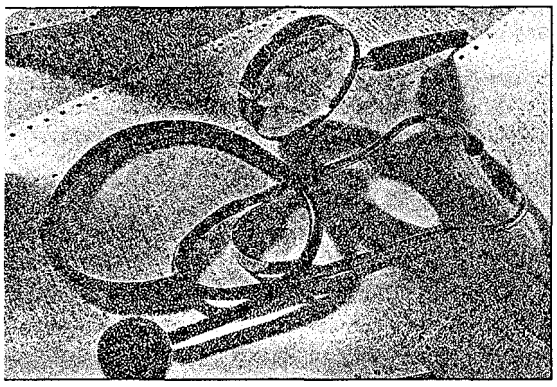

Examining the Hamilton C Shell

Unix power for OS/2

Scott Richman

Starting OS/2 for the first time was, for me, like unlocking a Ferarri, sitting behind its wheel and finding a Yugo's dash. What a disappointment. Sure, the engine and suspension were first rate, but the controls were minimal, the clutch was stiff, and the pedals were nonresponsive! OS/2 comes with great stuff, but CMD .EXE, the default command-line processor, is poor compared to the powerful operating system beneath. CMD.EXE appears to be a port of the MS-DOS COMMAND.COM and lacks the major features of a serious front end.

Fortunately, there's a tool that fills this gap. The Hamilton C Shell is a collection of programs that takes advantage of OS/2 features to create a faster, more powerful environment for serious OS/2 users. The Hamilton C Shell efficiently uses OS/2 to implement a superset of the C shell environment used in the Berkeley flavor of Unix. The Shell supports a powerful script language borrowing C's constructs.

C Shell for OS/2

The Hamilton C Shell is not a quick port of a Unix shell from another system. The Shell was created from scratch, implemented with modern compiler technology, and designed to fully take advantage of the powerful OS/2 architecture, including HPFS (high-performance file system), long filenames, and threads.

Additionally, the Shell supports large command lines and pipes (up to 64K) and includes faster and more powerful utilities than those supplied with OS/2. This is more than Unix — this is a powerful requirement for development under OS/2. The ability to execute C shells simultaneously in different Presentation Manager (PM) text windows converts your PC into a flexible workstation.

The Hamilton C Shell comes with many programs and shell scripts. To install the Shell, you simply copy the files to their new home, and modify your CONFIG.SYS. The Shell program, CSH.EXE, can be executed in a text window of the PM or as a non-PM character-mode application.

Scripts

Scripts allow you to program the many commands and features with full support for complex logic, looping, nested control statements, and symbols. Scripts are composed of C Shell commands, OS/2 programs, and comments prefixed by the pound character (#).

This combination can produce potent applications. Scripts can be composed and tested interactively at the command level or typed into files and run later. The Shell assumes that files with extensions of .CSH are C Shell script files. Scripts can read user input and can be used recursively.

For example, Listing One (page 106) presents CTL_T.CSH, a script to send a Ctrl-T to COM1: every 400 seconds. It's useful when logged onto a busy terminal server that impatiently bumps you off when there's no activity. Invoking this script, using *ctl_t &*, will execute it in the background and the server will be kept busy.

Supporting Procedures

Script programmers can create C Shell procedures, which are more like functions: They accept a parameter list and return a value. These procedures are compiled into C Shell memory and are then executed as new shell commands. Procedures can greatly extend the power and flexibility of your environment.

As an example, consider ZCW.CSH (Listing Two, page 106), which is used to build a C++ PM program. ZCW.CSH defines a procedure that receives a filename as its parameter. The script calls the procedure at the end: The Shell reads the file once, compiles the procedure and executes the compiled code from that point on. In other words, the *zcw* procedure is now treated like another C Shell command.

Listing Three (page 106) shows the global edit procedure *ged*, which can be used to globally edit several files. For instance, you can edit all .H files and change your last name from "Lovejoy" to "Stern," using the command *ged s/Lovejoy/Stern/ *.h*. As with *zcw*, the Shell reads and compiles the procedure and executes *ged* as it would any other C Shell command.

Variables

Users can create local, environmental, and C shell global variables. These symbols can contain any text representing pathnames, strings, numbers, and so on, which can be referred to by the other Shell components. Long pathnames, for instance, could be stored

Scott is an independent software consultant specializing in systems and applications programming under VMS, Unix, DOS, and the Macintosh. He can be reached at R.R. 3, Box 3471, Susquehanna, PA 18847.

in variables and used in a command line to refer to the target location. To define a variable, use the *set* command *(set a = "this is a ")*. To have the Shell calculate an expression, use @ instead of *set*.

Additionally, variables can be arrays with full support for C-style subscripting of the elements. The Shell makes it easy to access the words which make up a variable. The Shell supplies many internal variable functions to test and manipulate the text within a symbol. The *printf* function, for example, is used to format variables. There are also provisions to scan strings for substrings, concatenate variables, and return string lengths.

The Shell is also flexible in treating symbols as numbers and will allow complicated arithmetic calculations. The Shell handles integer and floating-point arithmetic and supports C-like calculations, evaluations and expressions, including *switch* and *case*. Variables can be tested for patterns using the Unix pattern-matching expressions.

Command	Description
!^	Inserts the first argument (or word) of the last line.
!*	Inserts all the arguments of the last line.
!!	Inserts the previous line.

Table 1: *History recall commands*

Taking Command

The Shell has full command history. It remembers previous command lines, which can be recalled through many different methods. Besides using the up and down arrow keys to recall past lines, you can recall a previous command line (or specific parts of it) by command sequence number, or you can recall the last command which contained a specific string. Groups of command lines can be saved into a text file and later read back into another session. The saved command lines can be edited by your favorite text editor and then submitted to the Shell as a script. The Berkeley history mechanism supplies many nifty ways to access parts of previous command lines. When a command line contains *!$*, the Shell inserts the last word (argument) of the previous command line: Repeated sequences of commands to the same file (such as edit, compile, link and print) are executed faster and with fewer typos because the argument is never retyped. Some of the other history-recall commands are shown in Table 1.

The Shell lets you define aliases, which allow you to abbreviate or rename any command. Complicated command lines are much easier to work with when they are defined by an alias. Once an alias is defined, it can be used as another command.

Because the C Shell furnishes many ways to group commands together on the same command line, the Enter key has much more power than under conventional PC systems. Command lines ending with an ampersand (&) will be executed in the background. The PS command will show the currently active processes and threads created by the Shell and their command lines, while the Kill command can terminate any job shown by PS, making it easy to manage a multithreaded system. On my wish list of future enhancements, however, is a feature that will display and manipulate the priority of a thread.

File and Command Accessibility

The Shell controls command-name parsing through efficient hashing techniques and sophisticated OS/2 features. Filenames are expanded within the command line with greater speed and flexibility than under OS/2. For example, when you press the Alt/Ctrl key combination, the Shell will complete a partially typed file or command name in the current command line. These features save much time and ensure more accuracy by reducing unnecessary typing.

C Shell supports full Unix filename wildcarding, to provide a very flexible means of describing groups of files. Subdirectories can also be wildcarded. The asterisk (*) and question mark (?) can represent any character except the colon (:) and backslash (\). However, the period between the filename and its extension is no longer sacred. A wildcard expression of *.[ch] will translate into all files with either .C or .H extensions. Square brackets declare a list of characters which can match one character. If the first character within the square bracket list is the escape character (^), the list will define all charac-

Utility	Description
Cut	Outputs specific parts of each line of its input, and allows you to specify the character positions and/or the field numbers to include.
Diff	Compares files or directories, and can be instructed to ignore case and spaces. *Diff* can recursively compare the contents of two directories. You can also define the minimum match length to insist on.
Strings	Searches binary files and displays the ASCII strings found within them. *Strings* is quite handy for finding the strings embedded within a program or database.
xd	Dumps the contents of its input to *stdout*. This wonderful dump utility can display its input by bytes, words, long words, or floating-point values. *xd* is fluent in decimal, hex, oct, and even other user-supplied radixes. *xd* can be told the offsets at which to begin (and end) its dump.
More	Flexible full-screen file browser. *More* will scroll up and down, and search for text and line numbers. It can also format lines with octal and hex values. C programmers will appreciate the feature of displaying the \n\r escape sequences.
Ls	The ultimate DIR program that specifies types of files and displays file information in many different sorted orders. *Ls* can also display file-size totals. The program will, if told, recursively search the directory structure.
Uniq	Displays the duplicate or nonduplicate lines found in a given file.
Fgrep and *grep*	Searches files (or standard input) for specific occurrences of text. *grep* works with regular expressions which can help find approximated text strings.
Tail	Shows the end of a file. If, however, the file is growing (another process or thread is expanding it), it can continue to show the growing file. I find *tail* indispensable for logging downloads while I am free to work in another window.
Sed	A stream editor —a filter which outputs an edited version of its input. *Sed* will replace strings, convert characters, delete text and insert text. *Sed* will work by ranges of line numbers or regular expressions.

Table 3: *My favorite C Shell utilities*

ters that will not match. These character lists may include ranges of characters: [A–Z] [0–9] will match any two characters starting with one alphabetic and ending with one digit.

The Shell also has built-in file tests to determine file type. The commands shown in Example 1, for instance, will print the attributes of the file whose name is stored in the variable *a*. Also, the Shell can be directed to parse full filenames into their component parts, and programming is not needed to edit the extension out of a filename. For example, if we set the variable *a* to "dir1\dir1\file.ext" the Shell will interpret the filenames according to the list shown in Table 2.

The Shell features a directory-stack mechanism comprised of the commands *pushd*, *popd*, and *rotd*. *pushd* is the CD command with memory. It will remember the current directory (by placing it on the directory stack) and then change to a new directory. *popd* will return to the directory at the top of the stack, and *rotd* will rotate the order of the directories saved. Jumping around from directory to directory is a snap, especially when you use wildcards to declare the directory to push.

Redirection

The Shell supports full I/O redirection of any of its components and allows you to build new commands from the output of other commands on the same command line. As an example, to browse the unknown .C or .H files that contain the string VIO, invoke the command: *more 'grep -l VIO *.[hc]'*

The command line within the single quotes is executed first, and its output is then inserted into its place. So, *more*'s arguments are the output of *grep*.

The command line in Example 2, which finds all duplicate filenames on the current disk, demonstrates how powerful a simple shell command can be. Example 2 starts by creating a list of all the full pathnames of every file using the *-r* (recursive) option of *ls*. The *:gt* means globally trim each pathname down to just the tail (no drive:\dir\\\). The *foreach* loop writes each name out to the pipe, one per line. All lines are sorted alphabetically and the *uniq -d* command outputs just the duplicates. Within moments, the current drive is scanned for all files with the same name.

Supplied Utilities

The Hamilton C Shell product comes chock-full of many wonderful utility programs. All utilities have the same homogeneous feel, a quality lacking in other software packages. Also, all Hamilton supplied programs will display help when invoked with the *-h* switch. Because there are so many utilities included in the package, I selected my ten favorites and described them in Table 3. Table 4 lists other utilities found in the package.

Expression		Description	C Shell result
$a:h	(head)	Directory	\dir1\dir2
$a:r	(root)	Path w/o .Ext	\dir1\dir2\file
$a:t	(tail)	File name	file.ext
$a:e	(ext.)	extension w/o .	ext
$a:f	(fullpath)	expanded file name	d:\top\dir1\dir2\file.ext

Table 2: *Parsing filenames into their component parts*

```
if ( -d $a) echo $a is a directory
if ( -H $a) echo $a is hidden
if ( -R $a) echo $a is ReadOnly
if ( -S $a) echo $a is SystemFile
if ( -e $a) echo $a exists
if ( -x $a) echo $a is executable
if ( -z $a) echo $a zero length
```

Example 1: *Commands to print the attributes of a specified file*

```
foreach i (`ls -r \`:gt) echo $i; end | sort | uniq -d
```

Example 2: *Command to find all duplicate file names on the current disk*

Utility	Description
chmod	Change mode bits on files (not directories)
markexe	Set OS/2 application type bits
pwd	Print the current working directories
mkdir	Make directories
sum	Checksum the contents of a file
tar	Read/write Unix tape archive format files
dt	Print the date and time
setrows	Set height of current window
patchlnk	Patch "the linker bug"
du	List disk usage statistics
vl	List volume labels
label	Read/write the volume label
newer	Test whether file1 is newer than all the others
older	Test whether file1 is older than all the others
tee	Copy Stdin to Stdout and to each file specified
tr	Translate characters filter
wc	Count words (and lines and characters)
split	Split a large file into chunks
tabs	Expand/unexpand tabs
cat	Concatenate files to Stdout
head	Copy first part of file to output
rmdir	Remove directories
cp & mv	Copy (or move) files or directories. These two programs can force read-only files to be overwritten. They can ask before acting on each file and can log the action. Both will merge subdirectories.
rm	Remove files or directories. *rm* can force read-only files to be overwritten. *rm* can ask before acting on each file and can log the action. *rm* can recursively remove non-empty directories. (System files or hidden files or directories can be removed.)

Table 4: *Hamilton utility programs*

Final Assessment

Of course, no product is without its blemishes. Although it improves with each update, the documentation is the weakest part of this otherwise fine product. The documentation is written for a highly technical user who understands OS/2 and Unix. Users new to Unix will need to read other documents (see Bibliography). The sparseness of complete shell scripts makes it hard for a novice C Shell user to appreciate the many wonderful features of this product. Unix can be cryptic and unfriendly, but the Shell's awesome power makes it worth the effort of learning.

While the C Shell is a text powerhouse, database capabilities would make this product more helpful for business-oriented tasks. More powerful record I/O procedures and structures for full record handling would help. If C Shell could integrate an ISAM engine, C Shell applications could be used to solve complex business and scientific problems.

A screen-capturing feature and internal date functions would be greatly appreciated. The Macintosh MPW commando facility of dialoguing a shell command would help users build complex commands without the aid of manuals.

While C Shell works fine in a PM text window, inevitably it will evolve into a full graphics PM application. Such a version should embody the programming strengths of HyperCard. Controls and gadgets should invoke scripts, and programmable dialogues could facilitate PM applications creation.

Bibliography

Anderson, Gail and Paul Anderson. *Unix C Shell Field Guide.* Englewood Cliffs, N.J.: Prentice Hall, 1986.

Muster, John and Peter Birns. *Unix Power Utilities.* Portland, Ore.: MIS Press, 1989.

The Waite Group. *Unix Primer Plus.* 2nd ed. Carmel, Ind.: Howard Sams, 1990.

DDJ

Reprinted with permission of Dr. Dobb's Journal, 1991

Listing One

```
# CTL_T.CSH

while (1)                       # endless
 echo -n ^x14 > coml:           # send control t to coml:
 sleep 400                      # Zzz for 400 seconds
end # while
```

End Listing One

Listing Two

```
# Procedure zcw builds Zortech C++ and creates a PM program,
        proc zcw(name)                          # param name
                ztc -W -c $name.cpp             # compile ($name is param
name)
                # we test name.obj for eistance,
                if (-e  $name.obj ) then        # got valid obj file to link
                        link $name,/align:16,NUL,os2+d:\oz\cp\srzpm.lib,$name
                        rm $name.obj            # remove the obj
                end     # if
        end  # proc

        zcw $argv  # now here's the invocation of my proc defined above.
```

End Listing Two

Listing Three

```
proc ged(edt_str,files)                 # 2 params
local i                                 # local variables used
local n
foreach i ($files)                      # loop thru the files
        @ n = concat($i:r,".bak")       # save a backup (:r is root name)
        cp -l  $i:f  $n:f               # copy it (:f is full name)
        sed "$edt_str" < $n:f > $i:f    # edit from new to i
end # foreach i
end # end ged proc()
```