Copyright © 1978 American Telephone and Telegraph Company
THE BELL SYSTEM TECHNICAL JOURNAL
Vol. 57, No. 6, July-August 1978
Printed in U. S. A.

UNIX Time-Sharing System:

The UNIX Shell

By S. R. BOURNE
(Manuscript received January 30, 1978)

The UNIX shell is a command programming language that provides an interface to the UNIX operating system. It contains several mechanisms found in algorithmic languages such as control-flow primitives, variables, and parameter passing. Constructs such as* **while**, **if**, **for**, *and* **case** *are available. Two-way communication is possible between the shell and commands. String-valued parameters, typically file names or flags, may be passed to a command. A return code is set by commands and may be used to determine the flow of control, and the standard output from a command may be used as input to the shell. The shell can modify the environment in which commands run. Input and output can be redirected and processes that communicate through "pipes" can be invoked. Commands are found by searching directories in the file system in a sequence that can be defined by the user.*

I. INTRODUCTION

The UNIX shell† is both a programming language and a command language. As a programming language, it contains control-flow primitives and string-valued variables. As a command language, it provides a user interface to the process-related facilities of the UNIX operating system. The design of the shell is based in part on the

* UNIX is a trademark of Bell Laboratories.
† This term (shell) seems to have first appeared in the MULTICS system (Ref. 1). It is, however, not universal; other terms include *command interpreter, command language.*

original UNIX shell[2] and the PWB/UNIX shell,[3,4] some features having been taken from both. Similarities also exist with the command interpreters of the Cambridge Multiple Access System[5] and of CTSS.[6] The language described here differs from its predecessors in that the control-flow notations are more powerful and are understood by the shell itself. However, the notation for simple commands and for parameter passing and substitution is similar in all these languages.

The shell executes commands that are read either from a terminal or from a file. The design of the shell must therefore take into account both interactive and noninteractive use. Except in some minor respects, the behavior of the shell is independent of its input source.

II. NOTATION

Simple commands are written as sequences of "words" separated by blanks. The first word is the name of the command to be executed. Any remaining words are passed as arguments to the invoked command. For example, the command

```
ls -l
```

prints a list of the file names in the current directory. The argument -l tells **ls** to print the date of last use, the size, and status information for each file.

Commands are similar to procedure calls in languages such as Algol 68 or PL/I. The notation is different in two respects. First, although the arguments are arbitrary strings, in most cases they need not be enclosed in quotes. Second, there are no parentheses enclosing the list of arguments nor commas separating them. Command languages tend not to have the extensive expression syntax found in algorithmic languages. Their primary purpose is to issue commands; it is therefore important that the notation be free from superfluous characters.

To execute a command, the shell normally creates a new process and waits for it to finish. Both these operations are primitives available in the UNIX operating system. A command may be run without waiting for it to finish using the postfix operator **&**. For example,

```
print file &
```

calls the **print** command with argument **file** and runs it in the

background. The & is a metacharacter interpreted by the shell and is not passed as an argument to print.

Associated with each process, UNIX maintains a set of file descriptors numbered 0,1,... that are used in all input-output transactions between processes and the operating system. File descriptor 0 is termed the standard input and file descriptor 1 the standard output. Most commands produce their output on the standard output that is initially (following login) connected to a terminal. This output may be redirected for the duration of a command, as in

```
ls -l >file
```

The notation >file is interpreted by the shell and is not passed as an argument to ls. If the file does not exist, the shell creates it; otherwise, the contents of the file are replaced with the output from the command. To append to a file, the notation

```
ls -l >>file
```

is provided. Similarly, the standard input may be taken from a file by writing, for example,

```
wc <file
```

wc prints the number of characters, words, and lines on the standard input.

The standard output of one command may be connected to the standard input of another by writing the "pipe" operator, indicated by |, as in

```
ls -l | wc
```

Two commands connected in this way constitute a "pipeline," and the overall effect is the same as

```
ls -l >file
wc <file
```

except that no file is used. Instead, the two processes are connected by a pipe that is created by an operating system call. Pipes are unidirectional; synchronization is achieved by halting wc when there is nothing to read and halting ls when the pipe is full. This matter is dealt with by UNIX, not the shell.

A *filter* is a command that reads its input, transforms it in some way, and prints the result as output. One such filter, grep, selects from its input those lines that contain some specified string. For example,

```
ls | grep old
```

prints those file names from the current directory that contain the string **old**.

A pipeline may consist of more than two commands, the input of each being connected to the output of its predecessor. For example,

```
ls | grep old | wc
```

When a command finishes execution it returns an *exit status* (return code). Conventionally, a zero exit status means that the command succeeded; nonzero means failure. This Boolean value may be tested using the **if** and **while** constructs provided by the shell.

The general form of the conditional branch is

```
if command-list
then command-list
else command-list
fi
```

The **else** part is optional. A *command-list* is a sequence of commands separated by semicolons or newlines and is evaluated from left to right. The value tested by **if** is that of the last simple-command in the *command-list* following **if**. Since this construction is bracketed by **if** and **fi**, it may be used unambiguously in any position that a simple command may be used. This is true of all the control-flow constructions in the shell. Furthermore, in the case of **if** there is no dangling **else** ambiguity. Apart from considerations of language design, this is important for interactive use. An Algol 60 style **if then else**, where the **else** part is optional, requires look-ahead to see whether the **else** part is present. In this case, the shell would be unable to determine that the **if** construct was ended until the next command was read.

The McCarthy "andf" and "orf" operators are also provided for testing the success of a command and are written **&&** and **||** respectively.

$$\mathbf{command}_1 \;\&\&\; \mathbf{command}_2 \qquad (1)$$

executes $\mathbf{command}_2$ only if $\mathbf{command}_1$ succeeds. It is equivalent to

```
if command1
then command2
fi
```

Conversely,

command$_1$ **||** **command**$_2$ (2)

executes **command**$_2$ only if **command**$_1$ fails. The value returned by these constructions is the value of the last **command** executed. Thus (1) returns **true** iff both **command**$_1$ and **command**$_2$ succeed, whereas (2) returns **true** iff either **command**$_1$ or **command**$_2$ succeeds.

The **while** loop has a form similar to **if**.

while *command-list*$_1$
do *command-list*$_2$
done

command-list$_1$ is executed and its value tested each time around the loop. This provides a notation for a break in the middle of a loop, as in

```
while a; b
do c
done
```

First **a** is executed, then **b**. If **b** returns **false**, then the loop exits; otherwise, **c** is executed and the loop resumes at **a**. Although this deals with many loop breaks, **break** and **continue** are also available. Both take an optional integer argument specifying how many levels of loop to break from or at which level to continue, the default being one.

if and **while** test the value returned by a command. The **case** and **for** constructs provide for data-driven branching and looping. The **case** construct is a multi-way branch that has the general form

case *word* **in**
    pattern) *command-list* ;;
    ...
esac

The shell attempts to match *word* with each *pattern*, in the order in which the patterns appear. If a match is found, the associated *command-list* is executed and execution of the **case** is complete. Patterns are specified using the following metacharacters.

* Matches any string including the null string.

? Matches any single character.

[...] Matches any of the enclosed characters. A pair of characters separated by a minus matches any character lexically between the pair.

For example, ***.c** will match any string ending with **.c**. Alternatives are separated by **|**, as in

```
case ... in
    x|y) ...
```

which, for single characters, is equivalent to

```
case ... in
    [xy]) ...
```

There is no special notation for the default case, since it may be written as

```
case ... in
    ...
    *) ...
esac
```

Since it is difficult to determine the equivalence of patterns, no check is made to ensure that only one pattern matches the **case** word. This could lead to obscure bugs, although in practice it appears not to present a problem.

The **for** loop has the general form

for *name* **in** $word_1$ $word_2$ ...
do *command-list*
done

and executes the *command-list* once for each *word* following **in**. Each time around the loop the shell variable (q.v.) *name* is set to the next *word*.

III. SHELL PROCEDURES

The shell may be used to read and execute commands contained in a file. For example,

sh file $\mathbf{arg_1}$ $\mathbf{arg_2}$ ...

calls the shell to read commands from **file**. Such a file is called a "shell procedure." Arguments supplied with the call are referred to within the shell procedure using the positional parameters **$1**,

$2, ... For example, if the file **wg** contains

```
who | grep $1
```

then

```
sh wg fred
```

is equivalent to

```
who | grep fred
```

UNIX files have three independent attributes, *read, write,* and *execute.* If the file **wg** is executable, then

```
wg fred
```

is equivalent to

```
sh wg fred
```

This allows shell procedures and programs to be used interchangeably.

A frequent use of shell procedures is to loop through the arguments (**$1, $2,...**) executing commands once for each argument. An example of such a procedure is **tel** that searches the file **/usr/lib/telnos** containing lines of the form

```
...
fred mh0123
bert mh0789
...
```

The text of **tel** is

```
for i
do grep $i </usr/lib/telnos; done
```

The default **in** list for a **for** loop is the positional parameters. The command

```
tel fred bert
```

prints those lines in **/usr/lib/telnos** that contain the string **fred** followed by those lines that contain **bert.**

Shell procedures can be used to tailor the command environment to the taste and needs of an individual or group. Since procedures are text files requiring no compilation, they are easy to create and maintain. Debugging is also assisted by the ability to try out parts of a procedure at a terminal. To further assist debugging, the shell

provides two tracing mechanisms. If a procedure is invoked with the −v flag, as in

```
sh −v proc
```

then the shell will print the lines of **proc** as they are read. This is useful when checking procedures for syntactic errors, particularly in conjunction with the −n flag which suppresses command execution. An execution trace is specified by the −x flag and causes each command to be printed as it is executed. The −x flag is more useful than −v when errors in the flow of control are suspected.

During the execution of a shell procedure, the standard input and output are left unchanged. (In earlier versions of the UNIX shell the text of the procedure itself was the standard input.) Thus shell procedures can be used naturally as filters. However, commands sometimes require in-line data to be available to them. A special input redirection notation "<<" is used to achieve this effect. For example, the UNIX editor takes its commands from the standard input. At a terminal,

```
ed file
```

will call the editor and then read editing requests from the terminal. Within a shell procedure this would be written

```
ed file <<!
editing requests
!
```

The lines between <<! and ! are called a *here* document; they are read by the shell and made available as the standard input. The string ! is arbitrary, the document being terminated by a line that consists of the string following <<. There are a number of advantages to making *here* documents explicitly visible. First, the number of lines read from the shell procedure is under the control of the procedure writer, enabling a procedure to be understood without having to know what commands such as **ed** do. Further, since the shell is the first to see such input, parameter substitution can, optionally, be applied to the text of the document.

IV. SHELL VARIABLES

The shell provides string-valued variables that may be used both within shell programs and, interactively, as abbreviations for

frequently used strings. Variable names begin with a letter and consist of letters, digits, and underscores.

Shell variables may be given values when a shell procedure is invoked. An argument to a shell procedure of the form *name=value* causes *value* to be assigned to *name* before execution of the procedure begins. The value of *name* in the invoking shell is not affected. Such *names* are sometimes called keyword parameters.

Keyword parameters may also be exported from a procedure by saying, for example,

```
export user box
```

Modification of such variables within the called procedure does not affect the values in the calling procedure. (It is generally true of a UNIX process that it may not modify the environment of its caller without explicit request on the part of that caller. Files and shared file descriptors are the exceptions to this rule.)

A name whose value is intended to remain constant throughout a procedure may be declared **readonly**. The form of this command is the same as that of the **export** command,

```
readonly name ...
```

Subsequent attempts to set **readonly** variables are illegal.

Within a shell procedure, shell variables are set by writing, for example,

```
user=fred
```

The value of a variable may be substituted by preceding its name with **$**; for example,

```
echo $user
```

will echo **fred**. (**echo** is a standard UNIX command that prints its arguments, separated by blanks.) The general notation for parameter (or variable) substitution is

```
${name}
```

and is used, for example, when the parameter name is followed by a letter or digit. If a shell parameter is not set, then the null string is substituted for it. Alternatively, a default string may be given, as in

```
echo ${d-.}
```

which will echo the value of **d** if it is set and "." otherwise. Substitutions may be nested, so that, for example,

```
echo ${d-$1}
```

will echo the value of **d** if it is set and the value (if any) of **$1** otherwise. A variable may be assigned a default value using the notation

```
${d=.}
```

which substitutes the same string as

```
${d-.}
```

except that, if **d** were not previously set, then it will be set to the string ".". (The notation **${...=...}** is not available for positional parameters.)

In cases when a parameter is required to be set, the notation

```
${d?message}
```

will substitute the value of the variable **d** if it has one, otherwise *message* is printed by the shell and execution of the shell procedure is abandoned. If *message* is absent then a standard message is printed. A shell procedure that requires some parameters to be set might start as follows.

```
: ${user?} ${acct?} ${bin?}
...
```

A colon (:) is a command built in to the shell that does nothing once its arguments have been evaluated. In this example, if any of the variables **user**, **acct** or **bin** are not set, then the shell will abandon execution of the procedure.

The following variables have a special meaning to the shell.

- **$?** The exit status (return code) of the last command executed as a decimal string.
- **$#** The number of positional parameters as a decimal string.
- **$$** The UNIX process number of this shell (in decimal). Since process numbers are unique among all existing processes, this string is typically used to generate unique temporary file names (UNIX has no genuine temporary files).
- **$!** The process number of the last process initiated in the background.
- **$-** The current shell flags.

The following variables are used, but not set, by the shell.

Typically, these variables are set in a *profile* which is executed when a user logs on to UNIX.

$MAIL When used interactively, the shell looks at the file specified by this variable before it issues a prompt. If this file has been modified since it was last examined, the shell prints the message **you have mail** and then prompts for the next command.

$HOME The default argument (*home* directory) for the **cd** command. The current directory is used to resolve file name references that do not begin with a /, and is changed using the **cd** command.

$PATH A list of directories that contain commands (the *search path*). Each time a command is executed by the shell, a list of directories is searched for an executable file. If **$PATH** is not set, then the current directory, **/bin**, and **/usr/bin** are searched by default. Otherwise **$PATH** consists of directory names separated by :. For example,

```
PATH=:/usr/fred/bin:/bin:/usr/bin
```

specifies that the current directory (the null string before the first :), **/usr/fred/bin**, **/bin** and **/usr/bin**, are to be searched, in that order. In this way, individual users can have their own "private" commands accessible independently of the current directory. If the command name contains a /, then this directory search mechanism is not used; a single attempt is made to find the command.

V. COMMAND SUBSTITUTION

The standard output from a command enclosed in grave accents (`...`) can be substituted in a similar way to parameters. For example, the command **pwd** prints on its standard output the name of the current directory. If the current directory is **/usr/fred/bin** then

```
d=`pwd`
```

is equivalent to

```
d=/usr/fred/bin
```

The entire string between grave accents is the command to be

executed and is replaced with the output from that command. This mechanism allows string-processing commands to be used within shell procedures. The shell itself does not provide any built-in string processing other than concatenation and pattern matching. Command substitution occurs in all contexts where parameter substitution occurs and the treatment of the resulting text is the same in both cases.

VI. FILE NAME GENERATION

The shell provides a mechanism for generating a list of file names that match a pattern. The specification of patterns is the same as that used by the **case** construct. For example,

```
ls −l *.c
```

generates, as arguments to **ls**, all file names in the current directory that end in .c.

```
[a−z]*
```

matches all names in the current directory beginning with one of the letters **a** through **z**.

```
/usr/srb/test/?
```

matches all file names in the directory **/usr/srb/test** consisting of a single character. If no file name is found that matches the pattern, then the pattern is passed, unchanged, as an argument.

This mechanism is useful both to save typing and to select names according to some pattern. It may also be used to find files. For example,

```
echo /usr/srb/*/core
```

finds and prints the names of all **core** files in subdirectories of **/usr/srb**. This last feature can be expensive, requiring a scan of all subdirectories of **/usr/srb**.

There is one exception to the general rules given for patterns. The character "." at the start of a file name must be explicitly matched.

```
echo *
```

will therefore echo all file names in the current directory not beginning with ".".

```
echo .*
```

will echo all those file names that begin with ".". This avoids inadvertent matching of the names "." and ".." which, conventionally, mean "the current directory" and "the parent directory" respectively.

VII. EVALUATION AND QUOTING

The shell is a macro processor that provides parameter substitution, command substitution, and file name generation for the arguments to commands. This section discusses the order in which substitutions occur and the effects of the various quoting mechanisms.

Commands are initially parsed according to the grammar given in Appendix A. Before a command is executed, the following evaluations occur.

Parameter substitution, e.g., **$user**.
Command substitution, e.g., **`pwd`**.

The shell does not rescan substituted strings. For example, if the value of the variable X is the string *$x*, then

```
echo $X
```

will echo *$x*.

After these substitutions have occurred, the resulting characters are broken into words (*blank interpretation*); the null string is not regarded as a word unless it is quoted. For example,

```
echo ''
```

will pass on the null string as the first argument to **echo**, whereas

```
echo $null
```

will call **echo** with no arguments if the variable **null** is not set or set to the null string.

Each word is then scanned for the file pattern characters *, ?, and [...], and an alphabetical list of file names is generated to replace the word. Each such file name is a separate argument.

Metacharacters such as < > * ? | (Appendix B has a complete list) have a special meaning to the shell. Any character preceded by a \ is *quoted* and loses its special meaning, if any. The \ is elided so that

```
echo \?\\
```

will echo ?\. To allow long strings to be continued over more than one line, the sequence **newline** is ignored.

\ is convenient for quoting single characters. When more than one character needs quoting, the above mechanism is clumsy and error-prone. A string of characters may be quoted by enclosing (part of) the string between single quotes, as in

```
echo '*'
```

The quoted string may *not* contain a single quote.

A third quoting mechanism using double quotes prevents interpretation of some but not all metacharacters. Within double quotes, parameter and command substitution occurs, but file name generation and the interpretation of blanks does not. The following characters have a special meaning within double quotes and may be quoted using \.

$	parameter substitution
`	command substitution
"	ends the quoted string
\	quotes the special characters $ ` " \

For example,

```
echo "$x"
```

will pass the value of the variable x to **echo**, whereas

```
echo '$x'
```

will pass the string *$x* to **echo**.

In cases where more than one evaluation of a string is required, the built-in command **eval** may be used. **eval** reads its arguments (which have therefore been evaluated once) and executes the resulting command(s). For example, if the variable X has the value *$x*, and if x has the value *pqr* then

```
eval echo $X
```

will echo the string *pqr*.

VIII. ERROR AND FAULT HANDLING

The treatment of errors detected by the shell depends on the type of error and on whether the shell is being used interactively. An interactive shell is one whose input and output are connected to a

terminal. Execution of a command may fail for any of the following reasons.

(*i*) Input output redirection may fail, for example, if a file does not exist or cannot be created. In this case, the command is not executed.
(*ii*) The command itself does not exist or is not executable.
(*iii*) The command runs and terminates abnormally, for example, with a "memory fault."
(*iv*) The command terminates normally but returns a nonzero exit status.

In all of these cases, the shell will go on to execute the next command. Except for the last case, an error message will be printed by the shell.

All remaining errors cause the shell to exit from a command procedure. An interactive shell will return to read another command from the terminal. Such errors include the following.

(*i*) Syntax errors; e.g., **if ... then ... done**.
(*ii*) A signal such as terminal interrupt. The shell waits for the current command, if any, to finish execution and then either exits or returns to the terminal.
(*iii*) Failure of any of the built-in commands such as **cd**.

The shell flag **−e** causes the shell to terminate if any error is detected.

Shell procedures normally terminate when an interrupt is received from the terminal. Such an interrupt is communicated to a UNIX process as a signal. If some cleaning-up is required, such as removing temporary files, the built-in command **trap** is used. For example,

```
trap 'rm /tmp/ps$$; exit' 2
```

sets a trap for terminal interrupt (signal 2) and, if this interrupt is received, will execute the commands

```
rm /tmp/ps$$; exit
```

exit is another built-in command that terminates execution of a shell procedure. The **exit** is required in this example; otherwise, after the trap has been taken, the shell would resume executing the procedure at the place where it was interrupted.

UNIX signals can be handled by a process in one of three ways. They can be ignored, in which case the signal is never sent to the

process; they can be caught, in which case the process must decide what to do; lastly, they can be left to cause termination of the process without it having to take any further action. If a signal is being ignored on entry to the shell procedure, for example, by invoking the procedure in the background, then **trap** commands (and the signal) are ignored.

A shell procedure may, itself, elect to ignore signals by specifying the null string as the argument to **trap**. A trap may be reset by saying, for example,

```
trap 2
```

which resets the trap for signal 2 to its default value (which is to exit).

The following procedure **scan** is an example of the use of **trap** without an exit in the trap command. **scan** takes each directory in the current directory, prompts with its name, and then executes the command typed at the terminal. Interrupts are ignored while executing the requested commands but cause termination when **scan** is waiting for input.

```
d=`pwd`
for i in *
do    if test -d $d/$i
      then  cd $d/$i
            while echo "$i:"
                  trap exit 2
                  read x
            do trap : 2; eval $x; done
      fi
done
```

The command

```
read x
```

is built in to the shell and reads the next line from the standard input and assigns it to the variable **x**. The command

```
test -d arg
```

returns **true** if **arg** is a directory and **false** otherwise.

IX. COMMAND EXECUTION

To execute a command, the shell first creates a new process using

the system call **fork**. The execution environment for the command includes input, output, and the states of signals, and is established in the created process before the command is executed. The built-in command **exec** is used in the rare cases when no **fork** is required.

The environment for a command run in the background, such as

```
list *.c | lpr &
```

is modified in two ways. First, the default standard input for such a command is the empty file **/dev/null**. This prevents two processes (the shell and the command), that are running in parallel, from trying to read the same input. Chaos would ensue if this were not the case.

```
ed file &
```

would allow both the editor and the shell to read from the same input at the same time.

The other modification to the environment of a background command is to turn off the **quit** and **interrupt** signals so that they are ignored by the command. This allows these signals to be used at the terminal without causing background commands to terminate.

X. ACKNOWLEDGMENTS

I would like to thank Dennis Ritchie and John Mashey for many discussions during the design of the shell. I am also grateful to the members of the Computing Science Research Center for their comments on drafts of this document.

APPENDIX A

Grammar

item:	*word*
	input-output
simple-command:	*item*
	simple-command item
command:	*simple-command*
	(*command-list*)
	{ *command-list* }

	for *name* **do** *command-list* **done** **for** *name* **in** *word* ... **do** *command-list* **done** **while** *command-list* **do** *command-list* **done** **until** *command-list* **do** *command-list* **done** **case** *word* **in** *case-part* ... **esac** **if** *command-list* **then** *command-list else-part* **fi**
pipeline:	*command* *pipeline* **|** *command*
andor:	*pipeline* *andor* **&&** *pipeline* *andor* **||** *pipeline*
command-list:	*andor* *command-list* **;** *command-list* **&** *command-list* **;** *andor* *command-list* **&** *andor*
input-output:	**>** *word* **>>** *word* **<** *word* **<<** *word*
case-part:	*pattern* **)** *command-list* **;;**
pattern:	*word* *pattern* **|** *word*
else-part:	**elif** *command-list* **then** *command-list else-part* **else** *command-list* *empty*
empty:	
word:	a sequence of non-blank characters
name:	a sequence of letters, digits or underscores starting with a letter
digit:	**0 1 2 3 4 5 6 7 8 9**

APPENDIX B

Metacharacters and Reserved Words

(*i*) Syntactic

|	pipe symbol
&&	"andf" symbol
||	"orf" symbol
;	command separator
;;	case delimiter
&	background commands
( )	command grouping
<	input redirection
<<	input from a *here* document
>	output creation
>>	output append

(*ii*) Patterns

*	matches any character(s) including none
?	matches any single character
[...]	matches any of the enclosed characters

(*iii*) Substitution

${...}	substitution of shell variables
`...`	substitution of command output

(*iv*) Quoting

\	quotes the next character
'...'	quotes the enclosed characters except for '
"..."	quotes the enclosed characters except for $ ` \ "

(*v*) Reserved words

```
if then else elif fi
case in esac
for while until do done
```

REFERENCES

1. E. I. Organick, *The MULTICS System,* Cambridge, Mass.: M.I.T. Press, 1972.
2. K. Thompson, "The UNIX Command Language," in *Structured Programming—Infotech State of the Art Report,* Nicholson House, Maidenhead, Berkshire, England: Infotech International Ltd. (March 1975), pp. 375-384.
3. J. R. Mashey, "Using a Command Language as a High-Level Programming Language," Proc. 2nd Int. Conf. on Software Engineering (October 13-15, 1976), pp. 169-176.
4. T. A. Dolotta and J. R. Mashey, "An Introduction to the Programmer's Workbench," Proc. 2nd Int. Conf. on Software Engineering (October 13-15, 1976), pp. 164-168.
5. D. F. Hartley (Ed.), *The Cambridge Multiple Access System — Users Reference Manual,* Cambridge, England: University Mathematical Laboratory, 1968.
6. P. A. Crisman, Ed., *The Compatible Time-Sharing System,* Cambridge, Mass.: M.I.T. Press, 1965.