, these settings could be made through the Control Panel.

Initial Current Directories

The `login.csh` file is also a convenient place to set up all you initial current directories. The Start Programs menu lets you specify a particular current disk and directory but not what your current directories are on the other disks when you start up; the current directories on those other disks are always ''\''. Generally, people find it convenient to be able to choose something else and they do this by putting the appropriate ''`cd`'' statements in `login.csh`.

The final use for `login.csh` is in setting up certain local variables that you want different in the login shell. For example, a login shell normally dumps a history list when it exits; you may want to turn this off by setting `savehist = 0`. Also, you may not want an end-of-file (from accidentally hitting `^Z` once to many times) to cause the shell to exit; you can tell to insist on an `exit` statement by setting `ignoreeof` = 1.

startup.csh

The `startup.csh` file is read by *all* copies of the C shell, not just login or root level copies. If you don't want the startup file read, you have to specifically tell it with the ''`-F`'' (*Faster* startup) option. `startup.csh` is read after `login.csh` whenever both are being read. This means you can depend on the environmental variables being set up already when the startup file runs.

`startup.csh` is a good place to define any aliases or function key definitions you use, since you'd probably always want them available but can't pass them in the environment to any child copies of `csh.exe`. The `startup.csh` file that comes with the shell defines a number of popular aliases including some for getting at some of `cmd.exe`'s internal functions; most people add a few of their own.

The other thing you may want to add to your `startup.csh` file are settings for some of the `set` variables that customize how the shell runs. These aren't passed in the environment. Look through the lists in the Language Reference section. Some you may want to set differently than the defaults are `bsdhistory`, `cdhome`, `chgdisk`, `escapesym`, `histchars`, `ignoreerrors`, `ignorestatus`, `noclobber`, `nonomatch`, `nullwords` and `tailstatus`. A lot of what you choose will depend on whether you're coming from a DOS or a UNIX background.

Change Directory

If your background is DOS, you'll probably want `cd` to just report the current directory if you don't give it an argument. Those with a UNIX background may want it to mean ''change to the home disk and directory.'' That's determined with the `cdhome` variable; the default is a DOS-style reporting only.

Another customization you may to do is to intercept `cd` so that you can capture your last current directory whenever you change directories:

```
proc cd(dest)
   @ lwd = cwd    # capture the last working directory
   if ($dest == "")
      chdir
```

```
      else
         chdir $dest
      end
   end
```

Berkeley-style History and Escapes

Also, if your fingers learned to use the "`!-`*n*" style of history references on a Berkeley UNIX system, you'll want to set `bsdhistory` = 1. True die-hard (and perhaps daring) former UNIX users may want try setting the `escapesym` back to a backslash; it'll work with the C shell but you're on your own with other applications or tools.

Berkeley Compatibility Mode

Hamilton C shell does implement a substantial number of significant improvements over the over original Berkeley C shell. By and large, we expect most users to find these changes welcome. But if you're trying to run a script developed using the Berkeley C shell or if you simply want get a more precise "Berkeley mode" interactively, you can do that.

Following a common convention on UNIX that the first line of a script can identify the language processor to be used with it, if the C shell encounters a script that starts with

```
#!/bin/csh
```

it will shift to a fairly precise emulation of the original Berkeley C shell.

To enter this mode for interactive work, start the C shell with the "`-B`" (Berkeley compatibility) option.

For more details on differences between the Hamilton and Berkeley C shells and on Berkeley compatibility mode, please please refer to the Compatibility section beginning on page 129.

Error Handling

`ignoreerrors`, `ignorestatus`, `noclobber`, `nonomatch` and `nullwords` let you tailer how the shell responds to various exception situations. They let you determine whether you think certain things are errors. For example, should a child process that returns a non-zero return code but otherwise seems to run okay be considered an error? If you set `ignorestatus` = 0, it will be. Similarly, `noclobber` lets you intercept accidental attempts to overwrite an existing file with i/o redirection. `nonomatch` tells what should happen if wildcarding doesn't match anything. `nullwords` tells whether you think it's an error to use a subscript that's off the end of an array.

Calling the C shell from Other Applications

Many applications, e.g., editors, make utilities and so on, depend on being able to call up the command processor. For example, make uses `cmd.exe` to actually process each command in the make file that it determines should be run. Most editors (and many other applications) provide a way of temporarily suspending themselves and invoking `cmd.exe` so you can run a few commands and then exit to return back to the editor.

Usually, these applications look at the `COMSPEC` environmental variable to determine the full pathname for `cmd.exe`. If, like make, they're just calling it with a single command on the command line, they use `cmd.exe`'s `/C` option.

If you'd like to use the C shell instead of `cmd.exe` with these applications, set `COMSPEC` to point to the C shell instead and use the `CSHOPTIONS` environmental variable to tell the C shell to interpret the `/C` option flag in a way that's compatible with the meaning `cmd.exe` would attach to it:

```
setenv COMSPEC = c:\hamilton\bin\csh.exe
setenv CSHOPTIONS = -X
```

Alternately, you can put these definitions right into your `config.sys`, rewriting them as:

```
set COMSPEC=c:\hamilton\bin\csh.exe
set CSHOPTIONS=-X
```

When the C shell starts up, if it discovers that `COMSPEC` points to itself, it will look through the search path to find the real `cmd.exe`. This is to make sure it will still be able to run `.cmd` files. Since the `-X` option is a toggling option, you can still get at the original meaning of the `-C` option by typing `-XC` to toggle back.

Customizing the Shell

Summary

The next few pages show a couple of somewhat more full-blown examples and outline the contents of the samples directory. There's also detailed discussion of the compatibility issues between the Hamilton and original Berkeley C shells.

Try some experiments. We hope you'll find this product powerful, fast, reliable and easy to use. We hope it will help you get your work done faster and perhaps, more pleasantly.

Summary

Examples

Factor.csh: A self-loading procedure which prints a list of the factors of a number, illustrating the use of recursion.

```
proc factor(n)
  if (n > 2) then
     for i = 2 to floor(sqrt(n)) do
        if (n % i == 0) then
           echo $i
            return factor(n/i)
        end
     end
  end
  return n
end

factor $argv
```

Invoked as:

```
factor 6324489
```

It would print:

```
3
3
702721
```

To print the factors on one line and time how long it takes:

```
time echo ‘factor 6324489‘
```

The ‘...‘ sequence means command substitution: run what's inside the backquotes and substitute the output back onto the command line. This would print:

```
3 3 702721
0:00:02.35
```

Examples

Whereis.csh: A self-loading procedure to find all the files anywhere on the search path corresponding to the command name, illustrating pattern matching and file system tests.

```
proc whereis(name)
  local i, j
  if (name =~ "*.*") then
     foreach i ($path)
        if (i =~ "*\") then
           if (-e $i$name) echo $i$name
        else
           if (-e $i\$name) echo $i\$name
        end
     end
  else
     foreach i ($path)
        if (i =~ "*\") then
           foreach j (.csh .exe .com .cmd)
              if (-e $i$name$j) echo $i$name$j
           end
        else
           foreach j (.csh .exe .com .cmd)
              if (-e $i\$name$j) echo $i\$name$j
           end
        end
     end
  end
end

whereis $argv
```

Invoked as:

```
whereis ls
```

It would print:

```
c:\os2\bin\ls.exe
```

`ls.exe` is the file directory lister. Invoked as:

```
time ls -l ‘whereis more‘
```

It would show the two versions of `more`. (Our `more` ''is less filling and tastes better.'')

```
---A-  Mar 20  8:00   20123  c:\os2\hamilton\more.exe
---A-  Oct 26 12:00   31658  c:\os2\ibm\more.com
0:00:00.97
```

Samples Directory

The sample C programs and C shell scripts in the samples directory are meant to help you install or experiment with Hamilton C shell. Deliberately, they’re relatively trivial. All were created assuming `TABS=3`.

`args.c`	A simple C program that prints out the `*argv[]` (argument) and `*envp[]` (environmental variable) arrays. Notice that wildcarding, variable substitutions, quoting and command substitutions are done before the C program is started. If you do a lot of wildcarding, you can create and pass VERY long parameter lists (up 64K characters.) Try some of these commands:

```
% args "ho"w ’no’w
% args "$cwd" ’$cwd’
% args * "*" ’*’
% args ‘whereis more‘
% args ’‘whereis more‘’
% args * *\* *\*\* | more
```

`bits.csh`	A simple self-loading procedure that calculates the minimum bits required to represent the argument it’s passed as a binary integer.
`bumpdate.csh`	Print the date n number of days forward or backward from a given date. If only the bump value is given, today’s date is bumped.
`caldate.csh`	Print the date corresponding to a given Julian day.
`calendar.csh`	A C shell script for printing out the calendar for any given month, highlighting the current date. If no date is given, this month’s calendar is printed.
`colors.csh`	Instructions and examples on customizing the screen colors.
`cl.csh`	On NT only, run the compiler and linker for an NT program. Avoids having to fool with a make file just to compile hello, world. Works pretty much just like the cl command on DOS would.
`deltaday.csh`	Print the number of days separating two dates. If only one date is given, the difference between it and today’s date is returned.
`dumpenv.c`	This C program writes out the environment it’s passed in the form of `setenv` commands. If you’re installing Hamilton C shell for the first time, `dumpenv` is a convenient way to snapshot the environmental variables you’ve been using with `cmd.exe` in a form you can append to your `login.csh` file.
`duplicat.csh`	Look for duplicate files anywhere in a directory tree.
`easter.csh`	A C shell script that calculates when Easter will occur in any given year. If no year is given, the current year is assumed.
`factor.csh`	The simple factor C shell script shown in the User Guide. It’s intended to show to show the use of recursion, expressions, and a self-loading procedure.

Examples

`finance.csh`

Another C shell script showing expression evaluation. This defines a number of routines for calculating financial conversion factors, e.g., from present to future value.

`getprio.c`

This C program (supplied with the OS/2 version) retrieves and prints its scheduling priority, demonstrating the effect of using the `eval` command to run a command at a higher or lower priority. Try these examples:

```
% getprio
% eval -i getprio
% eval +20 (getprio; eval +20 getprio;
getprio); getprio
```

`julian.csh`

Calculate the Julian day number (number of days since January 1, 4713 B.C.) for any given date. If you don’t give a date, it uses today’s date.

`makecpgm.csh`

A simple C shell script (supplied with the OS/2 version) showing how a “make” function might be written in the C shell language. This one rebuilds any `.exe` files in the current directory that are older than the corresponding `.c` file or any of the `.h` files. In fact, the `.exe`’s in this directory were built with this script.

`mcvisa.csh`

A simple C shell script that constructs a special checksum of a credit card number to tell if the card number is plausible or not. The checksum used is designed to catch transposed or incorrect digits. Try it on the cards in your wallet.

`myecho.c`

A variation on the built-in `echo` command that prints its `*argv[]` (argument) list with quotes around each word it’s passed and tells the total character count. Try these examples:

```
% myecho now     is the
% myecho "now     is" the
% myecho ‘ls‘
% myecho ‘echo‘
% myecho ‘echo hello‘
% myecho * *\* *\*\* | more
```

`newfiles.csh`

List all the files or directories in the current directory that do not occur in the specified directory.

`postage.csh`

Calculate the U.S. first class postage required for a given weight in ounces.

`rcode.c`

A trivial C program that just prints, then exits with the return code value you pass it. You can use this routine to see how the `status` variable is set and also, how the “`;`,” “`||`” and “&&” statement connectors work. Try these examples:

```
% rcode
% calc status
% rcode 1
% calc status
% echo $status
% echo status
% rcode 2
```

```
% calc status
% rcode 0 || rcode 1
% rcode 1 || rcode 2
% rcode 0 && rcode 1
% rcode 1 && rcode 2
% rcode 0 ;  rcode 1
% rcode 1 ;  rcode 2
```

`rcode` also illustrates another aspect of return codes to consider: if you use C library stdio (as `rcode.c` does) and you exit with a non-zero return code, stdio thinks it was an error and discards anything in the stdio buffers. In the following example, stdio writes to the screen are unbuffered so it works; but pipes are buffered, so nothing gets written to it:

```
% rcode 1
1
% rcode 1 | more
--- End of file ---
%
```

If you're writing an application that uses return codes, you should remember to explicitly flush the buffers with stdio `fflush()` or use the kernel routines, e.g., `DosWrite()`, directly

`sizeof.csh`	A short C shell script that calculates and prints the cumulative size of any number of files or directories.
`ts.csh`	A C shell script that searches for occurrences of a simple string in all the files with a given extension anywhere in a directory tree.
`viopaste.c`	A short C program (supplied with the OS/2 version) to enable pasting into a Presentation Manager text window under OS/2 1.2 or 1.3.
`weekday.csh`	Print the day of the week corresponding to any given date.
`winerror.csh`	Print the Win32 message corresponding to a given error code. (Only supplied with the Windows NT version.)

Examples

Compatibility Guide

This section details the specific differences between the Hamilton C shell and the original UNIX C shell[†]. It also describes the Hamilton C shell's Berkeley compatibility mode, used for running Berkeley C shell scripts.

Berkeley 4.3 Buglist problems have been fixed.

1. Shell procedures have been provided and the clumsy argument mechanism for aliases has been dropped.

2. Commands typed within loops or other control structures are properly added to the history list.

3. Control structures are recursively parsed, allowing piping between them. For example:

   ```
   foreach i (a b c) echo $a; end | wc
   ```

   properly displays

   ```
             3         3        12
   ```

4. Any of the ':' editing modifiers can be used on any substitution. Also, a space inside the search string in a '':s/.../.../'' command will match the space between two words. In the UNIX C shell, only certain modifiers could be used on a given type of substitution and it is not possible to perform a search/replace that crossed word boundaries.

The language has been regularized.

1. The **set**, **setenv** and **alias** commands will now accept the same basic syntax. The UNIX C shell had a number of anomalies: an '=' sign was required for a **set** but not for **setenv** and **alias**; parenthesis were required around a word list for a **set** but not for setenv and alias; the **set** statement ignored all but the first argument word but **alias** would not, etc.

2. Variables or word lists are always indexed counting the first word as element zero. The UNIX C shell counted from zero when indexing with '':*n*'' notation but from one when using ''[*n*]'' notation. **argv**[0] is the first argument word, not the name of the shell script being executed. The name of the script is kept in the local variable

---

[†] The references used for comparison are the Berkeley 4.3 *Unix User's Manual: Reference Guide* (University of California, 1986) and *The UNIX C Shell Field Guide* by Gail and Paul Anderson (Prentice-Hall, 1986.)

$scriptname. This can be overridden by setting the inheritable per-thread variable **bsdargv** = 1, causing **argv[0]** to be the name of the script.

3. In keeping with the desire to consistently index from zero, the last command entered into the history list, ‘‘!!’’, is considered the 0-th element; ‘‘!-1’’ is the line before it. The UNIX C shell considered these to be the same. A built-in variable, **bsdhistory**, is provided for those whose fingers prefer the Berkeley numbering convention: if you set **bsdhistory** = 1, ‘‘!!’’ and ‘‘!-1’’ are the same.

4. Where an expression is expected, conventional high level language syntax is now acceptable. The UNIX C shell required spaces around any expression operators, a variable reference required a ‘$’ to introduce it, parenthesis were required to avoid confusing ‘‘less than’’ with `i/o` redirection, etc. What had to be typed as

   ```
   @ i = ($j + 3 * $k < 10)
   ```

   under the UNIX C shell can now be typed (for example) as

   ```
   @ i=j+3*k<10
   ```

   (The original UNIX C shell expression syntax is still entirely acceptable and will still produce correct results.)

5. Inside a ‘‘[...]’’ array index, the shell always looks for an expression, never an editing-style word select. Syntax and keying rules are the same as with any expression.

6. The **case** statement now accepts an expression to be matched rather than only a pattern. (To specify a static pattern, enclose it in quotes.) To determine a match against a **case** clause, the **case** expression is evaluated, converted to a string and then used as a pattern to compare against the **switch** value.

7. The various different end statements used by the UNIX C shell, **end**, **endif** and **endsw**, have been replaced by a single **end** statement. Similarly, the two break statements, **break** and **breaksw**, have been replaced with a single **break** statement. For compatibility with existing scripts, the obsolete keywords are implemented as aliases.

8. Since Hamilton C shell is free format (i.e., new statements need not begin on a new line), the UNIX C shell convention of chaining **if** statements with a single **end** if the **else** and **if** are on the same line isn’t possible. Instead, an **elif** keyword has been added.

9. The obscure use of several **break** statements in a row on a single line to break out of several levels of control statements at once has been eliminated. In its place, a label may be specified as an operand to indicate the control structure to ‘‘break’’ out of.

Modern compiler technology has been employed.

Statements are parsed and compiled into an internal form before any substitutions or other evaluation is attempted. This offers an enormous performance improvement, particularly when iteration is involved. (The UNIX C shell would actually reparse each statement inside a **foreach** loop each time through the loop.)

If command- or variable-substitution creates any of the following reserved words or tokens, the special semantic meaning will be lost since substitution is done after parsing of statement structure. Instead, they will simply be treated as character strings. These reserved words are:

Introducing a clause in a structured statement:

alias
break
by
calc
case
continue
default
do
elif
else
end
eval
exit
for
foreach
goto
if
local
onintr
proc
repeat
return
set
setenv
setkey
source
switch
then
time
to
unalias
unlocal
unproc
unset
unsetenv
unsetkey
until
while
@

Anywhere:

( ) < > & | ;

In an expression:

+ - * / % =

Similarly, labels cannot be run-time evaluated to see what the label on a statement is; it must be evaluated when the statement is first parsed.

Extensions:

1. Command line editing with the arrow keys, etc., and the **setkey** statements are new.
2. The procedure mechanism, including the **proc**, **unproc** and **return** statements and the various built-in procedures, is new.
3. Local variables and **local** and **unlocal** statements are new.
4. The use of color highlighting to indicate exception situations in filename or command completion is new.
5. The **for** statement, providing numeric iteration, and the **calc** statement, which writes the result of expression evaluation to stdout, are new.
6. The ''**'' and ''**='' exponentiation operators are new.
7. Floating point arithmetic is new.

8. The path hashing mechanism is substantially less sensitive to blindspots caused by creating a new executable in one of the path directories and not manually specifying rehash. The UNIX C shell would not be able to find the new file; this shell makes a second pass through the path directories whenever hashing fails, looking for this sort of problem before it reports failure. If it finds a blindspot, it automatically rehashes that directory.

9. History references are allowed in the inline text supplied with the “<<” i/o redirection mechanism. Also, the inline text is remembered in the history list, each line as a single word. This avoids the user having to remember and retype the inline text any time one of these statements is recalled from the history list or if the history list is dumped for use in a script file.

10. Exclusion ranges, e.g., “[^a-z],” can be used in a wildcard pattern.

11. Escape sequences to encode special characters (e.g., “^a” for audible bell or “^b” for backspace) are recognized in the arguments to any command, not just echo. Because this processing is internal to the shell, it is not necessary to type two escapes in a row to access this feature. (Refer to the **echo** command help screen for a complete list.)

12. Argument lists passed to a child process can be much larger than are allowed under UNIX. The UNIX C shell allows only roughly 6K characters to be passed, depending on the revision level; this shell allows up to 64K to be passed to a child process under OS/2 or 32K under NT, the kernel limits on these systems. There is no command line limit to an internal command such as **echo**. This is of particular importance when wildcarding is used heavily.

13. Quoted strings are shown in the history list exactly as they would have to be typed. (The Berkeley UNIX C shell marked a character as quoted by setting its high-order bit; setting aside portability issues, it had the side-effect of not being visible in the history list.)

14. Parentheses in an argument list to an executable statement need not be escaped, so long as they are matched. Semicolons, i/o redirection symbols, etc., inside these parentheses are treated simply as text and are passed straight through to the application.

15. The “:b” (base) and “:#” (count) editing operators are new.

16. The indefinite directory wildcard construct, “`...`”, is new.

Restrictions and unimplemented features:

1. History references as way of picking up alias arguments are not supported. Procedures should be used instead.

2. Job control is not supported. Job control is not currently feasible under Windows NT or OS/2 because once one thread from any process within a window has started to

read the keyboard, the read can not be interrupted. (Fortunately, one can always open more windows.)

3. The use of ''\!'' inside a prompt string to get the statement number is not supported. Use $@ or $stmtnumber instead.

4. The following statements, all fairly specific to UNIX, are not supported: alloc, glob, limit, notify, stop.

5. The comment character, #, must be followed by some white space to be considered the start of a valid comment. (That's because # is a legal character in a filename under both NT and OS/2.)

Adaptation for OS/2 and NT:

1. OS/2 and Win32 NT do not provide a fork( ) call for inexpensively cloning an independent copy of a running process, complete with its own separate memory image. Instead, OS/2 and NT provide a faster alternative called threads, which creates an separately scheduled flow of control through the memory space of a single process.

   In general, the Hamilton C shell spawns a new thread anywhere the Berkeley UNIX C shell would have used a process. Using a new thread instead of a new invocation of the Hamilton C shell saves over a second each time. Individual threads manage their own notions of current directories and current disk and certain per-thread variables but the dictionary of aliases, procedures and most variables is shared among all threads.

   The result is that background activities and C shell scripts can change variables, define procedures, etc., for use by the other threads. For example, procedures can be written as self-loading scripts. (See the whereis.csh file for an example.)

2. OS/2 and NT conventions are followed: either the ''\'' or the ''/'' characters can be used in a filename; the ''^'' character is normally the escape character; directories in the **PATH** environment variable are separated by semicolons, etc.

3. Labels cannot be a single letter. (This is to avoid confusing the drive letter in the pathname of an executable file as a label.)

4. Since OS/2 and most NT filenames are case-insensitive, they are routinely translated to lower case for better readability. (This can be overridden using the **MIXEDCASEDRIVES** variable.)

5. Executable files are recognized by their extension. The following extensions are recognized (in this order): .csh, .exe, .com, .cmd, .bat. .csh files are interpreted as C shell scripts by a new thread, .exe and .com files are executed with the DosExecPgm and DosStartSession kernel functions under OS/2 or with the CreateProcess kernel function under NT, .cmd files are interpreted by a child process running cmd.exe,

and .bat files are passed to a Virtual DOS machine (VDM) under OS/2 2.x or to `cmd.exe` under NT.

6. **PROMPT1** and **PROMPT2** variables are used to set the primary and secondary prompt strings. Using the UNIX C shell variable **PROMPT** would have conflicted with cmd.exe's use of the same name and would have meant a nonsense prompt string any time either command processor was invoked by other.

7. The following startup or other files have been renamed to be more consistent with OS/2 and NT filename conventions: ~/.cshrc as ~\startup.csh; ~/.login as ~\login.csh; ~/.logout as ~\logout.csh; and ~/.history as ~\history.csh. The ~\login.csh file is read before, rather than after the ~\startup.csh file. When starting the shell as a new session, very little environmental information may be passed; the login.csh is more usefully the first file read in this situation. When starting a subshell, either from csh.exe or cmd.exe, the environment is presumably already set up.

Berkeley Compatibility Mode:

Berkeley Compatibility Mode provides fairly strict compatibility with the original BSD C shell. Triggered by trying to run a script that starts with `#!/bin/csh` or interactively if the shell is invoked with the `-B` option, it causes the C shell to process statements in a more fully Berkeley-compatible fashion. (Scripts that do not start with `#!/bin/csh` will still be processed according to Hamilton C shell rules, even if the `-B` option is used to request Berkeley compatibility interactively.) In compatibility mode:

1. The **status** variable will reflect the return code from the rightmost stage of a pipeline. The **tailstatus** variable will be ignored.

2. All the shell variables will be snapshotted and all new variables made local to the thread.

3. Berkeley-style **$***var***[...]** indexing notation will be used, where the indexing is by word selection operators (like the `:`-editing operators) rather than by expression.

4. All variable arrays (except **argv**) will start with element 1. Accessing element 0 will give a null.

5. **$0** or **$argv[0]** will be the scriptname. **$argv** will be the rest of the argument vector. The **bsdargv** variable will be ignored.

6. The **#** character will not need to be followed by white space to be considered the start of a comment.

7. The patterns in a **case** test (inside a **switch**) will be strings and need not be quoted, rather than arbitrary expressions. Also, the switch value is evaluated as a wordlist which may contain variable or command substitutions and wildcards and then rendered as a string.

8. **endif** and **endsw** will be predefined aliases for **end** (but only when closing an **if** or **switch**, respectively). **breaksw** will be a pre-defined alias for **break**.

9. ''**set foo**'' and ''**setenv foo**'' will set **foo** to a null string, not dump its value.

10. **/** and **/=** will perform integer division.

11. The right operand of the **=~** and **!~** pattern matching operators will be taken as a word which may contain wildcards.

12. In an expression, a variable name must be preceded by **$**. If it isn't, it'll be taken as a literal string.

These changes should allow most scripts to run without problems. However, there will still be a few differences:

1. The escape character will still be controlled by the **escapesym** variable (shared across all threads), which defaults to **^**, not ****.

2. Environmental variables will still be shared. Changing them in a script will change them as seen by the parent.

3. The special meaning of several **break** statements on one line will not be supported.

4. **unset** and **unsetenv** still do not accept patterns.

5. The following commands are not supported: **bg**, **exec**, **fg**, **glob**, **jobs**, **limit**, **nice** (but **eval** gives similar functionality), **nohup**, **notify**, **stop**, **suspend**, **unlimit** and %*job*.

6. History references inside **alias** definitions will still not be supported.

Compatibility

Language Reference

Basic Statements:

Same as `cmd.exe`: a file reference + arguments.

Examples:
```
cl -AS -G2 -Zi hello.c
cp hello.exe c:\os2\bin
```

Hamilton C shell maintains a hash structure which allows it to quickly search for a suitable `.csh`, `.exe`, `.com`, `.cmd` or (on OS/2 2.x or NT) `.bat` file (in that order) in each of as many as 256 path directories. Wildcarding is done by the shell before invoking the child. Under OS/2, up to 64K of environmental and 64K of command-line argument data can be passed to a child process; under NT, up to 32K of command-line data can be passed. These are the limits of the kernels, not the C shell; there is no limit on overall command line length in the C shell itself.

Condition-Testing:

Hamilton C shell provides both **if** and **switch** constructs. The **if** statement comes in both short and long forms. The long form uses a **then** keyword and allows an optional **else** clause. The short form, which must be typed on one line, dispenses with the **then** keyword and accepts a single statement to be executed if the condition is satisfied.

```
if ( <expr> ) then
    <statement_list>
else
    <statement_list>
end

if ( <expr> ) then
    <statement_list>
end

if ( <expr> ) <statement>
```

Where an expression is expected, a conventional high level language syntax is accepted: e.g., names refer to variables, '*' means multiply, not wildcard and '>' means greater than, not i/o redirection.

if statements can also be chained using the **elif** keyword. The last **if** in the chain may be either a short- or a long-form **if** statement.

```
if ( <expr> ) then
    <statement_list>
elif ( <expr> ) then
    <statement_list>
else
    <statement_list>
end

if ( <expr> ) then
    <statement_list>
elif ( <expr> ) then
    <statement_list>
end

if ( <expr> ) then
    <statement_list>
elif ( <expr> )  <statement>
```

In a **switch** statement, expressions are compared by pattern match: the **case** expression can be a string with wildcard characters. Comparisons are made down the list of alternatives until one matches. All following statements are executed until a **break** is encountered. A **default** clause is optional but is always satisfied if no other **case** matches.

```
switch ( <expr> )
    case <expr> :
        <statement_list>
    case <expr> :
        <statement_list>
    default :
        <statement_list>
end
```

Iteration:

foreach <name> **(** <word list> **)**
<statement_list>
end

for <name> **=** <expr> [**to** <expr>] [**by** <expr>] **do**
<statement_list>
end

while (<expr> **)**
<statement_list>
end

repeat <number> <statement>

repeat
<statement_list>
until (<expr> **)**

The **foreach** statement is intended for iteration over a list of words, often specified by wildcarding. The **for** statement offers the more conventional numeric iteration. Multiple iteration ranges, separated by commas, can be specified on the **for** statement.

Procedures:

proc <name> **(** [<namelist>] **)**
<statement_list>
return [<expr>]
end

proc

unproc <namelist>

Procedures defined by the **proc** statement can recursively call other procedures. They can be referred to inside an expression or as a new command, in which case any value returned is written to stdout. The **proc** statement with no arguments causes a list of the available procedures to be written. The **unproc** statement allows a procedure to be discarded.

Aliases:

alias <name> [=] **(** <word list> **)**
alias <name> [=] <word list>

alias
alias <name>

unalias <namelist>

Aliases can be referred to at the beginning of a command and provide a quick, user-defined shorthand. **alias** <name> with no arguments prints the value of the name. **alias** without any arguments prints the values of all aliases.

Variable and Expression Manipulation:

@ <expr>
calc <expr>

The @ and **calc** statements will each calculate the value of an expression; the @ statement does it silently while the **calc** statement writes the result to stdout.

set <named_ref> [=] **(** <word list> **)**
set <named_ref> [=] <word list>
setenv <named_ref> [=] **(** <word list> **)**
setenv <named_ref> [=] <word list>
shift [<name>]

set
set <name>
setenv
setenv <name>

unset <namelist>
unsetenv <namelist>

The **set**, **setenv** and **shift** statements manipulate variables as words rather than expressions. **set** defines a set variable that's shared between all threads in the shell; **setenv** puts it into the environment and inherited by child processes. **set** or **setenv** with no operands prints a list of all defined variables of that type. **set** <name> or **setenv** <name> with no arguments print the value of the named variable. **unset** or **unsetenv** let you discard a variable.

Local Variables:

The **local** command lets you define a list of variable names that you don't to share with other routines or other processes or threads (except your own child threads). When you define a local variable it hides any previous definition from any outer statement list. (But you are not permitted to redefine any of the built-in **set** or **setenv** variable names.)

local <namelist>
local

The <namelist> should be typed with commas between the names. When you create a new local variable, its initial value is always a null string. Typing **local** with no operands reports the currently defined and accessible local variables, if any.

When you spawn a child thread either implicitly, e.g., to run the second or following stage of a pipeline or explicitly, by typing an ampersand at the end of a command to run it in the background all your current local variables are snapshotted and copied to the child. If, following that, either the parent or the child changes the value of any of these local variables, it affects only its own copy.

Local variables are automatically discarded as soon as execution leaves the statement nesting level in which the variable was created. You can also explicitly discard local variables using the **unlocal** command.

unlocal <namelist>

In all other respects, local variables act just like any other variables, though you may find they're slightly faster since the shell doesn't need to semaphore its use of them.

Function Keys

setkey command:

The **setkey** command lets you define a list of words that should be stuffed back onto the command-line whenever you press a particular function key. The syntax is exactly the same as used in the **set**, **setenv** and **alias** commands:

setkey <fkey> [=] (<word list>)
setkey <fkey> [=] <word list>

where <fkey> is any of the function keys **f1** (or **F1**) through **f12** (or **F12**.)

Typing **setkey** with no operands reports the current function key bindings, if any. Also, a corresponding **unsetkey** command lets you discard key bindings:

setkey
unsetkey <fkeylist>

The <fkeylist> should be typed with commas between the keys. For example:

```
unsetkey f1, f2
```

Using the Function Keys

Key	Meaning
<F*x*>	Clear the command line, post the text bound to this key and execute the command.
Alt-<F*x*>	Insert the text bound to this key at the cursor location but don't execute it yet.
Ctrl-<F*x*>	Clear the command line and post the text bound to this key but don't execute it yet.

Since the function key's bound text is written back into the command line inside command line editor, the substitution happens ahead of any parsing of the command line into words or expansion of history ''!...'' or ''%...'' references so it is possible to meaningfully embed these kinds of references into the key binding.

Miscellaneous Statements

Statement	Function
<drive>**:**	Change current drive.
<label>**:** <statement>	Define a label.
(<statement_list> **)**	Group a list of statements, saving and restoring the current directory during execution
break [<name>]	Exit from the named or, by default, the innermost **switch**, **foreach**, **for**, **while** or **repeat** statement.
continue [<name>]	Continue with the next iteration of the named or innermost **foreach**, **for**, **while** or **repeat**.
exit [<expr>]	Exit from this thread or, if this is the main thread, from the C shell.
goto <name>	Continue at the labeled statement.
onintr <statement>	Define the action to be taken if an interrupt is signaled. Whatever's running is interrupted all the way back up to the block in which the **onintr** was defined. That statement is run in the current thread and execution continues with the next statement in the block where the **onintr** was defined. When execution leaves a block in which an **onintr** is defined, the previous **onintr** (if any) again takes effect. To ignore or defer interrupts, use the **irqmask** variable.
source <wordargs>	Read and process statements from a file as if they were typed into this thread.
time <statement>	Execute the statement and report how long it took.
#	Comment text up to the end of the line. (To be recognized as a valid comment, the **#** must be followed by at least one space or tab.)

Statement Relationships

In order of decreasing precedence:

Operator	Meaning
( )	Grouping
> >! >& >&! >> >>! >>& >>&! < <<	I/O Redirection
| |&	Piping (stdout only or stdout + stderr) between concurrent operations
... &	Background thread or process
&& ||	Conditional execution: only if first succeeds or only if first fails
;	Serial execution

I/O Redirection

Operator	Meaning
> >! >& >&!	Output to a file ('!' allows an existing file to be overwritten; '&' redirects both stdout and stderr.)
>> >>! >>& >>&!	Append to a file
<	In from a file
<< <string>	Inline data: the text on the following lines, up to the line containing only the specified <string> will be fed as stdin to the statement. Unless part of <string> is escaped or single, double or backquoted, command and variable substitution will be done on the inline text.

The grammar is completely recursive, so statements of arbitrary complexity can be freely nested, conditionally executed, piped or redirected.

Expression Operators

In order of decreasing precedence:

Operator	Meaning
( )	Grouping or Procedure call arguments
{ }	Run the enclosed statement list and return 1 if it succeeds or 0 otherwise.
[ ]	Array indexing. (The first element is element 0.)
-A -D -H -R -S -d -e -f -o -w -x -z	File system tests
++ --	Prefix and postfix increment/decrement
~ - ! +	Bitwise, arithmetic and logical complements and unary plus
**	Exponentiation
* / % //	Multiplication, Division, Remainder and Integer Division
+ -	Addition and Subtraction
<< >>	Bit Shifting
== != =~ !~ < <= >= >	Relation-testing and pattern-matching operators
&	Bit And
^	Bit Xor
|	Bit Or
&&	Logical And
||	Logical Or
?:	Conditional selection
= += -= *= /= %= //= >>= <<= &= ^= |= **=	Assignment operators

Expressions result in sensible types, considering both the types *and* the values of the operands. For example, 10/2 returns the integer 5 but 5/2 produces the floating point value 2.5. Also, the integer 1, the floating point value 1.0 and the string "1" all compare equal.

File System Tests

The operand of a file system test is interpreted as a word, not an expression, and may involve wildcarding. If wildcarding produces more than one match, the test is done on the first one.

Prefix Operator	True if
-A	Archive Bit Set
-D -d	Directory
-H	Hidden File or Directory
-R	Read-only File or Directory
-S	System File or Directory
-e	File or Directory Exists
-f	Ordinary File
-o	Ownership (Same as Existence on an OS/2 or NT FAT file system)
-r	Readable (Same as ordinary file on an OS/2 or NT FAT file system)
-w	Writable (Not Read-only and not a directory)
-x	Executable (Has a .csh, .exe, .com or .cmd extension and, if it's an .exe or a .com file, appears to be a valid OS/2 or NT binary executable.)
-z	Zero-length File

Example:

```
if (-d $a) then
  echo $a is a directory
end
```

Special Devices

Name	Use
aux†	The auxiliary port.
clock$†	The OS/2 system clock.
con	The OS/2 console. This is the text window the shell is running in. Reading from it reads the keyboard; writing to it writes to the screen.
conin$	The NT console keyboard. A read-only device.
conout$	The NT console display. A write-only device.
com1 .. com7	The various async communication ports you may have. They can be read or written.
kbd$	The OS/2 keyboard. A read-only device.
lpt1 .. lpt4†	The line printer ports.
nul	The null device. Reads return end-of-file; writes are discarded.
prn	The print spooler queue.
pointer$†	The OS/2 mouse and on-screen pointer combination. Reads from the mouse; writes to the screen.
screen$	The OS/2 text window. A write-only device.

Hamilton C shell recognizes the OS/2 and NT special device names regardless of whether they are typed in upper or lower case and regardless of whether there is a trailing colon. For example, `COM1:` is the same as `COM1` is the same as `com1`. Remember to quote the device name if it has a dollar sign at the end so it won't be misinterpreted as the start of a variable reference.

Not all devices will be installed on any particular machine. Also, the exact behavior of any device may depend which manufacturer supplied it. Devices marked with † should only be accessed with some caution.

Wildcarding and Pattern Matching

Characters	Meaning
?	Match any single character, including ‘.’ but not ‘\’ or ’/’.
*	Match any number of characters, including ‘.’ but not ’\’ or ’/’.
[**a**-**z**]	An example range: match any character **a** through **z**.
[^**a**-**z**]	An example exclusion range: match any character **not** in the set a through z.
{**a**,**b**}**c**	Alternation: generate both **ac** and **bc**.
`...`	Indefinite Directory: match any number of directory levels -- zero or more -- whatever it takes to make the rest of the pattern match.

Patterns are used both for traditional filename wildcarding in word mode and for examining strings in expression mode. Patterns are nestable arbitrarily and a recursive comparison algorithm is used to guarantee a sensible result no matter how complex the pattern. For example: `*r*` or even `*\[a-c]*.[ch]` operate sensibly.

Filename wildcards can be used wherever a word is expected and will match any filename except ‘‘`.`’’ and ‘‘`..`’’ unless it’s marked ‘‘hidden.’’ (To allow wildcarding to match hidden files, set `nonohidden` = 1.) Quoting the word prevents wildcarding. Since filenames are not case sensitive, filename wildcarding isn’t either.

A pattern can also be used to examine a string in a **switch** statement or with the ‘‘**=~**’’ (pattern matches) and ‘‘**!~**’’ (pattern fails) expression operators. In this context, the pattern must be inside quotes, since otherwise it would be parsed as an expression, with ‘‘*’’ being viewed as the multiply operator, etc. When examining a string with a pattern that uses alternation, the pattern ‘‘matches’’ if any of the alternatives matches. When matching strings, case does matter.

When using a pattern to examine a string with ‘‘**=~**’’ or ‘‘**!~**’’ or with a **switch**, the string being tested is the left operand and the pattern is on the right.

Filename Completion

Filename completion lets you type just the first part of a filename and have the shell fill in the rest. The two variations are using the F key for basic filename completion or the D key if you want all the duplicates listed.

Key	Meaning
Alt-F or Ctrl-F	Filename completion. Appending the ``*'' wildcard character onto the end, use the previous word as a wildcard pattern. If it matches a single file, substitute it in with a space following. (If you *don't* want a space following, use the `-N` option when you start the C shell.) If there were multiple matches, but they all had some common front-part that fully ``used up'' the pattern, substitute in just that common front-part and show it in the color specified by the `DUPLICATES` variable (default is green). If substitution wasn't possible, highlight the pattern in the color specified by the `MATCHFAIL` variable (default is bright red). (Any highlighting color is turned off when you press the next keystroke.)
Alt-D or Ctrl-D	Duplicate completions. Same wildcarding, but if there are multiple matches, show them all with a space following. (Here again, if you *don't* want a space following, use the `-N` option when you start the C shell.) If there were no matches, highlight the pattern in the color specified by the `MATCHFAIL` variable (default is bright red).

Command Line Editing

Key	Meaning
<Enter>	Accept the command as typed. Move to the end (if not there already) and carriage return to a new line.
<Home>	Beginning of command line.
<End>	End of command line.
›	Up one command in the history list. Each time it's pressed, it displays the preceding entry in the history list. Any ''!...'' or ''%...'' history references in the original text will have been fixed up unless it was the immediately preceding command and it had one these references that failed. If already at the first entry, the command line is highlighted in bright red.
fl	Down one command line in the history list. If already at the latest entry, the command line is highlighted in bright red.
‹	One character left.
fi	One character right.
Ctrl-<Home>	Move to the upper-leftmost character in the current screenful if the command is long enough that it actually wraps across several screens.
Ctrl-<End>	Move to the lower-rightmost character in the current screenful.
Ctrl-›	Up one row on the screen if the command is long enough that it runs over a row.
Ctrl-fl	Down one row on the screen.
Ctrl-‹	Backup word.
Ctrl-fi	Forward word.
Alt-<Home>	Delete all preceding characters on the command line.
Alt-<End>	Delete all following characters.
Alt-›	Delete up one row on the screen if the command runs over a row.
Alt-fl	Delete down one row.
Alt-‹ Ctrl-<Backspace>	Delete preceding word.
Alt-fi	Delete following word.
<Insert>	Toggle insert/overstrike mode. When inserting, the cursor is slightly thicker.
Ctrl-<Insert>	Insert the next word from the last section of deleted text. When it reaches the end of the deleted text, it starts over.
Alt-<Insert>	Insert all the rest of the previously deleted text.
<PageUp>	Backup to one past the last history reference. (Repeatedly typing <PageUp> <Enter> is a convenient way of picking up a whole series of commands from history.)
<PageDown>	Forward to the newest entry in the history list.
<Esc>	Clear the command line.

Note: Users lacking separate arrow keys must press Ctrl-Shift instead of Alt.

History Recall

Command	Meaning
!!	Last command
!^	First argument word of last command
!$	Last word of last command
!*	All arguments of last command
!**n**	Command **n**
!-**n**	**n**th command from the last
!**str**	Last command starting with **str**
!?**str**?	Last command containing **str**
%**str1**%**str2**%	Substitute **str2** for **str1** in last command. (Used only at the beginning of a line.)

History recall allows a previous statement to be quickly recalled and re-executed. It's a very fast shorthand, especially in the edit/compile/debug loop or to fix a typo. For convenience, "`!`" is taken as an ordinary character if followed by white space, "=", "~" or "`(`".

If you want, you can choose different characters to introduce history references by changing the `histchars` variable.

Command Completion

Command completion lets you type just part of a previous command and have the shell fill in the rest. As with filename completion, if no match is found, color highlighting as defined by the `MATCHFAIL` variable (default is bright red) will be used. Consecutive depressions cause the search to continue on back through the history list.

Key	Meaning
Ctrl-<Enter>	Search for the last command that starts with the characters in the previous word.
Alt-<Enter>	Search for the last command that contains the characters in the previous word anywhere on the command line.

Quoting

String	Meaning
’...’	Literal character string. Only do history substitutions.
"..."	Single word. Typically used if there are embedded blanks or wildcard characters you want treated as ordinary. Has no effect on command or variable substitutions: they’re still done.
‘...‘	Command substitution. Evaluate the string as a separate command and substitute its output back onto the command line. Newlines are turned into spaces and Ansi escape sequences (for highlighting, etc.) are filtered out.
^	Quote just the next character. Use to remove any special meaning from the next character, to specify a character by its binary value or to specify one following non-printable characters. If the NewLine character at the end of a line is quoted this way, it’s treated as ordinary white space. (You can choose a different escape character by changing the escapesym variable.)

Escape Sequences

String	Meaning
^a	Audible alert (bell)
^b	Backspace
^f	Form Feed
^n	New Line
^r	Carriage Return
^t	Tab
^v	Vertical Tab
^^	Single escapesym character

Variable Substitution

Reference	Meaning
$**var** ${**var**}	value of variable **var**
$**var**[<expr>] ${**var**[<expr>]}	value of **var**, indexed by an arbitrarily complex expression
$#**var** ${#**var**}	Number of words in **var**
$?**var** ${?**var**}	1 if **var** exists; 0 otherwise
$**procname**(<argument list>)	Procedure reference, used as a variable substitution. The arguments are expressions, separated by commas. Any value returned by the procedure will be substituted in place.
$<	Pseudo-variable result of reading one line from stdin each time it's evaluated
$*	Same as $argv
$0 .. $9	Same as $argv[0] .. $argv[9]

Variable substitution is typically used to pass the value of a variable as an argument to a command. For example: `cl -AS -G2 -Zi $a.c`

Substitution Modifiers

Operator	Meaning
:**n**	**n**th word
:#	Count the number of words
:^	Word number 1, counting from 0
:$	Last word
:%	Word matched by a !?**str**? history search
:**n**-**m**	**n**th through **m**th words
:-**n**	0 through **n**th words
:**n**-	**n** through next-to-last words
:**n***	**n** through last word
:*	1 through last word
:q	Single quote each word
:s/**str1**/**str2**/	Substitute **str2** for **str1** and then reparse into words. Match failures are considered to be errors unless ignoreerrors == 2.
:S/**str1**/**str2**/	Substitute **str2** for **str1** but leave it as a single string. Also, failure to match is not considered an error.
:&	Repeat last substitution
:g...	Global editing: apply the edit operation everywhere it matches, not just the first occurrence.
:x	Treat each word as a string and break it up into words
:p	Print the substitution but don't execute the statement. (Ignored except in history substitutions.)

Substitution modifiers can be applied to any command, variable or history substitution. Also, any number in a row can be applied, as desired.

Pathname Editing on x\y\z.c

Operator	Name	Description	Result
:h	head	Directory containing	x\y
:r	root	Path w/o .ext	x\y\z
:t	tail	Simple filename	z.c
:b	base	Filename w/o .ext	z
:e	ext	.ext w/o the ".″	c
:f	fullpath	Fully-qualified name	d:\bob\x\y\z.c

Pathname editing can be applied to any command, variable or history substitution. Also, any number of pathname editing operations can be applied, one after the other, to a given substitution.

Example: `echo $path:gt`

Environmental Variables Recognized

Name	Default	Use
`ADDITIONS`	Bright Green	Lines added found by `diff`.
`CDPATH`		List of directories to search for the subdirectory specified as the new current directory.
`COLORS`	White on Black	Normal screen colors.
`COMSPEC`		Usually, this is the pathname of `cmd.exe`. If you set it to point to the C shell, e.g., so other programs will invoke the C shell rather than `cmd.exe`, the C shell will try to look through the search path for `cmd.exe` if it needs to run a `.cmd` file.
`CSHOPTIONS`		Default set of command line options to be pasted ahead of any other command line options passed to `csh.exe`.
`DELETIONS`	Bright Red	Lines deleted found by `diff`.
`DIRECTORIES`	Bright	Directories listed by `ls`.
`DRIVEMASK`		Used by `du.exe`, `pwd.exe` and `vl.exe` to limit the default list of drives it will report on. Written as a list of alphabetic characters representing the drives you want listed; ranges are allowed. If you don't define this variable, all drives beginning with `C:` are normally reported.
`DUPLICATES`	Green	When filename completion matches matches more than one name.
`ESCAPESYM`	^	Character to be interpreted as a literal escape character. Placed in the environment only if it is not the default circumflex.
`FOREIGNFILES`	Bright Red	Filetypes in a `tar` file that have no counterpart on OS/2.
`HIGHLIGHT`	Bright	Current disk or directory.
`HOME`		Home directory (default is the initial current directory.)
`LSOPTIONS`		Default set of command line options to be pasted ahead of any other command line options passed to `ls.exe`.
`MATCHFAIL`	Bright Red	When filename or command completion doesn't match anything.

Environmental Variables Recognized

Name	Default	Use
`MIXEDCASEDRIVES`		List of drives and UNC names for which `ls.exe`, `pwd.exe`, wildcarding and the fullname and current directory functions should report filenames in mixed case rather than all lower case.
`MOREEOF`	Green	End or Top of File in `more`.
`MOREERROR`	Bright Yellow	Unrecognizable command to `more`.
`MOREFILLIN`	White	User response to `more` prompt.
`MOREPROMPT`	Green	Prompt line in `more`.
`MORETOPMEM`	Bright Yellow	Top of Memory message from `more`.
`NETWORKBUG`		If `NETWORKBUG` = 1, the shell and all the utilities will read directory entries only one-at-a-time as a workaround for a networking bug. (Not used on NT.)
`PATH`		Search path for executable files.
`PROMPT1`	`$@ $CDISK%`	Primary command prompt template.
`PROMPT2`	`$@ $CDISK?`	Continuation line prompt template.
`RADIX`	16	Default radix used by `more.exe` and other utilities when displaying binary data.
`SHELL`		Always set to the pathname of the Hamilton C shell `csh.exe` file.
`SWITCHCHARS`	`-/`	Characters that can be used as option introducers for the shell and utilities.
`SYSTEMDIRS`	Bright Green	Used by `ls.exe` for directories with the System bit on.
`SYSTEMFILES`	Green	Used by `ls.exe` for files with the System bit on.
`TABS`	8	Used by `more.exe`, `head.exe`, `tail.exe` and `tabs.exe` to tell them how many character positions there are between tab stops.
`TAPE`	`\\.\tape0`	Used by `mt.exe` under Windows NT to specify the pathname of the default tape drive.

Environmental Variables Recognized

Name	Default	Use
`TZ`		Used by `tar.exe` under OS/2 to tell it how to convert between local time and GMT. (Under Windows NT, the system keeps track of the difference between local time and GMT, so the `TZ` variable is not needed.) The `TZ` variable should be of the form of a three-letter timezone, e.g., EST, followed by a signed number giving the difference in hours between GMT and local time, followed by an optional daylight savings timezone. Examples are `EST5EDT` in New York or `PST8PDT` in California.

Environmental variables are passed to any child processes or screens you create by invoking an external utility or application. When Hamilton C shell starts up it looks for the ones shown here to be defined in the environment it inherits. `HOME`, `PATH` and `SHELL` are special: if they're not already defined, then the shell creates them.

On Windows NT, environmental variables are ***not*** case-sensitive, so, e.g., `Path` and `PATH` refer to the same thing.

Predefined Process-Wide Variables

Name	Default	Use
$		A synonym for the **processid** variable.
bsdhistory	0	By default, "`!!`" is the immediately preceding command and "`!-1`" is the one before that. Setting **bsdhistory** = 1 makes them the same.
cdpath	null	Same as the `CDPATH` environmental variable, broken into words.
escapesym	`^`	Character to be interpreted as a literal escape character.
histchars	`!%`	Characters which introduce long-form and short-form history references, respectively.
home		Same as the `HOME` environmental variable.
NTVersion		Build number of the NT system on which the shell is running, e.g., 340 for the October Beta. (Not defined on OS/2.)
os2version		Version number of the OS/2 system on which the shell is running. (Not defined on NT.)
path		Same as the `PATH` environmental variable, broken into words.
processid		The unique process ID assigned by the OS/2 or NT kernel to this copy of the C shell.
prompt1	`$@ $CDISK%`	Same as `PROMPT1` environmental variable.
prompt2	`$@ $CDISK?`	Same as `PROMPT2` environmental variable.
savehist	0	Save the history contents into `history.csh` in the **home** directory.
shell		Same as `SHELL` environmental variable.
WinVersion		Version number of the Windows API on which the shell is running, e.g., 3.1 for the October Beta. (Not defined on OS/2.)

Any user-defined variables (with the exception of the parameter names of a procedure or those variables explicitly defined as local) are shared globally by all threads.

Predefined Per-Thread Variables

Inherited from the parent thread:

Name	Default	Use
argv		Any argument words passed to the shell or to a `.csh` batch file.
bsdargv	0	If set, `$argv[0]` contains the name of the script, just as it would under the original Berkeley C shell; by default, `$argv[0]` is the first argument word.
cdhome	0	If set, ``cd'' with no argument is the same as ``cd $home''; default is to simply print the current directory name.
cdisk		Current disk, not including colon.
CDISK		Same as cdisk, but in upper case.
chgdisk	0	If set, `cd` automatically does a `DosSelectDisk` if the path is on another disk.
cwd		Full pathname of the current directory.
gotowindow	50	Number of statements a `goto` can jump over (when not inside a nested block) without being considered an error.
nohashing	0	Determine how/whether path hashing is done: 0 means full path hashing of the directories on the search path; 1 means turn off hashing completely; 2 means hash only the directories which do not depend on the setting of the current directory.
noclobber	0	If True, don't allow redirection to overwrite an existing file unless the ``!'' override is given.
noglob		A synonym for the nowild variable.
nonohidden	0	Determine whether wildcarding will match against hidden files: 0 means don't match hidden files; 1 means hidden files will be found.
nonomatch	0	Determine the response to a wildcard that doesn't match anything: 0 means it's an error; 1 means pass it through to the application; 2 means simply discard it.

Predefined Per-Thread Variables

Inherited from the parent thread:

Name	Default	Use
nonovar	0	Determine the response to a reference to a non-existent variable, procedure or alias. Same encoding as nonomatch.
nowild	0	If True, turn off filename wildcarding.
nullwords	0	Determines whether an array index off the end of a list is an error (0) or returns a null word (1).
precision	6	Number of decimal places to print when displaying floating point values.
tailstatus	0	Determines whether the status variable will reflect the reflect the return code from the leftmost or rightmost stage of a pipeline: 0 means leftmost; 1 means rightmost.
verbose	0	If True, print out all available information when reporting errors.

Each new thread is initialized with default on-interrupt processing (a forced exit), and null search and replace strings. `echoinput`, `ignoreerrors` and `interactive` are initialized for the main thread based on command-line options.

Each thread also has its own independent notion of current disk and current directories, initially inherited from its parent.

Predefined Per-Thread Variables

Always initialized to the default for each new child thread:

Name	Default	Use
*		A synonym for the *argv* variable.
<		A synonym for the *getline* variable.
@		A synonym for the *stmtnumber* variable.
child	0	Identification number of the last child process spawned.
echoinput	0	Copy the input to stdout as it's read.
eofgetline	0	Pseudo-variable to indicate if the last reference to getline encountered an end-of-file condition.
getchar		Read one character from stdin without echoing. If stdin is tied to the keyboard, outboard keys are returned as a two-character string.
getline		Read one line from stdin pseudo-variable. If stdin is tied to the keyboard, keystrokes are echoed as they're typed.
history	0	Number of statements to remember on the history list; 0 turns off the history mechanism. (If the thread is interactive, history is automatically set to 100.)
ignoreeof	0	If True, don't exit at EOF on stdin; insist on an `exit` command.
ignoreerrors	0	Determine whether execution should continue if an error occurs: 0 means the thread exits; 1 (the default for an interactive thread) means exit from loops or procedures and try to read a new command; 2 means ignore all errors.
ignorestatus	1	If True, a non-zero status code from a child process is ignored. Otherwise, it's an error.
interactive	0	If True, prompt for input.
irqmask	0	Determines whether interrupts are enabled (0), deferred until the mask is cleared again (1) or ignored (2).
scriptname		Name of the C shell script file being executed, if any.

Predefined Per-Thread Variables

Always initialized to the default for each new child thread:

Name	Default	Use
status	0	Exit code of the last child process.
stmtnumber	1	Autoincremented statement number used with the history list and in prompting.
threadid		Thread id of the currently executing thread.

Predefined Variables
Sorted by Name

Legend for this table:

`COURIER`	A **setenv** environmental variable. Environmental variables are passed to any child processes or screens you create by invoking an external utility or application. When Hamilton C shell starts up it looks for the ones shown here to be defined in the environment it inherits; if they're not already defined, the shell creates them.
bold	A **set** variable shared by all threads: if one makes a change, all will see it.
normal	Each thread gets its own copy but the initial value is inherited from its parent.
italics	Each thread gets its own copy but the initialization is always to a defined value.

Name	Default	Use
$		A synonym for the **processid** variable.
*		A synonym for the *argv* variable.
<		A synonym for the *getline* variable.
@		A synonym for the *stmtnumber* variable.
`ADDITIONS`	Bright Green	Lines added found by `diff`.
argv		Any argument words passed to the shell or to a `.csh` batch file.
bsdargv	0	If set, `$argv[0]` contains the name of the script, just as it would under the original Berkeley C shell; by default, `$argv[0]` is the first argument word.
bsdhistory	0	By default, "`!!`" is the immediately preceding command and "`!-1`" is the one before that. Setting **bsdhistory** = 1 makes them the same.
cdhome	0	If set, "`cd`" with no argument is the same as "`cd $home`"; default is to simply print the current directory name.
cdisk		Current disk, not including colon.
CDISK		Same as cdisk, but in upper case.
`CDPATH`		List of directories to search for the subdirectory specified as the new current directory.

Predefined Variables
Sorted by Name

Name	Default	Use
cdpath	null	Same as the `CDPATH` environmental variable, broken into words.
chgdisk	0	If set, `cd` automatically does a `DosSelectDisk` if the path is on another disk.
child	0	Identification number of the last child process spawned.
`COLORS`	White on Black	Normal screen colors.
`COMSPEC`		Usually, this is the pathname of `cmd.exe`. If you set it to point to the C shell, e.g., so other programs will invoke the C shell rather than `cmd.exe`, the C shell will try to look through the search path for `cmd.exe` if it needs to run a `.cmd` file.
`CSHOPTIONS`		Default set of command line options to be pasted ahead of any other command line options passed to `csh.exe`.
cwd		Full pathname of the current directory.
`DELETIONS`	Bright Red	Lines deleted found by `diff`.
`DIRECTORIES`	Bright	Directories listed by `ls`.
`DUPLICATES`	Green	When filename completion matches matches more than one name.
`DRIVEMASK`		Used by `du.exe`, `pwd.exe` and `vl.exe` to limit the default list of drives it will report on. Written as a list of alphabetic characters representing the drives you want listed; ranges are allowed. If you don't define this variable, all drives beginning with `C:` are normally reported.
echoinput	0	Copy the input to stdout as it's read.
eofgetline	0	Pseudo-variable to indicate if the last reference to *getline* encountered an end-of-file condition.
`ESCAPESYM`	^	Character to be interpreted as a literal escape character. Placed in the environment only if it is not the default circumflex.

Predefined Variables
Sorted by Name

Name	Default	Use
escapesym	`^`	Same as the `ESCAPESYM` environmental variable.
`FOREIGNFILES`	Bright Red	Filetypes in a `tar` file that have no counterpart on OS/2 or NT.
getchar		Read one character from stdin without echoing. If stdin is tied to the keyboard, outboard keys are returned as a two-character string.
getline		Read one line from stdin pseudo-variable. If stdin is tied to the keyboard, keystrokes are echoed as they're typed.
gotowindow	50	Number of statements a `goto` can jump over (when not inside a nested block) without being considered an error.
`HIGHLIGHT`	Bright	Current disk or directory.
histchars	`!%`	Characters which introduce long-form and short-form history references, respectively.
history	0	Number of statements to remember on the history list; 0 turns off the history mechanism. (If the thread is interactive, history is automatically set to 100.)
`HOME`		Home directory (default is the initial current directory.)
home		Same as the `HOME` environmental variable.
ignoreeof	0	If True, don't exit at EOF on stdin; insist on an `exit` command.
ignoreerrors	0	Determine whether execution should continue if an error occurs: 0 means the thread exits; 1 (the default for an interactive thread) means exit from loops or procedures and try to read a new command; 2 means ignore all errors.
ignorestatus	1	If True, a non-zero status code from a child process is ignored. Otherwise, it's an error.
interactive	0	If True, prompt for input.

Predefined Variables
Sorted by Name

Name	Default	Use
irqmask	0	Determines whether interrupts are enabled (0), deferred until the mask is cleared again (1) or ignored (2).
`LSOPTIONS`		Default set of command line options to be pasted ahead of any other command line options passed to `ls.exe`.
`MATCHFAIL`	Bright Red	When filename or command completion doesn't match anything.
`MIXEDCASEDRIVES`		List of drives and UNC names for which `ls.exe`, `pwd.exe`, wildcarding and the fullname and current directory functions should report filenames in mixed case rather than all lower case.
`MOREEOF`	Green	End or Top of File in `more`.
`MOREERROR`	Bright Yellow	Unrecognizable command to `more`.
`MOREFILLIN`	White	User response to `more` prompt.
`MOREPROMPT`	Green	Prompt line in `more`.
`MORETOPMEM`	Bright Yellow	Top of Memory message from `more`.
`NETWORKBUG`		If `NETWORKBUG` = 1, the shell and all the utilities will read directory entries only one-at-a-time as a workaround for a networking bug. (Not used on NT.)
nohashing	0	Determine how/whether path hashing is done: 0 means full path hashing of the directories on the search path; 1 means turn off hashing completely; 2 means hash only the directories which do not depend on the setting of the current directory.
noclobber	0	If True, don't allow redirection to overwrite an existing file unless the "`!`" override is given.
noglob		A synonym for the nowild variable.
nonohidden	0	Determine whether wildcarding will match against hidden files: 0 means don't match hidden files; 1 means hidden files will be found.

Predefined Variables
Sorted by Name

Name	Default	Use
nonomatch	0	Determine the response to a wildcard that doesn't match anything: 0 means it's an error; 1 means pass it through to the application; 2 means simply discard it.
nonovar	0	Determine the response to a non-existent variable, procedure or alias. Same encoding as nonomatch.
nowild	0	If True, turn off filename wildcarding.
NTVersion		Build number of the NT system on which the shell is running, e.g., 340 for the October Beta. (Not defined on OS/2.)
nullwords	0	Determines whether an array index off the end of a list is an error (0) or returns a null word (1).
os2version		Version number of the OS/2 system on which the shell is running. (Not defined on NT.)
`PATH`		Search path for executable files.
path		Same as the `PATH` environmental variable, broken into words.
precision	6	Number of decimal places to print when displaying floating point values.
processid		The unique process ID assigned by the OS/2 or NT kernel to this copy of the C shell.
`PROMPT1`	`$@ $CDISK%`	Primary command prompt template.
prompt1		Same as the `PROMPT1` environmental variable.
`PROMPT2`	`$@ $CDISK?`	Continuation line prompt template.
prompt2		Same as the `PROMPT2` environmental variable.
`RADIX`	16	Default radix used by `more.exe` when displaying binary data.
savehist	0	Save the history contents into `history.csh` in the **home** directory.
scriptname		Name of the C shell script file being executed, if any.
`SHELL`		Always set to the pathname of the Hamilton C shell `csh.exe` file.

Predefined Variables
Sorted by Name

Name	Default	Use
shell		Same as the `SHELL` environmental variable.
status	0	Exit code of the last child process.
stmtnumber	1	Autoincremented statement number used with the history list and in prompting.
`SWITCHCHARS`	`-/`	Characters that can be used as option introducers for the shell and utilities.
`SYSTEMDIRS`	Bright Green	Used by `ls.exe` for directories with the System bit on.
`SYSTEMFILES`	Green	Used by `ls.exe` for files with the System bit on.
`TABS`	8	Used by `more.exe` to tell it how many character positions there are between tab stops.
tailstatus	0	Determines whether the status variable will reflect the reflect the return code from the leftmost or rightmost stage of a pipeline: 0 means leftmost; 1 means rightmost.
`TAPE`	`\\.\tape0`	Used by `mt.exe` under Windows NT to specify the pathname of the default tape drive.
threadid		Thread id of the currently executing thread.
`TZ`		Used by tar.exe under OS/2 to tell it how to convert between local time and GMT. (Under Windows NT, the system keeps track of the difference between local time and GMT, so the `TZ` variable is not needed.) The `TZ` variable should be in the form of a three-letter timezone, e.g., EST, followed by a signed number giving the difference in hours between GMT and local time, followed by an optional daylight savings timezone. Examples are `EST5EDT` in New York or `PST8PDT` in California.
verbose	0	If True, print out all available information when reporting errors.
WinVersion		Version number of the Windows API on which the shell is running, e.g., 3.1 for the October Beta. (Not defined on OS/2.)

Built-in Procedures

Name	Function
Filename Functions:	
childpath(p, c)	Test whether filename c could be in a subdirectory of p. (Does not test for actual existence of either c or p.)
driveno(p)	Drive number implied by pathname p.
fullpath(p)	Fully resolve pathname p.
samepath(a, b)	Test whether two filenames, a and b, point to the same file.
Math Functions:	
abs(x)	Absolute value
acos(x) asin(x) atan(x) cos(x) sin(x) tan(x)	Trigonometric functions
cosh(x) sinh(x) tanh(x)	Hyperbolic functions
ceil(x)	Ceiling (lowest integer ‡ x)
exp(x) log(x) log2(x) log10(x)	Exponential and logarithmic functions
floor(x)	Floor (highest integer £ x)
round(x)	floor(x + 0.5)
sqrt(x)	Square root

Built-in Procedures

String Functions:

Name	Function
ansi(*color*)	Return an ANSI escape sequence which will produce the specified screen colors. If the argument is ambiguous, e.g., simply "bright," it's taken as a modification of the default colors specified by the COLORS variable.
char(*i*)	Return the character corresponding to the numeric value *i*.
code(*c*)	Return the numeric encoding of the character *c*.
concat(*a*, *b*, ...)	Concatenation of a series of strings.
isinteger(*x*)	Test whether *x* is an integer. (Remember that null strings and strings consisting only of white space are considered equal to 0.)
isnumber(*x*)	Test whether *x* is a number.
printf(*fmt*, ...)	Perform C language-style print formatting, returning the result as a string. (See table of format conversions on following page.)
reverse(*s*)	Reverse the order of characters in *s*.
strindex(*a*, *b*)	Return the position in *a* of the first occurrence of *b*. (0 means *b* was not found.)
strlen(*s*)	Number of characters in *s*, represented as a string
substr(*s*, *b*, *i*)	Substring of length *i* beginning at *b*-th character of *s*. (*i* = 0 means "rest of *s*")
upper(*s*) lower(*s*)	Translate a string to all upper- or all lower-case

Printf Format Directives

These argument formats are recognized by printf:

%c	Single character.
%d	Decimal number.
%e	[-]d.dddddde[+-]ddd
%f	[-]ddd.dddddd
%g	%e or %f formatting, whichever is shorter.
%o	Unsigned octal number.
%s	String.
%x	unsigned hexadecimal number.
%%	Literal % character.

Additional parameters may lie between the % and the control letter:

-	Left-justify expression in its field.
width	Pad field to this width as needed; leading 0 pads with zeros.
.prec	Maximum string width or digits to right of decimal point.

Built-in Procedures

Built-in Utilities

Command	Function
cd	Change working directory. Optionally, change disk.
chcp	Change code page. (Not implemented on NT.)
chdir	A synonym for cd.
cls	Clear the screen.
dirs	Print the directory stack.
echo	Echo arguments to stdout.
eval	Defer parsing of the word list until the evaluation phase.
hashstat	Print path hash statistics.
heapstat	Print heap usage statistics.
history	Display the history list.
kill	Kill background activity.
popd	Pop directory stack.
ps	List process and thread status.
pushd	Push a new current directory on the directory stack or exchange the top two items.
rehash	Rehash the path directories.
rotd	Rotate the directory stack.
sleep	Sleep for a specified period.
source	Read commands from a file.
unhash	Turn off path list hashing.
verify	Turn on write verification mode.
wait	Wait for children to complete.

All built-in utilities self-document with the -h option.

External Utilities

Command	Function
binedit	Binary edit.
cat	Concatenate files.
chmod	Change mode bits of file.
cp	Copy files or directories.
cron	Run commands at specified times.
cut	Cut out selected fields of each line of text.
des	Encrypt or decrypt data using the Data Encryption Standard (DES). Due to U.S. Federal Law, this utility cannot be exported outside the U.S. or Canada.
date	Display the date and time.
diff	Compare files or directories.
dim	Discard any ansi escape sequences in the input stream.
dskread	Read raw sectors from a disk.
dskwrite	Write raw sectors to a disk.
du	Display disk usage.
fgrep	Fast string search (fast grep) of text files.
grep	Regular expression pattern search of text files.
head	Copy the first few lines or bytes of a file to stdout.
label	Read/Write the volume label.
ls	List directory contents.
markexe	Mark an `.exe` file to indicate what sort of application it is. (OS/2 only.)
mkdir	Make a new directory.
more	A better more utility. (Able to search forward or backward or to a specific line.)

External Utilities

Command	Function
moreh	A large model version of more.
mt	Manipulate the tape device. (NT only.)
mv	Move files or directories.
newer	Test whether first file is newer than the others.
older	Test whether first file is older than the others.
patchlnk	A (very) special-purpose utility to patch a bug in the Microsoft linker on OS/2.
pwd	Print the working directories.
rm	Remove files or directories.
rmdir	Remove directories.
sed	Stream editor.
setrows	Set or report the number of rows in the display window. (OS/2 only.)
split	Split a large file into equal-sized chunks.
startwin	Start Win3.x applications under OS/2 2.x.
strings	Extract ASCII strings from a file.
sum	Checksum the contents of a file.
tabs	Expand/Unexpand tabs.
tail	Copy the last few lines or bytes of a file to stdout.
tar	Read/Write UNIX tape archive (tar) format files.
tee	Pipe fitting.
touch	Update the time-stamp on a file.
tr	Translate characters.
uniq	Report unique lines in text files.

External Utilities

Command	Function
ver	Display the current system and Hamilton C shell version numbers.
vol	Display the disk volume label.
wc	Count lines, words and characters.
whereis	Tell which PATH directory a given executable is in.
xd	Hex dump a file to stdout.

All external utilities except whereis self-document with the -h option. Any external utility may be renamed simply by renaming the executable file.

Popular Aliases

Command	Definition	Function
app	cat >>	Append to a file.
beep	eval echo -n ^a	Beep sound.
breaksw	break	Older BSD4.3 name for a break statement inside a switch.
cdd	cd +c	Change the directory and disk in one command.
copy	local s ; @ s = $nowild ; @ nowild = 1 ; safecopy	Invoke the standard IBM/MS copy command with shell wildcarding turned off so copy will work sensibly.
date	dt	Display the current time and date.
del	local s ; @ s = $nowild ; @ nowild = 1 ; safedel	Invoke the del command with shell wildcarding turned off so ‘‘del *.*’’ will still produce the ‘‘Are you sure?’’ message.
di	diff -b!	Run the diff command, ignoring white space differences and using color to highlight the changes.
dir	cmd /c dir	Invoke the cmd.exe dir command. (Alternately, you may wish to intercept dir the same way copy or del is done, turning off shell wildcarding before running the command.)
duc	du -c	Disk usage for the current disk only.
dumphist	history -s > ~\history.csh	Dump out the history list.
endif	end	Older BSD4.3 name for the end of an if statement.
endsw	end	Older BSD4.3 name for the end of switch statement.
erase	del	Alternate name for del.
f	fgrep	Quicker name for fgrep.
fn	fgrep -n	Fgrep and print line numbers.

Popular Aliases

Command	Definition	Function
g	grep	Quicker name for grep.
h	history	Quicker name for history.
help		Under OS/2, invoke the IBM/Microsoft help command. Under NT, this is a procedure using winerror (in the samples directory) to print the message corresponding to the return code from the last command.
home	cdd ~	Change to the home disk and directory.
ld	ls -a +D -.	List only the subdirectories.
ll	ls -L	List directories, long format.
loadhist	source -n ~\history.csh	Load the history list without executing any of it.
md	mkdir	Make directories.
mi	moreh -i	Quick interactive startup of your favorite version more. Clears the screen when it starts up and doesn't just exit if there's less than a screenful.
mih	moreh -i	Huge interactive more.
mis	more -i	Small interactive more.
pause	echo -n Press any key when ready ... ; @ getchar ; echo	Pause, waiting for any keystroke or character from stdin.
q	exit	Exit the C shell
rd	rmdir	Remove empty directories.
ren	rename	Another name for the rename alias.
rename	local s ; @ s = nowild ; @ nowild = 1 ; saferename	Invoke the standard IBM/MS rename command with shell wildcarding turned off so the rename will work sensibly.

Popular Aliases

Command	Definition	Function
start	cmd /c start	Start a new session via the cmd.exe start command.
type	cat	Copy files to stdout.
vol	vl	List volume labels.
w	(wait; beep)	Wait for background processes and beep.
xcopy	local s ; @ s = nowild ; @ nowild = 1 ; safexcopy	Invoke the standard IBM/MS xcopy command with shell wildcarding turned off so xcopy will work sensibly.

Popular Aliases

Help for Hamilton C shell

csh: Startup the Hamilton C shell

Usage: csh [-!BbefFhHiKLlNnstuXYZ-] [-IO] [+IO] [-cC command]
[arguments ...]

Options:

-!	Ignore errors: Continue execution even if a command terminates abnormally. (Implied by interactive.)
-B	Berkeley mode for better compatibility with the BSD4.3 C shell for commands typed, not just those read from a script starting with #!/bin/csh.
-b	No Berkeley compatibility mode at all, not even on scripts starting with #!/bin/csh.
-c	Execute the command following on the command line, then exit. (Implies not interactive.)
-C	Normally, this means immediately run the command on the command line, then continue with normal startup and processing of stdin. But for compatibility with cmd.exe, the -X option may be used to toggle this to have the same meaning as -c but with the initial greeting and copyright notice suppressed.
-e	Echo the raw input to stdout.
-f	Fast startup: Don't look for a startup.csh file.
-F	*Faster* startup: Don't look for a startup.csh file and don't hash the path directories.
-H	Heapstats. At some penalty in performance, this causes the shell to keep a count of the number and total size of all objects allocated or freed from the heap. Statistics can be displayed using the heapstat command.
-i	Interactive (even if stdin appears to be a file or a pipe): Prompt for input and show the result of history substitutions.
-I	Start with command-line editing in insert mode.
+I	Start every command in insert mode.
-K	Do a fast startup, skipping both the login.csh and startup.csh files, run the command on the command line, and exit.
-L	Login shell: Look for login.csh and logout.csh and do history save at exit if savehist == 1.
-l	same as -L.

-N	No space after the last filename generated by filename completion. (It's convenient to set this with CSHOPTIONS.)
-n	No execution: Parse commands looking for syntax errors but don't execute them.
-O	Start with command-line editing in overstrike (default) mode.
+O	Start every command in overstrike mode.
-s	Read and execute a single line from stdin. (Implies not interactive.)
-t	Suppress callstack tracing. Error messages will not include a dump of the callstack but performance will be improved slightly.
-u	Reads from devices or pipes should be unbuffered. (Intended to all the shell to be used over a comm line.)
-X	Toggle between the normal meaning of -C and a cmd.exe-compatible interpretation. You can set this with the CSHOPTIONS environmental variable in combination with setting COMSPEC to point to the C shell to allow programs that use /C to pass commands to cmd.exe to be used with the C shell instead.
-Y	No interrupts. Don't install an interrupt handler during startup. (Ignored under Windows NT.)
-Z	Very special purpose: Don't bump the maximum file handle count during shell initialization. Use this option under OS/2 as a workaround if you encounter an application that fails if it inherits a larger limit. This option only works from the Start Programs or Group menus, not the command line. Under NT, this option affects only the limit on the number of open file handles on a FAT filesystem; it has no effect on HPFS or NTFS files.
-h	Help.
--	End of options.

(If preferred, the slash, "/," may be used in place of a minus to introduce options.)

Help for the Utilities

binedit: Binary Edit

Usage: binedit [-ih!-] [-r replace] search file1 [file2 ...]

binedit is a very simple utility for scanning arbitrary files, looking for and optionally changing any references to the search argument to the replacement value. binedit is quite useful patching binary files, e.g., to replace all occurrences of C:\OS2 with C:\1.X.

Occurrences will be reported as hex offsets from the start of the files you name. You can use these numbers to go examine the file with IBM/MS patch.exe if you like and make the change manually.

You can also ask **binedit** to make the change, using the -r (replace) option. Nothing fancy. It just changes all occurrences to the string you specify and quits.

Neither the search nor the replace string may contain null characters, but otherwise there are no restrictions.

Options:

-h	Help. (This screen.)
-i	Ignore character case in the search argument.
-r replace	Replace any occurrences of the search string with this replacement value.
-!	Pad with nulls or truncate the replace string to be the same length as the search string. (Otherwise, it's an error if they're different lengths.)
--	End of options.

cat: Concatenate Files to Stdout

Usage: cat [-h-] [file1 file2 ...]

cat copies the files you specify, one immediately after another, onto stdout. No end-of-file or other delimiter characters are inserted between files.

Options:

-h	Help.
--	End of options.

cd: Change to a New Current Directory

Usage: cd [-chr-] [+chr] [directory]

cd works exactly like the one in cmd.exe by default: if you specify a directory, it tries to make that the current on appropriate disk but won't change your current disk. If you don't specify a directory, it reports the current setting.

If the path you specify is just 3 or more dots, **cd** will interpret that specially. Just as "`..`" means go up one level, "`...`" means up 2 levels, "`....`" means up 3 levels, etc. Note that "`...`" is also a wildcard notation for zero or more directory levels, but only if it's preceded by "\", "/", "~" or "`:`" or followed by "\" or "/", so usually there's no conflict with this notation for going up multiple directories with **cd**. The exception is when you'd like to go up multiple directories on an explicitly specified drive; in that case, you'll have to quote the word to turn off the wildcarding.

This version of **cd** also supports **CDPATH**. If you specify a path that isn't found, cd will look through the list of any directories you specify in the **CDPATH** environmental variable (or the equivalent **cdpath** shell variable) to see if the directory you want is in one of those directories. The syntax for **CDPATH** and **cdpath** is the same as for **PATH** or **path** except that the current directory need not be listed.

(See also the **dirs**, **pushd**, **popd** and **rotd** commands.)

Options:

If you set the **chgdisk** variable equal to 1 (its default is 0), **cd** will automatically change current disks if the directory is on another drive. The +c and -c options allow you to manually control this drive switching:

+c	Automatically change the current disk.
-c	Don't automatically change current disk.

If you set the **cdhome** variable to 1 (its default is 0), **cd** will change to your home directory if you specify one instead of reporting your current position. You can manually control this with the +r and -r options:

+r	Report but don't change the current directory if no destination directory is specified.
-r	Change to the home directory in this situation.

-h	Help.
--	End of options.

chcp: Report or Change the Code Page

Usage: chcp [-h-] [page]

Change to the specified OS/2 character set code page. If no page is specified, report the current active and prepared pages. (This command is not yet supported under NT.)

Options:

-h	Help.
--	End of options.

chmod: Change Mode Bits on Files or Directories

Usage: chmod [-+] [rRAHSh] pathname1 [pathname2 ...]

Options:

-r	Recursively change contents of directories.
+R	Read-only mode is set. (The file is write-protected.)
-R	Read-only mode is turned off.
+A	Archive bit is set.
-A	Archive bit is turned off.
+H	Hidden mode is set.
-H	Hidden mode is turned off.
+S	Mark as a System file.
-S	Mark as a normal user file.
-h	Help.
--	End of options.

cls: Clear the Screen

Usage: cls [-h-]

Options:

-h	Help.
--	End of options.

cp: Copy Files or Directories

Usage: cp [-filmh-] source1 [source2 ...] destination

cp can copy both files and directories. If the destination is an existing directory, the source object(s) will be copied into that directory. If more than one source object is specified, the destination must be a directory, but it need not already exist.

If a destination file already exists, it will be overwritten; this is not considered an error and no diagnostic message is given.

Options:

-f	Force read-only files to be overwritten.
-i	Interactive: ask before copying each object on the command line.
-l	Logging is on: display the name of each file or directory as it's copied.
-m	Merge sub-directories of same name in source and destination.
-h	Help.
--	End of options. (Useful if filenames start with ''-''.)

cron: Run Commands at Specified Times

Usage: cron [-asLwh-] [-o<opt>] [-d <dir>] [crontab1 crontab2 ...]

cron executes commands at times according to the crontab (.crt) files specifed. If any path specified is a directory, **cron** will run any crontab files in that directory or any subdirectory. If no files are specified, **cron** will read the crontab from stdin.

Each file is expected to contain a list of activities to be run at certain times, specified one per line. There are six fields per line, separated by white space. The first five specify:

minute	(0-59)
hour	(0-23)
day of the month	(1-31)
month of the year	(1-12)
day of the week	(0-6, 0 is Sunday)

Each of these fields may be an asterisk (meaning all legal values) or a comma-separated list of elements, where each element is either a single number or a range, given as two numbers separated by a minus. If either end of a range is omitted, it defaults to the highest/lowest legal value, as appropriate. If the days are specified in both day of the month and day of the week fields, the union of those fields is used; to specify the days by only one field, the other should be an asterisk. For example, 0 0 1 * 1 means run the command at midnight on the first of every month and also on every Monday.

The sixth field of each line is a command that is to be executed by the shell at the specified times. Any percent characters in this field (unless escaped by the escape character) are translated to carriage return/newline combinations. Only the first line (up to the first % or end of line) is passed as a command; the other lines are written to the shell as stdin. If no input lines are given, the shell will inherit stdin opened to the nul device.

If the crontab was read from a file, the shell will inherit stdout and stderr handles for writing (appending, if the file already exists) to a file created by replacing the .crt extension on the crontab file with .nnn, where nnn is the linenumber in the .crt file. (If the crontab was read from stdin, then the shell will inherit the stdout and stderr that **cron** inherited when it started.)

If a syntax or other error is encountered in a crontab, that line, but not the rest of the file will be ignored. The rationale is that **cron** should continue running even if it encounters problems; it should not be necessary to restart it just because one entry in one crontab contained an error.

By default, if the crontab is read from a file, commands will be run asynchronously, meaning that all entries that are ready to run at any given time will be quickly spawned, one right after another, without waiting for each to complete before the next is spawned. If the crontab is read from stdin, the default will be synchronous execution, meaning each must complete before the next starts; that's so any output from one won't be jumbled with that of another. These defaults can be overridden with command-line options.

Once it's started, **cron** will make an initial scan of the crontab files or directories. Once every minute after that it will look for any changes by comparing timestamps on the files.

Options:

-a	Asynchronout execution. Don't wait for one command to complete before spawning the next, even if the crontab was read from stdin.
-s	Synchronous execution. Always wait for one command to complete before spawing the next, even if the crontab was read from a file.
-L	Logging. As each command is spawned, timestamp and log it to stdout and to the output file if that's not the same as stdout.
-w	Wait 'till whole minute. If **cron** wasn't started near a whole minute (i.e., zero to three seconds past a whole minute mark), wait 'till the next whole minute before beginning to schedule **cron** events.
-o<opt>	Remainder of word contains options to be passed to the C shell with any commands. E.g., -oF means C shell should do fast startup when running commands.
-d <dir>	Output files should be created in the specified directory, not the directory where the .crt files were found.
-h	Help. (This screen.)
--	End of options.

cut: Cut out selected fields of each line of text,

Usage: cut [-hsr-] [-c<list>] [-f<list>] [-d<delims>] [file1 file2 ...]

cut can be used to select certain columns or fields from each line of input. If the -c option is used, fields are defined as specific character positions. If the -f option is used, the fields can be of variable length, each field separated from the next by the delimiter character. If no files are specified, **cut** reads input data from stdin.

A list of fields or character positions is specified as a comma-separated list of integers. The minus sign can be used to indicate a range. Here are some examples: 1,4,7; 1-3,8; -5,10 (short for 1-5,10); or 3- (short for third through last field.)

Options:

-h	Help. (This screen.)
-s	Suppress lines with no delimiters in case of -f option. Otherwise, lines with no delimiters are passed through unchanged.
-c<list>	The list specifies character positions.
-f<list>	The list specifies field numbers.
-d<delims>	The specified characters are used as the field delimiters with the -f option. Any number of delimiters can be specified. The default is the tab character.
-r	Repeated delimiters are treated as if only a single delimiter had appeared.

date: Print the Date and Time

Usage: date [-nh-]

date prints the day of the week, the date and time of day in 24-hour notation. This command is normally stored in the file dt.exe and invoked with an alias so it can be used from cmd.exe without colliding with the internal cmd.exe date function.

Options:

-n	Don't automatically append a Carriage Return/Line Feed sequence to the end of the output.
-h	Help.
--	End of options.

des: DES Data Encryption

Usage: des [-edbhnrz-] [-k key] [file1 file2 ...]

des is a filter that encrypts or decrypts data read from the files you specify, one after another, to stdout with the Data Encryption Standard (DES). If no files are given, **des** reads from stdin. If there are multiple files, they're simply concatenated as they're read.

Either -e (encrypt) or -d (decrypt) must be specified. If the key is not given on the command line with the -k option, **des** will prompt for it, suppressing echo.

Distribution of this software is controlled by U.S. Federal Law under Title 22, Code of Federal Regulations, Subchapter M, Category XIII(b). It may not be exported outside the U.S. or Canada without an export license.

Options:

-e	Encrypt.
-d	Decrypt.
-b	Electronic Code Book (ECB) mode is used. The default is to use DES Cipher Block Chaining (CBC) mode with an initial vector (IV) of all zeros. Under ECB mode, each block of 8 bytes is enciphered independently, depending only on the key. Under CBC mode, the enciphering of each block also depends on the data in the previous blocks. The default CBC mode is considered somewhat more secure.
-k key	Encryption key, typed as a simple ascii string. With an ascii key, DES ignores the low order bit of each key byte but the high order bit is set for odd parity, thus retaining the information contained in the low order bit.
-x	Hex key. The key string is a sequence of up to 16 hex characters, right padded with zeros. With a hex key, the low order bit of each byte is again ignored per the DES algorithm. This allows the use of any arbitrary 56-bit key, including bytes representing control characters that could not be typed.
-h	Help. (This screen.)

Special Interchange Options:

Not all DES implementations are the same. There are a number of early implementations in circulation that either poorly or improperly implement DES. Cipher Block Chaining may not be supported, forcing the use of the -b option. You may also discover other flaws, necessitating that you experiment with some of these special options:

-n	Suppress parity calculation on an ascii key. Just use the low-order 7 bits of each character as-is.
-r	Convert all \r\n sequences to \n on input and all \n characters to \r\n sequences on output. (Some implementors have used the C library stdio read and write routines but have forgotten to use binary mode.)
-z	Do not mark the last block with a length; just fill it with binary zeros. If you encipher, then decipher a file this way, the result will be padded with zeros out to an 8-byte boundary.

If you encounter problems exchanging encrypted data with another DES implementation, you should try all the various combinations of the -b, -n, -r and -z options. (We've seen one very poor implementation of DES that had all these flaws.)

Notes:

1. If you lose the key to a file encrypted with DES, there is no known way to decrypt it. The data is lost.

2. When choosing keys, avoid anything obvious that someone else might easily guess. E.g., don't use just your name or your date of birth or a common word. Instead, choose a key with a seemingly random mix of alphanumeric and punctuation characters.

3. No encryption system should be considered perfectly secure. Although there are no known practical methods for attacking DES, such methods may exist.

4. Encryption can only protect data that's actually encrypted. If you have copies of the clear text on your disk, anyone with access to your machine may be able to read the data. Also, even when a file is deleted, the contents may remain on your disk, accessible to anyone with knowledge of the file system.

5. If you would like to compress encrypted data, e.g., with utilities such as PKZip or ARC, compress first, then encrypt. The encryption process tends to destroy the redundancy in the data that compression programs depend on.

diff: Compare Files or Directories

Usage: diff [-bBefhiqrvw!-] [-m minmatch] [-Dstring] oldpath newpath

diff produces a list of differences between two files or directories. The working assumption is that newpath is a newer version of whatever's in oldpath. If they're files, **diff** assumes they contain ascii text but if it encounters a file containing lots of binary data, it switches to a binary comparison mode to avoid dumping gibberish to the screen.

Comparing files, **diff** looks for minimal sections of change. Each difference is shown as an add, delete or change with the appropriate line numbers or line number ranges for each file. Following are texts of the differing section. Lines that have been deleted are flagged with '<'; lines that have been added are flagged with '>'. Alternately, **diff** can be used to produce merged listings with #ifdefs for the C compiler or highlighting for quick visual scanning.

Comparing directories, the lists of files they contain all the way down through the tree are sorted, then compared. If the same filename exists in each directory tree, a quick binary comparison is normally made to give a quick yes-or-no are they different. If -r is specified, the **diff** text comparison is done recursively the whole way down the two trees.

If one of the arguments to **diff** is a file and the other is a directory, **diff** will look for and compare against a file of the same name in the directory.

Options:

-b	Blank spaces of any length compare equal. Ignore any leading or trailing white space on each line.
-B	Binary comparison even on text files.
-Dstring	Produce a merged #ifdef'ed listing, with the string being defined meaning use the older version. (Newer version is intentionally the default.)
-e	Turn off highlighting of empty lines. Byt default, if highlighting is used to show areeas of change, even empty lines will have Ansi sequences at the beginning and end so that if **more** is used to display the output with the default color stretch mode on, empty lines will still be highlighted.
-f	Force **diff**-style comparison to continue even on files that appear to contain binary data.
-i	Ignore character case.

-m minmatch	Minimum match length to insist on before resynchronizing two files. (Default is 2 lines.)
-q	Quiet: supress warnings about files containing binary data.
-r	Recursively **diff** the contents of any subdirectories.
-v	Verbose listing of all the contents of any entire subdirectory added or deleted.
-h	Help. (This screen.)
-w	White space is ignored totally.
-!	Produce merged, highlighted version. Text that's unchanged is normal, deleted text is red, new text is green.
--	End of options.

Colors:

You may set your own choices for screen colors using these environmental variables:

Name	Use	Default
COLORS	Normal screen colors	White on Black
ADDITIONS	Lines added	Bright Green
DELETIONS	Lines deleted	Bright Red

Colors recognized are black, red, green, yellow, blue, magenta (or red blue), cyan (or blue green) or white. Foreground colors may also be bright or blink. The names of the colors and the words bright, blink and on may be in either upper or lower or mixed case but the names of the environmental variables themselves must be all upper case.

Either or both the foreground and background colors may be specified; if you don't specify a value, it's considered transparent and inherits the color underneath it. **ADDITIONS** and **DELETIONS** inherit from **COLORS**.

dim: Discard any ansi escape sequences in the input stream

Usage: dim [-h-] [file1 file2 ...]

dim deletes any ansi escape sequences as it copies text from each of the files you specify onto stdout. If several files are given, they're concatenated one after another to the output.

If no files are given, **dim** reads from stdin.

Options:

-h	Help. (This screen.)
--	End of options.

dirs: List the Current Directory Stack

Usage: dirs [-h-]

Show the list of fully-qualified directory pathnames on the current directory stack, highlighting the current disk:directory pair.

Hamilton C shell maintains a stack representing the current directory and any previous current directories has saved there.

(See also the **cd**, **pushd**, **popd** and **rotd** commands.)

Options:

-h	Help.
--	End of options.

dskread: Read Raw Sectors from a Disk

Usage: dskread [-bcdhHL-] [-n name] [-N sectors] [-s size] [-t tracks]
[disk:] [<sectorlist>]

dskread copies low-level raw sectors on the disk you specify to stdout.

In conjunction with **dskwrite**, **dskread** is most useful as a quick diskette duplication utility: you can read a whole diskette image into a file with **dskread**, then write it back out with **dskwrite** to a new diskette. But it's also useful for restoring a long stream of data dumped across a series of diskettes with **dskwrite**.

The disk is given as a a single alphabetic drive letter plus a colon. If no disk is specified, the first logical drive is assumed (in most machines, this is the `a:` drive).

Operation:

dskread normally starts at cylinder 0, head 0, sector 0 and walks through the whole disk, reading all the sectors on a track under one head and then all tracks in a cylinder before repositioning the heads to the next cylinder.

dskread always reads whole sectors except when reading from a file using the -d option.

Options:

-b	Boot record. Skip sector 0 on the assumption it simply contains a standard boot record and parameter block corresponding to the particular media.
-c	Continuation. Assume the data has been split across several diskettes. After each disk has been read, prompt for the next one. If -d is specified also, the filename extension on each disk will be the disk number, i.e., 001, 002, 003, etc.
-d	Dummy filesystem. Assume the data is in a single file on the disk. If neither the -d nor the -b options is given, the disk will be read beginning with sector 0.
-h	Help. (This screen.)
-n <name>	The filename to look for if the -d (dummy filesystem) option is used. Default is '`bindata`'.
--	End of options.

Formatting:

Default is to let OS/2 or Windows NT try to determine the formatting that was used. If the disk was written in unusual format or has non-standard data in sector 0, the following options can be override OS/2 or Windows NT's attempt to guess the format:

-H	High density formatting.
-L	Low density formatting.
-N <sectors>	Sectors per track.
-s <size>	Sector size. Normally only 512 is supported but, depending on your hardware, you may also be able to create 128, 256 or 1024-byte sectors.
-t <tracks>	Number of tracks.

Sector lists:

Optionally, you can specify a list of sectors you want written. Sectors are given in

```
(cylinder, head, sector)
```

coordinates with parentheses around and commas or white space to separate the numeric values you write. Cylinders, heads and sectors are counted from zero and can be specified in decimal, hex or octal.

A single sector specified alone means just that sector. Listing several separated by commas or spaces means each one individually.

A pair of sectors joined by a hyphen means a range: all the consecutive sectors beginning with the first and running through to the second.

Specifying a negative number as an ordinate means use the highest possible value for that disk.

If any of the ordinates of a sector are omitted, they're assumed to be zero exept when it closes a range, in which case it means use the highest possible value.

Examples:

1. To duplicate a whole diskette image, autoformatting the output media if it's not already formatted:

```
% dskread a: > dskimage.a
% dskwrite -av a: < dskimage.a
```

2. To use a diskette as a serial archive media: The -d option allows you to request just enough dummy file system ''envelope'' around the otherwise arbitrary data you intend to write to ensure the disk can still be used with other OS/2 or Windows NT utilities. Here's an example writing the result of a **tar**'ing (archiving) a whole directory to a series of diskettes, again autoformatting:

   ```
   % tar -asr mydir < nul | dskwrite -avcd -n mydir a:
   ```

   It's restored with

   ```
   % dskread -dc -n mydir a: | tar -xsr
   ```

   The diskettes used do not all have to be the same density; any mix of high and low is okay. But remember that if a disk isn't already formatted, there's no way for **dskwrite** to tell whether it should be high or low density; any disks it has to format will all be formatted the same way.

   If putting the greatest possible amount of data on a diskette is more important than compatibility with other OS/2 or Windows NT utilities, you can also use the space normally devoted to the FAT file system and even the boot sector. Here is the same **tar**, written to a series of autoformatted high-density diskettes where every sector is used for data:

   ```
   % tar -asr mydir < nul | dskwrite -avcHx a:
   ```

   It's restored with

   ```
   % dskread -cH a: | tar -xsr
   ```

   Since no labeling of the data is written to the diskette, it's the user's responsibility to keep track of the order in which the disks were written and even what format was used.

dskwrite: Write Raw Sectors to a Disk

Usage: dskwrite [-abcdfFhHLTvxz-] [-Z!] [-n name] [-N sectors] [-s size] [-t tracks]
[-V volid] [disk:] [<sectorlist>]

dskwrite copies stdin to the low-level raw sectors on the disk you specify.

In conjunction with **dskread**, **dskwrite** is most useful as a quick diskette duplication utility: you can read a whole diskette image into a file with **dskread**, then write it back out with **dskwrite** to a new diskette. But it's also useful for dumping a long stream of data across a series of diskettes, formatting and filling each as much as possible before requesting the next one.

The disk is given as a a single alphabetic drive letter plus a colon. If no disk is specified, the first logical drive is assumed. (In most machines, this is the `a:` drive.)

Operation:

dskwrite normally starts at cylinder 0, head 0, sector 0 and walks through the whole disk, writing all the sectors on a track under one head and then all tracks in a cylinder before repositioning the heads to the next cylinder. You can also cause **dskwrite** to write to just specific sectors by giving it a sector list.

dskwrite always writes whole sectors and, if requested, formats whole tracks. If the last sector is only partially-used, **dskwrite** fills it out with binary zeros.

To avoid conflicts with other applications, **dskwrite** always locks a drive before actually writing to it.

Basic Options:

-a	Autoformat. If the disk appears to be unformatted, automatically do a low-level format of the entire disk.
-f	Low-level format the entire disk as data is copied to it from stdin.
-F	Low-level format the entire disk and create an empty FAT filesystem. Do not read anything from stdin.
-v	Verify. Read back and verify each write.
-h	Help. (This screen.)
--	End of options.

Options for using the disk as a serial media:

-b	Boot record. Write a simple non-system disk boot record and parameter block corresponding to the particular media into sector 0. If writing the entire disk, continue with the next sector.
-c	Continuation. If there's more data on stdin after an entire disk has been written, prompt for a new disk to be inserted and continue. If -d is specified also, the filename extension on each disk will be the disk number, i.e., 001, 002, 003, etc.
-d	Dummy filesystem. Write a boot record and a FAT filesystem into the initial sectors allocating whatever is written to a single file.
-n <name>	The filename to be given to any data written onto the disk if the -d (dummy filesystem) option is used. Default is 'bindata'.

Formatting:

Default is to let OS/2 or Windows NT try to determine how the media has been formatted and not to reformat the disk.

If the disk isn't already formatted and the -a (autoformat) option is selected or if formatting is explicitly requested with the -f or -F options, **dskwrite** will do a low-level format of the media.

dskwrite takes its specification of the format (e.g., low- or high-density) from the following sources, in decreasing priority:

1. Explicit specifications on the command line.
2. The OS/2 or Windows NT kernel's guess at the media format, assuming the disk is already formatted and not being re-formatted.
3. The format parameters in the boot sector copied from stdin, assuming neither the -b nor the -d options is given.
4. The previous format of the media, assuming it's already formatted in a standard configuration but being reformatted.
5. The highest-density format supported by the drive.

Unless you specify -b or -d, **dskwrite** will expect to copy a compatible disk image from stdin including a suitable boot sector, etc. That happens automatically when **dskread**'ing a whole disk is paired with **dskwrite**'ing a whole disk but not otherwise.

Explicit Format Specification Options:

-H	High density format.
-L	Low density format.
-N <sectors>	Sectors per track.
-s <size>	Sector size. Normally only 512 is supported but, depending on your hardware, you may also be able to create 128, 256 or 1024-byte sectors.
-t <tracks>	Number of tracks.

Other Advanced Options:

-T	Low-level format the media track-at-a-time as the disk is written. Only as much of the disk as is being written to will be formatted.
-V <volid>	The volume identifier to be written into the dummy filesystem header if -F or -d is used. (Ignored otherwise.)
-x	Force use of a non-standard format or writing of non-standard format parameters to the boot sector.
-z	Zero-length files written with -d option should still be added to the directory. (Otherwise, they're ignored.)
-Z!	Force write to a hard disk. Warning! Very risky unless you know exactly what you're doing.

Sector lists:

Optionally, you can specify a list of sectors you want written. Sectors are given in

```
(cylinder, head, sector)
```

coordinates with parentheses around and commas or white space to separate the numeric values you write. Cylinders, heads and sectors are counted from zero and can be specified in decimal, hex or octal.

A single sector specified alone means just that sector. Listing several separated by commas or spaces means each one individually.

A pair of sectors joined by a hyphen means a range: all the consecutive sectors beginning with the first and running through to the second.

Specifying a negative number as an ordinate means use the highest possible value for that disk.

If any of the ordinates of a sector are omitted, they're assumed to be zero exept when it closes a range, in which case it means use the highest possible value.

Examples:

1. To duplicate a whole diskette image, autoformatting the output media if it's not already formatted:

   ```
   % dskread a: > dskimage.a
   % dskwrite -av a: < dskimage.a
   ```

2. To use dskwrite to format a disk with an empty FAT file system, use either:

   ```
   % dskwrite -F a:
   ```

   or:

   ```
   % dskwrite -df a: <nul
   ```

3. To rewrite just the boot sector to make it a non-system disk:

   ```
   % dskwrite -b a: <nul
   ```

4. To use a diskette as a serial archive media:

   The -d option allows you to request just enough dummy file system ''envelope'' around the otherwise arbitrary data you intend to write to ensure the disk can still be used with other OS/2 or Windows NT utilities. Here's an example writing the result of a **tar**'ing (archiving) a whole directory to a series of diskettes, again autoformatting:

   ```
   % tar -asr mydir < nul | dskwrite -avcd -n mydir a:
   ```

   It's restored with

   ```
   % dskread -dc -n mydir a: | tar -xsr
   ```

   The diskettes used do not all have to be the same density; any mix of high and low is okay. But remember that if a disk isn't already formatted, there's no way for **dskwrite** to tell whether it should be high or low density; any disks it has to format will all be formatted the same way.

   If putting the greatest possible amount of data on a diskette is more important than compatibility with other OS/2 or Windows NT utilities, you can also use the space normally devoted to the FAT file system and even the boot sector. Here is the same **tar**, written to a series of autoformatted high-density diskettes where every sector is used for data:

```
% tar -asr mydir < nul | dskwrite -avcHx a:
```

It's restored with

```
% dskread -cH a: | tar -xsr
```

Since no labeling of the data is written to the diskette, it's the user's responsibility to keep track of the order in which the disks were written and even what format was used.

Non-standard Diskette Formats:

If you write anything but a standard boot record into the boot sector, you will not be able to read it with anything but **dskread** and even then only if you tell it explicitly what what format was used. Without a standard parameter block in the boot sector, the the OS/2 and Windows NT diskette device drivers just can't tell on its own how the diskette was written.

Similarly, it's possible you may be able to write a non-standard format (e.g., a sector size other than 512 bytes, etc.) depending on your hardware. But doing that may make the diskette impossible to read on other machines.

For that reason, **dskwrite** normally won't let you write a boot record that doesn't match the actual format of the diskette nor will it let you write anything other than one of the standard format. To override these checks, you must use the -x option.

Writing to a Hard Disk:

Deliberately, **dskwrite** normally only lets you write to diskettes.

Writing to a hard disk is very risky since the chance of catastrophic loss of information is so high. Also, only ESDI and SCSI disk controllers routinely map out bad blocks, so reading the disk image from one drive and writing it to another is unlikely to work if there are bad blocks on either one.

To force **dskwrite** to write to a hard disk, you must specify the -Z! option, meant to be hard to accidentally type. In effect, the -Z! option means you understand the risk you're taking.

du: List Disk Usage Statistics

Usage: du [-acdxvh] [disk1 disk2 ...]

du prints statistics showing the total, used and free space on each disk partition. If no disks are specified, **du** looks for a **DRIVEMASK** environmental variable that can be used to mask off just the drive you want reported. The **DRIVEMASK** is specified as a list of drive letters; ranges are allowed. Otherwise, all the fixed disk partitions beginning with c: are reported. The current disk's statistics are highlighted. Sizes are normally reported in binary (1000*1024) megabytes.

Option:

-a	If no disks are specified, report on all disks.
-c	If no disks are specified, report on just the current disk.
-d	Report in decimal (1,000,000) megabytes.
-x	Report in hex (1024*1024) megabytes.
-v	Give detailed cluster information.
-h	Help.
--	End of options.

Name	Use	Default
COLORS	Normal screen colors	White on Black
HIGHLIGHT	Current disk or directory.	Bright

Colors recognized are black, red, green, yellow, blue, magenta (or red blue), cyan (or blue green) or white. Foreground colors may also be bright or blink. The names of the colors and the words bright, blink and on may be in either upper or lower or mixed case but the names of the environmental variables themselves must be all upper case.

Either or both the foreground and background colors may be specified; if you don't specify a value, it's considered transparent and inherits the color underneath it. **HIGHLIGHT** inherits from **COLORS**.

echo: Echo the Arguments to Standard Output

Usage: echo [-n2h-] [text]

Echo the text to standard output (or stderr), substituting for certain character sequences preceded by the **escapesym** character. (The **escapesym** character is normally ‘^’ but can be changed with the **set** command.)

^a	Audible Alert (Bell)	^r	Carriage Return
^b	BackSpace	^t	Tab
^f	Form Feed	^v	Vertical Tab
^n	NewLine	^^	Single escapesym

The **escapesym** character may also be followed with the numeric value of the intended substitution character where the value is specified in hex (as an ′x′ followed by hex digits) or in octal.

Options:

-n	Don′t automatically append a Carriage Return/Line Feed sequence to the end of the output.
-2	Write to stderr instead of stdout.
-h	Help.
--	End of options.

eval: Parse & Evaluate at Run-Time

Usage: eval [delta] [-irfth-] [argument words]

Evaluate the words following on the command line as text to be parsed and evaluated as statements only after all the substitutions and wildcarding have been done. Optionally, under OS/2, **eval** lets you run the command at a higher or lower scheduling priority.

Command or variable substitutions aren't normally done until after statements have already been parsed and compiled into an internal form and, if it's a background statement, passed off to a background child thread.

The **eval** command is useful if either:

- You want command or variable substitutions to be recognized as any of the reserved words of the language or as an alias, or
- You want the substitutions done before a background thread is started, or
- You want to run the command at a different priority.

Options:

-i	Idle time priority. (In Unix terms, this is the ''nice'' option: if you're nice, you run at idle priority and let everything else run ahead of you.)
-r	Regular priority. (The usual initial value.)
-f	Foreground priority.
-t	Time critical priority.
delta	A signed integer amount by which the scheduling priority for this command is to be adjusted up or down. The '+' or '-' sign is required to distinguish this as an option. Priority ranges from 0 to 31 and is normally 0 when the C shell starts up or if a new priority class is chosen. Attempting to set priority below 0 or above the maximum results in priority 0 or 31 respectively.
-h	Help.
--	End of options.

fgrep: Fast string search (fast grep) of text files

Usage: fgrep [-hbcilnqsvwx-] [-f ptrnfile] [pattern] [file1 file2 ...]

fgrep does a quick, deliberately simple string search. It does not use regular expressions, but does have some **diff**-style options for ignoring upper-/lower-case differences or treating white spaces of any length as equal, etc. You can also give it a list of strings you want searched for, one per line in a pattern file or via stdin.

Each matching line is copied to stdout. If more than one file is being searched, each line is preceded by the name of file where the match occurred plus a ':' character.

Options:

-h	Help. (This screen.)
-b	Blank spaces of any length compare equal. Ignore any leading or trailing white space on each line.
-c	Just print a count of the number of lines which match.
-f ptrnfile	Read the patterns from a file.
-i	Ignore character case.
-l	Show just the names of any files containing at least one match. Show each name only once, each on a separate line.
-n	Show the line numbers of any matches.
-q	Quiet: don't show filenames where the matches occur.
-s	Read the patterns from stdin.
-v	Invert the pattern: show all lines **except** those that match.
-w	White space is ignored totally.
-x	Exact match: the whole line must match the pattern.
--	End of options.

grep: Regular expression pattern search of text files

Usage: grep [-hcilnqsv-] [-f ptrnfile] [pattern] [file1 file2 ...]

grep uses special patterns called regular expressions to filter what it reads from stdin or from any files you specify.

Regular expressions are written in this notation, in decreasing precedence:

c	Any ordinary character matches itself.
\c	Match the literal character c.
^	Beginning of line.
$	End of line.
.	Match any single character.
[...]	Match any single character in the list.
[^...]	Match any single character not in the list.
\n	Match whatever literal text the n'th tagged \(...\) expression matched.
r*	Match zero or more occurrences of r.
r1r2	Match expression r1 followed by r2.
\(r\)	Tagged regular expression. Match the pattern inside the \(...\), and remember the literal text that matched.

A regular expression pattern cannot contain Null, NewLine or CarriageReturn characters.

When typing a regular expression on the command line, remember that $, [,], ^, (and) have special meaning to Hamilton C shell. Put single quotes around the string to turn off that special meaning. Also, even inside quotes, type ^^ to mean ^ except when it immediately follows [.

Each matching line is copied to stdout. If more than one file is being searched, each line is preceded by the name of file where the matched occurred plus a ':' character.

Options:

-h	Help. (This screen.)
-c	Just print a count of the number of lines which match.
-f ptrnfile	Read the patterns from a file.
-i	Ignore character case.
-l	Show just the names of any files containing at least one match. Show each name only once, each on a separate line.

-n	Show the line numbers of any matches.
-q	Quiet: don't show filenames where the matches occur.
-s	Read the patterns from stdin.
-v	Invert the pattern: show all lines **except** those that match.
--	End of options.

hashstat: Print Path Hashing Performance Statistics

Usage: hashstat [-h-]

Summarize the recent effectiveness of the path hashing mechanism.

Hamilton C shell uses a hashing mechanism to help it decide which path directories and filetype extensions should tried first when looking for an executable file. This way, it avoids wasting time with requests to the OS/2 or NT kernels to look for files that are known to be non-existent. Sometimes, the first guess made this way is wrong (the hash says a file might exist, but it doesn't) and the shell has to try again with a different directory or extension.

hashstat shows you the hit rate, which is the number of hits divided by the number of tries (hits plus misses.) Blindspots are files that aren't found using the hash but are found on a second pass, using an exhaustive search. Blindspots happen when you add new files to a path directory; when detected, they cause the directory to be rehashed.

(See also the **rehash** and **unhash** commands.)

Options:

-h	Help.
--	End of options.

head: Copy the first few lines or bytes of a file to Stdout

Usage: head [-hbnq<size>-] [-t<tabs>][file1 file2 ...]

head copies the first part of each of the files you specify onto stdout. If several files are given, each is announced unless you specify quiet mode. You may specify how much to read from each file in either lines or bytes. If no files are given, **head** reads from stdin.

Options:

-h	Help. (This screen.)
-c	Count characters. (Default is lines.)
-q	Quiet mode. Don't announce the name of each file as it's read.
-<size>	Amount to be read from each file. (Default is 10 lines or 512 bytes.)
-t<tabs>	Tab settings to use for viewing text. (Default is to use value given by the TABS environment variable or, if that's undefined, to do no tab expansion.)
-n	No tab expansion.
--	End of options.

heapstat: Print Heap Usage Statistics

Usage: heapstat [-h-]

Summarize the current heap usage, i.e., the number of objects allocated on the heap and the total storage used. Heapstats are only available if they were enabled with the -H option when the shell was started. Because of the significant performance penalty associated with tracking heap usage, heapstats are not normally enabled.

Options:

-h	Help.
--	End of options.

history: Display the History List

Usage: history [-srh-] [n [m]]

Display commands from the history list. If no operands are given, show the entire list. If a single integer 'n' is specified, display the last n commands. If both integers 'n' and 'm' are given, then:

if m >= n:	Display commands numbered n through m.
if m < n:	Display m commands, beginning at command n.

(The size of the list is controlled by the **history** variable, changed using the **set** command.)

Options:

-s	Short form: Leave off the command numbers.
-r	Reverse order: Show the oldest first.
-h	Help.
--	End of options.

kill: Kill Specified Process or Thread Activity

Usage: kill [-xh-] scheduling_id [scheduling_id ...]

Terminate specific activities. Normally, only direct child screen groups and processes can be killed. Only certain threads will respond to a kill; you cannot inadvertently kill normal internal housekeeping.

Scheduling_ids are the thread, process or screen group identifiers as reported by the **ps** command. An id beginning with 't' indicates a thread; 'p' indicates a process; 's' indicates a screen group. (Screen groups are only on OS/2.)

Options:

-x	Kill even processes that are not direct children. (Has no effect on OS/2 screen groups that are not direct children; this is an OS/2 kernel restriction.)
-h	Help.
--	End of options.

label: Read/Write the Volume Label

Usage: label [-h-] [disk:] [label]

label reads or writes the label on the specified disk, given as a single alphabetic drive letter plus a colon. If no disk is specified, the current disk is assumed. If no label is specified, the current label name is printed. Otherwise label tries to set the label to specified name. Spacing between the disk: and the label operands is optional.

The principal differences between this and the standard OS/2 or NT label command are that (1) it can be invoked directly from the C shell (the standard OS/2 label had a bug in its command line processing that meant it had to be invoked only via cmd.exe), and (2) it never prompts, making it more suitable in scripts.

Options:

-h	Help. (This screen.)
--	End of options.

ls: List Files or Directories

Usage: ls [-+][lL!rqhx- ndtsez mU DHSARa 0-9 wME] path1 [path2 ...]

ls lists the specified files or directories. Directory names are highlighted; names of files or directories with the system bit set are shown in green.

File and directory names are normally converted to lower case for better readability. But **ls** will look for an environmental variable, **MIXEDCASEDRIVES**, giving a list of drives, e.g., f-hq, meaning drives F: through H: and drive Q:, which should be displayed in mixed case.

To customize **ls** with a default behavior of your own choosing, use the **LSOPTIONS** environmental variable. **ls** will first process any options specified in **LSOPTIONS** before any specified on the command line.

General Options:

-l	Long format listings of the files or directories.
-L	Very Long format listings. (Under OS/2 1.1, used and allocated media space is always reported. Under 1.2, the size of any extended attributes is shown instead of media space if -E is specified. Also, under 1.2, root directories do not have a timestamp and are reported as midnight, Jan 1, 1980)
-!	List just the names specified, not the contents of any directories. (Remember to type this option at the end of the list so the space afterward will prevent it being confused as a history reference.)
-r	Recursively list contents of any subdirectories.
-q	Quiet. Don’t generate an error message for non-existent files.
-h	Help (short list.)
-hx	Extended help (this description.)
--	End of options. (Useful if filenames start with ‘‘-’’.)

Sorting Options:

Sorts may be done in any order; default is -nds. (The filename is the most important sort criteria and ties are broken by date or size.)

-n	Name (lowest alphabetic first.)
-d	Date (oldest first.)
-t	Time (newest first; same as -dz)
-s	Size (largest first.)
-e	Extension (lowest alphabetic first.)
-z	Reverse the major or last specified sort.
-m	Merge the listings of directories and files.
-U	Unordered (in order of appearance in the directory.)

Selection Criteria:

+<type>	Include files/directories fitting this pattern.
-<type>	Leave out anything fitting this pattern.

where <type> is combination of following characters:

D	Directories.
H	Hidden files or directories.
S	System files or directories.
A	Archive bit set.
R	Read-only files or directories.
.	"." and ".." files.
a	All files or directories regardless of attributes.

The default is "+a -. -H", meaning list anything other than "." and ".." not marked Hidden. Anything specified is added to/subtracted from this set. Ordering of additions or subtractions does make a difference. If the slash, "/", is used to introduce options, it is treated as a plus when used with selection criteria.

Selection criteria normally applies only to directories and files found by listing a directory. All names given on the command line will be listed if they exist regardless of whether they match the selection criteria unless the -c option is specified.

-c	Command line names must match criteria also, unless only one name is given.

Listing Options:

-<integer>	Specified number of columns.
-0	Standard multi-columns on an 80-column screen.
-C	Capitalization. Display filenames in the upper- or lower-case characters actually stored in the file system.
-w	Walk subdirectories to show true sizes in long format listings.
-M	Media allocation is shown instead of amount used with -l.
-E	Extended attribute size is shown instead of media allocation with -L. (-E is ignored under OS/2 1.1.)

Colors:

You may set your own choices for screen colors using these environmental variables:

Name	Use	Default
COLORS	Normal screen colors	White on Black
DIRECTORIES	Directories	Bright
SYSTEMFILES	Files with system bit set	Green
SYSTEMDIRS	Directories with system bit set	(see below)

Colors recognized are black, red, green, yellow, blue, magenta (or red blue), cyan (or blue green) or white. Foreground colors may also be bright or blink. The names of the colors and the words bright, blink and on may be in either upper or lower or mixed case but the names of the environmental variables themselves must be all upper case.

Either or both the foreground and background colors may be specified; if you don't specify a value, it's considered transparent and inherits the color underneath it.

SYSTEMDIRS normally inherits a merging of the **DIRECTORIES** and **SYSTEMFILES** colors. **DIRECTORIES** and **SYSTEMFILES** inherit from **COLORS**.

markexe: Mark the application type bits in an .exe file

Usage: markexe [-ftpzslh-] file1 [file2 ...]

markexe reports or sets the application type bits on OS/2 .exe files to indicate how the application should be started, i.e., whether they must run full-screen, can run in a text window or must be started as full PM graphics applications.

If the .exe is a 16-bit application, **markexe** can also be used to indicate whether it supports long filenames. (All 32-bit applications must, by definition, support long filenames.)

If no flags are specified, the types are reported but not altered.

Application Types:

-f	Full-screen.
-t	Text windowable.
-p	PM graphics.
-z	Zero the type bits. (Undefined type.)

Long Filename Support (ignored for 32-bit .exe files):

-s	Short filenames only.
-l	Long (HPFS-style) filenames supported.

Other Options:

-h	Help. (This screen.)
--	End of options.

mkdir: Make Directories

Usage: mkdir [-h-] directory1 [directory2 ...]

Options:

-h	Help.
--	End of options.

more: A Better More Filter

Usage: more [-#IinNcvbodxCh-] [-r**radix**] [-t**tabs**] [-s**scroll**]
[file1 file2 ...]

This **more** provides a number of advantages over the standard more.com filter. It's faster, goes backwards and forwards through a file, can search for character strings and display binary data embedded in the text. As it reads a file, it builds an ISAM structure on-the-fly that lets it jump to a specific line number almost instantaneously. It's designed for very fast browsing.

On-line help is available to the key bindings by pressing ''h'' when you see the Press H for Help prompt at the bottom of a screenful of data.

For OS/2, there are two versions of **more**: **more.exe** is built in small model and can cache about 11K characters; a ''huge'' version, **moreh.exe**, is built in large model and while slightly slower, can cache about 4M characters. Where speed is less important than being able to scroll all the way back through a large amount of text coming through a pipe, use moreh.exe, which was compiled in large model.

Options:

-#	Show line numbers.
-I	Start up in case-independent search mode.
-i	Go into Interactive mode immediately, which means clear the screen first for faster painting and put up the Press H for Help prompt rather than just exiting if there's less than a screenful of input.
-n	Next file option. Pressing space bar when the end-of-file message is displayed causes more to continue with the next file or to exit if there are no more files.
-N	No stretch. Don't stretch color changes out to the right edge of the screen.
-c	Show non-printables in C language style.
-v	Make non-printables Visible as control characters.
-b	Show Binary values of non-printable characters.
-o	Use Octal for binary data.
-d	Use Decimal for binary data.
-x	Use Hexadecimal for binary data.
-C	Don't clear the screen before each new screenful.
-r**radix**	Use user-specified radix for binary data.
-t**tabs**	Set tabs every integer number of spaces.

-s**scroll** Set the integer default scroll amount.
-- End of options. (Useful if a filename begins with "-".)

Initial tab settings and radix values are taken from the environmental **TABS** and **RADIX** variables if they're defined. Otherwise, **TABS** = 8 and **RADIX** = 16 is assumed.

Colors:

You may set your own choices for screen colors using these environmental variables:

Name	Use	Default
COLORS	Normal screen colors	White on Black
MOREEOF	End or Top of File messages	Green
MORETOPMEM	Top of Memory message	Bright Yellow
MOREPROMPT	Prompt line at the bottom	Green
MOREFILLIN	Characters typed at the prompt	White
MOREERROR	Unrecognizable command errors	Bright Yellow

Colors recognized are black, red, green, yellow, blue, magenta (or red blue), cyan (or blue green) or white. Foreground colors may also be bright or blink. The names of the colors and the words bright, blink and on may be in either upper or lower or mixed case but the names of the environmental variables themselves must be all upper case.

Either or both the foreground and background colors may be specified; if you don't specify a value, it's considered transparent and inherits the color underneath it. **MOREERROR** and **MOREFILLIN** inherit from **MOREPROMPT**. **MOREPROMPT**, **MORETOPMEM** and **MOREEOF** inherit from **COLORS**.

mt: Manipulate the Tape Device (Windows NT only)

Usage: mt [-hv-] [-f device] command1 [command2 ...]

mt is used to give commands to the tape device. If a device is not specified, the device specified by the **TAPE** environment variable is used. If **TAPE** is not defined, \\.\tape0 is used. By default, each command is performed once. Many operations may be performed multiple times by specifying a count.

These are the available commands. Only as many characters as are required to uniquely identify a command need be given. If no commands are specified, status is given.

fss [n]	Forward space n setmarks.
fsqs [n]	Forward space to n sequential setmarks.
fsf [n]	Forward space n filemarks.
fsqf [n]	Forward space to n sequential filemarks.
fsr [n]	Forward space n records.
bss [n]	Back space n setmarks.
bsqs [n]	Back space to n sequential setmarks.
bsf [n]	Back space n filemark.
bsqf [n]	Back space to n sequential filemarks.
bsr [n]	Back space n records.
stp [n [p]]	Set tape position to block n in partition p. Default is block 1 in current partition.
stpa [n]	Set tape position absolute. Go to block n counting from the beginning of the tape.
eod [p]	Goto to end of data in partition p.
weod	Write end of data at current position.
wfm [n]	Write n filemarks.
wsfm [n]	Write n short filemarks.
wlfm [n]	Write n long filemarks.
wsm [n]	Write n setmarks.
rewind	Rewind the tape.
load	Load the tape and move to the beginning.
lock	Lock the tape ejection mechanism.
tension	Adjust tension by moving to the end of tape and then rewinding.
unload	Rewind and unload the tape.

unlock	Unlock the tape ejection mechanism.
blksize [n]	Set blocksize. (Default is device default.)
compress [n]	Enable (1) or disable (0) compression.
ecc [n]	Enable (1) or disable (0) ECC.
setmarks [1]	Enable (1) or disable (0) setmark reporting.
warning [1]	Set end-of-tape warning size.
padding [1]	Enable (1) or disable (0) data padding.
status	Print status information about the tape device.
erase	Erase from current position to end of partition.
create [n [size]]	Create partitions of device default number and size. Default number and size are determined by the device.

Options:

-h	Help. (This screen.)
-v	Verbose. Status command should dump out complete drive and media parameters. Default is to give just brief status and position information.
-f **device**	Name of the tape device to open.
--	End of options.

mv: Move Files or Directories

Usage: mv [-filmh-] source1 [source2 ...] destination

mv is an intelligent file or directory mover. It's able to move either files or directories as objects regardless of whether the source and destination are in the same directory or even the same drive.

Files being moved anywhere on the same partition are simply renamed; the data stays where it is. Moving a file to a different partition is done by copying the file and then deleting the original. If you move a directory but it keeps the same parent, that also is a simple rename. Moving a directory to a new parent is done by a **mkdir** in the new parent and moving all the directory contents using the same ''rename where possible'' strategy recursively.

Options:

-f	Force read-only files to be overwritten.
-i	Interactive: ask before moving each object on the command line.
-l	Logging is on: display the name of each file or directory
-m	Merge sub-directories of same name in source and destination.
-h	Help.
--	End of options. (Useful if filenames start with ''-''.)

newer: Test whether file1 is newer than all the others

Usage: newer [-h-] [file1 file2 ...]

Prints ''1'' if file1 is newer, otherwise prints ''0''.

Options:

-h	Help.
--	End of options.

older: Test whether file1 is older than all the others

Usage: older [-h-] [file1 file2 ...]

Prints ''1'' if file1 is older, otherwise prints ''0''.

Options:

-h	Help.
--	End of options.

patchlnk: Patch ''The linker bug'' (OS/2 only)

Usage: patchlnk [-hcw-] filename

patchlnk can be used to patch a specific bug in your linker if you're having problems linking under the Hamilton C shell on OS/2. (See also the discussion under ''Known Bugs'' in the readme.too file sent with Hamilton C shell.)

If you're having problems linking long lists of .obj files, you should save a copy of your link.exe the way it is, then patch your linker with this program and see if the problem goes away.

Microsoft has discovered they had a bug in the version of the C library used to build their link.exe releases (roughly) 5.01.21 through (and possibly past) 5.03. In the file open routine, they had an ''off by one error'' that caused problems if the DosSetMaxFH (maximum file handles) parameter for the process allowed more than 20 file handles to be open. There's a JLE (0x7e) that should have been a JB (0x72). It occurs in the following byte string (in hex):

```
3b 06 ?? ?? 7e 0d 9a
```

where the ?'s mark bytes that can change because they refer to data locations that move from one build of link.exe to the next.

(Since the problem is in the MS C library, the bug *could* affect *any* C program that tries to simultaneously open 20 or more handles with stdio. So far, MS link.exe is only application known to do this but there could be others. **patchlnk** should work on them also and probably even on your MS C library .lib files.)

patchlnk will scan through the file, looking for occurrences, which it will report as hex offsets from the start of the file you name. You can use these numbers to go examine the file with IBM/MS patch.exe if you like and make the change manually.

You can also ask **patchlnk** to make the change, using the -w (write) option. Nothing fancy. It just changes the first occurrence and quits. (If you really think you want any other occurrences patched, just keep rerunning this.) If want to look for changes already made, use -c.

Options:

-h	Help. (This screen.)
-c	Look for changes already made.
-w	Write the first change.
--	End of options.

popd: Pop a new Current Directory and Disk off the Stack

Usage: popd [-dsh-] [n]

Pop an integer n directory paths from the top of the stack, making the top one remaining the new current directory and disk. The default for n is 1.

(See also the **cd**, **dirs**, **pushd** and **rotd** commands.)

Options:

-d	Discard just stack entry n, counting from 0 at the top, popping everything below it up one level
-s	Silent. Don't print the resulting directory stack.
-h	Help.
--	End of options.

ps: Print Process & Thread Status

Usage: ps [-h-]

Show the status of all the screen groups (OS/2 only), processes and threads related to this invocation of Hamilton C shell, highlighting the current thread.

Screen groups are created when a command you type requires a different style of window than is currently running. Screen ids begin with 's'.

Processes are created when commands are typed that require other executable programs to run. Process ids begin with 'p'. OS/2 and NT will only provide information on direct child processes.

Threads are concurrent activities going on inside Hamilton C shell. Some, such as the cleanup activities that watch for other threads or processes to end always run in the background. Others are created when a command requires that an internal function be run in the background. Thread ids begin with 't'.

Options:

-h	Help.
--	End of options.

pushd: Push a new Current Directory and Disk onto the Stack

Usage: pushd [-hs-] [-c [n]] [directory]

Go to a new current directory and disk, saving the old values on the directory stack. If no operands are given, swap the top two elements of the stack. If the path you specify is just 3 or more dots, **pushd** will interpret that specially. Just as ".." means go up one level, "..." means up 2 levels, "...." means up 3 levels, etc. If the directory isn't found, the **CDPATH** is searched.

(See also the **cd**, **dirs**, **popd** and **rotd** commands.)

Options:

-c	Copy an item already on the stack onto the top. This form takes an optional parameter n which specifies the integer item number counting from 0 at the top. Default is n == 0.
-s	Silent. Don't print the resulting directory stack.
-h	Help.
--	End of options.

pwd: Print the Current Working Directories

Usage: pwd [-ch-] [disk1 disk2 ...]

pwd prints a list of the current directories. If no disks are specified, **pwd** looks for a **DRIVEMASK** environmental variable that can be used to mask off just the drive you want reported. The **DRIVEMASK** is specified as a list of drive letters; ranges are allowed. Otherwise, all the fixed disk partitions beginning with c: are reported. The current directory on the current drive is highlighted.

pwd routinely reports everything in lower case for better readability. But **pwd** will look for an environmental variable, **MIXEDCASEDRIVES**, giving a list of drives, e.g., f-hq, meaning drives F: through H: and drive Q:, which should be displayed in mixed case.

Options:

-a	If no disks are specified, report on all disks.
-c	If no disks are specified, report on just the current disk.
-h	Help.
--	End of options.

Colors:

You may set your own choices for screen colors using these environmental variables:

Name	Use	Default
COLORS	Normal screen colors	White on Black
HIGHLIGHT	Current disk	Bright

Colors recognized are black, red, green, yellow, blue, magenta (or red blue), cyan (or blue green) or white. Foreground colors may also be bright or blink. The names of the colors and the words bright, blink and on may be in either upper or lower or mixed case but the names of the environmental variables themselves must be all upper case.

Either or both the foreground and background colors may be specified; if you don't specify a value, it's considered transparent and inherits the color underneath it. **HIGHLIGHT** inherits from **COLORS**.

rehash: Reinitialize Path Hashing

Usage: rehash [-h-]

Hamilton C shell uses a hashing mechanism to speed up searches of the path directories. If you're performing system administration work, moving files in the path directories, the hash mechanism may not always reflect the latest changes you've made. The **rehash** command allows you to turn re-initialize the hash by re-reading all the path directories and setting nohashing = 0. (The **unhash** command can be used to turn hashing off.)

Options:

-h	Help.
--	End of options.

rm: Remove Files or Directories

Usage: rm [-rfHSxilh-] pathname1 [pathname2 ...]

rm can delete both files and directories. If you try to remove a file with one of the special mode bits set (see **chmod**) or a directory that's not empty, it'll refuse unless you use one of the options to let it know that's really what you mean.

Options:

-r	Recursively remove non-empty directories.
-f	Force read-only files or directories to be removed.
-H	Hidden files or directories can be removed.
-S	System files or directories can be removed.
-x	All of the above.
-i	Interactive: ask before removing each object on the command line.
-l	Logging is on: display the name of each file or directory as it's removed.
-h	Help.
--	End of options. (Useful if filenames start with "-".)

rmdir: Remove Directories

Usage: rmdir [-h-] directory1 [directory2 ...]

rmdir will only remove empty directories. Mode bits (system, hidden or read-only) are ignored.

Options:

-h	Help.
--	End of options.

rotd: Rotate the Directory Stack

Usage: rotd [-hs-] [n]

Rotate the directory stack an integer n positions. Positive values cause upward rotation; negative values cause downward rotation. The default is upward rotation by one position.)

(See also the **cd**, **dirs**, **pushd** and **popd** commands.)

Options:

-s	Silent. Don't print the resulting directory stack.
-h	Help.
--	End of options.

sed: Stream Editor

Usage: sed [-hins-] [-f scriptfile] [-e script] [script] [file1 file2 ...]

sed is a special text editor for use on streams of data where it cycles, reading a line from input, applying the editing operations you've specified, and writing the result to stdout. The input is read in a single pass and each line is acted on only once.

The editing script can be specified on the command line or, if it's long and complex, in a file. If you want to combine a script on the command line with any other script, you must use the -e option. The editing operations available are the usual search/replace, insert/delete, etc. With each operation, you generally can specify the lines in the file it should affect either by line number or matching a pattern or a range of lines.

Options:

-h	Help. (This screen.)
-f scriptfile	Read the script from a file. Multiple -f options are allowed and the scripts are concatenated.
-e script	Take the following argument word as a script. Multiple -e options are allowed.
-i	Ignore character case.
-n	Don't automatically write the contents of the edit buffer to stdout at the end of each cycle.
-s	Read the script from stdin. (The input stream to be edited must be in a file.)
--	End of options.

Scripts:

The script is written as a series of commands, each separated from the next by a line end or a semicolon. The format of a command is:

[address1 [, address2]] operation

Spacing between fields is arbitrary: you can use spaces or tabs between fields or have none at all as you prefer. Here are the fields:

address	is a line number or regular expression to be matched. When a regular expression is used as an address, it's normally written as /regexp/ but it's also possible to use the syntax \?regexp? where ? is any character.

Zero- or One-Address Operations:

If you don't give an address, the operation is applied to all lines.

If you give a single address, the operation is done on each line that matches.

Ranges:

When you give two addresses, you define a range. A range can be a single line, e.g., ''3,3'', or a whole group of lines. If the addresses involve pattern matching, e.g., ''/^#ifdef/,/^#endif/'', the range might occur over and over in the input and will be acted on each time.

Most operations, e.g., search/replace, are done against all lines in the range. The exceptions are i\ (insert), which is acted on at entry to the range; a\ (append), and q (quit), which are acted on at the end; and c\ (change), which deletes all the lines in the range but doesn't write any output until the end of the range.

operation	is one of the 28 basic operations provided, together with any arguments it takes.

Operations:

There are two buffers in **sed**: the edit buffer and the hold buffer. Most of the editing operations work on the edit buffer, doing search/replace operations, translating characters, and saving or retrieving text in the hold buffer. **sed** automatically puts each new line into the edit buffer (along with its trailing line end) at the start of each cycle unless there was something left over after a D (delete head) command.

A second group of operations provide ways of inserting blocks of static text.

The next group of operations provide rudimentary ways of condition-testing and branching and of nesting a series of operations together.

Finally there operations for printing and doing other i/o and for other miscellaneous things.

Basic Editing:

d	Delete this section. Start the next cycle.	
D	Delete from the beginning of edit buffer through and including the first line ending. If there's text remaining, immediately start over at the top of the script without reading a new line.	
s/.../.../[ngpw]	Search/replace using regular expressions. In the replace string, ''&'' means whatever the search string matched. ''\n'', where n is a number, means whatever matched that tagged expression in the search string. The search and replace strings are shown here delimited with /, but you may choose any character you like that doesn't occur in either string. The following modifiers are accepted:	
	n	A decimal number from 1 to 65,535. Substitute for just the n'th occurrence of the search pattern.
	g	Global. Substitute all occurrences.
	p	Print the edit buffer if a change was made.
	w file	Write the edit buffer onto the end of the file if a change was made.
y/.../.../[c]	Translate all occurrences of characters in the first string to the corresponding characters in the second string. As in the **tr** utility, ranges are allowed, as in ''y/a-z/A-Z/''. The optional ''c''	

flag can be used to complement the input range. Either string can contain \r and \n characters. If the first string is longer than the second, all the extra characters are mapped to the last character in the replacement string or to a null if there was none. If the second string is longer, the extras are ignored.

Using the Hold Buffer:

g	Get from the hold buffer. Replace the current contents of the edit buffer.
G	Get from the hold buffer and paste onto the end of the edit buffer.
h	Hold. Replace the contents of the hold buffer with a copy of the edit buffer.
H	Append to hold. Copy the edit buffer contents onto the end of the hold buffer.
x	Exchange the edit and hold buffers.

Inserting blocks of Static Text:

a\ \r\n text	Append the text to the output at the end of the cycle if this is the last line in the range. All but the last line of text should have a "\" just before the \r\n sequence at the end of each line.
c\ \r\n text	Change this section to read as shown in the following text. Start the next cycle.
i\ \r\n text	Insert. Immediately copy the following text to stdout if this is the start of the range.

Condition-Testing, Branching and Grouping operations:

b label	Branch to the label elsewhere in the script. (If no label is given, it's to the end of the script.)
q	If this is the end of the range, quit.
t label	Branch to the label if search/replace changes have been made since the most recent input line was read or a t operation was run.
: label	Label. (No address prefix is allowed.)
{ }	Group a series of operations together.

Printing:

=	Write the line number to stdout.
l	List the contents of the edit buffer in C language style with escape sequences for binary characters.
p	Print. Write the contents of the edit buffer to stdout.
P	Print from the beginning of the edit buffer through and including the first line ending.
#n	Suppress any automatic output at the end of each cycle.

Other I/O operations:

n	Next line. Write the contents of the edit buffer to stdout, dump any appended text and read a new line from stdin to the edit buffer.
N	Read the next line onto the end of the edit buffer with a \r\n sequence in between.
r file	Copy the contents of this file to stdout at the end of the cycle.
w file	Write the edit butter onto the end of the file, creating it if it doesn't exist.

Miscellaneous:

! operation	Don't apply this function unless the addressing *doesn't* match. Invert the line selections.
;	Null statement.
# comment	Comments and blank lines are ignored.

If multiple a\ (append) or r (read file) instructions are executed (or the same one is iterated in a loop), a new entry is made on the end of a list of all the appended text blocks to be copied to stdout at the end of the cycle. When the end of the cycle is finally reached, the whole list is dumped, in order from the top.

Regular Expressions:

Search patterns are specified as regular expressions like those used by grep. Regular expressions are written in this notation, in decreasing precedence:

c	Any ordinary character matches itself.
\c	Match the literal character c.
^	Beginning of line.
$	End of line.
.	Match any single character.
[...]	Match any single character in the list.
[^...]	Match any single character not in the list.
\n	Match whatever literal text the n'th tagged \(...\) expression matched.
r*	Match zero or more occurrences of r.
r1r2	Match expression r1 followed by r2.
\(r\)	Tagged regular expression. Match the pattern inside the \(...\), and remember the literal text that matched.

A regular expression pattern cannot contain Nulls but it can contain NewLine or CarriageReturn characters (which may be useful with the N or G commands.)

When typing a regular expression on the command line, remember that $, [,], ^, (and) have special meaning to Hamilton C shell. Put single quotes around the string to turn off that special meaning. Also, even inside quotes, type ^^ to mean ^ except when it immediately follows [.

Also, / at the beginning of a word on the command line is taken as introducing an option. If you mean it to be the start of a script, use the "--" option ahead of it.

setrows: Set/Report the number of rows in an OS/2 window

Usage: setrows [-h-] [rows]

setrows will try to set the vertical height of the current window to the specified number of rows. In a PM text window, almost any reasonable number of rows can be chosen, but full-screen, you can choose only from this set: 12, 14, 21, 23, 24, 25, 27 to 30, 33, 34, 39 to 43, 45 to 50 and 54 to 60. (Your hardware may not support all these possibilities.)

If the number of rows is not specified, **setrows** just reports the current value.

(**setrows** is supported only on OS/2.)

Options:

-h	Help. (This screen.)
--	End of options.

sleep: Sleep for a Specified Period

Usage: sleep [-hm-] [seconds]

Sleep the specified integer number of seconds (rounded to the next clock tick.) Default is 1 second or until an interrupt occurs.

Options:

-m	Sleep interval is in milliseconds rather than seconds.
-h	Help.
--	End of options.

source: Read Commands from a File

Usage: source [-nh-] filename

Commands are executed just as if they were typed directly in to this thread of Hamilton C shell and can change local variables and current directory settings.

(This differs from typing the name of a C shell script file as a command; if you do that, a separate thread is created.)

Options:

-n	No Execute (Just load the history list.)
-h	Help.
--	End of options.

split: Split a Large File into Chunks

Usage: split [-bhs<size>-] [infile [prefix]]

split breaks up a large file into chunks, written to a series of output files formed of the prefix plus a .nnn extension, where nnn is a 3-digit decimal number. The default prefix is 'chunk'. Each chunk will be of the specified number of lines or bytes. If only one operand is given, it's assumed to be the input filename unless -s is specified.

Options:

-h	Help. (This screen.)
-b	Count bytes. (Default is lines.)
-s	Stdin is split. (Implied if no files are specified.)
-<size>	Chunk size. (Defaults are 3000 lines/300000 bytes.)
--	End of options.

startwin: Start Win3.x Applications under OS/2 2.x

Usage: startwin [-h-] [-s type] [file] [arguments]

Start the specified Win3.x application in a Virtual DOS machine using the WinStartApp API call. This separate invisible PM application is used to run the Windows application because:

(a) WinStartApp is the only way to start Win3.x apps in seamless mode and
(b) the only way to find out when the Win3.x child started by WinStartApp completes is via a message queue -- and only PM applications may have a message queue.

By default, standard mode is used under OS/2 2.0 and enhanced mode under OS/2 2.1. If seamless support appears to be available, the default is to run the application in a window using a common (shared) copy of the Windows code; otherwise, a full-screen Windows session is used by default.

If the filename specified to **startwin** is not fully-qualified, **startwin** will search for it in the PATH directories. Just as with the C shell, the current directory will be searched only if it appears in the PATH list or if the file specified is clearly a relative path (i.e., it contains a \ or / or a drive specification). If necessary, **startwin** will append .exe to the name specified.

startwin is called automatically by the C shell if you type the name of a Win3.x app as a command under OS/2 2.x. The only reason you might manually invoke **startwin** would be to override the default session type.

Options:

-s type Session type to be used, overriding the default type. Session type numbers are as given in the OS/2 pmshl.h header, including the following:

3.0 Standard Full-screen	11
3.0 Standard Seamless VDM	13
3.0 Standard Seamless Common	14 <-2.0 default
3.1 Enhanced Seamless VDM	17
3.1 Enhanced Seamless Common	18 <-2.1 default
3.1 Enhanced Full-screen	19

-h Help.

-- End of options.

strings: Extract ascii strings from a file

Usage: strings [-habetqvlodx-] [-<min>] [-r<radix>] [file1 file2 ...]

strings will search for any occurrences of ascii text in the files you give it. The presumption is that the files are mostly binary and perhaps quite large, making it impractical to look at them directly.

A string is normally defined as 4 or more printable ascii characters terminated by a Null, CarriageReturn, a NewLine or a CarriageReturn-NewLine combination. All the white space characters are considered printable and are included in the length count except when they terminate a string. (To C programmers, these printable ascii characters are the isprint() and isspace() characters.)

If you specify a series of files, they're searched one after the other, each one introduced by name unless you specify Quiet mode. Each string that's found is listed on a separate line. Note that if a particular string contains NewLine or CarriageReturn characters, it will be displayed as a series of (possibly) very short substrings, one per line.

Options:

-h	Help. (This screen.)
-<min>	Minimum string length to report, specified as a decimal integer.
-a	Any string, even if not terminated with a line ending or a null character.
-t	Trim leading white space from each string.
-b	Discard strings containing only white space.
-e	European characters (accented alphabetics and European punctuation) will be considered as ordinary printable text.
-n	Control characters other than Tab, CarriageReturn and NewLine will be considered to be string terminators just like the null character.
-q	Quiet mode. Don't announce the name of each file as it's read.
-v	Verbose. Paste the name of the file in which it occurs onto the front of each string.
-l	Long mode. Show where each string occurs, counting bytes from the beginning of the file. The radix used can be explicitly chosen with -o, -d, -x, or -r; it defaults to the value specified by the RADIX environmental variable if defined or 16 otherwise.
-o	Octal offsets.

-d	Decimal offsets.
-x	Hex offsets shown.
-r<radix>	User-specified radix.
--	End of options.

sum: Checksum the contents of a file

Usage: sum [-rxh-] [file1 file2 ...]

sum checksums the contents of each of the files you specify, writing the results to stdout along with a count of the number of 512-byte blocks it read (including any partial blocks.) If no files are given, **sum** reads from stdin. **sum** is typically used to validate a file communicated over a possibly noisy communications line.

sum treats the characters it reads as 8-bit unsigned integers and normally just adds them together to form a 16-bit unsigned result. Overflows are ignored.

Options:

-r	Rotated checksum. Rotate the accumulated checksum right one bit position before adding each character.
-x	Xor'ed checksum. For each new character, `c`, the checksum, `i`, is calculated as `i += (i << 1) ^ c`.
-h	Help. (This screen.)
--	End of options.

(The default and rotated checksums are the same as those calculated by the UNIX System V **sum** command; the xor checksum is unique to this implementation.)

tabs: Expand/Unexpand Tabs

Usage: tabs [-hua-] [-<tabs>][file1 file2 ...]

tabs expands tab characters into spaces or, if -u is specified, unexpands spaces into tabs as it copies the input files to stdout. If no files are given, **tabs** reads from stdin. If multiple files are given, they're concatenated one after the other to stdout.

When unexpanding, only leading white space is normally converted to tabs. If -a is specified, tabs are inserted anywhere they would replace two or more characters.

Options:

-h	Help. (This screen.)
-u	Unexpand tabs.
-a	Unexpand all tabs.
-<tabs>	Tab settings to use for viewing text. Default is to use value given by the TABS environment variable or, if that's undefined, to assume tabstops every 8 characters. If desired, a list of tabstops can specified, separated by commas; the difference between the last two tabs given is used a standard increment for following tabs.
--	End of options.

tail: Copy the last few lines or bytes of a file to Stdout

Usage: tail [-hcfnq-] [-i<sec>] [m<msec>] [-+<start>]
[-t<tabs>] [file1 file2 ...]

tail copies the last part of the file you specify onto stdout. You may specify where to begin copying in either lines or bytes measured from either the beginning or the end of the file. If no file is given, **tail** reads from stdin. If several files are given, each is announced unless you specify quiet mode.

Options:

-h	Help. (This screen.)
-c	Count characters. (Default is lines.)
-f	Follow. At end of file, go into an endless loop, sleeping for a second, then waking up to see if more data's been added. (Use ^C to exit.) The follow option is not valid if more than one file is specified and is ignored if data is being read from a pipe.
-i<sec>	Interval in seconds between checking for more data if -f option is used. (If the interval specified is greater than one second, tail will temporarily drop back to a one second interval anytime it finds new data when it does wake up.)
-m<msec>	Interval in milliseconds between checking for more data if -f option is used.
-<start>	Starting point, relative to the end of the file. (Default is last 10 lines or 512 bytes.)
+<start>	Starting point, relative to the start of the file.
-t<tabs>	Tab settings to use for viewing text. (Default is to use value given by the TABS environment variable or, if that's undefined, to do no tab expansion.)
-n	No tab expansion.
-q	Quiet mode. Don't announce the name of each file it's read.
--	End of options.

tar: Read/Write UNIX Tape Archive (tar) Format Files

Usage: tar [-acCtMxXh FLvqs!rRfBQWZ-] [-d dir] [-m map]
[-g hours] [-O offset] [-E endset] [-b sex]
[tarfile] [file1 file2 ...]

tar is used to read or write a simple archive format popular for exchanging files between dissimilar machines.

tar normally expects the archive to be in a file specified by the tarfile operand. When listing or extracting files, the following file names are considered to be in the name space of what's in the archive and wildcards should not be used. When adding files, the names are in the user's normal file name space.

When extracting files, this version of **tar** incorporates logic to interactively crunch up a filename in the archive into something legal on an OS/2 or NT filesystem. If -F is specified (or you're using OS/2 1.1), FAT naming rules are enforced. Otherwise, HPFS rules are assumed. **tar** lists these renamings in a .map file. **tar** also incorporates logic to automatically convert between the \n line ending used in an archive and the \r\n line endings used under OS/2 or NT unless the file appears to be binary.

Basic Commands:

-a	Add files to the end of the archive.
-c	Copy the specified files in the archive to stdout.
-C	Copy entire tar segments (including headers and any padding) to stdout. After the last segment, write two blocks of zeros to mark the end of a tar archive. (If you intend to concatenate archive, use the -Z option to suppress writing the trailing blocks of zeros.)
-t	List table of contents (the default). List the files in the archive.
-M	Just build the mapfile; don't extract anything.
-x	Extract files from the archive. Default is all files in the archive. (Do not use wildcarding.)
-X	Extract everything EXCEPT the specified files from the archive.
-h	Help. (This screen.)

Basic Options:

-F	FAT filesystem naming. (Automatically chosen if the OS/2 release level < 1.2.)
-L	Long listing similar to ls -L showing the attributes, timestamp and length of each file in the archive.
-v	Verbose. Also show the offset of each file from the beginning of the archive.
-q	Quiet. tar normally prints the head of each file as it's extracted (-x) or added (-a) to the archive. This option turns that off.
-s	Read the archive from stdin when listing table of contents or extracting. Write the archive to stdout when adding files. (Implies non-interactive.)
-!	Non-interactive. Files are renamed as necessary for OS/2 or NT conventions. (Particularly useful with -M when trying to read a new, large archive file.)
-r	CarriageReturn/NewLine expansion is turned off. (Default is normally to convert between \n in the archive and \r\n under OS/2 or NT unless the file appears to be binary.)
-R	CarriageReturn/NewLine expansion is forced ON, even for files that appear to be binary.
--	End of options.

Advanced Options:

-d dir	Default destination drive and directory when extracting files.
-m map	Specific filename to be used for showing mappings from names in the archive to names used on OS/2 or NT. (Default is to paste a .map extension onto the name of the tar file; if -s is specified, no map file is used unless -m is given.)
-f	Fullpath option. Put the full pathname (minus any disk prefix) specified on the command line into the archive header when adding. (In this context, the full path means the full name given on the command line, not the fully-qualified name starting from the root directory.) When extracting, use the full pathname given in the header to determine where the files will go.
-g hours	The number of hours GMT is ahead of the local zone, e.g., 5 hours in New York or 8 hours in California. U.S. daylight savings is assumed in the summer months. (OS/2 only.)

-G hours	Same as -g, but without daylight savings adjustments. (Under OS/2, if neither -g nor -G is specified, tar looks for a TZ variable in the environment; if it's not defined, tar ignores the difference between local time and GMT used in the archive. Under Windows NT, the -g and -G options and the TZ variable are unnecessary since the system keeps track of the difference between local time and GMT.)
-O offset	Offset at which to start reading the archive file. Given in bytes from beginning of the file.
-E endset	Offset at which to stop reading the archive file.
-b sex	Byte sex in the archive: abcd (default), badc, cdab or dcba.
-B	Autosense byte sex when reading.
-Q	Very Quiet. tar normally warns of any garbled sections that it skipped; this turns off those warnings also.
-W	Warnings. Show just the files that can't be extracted to OS/2 or NT because of their file types. (Always shown in bright red.)
-Z	Suppress writing the trailing blocks of zeros normally written following the last of the segments extracted with the -C option. (Useful for concatenating segments extracted from several separate archives.)

File Format:

Tar files are organized as a series of 512-byte blocks. Individual files always start on a block boundary with a header block followed by the uncompressed data in the file. At the end of the file are two blocks filled with binary zeros. The header has the following format:

```
typedef struct {
      char  name[100],
            mode[8],
            userid[8],
            groupid[8],
            filesize[12],
            timestamp[12],
            checksum[8],
            linkflag,
            linkname[100];
            unused_chars[255];
      } header;
```

Everything is in ascii with nulls and spaces to punctuate the fields. Numbers are always in octal. The mode, user and group ids aren't meaningful on OS/2 and NT and are ignored when extracting and just filled in with read/write for owner, owned by root when adding. The timestamp is in seconds since Jan 1 00:00:00 GMT 1970. The checksum is calculated as if the field contained spaces. The linkflag tells the file type, reported in the long listing as one of the following:

-	Normal File
D	Directory
L	Link (not a separate file, just another name for one that already exists)
S	Symbolic Link
C	Character Device
B	Block Device
F	FIFO

Under OS/2 and NT, only the normal files and directories have any meaning. Directories are highlighted. The other file types are reported in bright red but otherwise ignored.

Colors:

You may set your own choices for screen colors using these environmental variables:

Name	Use	Default
COLORS	Normal screen colors	White on Black
DIRECTORIES	Directories	Bright
FOREIGNFILES	Filetypes not supported by OS/2 and NT	Bright Red

Colors recognized are black, red, green, yellow, blue, magenta (or red blue), cyan (or blue green) or white. Foreground colors may also be bright or blink. The names of the colors and the words bright, blink and on may be in either upper or lower or mixed case but the names of the environmental variables themselves must be all upper case.

Either or both the foreground and background colors may be specified; if you don't specify a value, it's considered transparent and inherits the color underneath it. **DIRECTORIES** and **FOREIGNFILES** inherit from **COLORS**.

tee: Copy Stdin to Stdout and to Each File Specified

Usage: tee [-a-] file1 [file2 ...]

tee is a ''pipe-fitting'' utility for snapshotting the data passing through the middle of a pipeline.

Options:

-a	Append to any output files that already exist instead of truncating and overwriting them.
-h	Help.
--	End of Options.

touch: Create File or Set Modification Time

Usage: touch [-rfc-] [mmddhhmm[yy]] pathname1 [pathname2 ...]

touch sets the timestamp on a file to the current or specified date and time. If the file doesn't exist, the normal action is to create it.

The OS/2 1.1 kernel ignores attempts to change a directory timestamp.

Options:

mmddhhmmyy	Month, Day, Hour, Minutes and Year in decimal.
-r	Recursively touch contents of directories.
-f	Force Read-only files to be touched also.
-c	File is Not created if it doesn't already exist.
-h	Help.
--	End of options.

tr: Translate Characters Filter

Usage: tr [-cdsnh-] [string1 [string2]]

tr is used to perform simple character substitutions as it copies data from stdin to stdout. Individual characters in string1 are mapped to the corresponding characters of string2. If string2 is too short, it is padded by duplicating its last character. In either string, the minus sign can be used to indicate a range of characters.

Here's an example which creates a list of all the words in file1, one per line, in file2, where a word is taken as a maximal string of alphabetics. (The ^n character is expanded by the C shell into a newline character.)

```
% tr -csn A-Za-z ^n <file1 >file2
```

Options:

-h	Help. (This screen.)
-c	Complement the set of characters in string1.
-d	Delete any occurrences of the characters in string1. (If present, string2 is ignored.)
-s	Squeeze all strings of repeated output characters in string2 to single characters.
-n	Normalize line endings. Ensure that every line ends with \r\n. Fixup any \r or \n by itself or any reversed \n\r character pair. When squeezing, normalization also causes repeated \r\n sequences to be squeezed if either \r or \n is in string2.
--	End of options.

unhash: Turn off Path Hashing

Usage: unhash [-h-]

Normally, a hashing mechanism is used in an attempt to speed up searches of the path directories. If you're performing system administration work, moving files in the path directories, the hash mechanism may not always reflect the latest changes you've made. The **unhash** command can be used to turn hashing off, setting nohashing = 1.

(See also the **rehash** and **hashstat** commands.)

Options:

-h	Help.
--	End of options.

uniq: Report unique lines in text files

Usage: uniq [-hbcdiuw-] [file1 file2 ...]

uniq does a quick, simple string comparison of adjacent lines in text files, normally just discarding any duplicates as it copies its input to stdout.

There are some **diff**-style options for ignoring upper-/lower-case differences or treating white spaces of any length as equal, etc. Also, you can optionally choose to list only the lines that occur just once or only those that have duplicates.

If several files are given, the effect is the same as if they were pasted together, one right after another into one long file.

Options:

-h	Help.
-b	Blank spaces of any length compare equal. Ignore any leading or trailing white space on each line.
-c	Preface each line with a count of the number of times it occurred.
-d	Duplicates. Only the lines which have at least one duplicate are reported.
-i	Ignore character case.
-u	Unique lines. Only the lines which occur only once are reported.
-w	White space is ignored totally.
--	End of options.

verify: Turn Write Verification Mode On or Off

Usage: verify [-h-] [mode]

When verification mode is turned on, the system will always verify that any data written to a disk can be read back. The mode can be specified as 0 or off, the default, or 1 or on. If no arguments are given, **verify** simply reports the current mode. Under NT, verify is always on.

Options:

-h	Help.
--	End of options.

vol: List Volume Labels

Usage: vol [-ach] [disk1 disk2 ...]

vol reports the volume label information on each of the specified disks. If no disks are specified, **vol** looks for a **DRIVEMASK** environmental variable that can be used to mask off just the drive you want reported. The **DRIVEMASK** is specified as a list of drive letters; ranges are allowed. Otherwise, all the partitions beginning with c: are reported. The current disk's label information is highlighted.

This command is normally stored in the file vl.exe and invoked via an alias so it can be used from cmd.exe without colliding with the internal cmd.exe **vol** function.

Options:

-a	If no disks are specified, report on all disks.
-c	If no disks are specified, report on just the current disk.
-h	Help.
--	End of options.

Colors:

You may set your own choices for screen colors using these environmental variables:

Name	Use	Default
COLORS	Normal screen colors	White on Black
HIGHLIGHT	Current disk	Bright

Colors recognized are black, red, green, yellow, blue, magenta (or red blue), cyan (or blue green) or white. Foreground colors may also be bright or blink. The names of the colors and the words bright, blink and on may be in either upper or lower or mixed case but the names of the environmental variables themselves must be all upper case.

Either or both the foreground and background colors may be specified; if you don't specify a value, it's considered transparent and inherits the color underneath it. **HIGHLIGHT** inherits from **COLORS**.

wait: Wait for Children to Complete

Usage: wait [-h-]

Sleep until a child thread or process completes or until an interrupt occurs.

Options:

-h	Help.
--	End of options.

wc: Count Words (and lines and characters)

Usage: wc [-lwr] [file1 file2 ...]

wc prints counts all the lines, words and/or characters in each of the files specified, and totals for the whole list. If no files are specified, it counts stdin.

Options:

-l	Count lines
-w	Count words
-c	Count characters
-h	Help.
--	End of options.

The default is -lwc.

whereis: Find where executable files are located

Usage: whereis command

whereis is a self-loading procedure that searches all the directories on the path, looking for a .csh, .exe, .com, or .cmd file corresponding to the command specified. All occurrences are printed.

xd: Hex dump a file to Stdout

Usage: xd [-hbwlFD Zdxor nNq-] [-r<radix>]
[-O <offset>] [-E <endset>]
[-L[<items>]] [file1 file2 ...]

xd dumps its input, presumed to be in binary, in variety of formats, the default being hex. If more than one file is given, each is dumped in sequence. If no files are specified, **xd** reads from stdin. Sequences of lines containing the same data are collapsed out and replaced by an asterisk.

Options:

-h	Help. (This screen.)

Basic display formats:

-b	Bytes. (Default.)
-w	16-bit words.
-l	32-bit long words.
-F	Short 4-byte floating point. Floating point values are always displayed in decimal.
-D	Double-precision 8-byte floating point.
-Z	Suppress leading zeros. Normally, leading zeros are shown except when displaying in decimal.

Radix to be used for offsets and data:

-d	Decimal (unsigned).
-x	Hex.
-o	Octal.
-r<radix>	User-specified radix. If a radix < 8 is chosen, offsets are always shown in hex.

Other options:

-O <offset>	Offset at which to begin dumping.
-E <endset>	Offset at which to stop dumping.
-L[<items>]	Number of items (bytes, words, etc.) to dump. If -L is specified, the default is one.
-n	No ascii column.
-N	No offset column.
-q	Quiet mode. Don't announce the name of each file as it's read.

-v	Verbose. Show lines of repeated data.
--	End of options.

Help

Index

Index

Index

Index

Index

Index

Index

Index