INTRODUCTION TO DEC SYSTEM-10: TIME-SHARING and BATCH

THIRD EDITION

T. W. SZE

PROFESSOR OF ELECTRICAL ENGINEERING

UNIVERSITY OF PITTSBURGH

Printed in the United States of America

Library of Congress Cataloging in Publication Data:

Sze, T. W.
Introduction to DEC System -10

Pittsburgh, Pa. : Univ. of Pittsburgh

Library of Congress Catalog Card Number: 80-54311

CONTENTS

PREFACE OF THE THIRD EDITION

Completion of the Third Edition marked the tenth year since the book project first started. Materials of the First Edition were the results of organizing the class notes of a freshman course I developed and taught. The organization of the text was aimed in such a way that (1) materials were presented in several levels of depth so that a beginner can quickly acquire a basic skill, and (2) a subjective judgement was exercised in the relevancy of materials to the intended readers, who will use the computer as a tool in their fields but have no desire to become professional programmers.

The experience of using these materials, class notes and earlier editions of the book, seems to bear out this rationale. So the Second Edition simply updated the progress in the DEC-10 hardware and softwares. However, during the past few years, there have been very significant changes in the computer maturity of our student body in Engineering. High school instructions, microcomputer projects, hobby electronics all have contributed to this. As a result, the changes in the Third Edition involve a great deal more than just updating the changes and progress in DEC-10. Specifically:

(1) Three chapters in PIL and BASIC languages are deleted, and they are replaced by chapters in Program Debugging, Modeling and Simulation, and Computer Graphics. Only a summary of PIL is retained as an appendix in the Third Edition.

(2) The book is now sharply directed to the goal of using the computer as a system. Therefore, although FORTRAN is the fundamental programming language, the book is not intended to be a programming manual. At the School of Engineering, this book was used in a second course, after the students have their initial instructions in the FORTRAN language.

(3) In using the computer as a system, the book aims to remedy the most neglected and yet the most important phase of computer processing, namely, the debugging of a program. Many people still consider that as an art, and cannot be taught. The Third Edition makes a serious attempt on the study of program debugging. An entire chapter is devoted to that subject.

(4) The chapter sequence is re-arranged so that the front part of the text would be appropriate as a text, and the latter part as a reference. In addition, exercise problems have been added to help readers sharpen their skills.

As in the last two editions, I am most indebted to my family. In spite of their own busy professional and college schedules, my daughter Deborah and my son Daniel found time to read the manuscript and made both technical and grammatical suggestions. My wife Frances, beside being understanding and encouraging, took charge of style review and proof reading, and made suggestions that increased the readability immeasurably. Students and colleagues, too numerous to list, have been most helpful; their questions, suggestions and ideas were indispensable. Finally, I wish to acknowledge the Computer Center at the University of Pittsburgh for providing the facilities and environment that made this book possible.

November 23, 1980
Pittsburgh, Pennsylvania

T. W. Sze

CHAPTER 1

INTRODUCTION

1.1 Batch Processing versus Time-Sharing

Once upon a time, when a computer user wanted to run a program, he would have to go through the following steps:

(1) The user submitted his program and data deck to the Computer Center.

(2) The decks of cards submitted by different users were stacked together to form a batch, each deck with its proper identification. All jobs in one batch were then executed in one "run", hence the name "batch processing". The information on the punched cards in a batch were first copied into a reel of magnetic tape by means of a small and relatively inexpensive computer. The reason for this was that the card-input to the main computer was a slow and therefore expensive process.

(3) The magnetic tape so prepared became the input medium to the main computer. At the scheduled time, the jobs in the batch were run and the outputs (printouts, cards, tapes, etc.) were obtained. Sometimes the outputs were recorded on another reel of magnetic tape; then output printing may be done off-line so as not to slow down the computer operation.

(4) The outputs were returned to a designated place of the Computer Center for the users to pick up.

During the execution of a job in one batch, such as to compile and execute a FORTRAN program, each job had the undivided service of the entire computer, with all of its memory, input and output devices, supporting services and library routines. When the next job entered the computer, that job in turn received the total service of the computer for the duration of the job execution, however brief.

Economics and efficiency considerations have led to the techniques of multi-programming in batch processing, so that several programs may be executed interleavingly when devices required for execution are not in demand at the same time, or if a priority of queuing can be clearly established.

From the point of view of economy and machine efficiency, batch processing indeed represents the best computer utilization because it can serve a maximum of users within a given span of time. The prime consideration is then

the efficient usage of computing resources, even if it is done at the expense of efficient usage of user resources. Therefore, from the users' point of view, batch processing has many limitations.

The time interval between submitting a card deck to the Computer Center and retrieving the results, called the turn-around-time, may vary from several minutes to several days. Such long intervals are most frustrating to a user during the program preparation and debugging stages. A minor error of an incorrect punctuation mark in a program can cause a delay of hours or days. Once the grammatical errors are removed, it still requires many successive runs to remove logical errors. These consecutive runs cannot be hastened because the second run depends on the first, the third depends on the second, and so on. That made the debugging stage the most tedious and frustrating part of the program development.

Thus the early work in time-sharing research was motivated by correcting the tedious and frustrating process of debugging in the batch mode of operation. The reasoning that led to time-sharing was that the human responses and the output device responses are very slow in comparison to the logic and computing speeds of the computer; hence, it may be possible to switch the computer from one user to another and still seem to maintain a continuity at each user's station.

In the time-sharing mode of operation, a computer will service the jobs entered at remote terminals by sequentially giving a short period of time, called a time slice, to each job. Once that time slice is exhausted, that particular job is returned to the end of the queue to wait for another turn. In the meantime, a monitor program will perform the necessary bookkeeping and housekeeping tasks so that when that job receives a time slice again from the computer, the execution will pick up where it was left off.

From early 1960's when the time-sharing system concept was first developed, this mode of operation for a computer became widely accepted as an augmental mode of operation. However, before very long, it became quickly apparent that the major benefit is not the reduction of programmer frustration, but an entirely new dimension of problem-solving not possible before, utilizing a high degree of interaction between man and machine as a team. The language processors and programs may then be so designed that during the execution of a program, not only can error messages be sent to the user to aid his debugging, but also the user is able to modify his problem solving tactics and procedure as he sees the partial results along the way. It is possible then to design programs subject to modification by the user during execution time to adapt themselves to the condition of the problem.

Figure 1.1 shows a typical time-sharing computing system hardware organization. The configuration consists of a computer located at the Computer Center and the communication control, transmission and receiving equipment to connect the computer with the users at the remote terminals. The data line multiplexer and controller is used to control and direct the schedule of time-sharing activities. At the user's terminals, each terminal is connected to a data set or modem (modulator-demodulator) that converts the output signals from the terminal into a form suitable for transmission by the communication channel. The communication channels are usually commercial telephone networks, although in many cases telegraph lines and microwave channels are also used. The data set or modem at the receiving end re-converts the transmitted signals back to a form suitable for processing by the computer circuits.

It is also of interest to note that the remoteness of remote terminals is only limited by the quality and the economy of the communication equipment. At the present level of communication technology, it is commercially practical for

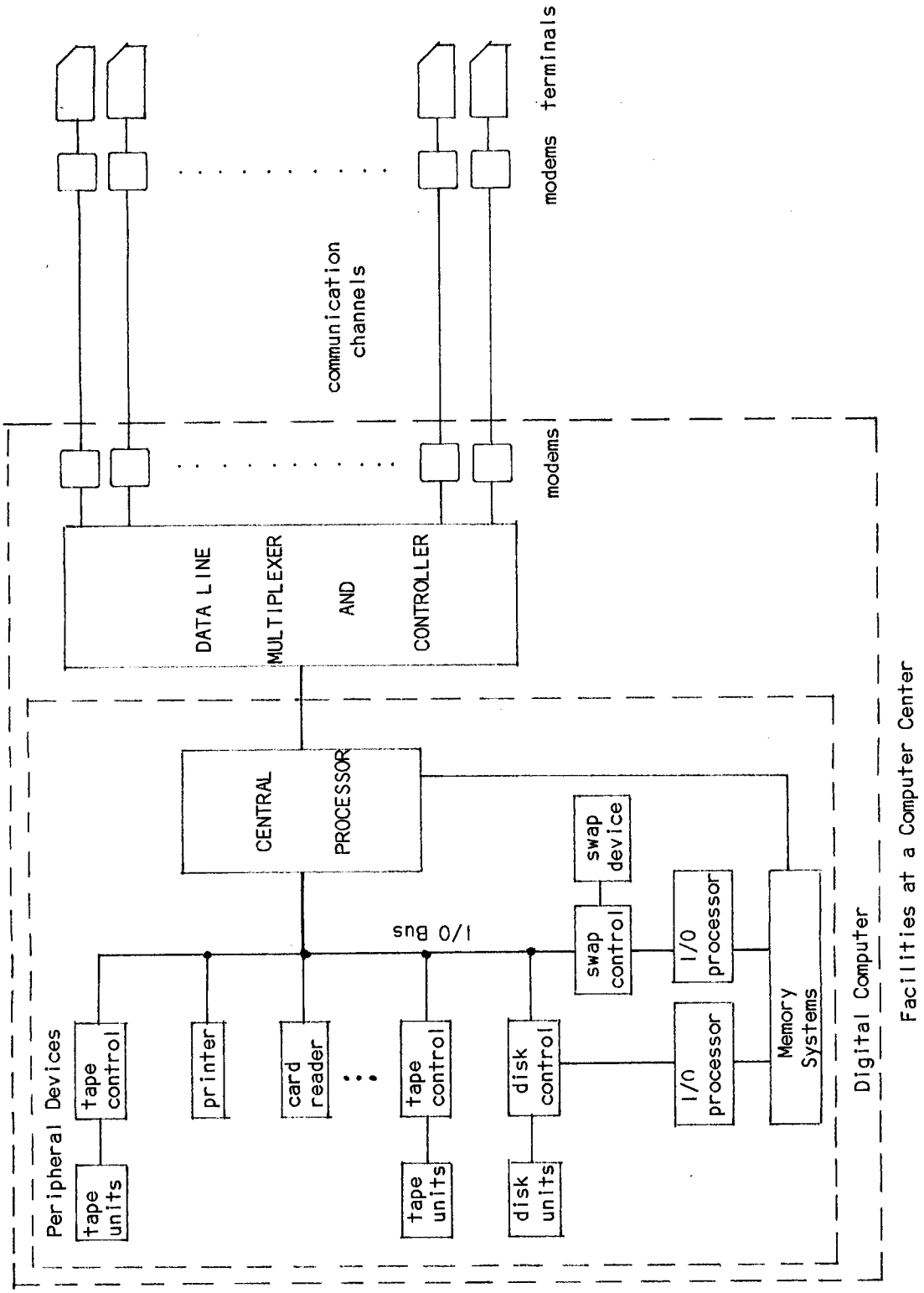


Figure 1.1 A Typical Time-Sharing System Hardware Configuration

a large, centralized computer to serve on demand users scattered over a wide area over the world. Thus through the time-sharing use of computer, an entirely new "utility" has emerged, just like electricity, gas, or water, to provide the users with the computer services when independent ownership of these services may be out of reach economically to these users.

1.2 Time-Sharing System at Pitt

University of Pittsburgh is one of early pioneers in the development of time-sharing computer system. Through a federal grant in 1965, the time-sharing facilities for the University community were established, using an IBM 360/50 system. Much of the software supporting facilities was developed in the subsequent years, resulting in a system then known collectively as the Pitt Time-Sharing System, or the PTSS.

In 1971, the time-sharing computer at the Pitt Computer Center was changed to a multiple PDP-10 system of Digital Equipment Corporation. This system has been upgraded and expanded several times, and the present configuration is a dual DEC-1099 system. Figure 1.2 shows the configuration of the system.

As in many similar environments, the current software system is a combination of vendor-supplied software and self-developed facilities. The readers are referred to the list of references at the end of this chapter for details of language processors and other software subsystems.

The software system of the time-sharing system contains, in addition to the language processors, a group of service routines. The most important one for the time-sharing operation is the executive system, also called a supervisor or a monitor. It is a master program which exercises an overall control on the time-sharing activities. It performs the scheduling of users from the queue, provides users with proper language processor and peripheral facilities as requested by the users, keeps an account of charges, and provides a variety of service functions.

Because of the control it exercises, the monitor is the highest-ranking program in the software system. The monitor controlls and dispatches a group of processors, collectively called the CUSP (Commonly Used System Programs), among which are the language processors such as BASIC and FORTRAN. In turn, under the control of each CUSP is a subgroup of routines for the execution and/or interpreting of the instruction set of the CUSP. Thus the software system has a distinct hierarchy structure, and this is shown in Figure 1.3.

There are several points regarding the software system structure worthy of note:

(1) There are three levels of hierarchy: the monitor level, the CUSP level, and the sub-CUSP level. The monitor level is the highest.

(2) It is a common practice in a time-sharing system for the computer to supply a prompting symbol through the user's terminal to indicate that the computer is ready to accept a command or input data. In the time-sharing system of DEC-10 system, different hierarchies use different types of prompt symbols:

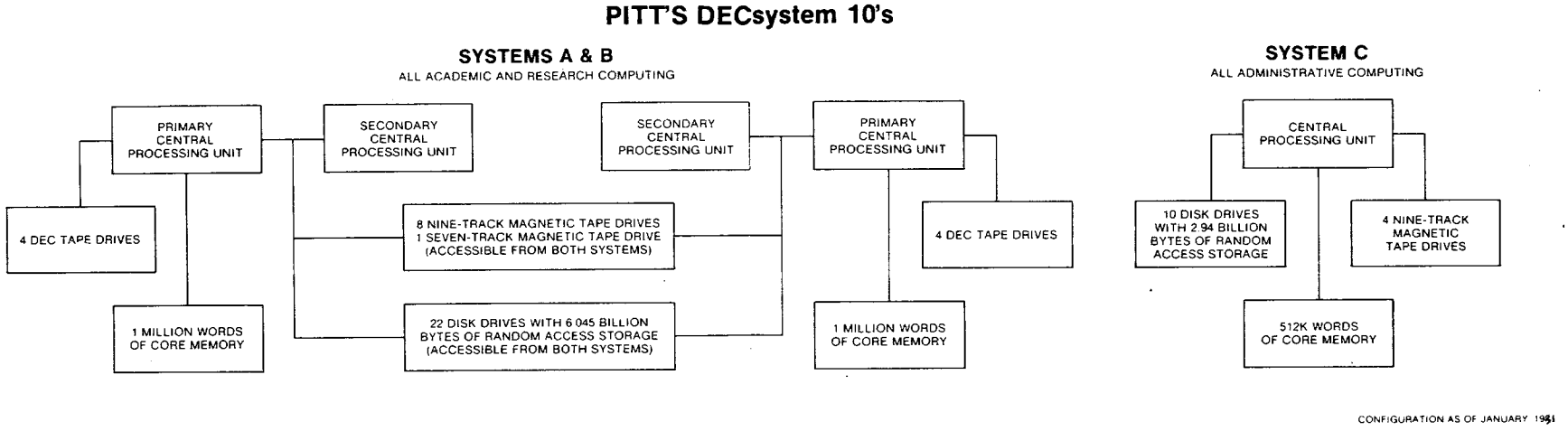


Figure 1.2 Configuration of a DEC-1099 System at Pitt

(Each central processing unit is a PDP-10 CPU)

Reprinted by permission, Reference 6
Computer Center, University of Pittsburgh, Pittsburgh, Pa.

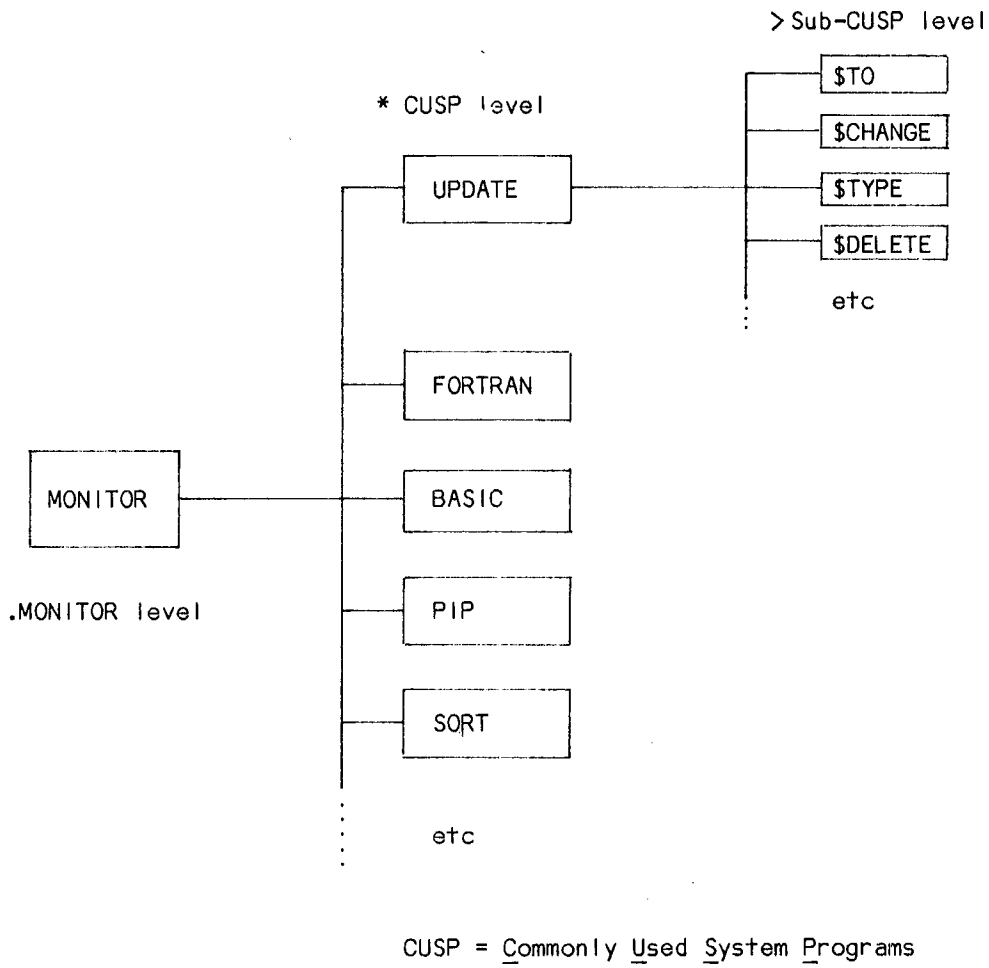
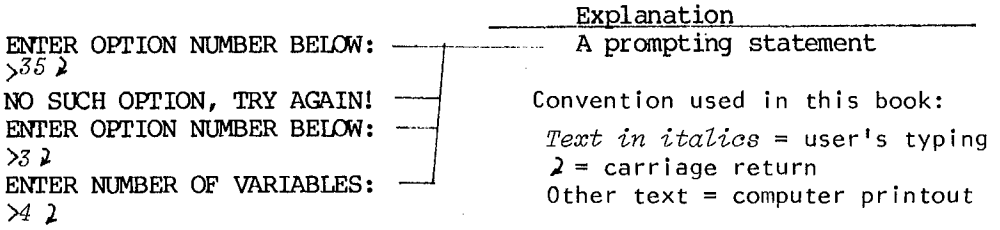


Figure 1.3 A Typical Time-Sharing Software System Organization

Prompt Symbols	Explanations	Hierarchy Level
.	A period	Monitor level
*	An asterisk	CUSP level
>>	Double ">" signs	
>	Greater-than sign	Sub-CUSP level
?	Question mark	

(3) When such a prompt symbol appears on the user's terminal, the computer is ready for a command or information, and the user must type in a command or data and terminate the typing with a carriage return. However, when

a program contains a number of such command/input breakpoints, it becomes difficult for the user to keep track exactly what command/input is expected at each breakpoint. It is therefore necessary for the program to be designed so that a statement of instruction or prompting message is printed on the terminal at each breakpoint in order to guide the user. The following shows a typical example: (user's typing in *italics*)



Thus, the combination of the prompting statement, the prompting symbol and the user's response constitutes a man-machine interaction, and is referred to as a man-machine dialogue. Programs using extensive dialogues to guide the users are called conversational programs.

(4) It is not possible to transfer directly from one CUSP-level language processor to another without first returning to the monitor. This transfer can be made conveniently by providing a special control key on the remote terminal keyboard. See Section 1.6 for the function of various keys on the keyboard.

1.3 Computer Service

The computing facilities in an academic institution are generally provided to serve a combination of instruction, research and administration functions. When the facilties are shared by different users for different functions, it is necessary to establish rules and regulations so that the resources may be most efficiently and equitably utilized. While it is outside the scope of this book to enumerate these rules and regulations, it is important for every user to be familiar with them. These include such matters as application procedures, allocation of computing time and resources, restrictions placed on the computing services, fiscal arrangements, ethical and legal stipulations regarding security, propriety and relevance of work using the computing resources, and policy on computer abuses.

Application procedures are generally defined by the Computer Center to determine the eligibility and extent of computer usage of an applicant. The application requires certain pertinent facts and the usual authorizing signatures. Readers are referred to their respective Computer Centers for current procedural details.

When an application is accepted, the applicant is assigned a pair of identifying numbers:

[m , n]

where m = a 6-digit (octal) project number, and
n = a 6-digit (octal) programmer (user) number.

The combination of these two numbers, referred to as the project-programmer numbers, is often abbreviated either as PPN or as P,PN. Note that a PPN is usually enclosed in a pair of square brackets.

REMOTE TERMINALS

1.4 Communication with the Computer

A remote terminal is used as an input or output device at the control of the user. Generally, it is a typewriter-like device with a keyboard, a typing or displaying element, and the interface between the user and the system. The performance of the remote terminals depends to a large extent upon the communication linkage between the terminals and the computer. Hence, some of the basic concepts and terminology will be described here to aid the understanding of a time-sharing terminal.

(1) Transmission line

Depending on the modes of information transmission, the transmission lines, also called channels, are classified as simplex, half-duplex, and full-duplex. A simplex channel can transmit information in one direction only. A half-duplex channel can transmit information in either direction, but only in one direction at a time. A full-duplex channel can transmit information in both directions at the same time.

Depending on the physical connections, transmission lines may be classified as dedicated, shared, hard-wired or dial-up lines. A dedicated line or channel is one assigned for the exclusive use of the terminal. A shared line is one assigned to the use of several terminals. A hard-wired line connects physically from the terminal to the system. A dial-up line is a shared line using the commercial dial telephone network for connection.

(2) Information code

Information to be transferred externally between a terminal and a computer on the transmission line is represented by character sets consisting of alphabetic characters, both upper and lower cases, numeric characters, punctuation marks and special characters. In addition, signals representing control action of transmission and processing are coded into "control characters". These information characters and control characters may be coded into a series of binary digits (called bits) so that information may be transmitted and processed by the computer and the terminals. Several systems of codes are in use. The code format used in most U.S.-made non-IBM machines, including the systems at Pitt, is the ASCII* code, which encodes 128 characters into 7 binary digits. Table 1.1 shows the ASCII code assignment of characters, where the code assignments are given in octal numbers. For example, the upper case letter "A" is coded as octal 101, or actually as 7-bit binary representation of 1000001.

Note that the character set shown in Table 1.1 is the ASCII information-character set, which is a subset of the complete ASCII code of 128 characters. The 32 characters not shown in Table 1.1 are all control characters. With ASCII code, the words PITT and Pitt are then transmitted respectively as:

10100000 1001001 1010100 1010100 (PITT)
10100000 1101001 1110100 1110100 (Pitt)

---

*Acronym for American Standard Code for Information Interchange, usually pronounced as "AS-KEY".

Character	ASCII 7-Bit	Character	ASCII 7-Bit	Character	ASCII 7-Bit
Space	040	@	100	'	140
!	041	A	101	a	141
"	042	B	102	b	142
#	043	C	103	c	143
$	044	D	104	d	144
%	045	E	105	e	145
&	046	F	106	f	146
'	047	G	107	g	147
(	050	H	110	h	150
)	051	I	111	i	151
*	052	J	112	j	152
+	053	K	113	k	153
,	054	L	114	l	154
-	055	M	115	m	155
.	056	N	116	n	156
/	057	O	117	o	157
0	060	P	120	p	160
1	061	Q	121	q	161
2	062	R	122	r	162
3	063	S	123	s	163
4	064	T	124	t	164
5	065	U	125	u	165
6	066	V	126	v	166
7	067	W	127	w	167
8	070	X	130	x	170
9	071	Y	131	y	171
:	072	Z	132	z	172
;	073	[	133	{	173
§	074	\	134	|	174
=	075	]	135	}	175
↑	076	↑	136	~	176
?	077	←	137	Delete	177

The code assignments of octal numbers from 000 to 037 are for control characters, and are normally of no concern to an average user. However, certain control characters pertain to printer control, and it will be useful to know their code assignments. These are:

Line Feed	012	Form Feed	014
Vertical Tab	013	Carriage Return	015
Horizontal Tab	021		

Table 1.1 ASCII Character Set
All numbers in octal codes.

In an actual transmission, each ASCII-coded character is packed together with additional bits that perform functions of synchronization (START and STOP of each character), error-checking (parity bit), and filler or dummy bit (to allow slower mechanical components to catch up with electrical and electronic components). The result is either an 11-bit group (for low-speed transmission) or a 10-bit group (for higher speed transmission) for each character transmitted.

Not all computers made in U.S. use the ASCII code. The IBM computers, such as System/360 and System/370 machines use a code system called EBCDIC (Extended Binary Coded Decimal Interchange Code) to adapt to its byte-structure (1 byte=8 bits). Hence, output media, such as magnetic tapes, are not compatible between ASCII-code machines and EBCDIC-coded machines without first going through a code conversion process. Because of wide-spread use of both code systems, such a code conversion routine is a part of standard service routines available at the Computer Center. For the same reason, a remote terminal wired to accept the EBCDIC code cannot be used in the DEC-10 system unless it is re-wired or it has a switchable option of code selection.

While the ASCII code has been adopted as the American standard for peripheral communication, it has shortcomings in certain particular applications. For example, the internal representation of a FORTRAN variable would be very awkward in a machine such as the DEC-10 with a 36-bit memory word format. Since the standard FORTRAN defines a variable name to contain one to six characters, an ASCII-coded six-character FORTRAN variable name will require 42 bits or 2 memory words for its storage, a rather inefficient usage. As seen in the Table 1.1, if we forego the difference between the upper and the lower cases of alphabetic characters, we can omit the right-hand column in that table. This would reduce the character set to only 64 characters. Since each of the 64-character set may be uniquely defined by a coding scheme of six binary digits, this results in a Sixbit Code. With each character code only six bits long, a six-character FORTRAN variable name can now fit snugly into a single 36-bit word. In this coding system, any lower case alphabetic character, when encountered, will be automatically coded as its upper case equivalent. The code assignment of each character in the Sixbit Code will not be tabulated here, but will be given later in Chapter 3 (FORTRAN-10) where its reference will be more relevant. The derivation of the Sixbit Code of a character from the 7-bit ASCII code may be obtained simply by dropping the second bit (counting from the left). For example, the letter "A" is coded as 1000001 in ASCII code, and is 100001 in Sixbit. Alternately, the Sixbit Code can be "computed" from the ASCII code by either of the following algorithm:

```
(SIXBIT)  = (ASCII) - 040      in octal arithmetic
(SIXBIT)  = (ASCII) + 040      in octal arithmetic
```

and then retain the two least significant octal digits.

Thus, the letter "A" is coded as octal 101 in ASCII, and as octal 41 in Sixbit.

(3) Speed of transmission

The speed of transmission of the signal is measured by the rate of transmission in signals per second, expressed in bauds*. In binary transmission, each signal contains one bit of information, and consequently the speed of signal transmission is numerically the same as the speed of information transmission. Thus a 300-baud line will transmit information at a rate of 300 bits/second. However, in polyphase modulation, each of the four predetermined phase-shifts represents two bits of information, and a 300-baud line will transmit information at a rate of 600 bits/second. Capability of commercial transmission services, such as telephone or telegraph lines, ranges from 100 to several hundred thousand bauds. The maximum capability of a "voice grade" dial-up telephone line is about 2000 bauds.

In a time-sharing system, information transfer may be initiated or terminated at the terminal. The ASCII coded 7-bit signals arriving at or departing from a terminal are packed with additional bits to perform functions of synchronization and parity error checking. The result is an 11-bit group for each ASCII character for 110-baud transmission, or a 10-bit group for 150 or 300 baud rate transmission. These transmission speeds are used to match the terminal output speed of 10, 15 or 30 characters/second respectively.

Since the remote terminals generate and receive information at relatively low speed, the capability of the transmission line is hardly taxed. Consequently, various line-sharing techniques are available, one of which involves the use of a concentrator. A concentrator is usually a minicomputer which collects information from several terminals in the area at a low speed, and then packs them and re-transmits. In the reversed direction, a concentrator receives information and distributes them to different terminals.

1.5 Description of a Remote Terminal, the DEC LA36 DECwriter

For several decades, the most commonly used communication terminals have been the Automatic Send-Receive Teletypewriter Set (ASR), model 33, 35 or 38. These are called ASR33, ASR35, and ASR38, and in most cases, simply Teletype ®. In fact, the standard abbreviation for terminal-like device in a computing system has been uniformly taken as TTY.

Rapid recent advances in technology have produced new generations of remote terminals. Relay circuits were replaced by transistorized circuits, which in turn are being replaced by microprocessors or microprogrammed controllers with semiconductor memory. Mechanical components are improved so that they are lighter and move faster. Clumsy typing heads with embossed characters are replaced with matrix wire impact printing, thermal or electrostatic non-impact printing. While the technological advances have made new generations of terminals lighter, faster, less expensive and more reliable, the basic operating principles procedures remain essentially unchanged, thanks to the tremendous steadying effect of Teletypes as the industry workhorse over the last three to four decades. It is therefore possible in the present discussion of remote terminals to deal specifically with one particular terminal and still retain generality of our discussion. It also means that although this presentation pertains to only one model of terminal, extension of the discussion

---

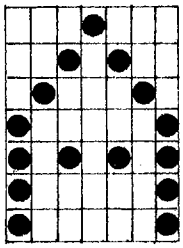
*Named after the French inventor of the telegraph code, Jean-Maurice-Emile Baudot, 1845-1903

® Registered trade mark, TELETYPE Corporation, Skokie, Illinois.

to another make or model would be no problem. This is why we will now concentrate on one particular terminal, the DEC LA36 DECwriter, for the subsequent discussion.

A remote terminal contains generally four major parts. They are the keyboard unit, the printer unit, the print control unit, and the call control unit. A simplified block diagram, with the important components within each part, is shown in Figure 1.4. The arrows in the diagram show the direction of signal flow and/or control when the terminal is connected to a computer. Its operation can be described briefly in this manner:

When the user strikes a key on the keyboard, say the upper case "A", the keyboard electronics encodes it into the ASCII code of signal 1000001. These electric signals are sent to the transmitter unit, in which additional start-bit, stop-bit, parity-bit and filler-bit (if needed) are added. The communication electronics in the transmitter transform these signals into modulated audio tones, which are transmitted serially through the interface to the computer over a transmission line. At the computer end, a buffer (or temporary) memory accepts the character after checking over any transmission error, repacks the character with start-, stop-, parity- and filler-bits, and re-transmits back to the terminal. When it reaches the terminal, the receiver demodulates the signals by removing the audio carrier, checks for any transmission error, and deposits the 7-bit ASCII code of "A" in the buffer memory.



When the printer unit is ready to accept a character, controlled by the printer control unit, the ASCII code inputs are sent to the character generator ROM (Read Only Memory, a semiconductor memory chip) which produces seven one-or-zero signals simultaneously 7 consecutive times. Each 7-signal group, after amplification, selectively actuates by solenoids vertically arranged wires to strike an inked ribbon, leaving a vertical column of selectively placed dots in that column.

This is repeated seven times, each time with the print head moving slightly to the right, and each time producing a different vertical pattern. The result is shown here. This is called a 7x7 dot matrix print.

It is interesting to note that the signals generated at the keyboard take a circuitous route before finally printed on the terminal printer. In fact, what is printed is actually what the computer thought the user has typed. This is a clever way of involving the user as a part of error-checking system, and is a standard feature in time-sharing system called echo print.

The individual parts of the DECwriter will now be described next:

(1) Print unit

The print unit is the receiving component of the terminal. It consists of seven vertically arranged print wires actuated by seven solenoids, which in turn are controlled by the character generator ROM as explained before. Other useful information about the print unit are as follows:

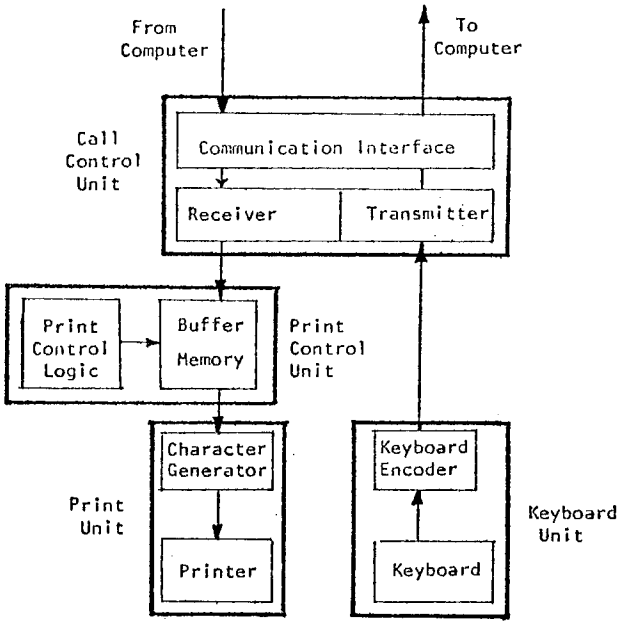


Figure 1.4 Block Diagram of a Typical Remote Terminal

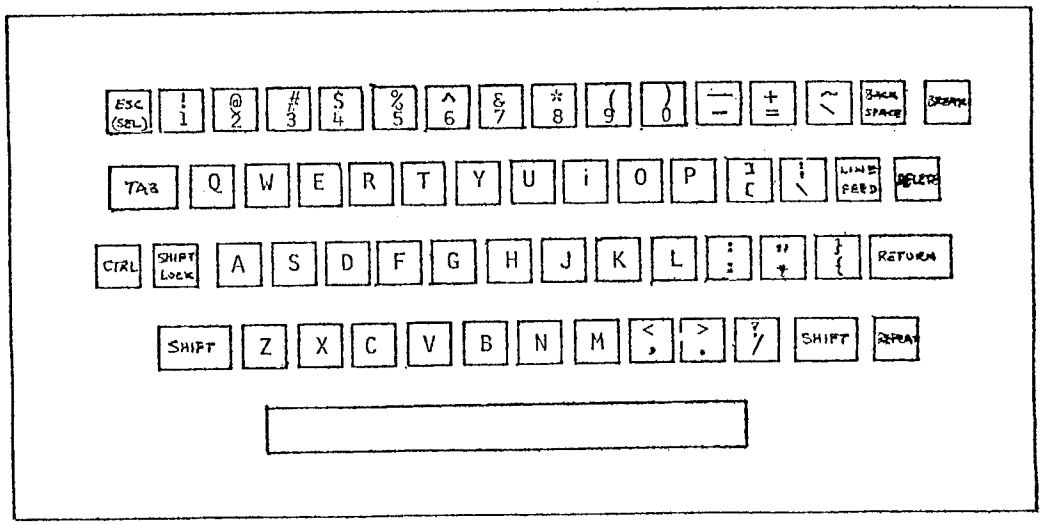


Figure 1.5 Standard ANSI Keyboard Layout

Paper size:	3" minimum, 14" maximum width
Print field:	132 characters maximum
Print spacing:	10 character/inch horizontal, 6 lines/inch vertical spacings
Print characters:	96 upper/lower case ASCII 7x7 dot matrix (0.07x0.10 inch)
Print speeds:	switch selectable at 10, 15 or 30 characters/second with 60 char/sec catch-up mode*

(2) Print control unit

The print control unit contains a buffer memory that accepts ASCII character codes received and a control logic unit which is a microprogrammed controller. Under the control of the microprogram, characters in the buffer are presented to the character generator on a first-in/first-out basis. The microprogram activates the carriage servo system and the print head system to control the mechanical movements. It also detects signals and actuates mechanisms such as line feed to advance the paper, ringing the bell for an error, etc.

(3) Call control unit

The call control unit consists of an asynchronous receiver-transmitter and a communication interface. It initiates, accepts, controls and completes the incoming and outgoing transmission of information.

(4) Keyboard

The keyboard is the information-sending component of the terminal. The mechanical linkages and electrical contacts translate the key action into a group of electrical signals. The arrangement of keys on the keyboard resembles that of a conventional typewriter with additional special features. These are discussed in a later section in this chapter.

Those terminals called ASR's (Automatic Send-Receive Sets) contain, in addition to the four units mentioned above, one of the following auxiliary input/output units: paper tape reader/punch, or tape cassette player/recorder, or floppy disk with read/write electronics. These serve as storage media for the terminal.

There are several switches placed adjacent to the keyboard which allow a user to power-up, select the transmission rate, and choose on-line or local operation. In local operation, a terminal will function as a typewriter, allowing a user to add information to the printout. It also permits maintenance work and testing of a terminal without disturbing the computer. For ASR-type terminal with either paper tape, digital cassette or floppy disk, the LOCAL position permits the ASR to be used as an off-line input/output device for such task as preparing, editing, reproducing and printing paper tapes, cassette tapes, or floppy disks.

---

*While the time-consuming action of carriage return, tab or line feed is taking place, characters received are stored in the buffer. When mechanical action is finished, characters in the buffer will empty into the printer unit at 60 char/sec catch-up speed.

1.6 The Keyboard

The keyboard arrangement of the DECwriter follows the ANSI (American National Standard Institute) standard. It has a format very similar to the conventional typewriter. Figure 1.5 shows a keyboard of the DECwriter.

(1) Alphabetic characters

Key positions of alphabetic characters are identical to those on a conventional typewriter. Both upper and lower cases are available. However, transmission of alphabetic characters are generally done in upper cases, unless specifically commanded to transmit as lower cases. Thus, pressing an alphabetic key without the shift key will transmit and print an upper case letter.

(2) Numeric and special characters

The character set on the DECwriter keyboard consists of the following:

Alphabetic:	A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
Numeric:	0 1 2 3 4 5 6 7 8 9
Special:	+ - * / ( ) = " $ % ' @ \ . , ; : ? [ ] < > { } _ |

(The underscore symbol _ is replaced with the left arrow symbol ← on certain keyboards.)

(3) Control keys

Certain special keys perform control functions:

a. LINE FEED — This key will cause the terminal paper to advance one line. When the terminal is operated on LOCAL, the carriage return does not automatically advance the paper, and the LINE FEED key must be pressed to do it.

b. RETURN — This key will return the print head and carriage. When the terminal is on line, returning the carriage return signifies the end of a unit of information, for example, an instruction to the computer. The computer will automatically respond with a line-feed control signal to advance the paper.

c. DELETE — This key permits the correction of typing errors on a line if the carriage has not yet been returned. This key is marked as RUBOUT on some older keyboards. When the DELETE key is pressed successively for n number of times, the last n characters typed (including spaces) will be deleted. As a signal to the user which characters are being deleted, the terminal will print out the deleted character each time the DELETE key is pressed. Also, before the first deleted character and after the last deleted character, a back slash "\" is printed. Thus the pair of back slashes serves as delimiters bracketing the string of deleted characters.

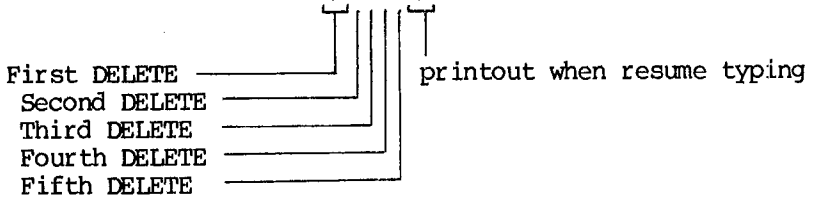
For example, if the following has been typed on the terminal:

```
FOUR SCORE AND SEVIN YE
                       ↑
                       |
```

Carriage position when mis-spelling in SEVIN is discoverd.

In order to delete the five character "IN YE", five successive DELETEs are required. Notice that a space or blank is also considered a character. To correct the typing, the user will DELETE five times and then retype the corrections. On the printout at the terminal, it will appear like this:

```
FOUR SCORE AND SEVIN YE\EY NI\EN YEARS AGO, OUR FATHERS ...
```



As shown in the example, a pair of back slashes brackets the deleted characters printed in the order of deletion (from right to left).

d. REPEAT This key, when operated together with another character key, will cause a repetition of that character to be printed (for LOCAL operation), or a repetion of that character to be transmitted and echo-printed (for on-line operation). For example, when the REPEAT and K keys are pressed down together, a string of K's will be sent and printed as long as both keys are held down.

e. SHIFT, SHIFT LOCK These keys have identical functions as those on a conventional typewriter, and will cause the upper case character marked on the key to be printed or transmitted.

f. ESC This key, appearing on older keyboards as an ALMODE key, directs the computer to treat the next received character as a command. The precise meaning of the ESC-character combination is defined by the software system employing this function.

g. BACK SPACE Depending on the software processor used at the time, the back space key either makes the last character sent to the computer available for deletion or correction, or makes it possible to overprint with a different character such as underscoring a certain text string.

h. TAB This key will direct the computer to advance the print head to the next tab stop.

i. BREAK Used for half-duplex transmission mode to interrupt reception of data from the computer. Ignored in ordinary full-duplex mode.

(4) Control characters

The key CTRL, when used together with an alphabetic character key, generates a code combination for control purposes. Such a combination of CTRL and alphabetic keys does not have any printing function, and therefore the computer will return an echo signal printed out on the user's terminal to inform him of the nature of the control function. The echo print has a format of "^" (a circumflex) or "↑" (an up arrow) followed by the character used, such as ^C or ↑C. These control characters will appear in this book frequently, and they will be referred to in several ways. For example, the control character C will be referred to as:

CONTROL-C
CTRL-C
or, ^C (or, ↑C)

Although there are 26 control characters, a beginning user need only be familiar with a few of them, and they are ^C, ^O, ^U, ^I, ^L, and ^R. Several other control characters, such as ^S and ^Q, will be explained at appropriate places where they are used.

a. CTRL-C (^C) The ^C key interrupts the program and returns the control to the system monitor. If a program execution is in progress, apply ^C twice or more to interrupt it. The first ^C stops the execution of the program, and the second one (and the succeeding ones) returns control to the system monitor. When the system monitor obtains the control, a prompt symbol (a period) is printed on the terminal, and the system awaits for a monitor command.

b. CTRL-O (^O) The ^O key suppresses the terminal output without interrupting the execution of a program. For example, when debugging a program, if you only want to see whether a program execution reaches the end, you can suppress all or specific parts of the put in order to avoid time-consuming printing of the results. Thus any time when output begins to appear, applying CTRL-O will suppress the remaining portion of that output.

c. CTRL-U (^U) The ^U key, applied at the end of one line of typing, will instruct the computer to ignore the entire line and therefore to perform the function of deleting that line. The system will respond with a carriage return and a linefeed, but no prompt symbol.

d. CTRL-I (^I) Terminal will tab to a pre-set column.

e. CTRL-R (^R) Terminal will re-type the current line.

f. CTRL-L (^L) This control character tells the computer to advance the paper to a new page. On a DECwriter terminal, it will advance the paper only 8 lines.

The keys for these functions are summarized in Table 1.2.

Special Key	Echo Print If any	Function
LINE FEED		Move paper up one line.
RETURN		Return the carriage.
DELETE (or RUBOUT)	/X	Delete character immediately before.
REPEAT		Repeat a character or a function.
CTRL-C	^C	Return to monitor mode.
CTRL-O	^O	Suppress current terminal output.
CTRL-U	^U	Ignore the current line input.
CTRL-I		Tab to a preset column.
CTRL-R		Retype the current line.
CTRL-L		Advance paper on terminal 8 lines.

Table 1.2 Function of Selected Special Keys

1.7 Other Types of Remote Terminals

The DECwriter terminal as described in the previous section is a keyboard printer terminal. The majority of remote terminals used in a time-sharing system are of this type. A variation of this type is the portable terminal, which incorporates in a single carrying case an acoustic coupler (for connecting the terminal to the computer by a telephone set), a keyboard, a printer and associated electronics. One ingenious product includes an acoustic coupler, a keyboard, and associated electronics, but no printer. It makes use of a conventional television set, and when combined, it becomes a time-sharing terminal.

As a result of rapid advances in MOS/LSI (metal oxide semiconductor and large-scale integrated circuits) technology, the size, weight and cost of electronic components and systems have been greatly reduced. These advances have caused rapid developement of other types of terminals, and they are briefly discussued next:

(1) Cathode ray tube (CRT) terminal

The convenience of a keyboard operating terminal is greatly enhanced if we use a cathode ray tube (CRT) terminal for the purpose of communication with the computer, preparation of programs and debugging. This is particularly useful if the user has an alternate means of producing hard copy as records.

A typical CRT terminal, also called a scope terminal, displays a subset of ASCII characters (such as upper cases of alphabet plus symbols). The display unit is similar to that used in an oscilloscope or television set. Other than the display unit and its control memory, it has the same organization as a keyboard printer terminal. Figure 1.6 shows a block diagram of a typical CRT terminal. If we compare this with Figure 1.4, we can see the obvious resemblance.

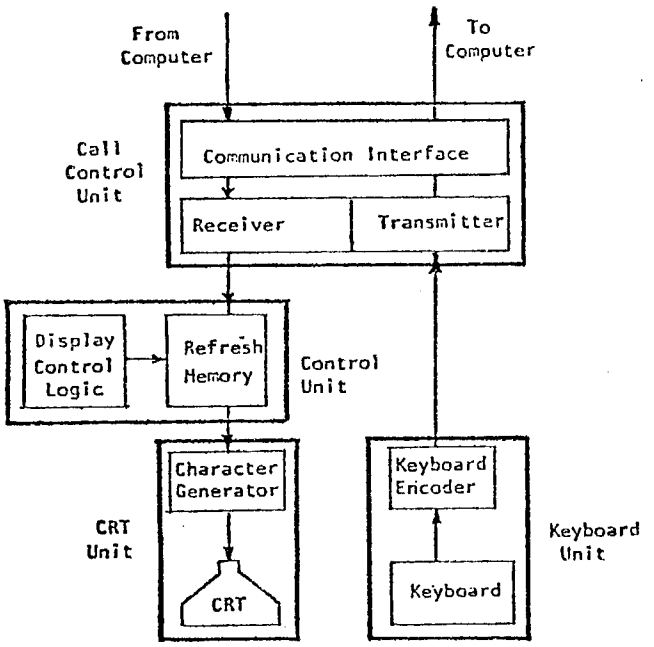


Figure 1.6 Block diagram of A CRT Terminal

The CRT produces an image by directing an electron beam against a phosphor-coated screen which emits light when struck by electrons. The control on the beam intensity can turn the beam completely off, thus allowing no electrons to strike the screen, a process called blanking. Positioning the beam in a CRT terminal is usually done by the raster scan method. The beam is first positioned at the upper left corner of the screen; it then moves across the tube face, producing a straight line. The beam is blanked while returning to the left at a level one line lower. The blanking is turned off, and a second line is traced. This is the same method used in a commercial television set that scans 525 lines/frame and at a rate of 1/30 second per frame. Forming characters on screen is very similar to the dot matrix print of a keyboard printer terminal. The scan scheme will position the rectangular area within which the character is displayed. The character, through a character generator ROM (read-only-memory), formulates a 5x7 dot matrix, with dots emitting light when the electron beam strikes the tube phosphor. Typically, the light-emitting period is very brief, ranging from microsecond to millisecond range. Therefore, a CRT using the scan method requires a refresh memory that stores the display and re-display at a refresh rate large enough to provide a constant intensity image and to eliminate flicker in the image.

Most CRT terminals are also teletype-compatible, and they are often interchangeable with keyboard printer type terminals. With no carriage, a CRT terminal is provided with a cursor, which may blink on and off to indicate the current position of the beam. The associated cursor control enables the user to move the cursor up, down, left or right, or to erase the screen. Unlike the teletype, data rolling off the top of a CRT screen are lost to the user. The operations of a CRT terminal and a keyboard printer terminal are very similar. A person familiar with the operation of a keyboard printer terminal should have no problem with CRT terminal operation.

(2) Graphics terminal

This is a terminal which maintains the capability of displaying not only characters, but also arbitrary figures. All of the man-machine interaction

previously described are retained, and the interaction is expanded to include the clarity of graphics.

When a graphics terminal displays characters, it emulates a CRT terminal, and outwardly it operates just like a CRT terminal. When a graphics terminal operates in the graphics mode, it provides both control of beam position and blanking. In the position control, the beam is deflected from a current position to another. If the blanking is on, only two end points are shown on the screen. If the blanking is off and if the terminal is equipped with a "linear interpolation vector generator," the electron beam will trace a straight line. Repeated programmed positionings of the beam, with blanking on or off as required, will produce a line drawing.

Graphics terminals normally utilize a cathode ray tube display, but some low cost units use a storage tube to retain the data which does not require a refresh memory. The disadvantage of the storage graphics display is that dynamic display and removing graphic information are not possible: any subtractive change of displayed data requires first an erasure of the entire image, and then a reconstruction of a new image, an event that will take at least half a second. Thus a storage tube may display at a maximum rate of about 2 frames per second, not a satisfactory speed to depict motion. On the other hand, graphics terminal using refresh memory imposes a heavy burden of memory and software support for its image generation and constant refresh. The heavy requirements of memory and software usually call for a minicomputer to provide the support.

(3) Intelligent terminal

For years, manufacturers have been offering terminal systems with fixed functional capability. For example, a terminal designed to be compatible with the IBM systems, which use a different character coding system (EBCDIC Code), is not compatible to a system using the ASCII code unless extensive re-wiring is done.

The rapid recent advances in MOS/LSI technology have now made it possible to incorporate microprocessors and memories, which greatly expand the flexibility and capability of a terminal. Instead of a simple function of transmitting and receiving data or programs, a terminal may now have additional processing power. Acquiring such additional processing power within the terminal is referred to as "making the terminal more intelligent", and therefore the name "intelligent terminal." Quite predictably, a terminal without additional built-in intelligence is called a "dumb terminal."

Intelligence in a terminal may take on many forms. It ranges from the simple ability of changing operating characteristics of the terminal to the power of a full-scale microcomputer. Intelligent terminals therefore are able to emulate many different communication line procedures and codes, so that a terminal may be coded to adjust to an existing line protocol and procedure. For many other various functions, the terminal may be tailored to suit the need of the particular user or industry segment by providing specific software for the intelligent terminal. For example, an editing program may be installed in the intelligent terminal so that the terminal becomes a word-processing machine. Word-processing tasks may then be carried out without loading down the central computer. Another example is an intelligent graphics terminal where the graphics are processed by a built-in graphic processor in the terminal. Again, in this way, the central computer will not be loaded down with detailed chores.

The main disadvantage of an intelligent terminal, at the time when the third edition of this book is being prepared, is its cost, although the gap is rapidly narrowing. In applications where only simple functions are required,

dumb terminals are more cost effective. In time, the difference in cost will become insignificant, and the intelligence of the intelligent terminal will be greatly expanded. The experiences of the hand calculator industry can very well be repeated in the remote terminal industry within the next decade. At the present time, the applications have been limited to such areas as point-of-sale credit authorization, bank teller systems, stock brokerages, airline reservation systems, hospital admissions, etc., where distributed data processing is highly desirable.

1.8 Sign-On at the Remote Terminal

Once a user has a valid pair of ID numbers (the PPN) and has a valid password, he may now sign on at any remote terminal by following the procedure outlined below:

Hard-Wired Units	Dial-Up Units
(1) Turn on switches. Press C if there is no prompt symbol ".". After the prompt "." appears, type "I" (for INITIATE) and the following lines will be typed out on the terminal:	(1) Turn on switches and dial the computer number.* If the line is busy, there is a usual busy signal. When the call gets through, a high-pitch tone can be heard. Place the phone set on the seat of the acoustic coupler. Wait until the READY or CARRIER light comes on, type C, and the following two lines will be typed out on the terminal:

```
PITT DEC-1099/A 63A.41B 15:36:41 TTY43 system 1237/1240
PLEASE LOGIN OR ATTACH
```

where "1099/A" indicates System A, "63A.41B" the monitor version, "15:36:41" the time of the day in 24-hour clock, "TTY43" the line number assigned. If "1099/B" appears instead of "1099/A", it means the user is in touch with System B. If the user finds himself in a wrong system, he requests a change by typing:

TTY SYSTEM B	or	TTY SYSTEM A

after the prompt symbol.

(2) Type the monitor command after the prompt symbol:

LOGIN m,n ↵

or

LOGIN m/n ↵

where m = project number, n = programmer number,
↵ = carriage return.

The difference between "m,n" and "m/n" in the two monitor commands is that the latter form will suppress the message of the day from the Computer Center when the sign-on procedure is completed. It is possible that you have seen the message several times already, and may not care to read it another time.

---

*For University of Pittsburgh users, dial (412) 621-5954.

The carriage return is a standard control signal to indicate to the computer the termination of a line, a command or a message. To avoid cluttering the text and to relieve the typing problem, the carriage return symbol " " will be used only in Chapter 1. For the remainder of the book, the readers should assume that there is always a carriage return at the end of every line.

(3) Enter the password when requested. The password will be entered in a non-print mode, and the typed password will not appear on the terminal. This is to maintain the security of the password.

If the entered password is an incorrect or invalid one, the system will respond with an error message and a request for the PPN. After supplying the PPN again, another password request will be made by the computer. The user has five chances to sign on correctly. After that number of unsuccessful trials, the job is killed, and the user must restart the entire procedure to sign on.

If the password is found to be valid, the system will respond with information on the status of the project, the last sign-on time and date, the time of day, and the "message of the day" from the Computer Center. The last item may be suppressed if the user uses the LOGIN command with the m/n specification.

After all preliminary reports are finished, a prompt symbol "." is printed on a new line, and the computer pauses and waits for input. The user is now connected to the computer at the monitor level, and the sign-on procedure is completed.

The following two cases are examples of sign-on. Explanatory remarks are also given along with the remote terminal printout. As used throughout this book, those lines entered by the users will be in *italics*:

Printout on Terminal	Remarks
.*INITIATE* ⟩	INITIATE command
PITT DEC-1099/A 63A.41B 16:19:17 TTY43 system 1237/1240	Computer's response
.*TTY SYSTEM B* ⟩	Request System B
PITT DEC-1099/B 63A.41B 16:19:50 TTY43 system 1237/1240	
.*LOGIN 115103,320571* ⟩	Sign-On command
JOB 35 PITT DEC-1099/B 63A.431B TTY43 Wed 7-May-80 1619	
Password: (*Your password*) ⟩	Supply password
Last login: 7-May-80 1617	
Usage ratio: 22.13 Units used: 33.5	Password valid
SYS B DOWN 0000-0800 MON MAY 12 FOR REGULAR HARDWARE MAINTENANCE	
SYS B DOWN 0000-0300 TUE MAY 13 FOR REGULAR SOFTWARE MAINTENANCE	
DUE TO HARDWARE PROBLEMS THE ARRAY PROCESSOR WILL BE TEMPORARILY OFF LINE UNTIL FURTHER NOTICE	Message of the day
.	System ready!
.*LOGIN 115103/320571* ⟩	Sign-On command
JOB 23 PITT DEC-1099/B 63A.41B TTY43 Wed 7-May-80 1815	
Password: (*Your password*) ⟩	Supply valid password
Last login: 7-May-80 1619	
Usage ratio: 22.13 Units used: 33.5	
.	System ready!

1.9 Password

To sign on the DEC-10 system, the required identifications are a valid PPN and the associated password. Security of PPNs is impossible because they are publicly displayed in many places - in LOGIN printout, in the file directory, in printout identification, etc. Thus the only real safeguard and security of a computer account is the password.

The need for protection against unauthorized use of your account by another person goes beyond accounting reasons. There have been numerous incidents of computer vandalism in the past. The most frequent vandalism was change or erasure of programs or data without the owner's knowledge.

The only protection against such unauthorized use is to install a password, to keep its security, and to change it frequently. As a matter of prudence and necessity, the user should change his password regularly as a standard practice and whenever he suspects the password is no longer secure.

Changing a password at a terminal can only be done at the LOGIN time by using either of the following LOGIN format:

LOGIN m,n/PASSWORD
LOGIN m/n/PASSWORD

or,

where "m" and "n" are the PPN. The following shows a sign-on session with a password change. Since the process is interactive, the explanation should be self-evident:

```
.LOGIN 115103/320571/password ↵
JOB 16 PITT DEC-1099/B 63A.41B TTY43 Wed 9-May-80 2003
Password: (Enter old password) ↵
New Password: (Enter new password) ↵

Retype for verification

New Password: (Enter new password again) ↵
Last password update: 24-Apr-80 1255
Last login: 22-Apr-80 1642
Usage ratio: 0.84 Units used: 33.1
```

1.10 Disk Storage Quota

One of the special features of a time-sharing computing system, as compared with a computer for batch processing applications only, is its very large capacity for on-line mass storage, such as the disk storage. It is a common practice to assign and allocate a part of that mass storage for users to store their programs, data or other files. These storage spaces are measured in "disk blocks", or simply "blocks". In DEC-10 system, each block contains 128 data words in DEC-10 format. Therefore, each block can hold a maximum of 640 characters, an equivalent of 8 fully punched cards.

Each authorized user is assigned a quota of disk space in blocks called logout quota, in which he may store his files permanently. These files will not be removed from the storage unless any one of the following situation occurs: (1) when a file is deleted by the user himself, (2) when a file is inactive and not accessed for more than a prescribed period (for example, a month), or (3) when the project has been cancelled or terminated.

When a user is LOGINed and on-line, the actual disk space assigned to him is five times the logout quota. The extra storage is assigned for storing temporary data, non-permanent program or data files needed for the execution of the user's work while he is on line. This on-line quota of disk space is called the login quota. The actual number of blocks assigned as the login quota depends on the logout quota and the available system capacity at the time.

After a user has LOGINed, he may enjoy the larger login quota for his on-line work. When he is ready to sign-off, he must make sure that his disk usage is under the logout quota, otherwise all efforts of signing off would fail, or else the computer will delete the stored files according to a predetermined order of priority until the logout quota requirement is met. In the latter case, the computer may very well delete some important files.

The monitor commands for managing the files are discussed in Chapter 8. However, several commands that are necessary in managing the quota will be briefly discussed here. For more details of these commands, the readers are referred to Chapter 8.

The monitor command R QUOLST is used to inquire about the current status of the disk quota (login, logout, and system status). An example follows. Again, lines in *italics* are typed by the user:

```
.R QUOLST ↵                                    Explanation

User:   115103,320571 ──────────────────  User's PPN
Str      used   left:(in)  (out)   (sys)
USRB:     180      120     -120    182616   Disk Status:
                                          └── System status
Timesharing Core Class: 0  └──────────────  Logout quota status
Batch Core Class: 0        └──────────────  Login quota status
                           └──────────────  Disk block used
   │
   └──User's core classes
   └──────Storage device specification
```

In this example, the user has a logout quota of 60 blocks and a login quota of 300 blocks. At the time of this inquiry, he has used up 180 blocks. Therefore, the above printout indicates that he is still 120 blocks under the login quota, but he is 120 blocks above the logout quota. Should he wish to sign off at this time, he must first delete his files for at least a total of 120 blocks. So, at this point it is important to him to know how to find out what he has in the storage and how he can selectively delete them. Two other monitor commands useful for quota management are:

and

```
DIRECT ↵

DIRECT name.ext ↵
```

When the command DIRECT (for "directory") is given, the terminal will print out a list of user's files in the disk storage, along with their names,

extensions, file sizes in blocks and other pertinent information. The total amount of storage occupied is printed out at the end of the list. A sample result of this command is shown below:

```
.DIRECT

TEST    DAT     60  <057>   18-MAY-79    USRB: [115103,320571]
SAMPLE  FOR     48  <157>   19-MAY-79
SAMPLE  REL     36  <057>   22-MAY-79
TEST    BAK     36  <057>   24-MAY-79
  TOTAL OF 180 BLOCKS IN 4 FILES ON USRB: [115103,320571]
```

The command DIRECT thus gives the user an inventory of files in the disk storage at that time. If he is then ready to sign off from the computer, and if he is over the logout quota, this inventory information will enable him to decide which file he should erase in order to get below the logout quota limit. The monitor command of DELETE is used to erase a file in the storage. If in the above example, the files TEST.DAT, SAMPLE.REL and TEST.BAK are to be erased, then the command issued is :

```
.DELETE TEST.DAT, SAMPLE.REL, TEST.BAK
```

After erasure is completed, the terminal will report the names of the erased files and the size of total restored storage. The details of file names, extensions and other information about file name structure are given in Section 1.12.

1.11 Sign-Off Procedure

To leave the system, the user must terminate his job by supplying a monitor command KJOB ("to kill the job"). The system will respond by requesting a code letter for confirmation and file disposition. Thus, the command format for signing-off is:

```
.KJOB ↵
CONFIRM: (code letter)↵
```

A shortened form of this command is:

```
.K/(code letter) ↵
```

The most commonly used code letters in the KJOB command are:

F = fast signoff; save all files

D = fast signoff; delete all files. Computer will respond with A confirming question: "DELETE ALL FILES?" Answer YES and return the carriage.

P = preserve all files except temporary files.

H = HELP! Computer will respond will detailed instructions.

I = list file names, one at a time, and apply code letter decision individually. The code letters for individual decision are:

P = preserve the file

S = save the file

K = delete the file

Q = learn if over logout quota on this file

E = skip to next file and save this file if below logout quota for this file. If not below logout quota, a message is typed and the same file name is repeated.

H = HELP. Computer will respond with the above information on code letters.

While files are disposed per user's code letter instruction, the computer will make a check on logout quota, gather all usage and accounting information, terminate the user's job and print out a summary of the job. For example:

```
.K/F ↵
JOB 16 [115103,320571] off TTY43  at 2032 9-May-80  Connect=29 Min
Disk R+W=83+76  Tape IO=0  Saved all files (450 blocks)
CPU 0:04  Core HWM=11P  Units=0.1263 ($9.48)
```

The printout indicates that this user, with PPN of 115103,320571, was assigned line 43 and job 16, signed off at 2032 on May 9, 1980. His terminal was connected to the system for 29 minutes, used CPU or computer time for 4 seconds. He used disk, but not magnetic tapes. He has 450 blocks of saved files. For this job, the highest core area used (HWM=High-Water-Mark) was 11 pages or 5.5K words, and the charge is 0.1263 unit or $9.48.

The "unit" is an accounting device which combines all charges of the service, including CPU time, disk usage, the length of connect time, the size of core used, and time of the day, and a base charge, each with an appropriate weighting factor to form an accounting formula.

FILES

1.12 Basic Concept of Files

One of the important and convenient features of a time-sharing system is that it is supported by mass storage devices. The need for mass storage during the early days of time-sharing is derived from the fact that only the most important service programs and the program being executed at the moment may be stored in the high-cost, high-speed magnetic core storage. The mass storage serves as a temporary storage for programs and data not being processed at the time. When the user's turn comes, his program and data will enter the core storage. When his allotted time is finished, the program and data in the core return to the mass storage. Such transfer of program and data is an important and unique operation in all time-sharing systems, and is called swapping. The portion of the mass storage, magnetic disk and/or magnetic drum, assigned for swapping is called a swapping device.

The space required for swapping is a relatively small portion of the storage available in the mass storage devices. Thus a time-sharing system generally is characterized by a very high reserve capacity of auxiliary storage. The most frequent use of this capacity is to accommodate users' programs and/or data. These stored programs and/or data are called files.

Each language processor in the time-sharing system contains facilities for file management and file manipulation, and this information will be discussed in various chapters in this book. It will be useful at this point, however, to introduce some basic information and concepts.

The basic unit of information in a file is called a record. If a file is visualized as consisting of a deck of punched cards, then each card becomes one record. The information content of one record varies from case to case. A blank card contains no information, and it is called a null record. A FORTRAN source program record is limited to a maximum of 72 characters/record. For a PIL program, there is no practical limit to the length of a record.

For the purpose of identification, each file is given a name. Once the names are established, the computer will maintain a directory so that users need not be concerned with the exact locations or addresses on the disk to locate their files. For the DEC System-10, the format of a complete name of a file is:

DEV: NAME.EXT [m,n] <xyz>

where:

DEV: = name of device on which the file is stored. If this part is omitted in the complete name, it is understood that the device is user's assigned disk area.

NAME = file name consisting of one to six letters and/or digits, with no embedded blank.

.EXT = file extension consisting of zero to 3 letters and/or digits with no embedded blank. See more explanations below.

[m,n] = the PPN of the person who created or owned the file. Note the use of square brackets.

<xyz> = a three-digit (octal) protection code. See more explanation below. Note the use of angular brackets.

The file extension is the part of file identification used to indicate the language or format of the file. The following are the most frequently used file extensions.

.PIL	A PIL (language) program file
.FOR	A FORTRAN source program file
.REL	A relocatable binary file, or the "object deck"
.BAS	A BASIC (language) source program file
.BAK	A backup file
.DAT	A data file
.TMP	A temporary file
	A null extension (no extension)

Examples:

NEWTON.PIL	A PIL program file named NEWTON.
NEWTON.FOR	A FORTRAN program file named NEWTON.
NEWTON.REL	An object program compiled from NEWTON.FOR
NEWTON.BAS	A BASIC program file named NEWTON.
FOR01.DAT	A data file named FOR01.

Symbols "*" and "?" are used as "wild cards" to represent a class of file names or extensions. The following examples will demonstrate their use:

Examples:

NEWTON.*	All files named NEWTON of any extension.
*.FOR	All FORTRAN files.
.	All files.
F????.DAT	All data files whose names are 5 characters or less and begin with F.
D12??.D??	A files whose names begin with "D12" and contain 5 characters or less, and whose extensions begin with the letter D and contain 3 or less characters.
D12??.*	All files whose names begin with "D12" and contain 5 characters or less.

The protection code is a 3-digit octal number xyz, each digit ranging from 0 to 7. Each digit defines a protection level of the file against a certain class of users:

x = protection level against the file owner himself.

y = protection level against users sharing the same project number.

z = protection level against the general public.

The levels of protection range from 0 to 7, and level 7 is the highest. The exact definition of each protection level is given below:

Code Digit	Access Protection*
7	No access privileges
6	Execute only
5	Level 6 + read privilege
4	Level 5 + append privilege
3	Level 4 + update privilege
2	Level 3 + write privilege
1	Level 2 + rename privilege
0	Level 1 + change protection privilege

The access protection can be changed by executing the RENAME or PROTECT monitor command (see Chapter 8) or by using the service program PIP (see Chapter 7). Since there are 8 levels of protection in each of three classes of users, there are 512 different shades of protection-level combinations possible. Normally, one need only be concerned with a few commonly used codes:

Protection Codes	Applications
077,177	Strictly private and non-sharable, such as grade files maintained by an instructor.
057,177	Sharable within a project, for example, a program to be shared by all students in a course.
055,155	Sharable with the computer community, but the file may not be modified by anyone except the file owner.

The System assigns a default protection level of 057, set automatically by the computer if the person does not specify any protection code when he creates the file. In some coursework, instructors may arrange to have the default protection level automatically set at 077. In such a case, the protection code of a student's file is 077 to his classmates, but is 057 to his instructor.

*Subject to minor local variations. For example, at the University of Pittsburgh, access protection designated by the x-digit has been modified slightly.

EXERCISE ON A TIME-SHARING TERMINAL

For a person with no prior experience with using a computer, it is quite natural for him to feel intimidated when he gets on the computer for the first time. Beginners should feel assured by the fact that very little they do can hurt the computer, except if he gets physically violent and abuses the computer equipment. A session on a terminal to become familiar with its function and operation is highly recommended. The following is a recommended exercise.

(1) With a valid PPN and a password, practise sign-on and change-password procedures. Warning: Do not mix up or forget your new password, or else you will not get back on the computer again.

(2) Once signed on, type any gibberish, return the carriage, and watch the error message from the computer. Always wait for the prompt symbol "." to appear, then type in your line. Don't leave a blank or space after the ".", and don't forget to return the carriage at the end of each line.

(3) Copy a file into your own disk for the terminal exercise. For example, to copy a new bulletin of the System, use the following command:

```
.COPY NEWS.DAT=SYS:NEWS
```

Note the period in the first column is already furnished by the computer; you just type in the rest of the line and return the carriage. After this, use the DIRECT command the and R QUOLST command to find out the status and quota of your disk storage.

(4) After the file is copied into your storage, do the following exercises:

a. Print out the file by the command:

```
.TYPE NEWS.DAT
```

After a few lines are typed out, kill the typing job by either a CTRL-O or multiple CTRL-C (twice or more). The news file is quite long and a complete typeout will take a long time. If you are curious about what the rest of the news bulletin is, apply the following command:

```
.PRINT NEWS.DAT
```

and a printer copy will be produced at the printer. The printout will have your programmer number printed in big block letters on the first page for identification.

b. There is a group of monitor commands that controls the functions of a terminal. They are discussed in Chapter 8 on Operating System commands. However, several commands may be useful enough to the beginner that they will be given here for exercise:

```
.TTY WIDTH n
```

This command will set the right margin of the terminal at the nth column. The value "n" may range from 17 to 200. When you sign on

to the System, the right margin is automatically set at 72.

.TTY PAGE

After this command is given, a CTRL-S will suspend the output (but not kill it), and CTRL-Q will resume it. The purpose is to stop the output in order to examine the output that has already been produced.

After setting the right margin at a new value and giving the TTY PAGE command, repeat the exercise of typing out NEWS.DAT. Use both CTRL-S and CTRL-Q to control the printing.

(5) While still signed on, try to change to another system. Can you do it?

(6) Check your logout quota status. If you are still under your quota, keep "stuffing" your storage by repeating step 3 above (each time using a new file name), until you have gone over the quota. Confirm that by using the R QUOLST and DIRECT commands. Try to sign off in this condition.

(7) Clean up your disk storage and sign off.

(8) Repeat steps 1 through 7 by first signing on purposely on the wrong system. What are the consequences? What is the warning message from the computer? What are the things you cannot do in the wrong system? What are the things you can do in either system?

When you complete this exercise with reasonable facility, you may consider yourself granted a beginner's driver license. Congratulations!

REFERENCES

1. A PRIMER FOR PITT TIME-SHARING SYSTEM (PTSS), T. W. Sze, University of Pittsburgh, Pittsburgh, Pennsylvania; 1970.

2. INTRODUCTION TO A TIME-SHARING SYSTEM, T. W. Sze, University of Pittsburgh, Pittsburgh, Pennsylvania; 1972.

3. ALL ABOUT TELEPRINTER TERMINALS, Datapro Research Coporation, Delran, New Jersey; 1976.

4. LA36 DECwriter II USERS MANUAL, Digital Equipment Corporation, Maynard, Massachusetts; 1974.

5. INTRODUCTION TO COMPUTING AT PITT, DEC-10 Documentation-1, Computer Center, University of Pittsburgh, Pittsburgh, Pennsylvania; April, 1980.

6. UNIVERSITY COMPUTER CENTER, ACADEMIC SERVICES, Computer Center, University of Pittsburgh, Pittsburgh, Pennsylvania; 1978.

7. INDEX OF COMPUTER CENTER DOCUMENTATION AND SERVICES, Computer Center, University of Pittsburgh, Pittsburgh, Pennsylvania; September, 1978.

8. INTRODUCTION TO DECSYSTEM-10: TIME-SHARING AND BATCH, T. W. Sze, University of Pittsburgh, Pittsburgh, Pennsylvania; First Edition, 1974; Second edition, 1977.

CHAPTER 2

TEXT EDITOR

Everything must have a beginning. From a user's point of view, his starting point is to enter his program and/or data into the computer. The DEC System-10 is mainly disk-based machine. That means, the computer will look in the user's disk area for the program a user wants to execute. Therefore, in order to do any computer processing, a user must first place his program and/or data in the disk. The text editor is a system program that will enable the user to perform this task.

2.1 Introduction

The UPDATE (University of Pittsburgh DAta and Text Editor)* is a service program with which a user can correct, modify, duplicate, or delete parts of a stored program or data file.

In order to edit a program or data file, it must be one already stored on disk, magnetic tape, or DECtape. However, if the source material is on tape, using UPDATE will result in an edited copy of material stored on disk, and the original material on tape is unchanged. Ultimate changes on tape still require the use of another service program, such as PIP (see Chapter 7), to delete the old file on tape and to transfer the new file from disk to tape. Otherwise, the new file is re-copied onto the tape so that the old and the new copies co-exist together on the tape. Therefore, for all practical purposes, UPDATE is used as a disk-to-disk editor, taking source material from the disk and storing the edited copy back on the disk. Discussions in this chapter are based on such disk-to-disk editing.

After the user signs on in the usual way, he can get the service of UPDATE by typing the following monitor command:

```
.R UPDATE
```

or

```
.UPDATE
```

*Developed by Gerald W. Bradley, University of Pittsburgh (Reference 7).

When the UPDATE editor is assigned, the computer will first ask for the name of the input file to be edited in this manner:

```
.R UPDATE
INPUT=>
```

The user will then type in after the greater-than sign the file name, extension, file owner's PPN if the user is not the owner. For example:

```
.R UPDATE                    .R UPDATE
INPUT=>SAMPLE.FOR            INPUT=>SAMPLE.FOR[115103,320571]
>                            >
```

When the greater-than sign is again printed by the computer, UPDATE is ready to accept editing commands, and editing on the specified file can begin. As the editing proceeds, whenever UPDATE is ready to accept a command or an insertion, a sign ">" is printed out as a prompt symbol. The first space after the sign should be considered as column No. 1.

The above process may be shortened by using the following formats:

```
.UPDATE SAMPLE.FOR           .UPDATE SAMPLE.FOR[115103,320571]
>                            >
```

2.2 Selected Terminology

The following terms will be used quite frequently in the discussion of the UPDATE commands:

(1) Record, or Line

A record or a line is a basic unit of information in a file. If a file consists of a deck of punched cards, each card becomes one record. For a file stored on disk, one record actually is a tiny length of track on the disk. The information content for one record varies from case to case. In a FORTRAN program, each record is limited to a maximum of 72 characters including blanks, although each statement may extend for several records if needed. Sometimes, there is no information at all on a record, such as a blank card, and this is called a null record.

(2) Pointer

Once the input file is specified and loaded, the UPDATE at that time is positioned at the first record, or line, of the file. At that point, editing commands will refer to text material with that line as a reference point. Later, if one wishes to make editing steps at another line, the UPDATE should be re-positioned by appropriate commands. For convenience, we shall assume an imaginery "pointer" which indicates the position of the record being aligned. Thus, such a statement as "moving the pointer forward 5 records" should now make sense.

(3) Line Numbers, Absolute and Relative

A file begins with record No. 1, then No. 2, etc. Such line numbers represent the true positions of the records in the file, and are called the absolute line numbers. On the other hand, it is often convenient to use as a

reference the line currently pointed to and say, for example, "move forward 5 lines" or "back up 3 lines". These are then relative line numbers. Absolute line numbers are always expressed by unsigned positive integers, and relative line numbers by signed integers. Use "+" sign for forward reference and "-" for backward reference in specifying relative line numbers. Note that a file always begins at line number 1, and its line numbers are always contiguous. Therefore, if lines 4 and 5 are deleted during editing, then line 6 becomes line 4, 7 becomes 5, etc.

(4) Delimiter While the pointer indicates the position of a line in a text, the position of text within a line is indicated by the use of delimiters. These delimiters may be thought of as quotation marks in the English language, except that any special character may be used as a delimiter in UPDATE. Thus, if one wishes to set off the last three words of this particular paragraph, he may specify:

"he may specify:" or /he may specify:/
or ?he may specify:? or $he may specify:$ etc.

Because of its similarity with the quotation, the string set off by a pair of delimiters will be referred to as a "quoted string" or simply a "quotation". There are several important rules of delimiter usage in the UPDATE editor:

A. Use consistent characters as delimiters for a quotation. While any special character may be used as a delimiter, the choice of the beginning-of-quotation (BOQ) delimiter automatically decides the use of the same character as the end-of-quotation (EOQ) delimiter. The following examples should be self-explanatory:

Valid Use of Delimiters	Invalid Use
"quoted text"	(quoted text)
(quoted text(	<quoted text>

B. If a quoted string contains a special character, that particular character should not be used as a delimiter for this quotation. For example, if we wish to quote a string "less than $5.00" and use "$" as a delimiter, the result will be misinterpreted by UPDATE.

C. If several quotations are placed in one UPDATE command, the following rules apply:

   a. The first BOQ delimiter determines the character to use.

   b. Multiple quotations must all refer to materials on the same record.

   c. When multiple quotations are placed together, two adjacent delimiters should always be merged into a single one to avoid ambiguity. In other words, a delimiter should not only serve as the BOQ delimiter for the following quotation, but also as the EOQ delimiter for the preceding quotation. Thus a general appearance of a multiple quotation will be something like this:

/QUOTE 1/QUOTE 2/QUOTE 3/

If this multiple quote is written as:

/QUOTE 1//QUOTE 2//QUOTE 3/

it will actually be interpreted by UPDATE as 5 quotations, the second and the fourth being null strings.

D. The contents of a quotation must be exact and unique. When UPDATE receives a quoted string, it will try to search in the pointed line for a group of characters exactly matching the quotation. In such a matching process, the capital letters, the lower cases, the blanks, special characters, and control characters are all legitimate and different characters. For this reason, a quoted string must be given in the exact way as in the pointed line.

Example: Suppose we wish to quote the underscored portion below:

50 Y1 = Y0 + Y 1

Correct Quotation	Incorrect Quotation
/Y 1/	/Y1/

E. When UPDATE searches a line text to match a quotation, it begins with the character in column one. As the search moves to the right, and a match is found, the search is completed. If a quoted string appears several times in a text line, UPDATE will always pick the string nearest to the first column. Therefore, if we wish to sepcify non-unique strings further to the right, the string must be expanded in front and/or in the back until the string is unique, or else it is the first such quotation when the search starts from the left end.

Example: Suppose we wish to quote the underscored portion below:

501 IF(Y.LT.0.0001) GO TO 510

Correct Quotation	Incorrect Quotation
/0.0001/	/0/
/.000/	/01/
/01)/ etc.	/00/

F. A single quotation followed immediately by an integer means this quotation begins from column indicated. For example, the quotation /01/19 means the character string "01" that begins at column No. 19.

G. All quotations must be bracketed within a pair of delimiters. Unclosed quotation is an error.

A PRIMER OF UPDATE EDITOR

The text editor UPDATE contains several dozens of editing commands. For a beginner, it would be a mistake to attempt to learn them all at one time. Experience has shown that most editings are done with a limited set of editing commands. Complex command functions can usually be accomplished by applying several simpler commands in sequence. For the sake of learning efficiency, it would be much more cost effective for a beginner to concentrate on a few basic editing functions and commands. They are:

(1) To move a pointer to a designated line.

(2) To make changes on a pointed line.

(3) To delete the pointed line or lines.

(4) To type out the content of the pointed line or lines.

(5) To insert a line at a designated place.

(6) To conclude the editing.

The commands given in the following sections pertain to these basic functions. All UPDATE commands must have a "$" in the first column. The spelling of each command may be shortened to just the first two letters. Misspelling after the first two letters will be ignored and will not be considered an error.

2.3 Movement of Pointer, $TO, $AT and $TRAVEL

When the UPDATE first opens an input file, the pointer is always positioned at line No. 1. There are three commands one may use to move the pointer elsewhere, and they are $TO, $AT and $TRAVEL.

$TO will move the pointer to a specified place, and once there the new line is typed out for verification.

$AT performs the same function as $TO, but the typing of a new line is suppressed.

$TRAVEL performs the same function as $TO, and the command is "remembered". The same $TRAVEL command can be executed again later by issuing a $GO command.

All three commands have the same command formats and variations, and those for $TO are listed below. Variations of formats would be the same for the other two commands, simply by replacing $TO in the following listing by either $AT or $TR.

A.	$TO N	Move the pointer to line N.
B.	$TO +N	Move the pointer N lines forward.
C.	$TO -N	Move the pointer N lines backward.
D.	$TO /TEXT/	Move the pointer forward from the present line until it encounters a line with the exact string /TEXT/ in it.
E.	$TO $TEXT$	Similar to case D above, with an exception that the match will not consider the difference between upper and lower case letters, nor will it take into account any blank between characters.
F.	$TO /TEXT/K	Search for the string TEXT that begins at column No. K.
G.	$TO $	Move the pointer to the last line.

Notice the mode of search in cases D, E and F. The search starts from the line below the pointed line. If there is a string TEXT in the pointed line, it will not be found. If one wishes to move to the first appearance of /TEXT/ or $TEXT$ while he is in the middle section of the file, he should issue the command $TO 1 first before giving a $TO/TEXT/, or $TO/TEXT/K, or $TO $TEXT$$ command. This is so that he will not miss any earlier existences of the string TEXT in the lines before the pointer. But even so, such a search would miss line 1, unless the user examines the line typed out after the command $TO 1. This problem may be solved by inserting a blank line as line 1 and later removing it before finishing the editing job.

To move forward one line, two ways are possible: Either use $TO +1 command, or simply return the carriage.

To move backward one line, either use the command $TO -1, or press backspace key and then return the carriage.

While moving the pointer back and forth during the editing job, it will become difficult to keep track of the line number of the pointer. The command $WHERE will cause a number typed out enclosed in brackets to indicate the current pointer line number.

Example: Suppose you wish to examine every FORMAT statement in your FORTRAN program. You will first call the file using the UPDATE. Then apply the following command:

```
>$TRAVEL /FORMAT/
```

The first time you apply it this way, it will move the pointer to the first FORMAT statement, and print it out. Any revision of the statement may be done there and then. The movement to the subsequent FORMAT statements may be accomplished by giving the command:

```
>$GO
```

Naturally, if a FORMAT statement in the program is misspelled, the $TRAVEL command will not find that statement.

2.4 Change of Text Material, $CHANGE, $ALTER and $SUBSTITUTE

When the appropriate line is positioned by the $TO, $AT or $TRAVEL command, editing changes may be performed using the $CHANGE, or $ALTER command. The standard format is:

```
$CHANGE   /OLD TEXT/NEW TEXT/
```

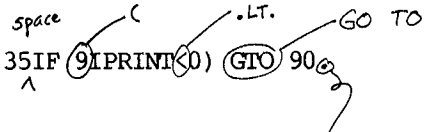
For multiple changes on the same line, the command format is:

```
$CHANGE   /OLD 1/NEW 1/OLD 2/NEW 2/OLD 3/NEW 3/...
```

The rules of delimiters in multiple quotations have been discussed before and are applicable here. Again, the delimiters for multiple quotations must be consistent for all quotations.

As a convenience feature, after the $CHANGE command is executed, the entire new line is typed out for verification. The pointer position remains unchanged.

Example: Suppose the indicated changes are required as shown below:



The following two lines show first the editing command of $CHANGE and then the edited text automatically typed out after execution. (Remember our convention in this book --- User's input line shown in *italics*):

```
>$CHANGE/5/5 /9/(/>/.LT./G/GO /.//
   35 IF (IPRINT.LT.0) GO TO 90
```

There are, of course, many other ways to write the above $CHANGE command to achieve the same result.

$ALTER is used in the same way as the $CHANGE command, except that $ALTER does not allow multiple changes. Its main usefulness is in the compounded editing commands, as will be illustrated in a later section.

$SUBSTITUTE differs from $CHANGE or $ALTER in this manner: The command $CHANGE or $ALTER is used to change a string in one single line positioned by the pointer. The command $SUBSTITUTE is used to alter a string in the entire file beginning from the pointed line. Again, the string of characters to be changed must be specified exactly and uniquely. Otherwise, inadvertent changes will result at unintended places. For example, if one wishes to change the variable X into Y in a certain program, specifying $SUBSTITUTE/X/Y/ would change every X-character into Y-character. Thus, inadvertently, another variable with the name "INDEX" would become "INDEY", and the exponential function name EXP would be changed to EYP.

There are two variations for the command $SUBSTITUTE:

A. $SUBSTITUTE /OLDTEXT/NEWTEXT/

Starting from the pointed line, this command will search for a string /OLDTEXT/ and each time upon finding it, change it into /NEWTEXT/ until the end of the file is reached.

B. $SUBSTITUTE /OLDTEXT/NEWTEXT/K

Starting from the pointed line, this command will search for a string /OLDTEXT/ that begins at the Kth column, and each time upon finding it change it to /NEWTEXT/ until the end of the file is reached. After the $SUBSTITUTE command is executed successfully, the pointer will be relocated at the last line of the file.

2.5 Deletion of Lines, $DELETE

When a line or a group of consecutive lines are to be deleted, first position the pointer to that line or the first line of that group, using either the $TO or $AT command. Then depending on what is to be deleted, use the command $DELETE in the following ways:

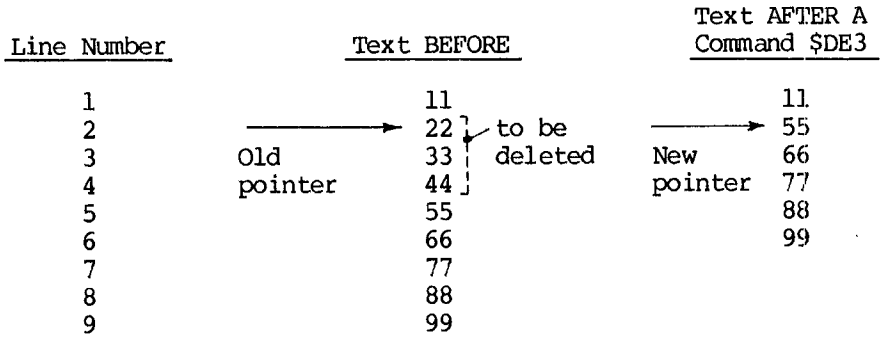
(1) $DELETE N Delete N lines beginning with the one presently pointed to. After the deletion, the pointer moves forward to the line immediately after the deleted group. If the deleted group happens to be the final N lines of the file, the pointer drops back one line and positions at the new last line. If N is larger than the number of lines left on the input file, a command $DELETE N will delete every line remaining and then type out a "?" to indicate error. This feature is actually quite useful when one wishes to delete the rest of the text but does not know how many lines there are. Then, he can simply issue a command of $DELETE 10000, or any number larger than the number of remaining lines.

(2) $DELETE This is automatically interpreted as $DELETE 1.

(3) $DELETE $ Beginning with the currently pointed line, this command will erase the rest of the file.

$DELETE -N is NOT a valid command.

Example: See below for the "Before" and "After" with a $DELETE command:



Note that although the pointer is positioned at a new line of text, the line number of the pointed line remains the same. The line numbers and the text are then automatically readjusted.

2.6 Output of Lines, $TYPE

Frequently, it is desirable to display the text of a line on the terminal for examination. The UPDATE command for this function is $TYPE. The following shows the variations:

A. $TYPE N — This command will type out N lines beginning with the present line. The position of the pointer remains unchanged.

B. $TYPE — Same as $TYPE 1.

C. $TYPE $ — This command will type out the currently pointed line and the last line of the file. Pointer position remains unchanged.

2.7 Line Insertion

UPDATE will regard any user input as UPDATE command if column 1 is a "$" character. Conversely, UPDATE will regard any user input as line insertion if the line does not begin with a "$" in column 1. When a line is inserted, it will always be inserted after the currently pointed line. If you wish to insert a line before the pointed line, you must precede your insertion by a "$BEFORE" command. When a new line has been inserted, the pointer will move forward one line, making the new insertion the currently pointed line.

Beside adding lines to an old file, this process is particularly useful in creating new files. The process of creating a new file is outlined as follows:

(1) Call for UPDATE and give a file name that does not yet exist. For example:

```
.UPDATE NEW.FOR
```

where NEW.FOR is a file name given to the new file to be created.

(2) The pointer of the blank file called by the UPDATE will be positioned at line zero. Type in the new file, one line at a time. Each line is terminated by a carriage return in the conventional way of typing.

(3) While the new file is being created, the editing commands can be applied to move the pointer, to type out the lines, to delete or to change the contents of a line.

(4) When all lines are entered, exit from UPDATE by a command $END.

2.8 Completion of an Editing Session, $DONE, $END and $FINISH

All three commands signify the end of current editing of the file. They differ in how the file should be stored and named.

When the $DONE command is issued, UPDATE will ask the user to supply a file name for the edited file, for example:

```
>$DONE
CATALOG NAME=>SAMPLE.FOR
6 BLOCKS WRITTEN ON SAMPLE.FOR[115103,320571]

EXIT

.
```

If the file name and the extension given here are exactly the same as those of the old file, the old file is replaced by the new file. As a safety measure, the old file is retained in the storage with the extension changed to BAK (for "backup"), in case the user changes his mind about his revisions. If either the name or the extension or both are different from those of the old file, a new file is created and stored on disk along with the old file, and the old file is not disturbed. If the name and the extension given during the cataloging are exactly the same with those of some other file in the disk storage, naming two different files with the same name causes an error, and the UPDATE will reject the duplicate name and ask for a new name. This is illustrated below:

```
>$DONE
CATALOG NAME=>NEW.FOR
FILE DSK:E20016.TMP[115103,320571]to NEW.FOR[115103,320571]
        RENAME error (4) - Already existing file
CATALOG NAME=>SAMPLE.FOR
6 BLOCKS WRITTEN ON SAMPLE.FOR[115103,320571]

EXIT

.
```

The catalog name can also contain a protection code specification, for example, SAMPLE.FOR<155>. When the protection code is omitted, the UPDATE will automatically assign a protection code of 057.

When the $END command is issued, a fast exit is accomplished and the edited file will have the same file name, extension and protection code as those of the old file. The old file becomes a BAK file. After the storage process is completed, UPDATE returns the user to the monitor.

When the $FINISH command is issued, it will perform the same function as $END. However, instead of returning the control to the monitor, the user will retain the service of the UPDATE editor and can then start a new editing job. Therefore, this command is equivalent to issuing two successive commands: an UPDATE command of $END followed by a monitor command of .R UPDATE.

OTHER UPDATE COMMANDS AND PROCEDURES

When UPDATE is called, several events happen:

(1) The UPDATE program is loaded into the computer memory assigned to the user.

(2) Two disk areas are assigned as working files. One is used as the input file labeled as E1 and the other is used as the output file labeled as E2. The actual file names assigned are E100xx.TMP and E200yy.TMP respectively, where "xx" and "yy" are numbers arbitrarily assigned.

(3) After the input file name is given by the user, as requested by the UPDATE, a copy of that file is loaded into E1. If no such file name exists, E1 remains a blank file. In either case, E2 is a blank file at this point.

(4) UPDATE will read up to 100 lines (which may be specified and modified by a $FACTOR command) from E1 file into the memory.

The logic flow of the text information during an editing session is shown in Figure 2.1.

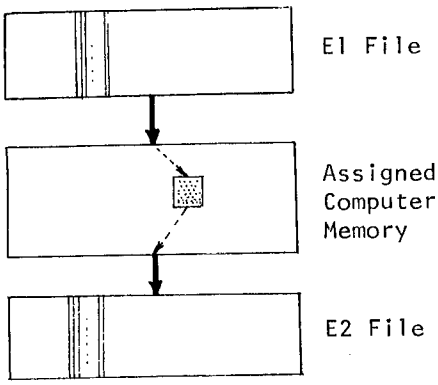


Figure 2.1
Editing Input/Output Files

Now, as the editing session progresses and the pointer advances through the file, more lines are read into the memory. When the number of lines in the memory is more than 100, or whatever value specified by a previous $FACTOR command, the lines behind the pointer are written into the E2 file. Thus, if the pointer keeps advancing forward, more lines are transferred into E2. When the editing is finally completed, all lines in the core, and all the remaining lines in the E1 file are copied onto the E2 file. The E2 file is then renamed by a name designated by the user, and the E1 file is erased. It is significant to note from Figure 2.1 that the movements of lines from E1 to the computer memory, then onto the E2 file, is always in one direction only.

Thus, if the pointer is moved backward, there will be complications. If the pointer, after moved backward, is pointing to a line still in the core memory, events are still normal. If the pointer is positioned at a line no longer in the core memory, that line has already been copied onto the E2 file and cannot be retrieved because the transfer between the memory and E2 is one-way only, as shown in Figure 2.1. This will set forth a sequence of events described as follows:

First, the lines in the memory and all the remaining lines in the E1 file will be copied onto the E2 file. The E2 file is then closed. The E1 file is erased. The E2 file is renamed as the E1 file. The new E1 file is read into the computer memory containing the line positioned by the pointer. The backing-up of the pointer is now finally accomplished. These events resemble a

situation when a driver misses an exit on a one-way urban beltway. In order to exit at the missed exit point, he must drive the whole way around the one-way highway and gets off at the desired exit. However, such events at the editing session are "transparent" to the user, because at the terminal he will be unaware of these. But this situation does suggest that backing up in positioning a line should be done sparingly.

When E2 is closed, it is renamed by a name designated by the user if the closing command is $DONE, and the input file is not disturbed. If the $END or $FINISH command is used, the E2 file is renamed by the same input name, and the input file is renamed as a BAK file.

If the editing involves an auxiliary file as an input or output for the editing, another disk file labeled E3 is assigned. This happens with the command $ONTO or $FROM (See Section 2.16).

2.9 Line Insertion Mode

UPDATE will treat all input lines that start with a $-sign in the column 1 as an UPDATE command. Conversely, UPDATE will treat any input information without a $-sign in column one as a non-command and as information to be inserted in the text.

There are two modes of line insertion:

(1) Insertion after the pointer

A. Insertion of lines typed at the terminal

Any input information to the UPDATE without a dollar sign in column one will automatically be inserted immediately after the current line. When the insertion of one line is completed, the pointer moves forward one number, so that it is now positioned at the newly inserted line. The next typed line will be inserted immediately after the previously inserted line, and again the pointer moves to the newly created line. This feature makes it very convenient to use the terminal keyboard to create a file.

The insertion mode is suspended whenever an UPDATE command (with a $-sign in the first column) is issued.

Example: Observe the "Before" and the "After" of an insertion procedure:

Line Number	Old Text and Pointer Position	User types in:	New Text and Pointer Position
1	11	*AA*	11
2	→ 22	*BB*	22
3	33	*CC*	AA
4	44		BB
5			→ CC
6			33
7			44

So the UPDATE interprets every input line beginning with a $-sign as an UPDATE command. This may develop into a dilemma if the user attempts to insert a line that begins with a $-sign. For example, consider the statement: "$5.00 IS TOO MUCH TO PAY". When this statement is inserted, UPDATE will puzzle over the meaning of "$5." as an UPDATE command, and the execution results in an error report.

There are several ways to solve this problem: One is to insert a line: "X5.00 IS TOO MUCH TO PAY", and then use $CHANGE command to change the first "X" into "$". Another way is to insert the line: " $5.00 IS TOO MUCH TO PAY", leaving a blank in column 1, and then remove it using the $CHANGE command. If there are many such statements to insert (for example, in preparing a control file for batch processing), the process may be simplified by an UPDATE command:

```
$IS #
```

where "#" can be any special character except ";". The effect of this command is to replace the format of all subsequent editing commands from $XX to #XX, therefore allowing insertion of lines beginning with "$", but disallowing insertion of lines beginning with "#". A command #IS $ later will restore the UPDATE to the normal command format.

To insert a blank line, one should not simply press the carriage return, because that action would merely move the pointer forward one line, and no insertion of any kind is accomplished. A blank line may be inserted by typing (at least) one blank then returning the carriage.

B. Insertion of a stored file

If the lines to be inserted are already stored on disk as a file whose name is given, for example, as NAME.EXT, by its owner with PPN of [m,n], the insertion can be made simply in this manner:

a. Position the pointer at the line immediately before the insertion.

b. Issue the following UPDATE command:

```
$INPUT = NAME.EXT[m,n]
```

As usual, if [m,n] are the user's own numbers, they may be omitted in the command. After the insertion, the pointer moves forward to the last inserted line. This command is frequently used for merging parts of program or data files.

Although the inserted lines come from a stored file on the disk, the UPDATE editor treats them the same way as if they come from the terminal. And hence, the lines in a stored file insertion are subject to the same UPDATE editing rules, particularly about the interpretation of the first column character. If a file will be used as a straight forward insertion of lines, it should be inspected first to see if there is no "$" sign in the first columns. If there is any "$" in the first column, appropriate action, such as "$IS #" command, should be taken prior to the insertion. On the other hand, another avenue of issuing editing commands in addition to the terminal is now opened up. One now may use either the terminal or a stored file to issue editing commands.

UPDATE is greatly enhanced when a sequence of fixed UPDATE commands, which will be executed frequently, is stored as a file. Execution of this sequence of commands can be carried out automatically simply by the $INPUT command. These stored files now become editing programs and can be used over and over.

Example: The following is a file ATTEND.DAT that needs updating each week. The contents with the column numbers are shown below:

```
(Column )                  11111111112222222223
(Numbers )        12345678901234567890123456789 0

                  PERSON A       10110      1
                  PERSON B       10011      2
                  PERSON C       01101      3
                  PERSON D       10111      4
                  . . .
                  PERSON Z       00110    126
```

Suppose the requirement of updating the file is as follows. Remove column-16; shift columns 17-20 to the left by one column; and replace column-20 by zeros.

An "editing program" may be designed and stored as EDIT.PRG that contains the folowing statements:

```
$AT1
$SUBSTITUTE /1/0/16
$AT1
$SUBSTITUTE /0//16
$AT1
$SUBSTITUTE / /0 /20              ( =blank)
```

This sequence may be executed as shown below:

```
.UPATE ATTEND.DAT
>$INPUT=EDIT.PRM
>$END
1 BLOCK WRITTEN ON ATTEND.DAT[115103,320571]

EXIT
```

When this editing program is executed, columns 17-20 will be shifted to the left by one column.

(2) Insertion before the pointer

To insert material before the pointer, first apply the command $BEFORE. Then all lines with no ($) sign at the column-1 will be inserted before the pointer. In the meantime, the pointer will move to the last inserted line. The Insertion mode is terminated by any UPDATE command. See the example below:

Example: Observe the "Before" and "After":

Line Number	Old Text and Pointer Position	User types in:	New Text and Pointer Position
1	→ 11	*$BEFORE*	AA
2	22	*AA*	→ BB
3	33	*BB*	11
4	44		22
5			33
6			44

2.10 Compounded Editing Commands

The UPDATE commands discussed so far have the format of one command per command line. When several commands are issued on a single command line, they become a compounded command. The general format of a compounded command is:

$COMMAND 1; COMMAND 2; COMMAND 3; ...

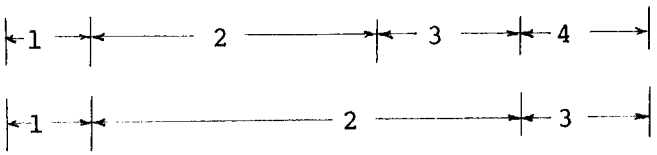
The semicolons ";" are used to separate the successive commands, and therefore no semicolon should appear after the last command in the compounded structure. Also, if any of the commands contains a quotation of string, the string must not contain any semicolon-character, because it will be misunderstood as a command delimiter. Note that the dollar sign "$" is needed only for the first command. There are several straight-forward rules for constructing a compounded UPDATE command:

(1) All commands of a compounded command must fit in a single command line.

(2) The individual commands in the compounded command are executed in their natural order from left to right.

(3) Certain commands may cause ambiguity and error if they are followed by other commands in a compounded structure. Consider the following compounded command:

$TO 5; CHANGE /OLD1/NEW1/; TO/TEXT/; TYPE 4



Interpretation 1:
4 single commands

Interpretation 2:
3 single commands
with multiple string
changes in $CHANGE

It can be seen that the interpretation is ambiguous and it will be unpredictable how this command would be actually executed. To avoid this problem, commands of this kind are always regarded as the last command in the structure, even if there are more commands after them. If more commands are given after them in a compounded command, the added commands are simply ignored, and no error return

signal is returned. Thus, when the above example is executed, the part "TO /TEXT/; TYPE 4" will not be executed. In order to accomplish the function of the above compounded command, the above example should be modified to:

```
$TO 5; ALTER /OLD1/NEW1/; TO /TEXT/; TYPE 4
```

The ambiguity is now removed because the $ALTER command can allow only one change of string.

There are certain UPDATE commands that must be physically the last command in a compounded structure. These commands are listed below:

Group	UPDATE Commands
Multiple string change	CHANGE
Change of command format	IS
Auxiliary file operations	INPUT, ONTO, FROM
End of editing session	END, DONE, FINISH

Commands appended to any of the above commands in a compounded structure will simply be ignored.

Compounded command structure format provides a convenience for input commands. It also is a basis on which a simple and powerful editing program can be built, especially when it couples the usage of $TRAVEL and $GO commands in the compounded structure:

Example: $AT 1; TRAVEL /FORMAT/7; WHERE; GO
Function: Beginning at line 2, search for the string of characters "FORMAT" that begins at column-7. When it is found, type out the line itself and the line number. Repeat the function until the end of the file is reached. In other words, this compounded command will print out all FORMAT statements and where they are in a FORTRAN program. Notice this compounded command will miss line 1; why?

Example: $AT1;TR/ /;AL/ / /;AT-1;GO
Function: Beginning from line 2, all multiple blanks will be reduced to single blank.

Example: $TR/C/1;DE;AT-1;GO
Function: Remove all Comment Lines in a FORTRAN program.

2.11 Move Command, $MOVE

This command will move a block of lines to somewhere else in the file. There are two general formats: One is a single-command format, the other a multiple-command format.

(1) Single command format

The merging of $MOVE N and $TO commands forms a single-command that will move an N-line block to a place designated by the $TO command. Before the move, the pointer should always be positoned at the first line of the N-line block. Immediately after the move, the pointer will always be at the last line of the N-line block at its new place. Because of the $TO command, there are many variations of the $MOVE N TO commands. They are listed below:

A.	$MOVE N TO M	Move N-line block to a new position so that the first line of the block is now line No. M.
B.	$MOVE N TO +M	Move an N-line block to a new position starting immediately after the line which has a relative line number of +M from the last line of the block before the move.
C.	$MOVE N TO -M	Move an N-line block to a new position immediately before the line which has a relative line number of -M, relative to the first line of the block.
D.	$MOVE N $	Move an N-line block to the end of the file.
E.	$MOVE N TO /TEXT/	Move an N-line block and place it immediately after the line beyond the pointer that has the first appearance of the string /TEXT/.
F.	$MOVE N TO /TEXT/K	Move an N-line block and place it immediately after the line beyond the pointer that has the first appearance of the string /TEXT/ that starts at the Kth column.

Example: $MOVE N TO M

Line Number	Old Text and Pointer Position	$MOVE Command	New Text and Pointer Position
1	11	*$MOVE 2 TO 3*	11
2	→ 22		44
3	33		22
4	44 ←		→ 33
5	55		55
6	66		66
7	77		77

Example: $MOVE N TO +M

Line Number	Old Text and Pointer Position	$MOVE Command	New Text and Pointer Position
1	11	*$MO 2 TO +3*	11
2	→ 22		44
3	33		55
4	44		66
5	55		22
6	66		→ 33
7	77		77

Example: $MOVE N TO -M

Line Number	Old Text and Pointer Position	$MOVE Command	New Text and Pointer Position
1	11	*$MO 2 TO -3*	11
2	22		55
3	33		→ 66
4	44		22
5	→ 55		33
6	66		44
7	77		77

Example: $MOVE N TO $

Line Number	Old Text and Pointer Position	$MOVE Command	New Text and Pointer Position
1	11	*$MO 2 TO $*	11
2	→ 22		44
3	33		55
4	44		66
5	55		77
6	66		22
7	77		→ 33

Example: To interchange a pointed line with the next line.

Line Number	Old Text and Pointer Position	$MOVE Command	New Text and Pointer Position
1	11	*$MO 1 TO +1*	11
2	→ 22		33
3	33		→ 22
4	44		44
5	55		55
6	66		66
7	77		77

Example: To interchange a pointed line with its preceding line.

Line Number	Old Text and Pointer Position	$MOVE Command	New Text and Pointer Position
1	11	*$MO 1 TO -1*	11
2	22		→ 33
3	→ 33		22
4	44		44
5	55		55
6	66		66
7	77		77

Example: $MOVE N TO /TEXT/

Line Number	Old Text and Pointer Position	$MOVE Command	New Text and Pointer Position
1	11	*$MO 2 TO '55'*	11
2	22		44
3	33		55
4	44		22
5	55		→ 33
6	66		66
7	77		77

Example: $MOVE N TO /TEXT/K

Line Number	Old Text and Pointer Position	$MOVE Command	New Text and Pointer Position
1	1b	*$MO 2 TO /3b/1*	1b
2	→ b1		b2
3	2b		3b
4	b2		b1
5	3b		→ 2b
6	b3		b3
7	4b		4b

(b=blank)

The search for /TEXT/ starts from the next line from the current line. Thus the search will omit the current line and all lines prior to that. The following example shows an error of search:

Example: $MOVE N TO /TEXT/

Line Number	Old Text and Pointer Position	$MOVE Command	New Text and Pointer Position
1	11	*$MO 2 TO /33/1*	11
2	22		22
3	33		33
4	44		44
5	→ 55]		→ 77
6	66]		
7	77 ↓		

(Search unsuccessful when reaching the end of file, and moved lines are lost.)

(2) Multiple command format

Moving an N-line block of text can also be achieved with first a $MOVE N command, and then when the destination is accurately positioned, with another $HERE command. What actually happened is that the N-line block is temporarily stored in an auxiliary file E3, and when the $HERE command is given, the lines will re-enter the computer memory. The advantage of moving lines in this manner is that the procedure becomes less error prone because of accurate positioning of the destination. In the nine examples shown above for the single-command format, movements of lines can also be accomplished by a three-step procedure: $MOVE N, accurate positioning by $TO, and then $HERE commands. Observe the difference in the line numbers used between the single-command and the multiple-command formats.

2.12 COPY Command

This command will duplicate a block of lines elsewhere in the file. The format of the command is very similar to that of $MOVE, and so are the variations. Instead of using TO for positioning in the $MOVE command, $COPY uses AT for positioning the pointer. The variations of $COPY are listed below with similar definitions as applied to the $MOVE variations:

(1) Single command format

A. $COPY N AT M

B. $COPY N AT +M

C. $COPY N AT -M

D. $COPY N AT /TEXT/

E. $COPY N AT /TEXT/K

F. $COPY N AT $

(2) Multiple command format

$COPY N

Accurate positioning command

$HERE

These variations are again illustrated by examples:

Example A: $COPY N AT M

Line Number	Old Text and Pointer Position	$COPY Command	New Text and Pointer Position
1	11	*$CO 2 AT 3*	11
2	→ 22		22
3	33		22
4	44		→ 33
5	55		33
6	66		44
7	77		55
8			66
9			77

Example B: $COPY N AT +M

Line Number	Old Text and Pointer Position	$COPY Command	New Text and Pointer Position
1	11	*$CO 2 AT +3*	11
2	→ 22		22
3	33		33
4	44		44
5	55		55
6	66		66
7	77		22
8			→ 33
9			77

Example C: $COPY N AT -M

Line Number	Old Text and Pointer Position	$COPY Command	New Text and Pointer Position
1	11	*$CO 2 AT -2*	11
2	22		22
3	33		33
4	44		44
5	→ 55		55
6	66	Watch out for tricky minus count here. A poor feature.	→ 66 -2 lines
7	77		55
8			66
9			77

Example D: $COPY N AT /TEXT/K

Line Number	Old Text and Pointer Position	$COPY Command	New Text and Pointer Position
1	1b	*$CO 2 AT /3b/1*	1b
2	→ b1		b1
3	2b		2b
4	b2		b2
5	3b		3b
6	b3		b1
7	4b		→ 2b
8			b3
9	(b=blank)		4b

The four examples above show how $COPY command may be used in a single-command format. If $COPY is used in a multiple-command format, the commands to produce the same results as the above four examples will be:

Example A	Example B	Example C	Example D
$CO 2	$CO 2	$CO 2	$CO 2
$AT 2	$AT+2	$AT -3	$AT/3b/1
$HE	$HE	$HE	$HE

Since the pointer will be positioned at a line beyond the line of the copied group after each $COPY N command, the counting of lines is different. Therefore, observe particularly the number of lines of movement for the pointer in the first three cases.

2.13 Editing-Control-Function Switch Commands

There is a group of editing control functions that UPDATE can turn them ON or OFF by commands. When a function is switched ON, that function will be in force. Such software switches have many similar properties as a hardware switch. For example, a function will be OFF unless explicitly turned OFF, or vice versa. Turning ON a switch several times in succession is equivalent to turn it on just once.

(1) Functions permanently switched ON or OFF by UPDATE

The following is a group of editing commands that provides a variety of control functions during an editing session. It has a general format of

```
$KEYWORD = YES
$KEYWORD = NO
```

where KEYWORD represents an option, and YES or NO to indicate whether such option is to be switched ON or OFF. When a function is switched ON, the effect is permanent for the remainder of the editing session or until the function is explicitly turned OFF. When UPDATE is first called, all these switches are in the OFF condition.

A. $ARROW=YES; $ARROW=NO When this option is turned on, all control characters in the current line can be displaced by the TYPE command as either ^-character or ^-character, such as ^L or ^L, ^I or ^I.

Example: Often, in entering a text line, the shift key of the terminal is used to enter special symbols, such as "*" or "]". If by mistake, the control key is used instead of the shift key, the mistake cannot be easily detected because a control character will not be echo-printed. Observe the following segment of an editing session:

UPDATE Commands	Comments
>*$TYPE*	
DIMENSION X(10)	Printout seems OK
>*$ARROW=YES*	
>*$TYPE*	
DIMENS^[ION	Hidden non-print character
>*$CHANGE /^[//*	Remove it.
DIMENSION X(10)	It's gone.
>	

B. $EDIT=YES; $EDIT=NO When a $EDIT=YES command is given, the UPDATE automatically inputs and prints out a "$" sign in column one. The user can thus enter the command keyword directly without the "$" sign. Unless there is a very heavy volume of UPDATE commands given in a session, $EDIT=YES is a command of convenience, sometimes of questionable merit. When this option is switched on, UPDATE will interpret every line as a command, because the computer already receives a "$" sign automatically as the first character. It causes a dilemma if you actually wants to insert a line. A command

$CREATE /TEXT/

will cause a line represented by TEXT to be inserted after the current line, and may be used for insertion when the $EDIT switch is on.

Example: A segment of editing involving $EDIT switch.

UPDATE Commands	Comments
>*$TYPE 2*	EDIT switch is OFF.
C Illustrative Example	
C Main Program	Two lines typed out
>*$EDIT=YES*	EDIT switch is now ON.
>*$C WRITTEN BY T. W. SZE*	Attempt to insert a line;
?>*$CR/C WRITTEN BY T. W. SZE/*	note error return
>$	"$" automatically given

C. $ECHO=YES; $ECHO=NO When this switch is turned on, an inserted line will be echo-printed on the terminal right after the insertion. This switch is very useful when used in conjunction with the case-shifting switch or the tab-setting switch.

Example: In the following example, a table is being constructed with data in columns 11-12, 21-22 and 31-32. Tabs are set at 11, 21 and 31. The $ECHO=YES switch will echo back entered data at the correct column positions.

UPDATE Commands			Comments
>$TAB=11,21,31			Set TAB.
>$ECHO=YES			Set ECHO switch.
>(Tab)23(tab)55(tab)92			Enter data.
23	55	92	Data echoed.
>(Tab)43(tab)12(tab)28			
43	12	28	
>			

D. $ERROR=YES; $ERROR=NO When this switch is turned on, an error message will be reported on the terminal when an error is committed. If the switch is turned off, only a "?" symbol is reported to indicate an error.

Example: Suppose UPDATE is editing a file that contains 5 lines. The contents of these 5 lines are 11, 22, 33, 44, and 55 respectively for each line starting at column-1. The pointer is now at line No.3. Observe the errors made in the editing session and error message received:

UPDATE Commands	Comments
>$TYPE	... Display line 3.
33	
>$CH/11/66/	... To make a change.
?>$ERROR=YES	... "?" symbol returned. Turn on
>$CH/11/66/	error message and try again.
?Sequence not in current line	... meaning can't find a match
>$CH/33/66/	... Try again.
66	... Change verified
>$TO/88/	... Move pointer.
?Reached last line of text	... Can't find /88/.
>$WHERE	... Check line number.
[5]	
>$DUNE	... Close the editing.
?Illegal command or structure	... Incorrect spelling
>$DONE	... Try again.
Catalog name=>DATA.DAT	... Name DATA.DAT given
File .... Already existing file	... Duplicate name error
Catalog name=>DATAX.DAT	... Give another name.
1 blocks written on DATAX.DAT[115103,320571]	

E. $GAG=YES; $GAG=NO After the commands TO, TRAVEL, CHANGE, SUBSTITUTE are executed, the terminal automatically prints out the new current line. While it serves as a convenience, it may become a nuisance if there is too much output. To suppress the printing, use $GAG=YES command, and the current line can only be printed out by an explicit $TYPE command. This function can be cancelled by a $GAG=NO command.

Example: To suppress unwanted verification printout:

```
>$SUBSTITUTE/ITEM1/ITEM2/
   volume
     of
 verification
 printout
>
```

```
>$GAG=YES
>$SUBSTITUTE/ITEM1/ITEM2/
>
```

F. $LINE=YES; $LINE=NO When a $LINE=YES command is given, the line number will be displayed along with the line text in all terminal displays.

Example: Observe the difference before and after the LINE switch is turned on:

UPDATE Commands	Comments
`>$AT1; TYPE 3`	Type out first 3 lines, no line numbers.
`11`	
`22`	
`33`	
`>$LINE=YES`	
`>$AT1; TYPE 3`	Type out first 3 lines with their respective line numbers.
`[1] 11`	
`[2] 22`	
`[3] 33`	
`>`	

G. $UPPER=YES, $UPPER=NO; $LOWER=YES, $LOW=NO
Many older or inexpensive terminals are built without capability of entering or outputing lower-case letters. Hence, it is often desirable to enter upper-case letters but store them as lower-cases. Since both $UPPER and $LOWER swtiches affect the cases, the aggregate effect depends on the combination of the two switches:

UPPER	LOWER	Aggregate Effect
NO	NO	Store as entered
YES	NO	Store as upper cases
NO	YES	Store as lower cases
YES	YES	store as entered

The readers are reminded that all switches ar originally at OFF or NO states. However, if there are large volume of text data containing both upper and lower cases, such as a report or a thesis, it is not practical to use this switch if one has a upper-case-only terminal. For such needs, the users are referred to the utility program RUNOFF (See Chapter 7) which contains many word-processing procedure including case-control.

(2) Format of functions switched ON temporarily by UPDATE

Frequently, it is desirable to switch certain functions ON only momentarily for the duration of one command. While the switch can always be turned on or off by commands, it will be convenient to construct a "spring-return" switch which will automatically be turned back to OFF after the command is executed. UPDATE provides this convenience by a command format in parenthesis:

```
$COMMAND (SWITCH FUNCTION) Argument
$COMMAND (FUNCTION-1) (FUNCTION-2) Argument
$COMMAND (FUNCTION-1, FUNCTION-2) Argument
```

Examples: The following examples show equivalent commands:

Equivalent		Equivalent	
$LINE=YES $TYPE 3 $LINE=NO	$TYPE(LINE)3	$ERROR=YES $TO/XYZ/ $ERROR=NO	$TO(ERROR)/XYZ/

Equivalent	
$GA=YES $ER=YES $SU/XX/YY/ $GAG=NO $ERROR=NO	$SU(GA,ER)/XX/YY/

2.14 Editing Function Value-Setting Commands

In this group of commands, the common format is:

```
$COMMAND = n
```

where "n" is an integer. The meaning of "n" is defined for each function, and they are presented as follows:

A. $FACTOR=N This command will modify the size of the memory "window". Normally, there is no need to adjust the window. Only when editing a very large file, there may be justification to adjust the window to a larger size in order to reduce the overhead file-copying when the pointer is backed outside the current window.

This was explained in a previous section in reference to Figure 2.1. When $FACTOR is given without an argument, it is an inquiry for the size of memory assigned behind the current line. A number typed out on the terminal indicates the size in number of lines.

B. $LENGTH=N, and $SIZE=N Either of the commands will set the length of each line to N characters long. If the text in a line is less than N-character long, spaces after the last character are padded with blanks. If the text in a line is more than N-character long, the extra characters are simply truncated and removed.

Complications arise when there are tabs characters in a line. Although each tab character counts only as one character in the line text, its effect is equal to multiple and variable blanks when it is translated. Therefore, if the $LENGTH or $SIZE command is used to prepare a fixed-length record file, it is desirable to let UPDATE translate tabs into blanks by $TAB command, so that a correct number of characters will be counted. The main usefulness of this command is to construct a data file in which the record size is uniform for every record. If $LENGTH command is applied without any argument, it becomes an inquiry about the length of the current line. The computer will respond with a number which is equal to the number of characters (including blanks) in the current line.

C. $SAVE=N This is a safety feature that can be very useful in long editing sessions. If the editing session in progress and the System must be re-initialized due to crash, broken communication linkage or any other emergency situation, all fruits of labor during that editing session are lost. Or, when the connect-time of the terminal expires, there will be no allowance for the user to finish or to close the editing, and he is forced to sign off immediately. The result of the current editing is also lost. If such contigency may be likely, it is prudent for a user to apply a command $SAVE=N. The following will then be accomplished:

When a command such as $SAVE=15 is issued, the output file E2 will be periodically closed, stored, and reopened to continue, for every 15 lines output into the E2. Thus, in case of a system failure, the user will in his disk a TMP file named E200xx.TMP that contains the status of last save. This would cut the loss of information to a small amount. The exact name of the TMP file is reported on the user's terminal. Should the editing goes to the completion successfully, that TMP file is deleted automatically. The disadvantage of such safety measure is that it significantly slows down the editing operation because of the extra file operations the computer is required to do every N lines.

D. $TAB=N1,n2,... When the UPDATE is first called, the tab settings are at the system default positions, namely at columns 9, 17, 25, etc (every 8 columns). To reset the values of tab setting to a different set, use the command $TAB=n1,n2,... where "n1", "n2", etc., are the new tab settings. When a tab key is subsequently entered, it will be translated into multiple blanks, the number of which depends on where the tab key is entered on the line. Since tab key often causes problem in the count of characters in a line, especially in the case of a fixed record-length file, it is useful to

use this command even though the tab settings may be the same as the system default.

The chief usefulness of this command is to prepare tables with fixed columns, or to prepare a fixed column data file.

Example: Construct a roster of names with last names starting on column-5 and initials starting on column-25:

```
>$TAB=5,25
>(T)Doe(T)JD                    (T) =Tab key
>(T)Jones(T)MS
>(T)Li  (T)JG
>(T)Kong(T)KK
>(T)Modzelewski(T)SW
>(T)Smith (T)YT
>$AT1; TYPE 6
    Doe                 JD
    Jones               MS
    Li                  JG
    Kong                KK
    Modzelewski         SW
    Smith               YT
>
```

Example: Prepare a data file that has a FORTRAN format of 2(7X,I3).

```
>$TAB=8,18
>(T)238(T) 23                (T)=TAB KEY
>(T) 12(T)856
>(T) 44(T)433
...
>$AT1; TYPE 3
       234        23
        12       856
        44       433
>
```

2.15 Miscellaneous Editing Commands

(1) Commands regarding to current line position

A. $WHERE and $LINE Either of these two commands will cause the absolute line number reported on the terminal.

B. $LENGTH This command will cause the length of the current line in number of characters reported on the terminal. Also refer to the command $LENGTH=n command. Note that $LENGTH is to inquire about the length, while $LENGTH=n is to set the length.

C. $POSITION /TEXT1/TEXT2/... This command will type out the positions (column numbers) of the first character of each of the string TEXT1,TEXT2,... in the current line.

(2) Insertion Commands While UPDATE will accept any input line without the "$" sign in column 1 to be an inserted line, there are occasions insertions may be made easier by the following commands:

A. OVERLAY /TEXT/K, or $K /TEXT/ This command will place a string of characters "TEXT" in the current line beginning at the Kth column and replacing whatever was there before.

Example: Observe the effect of a command $4/ABCD/:

Before	After
1234567890	123ABCD890

B. $PLACE/TEXT/K This command will insert the string "TEXT" in the current line starting at the Kth column. Unlike the $OVERLAY command, the displaced characters do not disappear; they are merely pushed back to the right to make room for the inserted string.

Example: Observe the effect of acommand $PLACE/ABCD/4 and compare it with that of the previous example:

Before	After
1234567890	123ABCD4567890

C. $REPLACE N When this command is given, the specified number of line in the file beggining with the current line is deleted, and the same number of lines subsequently typed on the terminal will take their palces. This command is equivalent to a compounded command of $DELETE(GAG); AT-1. The command $REPLACE is equivalent to $REPLACE 1.

Example: Observe the difference between REPLACE and DELETE commands:

$DELETE command	$REPLACE command
>$AT1; TYPE 5	*>$AT1; TYPE 5*
11	11
22	22
33	33
44	44
55	55
>$AT3; DE(GAG)	*>$AT3; RE*
>XX	*>XX*
>$AT1; TYPE 5	*>$AT1; TYPE 5*
11	11
22	22
44	XX
XX	44
55	55

(3) Length-manipulating commands

The end of a line is indicated by a carriage-return character. The number of characters between two carriage return characters, not counting the carriage return characters themselves, is the length of a line. Therefore, by adding a carriage return some place in a line, it may be broken into two lines. Conversely, if the carriage return at the end of a line is removed, that line is joined with the next. In manipulating the length in this manner, caution should be exercised regarding the blanks at the end of a line. Normally, when there are trailing blanks in a line, UPDATE simply ignores them in order to conserve storage spaces. Thus, the number of blanks at the joint should be carefully observed, otherwise the space at the "seam" will be in error. The associated commands are now discussed next.

A. JOIN command This command will remove the carriage return character at the end of the current line, thereby join it with the next line. Because all trailing blanks are deleted, any blanks required at the seam must be provided by the leading blanks of the second line in the joining process.

B. $BREAK command This command will insert a carriage return character into the current line, thereby braking it into two lines. It has two formats:

```
$BREAK N

$BREAK /TEXT/
```

Both "N" and "TEXT" indicate the end of the first line after the break. Thus, the second line after the break will begin with the old (N+1)th column as its first column, or the column immediately after the string "TEXT" as its first column.

Examples: Observe the effect of $JOIN and $BREAK. Pay attention specially to the "seam", before and after the operation.

```
>$TYPE 3
11
 22
33
>$JOIN
11 22
>$JOIN
11 2233
>
```

```
>$TYPE 2
12345 67890
>$BREAK/5/
12345
 67890
>
```

```
>$TYPE 2
12345 67890
>$BREAK/5 /
12345
67890
>
```

SELECTED ADVANCED TOPICS IN UPDATE

The materials presented in the PRIMER (pp.37-42) are for the beginning users. The materials presented in the COMMAMDS and PROCEDURES (pp.43-62) are for the average users. The combined materials should be more than adequate for most editing jobs. Occasionally, there may be special and frequent needs for very sophisticated editing and therefore a more complicated set of commands may be useful. However, unless you have special needs that require the commands in the following sections, your time may be better invested by thoroughly familiarizing yourself with the basic material and then going directly to the SUMMARY sections (page 72). It should be noted that the objectives accomplishable by the complex commands can also be accomplished by simpler commands in more steps. Or, it may require getting on and off from UPDATE several times.

Three topics will be presented: auxiliary files, conditional commands, and editing programs.

2.16 Preparation and Use of Auxiliary Files

Sometimes, it is desirable to construct an auxiliary file which contains a selected excerpts from a main file. Or, in creating a new file, certain of its lines may be contributed by an already established file. Using only commands presented so far, one can take the established file, delete all unwanted lines, and the result would be an excerpt. In this section, some special UPDATE commands are presented that will facilitate such a task.

(1) Auxiliary output file preparation

A main file is already in existence and has been called by UPDATE. It is required now to make one or more auxiliary files which contain excerpts from the main file. Three commands are provided for this purpose:

A. $ONTO command This command will open an auxiliary file in the disk into which excerpts of the main file will be transferred. The opened file will be given a filename in the ONTO command format:

$ONTO = standard file specification

where the standard file specification will contain a name and an extension.

B. $PUT N command This command will transfer N lines, beginning with the current line, from the main file to the auxiliary file opened by a previous $ONTO command. If N is omitted in the command, it is equivalent to $PUT 1. Caution: After the lines are transferred to the auxiliary file, those lines are no longer in the main file. If the editing session is allowed to end with a normal $END or $FINISH command, the new main file will be the old file minus those exerpts taken out. If you do not wish to disturb the old main file, you must not let the editing session come to a normal end. As soon as the auxiliary file preparation is completed and closed, apply CTRL-C to abort the editing job.

C. $CLOSE command — The command $ONTO opens an auxiliary file E3 as a working file; $CLOSE command closes it and stores it away in the disk.

Examples: Two auxiliary files X.DAT and Y.DAT are prepared composed of excerpts from a main file SAMPLE.DAT. Observe the sequence of editing commands and the "Before" and the "After" conditions of the files:

(BEFORE) SAMPLE.DAT	Editing Commands	(AFTER) SAMPLE.DAT	X.DAT	Y.DAT
11	*.UPDATE SAMPLE.DAT*	44	22	11
22	11	77	33	
33	*>$AT2; ONTO=X.DAT*		55	
44	*>$PU2; AT+1; PU2; CLOSE*		66	
55	*>$AT1; ONTO=Y.DAT*			
66	*>$PU1; CLOSE; END*			
77				

(BEFORE) SAMPLE.DAT	Editing Commands	(AFTER) SAMPLE.DAT	X.DAT	Y.DAT
11	*.UPDATE SAMPLE.DAT*	11	22	11
22	11	22	33	
33	*>$AT2; ONTO=X.DAT*	33	55	
44	*>$PU2; AT+1; PU2; CLOSE*	44	66	
55	*>$AT1; ONTO=Y.DAT*	55		
66	*>$PU1; CLOSE*	66		
77	*>↑C*	77		
	.			

(2) Auxiliary input file operation

Often, the input insertion to a file is preferred to be lines from an existing file if it is already available. Presumably, that file has been checked out already and it is not only convenient to copy those lines but also reduces the chances of error.

A. $FROM command — While the $ONTO command specifies a destination auxiliary file, the $FROM command specifies a source file. Its command format is similar to that of the $ONTO command:

$FROM = standard file specification

If this file resides in another user's disk area, his PPN should be a part of the file specification, such as NAME.EXT[m,n].

B. $ADVANCE command — When the $FROM command is first applied, its pointer is positioned at line 1. The $ADVANCE n command is used to position the pointer in the auxiliary file specified by the $FROM command. Although n is an unsigned integer, it is interpreted as a relative line number.

C. $GET command Once the pointer of the auxiliary file is positioned correctly in the auxiliary file, a command of $GET n will transfer n lines, starting with the current line, from the auxiliary file to the main file. After the transfer, the lines in the auxiliary file are not erased.

Sometimes, excerpts are taken from several auxiliary files. In changing from one auxiliary file to another, it is necessary to disengage the old one before engaging the new. For this reason, the command $FROM is designed to disengage automatically the old auxiliary file and engage the new file. Each time a file is engaged, the pointed will be positioned at line 1.

Examples: A file SAMPLE.DAT and two auxiliary files X.DAT and Y.DAT are all available in the disk storage. Their contents are as follows:

File Contents ---"BEFORE"

SAMPLE.DAT	X.DAT	Y.DAT
11	AA	XX
22	BB	YY
33	CC	ZZ
44	DD	UU
55	EE	VV
66		
77		

Another file Z.DAT is now prepared by inserting certain lines from X.DAT and Y.DAT into SAMPLE.DAT. This is shown below:

Editing Commands	File Contents --- "AFTER" SAMPLE.DAT	X.DAT	Y.DAT	Z.DAT
.UPDATE SAMPLE.DAT	11	AA	XX	11
11	22	BB	YY	22
>$AT2; FROM=X.DAT	33	CC	ZZ	BB
>$ADVANCE 1; GET 2	44	DD	UU	CC
>$AT/55/; FROM=Y.DAT	55	EE	VV	33
>$AD2; GET 2: DONE	66			44
	77			55
CATALOG NAME=>*Z.DAT*				ZZ
1 BLOCK WRITTEN ON Z.DAT				UU
				66
				77
EXIT				
.				

2.17 Conditional Editing Commands

The UPDATE is enhanced in capabiltiy by being able to make "decision" on which one of two alternate groups of editing commands are to be executed.

The basic structure of decision-making is as follows: First, a question is asked to which a true-false answer is stored. This is accomplished by issuing a $IF command. If the answer is affirmative, issuing a $THEN command will execute a group of "execute-if-true" eiditng commands. (If the answer is

negative, issuing a $THEN command will receive no response from them.) If the answer is negative, issuing a $ELSE command will execute a group of "execute-if-false" editing commands. (Similarly, if the answer is affirmative, issuing a $ELSE command will receive no response from them.) Such a structure is simialr to the conditional structure in many language processors, and is graphically illustrated in a flow chart as shown in Figure 2.2.

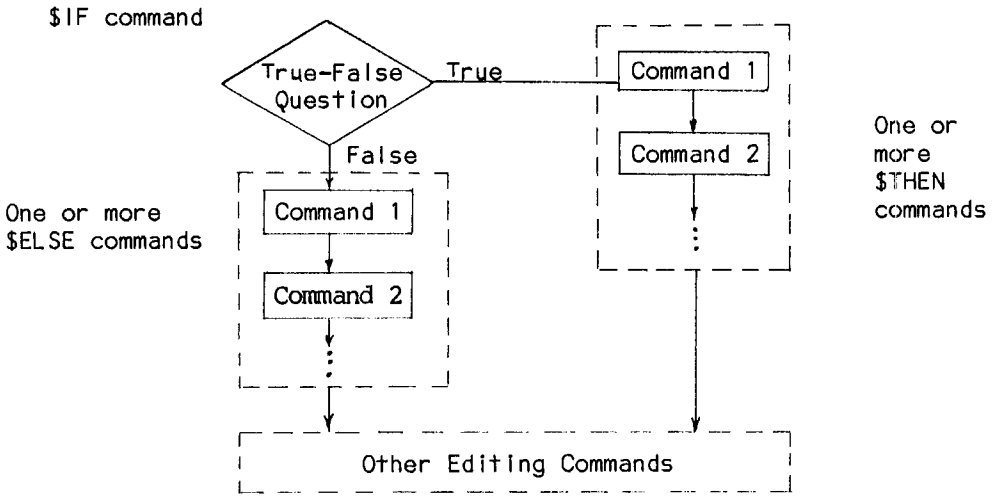


Figure 2.2 Flow Chart of Conditional Editing Commands

(1) Single conditional commands

A. $IF command As illustrated in Figure 2.2, the $IF command asks a true-false question, and its answer is stored away, setting the stage for subsequent actions of $THEN and $ELSE commands. Since the UPDATE has immediate information only on the current line, the question asked must pertain either to the current line number or to its content. Therefore, the formats of the $IF command are limited to the following:

$IF format	Question Asked
$IF /TEXT/	Is there a string "TEXT" in the current line?
$IF /TEXT/K	Is there a string "TEXT" in the current line that begins at the Kth column?
$IF $TEXT$	Ignoring blanks, tabs, and difference between upper and lower cases, is there a string "TEXT" in the current line?
$IF n	Is the current line number equal to n?

Notice that the formats of the first three are very similar to those of $TO commands.

B. $THEN and $ELSE commands The $THEN and the $ELSE commands will specify and execute the alternate sets of commands depending on the answers to a previously issued $IF command. The command format is as follows:

$THEN /command 1; command 2; .../

$ELSE /command 1; command 2; .../

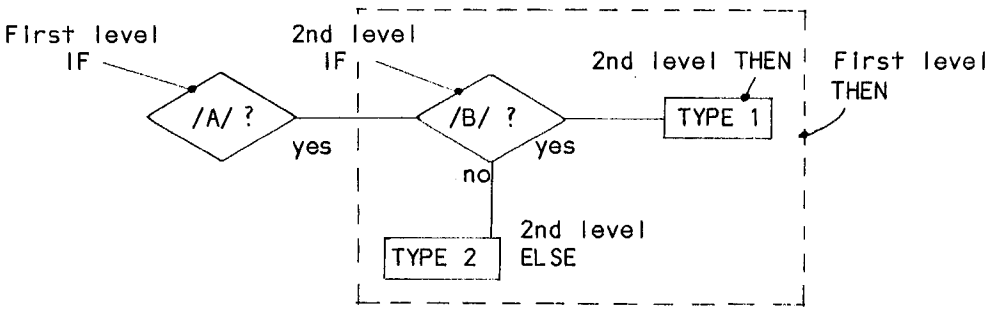
The commands between the delimiters "/" follow the rules of compounded command structure, as discussed in section 2.10.

Example: $IF/FORMAT/7; THEN/WHERE; TYPE 2/;ELSE/DELETE 2/

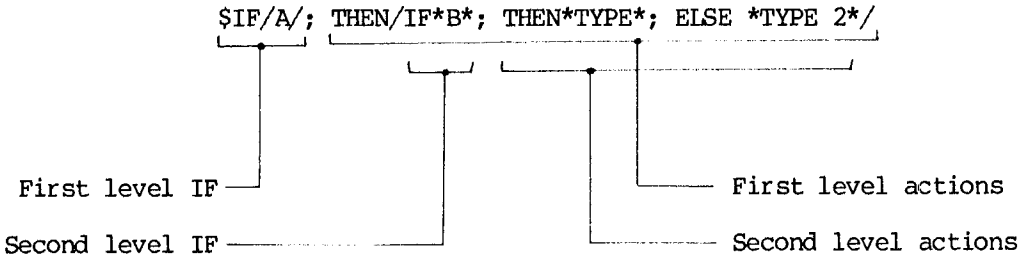
Function: Examine the current line. Does it have a string of characters "FORMAT" beginning at the 7th column? If yes, print out the line number and type two lines. If no, delete 2 lines.

(2) Nested conditional commands

Each of $THEN and $ELSE command contains a set of embedded commands in the compounded form. If the embedded commands contain another IF command, we now have a nested structure. The following flow chart shows a typical example of nested command structure:



The function of this flow chart is as follows: First examine the current line to see if there is a character "A". If no, do nothing. If yes, then examine if there is also a character "B" in the current line. If yes, type one line; if no, type 2 lines. These functions may be accomplished by the following nested command:

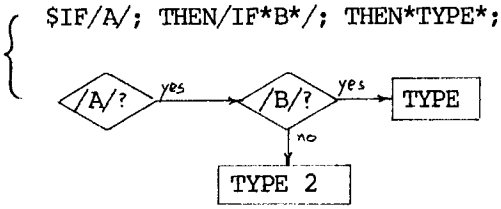


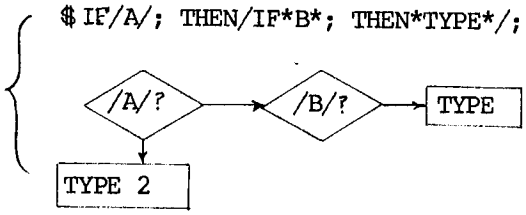
In using a nested conditional structure, one should be cautioned about the following:

A. The main advantage of the nested conditional structure is to compress many editing commands into a single compounded one, so that its execution will be more efficient. The UPDATE allows a maximum of ten nesting levels. The main drawback is that constructing a nested structure is a very error-prone process. Furthermore, more levels it goes into, less man-machine interaction is available to the user. Therefore, even though the UPDATE is machine-effective for high-level nesting, it is a poor practice for a user to go much beyond the second level. Otherwise, an editing session will be very likely degenerated into a debugging session for editing commands. An exception to this advice is when one has some nested commands that will be used repeatedly by the user or others. In such a case, it may be justifiable to spend a lot of time to debug it and store it for later repeated use.

B. In addition to the logic involved, the most likely source of error in a nest construction is the choice of delimiters. Normally, any special symbol pair may be used as delimiters (or as "quotation marks"). Since a nested structure is basically a compounded structure, the semicolons ";" must be reserved to separate the commands. Moreover, there should be no ambiguity between the command delimiters of IF, THEN, ELSE at different levels. Therefore, it is advisable to assign an unique delimiter symbol for each level. See the following illustrative examples:

Example: Consider the following nested commands with their respecitve interpretation of functions by means of flow chart:





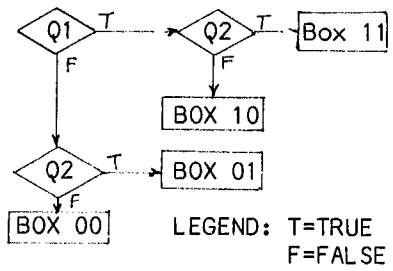
IF/A/; THEN/IF/B/; THEN/TYPE//; ELSE /TYPE 2/
Incorrect use of delimiters!

C. Several nested commands may be compounded together to form a compounded nested command. In doing so, one must be careful about the correct placement of the THEN, ELSE commands. Each time when a first-level IF command is executed, its true-false answer replaces that obtained in a previous IF command. The same goes for the subsequent level IF commands. Thus in the above example, the actions of both statements may be combined by this statement:

$IF/A/; THEN/IF*B*; TH*TY*;EL*TY2*/;TH/IF*B*;TH*TY*/;EL*TY2*

(3) Conditional with logic connectives

Consider the following fully-developed two-level nested structure:



Logic Connective Between Q1 & Q2	Accomplished by Placing the same actions in Box 00	01	10	11
AND				X
OR		X	X	X
NAND	X	X	X	
NOR	X			
XOR		X	X	

By placing identical editing actions in the appropriate boxes as shown in the accompanying table, a logical connective between the answers to Q1 and Q2 may be accomplished. For example, if one wants to type the line if either character "A" or character "B" (or both) is present, he should place the TYPE command in boxes 01, 10 and 11. The result is the following command:

$IF/A/; THEN/IF*B*; THEN *TYPE*; ELSE *TYPE*/; ELSE/IF*B*; THEN *TYPE*/

Actually, one can see that there is an INCLUSIVE OR, or logical union relation existed in this case. The UPDATE processor has simplified the matter by providing five commands specifying logic connectives: AND, OR, NAND, NOR and

XOR. They are respectively for logic intersection, logic union, negation of AND, negation of OR, and EXCLUSIVE OR. Thus, the above example can now be written as:

```
$IF/A/; OR/B/; THEN /TYPE/
```

There is one important caution in using the logic connective commands. Unlike the two-level nested commands, the second-level question does not establish an independent answer (True-False) but modifies the first one. Therefore, if the first-level question alone is to initiate some THEN or ELSE action, it better be done before the answer is changed by the logical connective commands. Observe the difference between the following two editing commands:

```
$IF/A/; ELSE /TYPE/; OR /B/; THEN /DELETE/

$IF/A/; OR /B/; ELSE/TYPE/; THEN /DELETE/
```

The difference would be the execution of ELSE/TYPE/ segment.

2.18 Editing Programs

In the UPDATE processor, the compounded command structure enables a series of command executions in one pass. The TRAVEL, GO, and STOP commands result in the looping capability. The conditional command group IF, THEN, ELSE, and logic connectives yield the decision-making capability. Combining all of these, one has the makings of a complete stored editing program. However, it is not always desirable to construct editing programs for one-shot usage as they are very wasteful of user resources. Moreover, accuracy of editing requires a high degree of user-machine interaction which a complete editing program will deprive. Therefore, construction of editing programs should be limited to applications of wide and frequent usages.

Two such program are given as illustrative examples:

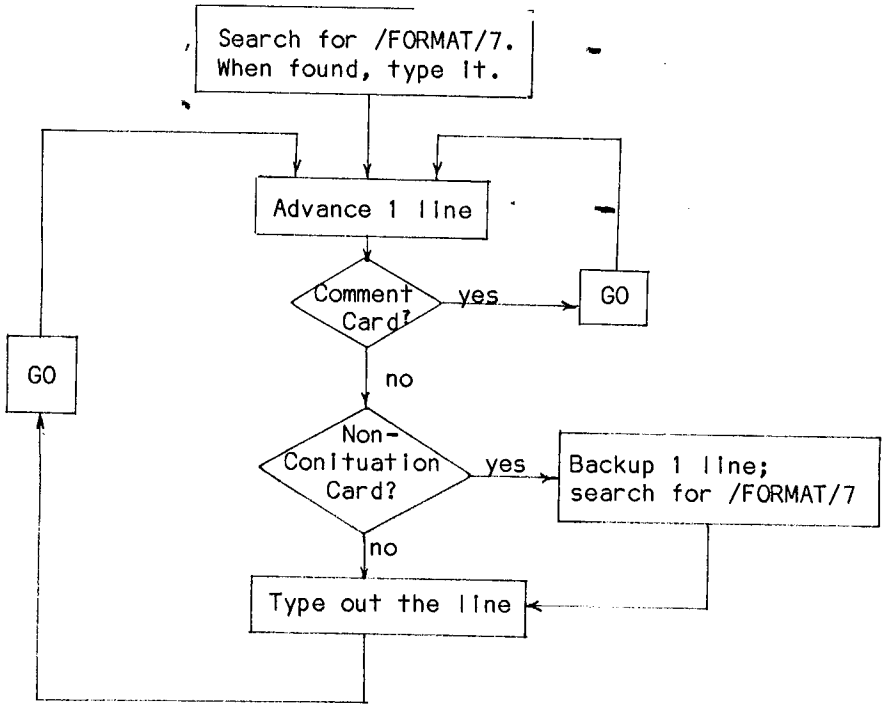
Example: Given a FORTRAN program, design an editing program that will print out all FORMAT statements. Assume all FORMAT statements have the keyword "FORMAT" beginning at column 7, but some of the FORMAT statement may have continuation cards.

The logic of the program may be described by means of a flow chart shown on the next page.

In the compounded structure form, the resulting program (one line) is as follows:

```
$GAG=YES;AT/FORMAT/7;TYPE;TR+1; IF/C/1;OR/*/1; THEN/GO/; IF/ /6;
    THEN/AT-1; AT'FORMAT'7/; TYPE; GO
```

After this command is completed, all FORMAT statements will have been typed out on the user's terminal. An error report sign "?" will also be typed out, because when the search reaches the end of file, the $TR+1 command will still attempt to advance 1 line. If the above program has PUT commands instead of TYPE commands (with ONTO command issued previously), this program would have prepared an auxiliary file that contains all FORMAT statements in the FORTRAN program.



Example: The equivalence between the 026 and the 029 key punch code is shown below:

026 Punch	029 Punch
#	=
&	+
@	'
%	(
<	)

Other characters have the same punch codes.

```
$TR+1;IF/ /;TH/AL. .;AT-1/;OR/&/;TH/AL.&.+.;AT-1/OR/@/;TH/AL.@.'.;
    AT-1.;OR/%/;TH/AL.%.(.;AT-1/;OR/</;TH/AL.<.).;AT-1/;GO
```

Since a single-line compounded command is limited to a maximum of 150 characters, two-letter abbreviations are used for all UPDATE keywords in the above command.

A SUMMARY OF FILE MANAGEMENT BY UPDATE

2.19 File management Tasks

(1) To create a new file from a terminal

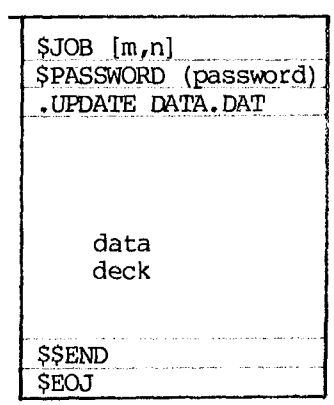
When UPDATE receives the input file name, the disk storage directory is searched. when the file is found, the file is loaded into the input working file.

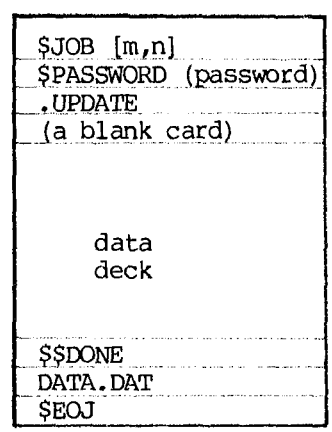
However, if the file name supplied by the user is null (represented by a carriage return and nothing else), or if the file does not exist for the name given, the input working file is entirely blank. Thus, the only information that may go to the output working file would be from the terminal or from other stored files by insertion mode. In this way, an entirely new file may be created from the user's terminal and stored in the disk.

(2) To create a new file by batch

The effectiveness of UPDATE to do editing job is mainly because its man-machine interaction. Therefore, UPDATE normally is not suitable for BATCH jobs. However, if the source materials are in punched card form, a file may be created from these cards by UPDATE submitted in BATCH.

Suppose we wish to store a deck of data cards in disk and will name the file as DATA.DAT. First, a batch deck of cards is prepared that contains the following. Either will do:





In the above deck setup, the single-$ cards are BATCH commands, and those double-$ cards are UPDATE commands read by BATCH. For more details on Multiprogram Batch, see Chapter 9.

After the cards are prepared, read the cards in at a RJE card reader. The job will be executed by the computer, and the file DATA.DAT is thus created from the cards. For card input used in this way, the same precaution should be exercised that there should be no "$" character in the first column in the data card deck.

(3) To copy a file

A file may be duplicated and stored in the user's disk area by using the UPDATE in the folowing way:

```
.UPDATE NAME.EXT[115103,320571]
(UPDATE prints out the first line of NAME.EXT)
>$DONE
CATALOG NAME=> NEW.EXT
```

This is equivalnet to a monitor command of:

```
.COPY NEW.EXT = NAME.EXT[115103,320571]
```

(4) To merge several files into one

For example, if three files D1.FOR, D2.FOR and D3.FOR are to be merged into one DX.FOR, it can be accomplished in the follwoing way:

```
.UPDATE D1.FOR
(UPDATE prints out the first line of D1.FOR)
>$AT $
>$INPUT= D2.FOR
>$INPUT= D3.FOR
>$DONE
CATALOG NAME=>DX.FOR
```

This is equivalent to applying the monitor command:

```
.COPY DX.FOR = D1.FOR,D2.FOR,D3.FOR
```

(5) To prepare an auxiliary file from a source file

The following is an example where an auxiliary file FORMAT.FOR is prepared by extracting all FORMAT statements (some of which may be multiple-line statements) from the FORTRAN file SAMPLE.FOR. Assume that all keyword FORMAT of the FORMAT statements starts at the 7th column.

```
.UDPATE SAMPLE.FOR
(UPDATE prints out the first line of SAMPLE.FOR)
>$BEFORE
>Δ                                                    (Δ=blank)
>$ONTO=FORMAT.FOR
>$GAG=YES;AT/FORMAT/7;PU;TR+1;IF/C/1;OR/*/1;TH/GO/;IF/ /6      one
    TH/AT-1;at'FORMAT'7/;PU;GO                                 line
?>$CLOSE
>↑C                    ?=error indication when reaching the end
                         of file and still wanting to "GO"
```

The logic of the long command line in this example was discussed in Section 2.18.

2.20 Examples of File Editing

Two examples of editing a complete file will be given using the UPDATE editor. The first one consists of entirely text eidting, while the second one is a stored program in FORTRAN. The following points will be helpful:

(1) A careful proof-reading of the old text is essential. It is also desirable to do the proof-reading "off-line" to conserve valuable terminal time.

(2) To increase the speed and efficiency of editing (and therefore to reduce time and cost), all corrections should be marked on the listing, together with their line numbers if appropriate.

(3) Moving from one record to another, the normal operation of the UPDATE editior is to go forward. In fact, backing up the pointer to some previous line may sometimes be costly because it will involve file re-writing and re-reading. Therefore, backing up is generally an inefficient process and should be used sparingly in view of processor efficiency. On the other hand, since deletions and insertions of lines during editing will change the line numbers of all lines of text beyond the pointer, it will be progressively difficult to locate the desired line by absolute line numbers. For this reason, in editing the text by its absolute line numbers, it is sometimes desirable that the editing be done in the reverse direction, starting from the end of the file and working backwards toward the front. In this manner, the deletion and insertion of lines will not affect the line numbers of the portions of the file not yet edited. Here we are trading off machine and processor efficiency for user convenience. This process is desirable only if the user has made preparations as outlined in (1) and (2). To improve processor efficiency, he can also readjust and enlarge the window by the $FACTOR command.

(4) Only the first two letters of any UPDATE command word need be given. Incorrect spelling of command is tolerated as long as the first two letters are spelled correctly.

Example 1: To edit the text taken from the School of Engineering Bulletin, University of Pittsburgh. The draft of text on disk file TEXT.EDT along with the revisions on the draft appears as follows:

```
THE MICHAREL L. BENEDUM HALL OF ENGINEERING

     STDUENTS ENROLED IN THE SCHOOOL OF ENGINEERING, UNIVER TY OF
PITTSBURGH, RECEIVE THEIR EDUCATION IN ONE OF THE COUNTRY'S MOST
MODERNAND BEST EQUIPPED ENGINEERING BUILDINGS, THE MICHARL L. BENNEDUM
HALL OF ENGINEERING.  THE BUILDING COMPLEXX IS NAMED IN HONOR OF
MICHAEL L. ENEDUM, A PIONEER IN THE OIL INDUSTRY AND CO-FOUNDER OF
MICHAEL L. ENEDUM, A PIONEER IN THE OIL INDUSTRY AND CO-FOUNDER OF
THE BENEDUM TREE OIL COMPNAY.  A GRANT FROM THE CLAUDE WORTHINGTON
FOUNDATION ENABLED THE UNIVERSITY TO PURCHASE ON WHICH THE ENGINEERS
COMPLEX IS BUILT.
```

The following is a printout of the editing session:

```
.UPDATE TEXT.EDIT
THE MICHAREL L. BENEDUM HALL OF ENGINEERING
>$CH/T/            T/R//
          THE MICHAEL L. BENEDUM HALL OF ENGINEERING
>$AT+2;CH/L/LL/OOO/OO/R TY/RSITY/
     STUDENTS ENROLLED IN THE SCHOOL OF ENGINEERING, UNIVERSITY OF
>$AT+2;CH/AND/ AND/ARL/AEL/NN/N/
MODERN AND BEST EQUIPPED ENGINEERING BUILDINGS, THE MICHAEL L. BENEDUM
>$AT+1;CH/X//
HALL OF ENGINEERING.  THE BUILDING COMPLEX IS NAMED IN HONOR OF
>$AT+1;CH/EN/BEN/
MICHAEL L. BENEDUM, A PIONEER IN THE OIL INDUSTRY AND CO-FOUNDER OF
>$AT+1; DELETE
THE BENEDUM TREES OIL COMPANY.  A GRANT FROM THE CLAUDE WORTHINGTON
>$AT+1;CH/CHASE/CHASE THE LAND/ENGINEERS//
FOUNDATION ENABLED THE UNIVERSITY TO PURCHASE THE LAND ON WHICH THE
>$AT+1; PLACE/ENGINEERING /1
ENGINEERING COMPLEX IS BUILT.
>$END
>
1 Blocks written on TEXT.EDT[33,33]

EXIT
```

The edited file is shown below:

```
                THE MICHAEL L. BENEDUM HALL OF ENGINEERING

     STUDENTS ENROLLED IN THE SCHOOL OF ENGINEERING, UNIVERSITY OF
  PITTSBURGH, RECEIVE THEIR EDUCATION IN ONE OF THE COUNTRY'S MOST
  MODERN AND BEST EQUIPPED ENGINEERING BUILDING, THE MICHAEL L. BENEDUM
  HALL OF ENGINEERING.  THE BULDING COMPLEX IS NAMED IN HONOR OF
  MICHAEL L. BENEDUM, A PIONEER IN THE OIL INDUSTRY AND CO-FOUNDER OF
  THE BENEDUM TREES OIL COMPANY.  A GRANT FROM THE CLAUDE WORTHINGTON
  FOUNDATION ENABLED THE UNIVERSITY TO PURCHASE THE LAND ON WHICH THE
  ENGINEERING COMPLEX IS BUILT.
```

Example: To edit a stored FORTRAN program. It is suggested that the readers follow the running comments marked on the printout.

```
.UPDATE SAMPLE.FOR
[CREATE NEW FILE]
>C     SAMPLE PROBLEM FOR THE TIME-SHARING NOTES
>      READ(5,10)A,B,C,D,X1
>   10 FORMAT(F20.7)
>    1 X2=X1-(A*X1**3+B*X1**2+C*X1+D)/(3,*A*X1**2+2.*X1+D)
>     WRITE(6,10)X2
>      IF(ABS((X1-X2)/X2-.001)3,3,2
>    2 X1=X2
>      GO TO 10
>    3 WRITE(6,11)X2
>   11 FORMAT(/' THE REAL ROOT = ',F20.7)
>      STOP
>      END
>$wh
12
```

```
>$AT1; TYPE 12
C     SAMPLE PROBLEM FOR THE TIME-SHARING NOTES
      READ(5,10)A,B,C,D,X1
   10 FORMAT(F20.7)
    1 X2=X1-(A*X1**3+B*X1**2+C*X1+D)(3,*A*X1**2+2.*X1+D)
      WRITE(6,10)X2
      IF(ABS((X1-x2)/X2-.001)3,3,2
    2 X1=SX
      GO TO 10
    3 WRITE(6,11)X2
   11 FORMAT(/' THE REAL ROOT = ',F20.7)
      STOP
      END
>$TO4
    1 X2=X1-(A*X1**3+B*X1**2+C*X1+D)(3,*A*X1**2+2.*X1+D)
>$CHANGE $D)(3,$D)/3.$2.$2.*B$D$C$
    1 X2=X1-(A*X1**3+B*X1**2+C*X1+D)/(3.*A*X1**2+2.*B*X1+C)
>$AT+1; CH/W/ W/
      WRITE(6,10)X2
>$AT+1; CH/X2-/X2)-/
      IF(ABS((X1-X2)/X2-.001)3,3,2
>$AT+1; CH/SX/X2/
    2 X1=X2
>$AT+1; CH/0//
      GO TO 1
>$END
1 Blocks written on SAMPLE.FOR[33,33]

EXIT

.TYPE SAMPLE.FOR
C     SAMPLE PROBLEM FOR THE TIME-SHARING NOTES
      READ(5,10)A,B,C,D,X1
   10 FORMAT(F20.7)
    1 X2=X1-(A*X1**3+B*X1**2+C*X1+D)/(3.*A*X1**2+2.*B*X1+C)
      WRITE(6,10)X2
      IF(ABS((X1-X2)/X2-.001)3,3,2
    2 X1=X2
      GO TO 1
    3 WRITE(6,11)X2
   11 FORMAT(/' THE REAL ROOT = ',F20.7)
      STOP
      END
```

EXERCISES

1. (a) Enter the following FORTRAN program in your disk by using UPDATE and name the file as PROB1.FOR:

```
C    PROBLEM NO. 1
       DIMENSION K(10)
       DO 5 I=1,10
5      K(I)=I**2
       WRITE(6,10)((I,K(I)),I=1,10)
10     FORMAT(2I7)
       STOP
       END
```

Purposely make some errors in your typing. For example, omit some commas and misspell a few words.

(b) When you are back at the monitor level, execute the incorrect program by a command:

.EXECUTE PROB1.FOR

and observe the proceedings.

(c) Make appropriate corrections, and execute again. Repeat until you get the program letter perfect.

2. What would each of the following UPDATE fragments do?

(a)
```
$AT 1
$SUB/XX/YY/
```

(b)
```
$AT 1
$TR/XX/
$CH/XX/YY/
$AT -1
$GO
```

(c)
```
$AT 1
$TR/XX/
$CH/XX/YY/
$GO
```

(d)
```
$SUB/READ(5,/READ(1,/
```

(e)
```
$TR+1;IF/READ/7;THEN/TYPE(LI)/;GO
```

(f)
```
$ONTO= READ.FOR
$TR+1; IF/READ/7; THEN/PUT/; GO
$CLOSE
```

3. Three different compounded MOVE commands are given:

```
$MOVE; HERE

$MOVE; AT-1; HERE

$MOVE; AT$; HERE
```

For each of these three commands, answer the following questions:

(a) Where is the line moved to?

(b) Where will be the pointer after the move?

4. Verify your answers to problem 3 by actually setting up a file, observing the BEFORE and AFTER of each of the above three commands.

5. Enter Lincoln's Gettisburg Address as a file and name it as ABE.DOC. Correct any error in the file.

6. For each line of ABE.DOC prepared in problem 5, edit the text so that the following results are obtained:

   a. Set the left margin at column 1; the right margin at column 45.

   b. The first line of a new paragraph is indented 5 spaces.

   c. Right justify by adding spaces between words.

   d. Space all punctuations so that there is one space after each comma or semicolon, and 3 spaces after each period.

7. After copying the file SYS:NEWS (see Exercise(3), Chapter 1) into your own disk area, use UPDATE and with one compounded instruction, search and type out all first lines of news items that were dated in 1980.

8. The instructor will furnish for this exercise a long FORTRAN program that contains many FORMAT, READ, WRITE and CALL statements. Prepare four auxiliary files that will contain the following information:

   (a) File FORMAT.FOR: a record of all FORMAT statements
   (b) File READ.FOR: a record of all READ statements
   (c) File WRITE.FOR: a record of all WRITE statements
   (d) File SUBR.FOR: a record of all subroutine CALL statements

   For a simple case, make the following assumptions:

   (1) All characters are upper cases.
   (2) All statement keywords begin on column 7.
   (3) No continuation statement.

9. For a more challeging case of problem 8, make the following assumptions and then prepare the required auxiliary files:

   (1) Mixed upper and lower cases in the FORTRAN program file.
   (2) A statement may begin anywhere between column-7 and column-72.
   (3) Some of the READ, WRITE or CALL statements may be imbedded in an IF statement, e.g., IF(I.EQ.1)READ(5,56)X
   (4) There may be continuation statements.

   You may modify this problem and generate a problem a varying degree of difficulty by selecting one or more of these assumptions.

10. The source program in FORTRAN-10 on DEC System-10 allows a special use of the tab key (or the CTRL-I character) to skip all or part of the label field. The purpose is to use a tab-character (1 character) to replace multiple spaces (multiple characters) to save storage space. Rules of interpreting a FORTRAN-10 statement using a tab in the initial field are as follows:

    (1) If the tab is immediately followed by one of the digits 1 through 9, that line is a continuation line of the previous one. The non-zero numeric character following the tab is considered in column-6.

    (2) Otherwise, the line is an initial line of a FORTRAN statement, and the character following a tab is considered to be in column-7.

    For example, both of the following versions of a source program are acceptable by DEC System-10:

Version 1	Version 2
C SAMPLE PROBLEM	C SAMPLE PROBLEM
bbbbbbDO 10 I=1,20	(T)DO 10 I=1,20
bbbbbbK=I**3	(T)K=I**3
bbb10 TYPE 20, I,	10(T)TYPE 20, I,
bbbbb1K	(T)1K
bbb20 FORMAT(2I12)	20(T)FORMAT(2I12)
bbbbbbEND	(T)END
b=blank space	(T)=tab

    For a FORTRAN-10 program entered by using the tab-key storage-saving technique, repeat problems 8 and 9.

11. For each of the following functions, write a single-line compounded UPDATE command to accomplish it:

    (1) To type out only those lines in a FORTRAN program that have lengths longer than 72 columns. The printout should contain absolute line numbers, line lengths, and the line itself. Do not print out all lines.

    (2) To insert the word EXERCISE in columns 73-80 of every line in a FORTRAN program.

    (3) to print out all subroutine call statements in a FORTRAN program.

    (4) To print out all FORMAT statements.

    (5) To print out all COMMENT statements.

REFERENCES

1. PTSS TEXT EDITOR, Class Notes of a Freshman Course "Engineering Analysis 2", T. W. Sze, University of Pittsburgh, Pittsburgh, Pennsylvania; 1969.

2. A PRIMER FOR PITT TIME-SHARING SYSTEM (PTSS), Chapter 5, Text Editor, T. W. Sze, University of Pittsburgh, Pittshurgh, Pennsylvania; 1970.

3. INTRODUCTION TO A TIME-SHARING SYSTEM, Chapter 6, Text Editor, T. W. Sze, University of Pittsburgh, Pittsburgh, Pennsylvania; 1972.

4. UPDATE Reference Card, Computer Center, University of Pittsburgh, Pittsburgh, Pennsylvania; June, 1979.

5. UPDATE/X - UNIVERSITY OF PITTSBURGH DATA AND TEXT EDITOR, Computer Center, University of Pittsburgh, Pittsburgh, Pennsylvania; 1976.

6. INTRODUCTION TO DEC SYSTEM-10: TIME-SHARING AND BATCH, T. W. Sze, Chapter 6, Text Editor, University of Pittsburgh, Pittsburgh, Pennsylvania; First Edition, 1974; Second Edition, 1977.

7. UPDATE, Gerald W. Bradley, Computer Center, University of Pittsburgh, Pittsburgh, Pennsylvania; 1979.

CHAPTER 3

FORTRAN-10

FORTRAN is the most widely studied and used programming language in the United States. Therefore, this chapter is prepared with the assumption that the readers already have some background knowledge of the language. For those who are not familiar with the language, please consult any one of many FORTRAN manuals available. Two typical ones are:

PROGRAMMING WITH FORTRAN, Byron S. Gottfried, Quantum Publishers, New York, 1972

PROBLEM SOLVING AND STRUCTURED PROGRAMMING IN FORTRAN, F. L. Friedman & E. B. Koffman, Addison-Wesley Publishing, Reading, Massachusetts; 1977

INTRODUCTION

There are not just a few versions of FORTRAN; there are dozens. Even on the DEC System-10 alone, there are several versions available. In attempting to unify all versions of FORTRAN developed in the computer industry, the American National Standards Institute (ANSI) in 1966 set up a standard for FORTRAN, now known as the "1966 ANSI Standard."* However, what has happened since is that the computer industry has used the Standard only as a minimum standard, and every company has extended far beyond that minimum for their own versions of the FORTRAN language. Unfortunately, while the ANSI standard part is uniform, the enhanced parts among different versions are not. Programs written in one enhanced version may require some modifications if run on a machine using a different compiler. The version of FORTRAN covered in this chapter, called FORTRAN-10 by the Digital Equipment Corporation, is a powerful superset of the ANSI standard version. A summary of FORTRAN-10 will be included in this chapter. However, readers are encouraged to seek more details from References 3 and 4.

---

*USA Standard FORTRAN (x3.9-1966), American National Standards Institute, 1966

RUNNING A FORTRAN PROGRAM ON DEC SYSTEM-10

After a FORTRAN program is written and thoroughly checked for its logic, running the program will require two major steps:

(1) To enter and store the FORTRAN program as a disk file.

(2) To compile, load, and execute the stored program.

The following discussion will be devoted to these two steps.

3.1 To Enter and Store a FORTRAN Program

In the DEC System-10 , the source program in FORTRAN, ALGOL, COBOL or MACRO should first be stored as a disk file because the most common way of execution is for the System to search in the disk for a specified program.

There are a number of utility programs available by which a user can enter and store his FORTRAN program. By far the best way is to use the UPDATE editor, which enables a user full editing facilities while entering a program. The details of UPDATE are given in Chapter 2. In this chapter, only the procedure relating to the entering of a FORTRAN program will be demonstrated.

As an illustration, let us consider two programs: one containing just the main program, and the other a main program plus a subroutine. The program listings are as follows:

Program 1

```
C SAMPLE PROGRAM 1
      DO 10 I=1,10
      K=I**3
   10 TYPE 20, I,K
   20 FORMAT(2I10)
```

Program 2

```
C  SAMPLE PROGRAM 2, WITH SUBROUTINE
      ACCEPT 10, M,N
   10 FORMAT(I3)
      CALL CUBE(M,N)
      END

C  SUBROUTINE FOR SAMPLE PROGRAM 2
      SUBROUTINE CUBE(M,N)
      DO 10  I=M,N
      K=I**3
   10 TYPE 20, I,K
   20 FORMAT(2I10)
      RETURN
      END
```

(1) To enter program by the UPDATE editor

The UPDATE editor was originally developed for the Pitt Time-Sharing system (PTSS) using an IBM/360 Model 50 computer, and has since been adapted for use on the DEC System-10. It enables a user not only to enter and store a program, but also to correct errors and to edit. The following shows a typical session with the UPDATE editor. The user's typings are shown in italics.

At a User's Terminal	Comments
.R UPDATE	Call for the editor
INPUT=> ↵	↵ = RETURN key of terminal
>*C SAMPLE PROGRAM 1*	
> *DO 10 I=1,20*	> = prompt from the computer
> *K=I**3*	
> *10 TYPE 20, I,K*	Enter FORTRAN program
> *20 FORMAT(2I12)*	
> *END*	
>*$DONE*	
CATALOG NAME=>*PRG1.FOR*	
6 BLOCKS WRITTEN ON PRG1.FOR[115103,320571]	

In a similar way, Program 2 may be entered and stored. Let us assume that the main program of Program 2 is stored and named as PRG2.FOR and its subroutine as CUBE.FOR.

If a program requires several subroutines, each subroutine may be entered and stored separately as a single file bearing a different name, or they may be combined into one file with one filename. At this point of the illustration, three files have been stored and they are PRG1.FOR, PRG2.FOR and CUBE.FOR. A listing of the programs can be made by using a monitor command:

.TYPE PRG1.FOR, PRG2.FOR, CUBE.FOR

The listings produced may be used as records or for proof-reading.

If the listing shows that the programs have been correctly entered, the programs are ready for compiling, loading and execution.

(2) To enter program via punch cards

The way to enter and store a program deck is to submit it by a batch job. Details of batch jobs are given in Chapter 9. Repeating the example above, the control file deck is first assembled as follows:

```
$JOB [115103,320571]
$PASSWORD DEBBIE
$DECK PRG1.FOR
  Program 1 deck
$DECK PRG2.FOR
  Main program deck
  Program 2
$DECK CUBE.FOR
  Subroutine deck
$EOD
$EOJ
```

```
$JOB [115103,320571]
$PASSWORD STEVE
.UPDATE PRG1.FOR
   Program 1 deck
$$END
.UPDATE PRG2.FOR
   Main program deck
   Program 2
$$END
.UPDATE CUBE.FOR
   Subroutine deck
$$END
$EOJ
```

There is often a need to enter and store a FORTRAN program via a punch card deck. For example, a card deck may have already been prepared. Perhaps the terminals are not available. Although there are more terminals than key punches, the latter are often less in demand and hence more available. After

the deck is assembled as shown in either makeup, the assembled deck is read by a system card reader, and the batch job is submitted. After the job is executed, there should be files PRG1.FOR, PRG2.FOR and CUBE.FOR in this user's disk area.

3.2 To Edit a Stored FORTRAN Program

If any typographical error, missing lines, or duplications are found in the listings of stored programs, the UPDATE editor may be used to make corrections. Suppose the PRG2.FOR listing is produced as follows and errors were found and marked as shown below:

```
C  SAMPEL PROGRAM 2, WITH SUBROUTINE
     ACCEPT 10, M,N
10FORMAT(I3)
     CALL CUBE(M,N)
     END
```

Move over one space

END .... Missing

To make corrections, the UPDATE editor may be used either on a terminal or in a batch job:

(1) Using UPDATE at a terminal

The following represents a terminal session of error correction:

```
.UPDATE PRG2.FOR
C SAMPEL PROGRAM 2, WITH SUBROUTINE
>$CHANGE/PEL/PLE/
C SAMPLE PROGRAM 2, WITH SUBROUTINE
>$AT+2; CHANGE/FOR/ FOR/
   10 FORMAT(I3)
>$AT+1; CHANGE/M,N/M,N)/
      CALL CUBE(M,N)
>      END
>$END
```

After the editing session, the listing should again be typed out for a final verification.

(2) Using UPDATE in a batch job

Assemble a batch job deck as follows. Notice that the order of the cards and their contents are identical to those input lines in the terminal session, with the exception that an UPDATE $-command should be punched as a $$-card.

```
$JOB [115103,320571]
$PASSWORD DEBBIE
.UPDATE PRG2.FOR -
$$CHANGE/PEL/PLE/
$$AT+2; CHANGE/FOR/ FOR/
$$AT+1; CHANGE/M,N/M,N)/
      END
$$END
$EOJ
```

There is, of course, a third way: Noting the errors on PRG2.FOR, repunch the incorrect cards. Insert any missing card. Resubmit the corrected deck as a new batch job. In the batch deck, include a command first to delete the old PRG2.FOR before storing the new PRG2.

3.3 To Compile, Load and Execute a Stored FORTRAN Program

The sequence of executing a FORTRAN-10 program is as follows:

(1) To compile the specified source programs and store the binary object or relocatable files (with extensions of REL) in the disk.

(2) To load the REL files of the main program and all subprograms or subfunction programs called by the program into the core memory.

(3) To begin the execution of the loaded object program from an address determined by the compiler and the loader.

All these steps can be accomplished in sequence by a single monitor command:

```
.EXECUTE list
```

where "list" is a list of all FORTRAN programs (or their REL files if available) including any other subprogram files needed for execution in one problem. If a program belongs to another user but is accessible, the PPN of the owner should be specified along with the filename. If the file is on tape which is already mounted, then the device name should also be specified. Thus, to execute Programs 1 and 2 respectively, issue the following commands:

```
.EXECUTE PRG1.FOR
.EXECUTE PRG2.FOR, CUBE.FOR
```

When an EXECUTE command is issued, the computer will go through a sequence of compiling, loading and execution. The sequence of operations to carry out the command EXECUTE PRG2.FOR, CUBE.FOR is represented by the flow chart shown in Figure 3.1. Note particularly the processing logic by which any unnecessary compiling is avoided.

When a source FORTRAN program is compiled for the first time, a REL file is created and stored. In the user's file directory, pertinent information are also stored, such as the creation time accurate to the minute. When the program is executed again and if the program has not been modified in any way, the REL file is still valid, and compiling again would be superfluous. On the other hand, if the program has been modified since the last compiling, then the existing REL file is not valid, and compiling again during the next execution is necessary. The processing logic does it by comparing the creation time between the source porgram and its REL file. If the creation time of the source is earlier, then the REL file is still valid. If the creation time of the source is later, then the REL is not valid, and compiling should be done again. After a new REL file is created by the re-compiling, its creation time is updated also. This logic is handled by the System and the user is spared the decision. Execution of Program 1 and Program 2 are given below as illustration:

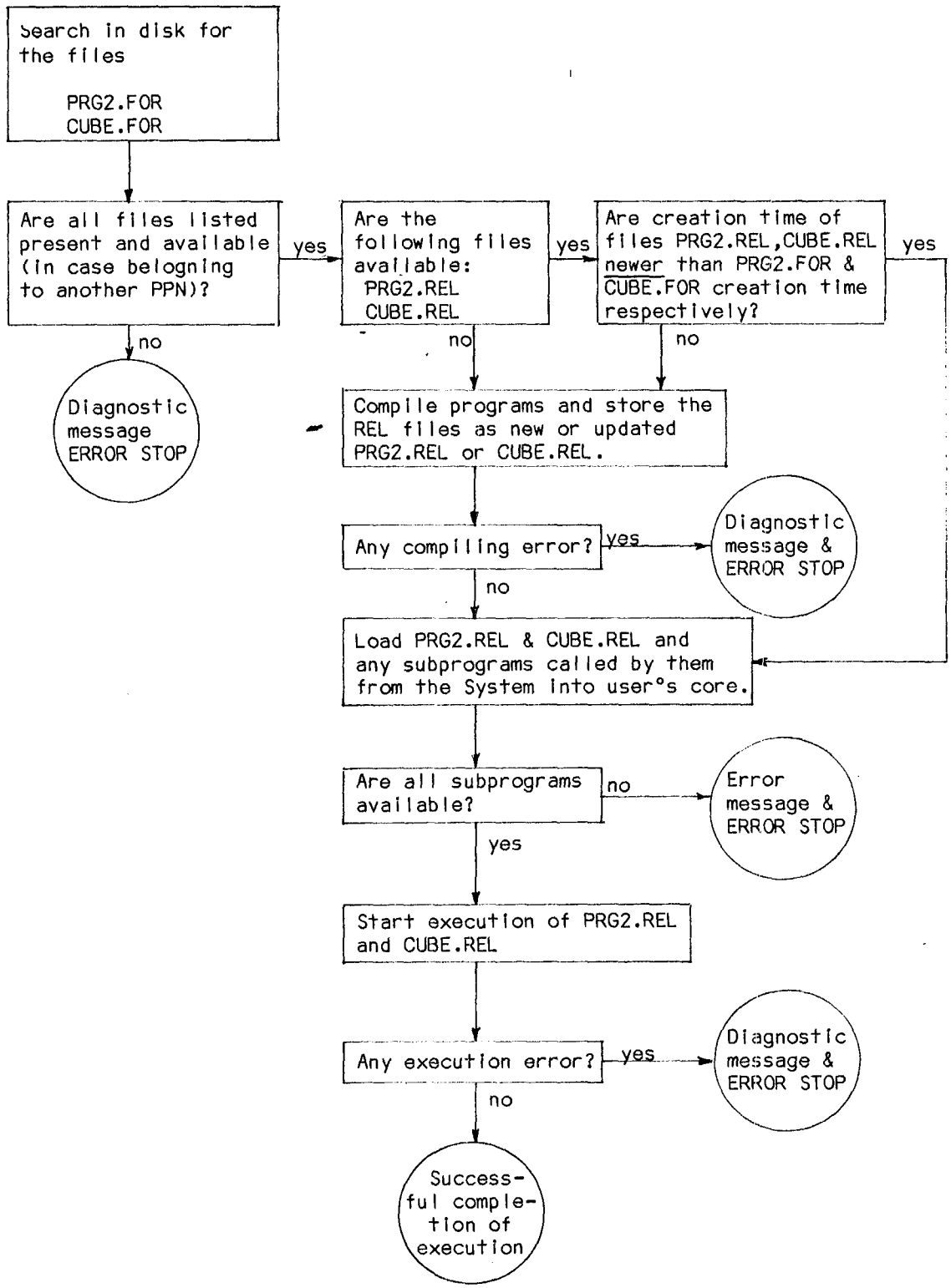


Figure 3.1 Sequence of Operations for "EXECUTE PRG2.FOR,CUBE.FOR"

```
.
FORTRAN 5A(621): PRG1.FOR
MAIN.   OCTAL PROG SIZE=43
LINK:   Loading
[LNKXCT PRG1 execution]

        1         1
        2         8
        3        27
        4        64
        5       125
        6       216
        7       343
        8       512
        9       729
       10      1000

End of execution    FOROTS 5B(1001)
CPU time: 0.08  Elapsed time: 1.05
EXIT
```

```
.
FORTRAN 5A(621): PRG2.FOR
MAIN.   OCTAL PROG SIZE=35
FORTRAN 5A(621): CUBE.FOR
CUBE    OCTAL PROG SIZE=52
LINK:   Loading
[LNKXCT execution]
>
>
        1         1
        2         8
        3        27
        4        64
        5       125
        6       216
        7       343

End of execution    FOROTS 58(1001)
CPU time: 0.05  Elapsed time: 7.50
EXIT
```

The three stages of compiling, loading and execution of a FORTRAN-10 program are carried out by a single EXECUTE command. These steps can also be carried out one at a time.

The monitor command *COMPILE list* will compile the FORTRAN files in the list and store the generated REL files, giving them the same filename but with an extension of REL.

The monitor command *LOAD list* will compile the programs, store the generated REL files, and also load them into the core.

The execution of the stored FORTRAN programs can also be accomplished by submitting the EXECUTE commands in cards. The following are two card assemblies for the batch jobs of executing Program 1 and Program 2:

```
$JOB [115103,320571]
$PASSWORD DEBBIE
.EXECUTE PRG1.FOR
$EOJ
```

```
$JOB[115103,320571]
$PASSWORD DEBBIE
.EXECUTE PRG2.FOR,CUBE.FOR
  1
  7
$EOJ
```

Once the compiling is done on a FORTRAN program, its object program is stored on the disk, and subsequent execution of the same program will bypass the compiling stage. In this manner, unnecessary compiling may be avoided. However, if the FORTRAN program belongs to another PPN, a user should not only ascertain if the FORTRAN program is protected against his access, but he should also determine whether he can gain access to a compiled REL file. If a REl file is already available and accessible, the command EXECUTE will directly access the REL files. In many cases, the source programs are proprietary, but the REL files are available for public access.

If a program will be used many times, a more efficient way of loading can be done in this way. After the program in the "list" of the "LOAD" command are loaded, the core content of the user's area in the core memory may be saved as a file with an EXE extension. The monitor commands to save a core image are LOAD

and SAVE as shown below:

```
.LOAD list
.SAVE NAME
```

and the saved file will have a name of NAME.EXE. Once that is done, subsequent execution of the program may be done by a command of:

```
.RUN NAME
```

where "NAME" is the the name of the specified EXE file.

This procedure is particularly advantageous if (1) a program will be used repeatedly, or (2) the list of programs in the EXECUTE command contains many files and many file specifications. Some of the files may reside on slow and busy peripherals such as the DECtape.

3.4 Optional Switches

The monitor command EXECUTE requires the use of three service programs: the monitor, the FORTRAN compiler, and the loader. In each of the three processors, options are implemented to allow a user to select some variation of services. These options are called switches. Switches are available on all three service processors, and they are separately discussed next.

(1) Monitor switches The details of the switches for the command COMPILE, LOAD and EXECUTE will be given in Chapter 8, so only the most frequently used switches are listed below. The monitor switch has a form of a slash followed immediately by a word which can be abbreviated. These switches and their functions are listed in Table 3.1.

(2) Compiler switches While the monitor program is somewhat uniform among the DEC System-10 users, the compilers--particularly the FORTRAN compiler-- may have many versions, and some with local modifications. Selected switches which appear on the same command line as those of the compiler switches are words enclosed in parentheses. These switches are listed in Table 3.2.

(3) Loader switch The format of a loader switch is a percent sign (%) followed by one or two characters. Three such switches are listed in Table 3.3.

Example: *.EXECUTE SAMPLE.FOR/LIST*
Function: Compile SAMPLE.FOR, store SAMPLE.REL on disk, load it into the core, and execute. Also, generate a source listing file SAMPLE.LST.

Example: *.EXECUTE SAMPLE.FOR/CREF (I) %OM*
Function: Compile (including all D-statements), load and execute. Generate a cross reference file for later CREF program, and produce a loader map at the terminal.

Monitor Switch	Function
/COMPILE	To force a compiling even if there already exists a REL file. The purpose of this switch is to force the use of compiler because certain compiler switches are also chosen in the EXECUTE command. Otherwise, the compiler is bypassed if there already exists a valid REL file bearing the same filename.
/CREF	To produce a cross-reference listing file on the disk for each file compiled for later processing by the CREF program. The cross-references include such information as variable names, statement labels, and their cross references. Before the user signs off, he may get a printout copy of the cross-reference by another monitor command: CREF. If the CRF file generated during a previous session at the terminal still is stored on disk, a list may be obtained by running the CREF program in the following ways: .R CREF *LPT:=NAME.CRF .R CREF *TTY:=NAME.CTF This will produce a copy of listing on the line printer (the left version) or on the terminal (the version on the right).
/LIST	To generate a disk listing file for each file compiled with the same filename, but with an extension of LST. These files can be listed with the PRINT or QUEUE command (see Chapter 8). If a REK file already exists, this switch will be ignored unless a forced compiling is ordered by the /COMPILE switch.
/LIBRARY	To select the loading of only those subroutines and functions referenced in the programs. Otherwise, the entire library file will be loaded.
/DEBUF:BOUNDS	To report if subscripts get out of bounds as defined by the DIMENSION statement for that array. This is one of the most common errors.

Table 3.1 Selected Monitor Switches

Example: *.EXECUTE SAMPLE.FOR, PRG:IMSL/LIBRARY*
Function: Compile the source program SAMPLE.FOR and thus generate SAMPLE.REL. Then load it along with those subroutines in PRG:IMSL that are called by the program SAMPLE.FOR. The LIBRARY switch here is absolutely necessary because the package PRG:IMSL contains about 400 subroutines. Execute when loading is completed.

Compiler Switch	Function
(INCLUDE) or (I)	To compile the program by regarding all statements with "D" in column 1 as FORTRAN statements. If this switch is *not* specified, those statements will be regarded as comments and bypassed. The frequent uses of this switch is to insert the debugging statement as the "D-statements," which are usually output statements to type out intermediate results or to type out tracing progress, such as a message "Reaching check point 5." Once a program is completely debugged, it can be compiled again, but this time without the INCLUDE-switch, and all D-statements will be ignored.
(NOERROR) or (NOE)	To suppress error message on user's terminal. The error message will only appear on the listing file if it is requested by the /LIST or the /CREF switch.
(NOWARNINGS) or (NOW)	To suppress warning messages on the terminal.
(OPTIMIZE) or (O)	To perform global optimization of compiling.

Table 3.2 Selected FORTRAN Compiler Switches

Loader Switch	Function
/S	To load local symbols used primarily for debugging purpose along with the program.
/1M	To type out a loader map at the user's terminal and include local symbols. In a batch job, the loader map with this switch will be included in the log file.
/0M	To type out a loader map at the user°s terminal. In a batch job, this switch will include the loader map in the log file.

Table 3.3 Selected Loader Switches

3.5 An Example of FORTRAN Processing

As an illustration of FORTRAN-10 programming and processing on a time-sharing system, an example will be carried through in all steps. The problem deals with the solution of an equation $Ax + Bx + Cx + D = 0$ with significance to 3 digits. The FORTRAN program for the problem is listed below:

```
C     SAMPLE PROBLEM FOR FORTRAN-10
      READ(5,10)A,B,C,D,X1
   10 FORMAT(F20.7)
    1 X2=X1-(A*X1**3+B*X1**2+C*X1+D)/(3.*A*X1**2+2.*B*X1+C)
      WRITE(6,10)X2
      IF(ABS((X1-X2)/X2)-0.001)3,3,2
    2 X1=X2
      GO TO 1
    3 WRITE(6,11)X2
   11 FORMAT(/' THE REAL ROOT = ', F20.7)
      STOP
      END
```

The rest of this section shows a case history of running this problem, from entering the program, through debugging and editing and finally executing it. Written running comments were added to aid understanding. All text in italics represent the user's own typing; all others are the computer's printout.

```
.UPDATE NEWTON.FOR
[CREATING NEW FILE]
XC     SAMPLE PROGRAM FOR FORTRAN-10
>      READ(5,10)A,B,C,D,X1
>   10 FORMAT(5F)
>    1 X2=X1-(A*X1**3+B*X1**2+C*X1+D)/(3.*A*X1**2+2.*B*X1+C)
>      WRITE(6,19)X2
>      IF(ABS((X1-X2)/X2 .001)3,3,2
>    2 X1=X2
>      GO TO 1
>    3 WRITE(6,11)X2
>   11 FORMAT(/' THE REAL ROOT = ', F20.7)
>     STOP
>      END
>$END
1 blocks written on NEWTON.FOR[115103,320571]
```

Enter a new program by UPDATE

```
.EXECUTE NEWTON.FOR
FORTRAN 5A(621): NEWTON.FOR
00006           IF(ABS((X1-X2)/X2 .001,3,3,2
?FTNUMP LINE:00006 UNMATCHED PARENTHESES
00010      11 FORMAT(/' THE REAL ROOT = ', F20.7)
00011       STOP
?FTNFWE LINE:00010 FOUND "T" WHEN EXPECTING A END OF STATEMENT

UNDEFINED LABELS

   19      11

?FTNFTL   MAIN.            4 FATAL ERRORS AND NO WARNINGS
LINK:   LOADING
[LNKNSA No start address]
EXIT
```

Missing minus sign

Use error diagnosis message to help with debugging.

```
.UPDATE NEWTON.FOR
C     SAMPLE PROGRAM FOR FORTRAN-10
>$TO 6
      IF(ABS(ABS((X1-X2)/X2.001)3,3,2
>$CHANGE/ .001/)-.001/
      IF(ABS((X1-X2)/X2)-.001)3,3,2
>$TO/STOP/
      STOP
>$CHANGE/STOP/ STOP/
      STOP
>$END
1 blocks written on NEWTON.FOR[115103,320571]

EXIT
.EXECUTE NEWTON.FOR
FORTRAN 5A(621): NEWTON.FOR

UNDEFINED LABELS

   19       11

?FTNFTL   MAIN.          4 FATAL ERRORS AND NO WARNINGS
LINK:   Loading
[LNKNSA No start address]

EXIT


.UPDATE NEWTON.FOR
C     SAMPLE PROGRAM FOR FORTRAN-10
>$TRAVEL/19/
      WRITE(6,19)X2
>$CHANGE/19/10/
      WRITE(6,10)X2
>$GO
?>$END
1 blocks written on NEWTON.FOR[115103,320571]

EXIT


.EXECUTE NEWTON.FOR
FORTRAN 5A(621): NEWTON.FOR
MAIN.   OCTAL PROG SIZE=145
LINK:   Loading
[LNKXCT NEWTON execution]
1.0  -16.0  65.0  -50.0  16.0
    12.9158000
    11.1082200
    10.2498400
    10.1173400
    10.0000900
    10.0000000

THE REAL ROOT =            10.0000000
STOP

End of execution   FOROTS 5B(1001)
CPU time: 0.09  Elapsed time: 23.98
EXIT
```

Make changes by UPDATE

Error still exists.

Search for statement 19

Search for more "19".

"?" says "can't find any more."

Execute again

Compile and load successfully

Input data for

$x^3 - 16x^2 + 65x - 50 = 0$

with initial trial value $X1 = 16$

Answer: $x = 10$

A SUMMARY OF FORTRAN-10

This part of the chapter is devoted to a summary of the FORTRAN-10 language, which is an enhancement of the ANSI standard FORTRAN. The enhancement may be a new FORTRAN statement, such as the IMPLICIT-statement; or it may be some additional features in a standard FORTRAN statement, such as those in the DIMENSION-statement. These enhancements will be identified in the summary by a heavy vertical line on the left side of the page, for example:

(5) A debug line A debug line has a character "D" or "d" etc etc etc

The identification of the enhancement will be useful in the conversion of a FORTRAN-10 program to other versions of FORTRAN, or vice versa.

3.6 A Summary of Constants, Variables and Expressions

(1) Constants There are nine types of constants in FORTRAN-10: integer constants, real constants, double precision constants, complex constants, logical constants, literal constants, octal constants, double octal constants, and statement label constants, as summarized in Table 3.4:

Constant	General Form	Remarks and Examples
Integer constant	no decimal point	ranging from $-2^{35}+1$ to $2^{35}-1$
Real constant	always with a decimal	7 to 9-digit precision in mantissa
Double precision constant	exponent symbol is D	3.00D2=300.0000000000000 (16-digit precision)
Octal constant	signed or unsigned octal preceded by a '"	"567, "-567
Double octal constant	same as single precision octal	"1234567000123456700
Complex constant	(x,y)	(3.1,-4.7) for 3.1-j4.7
Logical constant	.TRUE. .FALSE. "-1 "0	
Literal constant	'QUOTE' nHxxxxx	'TIME' 4HTIME
Statement label	1 to 5 decimal digits preceded by "$" or "&"	$1234 &999

Table 3.4 A Summary of FORTRAN-10 Constants

(2) Variables Variables are specified by names and types. The name of a variable consists of one to six alphanumeric characters, the first of which must be alphabetic. The type of a variable may be specified explicitly by a type declaration statement or implicitly by the IMPLICIT statement. If the variable is not specified in this manner, then a first letter of I, J, K, L, M or N indicates an integer variable; any other first letter indicates a real variable.

Variable arrays carry subscripts that are integer constants, variables or expressions. In addition, the following are permitted in FORTRAN-10:

A. A subscript may contain a non-integer arithmetic expression. However, when such a subscript is evaluated, it is truncated and converted to an integer after its evaluation.

B. A subscript may contain a function reference such as A(10*SIN(X)).

C. Subscripted variables may be used as subscripts or nested subscripts of subscripted variables.

(3) Expressions Compounded numeric expressions must be constructed according to the following rule. With respect to the numeric operators of +, -, *, /, any type of quantity (integer, real, double precision, complex, logical, literal, octal or statement label) may be operated with any other, with one exception: A complex quantity may not be operated with a double precision quantity. The result of these mixed mode operations are tabulated in Table 3.5. (Mixed mode operations are not allowed in ANSI FORTRAN.)

Operation +, -, *, /		Type of Argument 2				
		Integer	Real	Double Precision	Complex	Others
Type of Argument 1	Integer	Integer	Real	Double Precision	complex	Integer
	Real	Real	Real	Double Precision	Complex	Real
	Double Precision	Double Precision	Double Precision	Double Precision	Not Allowed	Double Precision
	Complex	Complex	Complex	Not Allowed	Complex	Complex
	All Others	Integer	Real	Double Precision	Complex	Octal

Table 3.5 Results of Mixed Mode Operations

For example, if X is real in an expression (3.1,-4.1)*X, the expression will be complex after evaluation.

The logical operators and relational operators are listed in Table 3.6 and Table 3.7 respectively.

Logical Operators	Meaning	Example
.NOT.	Negation	.NOT.P
.AND.	$\cap$	P.AND.Q
.OR.	$\cup$	P.OR. Q
.XOR.	$\oplus$	P.XOR.Q
.EQV.	$\odot$	P.EQV.Q

Table 3.6 Logical Operators

Relational Operators	Meaning
.GT.	$>$
.GE.	$\geq$
.LT	$<$
.LE.	$\leq$
.EQ.	$=$
.NE.	$\neq$

Table 3.7 Relational Operators

A summary of FORTRAN-10 library functions is shown on Table 3.8.

3.7 FORTRAN-10 Statements

The field format of a FORTRAN-10 statement follows the general rules of FORTRAN-IV statement. There are certain differences associated with a FORTRAN-10 line. In FORTRAN-10, there are following different types of statement lines:

(1) An initial line If a FORTRAN-10 statement has continuation lines, the first line of the group is called an initial line.

(2) A continuation line A continuation line is identified by any character (except for a blank or zero) placed in column 6. A maximum of 20 lines are permitted in a FORTRAN-10 statement including the initial line. Continuation lines may not be interrupted by comment lines.

(3) A multi-statement line A multi-statement line combines several successive statements in a single statement, each component separated from the other by a semicolon (;). If the multi-statement carries a statement number, it is always associated with the first component. For example, two separate statements:

```
A = B*C
X = Y+Z
```

can be combined into a single line as: `A = B*C; X = Y+Z`

(4) A Comment line A comment line has one of the characters (C,$,/,*,!) placed in column 1. Comments may also be added to any statement in the field of columns 7-72, provided that a character (!) precedes the text. For example:

```
A = B*C ; X = Y+Z  !STEP NO. 1
```

(5) A debug line A debug line has a character "D" or "d" in column 1. When the program is compiled, it is ignored unless there is an "(INCLUDE)" switch in the command. This is used for debugging purposes, such as an output line for tracing.

Function	Form	Definition	Type of Argument	Type of Result
Absolute values:				
Real	ABS	|arg|	Real	Real
Integer	IABS		Integer	Integer
Double	DABS		Double	Double
Complex to real	CABS	$c = \sqrt{x^2 + y^2}$	Complex	Real
Conversion:				
Integer/real	FLOAT	Float(Arg)	Integer	Real
Real/integer	IFIX	Integer(arg)	Real	Integer
Real(cmplx)	REAL	REAL part(cmplx arg)	Complex	Real
Imag(cmplx)	AIMAG	IMAG part(cmplx arg)	Complex	Real
Real/Cmplx	CMPLX	c=Arg1 + j Arg2	2 Reals	Complex
Cmpx conjugate	CONJG	c=conjugate(cmplx arg)	Complex	Complex
Truncation:				
Real/real	AINT	Real truncation	Real	Real
Real/integer	INT	Integer truncation	Real	Integer
Remaindering:				
Real	AMOD	Remainder(arg1/arg2)	2 Reals	Real
Integer	MOD	Remainder(arg1/arg2)	2 Integers	Integer
Square root:				
Real	SQRT	$\sqrt{arg}$	Real	Real
Double	DSQRT		Double	Double
Complex	CSQRT		Complex	Complex
Logarithm:				
Real	ALOG	Ln (arg)	Real	Real
	ALOG10	Log (arg)	Real	Real
Double	DLOG	Ln (arg)	Double	Double
	DLOG10	Log (arg)	Double	Double
Complex	CLOG	Ln (arg)	Complex	Complex
Sine:				
Real (radians)	SIN		Real	Real
Real (degrees)	SIND		Real	Real
Double (radians)	DSIN	sin (arg)	Double	Double
Complex	CSIN		Complex	Complex
Cosine:				
Real (radians)	COS		Real	Real
Real (degrees)	COSD		Real	Real
Double (radians)	DCOS	cos (arg)	Double	Double
Complex	CCOS		Complex	Complex
Arc sine	ASIN	$\sin^{-1}$(arg)	Real	Real
Arc cosine	ACOS	$\cos^{-1}$(arg)	Real	Real
Arc tangent:				
Real	ATAN	$\tan^{-1}$(arg)	Real	Real
Double	DATAN	$\tan^{-1}$(arg)	Double	Double
Two real arg	ATAN2	$\tan^{-1}$(arg1/arg2)	Real	Real
Exponential:				
Real	EXP	$e^{(arg)}$	Real	Real
Double	DEXP		Double	Double
Complex	CEXP		Complex	Complex
Hyperbolic:				
Sine	SINH	sinh (arg)	Real	Real
Cosine	COSH	cosh (arg)	Real	Real
Tangent	TANH	tanh (arg)	Real	Real
Maximum value:				
Real	AMAX1	Max(a1,a2,...)	Reals	Real
Integer	MAX0	Max(k1,k2,...)	Integers	Integer
Minimum value:				
Real	AMIN1	Min(a1,a2,...)	Reals	Real
Integer	MIN0	Min(k1,k2,...)	Integers	Integer
Random number	RAN	random number between 0 and 1	dummy	Real

Table 3.8 FORTRAN-10 Library Functions

(6) A blank line This is ignored in compiling, but useful in making the listing easier to read.

Various types of FORTRAN-10 statements will now be discussed. As in all versions of the FORTRAN language, the order of the FORTRAN-10 statements is important in a program. The proper order of the statements is summarized in Table 3.9.

<table>
<tr><td rowspan="6">COMMENT</td><td colspan="3">PROGRAM. FUNCTION, SUBPROGRAM
or BLOCK DATA statements</td></tr>
<tr><td rowspan="5">FORMAT statements</td><td colspan="2">IMPLICIT statements</td></tr>
<tr><td colspan="2">PARAMENTER statemets</td></tr>
<tr><td colspan="2">DIMENSION, COMMON,
EQUIVALENCE, EXTERNAL
NAMELIST, or TYPE
Specification
statements</td></tr>
<tr><td rowspan="2">DATA
statements</td><td>Statement
function
Definitions</td></tr>
<tr><td>Executable
statements</td></tr>
<tr><td colspan="3">END statement</td></tr>
</table>

Table 3.9 A Summary of FORTRAN-10 Statement Sequence

The list of statements in each box indicates the order in which these statements must appear. The table also indicates that certain statements may be placed anywhere in the range shown in the Table. For example, a FORMAT statement may be placed anywhere after the PROGRAM statement and before the END statement.

3.8 A Summary of FORTRAN-10 Compilation Control Statements

Statement	Function
PROGRAM *name*	This statement instructs the compiler to assign "*name*" instead of MAIN as the name of a program. "*name*" must be 6 characters or less. This statement, if written, must be the first statement of a program.
INCLUDE *'file'*	*file*= standard file specification. This statement allows an inclusion of a code segment in a program unit.
END	Physically the last statement of a program or a subprogram.

Table 3.10 A Summary of FORTRAN-10 Compilation Control Statements

3.9 A Summary of Specification Statements

The specification statements specify the type characteristics, storage allocations, and data arrangements. They are summarized in Table 3.11:

Statement	Function
DIMENSION *S1,S2,...Sn*	where *Si* is an array declarator of either of two form: *VARIABLE(max1,max2,...,maxn)* *VARIABLE(min1:max1, min2:max2,..., minn:maxn)* and "mini:max" value represents the lower and upper bounds of an array dimension. The symbol colon (:) may be replaced by a slash (/) as a delimiter. When used in a subprogram, the array dimension may be an integer constant or an integer variable, thus making the dimension adjustable in a subprogram.
TYPE *list*	where TYPE may be one of the following: INTEGER, REAL, DOUBLE PRECISION, COMPLEX or LOGICAL. Size modifiers are acceptable in FORTRAN-10 but are interpreted differently: type*1 = acceptable but interpreted as a full word type*2 = full word type*4 = full word type*8 = double precision
IMPLICIT *TYPE(a1,A2..)* *TYPE(b1,B2,..),...*	where A1,A2,...,B1,B2,... are letters. This statement declares the data type of variables and functions according to the first letters. A range of letters may be specified by a dash between the first and the last letters, for example: IMPLICIT INTEGER (A-N)
COMMON */block indentifier/identifier,identifier,...identifier*	The COMMON statement causes specified variables or arrays to be stored in an area available to other programs. By means of COMMON statements, the data of a main program and/or its subprograms may share a common storage area.
EQUIVALENCE $(V_1,V_2,...)$, $(V_k,V_{k+1},...)$, ...	The EQUIVALENCE statement causes more than one variable within a given program to share the same storage area.
EXTERNAL *name1, name2,* ...	Distinguish the names as names of subprograms to be used as arguments to other subprograms.
PARAMETER *P1=C1, P2=C2,* ...	where Pi = a standard user-defined identifier, Ci = any type of constant This statement defines constants symbolically during compilation.
DATA $list/d_1,d_2,.../$, $list2/d_k,d_{k+1},.../$, ...	The data to be compiled into the object program is specified in this statement. The "list" may be a full array or an partial array in an implied DO format.

Table 3.11 A Summary of Specification Statements

3.10 A Summary of Assignment Statements

The assignment statements are summarized in Table 3.12:

<table>
<tr><th>Statement</th><th>Function</th></tr>
<tr><td>VARIABLE = EXPRESSION</td><td>The EXPRESSION in an assignment statement may be an arithmetic or a logical expression. Their formats are the same. In an arithmetic expression, mixed mode is permitted in FORTRAN-10. The rules of mixed mode expression results depend on the type of VARIABLE in the statement. Note that we are dealing with FORTRAN statements here, while a previous Table 3.5 lists the results of mixed mode operations in a sub-expression. The rules are now summarized below:

Mixed Mode Statement

<table>
<tr><th rowspan="2">Expression Type</th><th colspan="5">Variable Type</th></tr>
<tr><th>Real</th><th>Integer</th><th>Complex</th><th>Double</th><th>Logical</th></tr>
<tr><td>Real</td><td>D</td><td>C</td><td>R,I</td><td>H,L</td><td>D</td></tr>
<tr><td>Integer</td><td>C</td><td>D</td><td>R,C,I</td><td>H,C,L</td><td>D</td></tr>
<tr><td>Complex</td><td>R</td><td>C,R</td><td>D</td><td>-</td><td>R</td></tr>
<tr><td>Double</td><td>H</td><td>C,H,L</td><td>-</td><td>D</td><td>H</td></tr>
<tr><td>Logical</td><td>D</td><td>D</td><td>R,I</td><td>H,L</td><td>D,H</td></tr>
<tr><td>Literal</td><td>D,H %</td><td>C,H %</td><td>D &</td><td>D &</td><td>D %</td></tr>
</table>

Legend: D = direct replacement
C = conversion with truncation
R = real part only
I = imaginary part set to 0
H = high order only
L = low order part set to 0

Note: % = use of the first part of the literal
& = use the first two words of the literal</td></tr>
<tr><td>ASSIGN n TO I</td><td>This is used to assign a statement label constant to a variable name, which will become a statement label variable.</td></tr>
</table>

Table 3.12 A Summary of Assignment Statements

3.11 A Summary of Control Statements (Table 3.13)

Statement	Function
GO TO *n*	An unconditional transfer statement
GO TO (*n1*,*n2*,...,*nk*) or GO TO (*n1*,*n2*,...,*nk*)	Assigned GO TO statement
GO TO *k* OR	GO TO *k*, (*L1*,*L2*,...*Ln*) Assign GO TO statement
IF (*E*) *L1*,*L2*,*L3*	Conventional arithmetic IF statement where *E* = an arithmetic expresssion
IF (*E*) *S*	where S is an executable statement. This is a conventional logical IF statement, where E is a logical expression.
IF (*E*) *n1*,*n2*	where *n1* and *n2* are two statement labels. This is a two-exit logical IF statement and *E* = a logical expression. This statement will transfer the execution to statement label *n1* if *E* equals .TRUE., and to statement *n2* if *E* = .FALSE. In other words, this is an "IF-THEN, OTHERWISE" statement.
DO *n I* = *m1*, *m2*, *m3*	where *n* = terminal statement label *I* = index variable *m1* = initial parameter *m2* = terminal parameter *m3* = increment parameter Note: (1) Nested DO's follow conventional rules. (2) Index variable should not be altered within the loop range. Even an inclusion as a subprogram argument may produce a warning message during compiling. (3) The index variable may be an integer or a real variable. The parameters may be integer or real expressions, which will be calculated at the beginning of the DO loops. (4) Real, integer, positive, negative, zero constants are all permitted for *m1*,*m2*,*m3*. Thus the FORTRAN-10 DO-statements allow decrements, negative indices, non-integer numeric indices.
STOP , or STOP '*literal string*' , or STOP *n*	Terminal will print the *literal string* as a message or *n* as a message.
PAUSE, or PAUSE *literal string* , or PAUSE *n*	The PAUSE statemnt will cause the following message to be printed at the terminal: TYPE G TO CONTINUE, X TO EXIT, T TO TRACE

Table 3.13 A Summary of Control Statements

3.12 Terminology Used in FORTRAN-10 INPUT/OUTPUT (I/O) Statements

One powerful feature of FORTRAN-10 is that it possesses a set of extremely powerful input/output statements, far more powerful than the standard set in the 1966 ANSI standard. In order to present the I/O statements, we will first get acquainted with some terminology:

(1) Transfer mode Data transfer between storage and I/O devices or between storage locations is done in several different modes:

a. Sequenctial mode This is the most common mode, in which the records are accessed or transferred in a sequential order immediately following the last accessed or transferrecd record.

b. Random access mode This permits the access and transfer of records from a file in any desired order. The OPEN (see Section 3.16) statement is required to establish an I/O mode of this kind.

c. Append mode This is a variation of the sequential mode. It permits writing a record immediately after the last record of the accessed file. The OPEN statement is required to establish an I/O mode of this kind.

d. Dump mode

(2) Keywords of I/O statements (Table 3.14)

Keyword	Transfer of Data
READ	from a peripheral device to the processor storage
REREAD	repeat the last READ
ACCEPT	from a terminal to storage
FIND	to locate the next recoprd to be read during a random access READ operation
DECODE	from a specified storage area into the
WRITE	from storage to a peripheral device
PRINT	from storage to a printer
PUNCH	from storage to a card punch
TYPE	from storage to a terminal
ENCODE	to transfer from the variables of a specified I/O list into a specified storage area

Table 3.14 A Summary of Keywords of FORTRAN-10 I/O Statements

(3) Basic formats and components of READ and WRITE statements

Basic Statement Form	Function
KEYWORD (u,f) list	Formatted I/o transfer
KEYWORD (u#R,f) list	Random access formatted I/O transfer
KEYWORD (u,) list*	Listed-directed I/O transfer
KEYWORD (u,name)	NAMELIST-controlled I/O transfer
KEYWORD (u) list	Binary I/O transfer
KEYWORD (u#R) list	Random access binary I/O transfer

where:

KEYWORD = READ or WRITE
u = logical unit number
f = format statement number
list = I/O list
#R = the delimiter # followed by the number of a record in an established (by an OPEN statement) random access file
* = symbol specifying a list-directed I/O transfer
name = the name of an I/O list defined by a NAMELIST statement

In additon, when a unit *u* is specified, the optional argument

ERR=c and *END=d*

may be added to any of the READ or WRITE statement.

Table 3.15 A Summary of READ/WRITE Basic Formats

(4) Logical unit number (Table 3.16)

Unit Number xx	Default Filenames	Use	
		Time-sharing	Batch
1 - 4	FORxx.DAT	DSK	DSK
5		TTY	CDR
6		TTY	LPT
7		CDP	CDP
8-30		DSK	DSK

Table 3.16 Logical Unit Number Assignments

These are decimal numbers to identify the physical devices used for most FORTRAN I/O operations. The devices should be explicitly specified in the OPEN

statement. The definitions of these unit numbers as well as how many are allowed are determined by the local installation. The typical DEC definition specifies units ranging from 1 to 63 assigned to the devices DSK, DECtapes, magtapes, CDR, LPT, PTR, PTP, etc. However, since a different system of peripheral device allocation is used at the University of Pittsburgh, the logical unit numbering system is revised and shown in Table 3.16. Installation at other institutions may have still diffeent definitions depending on the local configurations.

(5) Formatted and unformatted files Files transferred under the control of a format specification are called formatted files. Unformatted files are binary files transferred without a reference to a format specification and are transferred on an one-to-one correspondence between the source and the destination.

(6) Random access records The random access records are specified by an integer preceded by an apostrophe or a pound sign, for example, '123 or #123.

(7) List directed I/O The asterisk (*) is an I/O statement in place of a FORMAT statement number tells the compiler that the specified transfer operation is "list-directed." In a list-directed transfer, the data and their type are specified by the READ/WRITE I/O list. If a READ statement has an asterisk (*) where the FORMAT number usually is, the list-direct I/O will follow the rules listed below:

a. Octal constants in the list-directed I/O are not permitted.

b. Literal constants must be enclosed in single quotes, such as 'TIME'.

c. Blanks and commas are delimiters to separate different items in the I/O list.

d. Complex constants must be enclosed in parentheses.

e. If an item is inputted as a null (blanks, tabs, carriage returns, or linefeeds, but no data), the item will retain a previously inputted value.

f. A slash at any time will terminate the input operation even if the I/O list is not yet satisfied.

g. the repeat of a constant may be written as n*K, which means the constant K repeated n times.

(8) NAMELIST I/O lists The I/O lists are defined by a NAMELIST statement (see Section 3.17) in which each I/O list is named by a one- to six-character name that may be referenced by a READ/WRITE statement. I/O statements with a NAMELIST-defined I/O list cannot contain a FORMAT statement reference or a conventional I/O list. The only type of formatting permitted in the NAMELIST-controlled statements is an input record of $NAME var1=value1, var2=value2,...$.

3.13 A Summary of FORTRAN-10 READ Statements

Table 3.17 shows a summary of different types of FORTRAN-10 READ statements:

Statement	Function
Sequential Formatted READ:	
READ (u,f) list	This is the most frequently used form. It transfer data from logical unit *u* to storage.
READ (u,f)	Input data from unit *u* into either a H-field descriptor or a literal field descriptor given within the referenced format.
READ f	Same as *READ(u,f)* where =default unit for a card reader.
READ f, list	Read data from a card reader into storage.
Sequential Unformatted Binary READ:	
READ (u) list	Read one record from unit *u* into storage. The record must be previously prepared by a FORTRAN-10 unformatted WRITE statement.
Sequential List-Directed READ:	
READ (u,) list*	Read data from device unit *u* into storage as values of items in the list. If necessary, each item is converted to the type assigned in the list.
*READ *, list*	Read data from a card reader a list-directed list.
Sequential NAMELIST-Controlled READ:	
READ (u,name)	Read data from unit *u* into storage as the values of the items identified by the NAMELIST input specified by the name .
Random Access Formatted READ:	
READ(u#R,f) list	Input data from record *R* of unit *u* according to the referenced FORMAT *f*. The input files must be previously set up either by an OPEN or a DEFINE FILE command.
Random Access Unformatted READ:	
READ (u#R) list	Input data from record *R* of unit *u*. Place data into storage as values of items in the *list*. The input file must be a binary file prepared by a previously applied FORTRAN-10 unformatted random access WRITE statements.

Table 3.17 A Summary of FORTRAN-10 READ Statements

3.14 A Summary of FORTRAN-10 WRITE Statements

The WRITE statements resemble the READ statements in formats. Diffeent types of FORTRAN-10 WRITE statements are now summarized in Table 3.18:

Statement	Function
Sequential Formatted WRITE:	
WRITE (u,f) list	This is the most commonly used WRITE form. It transfers data from storage and outputs it on logical unit *u*.
WRITE (u,f)	Output the contents of any H-field or literal descriptor contained by to the logical unit *u*.
WRITE f	Same as *WRITE(u,f)* where *u*=default unit for a line printer.
WRITE f, list	Same as *WRITE(u,f)list* where *u*=default unit for a line printer.
Sequential Unformatted Binary WRITE:	
WRITE (u) list	Output the values of items in the *list* into the file associated with logcial unit *u*.
Sequential List-Directed WRITE:	
WRITE (u,) list*	Output data from storage into logical unit *u*.
Sequential NAMELIST-Controlled WRITE:	
WRITE (u,name)	Output data from storage into logical unit *u* with the values of items as identified by the NAMELIST-defined list specified by the name *name*.
Random Access Formatted WRITE:	
WRITE(u#R,f)list	Output into unit *u* the values from the storage identified by the contents of list to record *R*. Only the disk files that have been set up by either an OPEN statement or a call to the subroutine DEFINE FILE may be accessed by a WRITE statement of this form.
Random Access Unformatted WRITE:	
WRITE (u#R) list	Output into unit *u* the values from the storage identified by the contents of list to record *R*. Only the disk files that have been set up by either an OPEN statement or a call to the subroutine DEFINE FILE may be accessed by a WRITE statement of this form.

Table 3.18 A Summary of FORTRAN-10 WRITE Statements

3.15 A Summary of FORTRAN-10 I/O Statements

All FORTRAN-10 I/O statements, including the READ/WRITE statements already discussed are now summarized together in Table 3.19:

I/O Statement	Formatted	Transfer Format Control		List-Directed
		Unformatted	Namelist	
READ Sequential	*READ(u,f)list* *READ f,list* *READ f*	*READ(u)list*	*READ(u,name)*	*READ(u,*)list* *READ *,list*
Random	*READ(u#R,f)list*	*READ(u#R)list*		
WRITE Sequential or, Append	*WRITE(u,f)list* *WRITE f,list* *WRITE f*	*WRITE(u)list*	*WRITE(u,name)*	*WRITE(u,*)list*
Random	*WRITE(u#R,f)list*			
REREAD Sequential	*REREAD f,list*			
FIND Random only	*FIND(u#R)*			
ACCEPT Sequential only	*ACCEPT f,list*			*ACCESPT *,list*
PRINT Sequential only	*PRINT f,list*			*PRINT *,list*
PUNCH Sequntial only	*PUNCH f,list* *PUNCH f*			*PUNCH *,list*
TYPE Sequential only	*TYPE f,list* *TYPE f*			*TYPE *,list*
ENCODE Sequtntial only	*ENCODE(c,f,s)list*			
DECODE Sequential	*DECODE(c,f,s)list*			

Legend:
u = logical unit number
f = format number
list = I/O list
name = name of specific NAMELIST I/O list
* = specify list-directed I/O
#R = local record position
c = number of characters per internal record
s = address of first storage

Table 3.19 A Summary of FORTRAN-10 I/O Statements

3.16 FORTRAN-10 File Control Statements

The FORTRAN-10 file control contains only two statements: OPEN and CLOSE. They are, however, among the most powerful and versatile statements in specifying the input/output files. The general forms are:

```
OPEN(arg1,arg2,...)
CLOSE(arg1,arg2,...)
```

The arguments have a genral form of *ITEM = value*. The power and versatility of the OPEN and the CLOSE statements are derived from the many options available as the arguments. These arguments are summarized and tabulated in Table 3.20(A&B).

Although there are many available options, many are special purpose type and not frequently used. The simplified version is just to take the most often used arguments: "unit", "file", "dispose" and "directory" in the OPEN statement, and just the "unit" in the CLOSE statement. Thus, the most often used forms are:

```
OPEN(UNIT=u,FILE='NAME.EXT',DISPOSE'value',DIRECTORY='m,n')

CLOSE(UNIT=u)
```

Example: `OPEN(UNIT=5, FILE='INPUT.DAT')`
Function: The disk file INPUT.DAT is opened on unit 5. If the FORTRAN program is written with unit 5 as the input unit, such as in the READ(5,f)list statement, the OPEN statement will change the program execution from TTY input to a file input. This is a convenient way of adapting an existing program from the TTY input to a disk file input.

Example: `OPEN(UNIT=1, FILE='INPUT.DAT', DIRECTORY='115103,320571')`
Function: The disk file IMPUT.DAT[115103,320571] is opened on unit 1.

Example:
```
OPEN(UNIT=3,ACCESS='SEQOUT',FILE='DATA.TMP')
  WRITE-statements on unit 3
CLOSE(UNIT=3)
OPEN(UNIT=1,ACCESS='SEQIN',FILE='DATA.TMP',DISPOSE='DELETE')
  READ-statements on unit 1
CLOSE(UNIT=1)
```
Function: An output file is opened on unit 3, to be named as DATA.TMP. The file is closed after output statge is completed; the file is reopened on unit 1 as an input file. The file is deleted from the disk when the CLOSE statement is executed.

Example:
```
OPEN(UNIT=1,FILE='INPUT.DAT',ACCESS='RANDOM',MODE='ASCII',
1    RECORD SIZE=80,PROTECTION="177)
```
Function: Open on unit 1 a disk file INPUT.DAT for random access I/O operation in ASCII mdoe. The records in the file are 80 characters long. When the CLOSE statement is executed, the file will be given a protection code of 177.

Argument	Possible Value	Function	Open*	Close*	Default Value
UNIT =	lv,ic	To define the logical unit number.	Req	Req	
DEVICE =	lv,ic	To specify the physical name or the logical name of an	Op	Op	logical name u
ACCESS =	Six possible values	To sepcify the type of input qnd/or output statements and the file access mode to be used in a specified I/O operations. The six possible values are: 'SEQIN' = to be read in sequential access mode 'SEQOUT' = to be written in sequential access mode 'SEQINOUT' = data file may be first read, then written record-by-record in a sequential access mode. At this access, a WRITE/READ sequence is illegal. 'RANDOM' = to specify random access mode in either READ or WRITE operation. The RECORD SIZE option is required when this access mode is specified. 'RANDIN' = to specify a read-only random access mode with a named file. 'APPEND' = to specify the APPEND mode. The record specified by an associated WRITE statement is to be added to the end of a named file. You must close it and then reopen the modified file to permit it to be read.	Op	Ig	'SEQINOUT'
MODE =	four possible values	To define the character set of a file or record. Four possible values are: 'ASCII' = to specify an ASCII file 'BINARY' = to specify a FORTRAN formatted binary file 'IMAGE' = to specify an unformated binary file 'DUMP' = to specify the file to be handled in DUMP mode	Op	Ig	'ASCII' for formatted file 'BINARY' for unformatted file
DISPOSE =	six possible values	To specify the action to be taken regarding a file at the close time. Six values are possible: 'SAVE' = to leave the file on the device 'DELETE' = to delete the file if it is on disk or on a DECtape. Otherwise, take no action. 'PRINT' = to queue the file for printing if it is a disk file. Otherwise, take no action. 'LIST' = to queue the file for printing and delete it if it is a disk file. Otherwise, take no action. 'PUNCH' = to output on paper tape punch. 'RENAME' = to change filename	Op	Op	'SAVE'
FILE =	lv,ic	To specify the name of the frile involved in the OPEN or CLOSE statement. The file name format is FLNAME.EXT. Default conditions: FLNAME = FLNAME.DAT FLNAME. = FLANME. (null) = FORxx.DAT where xx = two-digit unit number If the filenames of the same file in the OPEN and the CLOSE statements are different, the file is renamed.	Op	Op	'FORxx.DAT'
PROTECTION =	oc,iv	To specify a protection code. For example: PROTECTION = "155	Op	Op	"057

Table 3.20A FORTRAN-10 OPEN and CLOSE Statements

Argument	Possible Value*	Function	Open*	Close*	Default Value
DIRECTORY =		To specify the directory of the file. Most frequent use is to specify the PPN of the file. To specify a PPN of [123456,654321], use any of the three ways: (1) Single-precision array: *OPEN(unit=1,DIRECTORY=PATH,...)* where PATH and its elements are: *DIMENSION PATH(2)* *PATH(1)="123456* !project number *PATH(2)="654321* !programmer number (2) Double precision array: *OPEN(unit=1,DIRECTORY=PATH,...)* where PATH and its elements are: *DOUBLE PRECISION PATH(2)* *PATH(1)="000000123456000000654321* *PATH(2)="0* (3) Literal constants: *OPEN(unit=1,DIRECTORY='123456,654321',...)*	Op	Op	User's own PPN
BUFFER COUNT =	iv,ic	To specify the number of I/O buffers to be assigned to a particular device.	Op	Ig	Monitor default value
FILE SIZE =	iv,ic	To specify disk file size in words	Op	Ig	Monitor default
VERSION =	oc,iv	To specify the version number of the named file	Op	Op	0
BLOCK SIZE =	iv,ic	To specify block size for all storage media except disk and DECtape.	Op	Ig	Monitor default
RECORD SIZE =	iv,ic	To specify record size in words. Required argument when specifying random access mode.	Op	Ig	Monitor default value
ASSCIATE VARIABLE	iv	In random access mode, it provides storage for the number of the record to be accessed next if the program being executed were to continue to sequential access records starting from the current READ. For example, if record number 3 was read, the ASSOCIATE VARIABLE is 4.	Op	Ig	
PARITY =	two possible values	To set the parity check system for magtape operation. Two possible values are 'ODD' and 'EVEN'.	Op	Ig	System default value
DENSITY =	five values	To set the packing density of magtape. Five values are '200', '556', '800', '1600', and '6250'.	Op	Ig	System default
DIALOG =	none iv,array	The use of this option in an OPEN statement enables you to supersede or defer, at execution time, the values previously assigned to the arguments of the statement. The System will return a message at the user's terminal: UNIT=n:/ACCESS=SEQINPUT/MODE=ASCII ENTER NEW FILE SPECS. END WITH AN ESC. Only the changed file specs needed be entered.	Op	Ig	
ERR =	s	To go to statement No. s when there is an error during the execution of the OPEN or the CLOSE statement.	Op	Op	Error stop

*Legend: ic = integer constant; iv = integer variable;
lc = literal constant; lv = literal variable;
oc = octal constant,

Op = optional; Ig = ignored.

TABLE 3.20B FORTRAN-10 OPEN and CLOSE Statements

Example:
```
OPEN(UNIT=1, FILE='INPUT.DAT')
   Other FORTRAN statements follow.
CLOSE(UNIT=1,FILE='OLD.DAT')
```
Function: Here we have the same unit number for the OPEN and the CLOSE statements, but they are different file name arguments. This is equivalent to renaming a file at the CLOSE time. The INPUT.DAT is renamed as OLD.DAT.

3.17 Format Statements

The FORMAT statements in FORTRAN-10 are in general compliance with the standard FORTRAN. Therefore, only a brief summary will be given here.

The FORMAT statement has a general form of

```
n FORMAT (S , S , ...)
```

where n is the statement number and each S is a data field specifier. The various data field specifiers are now summarized as follows:

(1) Numeric fields In the following list, "w" is an integer specifying the field width; "d" is an integer specifying the number of decimal places to the right of the decimal point or, for the G-format, the number of significant digits. For the D, E, F, and G inputs, the position of the decimal point in the external field takes precedence over the value of d in the format. This means that the decimal point of the input data need not be exactly at the specified column of the format. However, the data must be entered within the field specified in the format.

Floating-point type format	Fw.d
Exponent-type format	Ew.d
Double precision	Dw.d
General format:	
Real & double precision	Gw.d
Integer & logical	Gw
Complex	2Gw.d
Integer format	Iw
Octal format	Ow

(2) Numeric fields with scale factor Scale factors may be specified for D, E, F and G formats. A scale factor is written as nP where P is the identifying character and n is a signed or unsigned integer that specifies the scale factor.

For the F-type conversions (or G-type, if the external field is decimal fixed point), the scale factor specifies a power of ten so that:

$$\text{External number} = (\text{internal number}) * 10^{P}$$

For the D, E, and G (external field not decimal fixed point) formats, the scale factor multiplies the number by a power of ten, but the exponent is changed accordingly leaving the number unchanged except in form. For example, if the statement: FORMAT(F8.3,E16.5) is used to print out two values A and B:

the same numbers under a format of FORMAT(-1PF8.3,2PE16.5) would produce a printout of:

In input operations, the F-type data are the only type affected by the scale factor.

(3) Logical field The logical data field specifier is:

Lw

where "w" is an integer specifying the field width. If the format is used in an input operation, the first nonblank character in the data field is T or F, the value of the logical variable will be stored as TRUE or FALSE respectively. If the entire data field is blank or empty, a value of FALSE is stored. If the format is used in an output operation, (w-1) blanks followed by T or F will be output if the value of the logical variable is TRUE or FLASE respectively.

(4) Variable field width The numeric fields may appear in a FORMAT statement without the specification of the field width "w" or the number of places after the decimal point "d". When this format is used in an input operation, the input data can be entered in a "free form" style so long as a delimiter is used to separate two neighboring data. Any illegal character in a numeric field can be used as a delimiter. However, a good practice is to use either a comma (,) or a blank ( ) as a delimiter. For example, input according to the format:

```
10 FORMAT(2F,E,2I,D)
```

might appear as:

```
-2.34, 2.345, 0.5623E-01, 56, 783, 3.4567234569D+01
```

If such a format is used in an output operation, FORTRAN automatically assume the following field specifiers:

Format	Becomes
D	D25.16
E	E15.7
F	F15.7
G	G15.7 or G25.16
I	I15
O	O15

(5) Alphanumeric fields The format of an aphanumeric field is:

Aw or Rw

The maximum value of "w' is 5 for single precision, 10 for double precision. The A-field deals with variables containing left-justified, blank-filled characters; the R-field deals with variable containing right-justified, zero-filled characters.

(6) Alphanumeric data within a format statement Use nH format or enclose the alphanumeric data in single quotes. See examples below:

```
10 FORMAT(17H PROGRAM COMPLETE)
10 FORMAT(' PROGRAM COMPLETE')
```

(7) Complex field Complex quantitites are transmitted as two independent real quantities. The format specifier consists of two successive real specifiers or one real repeated specifier. For example, the following format can accommodate four complex quantitites:

```
10 FORMAT(4F10.4, 2E14.5, F10.5, F10.3)
```

(8) $ format descriptor A "$" format descriptor at the end of an output FORMAT is used to suppress the carriage return (and the associated line feed) at the end of the current record, except when the FORMAT is automatically repeated when the WRITE statement list contains more items than those in the FORMAT. One typical application is shown in the example below:

Example: The following is a segment of a FORTRAN-10 program:

```
10 FORMAT(' ANSWER YES OR NO '$)
11 FORMAT(A3)
   WRITE(6,10)
   READ(5,11)ANSWER
```

Function: When this segment of the program is executed, the following will appear on the user's terminal:

ANSWER YES OR NO > *(User answers YES or NO here)*

(9) Print control descriptor When FORTRAN output file is printed on a printer or a terminal, the first character of each line (or record) is reserved for the carriage control character which controls the spacing operations of the printer or the terminal. The FORMAT should have a beginning field of 1Ha where "a" is a desired control character. Table 3.21 lists the FORTRAN-10 print control characters.

3.18 FORTRAN-10 Device Control Statements

The FORTRAN-10 device control statements are normally used for magtape operation control, although they also work well with DECtapes and can be used to simulate disk devices. These tape control statements provide a set of run-time tape control instructions.

In order to execute these stantements, magtapes must first be MOUNTed, and a logical name of be given, where "u" is the logical unit for that tape unit in the FORTRAN program. Therefoe, if the device control statements are used in a FORTRAN-10 program, preliminaries such as the following must be carried out before the execution of the FORTRAN program*:

---

*Unless a run-time subroutine, such as RMOUNT, is available to mount a tape. See Section 3.21.

Print Control Character	ASCII Octal Value	Function
space	012	Skip to next line; skip to next page (form feed) after 60 lines.
0 zero	012	Skip a line
1 one	014	Form feed - go to top of next page
+ plus		Suppress skipping - overprint the line
* asterisk	023	@Skip to next line with no formfeed.
- minus	012	@Skip two lines.
2 two	020	@Space 1/2 of a page.
3 three	013	Space 1/3 of a page.
/ slash	024	@Space 1/6 of a page.
. period	022	@Triple space with a formfeed after every 20 lines printed.
, comma	021	@Double space with a formfeed after every 30 lines printed.

Table 3.21 FORTRAN-10 Print Control Characters
@=No effect on a terminal.

```
.DRIVE MT9
.MOUNT MT9:u/WE/VID:B313
```

Here, the VID used is for illustration. If there are more than one tape for the job, the above preliminaries must be done for every tape unit needed in the program.

The device control statements are now summarized in Table 3.22:

Statement	Function
REWIND u	Move and re-position the file back to the first record.
UNLOAD u	Rewind the source reel so that the tape is compiely off the take-up reel. The tape will be ready for unloading.
BACKSPACE u	Backspace one record except if it is already at record No.1. This statement cannot be used for files set up for random access,list-directed,or NAMELIST-controlled I/O operations.
ENDFILE u	Write an endfile record in the file located on device u.
SKIP RECORD u	Skip one record on device u.
SKIP FILE u	Skip one file which follows immediately the current one.
BACKFILE u	Backspace to the first record of the file preceding the current one.

Table 3.22 FORTRAN-10 Device Control Statements

3.19 FORTRAN-10 Subprogram Statements

Subprograms are procedures that are used repeatedly in a program or among the users, and therefore it is more convenient to define such common procedures so that they may be referenced. The arguments for such a common procedure are made general enough so that the subprograms can be utilized widely. These arguments are called dummy arguments. Dummy arguments in a FORTRAN-10 program may be one of the following: (1) variables, (2) array name, (3) subroutine identifiers, (4) function identifiers, or (5) statement label identifiers that are denoted by the symbol "*", "$", or "&".

These subprogram statements are now summarized in Table 3.23:

Statement	Function
NAME(arg1,arg2,...,argn) = expression	This defines an internal subprogram, where *NAME* is the name assigned, (*arg1,arg2,...*) is a list of dummy arguments.
TYPE FUNCTION NAME(arg1,arg2,...,argn)	where *TYPE* = optional type specification such as INTEGER, REAL, et. (*arg1,arg2,...*) = a list of dummy arguments.
SUBROUTINE NAME(arg1,arg2,...,argn)	
CALL NAME(arg1,arg2,...,argn)	Definition of a subroutine and calling a subroutine
ENTRY NAME(arg1,arg2,...,argn)	Multiple entry specification where: *NAME* = name to be assigned to the desired entry point. Rules of multiple entry in a FORTRAN-10 subroutien are given later.
RETURN	Return the control form the subroutine to the calling program. Next statement executed is one immediately following the calling statement in the calling program.
RETURN k	This is a multiple-return statement, where *k* is an integer constant, variable or expression. Rules of multiple return are given alter.

Table 3.23 A Summary of FORTRAN-10 Subprogram Statements

Often, many subprograms share a common computational procedure. Although these common procedures can again be made into subprograms to be called by subprograms, an alternative is to construct one subprogram with many entrance points. In Figure 3.2, a flow chart is shown for three entrance points and one exit. The entrance points are labeled as SUB (the front entrance), PTA and PTB (two side entrances). The program segments are represented as Segments 1, 2 and 3.

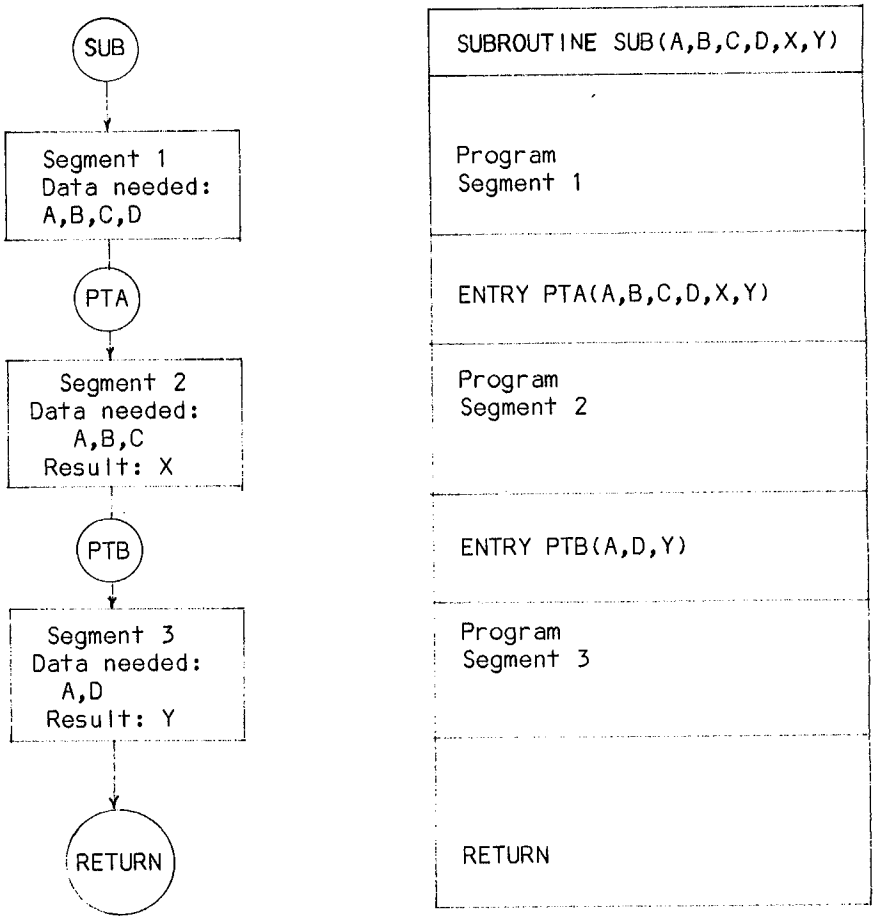


Figure 3.2 An Example of Multiple Entry Subprogram

The following rules on ENTRY should be noted:

(1) An ENTRY statemnet may not be placed in the main program.

(2) An ENTRY statement may not be placed in a DO loop.

(3) There is no need for the arguments of various ENTRY statements to agree with each other.

(4) Value of function must be returned by the use of current ENTRY name.

The statement *RETURN k* enables the selection of any labeled statement of the calling program as a return point. When the multiple returns form of this statement is executed, the assigned or calculated value of *k* specifies that the return is to be made to the kth statement label in the argument list of the calling statement. The value of k should be a positive integer that is equal to or less than the number of statement labels given in the argument list of the calling statement. If k is less than 1 or is larger than the number of available statement labels, a standard return operation is performed.

SUBPROGRAM LIBRARIES IN FORTRAN

3.20 Selected FORTRAN-10 Subprograms Developed by DEC (Table 3.24)

Subprogram Name	Effect
DATE(ARRAY)	"ARRAY" is a dimensioned variable in the calling pr5ogram with 2 elements. The suroutine will return the values: ARRAY(1) = 'DD-Mm', ARRAY(2) = 'm-YY' When ARRAY is printed with a 2A5 field format, the result is DD-Mmm-YY, for example, 19-Aug-80, the date when the subprogram was executed. To force the "month" part into all upper case letter, the following two statements should be inserted between the CALL DATE and WRITE statements: ARRAY(1) = ARRAY(1) .AND. "777777777677 ARRAY(2) = ARRAY(2) .AND. "577777777777 Then the above date example would be printed as 19-AUG-80.
TIME(X) or TIME(X,Y)	These subroutines will return a string constant X as 'HH:MM' as the current time in a 24-hour clock notation, and ' SS.S' for Y, where HH=hour, MM=minutes, SS.S=seconds.
ERRSET(N)	To control the typeout of execution-time arithmetic error messages. Message is suppressed after N occurences.
ERRSNS(I,J)	To determine the exact nature of an error on READ, WRITE, OPEN and CLOSE that was trapped with the "ERR=s" option in the statement. The subroutine will return two integers I,J. The (I,J) combination describes the nature of error according to a code table defined by DEC. (See Appendix H of Reference 4.)
EXIT	To terminate the subprogram.
RELEAS(u)	To release the logical unit u.
SAVRAN(I)	It sets its argument ot the last random number (interpreted as integer) that has been generated by the function RAN.
SETRAN(I)	The starting value of the function RAN is set to 1. If I=0, RAN uses its normal starting value.
SORT('OUTPUT=INPUT/switches')	The argument is a string representing a SORT prgram command. The details on the SORT program are given in Chapter 7. Check with local installation whether this subprogram is installed in the system.

Table 3.24 A Selection of FORTRAN-10 Subprograms Developed by DEC

3.21 Selected Subprograms Developed at the Pitt Computer Center

A group of subprograms have been developed and implemented in the FORTRAN-10 at the installation of the University of Pittsburgh. These subprograms are included for the convenience of Pitt users. DEC System-10 installation elsewhere would have similar types of subprograms but geared particularly to the local needs. These programs are often made available to other installations by exchange, lease or purchase. Since these suprograms have beeen implemented already in the Pitt FORTRAN-10, no additional monitor commands are needed to call them. For users elsewhere, they must confirm first with their installation personnel whether such or similar subprograms are available in their facilities.

The subprograms will be outlined according to their general functions:

(1) Supplementary library functions This group of subprograms are all functions and is used to supplement the DEC-supplied library functions (such as square root and sine function) which are given in Section 3.6 as Table 3.8. These supplementary functions are listed in Table 3.25:

Function	Form	Definition	Type	
			Argument	Function
Tangent				
Real (radians)	TAN	tan(x)	real	real
Real (degrees)	TAND	tan(x)	real	real
Cotangent				
Real (radians)	COTAN	cot(x)	real	real
Real (degrees)	COTAND	cot(x)	real	real
Gamma function	GAMMA	(x)	real	real
Error function	ERF	erf(x)	real	real
Complementary erro function	ERFC	1 - erf(x)	real	real
CPU time	XEQTIM	CPU time in milliseconds	dummy	real

Table 3.25 Supplementary FORTRAN-10 Library Functions Developed at the University of Pittsburgh

(2) Bit manipulation in a memory word

A DEC-10 memory word contains 36 bits. The hardware store a 37th bit for parity check, but that is of no concern to the user. The bits are numbered from 0 to 35 (from the most significant bit side to the least) as shown in Figure 3.3(a).

The group of bit-manipulation subprograms can be used for a wide range of applications, such as data re-formatting in data transfer between a magtape and disk storage. One particular application is in the area of character-storage manipulation. Since ASCII-coded characters are coded into 7-bit bytes, where a "byte" is a unit consisting any number of bits, each memory word can accommodate

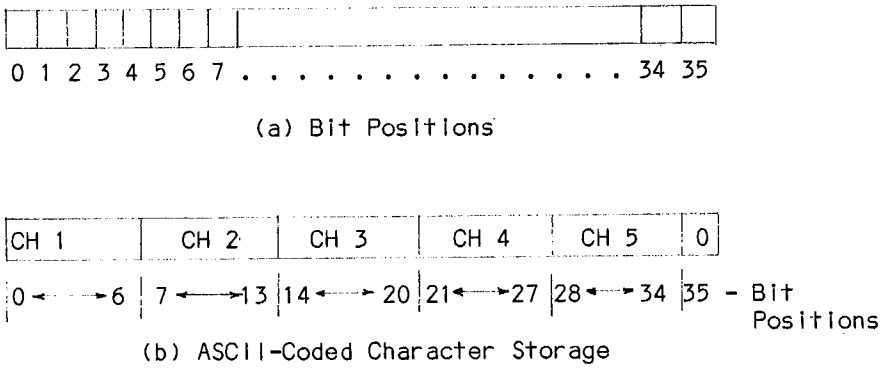
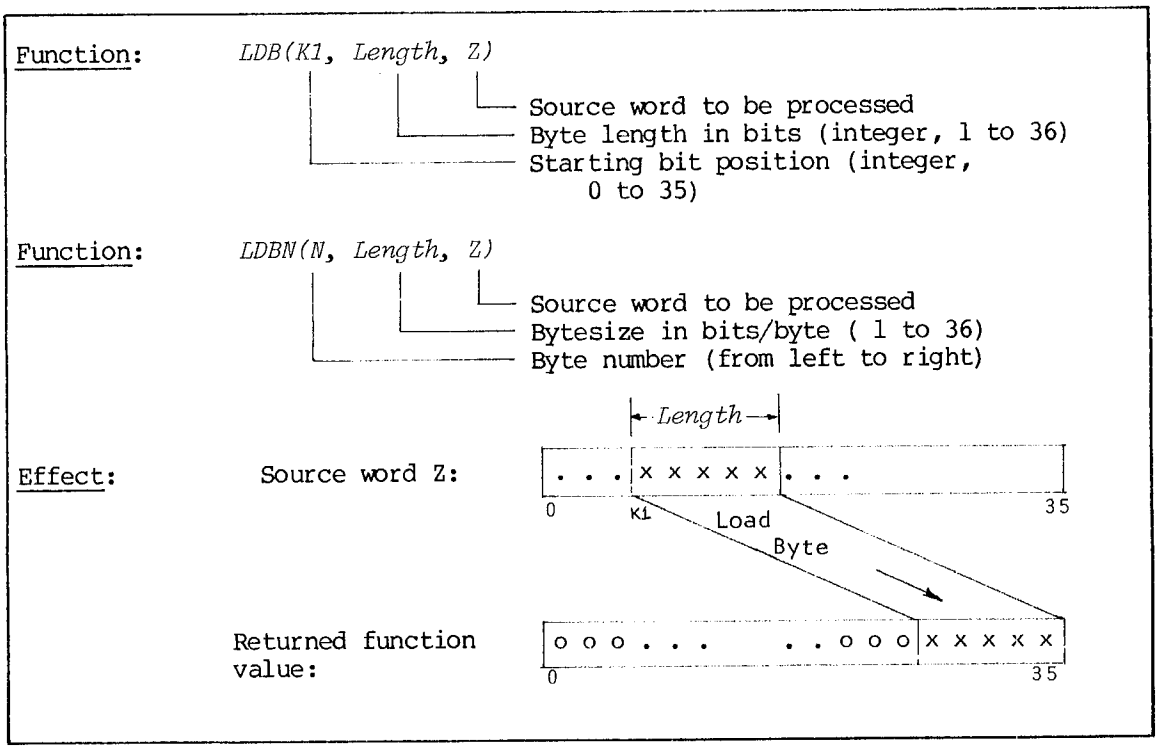


Figure 3.3 DEC-10 ASCII Storage Format

5 characters with one bit left over. The standard ASCII coded storage format is shown in Figure 3.3(b). As a result, bit-35 is always filled with a zero-bit when the word is an ASCII-coded word.

These subprograms are now outlined below:



Subroutine: *CALL* *DPB (K1, Length, Z2, Z1)*

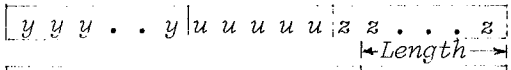
Starting bit position
Byte length
Destination word
Source word

Subroutine: *CALL* *DPBN (N, Length, Z2, Z1)*

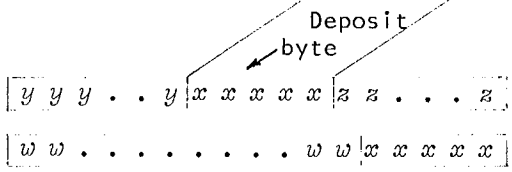
Starting byte number
Bits/byte
Destination word
Source word

K1 and Length are integer constants or variables.

Effect "Before":

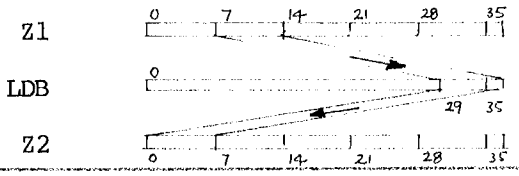


"After":



Z1 is unchanged. Z2 is unchanged except the deposited bute field.

Example: *CALL DPB(0,7,Z2,LDB(7,7,Z1))*
Exexution of this call will replace the first character in in the word Z2 by the second character in Z1 as shown below:



Function: *LSH (word, shift)*

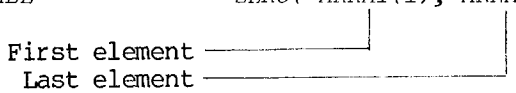
number of places to be shifted:
+integer=shift left
-integer=shift right
word to be processed

Effect: *word:* y y y . . . y x x x x x

LSH(word,+k) y y···y x x x x x 0···0

Example: *M = LSH('ABCDE',-14)*
The string word 'ABCDE' is shifted right 14 places. The shifted-out bits are replaced by 0's. Thus, the returned function value is M = ' ABC', because the ASCII-coded character for code 0000000 is null.

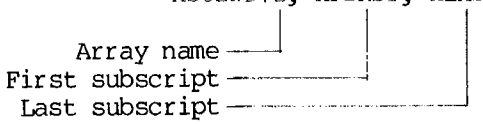
Subroutine: *CALL ZERO(ARRAY(I), ARRAY(J))*



Effect: Set all elements within the specified range to zero. Array may be of any type.

Example: *CALL ZERO(A(1), A(100))*
Set A(1), A(2), ...A(100) to 0.

Subroutine: *CALL ASCEND(Z, KFIRST, KLAST)*



Effect: Sort the Z-array from Z(KFIRST) to Z(KLAST) in an ascending order and then store them in the same array locations.

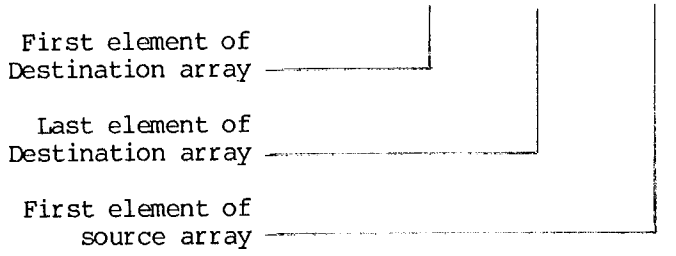
Example: *CALL ASCEND(X,1,100)*
Sort the X-array from X(1) to X(100) and store them in ascending order as the new X-array from X(1) to X(100).

Subroutine: *CALL SPRAY (Z(I), Z(J), VALUE)*

Effect: Set the Z-array of the specified subscript range to equal to the *VALUE*.

Example: *CALL SPRAY(Z1),Z(100),1.5)*
Set Z(1), Z(2),..., Z(100) to equal 1.5.

Subroutine: *CALL MOVE (A2(I), A2(J) , A1(K))*



Effect: Copy an array A2 from A1 in this manner:

```
A2(I) = A1(K)
  .....
A2(J) = A1(K+J-1)
```

A1 and A2 arrays should be of the same type, and avoid double precision or complex array because the second word of each two-word element won't copy.

(4) Device and file specifications

Subroutine: *CALL RMOUNT (u , VID , WE , Label , Serial)*

- Integer, logical unit unmber — *u*
- String constant or variable, VID: — *VID*
- 'WE' (or 0) or 'WL' — *WE*
- 'SL' (or 0) or 'NL' — *Label*
- Used only if Label='NL' — *Serial*

Effect: A run-time MOUNT instruction for a magtape or DECtape.

Example:
```
CALL RMOUNT(1,'B313',0,0)
```
This is equivalent to issuing two monitor commands before the execution of the FORTRAN program:
```
.DRIVES MT9
.MOUNT MT9:1/WE/VID:B313
```

Subroutine:
```
CALL  IFILE (unit, filename, extension, PPN)
CALL  OFILE (unit, filename, extension, PPN)
```

where *unit* = integer constant, logical unit number
filename = 5-character or less string
extension =3-character or leess string
PPN = 12-digfit octal constant

Default extension is 'DAT'.
Default PPN is user's own PPN.

Effect: These are respectively equivalent to:

```
OPEN(unit=u,file='filename,extension',directory='p,pn',
            access='seqin')
CLOSE(unit=u,file='filename,extension',directory='p,pn',
            access='seqout')
```

Example:
```
CALL IFILE(1,'INPUT')
```
Specify user's INPUT.DAT as an input file on unit 1.
```
CALL IFILE(2,'SAMPLE','TMP',"115103320571)
```
Note that although 6-character filename is given, IFILE and OFILE will only treat it as a maximum of 5-character string (because it is coded as ASCII instead of SIXBIT). Hence the search will be for a file SAMPL.TMPin the PPN of [115103,320571], instead of the specified file SAMPLE.TMP. If there is actually a file named SAMPL.TMP, this wrong file will be called. If there is no SAMPL.TMP, execution comes to an error stop.

3.22 The SUBSET Subprogram Package

Many subprograms have been developed by the faculty, staff and students at the University of Pittsburgh. Many of these are polished, optimized, and well documented. One such work is the SubSET (SUBprograms to Simplify Encoding Tasks), written by Ronal K. Nicholas* and stored under the PPN of [121403,250321]. By permission of Mr. Nicholas, a selection of SUBSET programs with their *subset* properties will be outlined. These subset properties are so chosen as to represent the salient points in these subprograms. For more details, the readers are referred to Reference 7, the SUBSET manual.

(1) Subprograms to report job information

Subprogram Name	Function or Subroutine	Effect
CORE(IP)	subroutine	Return an integer *IP* which is equal to the number of pages of core memeory for the current program with the fractions of page rounded to the next higher value.
IDENT(ID)	Function or subroutine	As a subroutine, it returns the argument *ID* as 15 ASCII characters in a 3-word array. The form of the ASCII string is '[m,n]' in three words. As a function, it also returns with "m" in the left half, and "n" in the right half of the returned word, both as 6-digit octal constants.
LOCATE(L)	Function or subroutine	As a subroutine, it sets the user's job to station *L*. If used as a function, it returns a functional value of .TRUE. if successful. Otherwise, it returns a value of .FALSE.
MYJOB(JOB)	Function or subroutine	Return a functional value or argument *JOB* the job number.
MYLINE(LINE)	Function or subroutine	It returns the argument *LINE* as the user's TTY line number. If it is a Batch job, the value is negative.
MYNAME(NAME)	subroutine	It returns a 3-word array containing 15 ASCII characters left-justified, which is the user's name as stored in the system.
WKDAY(TODAY)	subroutine	It returns a 3=character string which is the day of the day of the week, such as 'Mon', 'Tue', etc.

(2) Subprograms to manipulate arrays

These subprograms deal with initializing an array, copying one array onto another, and finding minimum and maximum elements in an array.

---

*Ronal K. Nicholas, Research Associate, Division of Research in Medical Education, School of Medicine, University of Pittsburgh

Subroutine: *CALL COPY (Z1, INCZ1, Z2, INCZ2, NTOTAL)*

- Z1 — First source array element
- INCZ1 — Z1-increment
- Z2 — First destination array element
- INCZ2 — Z2-increment
- NTOTAL — Total number of elements to be copied

Effect: Copy the Z1-array by the Z2-array with indicated starting array elements and subscript increments. Z1 and Z2 are the first array elements in the specified copying process. If the first element has a subscript of one, the subscript may be omitted, and Z1 or Z2 may appear as array name.

Example: *CALL COPY(X,1,Y,1,10)*
This is equivalent to:

```
      DO 5 I=1,10
    5 Y(I)=X(I)
```

CALL COPY(X(2),2,Y(9),3,10)
This is equivalent to:

```
      DO 5 I=1,10
    5 Y(3*I+6)=X(2*I)
```

Subroutine: *CALL INIT(Z, NTOTAL, ZVALUE)*

- Z — Array element or array name
- NTOTAL — Total number of Z-element to be initialized
- ZVALUE — Common value

If Z is an array name, initialization begins from Z(1). If Z is an array element, initialization begins from the given element. Z must be a single precision, real or integer type.

Effect: Initialize the array by the common value given

Example: *CALL INIT(X,50, 0.0)*
Set X(1),X(2),...X(50) to zero.

CALL INIT(K(10), 50, KODE)
Set K(10),K(11),...K(59) to a pre-defined value KODE.

CALL INIT(POINT, 132, ' ')
Set POINT(1),POINT(2),...POINT(132) as blanks.

Function or Subroutine:

MINX (ITEM(I) , ITEM(J) , INDEX)
MAXX (ITEM(I) , ITEM(J) , INDEX)
AMINX (ITEM(I) , ITEM(J) , INDEX)
AMAXX (REAL(I) , REAL(J) , INDEX)

First element in the specified array, integer or real as indicated.

Last element in the specified array, integer or real as indicated.

Order of Min or Max element in the specified list.

Effect: As a subroutine, it returns as *INDEX* the order of the minmax number in the given array. The actual subscript of the minmax element and the value of that minmax will require additional computation:

subscript of the minmax element = I + INDEX -1
MINMAX = ITEM(I+INDEX-1) or REAL(I+INDEX-1)

As a function, it only returns the value of the minmax element. The subprogram is not applicable to double precision or complex list.

Example: *CALL AMAXX(X(3),X(300),INDEX)*
If the subroutine returns a value of INDEX as 59, then the maximum of the X-list is X(61).

(3) Subprogram to control TTY characteristics

This subprogram will accomplish at execution-time a control of terminal characteristics properties in the same manner of what the monitor command "SET TTY" can accomplish at the monitor level. In a monitor command "SET TTY" (or "TTY" in its short form), the general form is: TTY *keyword* , where *keyword* is either one of a complementary pair of arguments, such as *PAGE* or *NO PAGE*. In the subprogram shown here named as SETTTY, the "PAGE" part of the example is called a Code Parameter, and yes-or-no part is called a Logic Parameter. Thus the entire group of TTY commands can be coded into a single subroutine. This is shown next.

Function or Subroutine: *SIXBIT (Z, I, J)*

- J — number of character to be converted to the SIXBIT code
- I — Destination of character after conversion
- Z — Source of ASCII character to be converted

Effect: When used as a subroutine, it returns an array I which is the SIXBIT code of Z. If it is used as a function, the first 6 characters (padded with blanks if necessary) is returned as the value of the function.

Note: Both Z and I are dimensioned variables for the same ASCII characters. However, SIXBIT codes contain six characters per word, while the ASCII codes contain five characters per word. So, the dimensions of Z and I could be different.

Example: *CALL SIXBIT('SYS', IDUM, 3)*
Convert the ASCII string 'SYS' into SIXBIT code as IDUM.

Subroutine: *CALL RUN(DEVICE,SAVEFILE,PPN)*

- PPN — PPN (octal) where file is stored. PPN=0 if in own disk.
- SAVEFILE — Octal number, SIXBIT code of filename of the EXE file to be run.
- DEVICE — SIXBIT code of the device (no colon)

e.g. DSK = "446353000000
(or = "0)
DTA0 = "446441200000
SYS = "637163000000

Effect: This is equivalent to STOP for the current program; then apply a monitor command of ".RUN DEV:NAME[m,n]".

If DEVICE='SYS', 'NEW' or 'OLD' in SIXBIT codes, then PPN=0. If DEVICE='MT7', 'MT8', or 'MT9' in SIXBIT code, the tape must be already properly mounted and positioned.
The RUN subroutine will drop all files in the old program. If files in the old program are dropped without first a CALL RELEAS call, the files will be lost if they are output files, and will not be available as intermediate data for running the chained programs.

Example: *CALL RUN(0,SIXBIT('DEPT',IDUM,4),"115103320571)*
This is equivalent to STOP the current program and then issue a monitor command of "RUN DEPT[115103,320571]"

For the convenience of users and by the permission of Mr. Nicholas, a copy of the SUBSET package is stored also in ENG: , which is the depository of the Engineering Program Library.

Since SUBSET is not in the FORTRAN-10 Library but in the user-library the EXECUTE command of a FORTRAN program should specifically includes "ENG:SUBSET.REL/LIB" in its list, if the program calls any subprogram in the SUBSET package. In a batch job, a $INCLUDE card is necessary. For example, the following is an execution command for a program that calls the SUBSET subprograms:

```
.EXECUTE MAIN.FOR, SUB1.FOR, ENG:SUBSET.REL/LIB
```

3.23 Comprehensive FORTRAN Subroutine Libraries

In an academic user community of the size of the University of Pittsburgh, it has been estimated that more than 500 "new" Gaussian Elimination programs for simultaneous equations were written, debugged, and run each year. Many of these came out of courses in programming, numerical methods, engineering analysis, economics, statistics, etc. Most of them are justifiable as they provide the students opportunities to sharpen their skill on a familiar problem with proven methods of solution. But some were unnecessary exercises to "re-invent the wheels" since the elements of student learning are absent in those exercises. Such activities are pure waste of human resources and computer resources.

It may be said that computer applications in radically different disciplines share a common ground that an application must be first mathematically formulated. Once so done, the differences between disciplines disappear. For example, the Gaussian Elimination method would be applicable whether the problem was orginated from a power system load flow study or a regression study from an economics model, so long as the problem is formulated as a system of linear simultaneous algebraic equations. Thus a software package containing standard solutions to various mathematical problems is a very useful tool to computer users in all disciplines.

In order for such a software package to serve a large group of users in many diversified fields, there are several important requirements that must be satisfied:

(1) These programs should be callable in the forms of subprograms (subroutines or functions), so that the user's program remains in control.

(2) These subprograms should be self-contained so that they will not require further attention from the users other than passing the values of the subprogram parameters into the subprograms. In particular, there should not be any input/output statements in the subprogram. Thus the input/output operations become the responsibility of the user's main program. There are exceptions, of course. A subprogram may be designed explicitly for input or output operations, for example, to list and tabulate a matrix.

(3) In order to adapt to the need of different users, each subprogram should have capability of adjustable dimension size as well as user-controllable error level. At least an estimate of error level should be available as a return value of the subprogram, so that the user, who has no knowledge of how this subprogram was constructed, will know the level of performance of the program.

(4) There should be clear and uniform documentations available to guide the users in defining the subprograms, including the dummy parameters, their types, array sizes, order in the parameter list, and their meaning.

At the University of Pittsburgh, two such packages are available. One is the International Mathematical & Statistical Library (IMSL) which in on-line as PRG:IMSL.REL. The other is the IBM Scientific Subroutine Package (SSP)*, which is not on-line but may be placed on-line by running a UARC program, as it to a great extent duplicates the IMSL coverage. Both packages are comprehensive in their coverage, and their documentations are excellent but voluminous. However, when a user is faced with a big programming job whose purpose may be mode than a programming exercise, it will be cost-effective to use these library facilities, even to the extent of modifying the program in order to fit.

Both IMSL and SPP contain several hundred subprograms in the package, and therefore are too voluminous to include in this bbok even in a summarized form. Only the areas of coverage will be given here to give the readers some idea about the comprehensiveness of the package:

IBM SSP Package:

Statistics:

Probit analysis
Variance analysis
Correlation analysis
Multiple linear regression
Polynomial regression
Canonical correlation
Factor anaysis
Discriminant analysis
Time series analysis
Data screening and analysis
Nonparametric tests
Random number generation
Distribution functions

Mathematics:

Inversion
Eigenvalues and eigenvectors
Simulataneous linear algebraic equations
Transpositions
Matrix arithmetics
Matrix partitioning
Matrix tabulation and sorting of rows or columns
Elementary operations on rows or columns of matrices
Matrix factorization
Integration and differentiation of given or tabulated functions
Solution of systems of first-order differential equations
Fourier analysis of given or tabulated functions
Bessel and modified Bessel function evaluation
Gamma function evaluation
Jacobina elliptic functions
Elliptic, exponential, sine cosine, Fresnel integrals
Real roots of a given equation
Real and complex roots of a real polynomial equation.
Polynomial arithmetic
Polynomial evaluation, integration, differentiation

---

*For Pitt users, the SSP source programs are stored and available on a UARC tape B4473. See Section 10.7 for the UARC procedure.

Chebyshev, Hermite, Laguerre, Legendre polynomials
Minimum of a function
Approximation, interpolation, and table construction

IMSL Package: Chapter headings:

Analysis of Variance
Basic Statistics
Catagorized Data Analysis
Differential Equations; Quadrature; Differentiation
Eigensystem Analysis
Forecasting; Econometrics; Time Series; Transforms
Generation and Testing of Random Numbers
Interpolation; Approximation; Smoothing
Linear Algebraic Equations
Mathematical and Statistical Special Functions
Non-Parametric Statistics
Observation Structure; Multivariate Statistics
Regression Analysis
Sampling
Utility Functions
Vector, Matrix Arithmetic
Zeros and Extrema, Linear Programming

Example: Suppose we are to solve a system of 50 simultaneous equations. In matrix form, the equation is Ax=B. Suppose the matrices have been stored as DATA.DAT file with a format of (10E12.4). In the file, the first 250 records are the A-matrix by rows, and the last 5 records are the B-matrix. Obtain the solution by using the IMSL package.

Program: The first step of this problem is naturally to search through the IMSL documentation to see if there is one that fits the problem. Such a problem would of course be under the category of "Linear Algebraic Equations." When such a program is found, the user's task is to prepare a main program which calls this IMSL routine. To do so, the main program will include the following parts:

(1) To provide storage (the DIMENSION statement) for all variables required for the problem. This not only includes the problem variables but also the working variables. The IMSL documentation gives detailed and exact requirements of DIMENSION.

(2) To input the data needed by the Library subprogram. This includes opening of files, reading of data from file or terminal, calculations needed for the subprogram parameters, etc.

(3) To call the IMSL subprogram.

(4) To output the results.

IMSL Reference Manual (Reference 10) is a seven-inch thick reference book. The content is divided into 17 chapters, and Chapter L is on Linear Algebraic Equations. In going through the routines in that chapter, the routine

LEQT1F lists the following headings:

```
IMSL ROUTINE NAME - LEQT1F
PURPOSE           - LINEAR EQUATION SOLUTION - FULL STORAGE
                    MODE - SPACE ECONOMIZER SOLUTION
```

This seems to satisfy our need. The other information listed by the Manual are included below:

```
USAGE   - CALL LEQT1F(A,M,N,IA,B,IDGT,WKAREA,IER)

A       - Input matrix of dimension N by N containing the
          coefficient matrix of the equation Ax=B. On
          output, "A" is replaced by the LU decomposition
          of a rowwise permutation of "A".
M       - Number of right-hand matrix columns (input)
N       - Order of "A" and number of rows in "B".
IA      - Row dimension of A and B exactly as specified in
          the DIMENSION statement of the calling program.
B       - Input matrix of dimension NxM containing
          right-hand side of the equation Ax=B. On output,
          the NxM solution X replaces B.
IDGT    - Input option: If IDGT>0, the elements of A and B
          are assumed to be correct to IDGT decimal digits
          and the routine performs an accuracy test. If
          IDGT equals zero, the accuracy test is bypassed.
WKAREA  - Work area of dimension >= N.
IER     - Error parameter (output).
          Terminal error: IER=129 indicates that matrix A
          is algorithmically singular.
          Warning error: IER=34 indicates that the
          accuracy test failed. The computed solution may
          be in error by more than can be accounted for by
          the uncertainty of the data. This warning can be
          produced only if IDGT is greater than 0.
```

In checking over these specifications, the following should be noted:

(1) The matrices A and B will be destroyed after the execution of the subprogram. If they are needed later, protect them by copying them into another set of variables, or else later re-read the input data A and B.

(2) The DIMENSION for the storage declaration should be A(IA,IA), B(IA,M). In addition, it is also the responsibility of the calling program to dimension WKAREA(IA). Note that B is dimensioned as a matrix with two subscripts. If B is a vector, as in most linear systems, B should be dimensioned as B(IA,1).

(3) N and IA need not be the same, but N should never exceed IA. If N is an input quantity and made to be less than IA, such a calling program would be able to solve a system of linear algebraic equations of an order specified by the user up to IAth order. Such a program would increase its flexibility immensely.

The program for this problem is listed below:

```
      DIMENSION A(100,100),B(100,1),WKAREA(100)
***** DEFINE THE SIZE OF PROBLEM "N"
      WRITE(6,100); READ(5,101)N
100   FORMAT(/' ENTER NUMBER OF VARIABLES = '$)
101   FORMAT(I)
***** GET INPUT DATA FOR THE SUBPROGRAM
      OPEN(UNIT=1,FILE='DATA.DAT',ACCESS='SEQIN')
102   FORMAT(10E)
      DO 10 I=1,N
10    READ(1,102) (A(I,J),J=1,N)
      READ(1,102) (B(I,1),I=1,N)
***** CALL IMSL SUBPROGRAM LEQT1F
      M=1; IA=100; IDGT=0  !SUBROUTINE PARAMETERS
      CALL LEQT1F(A,M,N,IA,B,IDGT,WKAREA,IER)
***** OUTPUT THE RESULTS
103   FORMAT(/' X(',I2,') = ', E12.4)
      WRITE(6,103) ((I,B(I,1)), I=1,N)
      STOP
      END
```

Suppose we name the stored program EQUAT.FOR. This program may be executed by a monitor command of:

```
.EXECUTE EQUAT.FOR, PRG:IMSL/LIB
```

With the dimension set up in EQUAT.FOR, it is capable to solve a system of up to 100 equations. However, when solving a large system, the accuracy requirement may be difficult to satisfy because of the accumulation of round-off and truncation errors during computations. Then the accuracy test would fail in the subroutine execution, giving the output IER a non-zero report.

The following is the computer printout of the execution:

```
.EXECUTE EQUAT.FOR, PRG:IMSL/LIB
FORTRAN 5A(621): EQUAT.FOR
MAIN.   OCTAL PROG SIZE=24167
LINK:   Loading
[LNKXCT EQUAT execution]

ENTER NUMBER OF VARIABLES = >100

X( 1) =  0.9996E+00

X( 2) =  0.1996E+01

X( 3) =  0.3000E+00

X( 4) =  0.4000E+00

  etc.....
```

3.24 Array Processor

In many engineering and scientific applications, the computations often involve a relatively simple algorithm done repeatedly on log sequences of data. The data may be one-dimensional sequence of numbers (called vectors), or two or more dimensional sequences, (called arrays), for example, a matrix. In such computations, heavy overhead must be absorbed on such "book-keeping" chores of array indexing, loop counting, and data fetching. In conventional computer organization, such overhead must be absorbed by incorporating them sequentially into the program, thus competing for machine time with the actual computations.

The concept of parallel processing is to provide hardware so that independent computations can be performed at the same time and result in a much faster program execution.

At the DEC-10 installation at the University of Pittsburgh, one such parallel processor, the Floating Point 190L Array Processor, is attached to the System B. The AP190L is a pipe-line machine that allows the calculations of overhead for elements up stream to be performed simultaneously with the element computations down stream (therefore, the name pipe-line).

To the FORTRAN users, the usage of the AP190L means to incorporate certain AP190L subroutines calls in the main program. Thus, writing a FORTRAN program that uses the array processor to process data follows the general rules of FORTRAN subroutine calls. There are a few exceptions:

(1) The array processor must be initialized before using other AP190L subroutines.

(2) Data must be transferred from DEC-10 to AP190L main data memory before the array processor can operate on it.

(3) In order to synchronize the AP190L with the DEC-10, wait calls (in FORTRAN subroutine) must be inserted in the program whenver the DEC-10 and AP190L interact.

(4) At the end of array processor execution, data must be transferred back to DEC-10.

All of these steps are done by calling certain appropriate AP190L subroutines. These subroutines are listed and explained in details in Reference 11. The AP190L Math Library contains subroutines distributed in the following areas:

(1) Data transfer and control operations

(2) basic vector arithmetic

(3) Vector-to-scalar operations

(4) Vector comparison operations

(5) Complex vector arithmetic

(6) Data formatting operations

(7) Matrix operations

(8) Fast Fourier Transform operations

(9) Auxiliary operations

(10) Utility operations

(11) signal processing operations

(12) Table memory operations

Users should consult Reference 12 concerning the usage of the AP190L. Specifically, note the following:

(1) AP190L is attached to DEC-10 System B as a periperal device. Therefore, just as a tape unit, it requires the "DRIVE" monitor command to reserve it. See Section 8.10.

(2) It requires large core memory, larger than most time-sharing allocations. Therefore, array processor runs should be submitted as batch jobs. See Chapter 9 on how to submit batch jobs.

3.25 FORTRAN 77

The FORTRAN programming language is one language that is universally available, on computers, large or small, in the United States, Europe or the rest of the world. Thus, its greatest contribution is that a program written in FORTRAN can be run on any machine, after some minor modifications are made if required.

The ANSI FORTRAN IV, standardized by ANSI in 1966, has exercised a powerful influence on the portability characteristics of the language. In the past fifteen years, there have been many enhancements of the ANSI standard, and FORTRAN-10 is one such enhancement. Varieties of these enhanced versions generate a new need for standardization. Thus, an updated standard language was announced in 1977, unofficially known as FORTRAN 77, and was formally standardized in 1978 by ANSI (ANSI Standard X3.9-1978). While compliance with the ANSI standard is voluntary, it is expected that all FORTRAN languages will be in time evolved into this new version. By necessity, programming languages must have universal portability, and the ANSI standard has powerful influences. FORTRAN-10 already possesses most of the new attributes of FORTRAN 77, but many keywords and syntax are different. It is expected that in a few years, FORTRAN-10 will be replaced by some version of FORTRAN 77. Details of FORTRAN 77 are outside the scope of this book. Interested readers are referred to References 13 and 14 for more details.

REFERENCES

1. PROGRAMMING WITH FORTRAN, Byron S. Gottfried, Quantum Publishers, New York; 1972.

2. PROBLEM SOLVING AND STRUCTURED PROGRAMMING IN FORTRAN, F. L. Friedman and E. B. Koffman, Addison-Wesley Publishing, Reading, Massachusetts; 1977.

3. DEC SYSTEM-10 FORTRAN-10 LANGUAGE MANUAL, Second Edition, DEC-10-1FORA-B-D, Digital Equipemnt Corporation, Maynard, Massachusetts; 1974.

4. DEC SYSTEN-10 FORTRAN PROGRAMMER'S REFERENCE MANUAL, AA-0944E-TB, Digital Equipemnt corporation, Maynard, Massachusetts; 1977.

5. FORTRAN-10 USERS GUIDE, Computer Center, University of Pittsburgh, Pittsburgh, Pennsylvania; 1977.

6. PITT Programmer Notes, Special FORTRAN-10 Issue, Vol. 6, No. 5, August 1, 1977, Computer Center, University of Pittsburgh, Pittsburgh, Pennsylvania; 1977.

7. SUBSET MANUAL, Ronal K. Nicholas, University of Pittsburgh, Pittsburgh, Pennsylvania; 1977.

8. SYSTEM/360 SCIENTIFIC SUBROUTINE PACKAGE (360A-CM-03X) PROGRAMMER MANUAL, IBM Corporation, White Plains, New York.

9. Help File PRG:IMSL.HLP, the Computer Center, University of Pittsburgh, Pittsburgh, Pennsylvania; 1980.

10. IMSL LIBRARY REFERENCE MANUAL, Edition 7, International Mathematical and Statistical Library, Houston, Texas; 1979.

11. AP MATH LIBRARY MANUAL, Volumes 1,2,3, Floating Point System, Inc.; 1979.

12. Help File PRG:APU.HLP, Computer Center, University of Pittsburgh, Pittsburgh, Pennsylvania; 1980.

13. FORTRAN 77, FEATURING STRUCTURED PROGRAMMING, L. P. Meissner and E. I. Organick, Addison-Wesley Publishing Company, Reading, Massachusetts; 1980.

14. PROGRAMMING IN STANDARD FORTRAN 77, A. Balfour and D. H. Marwick, North-Holland Inc., New York, New York; 1979.

CHAPTER 4

FORTRAN PROGRAM DEBUGGING

4.1 Introduction

One of the most important but unpleasant stage in the computer usage is the necessity to debug a program. The development of programs and the subsequent computer execution involve a long chain of events that requires error-prone human actions. These errors can be committed by beginners as well as by experienced users. The detection and the correction of such errors affect seriously the productivity of computer processing applications. These errors are colloquially referred to as "bugs", and the process of detecting and correcting them as "debugging."

The following are some typical statistics regarding the productivity of professionals in the software industry:

The average productivity of a professional programmer in U.S. is seven (7) FORTRAN statements per working day.

For the software development done at a commercial software firm, 65% of the software cost is attributed to debugging.

Breakdown of computer processing failures: (From Reference 1)

Hardware failure	1%
System software failure	2%
Operator mistakes	5%
System failure	2%
Programming errors	90%

It becomes increasingly obvious in the commercial software industry that debugging is by far the major component of the software cost. Conversely, when a software is developed on a fixed budget, the extent of testing and debugging becomes the deciding factor for the software product reliability. In the recent decade, considerable efforts have been spent on the optimal allocation of resources, design of software structure for easy testability and maintainability, test and validation procedures for softwares, and various diagnostic aids, resulting collectively in a new discipline known as "software engineering."

Unfortunately, in spite of advances in the software engineering practices, the degugging of a computer program still depends heavily on the user's knowledge and experiences in the problem, the language, and the computer, and hence it still remains largely as an art. However, over the years, accumulation of expertise and experience has resulted in the formulation of reliable guide lines, good programming styles and practices, checklists for DO's and DON'T's, error reporting and diagnostic facilities in the language processors, and on-line debugging tools. It is, therefore, the purpose of this chapter to present a summary of these practices, with particular emphasis on FORTRAN program debugging.

4.2 Types of Errors

When a FORTRAN program fails, a very natural inclination of the user is to suspect that "the computer is acting up again." Mercifully, the computer system hardware and system software failures are quite rare nowadays, and program errors can usually be blamed as the culprits.

Program errors are the most numerous and also the most complicated. They may be divided into the following categories:

(1) Errors in problem definition They are errors resulted from failures to translate the problem requirement faithfully into the program requirements.

(2) Coding errors They appear in several different forms:

a. Transcription errors, such as incorrect punctuations and misspellings. Such errors will usually be caught at compiling, but some errors may go undetected as perfectly legal program statements and a compiler may not always be able to spot them.

b. Syntax errors, or improper use of FORTRAN statements. Such errors can usually be detected by the compiler.

c. Structural errors or failures to provide correct interaction between two parts of a program, for example, failure to pass the values of parameters from the main program to a subprogram correctly.

(3) Logic errors. These are failures to sequence the problem properly at a detailed level.

For the remainder of the chapter, we will be mainly concerned in two areas of the debugging process:

(1) How can we reduce the incidence of all types of bugs?

(2) If a bug exists, how do we detect and correct it?