

To the Graduate Council:

I am submitting herewith a thesis written by Saleem Ur Rehman entitled "A Comparative Study of Four Operating-Systems Command-Language 'Macros.'" I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Electrical Engineering.

R. W. Rochelle
R.W. Rochelle, Major Professor

We have read this thesis and
recommend its acceptance:

Herbert Dreyer Jr.
Robert A. Baleschman

Accepted for the Council:

C. W. Mink
The Graduate School

A COMPARATIVE STUDY OF FOUR OPERATING-SYSTEMS
COMMAND-LANGUAGE 'MACROS'

A Thesis
Presented for the
Master of Science
Degree
The University of Tennessee, Knoxville

Saleem Ur Rehman

June 1984

ACKNOWLEDGEMENTS

The author wishes to express his sincere appreciation to Dr. R.W. Rochelle for his encouragement, valuable assistance and general helpfulness throughout the course of this study. Special thanks are due to Dr. R.E. Bodenheimer and Dr. H.P. Neff for reviewing this thesis and assuming the responsibility as committee members.

The author wishes to express his deep gratitude to his mother thousands of miles away, who kept him free of all worries. Appreciation is also extended to Mrs. Phyllis Davis for her skillful service in the preparation of this manuscript.

ABSTRACT

The Power Systems Technology Program at Oak Ridge is undertaking a project in Computer-Aided Control System Design (CACSD). The objective of the project is the specification of requirements for CACSD computer software.

The purpose of *this* study was to recommend a command language for CACSD software. The command language should parallel the capabilities given a user writing in the implementation language. The purpose of the command language is to allow the one-time solution of problems without requiring the detail of the implementation language. A good command language with macro capability was needed. This study was conducted considering the command languages which are applicable in CACSD packages. Four operating systems, RT-11, TOPS-10, VM/CMS and UNIX, were considered.

In order to find out about the desired command language, it was necessary to first understand the command languages of these operating systems with emphasis on combining a sequence of commands into a unit. This unit may be substituted as a 'macro' or invoked as a sub-routine, probably with parameterization. This macro facility is available with RT-11 as 'Indirect command files,' TOPS-10 as 'MIC,' with CMS as 'EXECS,' and with UNIX as 'SHELL.'

A comparison is made between these operating-system command language 'macros' or procedures.

TABLE OF CONTENTS

CHAPTER	PAGE
I. INTRODUCTION	1
Need for a Command Language	1
Characteristics of a Command Language	1
Qualities of a Command Language	5
II. A BRIEF OVERVIEW OF RT-11	8
General	8
Monitor Command Language	10
Indirect Files	13
III. A BRIEF OVERVIEW OF VM: CONVERSATIONAL MONITOR SYSTEMS (CMS)	16
General	16
CMS Command Language	17
EXEC	19
Summary of the CMS EXEC Language Facilities	21
Other Facilities in EXEC	22
IV. A BRIEF OVERVIEW OF TOPS-10	26
General	26
The TOPS-10 Command Language	26
MIC Commands	29
Other Useful Facilities Found in MIC	34
V. A BRIEF OVERVIEW OF UNIX	35
General	35
The UNIX Shell	36

CHAPTER	PAGE
V. (Continued)	
Implementation of the Shell	41
Other Programs as Shell	43
VI. COMPARISON OF 'MACRO' CAPABILITIES OF FOUR COMMAND LANGUAGES	44
Comparison of Language Features	47
VII. UNIX AS A BASIS FOR CAD AND SOFTWARE DEVELOPMENT	52
Some Basic Facilities for CAD	53
Software Development Tools	55
Other Advantages	58
Applications of UNIX	58
Nonapplications	59
Conclusions	59
LIST OF REFERENCES	61
APPENDICES	65
VITA	81

CHAPTER I

INTRODUCTION

Need for a Command Language

A command language is designed to allow a human being to communicate with the operating system. The operating system is responsible for coordinating and managing resources in real time¹ on the computer. The command language is used to guide or direct the operating system. On most systems, a single 'control module' is assigned the responsibility for reading, interpreting, and acting on the commands. The command language allocates main memory space to application programs¹ and contains provisions for defining I/O device requirements.

One key source of the commands is the operator's console. Through the console, the operator communicates directly with the control module of the operating system.

The central aspect is that a command language is a form of communication directed to a language processor¹. The language processor is expected to understand correct utterances in the language, and to do its best in the presence of errors.

Characteristics of a Command Language

In order to define the special characteristics of a command language within the family of computer languages, the hardest task now-a-days is to distinguish it from a programming language. Gone is the time when a command language could be counted upon to omit

certain properties indispensable to a programming language, such as 'branching' or 'iteration'², or when a programming language was able to invoke the functions offered in a command language.

Often a command language is the outermost interaction with a system, but this is neither a necessity nor a sufficient property². It is not necessary, because we might be able to invoke a different language interpreter from the outermost level, and if it provided the similar functions, it could reasonably be said to offer an alternative command language. Another common characteristic of a command language which lacks definitive force is that they are invariably interpreted rather than compiled². So, there seems to be no more distinctive characteristics of a command language than that it is concerned with instructions given to a system of some kind. Because a programming language may also include such instructions, so the difference is one of emphasis.

A command language is primarily concerned with invoking system functions, whereas a programming language is not. One might even say of a command language that it is used sometimes as a programming language, and sometimes as a command language, depending upon the proportions of usage of its features.

A command language may be defined as a computer language whose primary concern is to invoke system functions, and to convey system responses².

The distinction made above seems to correspond very roughly to that made between hardware and software--a programming language

exposes fairly directly the instruction set of a computer, while a command language is the interface to some software running on the machine. But many programming languages require extensive software support, e.g., for their I/O facilities; and sophisticated hardware may be used to support what would generally be classified as a system function.

Some other characteristics of a command language stress the need for such a language to be easy to learn, easy to use, flexible, powerful, simple², etc.

Given that all desirable qualities of a command language syntax may not be compatible, the best criterion for making trade-offs among these qualities is the amount of use expected for the language. The attributes that most strongly affect the appropriateness of a command language for its users are 'style,' 'structure,' and 'level of abstraction'³. These three form the skeleton of command language. With an English-like style, hierarchical structure and high level of abstraction enhancing ease of learning, and resistance to semantic errors; command language can be more flexible and efficient. Vocabulary has some effect on ease of learning a command language, but negligible effect on other qualities.

The command language syntax should be simple and free from arbitrary rules, so the major design criterion is to avoid excessive amount of typing³. The arbitrary use of punctuation characters should also be avoided. Also a variety of delimiters and special characters seem to confuse users, so wherever possible their use should be

avoided and English words used in their place. Command names should be brief, but more important, they should be meaningful and easy to remember.

Command language should be extensible⁴. It is impossible to predict future trends, and it is particularly important to be able to add new facilities to existing computer network interfaces. It is also desirable for users to be able to change their own interface, so that it provides facilities that are tailored to their own particular needs. This is usually provided in simple computer systems by a procedure or macro facility, which enables one to store series of operating system commands in files and invoke these files by a simple call. Often the form of that call is similar to an operating system command, and thus users need not differentiate between the two. It is useful for user-defined and system-defined commands to be treated alike, thus reducing the variety of procedures necessary for the users to learn.

A good operating system command language (CL) is an invaluable tool for large programming projects⁵. A common occurrence in such projects is the diversion of significant resources from building the end product to creating internal support programs and procedures. If a CL is a flexible programming language, it can be used to solve many internal support problems, without requiring compilable programs to be written, debugged, and maintained.

Existing CLs include a wide range of structures and abilities, ranging from those quite similar to assembly language to a few

having the appearance and abilities of high-speed programming languages.

Qualities of a Command Language

The following qualities are valuable in a Command Language (CL):

1. Interactive working and batch processing

Provision for interactive working, and as a secondary consideration, batch processing.

2. Ease of on-line use

Many advantages can be gained by replacing key punches with terminals, even when jobs are prepared for batch execution⁵. To support this approach, CL syntax should emphasize simplicity and avoid the need for redundant typing.

3. Convenient use of the same language as an on-line CL and as a programming language

It must be possible to build and store sequences of commands (CL procedures) that can be invoked at a later time.

4. Interchangeability of programs and CL procedures

No user should need to know whether a given command is implemented as a compiled, executable program or as a CL procedure. Syntactic differences should be avoided.

5. Ease of creation and maintenance of CL procedures

It should require very little effort to write a CL procedure, save it for later use, and maintain it. The overhead of writing a small procedure should be especially small, as many procedures are only a few lines long⁵.

6. Organization of CL procedures

It should be possible to share easily the use of procedures among people in a way consistent with organizational structure.

7. Programming language features

It should be possible to write procedures with convenient conditional branching, looping, argument handling, variables, string manipulation, and occasional arithmetic. Thus, many of the capabilities of a typical procedural language are *necessary* for a good CL⁵. However, they may not be sufficient: a CL may need additional capabilities, a somewhat different syntax, and a radically different set of priorities concerning the importance of various constructs. For example, a crucial CL capability is that of connecting executing programs in a variety of ways. Pattern-matching and other string manipulation strings are quite useful.

8. Separation from operating system

A CL interpreter should normally be an ordinary command, treated like other commands, and definitely not embedded in the heart of the operating system. The operating system should *not* be viewed as an implementation of the CL, but as an environment that can support a variety of good CLs. Different users should be able to have different CLs if they feel like it. This approach permits experimentation and evaluation without bothering other users.

An overview of interactive command languages, particularly from a 'macro' point of view on TOPS-10; CMS; RT-11 and UNIX is given in the following chapters. In chapter six, a comparison is made of these languages 'macro' facilities. In chapter seven the usefulness and advantages of the UNIX Operating System are discussed in detail.

CHAPTER II

A BRIEF OVERVIEW OF RT-11

General

RT-11 is DIGITAL's smallest real-time and program development operating system for the PDP-11 family of minicomputers⁶. This single-user operating system runs on hardware configurations ranging from the microprocessor-based PDP-11/03 through the larger PDP-11/60. RT-11's design philosophy is to be small, efficient, reliable, and easy to use.

The software components of the RT-11 computer system can be divided into four general groups:

1. monitors;
2. device handlers;
3. utility programs;
4. language processors.

An RT-11 monitor is a collection of routines that control the operation of programs, schedule operations, allocate resources, and perform input and output⁶. A monitor comprises of three major components: RMON (resident monitor); USR (user service routine); and KMON (keyboard monitor). The resident monitor is the executive controller of the entire system.

RT-11 provides a number of utility programs that are designed to help the user develop programs and perform system housekeeping⁶.

Utility programs cover a wide range of resources; which allow the user to create and edit text, maintain other programs, and locate user-programming errors. Some specific utility programs⁶ in the RT-11 operating system are the following:

1. an editor, which allows the user to create and modify textual material;
2. file maintenance utility programs, which allow to manipulate and maintain the program and data;
3. a librarian, which makes it easy to store and retrieve often used programming routines;
4. a linking program, which converts object modules into a format suitable for loading and execution;
5. a source comparison program, which is used to compare two ASCII files and to output any differences to a specified output device;
6. a dump program, which outputs to the console or line printer all or any part of a file in octal.

The RT-11 operating system also provides support for several programming languages and their respective language processors. It also supports several application packages.

Computer memory must contain the RT-11 operating system, and specifically the RT-11 monitor program. Thus, the user has to transfer the monitor program from the disk device, where it was stored during system installation, to computer memory where it can be used. The process of transferring the RT-11 monitor to memory

is called bootstrapping the system. During bootstrap procedure the RT-11 system is activated. The console terminal is used to interface between the system and user. User can type the commands that control the system operation and also receive the messages and responses from the system.

Monitor Command Language

The RT-11 monitor supplies the monitor command language that is used to control system operations.

The monitor command language is a set of English-like command words that are typed on the terminal keyboard to initiate and control system operations. There are two general formats that one can use to type a command; one is long format and the other the short format. The short command format allows entry of all required information on a single command line.

All monitor commands are terminated by typing a carriage return which instructs the monitor to initiate command and to perform the operation.

The monitor prints a period at the left margin, which is a cue that the system is in monitor command mode and ready to accept a monitor command.

Command syntax. The system accepts commands as either: (1) a complete string containing all the information necessary to execute a command or (2) as a partial string. In the latter case, the

system prompts the user to supply the rest of the information. The general syntax for a command is:⁶

```
Command [/option...] input-filespec [/option...]
Output-filespec [/option...]
```

or

```
Command [/option...]

prompt 1 ? input-filespec [/option...]

prompt 2 ? output-filespec [/option...]
```

where

Command	is the command name;
/option	represents a command qualifier that specifies the exact action to be taken;
prompt	represents the keyboard monitor prompt for more information;
input-filespec	represents the file on which the action is to be taken;
output-filespec	represents the file that is to receive the results of the operation.

Control commands are special function commands used to interrupt program execution, to inhibit terminal output, and to perform other similar special system operations. These commands require no

terminator⁶; the system performs the function as soon as the command is typed. These special commands are:

1. CTRL/C CTRL/C

Interrupt the current operation or program and return control to monitor command mode.

2. CTRL/E

Direct terminal output to both the graphics screen and the terminal printer simultaneously. Type a second CTRL/E to return output control to the graphics screen only. (Valid only when VT11 display hardware is enabled.)

3. CTRL/O

Inhibit the remainder of output from printing on the terminal.

4. CTRL/U

Cancel every character in the current line.

5. DELETE

Cancel the last character typed on the current line.

6. CTRL/X

Erase multiple editing commands. The CTRL/X command causes the editor to ignore an entire command string that might extend over several lines.

7. CTRL/L

Type the form feed as CTRL/L, which the editor recognizes as a page break.

Indirect Files

A collection of keyboard commands can be grouped together as a file that the user wants to use sequentially. This collection is called an Indirect Command File, or Indirect File⁶. Indirect files are best suited to perform tasks that require a significant amount of computer time and do not require any supervision or intervention. Any series of commands that a user is likely to type often can also run easily as an indirect file. The indirect file concept is similar to BATCH processing¹⁷. Although indirect files lack some of the capabilities of BATCH, they are easier to use, use the same commands as normal operations and generally require less memory than the BATCH processor.

Creating indirect files. Indirect files can be created by using the EDIT/CREATE command. It is conventional to use a .COM file type¹⁶ for an indirect file. Lines can be structured to look like keyboard input, placing one command on each line of the file and terminating each line with a carriage return. Prompt character (.) is not included in the line. In addition to using the keyboard monitor commands, RT-11 system utility programs can also run. The last command of the file is ^C, consisting of two separate characters: a circumflex (up arrow) followed by a C¹⁶. This sequence represents CTRL/C in indirect files. This sequence terminates the directory program so that control returns to the monitor when the indirect files finished executing. Otherwise, the directory program would be left waiting

for input from the console terminal when the indirect file finished executing. The last command is terminated with a carriage return.

Some commands normally require a response from the user as they execute. There are some ways to control interaction with the executing command. One way is to use /NOQUERY option on each command that allows it. This option suppresses the confirmation messages entirely when the command is used in an indirect file.

Another way to deal with commands that require a response from the user is to specify the responses in an indirect file. The same information can be specified interactively.

Executing indirect files. To execute an indirect file, a command string is specified according to the following syntax:⁶

@ filespec

where

@ is the monitor command that indicates an indirect file;

filespec represents the name and file type of an indirect file, as well as the device on which it is stored.

The default file type is .COM.

It is conventional to type the indirect file command directly in response to the monitor's prompt. However, indirect command can be placed anywhere in a keyboard monitor command string, as long as it is the last element in the string, not including the comments.

When the monitor or any program that has control of the system encounters an illegal command line, or if an execution error of any kind occurs, that particular line does not execute properly. Execution of the indirect file does proceed. It is possible to destroy files accidentally because of the way indirect files execute. It is safer to use only keyboard monitor commands in an indirect file. This way the monitor regains control after each operation and can abort the indirect file as soon as it detects an error.

Normally, as each line of an indirect file executes, it echoes on the console terminal so that the progress of the job can be observed. SET IT QUIET⁶ command can be used to suppress this printout. Execution of an indirect file at any time can be stopped by typing two CTRL/C characters. Control returns to the monitor, and a new command can be entered.

Another indirect file can be called from within an indirect file. This procedure is called nesting. Nesting is restricted to three levels¹⁷ of indirect files.

CHAPTER III

A BRIEF OVERVIEW OF VM: CONVERSATIONAL
MONITOR SYSTEMS (CMS)General

A Virtual Machine (VM) is an illusion of a real machine⁸. It is the functional equivalent of a real computer which is controlled from the terminal. VM manages an IBM System 370. Each user can choose a different operating system--VM can actually run several different operating systems at once, each of them on its own virtual machine. VM runs versions of the IBM operating systems, the Conversational Monitor System (CMS) being one of them.

The main components⁸ of VM are:

1. the Control Program - (CP);
2. the Conversational Monitor System (CMS);
3. the Remote Spooling Communication Subsystem (RSCS);
4. the Interactive Problem Control System (IPCS);
5. the CMS Batch Facility.

CP creates the environment in which virtual machines can execute. It provides the support for the various operating systems normally used to control actual System/360, System/370, and other compatible computer systems.

CMS is an application system with powerful features for interactive development of programs. It contains editors, language translators, various applications packages, and debugging tools.

Initially CMS stood for Cambridge Monitor System, but more recently it is called Conversational Monitor System⁸. It is a disk-oriented operating system that gives the user complete access to what appears to be a dedicated real machine but which is, in fact, a virtual machine created by CP.

Sharing of data between virtual machines is controlled by CP that provides for multiple virtual machines to access common disks.

All CMS commands available to the user are stored in disk files:⁸ they are not actually part of the system. This makes it easy for users to tailor the instruction set of their virtual machines to their own needs by adding, deleting, or changing the command files.

CMS gives the user powerful program development capabilities. A CMS user sees a dedicated virtual machine. VM provides a multiuser time-sharing environment by supporting many separate CMS virtual machines.

The user communicates with CMS via a terminal and with a series of commands, rather than with job control statements. CMS presents a user-friendly interface. It provides particularly useful diagnostic messages directly to the interactive user so that errors may be immediately corrected.

CMS Command Language

The CMS command language allows one to create, modify, and debug problem or application programs and, in general, to manipulate data files.

Many OS language processors can be executed under CMS: the assembler, VS BASIC, OS FORTRAN, VS FORTRAN, OS/VS COBOL, and OS PL/1 optimizing and checkout compilers⁹.

CMS executes the assembler and the compilers where they are invoked with CMS commands.

When EDIT command is invoked, the System Product Editor places the user in CMS (EDIT) compatibility mode⁹. In this mode the CMS editor and the System Product Editor both allow the user to create and modify files. The CMS EXEC interpreter provides the execution procedures consisting of CP and CMS commands; they also provide the conditional execution capability of a macro language. The DEBUG Commands gives several program debugging subcommands.

The HELP command is used to display at terminal information on how how to use CP commands and CMS commands, subcommands, and EXECs, and explanations of CP and CMS messages; HELP command can be issued when a brief explanation of syntax, a parameter, or function is sufficient, thereby avoiding interrupting the terminal session to refer to a manual.

A CMS command consists of a command name, usually followed by one or more positional operands and, in many cases, by an option list. CMS commands have the format:⁹

```
command name / [operands...] [(options...[...])]
```

The command name is an alphameric symbol of one to eight characters. The command operands are keywords and/or positional operands.

The operands specify the information on which system operates when it performs the command function. The option list must be preceded by a left parenthesis; the closing parenthesis is not necessary.

EXEC

The EXEC command is used to execute one or more CMS commands or EXEC control statements contained in a specified CMS EXEC file called EXEC processor. The format of the EXEC command is:⁹

```
[EXEC] | fn [args...]
```

where

[EXEC] indicates that the EXEC command may be omitted if the user is executing the EXEC procedure from the CMS command environment.

fn is the file name of a file containing one or more CMS commands and/or EXEC control statements to be executed. The filetype of the file must be EXEC. A text editor or a user program can create EXEC files.

args are any arguments to be passed to the EXEC. The CMS EXEC processor assigns arguments to special variables &1 through &30 in the order in which they appear in the argument list.

The execution can be conditionally controlled with additional EXEC statements, or it may contain no EXEC statements at all. In

its simplest form, an EXEC file may contain only one record, have no variables, and expect no arguments to be passed to it. In its most complex form, it can contain thousands of records and may resemble a program written in a high-level programming language¹⁰. An EXEC procedure can be created that functions like a catalogued procedure, and can be set up to receive an argument, so that it executes somewhat differently each time it is invoked. As many as 30 arguments can be passed to an EXEC in this manner.

Creating EXEC files. EXEC files can be created with the CMS editors, by punching cards, or by using CMS commands or programs. Two CMS commands create EXEC files¹⁰. One is LISTFILE, which can be invoked with the EXEC option; it creates a file named CMS EXEC. The CMS/DOS command LIST10 creates an EXEC file named \$LIST10 EXEC, which creates records for each of the system and programmer logical unit assignments.

Invoking EXEC files. EXEC procedures are invoked when the file name of EXEC file is entered. An EXEC procedure having a synonym defined for it can be invoked by its synonym if the implied EXEC (IMPEX) function is on¹⁰. However, within the EXEC procedures, only the EXEC file name can be used. A synonym is not recognized within an EXEC since the synonym tables are not searched during EXEC processing.

A CMS EXEC is like any other CMS file; it can be edited, erased, renamed, or changed.

Summary of the CMS EXEC Language Facilities

The CMS EXEC processor, or interpreter, recognizes keywords that begin with the special character ampersand (&). Key words may indicate:

1. control statements;
2. built-in function;
3. special variables;
4. arguments.

Variables can also be defined in an EXEC file; the CMS EXEC interpreter can process them as long as they begin with an ampersand.

Arguments and variables. Most EXEC Processing is contingent on the value of variable expressions. A variable expression in an EXEC is a symbol that begins with an ampersand (&)¹⁰. When the EXEC interpreter processes a line and encounters a variable symbol, it substitutes the variable with a predefined value, if the symbol has been defined. Symbols can be defined in three ways: (1) when passed as arguments to the EXEC, (2) by assignment statements, and (3) interactively, as a result of &READ ARGS or &READ VARS control statement. Arguments can be passed to EXEC files when they are invoked.

The values of the arguments can be changed or values assigned using the &ARGS control statement.

The &READ ARGS control statement are used to enter arguments interactively. For example, if the EXEC file contains the line:

&READ ARGS

When this line is executed, the EXEC issued a read to the virtual machine so that user can enter up to 30 arguments, to be assigned to the variable &1, &2, etc.

Assignment statements. User-defined variable names begin with an amperstand (&) and contain up to seven additional characters¹⁰. These variables can contain numeric or alphameric data. EXEC variables can be defined and initialized in assignment statements. In an assignment statement, the first data item starts with an amperstand (&) and the second data item is an equal sign (=). The value of the expression on the right side of the equal sign is assigned to the variable named on the left of the equal sign.

Built-in functions and special variables. The EXEC built-in functions are similar to those of higher-level languages. EXEC built-in functions can be used to define variable symbols in an EXEC procedure. Table 1 summarizes the built-in functions.

The table shows, given the variables &A, the values resulting in a variable &B when a built-in function is used to assign its value.

Other Facilities in EXEC

Flow control in an EXEC. An EXEC is processed line by line: if a statement is encountered that passes control to another line in the procedure, execution continues there and each line is, again, executed sequentially. The control can be passed with a &GOTO control statement.

Table 1. Summary of CMS EXEC Built-in Functions.

Function	Usage	Example	&B
&CONCAT	Concatenates tokens into a single token.	&A = 123 &B = &CONCAT &A 55	12355
&DATATYPE	Assigns the data type (NUM or CHAR) to the variable.	&B = &DATATYPE &A	NUM
&LENGTH	Assigns the length of a token to a variable.	&B = &LENGTH &A	3
&LITERAL	Prohibits substitution of a variable symbol.	&B = &LITERAL &A	&A
&SUBSTR	Extracts a character string from a token.	&B = &SUBSTR &A 2 2	23

Source: IBM Virtual Machine/System Product: CMS Users Guide (Release 2), International Business Machines Corporation, 1980.

Comparing variable symbols and constants. In an EXEC, user can test whether a certain condition is true, and then perform the same function based on the decision. While testing a condition in an EXEC file, user can use either the mnemonic or the symbol to represent the conditions.

Doing I/O with an EXEC. User can communicate with the terminal using the &TYPE and &READ control statements¹⁰. &TYPE is used to display a line at the terminal. &READ control statement is used when the user wants to be able to enter data, variables, or control statements into the EXEC file while it is executing.

Monitoring EXEC procedures. Two EXEC control statements, &CONTROL and &TIME, control how much information is displayed at the terminal while EXEC file is executing. This display is called an execution summary. &TIME gives the timing information. The &CONTROL control statement allows one to specify whether certain lines or types of information are displayed during execution. By default, CP and CMS commands are displayed before they are executed.

Often, computer users take great care in determining an effective combination of computing functions for a frequently performed task. Once that is done, the combination of the function is given a name and stored on secondary storage, so that the person need not repeat those thought processes again. These procedures (called EXECs) may include useful default values for parameters as well as logic to make the system adapt to the user's environment, rather than requiring the user to adapt to the machine. Once someone has

constructed an EXEC, others can and do take advantage of it.

The naive user need not even be aware of the existence of the commands or parameters inside EXEC. By this process, complexity is reduced, typing errors are avoided¹¹, and the user-effectiveness of interaction increases.

CHAPTER IV

A BRIEF OVERVIEW OF TOPS-10

General

TOPS-10 (Time-Sharing Operating System) is the DEC System-10 operating system, a complex collection of subprograms¹². The part of the TOPS-10 interacting with users and their programs is called the MONITOR and is mostly resident in memory. The DEC System-10 is also equipped with non-resident software such as text editors, translators and utility programs.

TOPS-10 is a multiprogramming system¹²; that is, the system transfers control rapidly among a number of jobs, so that all the jobs appear to be running simultaneously.

Jobs can be run on the TOPS-10 operating system in two modes--interactive mode and batch mode.

Jobs that run in the interactive mode use a terminal to access the system. The language used to run a job is the command language.

The TOPS-10 Command Language

The commands that are typed in from the terminal are received and processed by the command language interpreter of the TOPS-10 monitor.

When the job is in the interactive mode, work is completed at two levels--monitor and user level. While at the monitor level, the command language interpreter processes each command that is typed, and the monitor prompts with a period (.). Every monitor command

should follow this prompt. Monitor command is ended by pressing the RETURN key (<RET>).

The format of each command is variable because many commands are followed by optional switches, arguments or values.

After typing some operating system commands, e.g., RUN, the job moves from monitor level to user level. To move back to monitor level, CTRL/C is typed. When the job is at the user level, communication is done with a program other than the operating system itself. Each program has its own set of commands and its own command interpreter. These are called system programs, which usually prompt with an asterisk (*).

At user level, control characters and special keys can have a different effect than they have at monitor level.

Command language control characters. The user, from the terminal, may transmit control characters to the computer. On the keyboard, a control character is typed simultaneously depressing the key designated "CTRL" and any alphabetic character key¹². Table 2 gives a description of the control characters that have special significance to the DEC-10.

Command formats. The general format of each command is a line of ASCII uppercase or lower case characters. Spaces and tabs after the TOPS-10 monitor prompt (.) and before a command name are ignored by the system. The commands are one to six characters long and past the sixth character, the system ignores it¹². Command arguments have

Table 2. DECsystem-10 Terminal Control Characters.

Keyboard Character	Echo	Description
CTRL-C	$\wedge$ C	Abort the current process
CTRL-H	backspace	Same as backspace key
CTRL-I	tab	Same as tab key
CTRL-J	Linefeed	Same as Linefeed key
CTRL-L	Formfeed	Same as Formfeed key
CTRL-M	Carriage Return	Same as Carriage Return key
CTRL-O	$\wedge$ O	Suppress output to terminal
CTRL-Q	no echo	Continue transmission (XON - complement of XOFF)
CTRL-R	no echo	Retype partial line
CTRL-S	no echo	Stop transmission (XOFF - complement of XON)
CTRL-T	no echo	Status information on current job
CTRL-U	$\wedge$ U	Kill partially entered line
CTRL-Z	$\wedge$ Z	End of file mark from terminal

Source: TOPS-10 Operating System Command Manual,
Digital Equipment Corporation, 1980.

to be specified after the command name. Some commands can be modified by including a switch in the command line. Each switch is preceded with a slash (/) and is terminated usually with <RET>, a comma, or another switch. Switch can be abbreviated if its name remains unique.

Comments can be typed on the same line as a command by preceding the comment with a semicolon (;) or exclamation point (!).

The operating system contains defaults for many arguments, switches and parts of file specifications. Defaults are the values or instructions that the system uses if these are not specified in the command string.

MIC Commands

MIC (Monitor Interpreted Commands) allows one to create a new command by writing any desired sequence of monitor-mode and user-mode commands in a command file¹³. MIC is particularly useful for a regular user of the system and who find it boring to frequently type the same sequence of commands. A single command can perform the functions of all commands. In computer jargon this is called a MACRO command¹³ and on the DEC-10 they are known as macro commands, MIC commands or MIC command files.

SOS or TECO can be used to create a MIC file with the MIC extension. MIC files may contain any number of commands, of any type, including macro commands. In order to make MIC commands more useful, MIC files may use many other facilities, including command arguments (or parameters) and switches in the command call, conditional and branching statements to enable jumps to labelled lines,

and assignment statements which allow parameters and variables to be assigned or altered during the processing of MIC command.

Creating MIC files. MIC files may be created by using a normal text editor. Each line should have as its first non-space or tab character one of the characters =, ., *, ! or ; which are interpreted as follows:¹³

- . monitor command follows;
- * data to be input to a program follows;
- = ignore carriage return at the end of this line;
- ! comment line follows;
- ; comment line follows.

In order to get control characters such as CONTROL-C, CONTROL-Z or altmode into a MIC file, the control key and the character are replaced by the up-arrow key followed by the character, e.g.,

^ c for Control-C .

The ' (single quotation mark) followed by a letter indicates a parameter in MIC.

Calling MIC files. MIC files are called into action with a command of the form¹³

.DO mic-filespec

where the device, ppn and file extension are optional.

Parameters in the MIC macro commands. More generalized commands may be constructed using parameters in the MIC file to stand for filenames or other quantities which would be entered in the MIC command call. The full specification of a MIC call is

```
.DO dev:mic filespec[ppn] parameter A, parameter B.....
```

A parameter may be any string of characters separated by commas¹³. Up to 26 parameters are allowed and within the MIC file itself they are introduced by 'A, 'B ...'Z i.e., a letter of the alphabet preceded by a single quote. During execution of a macro a single quote followed by the *nth* letter of the alphabet will always be replaced by the *nth* parameter in the MIC call.

Parameters are very useful when setting up MIC files which will act on filenames at run time; the parameters will then be filenames.

Labels and branching facilities. The normal sequential processing of a MIC file may be interrupted by using jump commands to labelled lines.

A line in MIC may be labelled by placing the label at the beginning of the line. A label consists of one to six alphanumeric characters¹³ and is terminated by a double colon(::).

Two jump statements are available:

```
.GOTO
```

and

```
.BACKTO
```

The GOTO statement is used for a forward branch and the BACKTO for a backward branch. ('Backward' here means as usual a search starting at the beginning of the macro.)

Error detection and processing. When a monitor or system program error occurs, an error message is normally output. Conventionally error messages are preceded by ? for fatal errors, and % for warnings. MIC, like the batch system, allows one to monitor this first character in the output line and take special action if the defined error character of the user occurs in this position. To use it, a command of the form

.ERROR character

is included to inform MIC which character in the first character position of the output line is to be regarded as preceding an error message. This single command allows one to introduce very simple error processing into MIC macro. Whenever the error character is detected, processing of the MIC file will cease and the message

[ABORT ON ERROR]

will be displayed. On the detection of this error, a specified action can take place, e.g., error recovery labels can be included at suitable points within the MIC macro.

Manipulating parameters in a command file. User's parameters to MIC command files are held within MIC as ASCII strings¹³. The LET

statement allows a user to change parameters and define new parameters during the processing of a MIC file by assigning a string or integer expression to the parameter thus:

```
.LET <parameter name> = <expression>
```

where

<parameter name> is a parameter to the MIC file either passed in the calling statement or newly constructed.

<expression> is either a string expression or a simple integer expression.

An integer expression may also have parameters and involve the arithmetic operators +, -, *, /, and ^ where the normal precedence rules apply.

Character subscripting facilities. Any single character or a group of characters in a string may be referred to individually by using the subscripting facilities. These allow a user to give a MIC command which will loop around a set of monitor commands substituting a different parameter from a sequence of parameters each time the loop is obeyed. A subscripted string expression used in a LET statement has the form¹³

```
$<parameter> . [<subscript 1> , <subscript 2>]
```

The first subscript specifies the first character to be extracted from the string contained in the parameter, the second subscript specifies the total number of characters to be extracted.

Negative subscripts may be used to count backwards from the end of the string.

Other Useful Facilities Found in MIC

The PLEASE command. A PLEASE command exists to enable a message to be transmitted to the user of a MIC file. The form of the command is

.PLEASE <message>\$

The message is displayed on the terminal when the PLEASE command is processed in the MIC file.

Suspending MIC processing. The processing of a MIC file may be suspended by typing CTRL-B. MIC will then output the message [BREAK] and await further action. At this stage, any monitor command or any MIC command may be typed in. To continue the processing of the MIC file, CTRL-P is typed. A CTRL-C typed in during the BREAK period will cancel the suspended MIC process.

Nested MIC process. MIC command files may themselves contain MIC commands. Each file may have its own parameters but inner processes may access the parameters in an outer process¹³.

CHAPTER V

A BRIEF OVERVIEW OF UNIX

General

UNIX has become one of the most widely known and imitated operating systems of all time. The name, UNIX, was a transformation of MULTICS, with MULTI- becoming UNI- and -CS becoming -X¹⁴. These transformations connoted the simplifications in the operating system, its orientation toward single-user systems, and its efficiency. UNIX was not written all at once, but rather evolved in response to more immediate needs¹⁵.

UNIX was originally written in B, a condensed version of BCPL¹⁴ (Basic Combined Programming Language). UNIX was rewritten in C, which was also developed at Bell Labs. C is a language suitable for systems programming--one that allows a programmer to express concepts clearly without being tied to one machine's architecture, and yet is efficient enough so that assembly language is not needed for speed. C is relatively high-level language, compilers can be written on other computers to give them C capability too. Also because UNIX was written in C, theoretically, UNIX could then be moved to other machines¹⁸. Originally UNIX was developed for the PDP-11 family of computers, later it was transported to other machines like Interdata, IBM/370, DEC-VAX and many other systems as well.

Until UNIX, operating systems were written exclusively in assembly language¹⁴. Compared to other languages, assembly language

allows the fastest execution of instructions and takes up the least memory space; therefore, programs as important as operating systems could only be written in assembly language. UNIX forever broke the notion that a system had to be written in assembly language and therefore tied to a specific computer design, word size, or architecture. For the first time, an entire programming environment, including file system, kernel, applications packages, utility programs, and user interface, could be moved to an entirely different type of machine.

UNIX or UNIX-compatible systems can be run on computers based on any of these processors:¹⁴ 8080, Z80, 8086, 8088, Z8000, 68000, 16032, LSI-11, PDP-11, VAX, HP-9000, Perkin-Elmer, Gould S.E.L., BBN C-Machine, IBM Series/1 and IBM/370.

The UNIX Shell

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 filenames or flags, may be passed to a command. A return code is set by commands¹⁹ 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 invoked²⁰. Commands are found by searching directories in the file system in a sequence that can be defined by the user.

Shell serves as the command interpreter. The critical point is that it is an ordinary user program²¹, not a part of the system.

The Shell makes it easy for the user to combine existing commands. The user is encouraged to write commands that are flexible and have more than one use. The user is encouraged to develop "shell scripts"²⁴ which are cataloged procedures combining existing commands.

In its simplest form, a Shell command line consists of the command name followed by arguments to the command, all separated by spaces:²³

Command arg1 arg2...argn

The Shell splits up the command name and the arguments into separate strings. Then a file with name *command* is sought; *command* may be a path name including the "/" character to specify any file in the system. If *command* is found, it is brought into core and executed. The arguments collected by the Shell are accessible to the command. When the command is finished, the Shell resumes its own execution, and indicates its readiness to accept another command by typing a prompt character.

If file *command* cannot be found, the Shell prefixes the string /bin/ to *command* and attempts again to find the file. Directory/bin contains all the commands intended to be generally used²⁰.

Standard I/O. Programs executed by the Shell, start off with two open files which have file descriptors²⁰ 0 and 1. As such a program begins execution, file 1 is open for writing, and is best understood as the standard output file. File 0 starts off open for reading.

The Shell is able to change the standard assignments of these file descriptors from the user's typewriter printer and keyboard. Thus, most commands use the standard I/O files for interactive communication with the user. The syntax:

```
Sh < filename
```

instructs the Shell to run the Sh command with the standard input switched to the file. This happens to be the name of the Shell itself. Thus this invokes a second Shell to read and execute the commands specified in the file. The syntax:

```
ls > output
```

instructs the Shell to run the ls command with standard output re-directed to a file called output. This file will be created by the Shell if it does not already exist²⁵. The ls (abbreviation of list) command writes an alphabetical list of the names of all of the files in the current working directory. It is totally unaware of the re-direction of the standard output that has taken place. No special coding to handle I/O redirection is needed within each command; the command need merely use the standard file descriptions 0 and 1 where appropriate.

Filters. An extension of the standard I/O notion is used to direct output from one command to the input of another. A sequence of commands separated by vertical bars causes the Shell to execute all the commands simultaneously and to arrange the standard output of each command be delivered to the standard input of the next command in the sequence. A program which copies its standard input to its standard output (with processing) is called a filter²⁶.

A filter is a command that reads its standard input, transforms it in some way, and prints the result as output. The *sort* command is a filter in that it accepts the standard input, sorts it according to the arguments provided and outputs the result.

Command separator: multitasking. Another feature provided by the Shell is relatively straightforward. Commands need not be on different lines; instead they may be separated by semicolons.

A related feature is more interesting. If a command is followed by "&", the Shell will not wait for the command to finish before prompting again; instead, it is ready immediately to accept a new command.

When the Shell does not wait for the completion of a command, the identification of the process running that command is printed. This identification may be used to wait for the completion of the command or to terminate it. The "&" may be used several times²⁶ in a line. The Shell also allows parentheses in these operations.

Shell procedures. The Shell is itself a command, and may be called recursively²⁰. It can execute the command lines contained in a file sequentially. 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, etc. UNIX files have three independent attributes--read, write and execute. This allows Shell procedures and programs to be used interchangeably. A frequent use of Shell procedures is to loop through the arguments (\$1, \$2, etc.) executing commands once for each argument. 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 compilation¹⁹, they are easy to create and maintain.

The Shell has further capabilities, including the ability to substitute parameters and to construct argument lists from a specified subset of the filenames in a directory. It is also possible to execute commands conditionally on character string comparisons or on existence of given files and to perform transfers of control within filed command sequences.

All this gives the Shell and the commands that it executes the appearance of a programming language.

Shell variables. The Shell provides string-valued variables¹⁹ 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 the name in the invoking Shell is not affected. Such names are sometimes called keyword parameters. Modification of such variables within the called procedure does not affect the values in the calling procedure.

A name whose value is intended to remain constant throughout a procedure may be declared *read only*¹⁹. Subsequent attempts to set *read only* variables are illegal.

The value of a variable may be substituted by preceding its name with `$`. The general notation for parameter substitution is

`$(name)`

The following variables²⁵ 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.
- `$-` The current Shell flags.
- `$_` The process number of the last process initiated.

Implementation of the Shell

Most of the time, the Shell is waiting for the user to type a command. When the new-line character ending the line is typed, the

Shell's read call returns. The Shell analyzes the command line, putting the arguments in a form appropriate for execute. Then *fork* is called. The child process, whose code of course is still that of the Shell, attempts to perform an execute with the appropriate arguments. Meanwhile, the other process resulting from the *fork*, which is the parent process, waits for the child process to die. When this happens, the Shell knows the command is finished.

When a process is created by the *fork* primitive¹⁹, it inherits not only the core image of its parent but also all the files currently open in its parent, including those with file descriptors 0 and 1. The Shell, of course, uses these files to read command lines and to write its prompts and diagnostics, and in the ordinary case its children--the command programs, inherit them automatically.

In ordinary circumstances, the main loop of the Shell never terminates.

Initialization. The instances of the Shell to which users type commands are themselves children of another process²⁶. The last step in the initialization of UNIX is the creation of a single process and the invocation (via execute) of a program called *init*. When *init* is invoked, the various subinstances of *init* open the appropriate typewriters for input and output. Each process types out a message requesting that the user login and wait, reading the typewriter for a reply. Finally, someone types his name or other identification. The appropriate instance of the *init* wakes up, receives the log-in line, and reads a password file. If the user's

name is found, and if he is able to supply the correct password, *init* changes to the user's default current directory²⁶, sets the process's user ID to that of person logging in, and performs an *execute* of the Shell. At this point the Shell is ready to receive commands and the logging-in protocol is complete.

Meanwhile, the mainstream path of *init* (the parent of all the subinstances of itself which will later become Shells) does a *wait*. If one of the child processes terminates, either because a Shell found an end of file or because a user typed an incorrect name or password, this types another log-in message. Thus a user may log-out simply by typing the end-of-file sequence in place of a command to the Shell.

Other Programs as Shell

The Shell is designed to allow users full access to the facilities of the system since it will invoke the execution of any program. Sometimes, however, a different interface to the system is desirable, and this feature is easily arranged.

After the user has successfully logged in by supplying his his name and password, *init* ordinarily invokes the Shell to interpret command lines. The user's entry in the password file may contain the name of the program to be invoked after login instead of the Shell. This program is free to interpret the user's messages in any way it wishes.

CHAPTER VI

COMPARISON OF 'MACRO' CAPABILITIES OF
FOUR COMMAND LANGUAGES

Four command languages are compared in the light of 'macro' capabilities. The analysis is directed at distinctions and similarities of the command languages. The comparison focuses on those dimensions that bear on procedures composed of commands and could be useful in software development tools to be used in CACSD.

The reason for evaluating command languages is to help select one for use in a Computer Aided Control System Design project.

There has been no evaluation of operating system command languages so far--particularly from the 'macro' point of view. The study is done considering the features that should be found in a procedure and *not* the command language itself.

The original objectives of this study were:

1. to identify language features currently in use, to evaluate them in terms of their specification and scope of effects, their use in practice, and their suitability to the user's needs;
2. to enumerate language features conducive and detrimental to good macro capability, and to identify small variations that may have significant impact.

A comparison of 'macro' features of RT-11, CMS, TOPS-10 and UNIX command languages is made in Table 3.

Table 3. Comparison of 'Macro' Features of Four Command Languages.

	RT-11		CMS		TOPS-10		UNIX	
	Indirect Files		Execs		MIC		Shell	
Language Basis	Assembler	Assembler	Assembler	Assembler	Assembler	C		
Macro Type	Command file	Processor	Processor	Command file	Command file	Interpreter		
Invoking 'Macro'	.@ INDCT	[EXEC] Fn (Args...)		DO dev:micfilespec [ppn] parameters...		Sh file arguments...		
Variables	NONE	Begins with &, arg are assigned variables		---		String valued, up to 26 a + z Also special variables		
Procedures Structured	No	No	No	No	No	Yes		
Compilation	Yes	---	---	Yes	No	No		
Labels	---	---	---	Six characters, start of line, end with :	---	---		
Control Structures	---	&IF, &GOTO, &HEX, &ARGS, &EXIT, &CONTROL, &SKIP, &LOOP, &END		GOTO/BACKTO, Label:, IF		GOTO/Case/For/If If-then-else/while		
Built-in Functions	---	used to define variables &CONCAT, &DATATYPE, &LITERAL, &SUBSTR, &LENGTH		---		Built-in commands to serve interrupts, e.g., cd, trap, etc.		
Arguments, Parameters	---	Up to 30, assigned &l - &o		Up to 26 parameters 'A - 'Z		Up to 10, additional arguments can be passed using shift command		
Parameter Substitution	---	Assignment statement is used for substitution		'X, x is an alphabetic character, nested substitution allowed		\$(name) nested substitution allowed		
Command Substitution	---	---		---		'.....' allows string processing commands		
File Name Generation	---	LISTFILE is used to generate any form of files		---		*?,[...] for any pattern characters		

Table 3. (continued)

	RT-11		CMS		TOPS-10		UNIX	
	Indirect Files		Execs		MIC		Shell	
Interrupt Handling	SET, abort on error, Error message is displayed, Execution terminated	---	Displays a message, &RETCDD, &ERROR, &CUTO, &EXIT	---	Message displayed preceded by ? or % if (error) IF (no error) conditions can be set	---	error message printed, Shell flag - e causes Shell to terminate, built-in commands can be used to serve interrupts.	---
Filters and Pipeline	---	---	---	---	---	---	Yes Selection of line according to pattern, sorting of input, encryption and decryption, etc.	---
I/O Redirection	---	---	---	---	---	---	Yes	---
Multitasking	---	---	---	---	---	---	& is typed after the command, Shell is executing the command and ready to accept next command	---
Monitoring Procedures	Command before being executed is displayed	---	&CONTROL, &TIME	---	After execution, each command is displayed	---	---	---
Interactive use of Procedures	Yes	---	&READ control statement &TYPE	---	Response, Display, Type, What, Sleep, Input and Operator Commands	---	&MAIL etc.	---
Debugging of Procedures	---	---	EXEC 2 provides these facilities	---	---	---	-v flag, -n flag, -x flag	---
Procedure CL a part of system	Yes	---	Yes	---	Yes	---	No	---
Arithmetic Functions	---	---	Multiplication & Division (EXEC 2)	---	---	---	---	---

Comparison of Language Features

In the analysis of a macro, it is essential to consider several factors in addition to specific language features. More general in nature than the precise syntax and abilities of a language, those factors are of equal importance to its success in CAD environment. A command language macro should have the following characteristics.

Clarity and simplicity. RT-11 indirect files are clear and simple, but it does not have the 'macro' capabilities of other command languages.

MIC has a number of commands but few control constructs. Although the commands themselves are simple and clear, error detection and processing, parameter handling in procedures are not. EXEC has too many commands. The EXEC procedures or files have to be created with LISTFILE, LIST10 commands with EXEC option¹⁰. There are a number of control statements also.

Shell implemented in C (which is a structured language²⁹) was developed with consistent command language characteristics in mind. The use of structured facilities in macro eases the demand for specific commands in the language. Every file is a callable routine; there is no distinction between 'macro' files and other programs.

Clarity of 'macro' structure. Structured programming techniques were developed to make programs easier to comprehend and debug²⁷. The simple conditional and unconditional jump statements were replaced by English-like structured condition statements²⁸,

such as while-do and if-then-else, which allow the user to use more familiar logic to produce and debug macros. Shell employs these constructs. MIC and EXECS, on the other hand have only simple GOTO and IF statements. This restriction decreases the clarity of the macro structure itself. However, structured programming techniques, being more sophisticated, require more time to master than simple jump statements²⁷. For a trained computer expert, structured programming techniques are essential, although for CAD programmers, structured techniques could prove less cost-effective.

Ease of extension. It is extremely important that command languages should be easily extensible. They should have a modular, expandable structure²⁷ that both meets the needs of macro today and can easily be extended to handle the needs of the future, as new applications for macros are discovered. The most vital way to facilitate expansion capability is to provide user defined subroutines that allow the definition of further commands. Parameter passing is fairly essential to expandability to avoid the reuse of global variables. Subroutines providing expandability should be allowed to be nested to several levels in order to permit the macro to grow upon itself.

RT-11 indirect files are difficult to expand because they have no branching features. MIC files are contained in files with the extension MIC. A language of many commands from which the user is allowed a small subset or procedure, the Shell is constantly expanding to meet the specific needs of different applications.

Shell also allows user-defined subroutines that become user defined commands. Being structured files³⁰, it has powerful subroutine capabilities (including parameter passing and nesting to several levels). EXEC has the same subroutine-calling capabilities as the Shell, but as it is not structured, it has its limitations.

Error detection and fault handling. Efficient error detection and fault handling facilities are also of extreme importance in the execution of macros. The treatment of errors detected should depend on the type of error and whether the macro is being used inter-actively at such time.

RT-11 indirect command files abort the execution on ERROR. Error message is displayed if already set⁶. MIC also aborts the execution on error. An error message is normally output. In addition, it can take specified actions on occurrence of faults, e.g., labels and error conditionals. EXEC has also the same facilities, a return code can be set for the execution of a command in a procedure. Shell handles the errors and interrupts in a reasonable manner. An error message is displayed if the execution of a command fails due to failure of input-output redirection, nonexistence of command, abnormal termination²⁰ of command, etc.

All remaining errors cause the Shell to exit from a procedure. Such errors include syntax errors, terminal interrupts, built-in command failure. Shell flag-e causes the Shell to terminate if any error is detected.

Evaluation and quoting. Parameter substitution, command substitution and file name generation are of extreme importance in the development of procedures. A macro processor should be able to handle the order in which these substitutions occur and effects of various quoting mechanisms²⁶.

RT-11 indirect files do not provide any such facilities. MIC has system parameters; other parameters can be defined which are substituted at the time of processing¹³. They are introduced in MIC file by an alphabet preceded by a single quote. In addition, there are action parameters available. EXEC is processed line by line--when it encounters a variable symbol, it substitutes the variable with a predefined value, if that symbol has already been defined.

In Shell, before a command is executed, the evaluation occurs in the following way:¹⁹

1. parameter substitution, e.g., \$user;
2. command substitution, e.g., 'pwd.'

The Shell does not rescan substituted strings. After the substitution has occurred, the resulting characters are broken into words. 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 < >, +, ?, | have a special meaning to the Shell¹⁹. 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 commands.

Additional characteristics of a command language have been discussed in Chapter one of this report. Although these features are important to a good command language, they are important in building of procedures as well.

CHAPTER VII

UNIX AS A BASIS FOR CAD AND
SOFTWARE DEVELOPMENT

On the basis of observations, analyses, and comparisons, general conclusions can be formulated about 'macro' capabilities of command languages.

An operating system available on a wide range of hardware providing the facilities required to construct CAD systems would greatly improve the portability of CAD systems. Interactive CAD programs require a far wider range of services. They may need to manipulate terminal modes, change file access permissions, intercept signals, create and manage sub-processes, send and receive messages, etc.

A good command language is an invaluable tool for large CAD projects. A common occurrence in such projects is the diversion of significant resources from building the end product to creating internal support programs and procedures. If a command language is a flexible programming language, it can be used to solve many internal support problems⁵, without requiring compilable programs to be written and debugged.

TOPS-10 'MIC,' CMS 'EXECs' and UNIX 'Shell' include a range of structures and abilities. Various command language characteristics have been outlined in Chapter one. Some of those qualities in a command language are:

1. ease of on-line use;
2. convenient use of the same language as an on-line command language and as a programming language;
3. interchangeability of programs and command language procedures;
4. ease of creation and maintenance of command language procedures;
5. organization of command language procedures;
6. programming language features;
7. separation from operating system.

Although TOPS-10 'MIC' and CMS 'EXEC' include some of the above mentioned qualities, they lack in others, as discussed in Chapter six on the comparison of different command language macros.

UNIX Shell being a command language and a programming language, has shown it to be an effective tool for automating procedures and processes. It provides wide range of control structures, redirection of input and output, interrupt handling, creation of Shell processes and independence of the operating system.

Some Basic Facilities for CAD

File System. The file system is the mechanism whereby the operating system stores and retrieves information for users. The UNIX file system is fairly conventional, but extremely simple. It consists of a rooted tree of files, each interior node of which is a directory (a file containing names and addresses of files), and

each of which is either a file or directory³¹. Normally each user has a "home" directory, in which he or she creates files (programs, data, documents), and perhaps other directories to help organize large collections of files. There are also directories of system programs available to everyone. User sees no tracks, cylinders, blocks, or other device characteristics that typify commercial operating systems.

Because the file system accesses large numbers of small files efficiently, and because the same file may appear in different directories under different names (multiple links), it is possible to use the file system to provide multiple access paths to application data. Data stored in this way is easily accessible to programs written in the Shell language, which is an extremely quick way to create new applications.

Processes. The UNIX environment encourages the construction of applications from collection of small programs communicating via files and pipes, rather than as large, monolithic programs³². Each user may have many simultaneously executing processes. Executing a new program is a frequent operation.

In UNIX a process is simply a unit of work, be it a command or a subprocedure spun off from a command. UNIX allows programmers to connect different processes in a logical chain called a *pipe*. At each end of the pipe there needs to be a starting point and a destination (a place to store result), which may be indicated by

redirect commands. Thus, one need not worry about buffering, formatting, or other nagging concerns.

Input/output redirection. Normally, input and output for a program takes place on the user's terminal, but the Shell can be told to change either assignment to aim it at a file when the program is executed. Input and output devices are handled in the same manner as ordinary files²⁸. Device files are read and written like ordinary disk files, except that reference to a device file activates the device and passes data to or from it by whatever protocol is appropriate. A new device is added to the system by writing a device driver (in C) to make the device look like another file.

Programs designed to read and write files can read and write terminals with no special preparation, and *vice versa*. The I/O redirection facilities of the Shell makes this usual.

Terminal facilities can be used to build interactive graphics systems, spoolers for various plotters and printers, and computer-to-computer links. The complete control over special characters, and the acceptable efficiency of character-at-a-time I/O are very useful.

Software Development Tools

The interfaces to the basic system capabilities of UNIX, particularly input/output system, are strikingly simple compared to those of other systems. A ten-line C program suffices to copy any file in the file system to any other²⁸. Indeed, since peripheral

devices (tapes, printers, terminals, autodialers) are also treated as files, the same program can handle utility functions like tape to printer, interuser communications and telephone calls.

The Shell is an ordinary program--not part of the system kernel, so it may be invoked explicitly. This has some interesting consequences when the Shell takes its input from a file instead of a terminal.

Shell is a programming language in its own right, with variables, control flow, subroutines, and interrupt handling. As the Shell has become more powerful, there has been a steady trend toward writing complicated sequences in the Shell rather than in C. Since the search path that the Shell uses to find programs can be set by each user for himself, most users have a directory of their own private commands that is searched before the normal ones. In this way, users can tailor the environment to their own preferences.

Many applications programs can be organized as language processors. They recognize some structured input and perform actions based on it. UNIX provides several tools for programming language development, including a compiler-compiler called YACC and lexical-analyzer generator called LEX (11)³³. The syntax of a language is specified by a grammar, with semantic actions written in C and attached to the rules of the grammar.

Flexibility and ease of change. One strength of UNIX is the degree to which it can be adapted to different requirements and

environments. The use of search paths, and indeed shell programs in general, makes it possible to change the actions of commands easily.

Any program that is not part of system kernel can be replaced by a user with one of his own. The Shell itself is the most obvious example--since it is just a user program, any user can create his own Shell. Many systems have several Shells in coexistence. Since the kernel itself is essentially all in C, it, too, is relatively easy to change; consequently, there are also multiple versions of the UNIX kernel.

Portability. Software written in assembly language (as most operating systems are) is forever wedded to one kind of machine²⁸. By contrast, software written in a high-level language like C is potentially portable. Once a C compiler is available for a new machine, the UNIX software can be moved to the new environment with substantially less effort than would be required to duplicate it from scratch.

The economic advantages of portability are great. It is highly desirable to run the same software on a variety of machines, to make use of available hardware, to avoid being tied to obsolete hardware that is no longer cost effective, and to avoid being dependent on a single vendor.

Other Advantages

UNIX runs effectively on small machines, which makes it feasible for groups with small budgets. Furthermore, the facilities that UNIX provides--editing, text formatting, and keeping track of files, are all jobs that pervade programming, so one does not have to make radical changes in one's approach to programming to make effective use of the system. This has been particularly important for large software development projects in which target computer was already specified.

Applications of UNIX

Text processing. UNIX programs are used for preparing manuals, patent applications, and manuscripts for publication. For technical articles the UNIX system is about twice as fast as typewriter composition.

Software development. UNIX has been used to develop software for a large number of different computers.

Laboratory automation. Inexpensive microcomputers, acting as satellite to a standard UNIX system²⁸ control laboratory experiments and analyze and display results.

Information systems. It is easy for an individual, using standard UNIX tools such as the pattern scanning and processing language, to put together programs to retrieve information from small data bases.

Computer science education. UNIX has been popular in computer science departments because of its small size and clean structure.

Nonapplications

Real-time systems. UNIX was designed for a time-sharing environment in which users give commands to a system that does a significant amount of computation in response to each command. It was not designed for real-time control of high-speed equipment²⁸, in which responses must be made to critical inputs in milliseconds.

Large data bases. Similarly, UNIX was not designed to handle large volumes of high-speed transactions, as would be generated for example, by an airline reservation system.

Nonprogrammers. The interface between UNIX and users is by no means as elegant as the underlying system design²⁸. Professional programmers accordingly tend to be much more enthusiastic about the beauties of UNIX than casual users or nonprogrammers.

Conclusions

UNIX is by no means the end of the road in operating systems, but there are some technical lessons in its success for designers of future operating systems and other software. UNIX demonstrates that the right combination of ideas implemented straightforwardly can be remarkably effective. A simple file system is much easier to build than the traditional commercial ones and more convenient to use. A separate command interpreter is an excellent way to

organize command execution in a time-sharing system. Program inter-connection not only makes it easier to write and use programs, but seems to foster good design and a tool building attitude among its users. High level languages extract a moderate cost in space and time but pay off in comprehensibility, ease of change, and portability. UNIX environment provides all basic facilities required by CAD systems and software tools development, on a range of machines from 16-bit microprocessors to large mainframes.

LIST OF REFERENCES

LIST OF REFERENCES

1. Davis, W.S., Operating System A Systematic View, Addison-Wesley, Massachusetts, 1983.
2. Beech, D., "What is a command language?" Proceedings of the IFIP TC2.7 Working conference on command languages, pp. 7-34, 1979.
3. Hardy, T., Jr. "The syntax of interactive command languages: A framework for design," Software-practice and experience, pp. 67-75, Vol. 12, 1982.
4. Huckle, B.A., "Command language for inexperienced users," Proceedings of IFIP TC2.7 working conference on command languages, pp. 201-212, 1979.
5. J.R. Mashey, "Using a command language as a High-Level Programming Language," IEEE Software Engineering Conference, pp. 169-176, 1976.
6. Introduction to RT-11 Manual, Digital Equipment Corporation, March, 1981.
7. Gram, Chr., and Hertweck, F.R., "Command Languages: Design Considerations and Basic Concepts," Proceedings of IFIP working conference on command languages, pp. 43-67, 1974.
8. Deitel, H.M., An Introduction to Operating Systems, Addison-Wesley, Massachusetts, 1983.
9. IBM Virtual Machine/System Product: CMS Command and Macro Reference (Release 3), International Business Machines, 1983.
10. IBM Virtual Machine/System Product: CMS Users Guide (Release 2), International Business Machines Corporation, 1980.
11. Daherty, W.J., and Kelisky, R.D., "Managing VM/CMS systems for user effectiveness," IBM SYST J., pp. 143-162, Vol. 18, No. 1, 1979.
12. TOPS-10 Operating System Command Manual, Digital Equipment Corporation, 1980.
13. MIC-Reference Manual - Edition 8.1, University of York Computing Service, May, 1982.

14. Fielder, D., "UNIX Tutorial PART 1: An Introduction to Features and Facilities--A look at some of the programming tools available to a UNIX user," BYTE, pp. 186-211, August, 1983.
15. Plauger, P.J., and Krieger, M.S., "UNIX-like software runs on mini- and microcomputers," Electronics, pp. 125-129, March 24, 1981.
16. RT-11 System Users Guide, Digital Equipment Corporation, March, 1981.
17. RT-11 Installation and System Generation Guide, Digital Equipment Corporation, March, 1981.
18. Johnson, R.C., "A short history of UNIX," Electronics, p. 126, March 24, 1981.
19. Bourne, S.R., "UNIX Time-Sharing System: The UNIX Shell," Bell System Technical Journal, pp. 1971-1990, July-August, 1978.
20. Ritchie, D.M., and Thompson, K., "The UNIX Time Sharing System," Communications of ACM, pp. 365-375, Volume 17, number 7, July, 1974.
21. Goble, C., "UNIX Commands simplified," Systems & Software, pp. 102-104, August, 1983.
22. Kernigham, B.W., and Mashey, J.R., "The UNIX programming Environment," Software--Practice and Experience, pp. 1-15, Vol. 9, 1979.
23. Thompson, K., "The UNIX Command Language," INFOTECH State of the Art Report, pp. 375-384, 1976.
24. Matthews, M.M., "Functional Aspects of UNIX," IEEE Southeast Conference Proceedings, pp. 91-93, April, 1983.
25. Bourne, S.R., "The UNIX Shell," BYTE, pp. 187-208, October, 1983.
26. Ritchie, D.M., and Thompson, K., "The Unix Time-Sharing System," Bell System Technical Journal, pp. 1905-1929, July-August, 1978.
27. Bonner, S., and Shin, K.G., "A Comparative Study of Robot Languages," Computer, December, 1982.
28. Kernighan, B.W., and Morgan, S.P., "The UNIX Operating System: A Model for Software Design," Science, pp. 779-783, Vol. 215, 12 February, 1982.

29. Kernighan, B.W., and Ritchie, D.M., The C programming language, Prentice-Hall, 1978.
30. Cole, B., "The UNIX operating system and its clones," Popular Computing, pp. 174-178, July, 1983.
31. Gauthier, R., Using the UNIX System, Prentice-Hall, Virginia, 1981.
32. Rosenthal, D.S., "UNIX as a basis for CAD software," Computer-Aided Design, pp. 131-135, Volume 14, number 3, May, 1982.
33. Johnson, S.C., and Lesk, M.E., "UNIX Time-Sharing System: Language Development Tools," Bell System Technical Journal, pp. 2155-2175, July-August, 1978.

APPENDICES

APPENDIX A

SUMMARY OF CMS EXEC CONTROL STATEMENTS
AND SPECIAL VARIABLES

Control Statement	Function
$\&$ variable = { string ae function X'xxxxxx' }	Assigns a value to the symbol specified by $\&$ variable; the equal sign must be preceded and followed by a blank.
$\&$ ARGS [arg1 [arg2 ...[arg30]]]	Redefines the variable symbols $\&$ 1, $\&$ 2... with the values of 'arg1', 'arg2', ..., and re-sets the variable $\&$ INDEX.
$\&$ BEGMSG [ALL] line1 line2 . . $\&$ END	Displays the following lines as CMS error messages, without scanning them.
$\&$ BEGPUNCH [ALL] line1 line2 . . $\&$ END	Punches the following lines in the virtual card punch, without scanning them.
$\&$ BEGSTACK [FIFO] [ALL] line1 [LIFO] [] line2 [] . . $\&$ END	Stacks the following lines in the console input buffer, without scanning them.
$\&$ BEGTYPE [ALL] line1 line2 . . $\&$ END	Displays the following lines at the console, without scanning them.
$\&$ CONTINUE	Provides a branch address for $\&$ ERROR, $\&$ GOTO, and other conditional branching statements.

Control Statement	Function
ECONTROL [OFF] [MSG] [TIME] [PACK] [ERROR] [NOMSG] [NOTIME] [NOPACK] [CMS] [ALL]	Sets, until further notice, the characteristics of the execution summary of the EXEC, which is displayed at the console.
EEMSG mmmnnns [tok1 [...tokn]]	Displays a line of tokens as a CMS error message.
END	Terminates a series of lines following an EBEGMSG, EBEGPUNCH, EBEGSTACK, or EBEGTYPE control statement.
ERROR [executable-statement] [ECONTINUE]	Executes the specified statement whenever a CMS command returns a nonzero return code.
EXIT [return-code] [0]	Exits from the EXEC file with the given return code.
EGOTO { TOP linenumber } -label	Transfers control to the top of the EXEC file, to the given line, or to the line starting with the given label.
EHEX { ON } OFF	Turns on or off hexadecimal conversion.
EIP { tok1 } { EQ NE } { tok2 } executable- { E\$ } { LT } { E\$ } statement { E* } { GE } = ~ = < < = > > =	Executes the specified statement if the condition is satisfied.
ELOOP { n } { m } -label } condition	Loops through the following n lines, or down to (and including) the line at label, for m times, or until the condition is satisfied.
EPUNCH [tok1 [...tokn]]	Punches the specified tokens to your virtual card punch.

Control Statement	Function
&READ [n] [1] [ARGS] [VARS [&var1 [...&var17]]]	Reads lines from the terminal or from the console stack. ARGS assigns the tokens read to the variables &1, &2 ... VARS assigns the tokens read to the specified variable symbols.
&SKIP [n] [1]	Transfers control forward or backward a specified number of lines.
&SPACE [n] [1]	Displays blank lines at the terminal.
&STACK [FIFO] [tok1 [... tokn]] [LIFO] [HT] [] [RT]	Stacks a line in the terminal input stack.
&TIME [ON] [OFF] [RESET] [TYPE]	Displays timing information following the execution of CMS commands.
&TYPE [tok1 [... tokn]]	Displays a line at the terminal.

Source: IBM, VM/SP: CMS User's Guide, International Business Machines Corporation, 1980.

CMS SPECIAL VARIABLES

Variable	Usage	Set By
%n	Arguments passed to an EXEC are assigned to the variables %1 through %30.	User
%* %\$	Test whether all (%*) or any (%\$) of the arguments passed to EXEC have a particular value.	EXEC
%DISKx	Indicates whether the disk access at mode 'x' is a CMS OS, or DOS disk, or not accessed (CMS, OS, DOS, or NA).	User
%DISK*	Contains the mode letter of the first read/write disk in the CMS search order, or NONE if no read/write disk is accessed.	User
%DISK?	Contains the mode letter of the read/write disk with the most available space or NONE, if no read/write disk is accessed.	User
%DOS	Indicates whether or not the CMS/DOS environment is active (ON or OFF).	User
%EXEC	Contains the filename of the EXEC file currently being executed.	EXEC
%GLOBAL	Has a value ranging from 1 to 19, to indicate the recursion (nesting) level of the EXEC that is currently executing.	EXEC
%GLOBALn	The variables %GLOBAL1 through %GLOBAL9 can contain integral numeric values, and can be passed among different recursion levels. If not explicitly set, the variable will have a value of 1.	User
%INDEX	Contains the number of arguments passed to the EXEC on the command line or the number of arguments entered as a result of an %ARGS or %READ ARGS control statement.	EXEC
%LINENUM	Contains the current line number in the EXEC.	EXEC
%READFLAG	Indicates whether (STACK) or not (CONSOLE) there are lines stacked in the terminal input buffer (console stack).	EXEC
%RETCODE	Contains the return code from the most recently executed CMS command.	CMS
%TYPEFLAG	Indicates whether (RT) or not (HT) output is being displayed at the console.	EXEC
%0	Contains the name of the EXEC file.	User

EXEC: Variables may not be modified.

CMS: One may assign a value to this variable but it is reset at the completion of each CMS command.

USER: Variables are assigned values by EXEC but one may modify them.

Source: IBM, VM/SP: CMS User's Guide, International Business Machines Corporation, 1980.

APPENDIX B

SUMMARY OF MIC COMMANDS

MIC Commands

Function

MIC allows you to create a new command by writing any desired sequence of monitor-mode and user-mode commands in a disk file. MIC commands are discussed in the file DOC:MICV2.DOC, on the system. MIC is not a supported product.

You may include any of the following commands in your MIC command file, along with any number of monitor commands.

Commands

BACKTO	Specifies a 1- to 6- character label at which MIC processing is to resume. Note that labels in MIC command files are terminated by :: and must be at the beginning of a line.
ERROR/ NOERROR	Specifies a character that is used to denote an error condition when displayed at the beginning of a line.
GOTO	Specifies a 1- to 6- character label at which MIC processing is to resume. Note that labels in MIC command files are terminated by :: and must be at the beginning of a line.
IF	Conditionally processes a monitor command.
LET	Allows you to change the value of any user parameters in a MIC file. For example, LET C= "THIS IS C" substitutes the text within double quotes as the value of C.
MIC Arg	Allows you to ABORT, SUSPEND, CANCEL, or EXIT from MIC processing.
OPERATOR/ NOOPERATOR	Specifies one ASCII character to be treated as introducing a line that requires user attention. For example, if the operator character is output in column 1, MIC suspends output and displays [BREAK]. After this, type the requested information. Then type CTRL/P (PROCEED), which will continue the processing of the command file.
SILENCE/ REVIVE	Suppresses and resumes output to the terminal. You can type another monitor command on the same line as the SILENCE/REVIVE command. For example, .REVIVE.DIR causes terminal output to be revived after the .DIR command has been typed. Therefore, only the output from the .DIR command will be printed on the terminal.
WHENEVER/ ON	Changes the default action whenever a particular action or event occurs. ON and WHENEVER are equivalent commands.

Source: TOPS-10 Operating System Command Manual,
Digital Equipment Corporation, 1980.

APPENDIX C

GRAMMAR OF UNIX COMMAND LANGUAGE

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 comand-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

Source: Bourne, S.R., "UNIX Time-Sharing System:
The UNIX Shell," Bell System Technical
Journal, pp. 1971-1990, July-August, 1978.

UNIX COMMAND LANGUAGE 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 <i>here</i> 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

Source: Bourne, S.R., "UNIX Time-Sharing System:
The UNIX Shell," Bell System Technical
Journal, pp. 1971-1990, July-August, 1978.

SPECIAL VARIABLES IN

THE SHELL

(a)	
\$?	The exit status (return code) of the last command executed as a decimal string. Most commands return a zero exit status if they complete successfully and return a nonzero exit status otherwise. Testing the value of return codes is deferred until the section on if and while commands.
\$0	The name of the command procedure being executed. This variable can be used to distinguish cases when a command has more than one name. For example, the following script is called <code>filek</code> and a link to the same file exists called <code>lfilek</code> . If the file has any other name, and is executed by that name, the default case applies. The command sets terminal options using the <code>stty</code> command. <pre> case \$0 in lfilek) stty erase ^H; kill @ 100; tabs ;; filek) stty erase ^H; kill @ 100; tabs ;; *) stty erase ^H; kill @ ;; esac </pre>
\$*	The number of positional parameters (in decimal), used, for example, in the <code>append</code> command to check the number of parameters. <code>\$*</code> is also updated by the <code>set</code> command.
\$\$	The process number of this shell (in decimal). Because process numbers are unique among all existing processes, this string is frequently used to generate temporary filenames. For example, <pre> ps a > /tmp/ps\$\$ rm /tmp/ps\$\$ </pre>
\$!	The process number of the last process run in the background (in decimal).
\$-	The current shell flags, such as <code>-x</code> and <code>-v</code> .

(b)	
\$MAIL	When used interactively, the shell looks at the file specified by this variable before it issues a prompt. If the specified file has been modified since it was last looked at, the shell prints the message <code>you have mail</code> before prompting for the next command. For user <code>fred</code> , this variable is set as <code>MAIL=/usr/spool/mail/fred</code> .
\$HOME	The default argument for the <code>cd</code> command. The current directory is used to resolve filename references that do not begin with a <code>/</code> and is changed using the <code>cd</code> command. For example, <code>cd /usr/lib/bin</code> makes the current directory <code>/usr/lib/bin</code> . <code>cd</code> with no argument is equivalent to <code>cd \$HOME</code> .
\$CDPATH	The list of directories searched by <code>cd</code> . Each directory name is separated by <code>:</code> . A typical setting of this variable, <code>CDPATH=.:\$HOME/bin</code> , specifies that <code>cd</code> should search the current directory, the parent directory, <code>..</code> , and <code>\$HOME/bin</code> . If the directory <code>.src</code> exists, and there is no <code>src</code> directory in the current directory, <code>cd src</code> changes the directory to <code>.src</code> and prints this string as confirmation.
\$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 <code>\$PATH</code> is not set, the current directory, <code>.</code> , <code>bin</code> , and <code>/usr/bin</code> are searched by default. <code>\$PATH</code> consists of directory names separated by <code>:</code> . For example, <code>PATH=.:bin:\$HOME/bin:/usr/bin</code> specifies that the current directory (the before the first <code>:</code>), <code>bin</code> , <code>\$HOME/bin</code> , <code>bin</code> , and <code>/usr/bin</code> are to be searched in that order. In this way, individual users can have their own commands in <code>\$HOME/bin</code> accessible independently of the current directory. The directory <code>bin</code> allows access to any directory named <code>bin</code> from the current directory. This separates commands from data files within a directory associated with some project or activity. If the command name contains a <code>/</code> , the directory search is not used, a single attempt is made to execute the command. The form <code>&cmd</code> may be used to bypass the search path for command in the current directory.
\$PS1	The primary shell prompt string, by default, <code>\$</code> .
\$PS2	The shell prompts with <code>\$PS2</code> when more input is needed; by default, the value is <code>></code> .
\$IFS	The set of characters used for blank interpretation.

In part (a), the variable `$?` is set after each execution of a command; all others are set initially by the Shell. The variable in part (b) are typically set in the file `.profile` in the user's home directory.

Source: Bourne, S.R., "The UNIX Shell," BYTE, pp. 187-208, October, 1983.

SUMMARY OF UNIX COMMANDS

Archive and Library Maintainer	
ar option [posname] archive_file [file1...]	4000 execution mode user
options	2000 execution mode group
d delete files from lib	0400 read permission
r replace files in lib	0200 write permission
t print files in lib	0100 execute permission
x extract files from lib	0070 group read,write,execute permission
v verbose	0007 all others
c create lib	Change Owner
Build Special Files	chown owner file...
/etc/mknod name [c] [b] major minor	Clear i-nodes
-c file is character file	clri filesystem i-number...
-b file is block file	Compare Two Files
major minor specific driver	comm [-[123]] file1 file2
Change Group	123 represent columns displayed
chgrp group file...	Compare Two Files
Change Mode	cmp [-l][-s] file1 file2
chmod option file...	-l provide complete diff table
options	-s return code based on results
	Concatenate Files

cat [-u] [file1...]

-u specify block size other

Construct a File System

/etc/mkfs [filesystem] [block count]

Convert and Copy a File

dd [option-value]...

options

if= input file

of= output file

bs=n set both ip/op block size

skip=n skip n ip records

files=n copy n files from tape

seek=n seek n rec's

count=n copy only n ip rec's

conv=

ascii EBCDIC to ASCII

ebcdic ASCII to EBCDIC

lcase map to lower case

ucase map to upper case

Copy Files

cp file1 file2

or

cp file1...filen directory

Determine File Type

file filename

le1...filen directory

Diff Between Versions of a File

diff3 [-ex3] file1 file2 file3

-e output accepted by editor

-x3 only differences in file3

Differences Between Two Files

diff [-efbh] file1 file2

-e produces editor commands

-b ignore trailing blanks and tabs

-f produce script of diff's

-h faster check

Disk Free Space

df [file system]

Dismount a File System

/etc/umount filesystem

Echo Argument

echo [-n] [arg]...

-n eliminate new lines

Directory Consistency Check

dcheck [-i numbers] [filesystem]

numbers obtained from itself or icheck

Storage Consistency Check

icheck [-s] [-b numbers] [filesystem]

-s reconstruct new free list

-b issue error when block #found

Find Pattern Matches in Files

grep [option]... expression [file]

options

-v print lines not matching

-n line preceded by its line number

-y match lower case letters to upper

Generate Names from i-numbers

ncheck [-i numbers] [-a] [-s]
[filesystem]

-i produces path names

-s produces path's for special files

Get Terminal Name

tty

Incremental File System Dump

dump [option [argument...] filesystem]

options

f place dump on next arg file

u after completing, write date

0-9 dump level

s specify size of tape

d specify density in "BPI"

Incremental File System Restore

restor option [argument...]

options

f specify specific tape

r or R specify filesystem

x extract each named file

t prints date

y size of tape

List the Contents of a Directory

ls [-option...] name...

options

l long list

t sort by time

a list all entries(. and ..)

d status of named directory

Mount a File System

/etc/mount [filesystem [-r]]

-r mount as read only

Move Files

mv file1 file2

or

mv directory1 directory2

or

mv file1...filen directory2

Octal Dump

od [-bcdox] file [[+offset[.][b]]

c interpreted in ASCII

d interpreted in decimal

o interpreted in octal

x interpreted in hex

Permit or Deny Messages

mesg [n] [y]

Pipe Fitting

tee [options] [file]...

options

-i ignores interrupts

-a append output to file

Print and Set Date

date [yymmddhhmm[.ss]]

Print Calendar

cal [month] year

Print File

pr [option]... [file]...

options

-n n column output

+n start print nth page

-h next arg as page header

-wn width of page

-ln length of page

Process Status

ps [option...] [namelist]

options

a list all process from terminals

x list all process (system)

l long list

Remove Files

rm [options] file...

options

-f force all files to be deleted

-r delete all files & subdirs

-i interactive delete

Report Repeated Lines in a File

uniq [-options [+n] [-n]] [input]
[output]

options

u lines not repeated are output

d only repeated lines are output

Run a Command at Low Priority

nice [-number] command [arguments]	su [userid]
number 1-20 with 20 lowest priority	Suspend Execution for an Interval
Sending and Receiving Mail	sleep time
mail [login name]...	time is in seconds
Setting a Terminal	Table of Contents for Archive Files
stty [option...]	ranlib [archive file....]
Setting the Terminal Tabs	Tape Archiver
tabs [-n] [terminal]	tar [option] [name]
-n left margin not indented	options
Sort or Merge Files	r names written on end of tape
sort [-option...] [+pos1 [-pos2]]...	x names are extracted from tape
[-o name] [-T directory] [name]...	t list names
options	u added to tape if not there
b ignore leading blanks	0,...,7 select tape drive
f upper lower case letters	v print names
m merge	b blocking factor
n sorted by arithmetic value	Terminate a Process
o output file	kill [-option] processid...
u eliminate duplicates	option = signal number
Split a File into Pieces	Update the Super Block
split [-n] [file [name]]	sync
-n no. lines to be placed in output	Who is on the System
Substitute User ID Temporarily	who [who-file] [am i]

Word Count

wc [-lwc] [file....]

l count number lines

w count number words

c count number characters

Working Directory Name

pwd

Write to all Users

wall

Write to Another User

write user [ttyname]

Find Files

find pathname... option...

option

-name filename

-mtime n T if modified in n days

-print prints current pathname

Summarize Disk Usage

du [-s] [-a] [name...]

-s provide only grand total

-a generate entry for each file

Source: Gauthier, R., Using the UNIX System,
Prentice-Hall, Virginia, 1981.

VITA

Saleem Ur Rehman was born in Jhelum, Pakistan as the sixth child of a family with eight children. His mother, a former school teacher lives in Rawalpindi, twin city of Islamabad. He attended Government College Rawalpindi from where he received his first B.S. degree in 1974. In February 1973 he began his second B.S. in Electrical Engineering (Electronics). He graduated from The University of Engineering and Technology, Lahore, in November 1976. Saleem worked as an assistant design engineer from December 1977 to June 1981, with NESPAK-HARZINT a U.S.-Pakistan Collaboration company. After being admitted to the Graduate School at The University of Tennessee, Knoxville in June 1981, he accepted a teaching assistantship with the Electrical Engineering Department. He also participated as a research assistant in Computer-Aided Control System Design project of Oak Ridge National Laboratories from January 1983 to December 1983. He received his M.S. in Electrical Engineering in June 1984.

Saleem has always been a keen sportsman and actively participated in many extracurricular activities. He is a student member of IEEE and ACM.