OS/2, Unix Style

Tom Yager

The promise of OS/2 is to release users and developers alike from the shackles of 8088-compliant environments. Lifting the 640K-byte memory restriction opened the door to all kinds of more potent applications. In this case, the applications are a pair of Unix-like shells that add command-line and interpreted language-processing capabilities to OS/2.

Hamilton Laboratories has created its own version of the popular Berkeley C shell, the Hamilton C Shell 1.04. Its name is derived from the C-like syntax of its shell scripts. Mortice Kern Systems’ MKS OS/2 Toolkit 3.1 includes a port of AT&T’s Kornshell (named for the shell’s original author, David Korn). This shell is a superset of an older AT&T invention, the Bourne shell.

Both packages include a variety of Unix-like commands to make the OS/2 environment a bit more palatable. Hamilton ships 22 additional executable files with its C shell, while MKS provides 102. Still, the Unix user trained to type ls and pwd will derive great comfort from the availability of these and other frequently used Unix commands.

To install the C Shell, you manually copy its executable files to their own directory on the hard disk. The binary files are in a subdirectory (\ bin) on the floppy disk. Next, you execute a utility, dumpenv; it resides in the floppy disk’s root directory and must be copied separately.

The MKS Toolkit includes an installation utility, but it also offers the option of copying the files manually. The automatic installation places files in Unix-like directories under the directory that is named in the environment variable ROOTDIR.

Using the Shells

Both shells run in either full-screen or windowed mode, and it’s easy to set up selections for them in the Start Programs window. As in Unix, both shells read start-up commands from a home directory. OS/2 has no concept of separate users, so this home directory is defined through an environment variable. The shells also require the definition of a separate command search path, usually via the start-up files.

Several commands that OS/2 users take for granted are implemented inside CMD.EXE, the default OS/2 command interpreter, and disappear when an alternative shell is used. The Hamilton C Shell is shipped with aliases that invoke CMD.EXE to execute the built-in commands, such as DIR and COPY. You can modify the MKS Toolkit shell similarly, but the standard configuration includes no predefined aliases for OS/2 commands.

This brings up an interesting point about the differences between the two shells. The MKS Toolkit shell mimics a Unix environment as closely as possible. When a decision had to be made between Unix behavior and that of OS/2, Unix frequently won out. As a result, filenames are built with forward slashes (/) instead of backslashes, and the escape (or “next character is literal”) character is the backslash, not OS/2’s caret (^). In contrast, the Hamilton C Shell is built to let experienced OS/2 users adapt with little hassle. The default filename, escape, and command option characters are those of OS/2.

This rule doesn’t always apply, however. While both shells provide the ability to run processes in the background, the Hamilton C Shell offers a more Unix-like implementation. With the C Shell, you can list background jobs with ps and terminate them with kill (Unix commands that the MKS Toolkit does not provide). In fact, background jobs started from the MKS Toolkit shell seem unstoppable.

Command-Line Processing

The ability to interactively edit the command line is something relatively new, even to Unix. The standard Unix C shell doesn’t have this capability, although modified versions exist that can handle it. The Hamilton C Shell features a comfortable mix of command history and editing, using the editing keys. The up and down arrow keys scroll through previously entered commands, and other keys act as labeled. The MKS Toolkit shell follows the lead of its implementer, providing command-line editing in the style of either vi, the standard Unix full-screen editor, or EMACS, a popular alternative. In this case, only the arrow keys have significance. Other functions must be invoked through editor-specific commands or control sequences. Users familiar with either vi or EMACS will feel right at home.

Both shells maintain a running history of shell commands, and you can reexecute previously executed command lines by reference, using either the command’s sequence number or a portion of its content. The C Shell is a little better at this, since a command line can refer to any number of previous commands. For instance, to reexecute the first three commands of the session, the C Shell sequence would be 1; 2; 3. The MKS Toolkit shell mechanism provides no such straightforward way to combine previous commands. It does, however, allow editing the history file so that you...
and each user has a mail directory.

list and read incoming mail. Each

the shells as programming languages, I
would have given the
sage is kept in a separate, numbered file,
lets you send mail to other users and to

original work, however. The C
much richer than its
part, so any shell programmer would do
selected a simple task: a multiuser mail
system. Working through a primitive
menu-driven interface, this shell script
required reading every message file and displaying lines starting with From:, Subject:, and Date:

The MKS Toolkit shell script came
together quickly and ran smoothly at the
first attempt. This shell’s ability to open
and close files from within a script made
programming easier. While the syntax
took some getting used to, this capability
allowed the entire mail system to fit into
a single script.

The C Shell was only a little more
difficult to manage, lacking the ability
to open and close files on the fly. It is, how­ever, robust in its own right, and
although the “list headers” function had to
be split into a separate script, control
passed to and from it quickly and unno­ticeably.

There was no significant difference in
speed. Both shells hesitated for a bit
before executing while they cached the
function definitions, but once the func­tions began running, performance was
satisfactory.

The effort required to pull together
working scripts was minimal: The MKS
Toolkit shell version took about 2 hours
to produce, and the C Shell took a bit
longer. The MKS Toolkit shell script
was only slightly smaller at 150 lines,
compared to the C Shell’s 187 lines.
Most of the time needed to produce
the scripts was spent flipping through the
documentation.

Unix-Like Documentation
The MKS Toolkit shell has more docu­mentation than the Hamilton C Shell.
The reference pages alone for the dozens
of additional commands in the MKS
Toolkit account for a lot of space, but
there is also a noticeable difference in
quality. Someone unfamiliar with Unix
and its shells would have a much easier
time learning from the MKS Toolkit
manuals, even though there’s more to
read.

Still, the Hamilton C Shell manual is
complete enough, and the company states
that it intends to appeal to "relatively
technically oriented computer users’’
and software developers. Anyone expect­ing
to graduate from batch files directly
to the C shell might be better off finding
another tutorial. I’m familiar with the
Unix versions of the C shell but was con­fused by some of the manual’s tutorial
sections. Even so, it would be possible
for a newcomer to grasp the shell, armed
with the manual and plenty of time to try
the examples and permute them into use­ful variations.

The MKS Toolkit manuals show ex­cellent organization, but the content
needs work. The reference manual is laid
out as Unix documentation, so anyone fa­miliar with Unix should find his or her
way easily. In the case of the MKS Tool­kit shell, however, built-in commands like echo and export have their own refer­ence pages and little or no mention (ex­cept to “see also”) on the shell page it­self. This forces the reader to jump
around the document, when all the shell­related information should have been
presented under $n, the command used
to invoke the MKS Toolkit shell. This
scattering also hampers application de­velopment and seems to be a throwback
to DOS and OS/2 manuals. Users of
these environments might enjoy MKS
Toolkit’s layout.

The MKS Toolkit user’s guide is bet­ter. The most complex of the MKS Tool­kit’s commands are covered by tutorials
in this manual, and they are reasonably
good. The coverage is limited, and you
shouldn’t expect to be introduced to all, or even most, of a command’s features.
Upon finishing the tutorial, you’ll have a
good feel for the command.

Worthwhile Shells
I consider very few products, as a class, indispens­able. These shells fit comfort­ably in that category. No programmers or
systems integrators should consider saddling themselves or their clients with the
incompetent CMD.EXE with these fine
alternatives available.

The MKS OS/2 Toolkit delivers a
healthy dose of Unixness. The whole
MKS Toolkit is well done and feels, with
few exceptions, just like the real thing.
Still, if what the doctor ordered is simply
a better shell for OS/2, then the C Shell
stands out as a finely crafted choice.
If I were to shop today for an OS/2 sys­tem, I’d make sure that my budget in­cluded room for one of these shells. For
those things that you cannot do through
Presentation Manager, these shells and
their accompanying commands make
short work of what can be hours of cod­ing in a compiled language.

Tom Yager is a technical editor for BYTE.
You can reach him on BIX as "tyager."