digital

pdp10 timesharing handbook 1970

pdp10 timesharing handbook

digital pdp10 handbook series

MONITOR COMMANDS

NAME	ABBRE-VIATION	ARGUMENTS 1	2	3	4	5
ASSIGN	AS	$\underline{\text{dev}}$	ldev			
ASSIGN†	AS	$\underline{\text{SYS}}$	$\underline{\text{dev}}$			
ATTACH	AT	$\underline{\text{job}}$	[proj, prog]			
ATTACH†	AT	$\underline{\text{dev}}$				
CCONT	CC					
COMPILE	COM	$\underline{\text{list}}$				
CONT	CON					
CORE	COR	core				
CREATE	CREA	$\underline{\text{file}}$	.ext			
CREF	CREF					
CSTART	CS	adr				
CTEST						
D(deposit)	D	$\underline{\text{lh}}$	$\underline{\text{rh}}$	adr		
DAYTIME	DA					
DDT	DD					
DEASSIGN	DEA	dev				
DEBUG	DEB	$\underline{\text{list}}$				
DELETE	DEL	$\underline{\text{list}}$				
DETACH	DET					
DETACH†	DET	$\underline{\text{dev}}$				
DIRECT	DI	dev				
E(examine)	E	adr				
EDIT	ED	file	.ext			
EXECUTE	EX	$\underline{\text{list}}$				
FILE	FIL	arg				
FINISH	FIN	$\underline{\text{dev}}$				
GET	G	$\underline{\text{dev}}$	$\underline{\text{file}}$	.ext	[proj, prog]	core
HALT	↑C					
KJOB	K					
LIST	LI	$\underline{\text{list}}$				
LOAD	LOA	$\underline{\text{list}}$				
LOGIN	LOG					
MAKE	M	$\underline{\text{file}}$	.ext			
PJOB	PJ					
R	R	$\underline{\text{file}}$	.ext	core		
REASSIGN	REA	$\underline{\text{dev}}$	$\underline{\text{job}}$			
REENTER	REE					
RENAME	REN	$\underline{\text{arg}}$				
RESOURCES	RES					
RUN	RU	$\underline{\text{dev}}$	$\underline{\text{file}}$	.ext	[proj, prog]	core
SAVE	SA	$\underline{\text{dev}}$	$\underline{\text{file}}$	.ext	core	
SCHEDULE†	SC	$\underline{\text{n}}$				
SSAVE	SS	$\underline{\text{dev}}$	$\underline{\text{file}}$	.ext	core	
START	ST	adr				
SYSTAT	SYS					
TALK	TA	$\underline{\text{dev}}$				
TECO	TE	file	.ext			
TIME	TI	job				
TYPE	TY	$\underline{\text{list}}$				

Key:

adr	octal address	lh rh	octal value of left and right half words
core	decimal number of 1K blocks	[proj, prog]	project-programmer numbers
dev	physical device name	list	a single file specification or a string of file specifications
ldev	logical device name		
.ext	filename extension	arg	a pair of file specifications or a string of pairs of file specifications
file	filename		
job	job number assigned by Monitor	n	scheduled use of the system.
	privileged command	___	underline means always required

See Book 2 and Book 7 for further explanation of commands. These abbreviations are accurate and unique as of now, but their accuracy and uniqueness may be changed in the future by the addition of new commands.

PDP-10 TIMESHARING HANDBOOK

Prepared by
The PDP-10 Software Writing Group
Programming Department
Digital Equipment Corporation

Additional copies of this handbook may be ordered from the Program Library, Digital Equipment Corporation, Maynard, Mass. 01754. Order code AKW. $5.00 each. Discounts are available on five or more copies.

PDP-10 HANDBOOK SERIES

All rights reserved.
Permission to reproduce this handbook or any parts thereof may be obtained from the PDP-10 Product Line Manager, Digital Equipment Corporation, Maynard, Mass.
The material in this handbook is for information purposes and is subject to change without notice.

Copyright © 1968, 1969, 1970 by
Digital Equipment Corporation

The following are trademarks of Digital Equipment Corporation, Maynard, Massachusetts:
DEC Digital PDP

INTRODUCTION TO TIMESHARING
(pages 1-1 thru 1-12)

GETTING STARTED WITH THE MONITOR
(pages 2-1 thru 2-20)

BASIC
(pages 3-1 thru 3-78)

AID
(pages 4-1 thru 4-99)

FORTRAN
(pages 5-1 thru 5-133)

DEMONSTRATION PROGRAMS
(pages 6-1 thru 6-49)

ADVANCED MONITOR COMMANDS
(pages 7-1 thru 7-21)

UTILITY PROGRAMS
(pages 8-1 thru 8-89)

APPENDICES
(pages A-1 thru B-4)

INDEX
(pages Index-1 thru Index-10)

FOREWORD

We have written this handbook for the individual with little or no programming skill in an attempt to bring timesharing programming competence to an ever-expanding circle of new computer users. With this volume as his guide, we hope he can soon acquire the necessary programming knowledge to improve his business or professional activity by the application of computer technology.

I'm pleased to acknowledge here the work of the many DEC programmers, designers, and engineers who continue to advance the state of the timesharing art in both hardware and software, and the DEC software writers and technical artists who prepared this volume.

[illegible]

President, Digital Equipment Corporation

PREFACE

In developing its timesharing capability, Digital has built a history of success very similar to the company's record in realtime applications. That history started in 1960 when Digital's customers began building timesharing systems around PDP computers. Three years later Digital itself started development of its own timesharing system, the PDP-6; and in 1964 the PDP-6 became the first timesharing computer to be delivered with manufacturer-supplied hardware and software.

The PDP-10, which emerged in 1967, is the successful culmination of many years of computer research. Its power, versatility, and low cost make it a leader in the general-purpose timesharing field. For its timesharing users, the PDP-10 performs scientific data analyses, helps make better management decisions, aids in engineering and architectural design, makes investment analyses, and provides management information services.

With this handbook, Digital attempts to bring its documentation on timesharing to a par with its hardware and software accomplishments. The handbook is intended primarily for students, scientists, engineers, and financial analysts who have little or no experience in programming. From it they can learn timesharing programming from a remote Teletype using disk input/output.

This is not to say that an experienced programmer is automatically debarred from using this document. If the reader happens to be a programmer, he should skip the preliminary books, go straight to the computer language in Book 5, and commence programming. In Book 6 he will find that Demonstration Programs 3 and 4 are geared to his level of programming knowledge and competence.

A synoptic view of the contents of the handbook is as follows. Book 1 describes the evolutionary history of timesharing and gives the reader an insight into the way it operates. Book 2, in explaining the elementary monitor commands, shows the reader how to get on the system. In Books 3 and 4 the reader will find conversational programming with BASIC and AID, respectively. Book 5, as already indicated, contains FORTRAN. Four demonstration programs constitute Book 6; advanced monitor commands are found in Book 7; and the four utility programs Batch, CHAIN, LINED, and TECO appear in Book 8.

Since the handbook will be revised periodically in order to improve it and keep it up to date, we solicit the reader's constructive evaluations in the questionnaire at the back of the book. Please fill out the questionnaire and return it to

PDP-10 Software Writing Group
Programming Department
Digital Equipment Corporation
Maynard, Massachusetts 01754

A companion volume, the PDP-10 Reference Handbook, is likewise in print. It is oriented toward experienced programmers who are interested in writing and operating assembly-language programs.

CONTENTS

Foreword V

Preface VI

Book 1 Introduction to Timesharing 1-1

A general description of the operation of a variety of factors in the evolution of timesharing.

Book 2 Getting Started with the Monitor 2-1

Logging in, description of files, elementary commands to create, edit, manipulate, translate, load, and execute files, getting information from the system, and logging off the system.

Book 3 Conversational Programming with BASIC 3-1

A complete book, explaining the procedures for logging in, logging off, and writing, editing, and running programs in BASIC—a problem-solving conversational language that can be used to solve both simple and complex mathematical problems.

Book 4 Conversational Programming with AID 4-1

A complete book explaining how to program in AID, an algebraic conversational language designed to solve both simple and complex numerical problems.

Book 5 Programming in FORTRAN 5-1

A reference book describing the specific statements and features of the FORTRAN IV language, designed primarily for the experienced FORTRAN programmer.

Book 6 Demonstration Programs 6-1

Four programs designed to utilize concepts and commands discussed in this manual. Demonstrations three and four also make use of advanced commands in TECO and DDT.

Book 7 Advanced Monitor Commands 7-1

Explanation of monitor commands not covered in Book 2—commands to allocate system resources, produce line printer listings, manipulate core images, start a program, and get information from the system.

Book 8 Utility Programs 8-1

Includes BATCH—sequential execution of a series of jobs, CHAIN—allows users to deal with FORTRAN programs too large to fit or load into the amount of core available, LINED—a line-oriented editor designed for use on the disk, and TECO—description and explanation of the most frequently used commands of this powerful text editing program.

Appendices A-1

Master Index Index-1

Book 1

Introduction to Timesharing

INTRODUCTION TO TIMESHARING

Why Timesharing?

Early computers were the province of the mathematician. Used mainly to solve differential equations, the systems were narrow in scope and poorly utilized. Since few persons were knowledgeable enough to employ the enormous processors, one individual could monopolize computer time—sit at the console and solve problems in step-by-step fashion.

batch processing

As more people discovered computing techniques, it was no longer practical to let a few persons monopolize computer time. To increase machine efficiency, batch processing was introduced. In this mode of operation; no time was wasted between jobs. Programs were punched on cards and the cards stacked and fed to the computer in batches. Operation of each program was governed by control cards that took the place of the human operator.

Since card reading is a relatively slow process, some early systems employed a small computer to read the cards and transfer program information to magnetic tape that was then input to the large computer. As a further refinement, programs were assigned priorities, with short jobs being executed first to minimize job turnaround.

But what about the computer user? As computer utilization improved, program development took more time. To develop a new program, a user performed the following procedure. After writing the program on paper, he carried it to a keypunch operator to have the cards punched and verified. A day or so later, when the program was returned, the user checked for punching errors, then returned to the keypunch for corrections.

Next, he sent the cards to the computer center for compilation. The compilation, which might not be returned for a half day or more, could reveal spelling or syntactical errors. The cards then had to be changed and resubmitted—another half day's wait. If the next compilation was successful and the program was run, program logic errors might be discovered—new cards, new compilation, etc., etc. In addition, the user often studied reams of computer listings to find the errors. Using these inefficient methods, even simple programs might take weeks to develop.

interaction

Batch processing maximizes machine efficiency in routine data processing operations where turnaround is not critical. But for program development and modification, the user requires another mode of operation. The user needs a way to "interact" with the computer—to feed his program to the system, line by line, and continuously check the results.

In fact, the user may want to develop interactive programs. These programs, which are extremely productive tools, ask the user questions and perform an analysis based on his answers. Electronic circuit design programs are a prime example. The computer actually designs the circuit by asking the engineer questions and manipulating his answers. In addition, interaction provides a new dimension in management information reporting. Via an interactive terminal, a manager can request summaries, plot trends in plant operation and sales, and select special data for use in decision making.

dedicated system

If the user had unlimited funds, he might be tempted to buy or lease a large computer—a system he could dedicate to his work that would provide sufficient power, many peripherals, and a large variety of software. With such a system, the user could develop programs interactively or utilize batch processing for routine tasks. However, costs in excess of $20,000 per month normally preclude the dedication of a large system to a single user.

timesharing

By using timesharing, the user has most of the benefits of a dedicated system at a small fraction of the cost. Timesharing with today's technology allows a large powerful computer to handle 20, 50, 100 or more users simultaneously. Through a choice of terminals, the user can interact with the system or initiate batch processing which runs concurrently. The user also has access to a choice of mass storage and peripherals and a selection of languages and application programs. Since response is fast, the user appears to have a dedicated system. Yet costs are shared. He pays only for the time and facilities that he requires and doesn't pay for the time the machine is idle.

The Operation of a Timesharing System

A timesharing system isn't just any computer with some additional hardware and software. It's a system designed specifically for timesharing. Otherwise, facilities are limited, fewer users can be handled efficiently, and economics are unattractive. At a minimum, a timesharing system requires a central processor with sufficient speed and power, input/output terminals, and an amount of core memory adequate to hold several users.

time slice
time quantum
round robin operation

In a simple timesharing system, each program is assigned a fixed time slice or time quantum and operation is switched from one program to another in round robin fashion until each program is completed. Essentially, if each user receives 1/60 of a second and 12 users are "on" the system, each user will receive service every 1/5 of a second.

The timesharing system performs multiprogramming; that is, it allows several programs to reside in core simultaneously and to operate sequentially. The switching between programs, called context switching, is initiated by a clock which interrupts the central processor to signal that a certain time period has elapsed. The interrupt function is provided by a priority interrupt system. A monitor, also called an operating system or executive program, directs the execution of these tasks and performs other housekeeping duties.

multiprogramming

context switching , **clock**

priority interrupt system

monitor , **operating system**
executive program

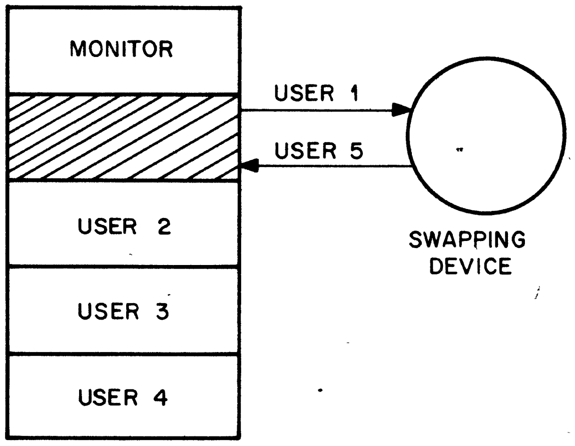
The monitor is also involved in keeping the actions of a user within his assigned memory space. A hardware device, a memory protection register, which is set by the monitor, limits the core area that a particular user can access. Any attempt by the program to read or change information outside that limit will automatically stop the program and notify the monitor.

memory protection register

The system discussed so far services a number of users sequentially in round robin fashion. To increase the number of users serviced, more main memory or core is required. However, since core is expensive, a secondary memory is employed. This memory —usually magnetic disk or drum—is slower than core or main memory but provides greatly increased capacity at reasonable cost. User programs can be located in secondary memory and moved into main memory for execution. Programs entering main memory exchange places with a program (or programs) that has just been serviced by the central processor. This operation is called swapping (see diagram).

main memory
secondary memory

swapping



SWAPPING

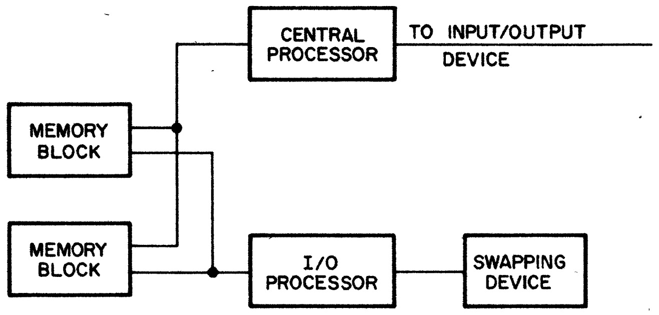
memory blocks

input/output processor

asynchronous design

In operation, main memory is divided into separate memory blocks. Secondary memory is connected to these blocks through a high speed input/output processor—a hardware device that allows the disk or drum to swap a program into any one of the main memory blocks without any aid from the central processor. This structure allows the central processor to be operating a user program in one block of memory while programs are being swapped to and from another block. This independent overlapped operation, which greatly improves efficiency and processing power, is characteristic of an asynchronous system design philosophy. See diagram.

MEMORY STRUCTURE



Dynamic Scheduling

compute bound
I/O bound

scheduling algorithm

queue

Round robin scheduling, in which each program operates in sequence and receives a fixed amount of time, is effective only if all programs have similar requirements. Such is not the case, however. At any particular time, a timesharing system will be handling some programs which require extensive amounts of computing time (and are said to be compute bound) and other programs that must stop frequently for input or output (I/O bound).

To serve programs at and between these two extremes, the scheduling algorithm must provide frequent service to I/O bound programs and must give compute bound jobs longer time quantums to prevent wasteful swapping. A simple dynamic scheme could provide two queues—one for each type of job. When a user first logs on to the system, he is placed in an I/O bound queue (waiting line) where he receives frequent service and small time quantums. If the program isn't completed or does not request input or output during the time allotted to him, the job needs more computing time and is placed in the compute bound queue. Thus the scheduling algorithm optimizes system efficiency by automatically adjusting to program requirements.

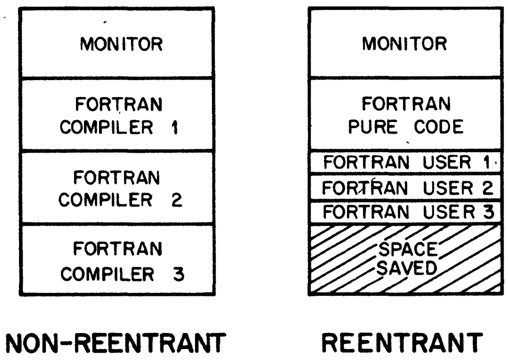
In the present state of scheduling art, algorithms are constantly being changed and improved. Current algorithms are extremely sophisticated, providing excellent service for most timesharing job mixes. They also allow fine tuning, if such modifications are necessary. The ability of the algorithm to match processing to program requirements insures the best service possible for all user programs.

monitor overhead

In an efficient timesharing system, monitor functions (referred to as monitor overhead) take 5 to 10 percent of central processor time, making 90 to 95 percent of the time available to users.

Sharing Software

Since users of large timesharing systems have varying requirements, a good system provides a wide variety of software—interactive languages such as BASIC and AID for the computations of the engineer and scientist, FORTRAN for more complex calculations, COBOL for data processing functions. Therefore many users can have compilers and other common programs in core at the same time.



reentrant software

pure code

impure code

To prevent excessive core usage which results when a program is duplicated for several users, reentrant software is employed. That is, the program is written in two parts. One part contains pure code that is not modified during execution and can be used to simultaneously service any number of users. For example, the pure code portion of FORTRAN can service multiple FORTRAN users. A separate second part of the program belongs strictly to each user and consists of the code and data that is developed during the compiling process (impure code). This section is stored in a separate area of core. A comparison of memory usage in the non-reentrant and reentrant systems is shown in the diagram above.

What are the benefits of reentrant software? First, less core is required. For example, a reentrant system can service three FORTRAN users with one 8K compiler and three 2K user areas, a total of 14K. A non-reentrant system would require 30K for the three 8K compilers and three 2K user areas. Total savings in this case is 16K of core. Using less core means that more programs can fit into a given amount of space. The monitor then swaps less often and spends less time swapping the smaller impure sections.

There are other savings too. Since the pure code never changes, it doesn't have to be returned to disk storage (swapped out). As long as a single copy is maintained on the disk, it can be called into core at any time. Programs can be swapped in or "overlayed" on top of the compiler to take its place in core whenever the compiler is not needed.

overlay

To protect the pure code from being modified, a hardware feature is provided—dual memory protection and relocation. This feature allows a program to execute as two separate segments, one of which is protected. User programs can also be written to make use of this protection. For example, a user might develop a reentrant information retrieval system written in COBOL.

dual memory protection and relocation

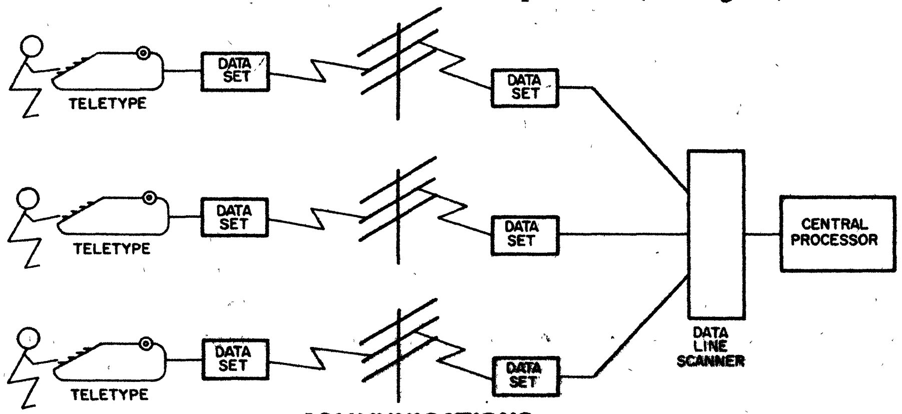
Communications

Communication between the remote user and the computer passes over the conventional dial-up telephone network. User terminals can therefore be located anywhere that phone service is available and connected to any computer system, feasibility limited only by long distance phone rates.

Each user terminal is connected to a data set or modem (modulator-demodulator) which converts user terminal output into a signal suitable for the telephone network. At the computer end of the phone lines, there is another data set which reconverts the signal and feeds it to a device called a data line multiplexor or data line scanner. This device, in turn, feeds the information from a number of terminals to the central processor (see diagram).

data set
modem

data line multiplexor
data line scanner



COMMUNICATIONS

The number of data sets employed at the user end of the system is unlimited. At the computer end of the communications network, however, the number of data sets is limited by the number of users that can be serviced simultaneously by the system.

In order to gain access to the system, the user dials the system phone number from his data set. The telephone network handles the call, scanning the data sets at the computer system. If all of the sets are busy, the user receives a busy signal, just as he would with normal phone service. If a set is available, the telephone network rings it, causing the data line scanner to interrupt the monitor. The computer answers the call, placing the user in communication with the monitor. The terminal is then on-line and ready for operation.

on-line

Control of Input/Output

A timesharing system has performed its basic function if it allows a number of users simultaneous access to a central computer. However, to be fully useful, the system should also allow the users access to other system resources—storage devices for his programs and data, line printers, card readers, etc. For example, the user should be able to choose between magnetic tape and disk for program storage. And if he has a 50-page report to produce, he should be able to employ a line printer instead of his Teletype®. If users controlled these devices, however, much confusion might result. For example, two users might select the line printer at the same time. If one user was processing Abraham Lincoln's Gettysburgh Address and another, Mark Anthony's funeral oration, the report might look like the following:

I COME TO BURY CAESAR NOT TO PRAISE HIM
FOUR SCORE AND SEVEN YEARS AGO
THE EVIL THAT MEN DO LIVE AFTER THEM
OUR FATHERS BROUGHT FORTH ON THIS CONTINENT

To prevent users from interfering with each other, the monitor coordinates input and output (I/O). The processor has an operating mode switch which the monitor sets before a user program is run. If the program attempts to perform input or output, the user program is stopped and the monitor takes over. Control thus diverted to the monitor is called trapping. When input/output is prevented or trapped, the computer is said to be in user mode; when I/O can be performed, the system is in executive or monitor mode.

input/output control

trapping
user mode
executive mode
monitor mode

®—registered trademark of Teletype Corporation, Skokie, Illinois

When the system is in user mode, the memory protection feature is in operation. In monitor mode, this feature is disabled and the monitor has access to all of core. User mode also prevents the user from issuing a HALT command, which could stop operation of the entire system.

User-to-monitor-mode switching occurs when the user requests I/O or other special functions to be performed by the monitor. The requests are made by using computer instructions referred to as monitor calls or programmed operators. For more information see PDP-10 REFERENCE HANDBOOK.

monitor calls
programmed operators

Since I/O is handled by the monitor, input or output can be transferred even if the user program is not in main memory. The monitor can also optimize throughput, keeping all devices busy simultaneously (overlapping of I/O operations) and executing jobs in the most efficient order. For example, it will start the read mechanisms on several disk packs in motion, simultaneously, to reduce the time required to find the desired data on each pack (access time). In addition, by means of the disk pack controller, the monitor can determine which of all needed data on a pack is closest to the read mechanism and can be obtained in the shortest amount of time (latency optimization).

overlapped I/O

access time

latency optimization

File Handling

If a user does not require a fast device for his exclusive use (private device), he can elect to use a public device, in effect performing timesharing with a disk or drum. Under these conditions, user programs and data coexist on the device. Therefore, a filing system is necessary if program and data segments are to be retrieved in proper order.

private device
public device

filing system

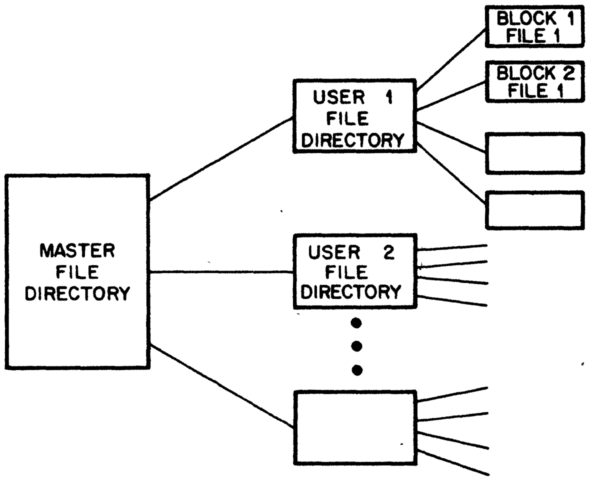
Data is transferred from memory to a peripheral device as a block of words or a record. (A word is the number of binary digits or bits that the central processor can retrieve and "operate on" at one time.) Record length can be arbitrary or dictated by the physical device being used, for example, the number of columns on an 80 column card or on a 132 column line printer. For PDP-10 disk files, the length is 128 words, so that blocks of 128 words are written at one time on a disk or other similar device.

block of words
record

For convenience each user's blocks are organized in groups called files which are listed in proper order in a special block on the disk called the User's File Directory (UFD). A Master File Directory (MFD) is then required to maintain the locations

User's File Directory (UFD)
Master File Directory (MFD)

of the User's File Directories and also keep track of the number of blocks of free storage that can be assigned to new files. The resulting hierarchy is shown in the following diagram.



FILE STRUCTURE

Files, like memory, must be protected from access by unauthorized users. When a user closes a file, he can restrict it, specifying whether others can have access, and if access is permitted, whether the files can be modified or only read. With such an arrangement, programmers in various plant locations can use the same data to work simultaneously on the same project. But unauthorized personnel cannot modify or read the files.

file protection

Slow Peripherals

Fast peripherals can be timeshared. But what about the slow peripherals, such as the line printer and the card reader? Should other users be required to wait 20 minutes or so while one user ties up the line printer?

To eliminate conflicts, the user can request a slow device for his exclusive or private use. For example, he can request the line printer or card reader. Also available for private use are removable storage devices such as magnetic tape, DECtape, (DIGITAL'S low cost, high reliability magnetic tape), or disk packs. If the device is not already assigned to another user, the

removable storage device

monitor grants his request and the user has the device at his disposal until he releases it. For example, the user could request the use of multiple disk pack drives (exclusive use) to sort a payroll transaction file. Or he could assign himself a DECtape drive and ask the system operator to mount the DECtape that contains his own personal library of programs.

spooling
symbiont operation

Spooling or symbiont operation is another method for handling slow peripherals. In this method, the slow device is simulated by a fast peripheral such as a disk. That is, all output for the line printer or card punch is deposited on the disk. The disk is later "unspooled", with a special program transferring information to the slow device.

A program that has data for a slow device thus waits only milliseconds while the data is being deposited on disk, instead of minutes or hours for a turn at the line printer. Input from slow devices can also be spooled, a particularly useful method for batch processing.

Reliability

With a large number of users depending on its operation, the timesharing system must be extremely reliable. A system with 99 percent reliability can be "down" 14 minutes during a 24-hour working day. If that 14 minutes affects only one user, reliability may be acceptable. However, if it affects a large number of users, the consequences are much more serious.

The problem is also complicated by the fact that reliability is a function of both hardware and software. It may take years, for example, to experience all the events that could uncover an error in software as complex as a timesharing monitor.

modularity

defensive software

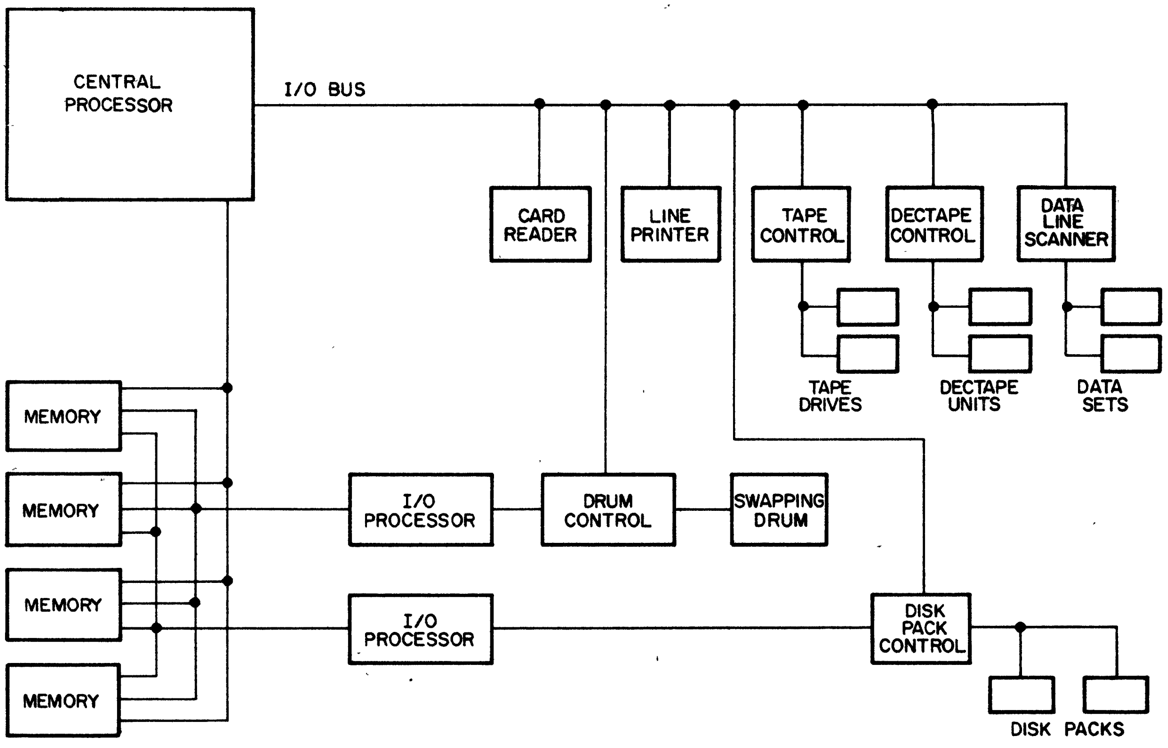
Today's hardware and software has reliability built in. Hardware is designed in modular fashion so that failed components can be removed and new replacements "plugged in". Some components also contain self-testing features that detect potential failures. Software is designed to be "defensive," that is, it anticipates certain types of failures and helps to minimize their effects. For example, the software might note parity errors and limit their effect to the program being operated.

diagnostic software

Diagnostic software can run routinely as one of the timesharing users. Software can also maintain a log of failures, so that patterns can be established and problems remedied before serious damage occurs. Systems that employ these reliability techniques keep downtime at a minimum.

Future of Timesharing

The advanced technology described in these pages is demonstrated by the PDP-10 systems serving timesharing users throughout the world. Typically, one of these large scale systems includes the equipment shown in the accompanying diagram—one or more swapping drums, disk packs for fast storage, magnetic tapes and DECtapes for additional secondary storage. Other peripherals include a line printer, card reader, and plotter. The data line scanner services the desired number of data sets or modems. This equipment, together with the concepts of multiprogramming, reentrant software, and advanced scheduling algorithms, provide excellent service for today's user. But tomorrow's user can expect even more.



TYPICAL PDP-10 TIMESHARING SYSTEM

intelligent terminal

In a new "intelligent" terminal concept, the conventional terminal is replaced by a small computer and peripherals. The small computer will provide local computing capability and, in addition, will have direct access to the central timesharing computer when more power is required. The local system will offer line printers, card readers, and other peripherals as options.

Central processors now under development will be larger, faster, and more powerful, with the ability to serve more users at lower timesharing rates. Hardware will be more sophisticated, implementing more of the monitor's functions.

System reliability and load handling capacity will be improved through greater use of multiprocessor configurations. These configurations allow two or more central processors access to the same memory, mass storage, and peripherals.

As the user will witness, tomorrow's systems will provide better facilities, more power, faster processing, and higher reliability. And with these advances . . . even greater possibilities for new timesharing applications.

Book 2

Getting Started With The Monitor

GETTING STARTED WITH THE MONITOR[1]

2.1 INTRODUCTION

There are basically four phases of programming: (1) writing the program, (2) inputting the program, (3) translating and loading the program, and (4) testing and debugging the program. Since the computer must be instructed in order to know what to do, the first phase is writing the program and supplying data for that program. The program may be written in a programming language that the computer is preconditioned to understand, such as BASIC, AID, COBOL, FORTRAN. A program written in the symbolic notation of one of these languages is called the source program. In the second phase of programming, the source program is inputted into the computer and stored on the disk. Although there are several ways of inputting the source program into the computer (e.g., tapes, cards), the Teletype as the device used for input and output is the main concern of this section. (See Book 7, Advanced Monitor Commands, for a discussion of other input and output devices) In the third phase, the source program is translated by the computer into a binary machine language program, and this binary program is loaded into core memory to form the core image of the translated source program.
Ideally, a program should run correctly the first time, but in reality, this is not the case. A program may contain errors of many types, ranging from simple errors in typing to complex errors in the logical design of the program. Therefore, the fourth phase of programming is program testing and debugging. When errors are found, corrections are made to the source program still on the disk. The sequence of program testing and debugging is repeated until the program runs properly.

Programs are typed directly into the computer by means of the Teletype, a typewriter-like console. By typing in programs, you establish communication with other programs already resident in the computer. The first resident program you communicate with is the time-sharing

[1]We wish to express appreciation to Stanford University for the use of their Stanford A-I Project User's Manual, Chapter I, SAILON No. 54, as a guide in writing the material in this section.

monitor, the most important program in the computer. (The terms monitor and system are used interchangeably to mean the time-sharing monitor.) The monitor is the master program that plays an important role in the efficient operation of the computer. Just as the Teletype is your link with the computer, the monitor is your link with the programs within the computer.

The monitor has many functions to perform, like keeping a record of what each user is doing and deciding what user should be serviced next and for how long. The one function of the monitor that is of greatest concern at this point is that the monitor retrieves any resident programs that you need. This retrieval happens only if the monitor "understands" what is expected of it. The commands to the monitor which are explained in this chapter are sufficient for the Teletype to be the device by which information is inputted into the system and by which the system outputs its results.

See section 2.10 for a discussion on How to Live With the Teletype.

2.2 GETTING ON THE SYSTEM

In order to gain access to the time-sharing system, you must say hello to the system by "logging in". The first move is to make contact with the computer facility by whatever means the facility has established (e.g., acoustic coupler, telephone, or dataphone). Next, notice the plastic knob (the power switch) on the lower right-hand side of the Teletype. This knob has three positions: on, off, and local (turning clockwise). When the knob is in the local position, the Teletype is like a typewriter; it is not communicating with the system at all. The knob must be turned to the on position in order to establish communication with the computer. When the Teletype is turned on, type a ↑C (depress the CTRL key and type C). This action establishes communication with the monitor. The monitor signifies its readiness to accept commands by responding with a period (.). All the commands discussed in this chapter can only be typed to the monitor. They are operative when the monitor has typed a period, signifying that it is waiting for a command.

The first program the monitor should call in for you is the log-in program. This is accomplished by typing LOGIN followed by a carriage-return (depress the RETURN key). All commands to the monitor are terminated with a carriage-return. When the monitor "sees" a carriage-return, it knows that a command has been typed and it begins to execute the command.

In the text, underscoring is used to designate Teletype output.
A carriage-return is designated by a ↵

By typing LOGIN, you cause the monitor to read the login program from the disk into core memory and it is this program that is now in control of your Teletype. Before the login program is called in, the monitor assigns you a job number for system bookkeeping purposes. The system responds with an information message similar to the following.

```
JOB 17    4SP74G
#
```

In the first line, the system has assigned your job number (17) and has given the name of the monitor and its version number. This version number changes whenever a change, or patch, is incorporated into the monitor. In the second line, the number sign (#), which is typed out by the login program, signifies that it wants your identification.

The standard identification code is in the form of project numbers and programmer numbers, but individual installations may have different codes. The numbers, or whatever code each installation uses, are assigned to each user by the installation. The login program waits for you to type in your project number and your programmer number, separated by a comma and terminated with a carriage-return, following the number sign.

```
JOB 17    4SP74G
#27,400 ↵
```

The login program needs one more item to complete its analysis of your identification. This it requests in the next line by asking for your password.

```
JOB 17    4SP74G
#27,400 ↵
PASSWORD:       ↵
```

Type in your password, which is also assigned by the installation, followed by a carriage-return. To maintain password security, the login program does not print the password.

If the identification typed in matches the identification stored in the accounting file in the monitor, the login program signifies its acceptance by responding with the time, date, your Teletype number, the message of the day (if any), and a period.

```
JOB 17    4SP74G
#27,400 ↵
PASSWORD:       ↵
1050   4-MAY-70   TTY9
COBOL IS NOW AVAILABLE ON THE SYSTEM
.
```

This typeout indicates that the login program has exited and returned control to the monitor. You have successfully logged in and may now have the monitor call in other programs for you. If the identification typed in does not match the identification in the accounting file, the monitor types out the error message

```
?INVALID ENTRY-TRY AGAIN
#
```

If this error message occurs, type in the correct project-programmer numbers and password.

2.3 FILES

When you want to run a program, first type in the program and decide on a name for it. The program is stored on the disk with the specified name. Then translate the program by calling in a translator and giving it the name of the program you wish to translate.

A program, or data, is stored on the disk in files. If a program is being typed in to a text editor (for example, TECO), the editor is busy accepting the characters being typed in and generating a disk file for them. Then, when the program is to be translated, the translator reads this file just created and generates a relocatable binary file. Since you may have many files and the other users on the computer may have files, there must be a method for keeping all of these files separate. This is accomplished by giving each user a unique area on the disk. This area is identified by your project and programmer numbers. For example, if your project and programmer numbers are 27,400, you have a disk area by that name. Each file you create goes to your disk area and must be uniquely named.

Files are named with a certain convention, the same as a person is named. The first name, the filename, is the actual name of the file, and the last name, the filename extension, indicates what group the file is associated with. The filename and the filename extension are separated by a period.

Filenames are from one to six letters or digits. All letters or digits after the sixth are ignored. The filename extension is from one to three letters or digits. It is generally used to indicate file format. The following are examples of standard filename extensions.

.TMP	Temporary file
.MAC	Source file in MACRO language
.F4	Source file in FORTRAN IV language
.BAS	Source file in BASIC language
.CBL	Source file in COBOL language
.REL	Relocatable binary file
.SAV	A saved core image

Since files are identified by the complete name and the project and programmer numbers, two users may use the same filename as long as they have different project and programmer numbers; the files would be distinct and separate. The following are examples of filenames with filename extensions.

MAIN.F4	A FORTRAN file named MAIN
SAMPLE.BAS	A BASIC file named SAMPLE
TEST1.TMP	A temporary file named TEST1
NAME.REL	A relocatable binary file named NAME

2.4 CREATING FILES[1]

The two commands mentioned in this section use two editors to create a new disk file. One of the editors is LINED, a disk-oriented editor, and the other is TECO, the Text Editor and Corrector (see Book 8 for discussion of both editors). Each command requires a filename as its argument and should have a filename extension. A new file may be created with either of these commands, depending on the editor desired. If line numbers are desired, LINED should be used, since TECO generates a non-sequence numbered file.

2.4.1. The CREATE Command

The CREATE command is used only to create a new disk file. When this command is executed, the monitor calls in LINED to initialize a disk file with the specified name and to accept input from the Teletype. At this point, begin to type in your program, line by line. LINED types a line number at the beginning of each line so that later a reference to a given line may be made in order to make corrections. Below is a sample program using the commands discussed so far.

[1] A BASIC or AID user does not need the following sections. These two compilers have built-in facilities to create and edit files. See Book 3 for BASIC and Book 4 for AID.

↑C	Establish communication with the monitor. Type C while depressing the CTRL key.
.LOGIN ↵	Begin the login procedure.
JOB 17 4SP74G # 	Job number assigned, followed by monitor name and version. Login program requests identification (project number and programmer number).
27,400 ↵	Type in project-programmer number.
PASSWORD: ↵	Login program requests password. Type it in; it is not printed.
1050 4-MAY-70 TTY9 COBOL IS NOW AVAILABLE ON THE SYSTEM .	If identification matches identification stored in the system, the monitor responds with the time, date, Teletype number, message of the day, and a period.
CREATE MAIN.F4 ↵	A new file on the disk is to be created and called MAIN.F4. The extension .F4 is used because the program is to be a FORTRAN source file. LINED is called in to create the file.
*	Response from LINED signifying it is ready to accept commands.
I ↵	A command to LINED to insert line numbers starting with 10 and incrementing by 10 (see Book 8).
00010 TYPE 53 ↵	Type in your FORTRAN PROGRAM.
00020 53 FORMAT (' THIS IS MY PROGRAM') ↵	
00030 END ↵	
00040 $	The ($) (altmode) is a command to LINED to end the insert. On the Teletype this key is labeled ALT, ESC, or PREFIX.
*	Response from LINED signifying it is ready to accept another command.
E ↵	A command to LINED to end the creation of the file.
*	Response from LINED indicating readiness to accept a command.
↑C	Return to the Monitor.
.	The monitor now has control of the program

The three LINED commands (I, altmode, E) shown in the example are fully discussed in Book 8.

2.4.2 The MAKE Command

This command can also be used to open a new disk file for creation. It differs from the CREATE command in that TECO is used instead of LINED. (TECO is discussed in Book 8.) Otherwise, the CREATE and MAKE commands operate in the same manner.

```
.MAKE    FILEA.F4 ↲
* I (Text input)$$
EX$$
EXIT
↑C

.
```

The altmode ($) and the EX command are commands to TECO and are explained in the TECO section of Book 8.

2.5 EDITING FILES

After creating a text file, you may wish to modify, or edit, it. The following two commands cause an existing file to be opened for changes. One command (EDIT) calls in LINED, and the other (TECO) calls in TECO. In general, the editor used to create the file should be used for editing. Each command requires, as its argument, the same filename and filename extension used to create the file.

2.5.1 The EDIT Command

The EDIT command causes LINED to be called in and, as the name implies, signifies that you wish to edit the specified file. LINED responds with an asterisk and waits for input. The file specified must be an already existing sequence-numbered file on the disk. For example, in section 2.4.1, the file MAIN.F4 was created. If the command

```
. EDIT MAIM.F4 ↲
```

is given to edit the file, the computer responds with an error message (assuming that there was no file named MAIM.F4). The command

```
.EDIT MAIN.F4 ↲
```

causes the right file to be opened for editing.

2.5.2 The TECO Command

The TECO command is similar to the EDIT command except that it causes the TECO program to open an already existing non-sequence-numbered file on the disk for editing purposes. The command sequence

```
.TECO    FILEA.F4 ⤸
*(editing)$$
*EXIT$$
```

causes TECO to open FILEA.F4 for editing and close the file upon completion, creating a backup file out of the original file. Whenever one of the commands used to create or edit a file is executed, this command with its arguments (filename and filename extension) is "remembered" as a temporary file on the disk. Because of this, the file last edited may be recalled for the next edit without having the filename specified again. For example, if the command

```
.CREATE   PROG1.MAC ⤸
```

is executed, then you may type the command

```
.EDIT ⤸
```

instead of

```
.EDIT    PROG1.MAC ⤸
```

assuming that no other CREATE, TECO, MAKE, or EDIT command was used in-between. As mentioned before, if a command tries to edit a file that has not been created, an error message is given.

2.6 MANIPULATING FILES

You may have many files saved on your disk area. (For discussion on how to save a file on your disk area, see Book 7.) The list of your files, along with lists of other users' files, is kept on the disk in what are called user directories. Suppose you cannot remember if you have created and saved a particular file. The next command helps in just that type of situation.

2.6.1 The DIRECTORY Command

The DIRECTORY command requests from the monitor a listing of the directory of your disk area. The monitor responds by typing on the Teletype the names of your files, the date on which each file was created, and the length of each file in PDP-10 disk blocks. A disk block consists of 128_{10} PDP-10 words. Names of files not explicitly created by you may show up in the directory. These files were created as intermediate files for storage by programs you may have used. For example, in translating a file, the translator generates a file with the same filename but with a filename extension of .REL. This file contains the relocatable binary translation of the source file. You may also notice filenames with the filename extension of .TMP. This extension signifies a temporary file created and used by different CUSPs.

2.6.2 The TYPE Command

By listing your directory on the Teletype, you know the names of the files on your disk area. But what if you have forgotten the information contained in a particular file? The TYPE command causes the contents of source files specified in your command string to be typed on your Teletype. For example, the command

```
.TYPE   MAIN.F4 ⤸
```

causes the file MAIN.F4 to be typed on the Teletype. Multiple files separated by commas may be specified in one command string, and only source files, not binary files, may be listed.

This command allows the "asterisk construction" to be used. This means that the filename or the filename extension may be replaced with an asterisk to mean any filename or filename extension. For example, the command

```
.TYPE    FILEB.* ⤸
```

causes all files named FILEB, regardless of filename extensions, to be typed. The command

```
.TYPE   * . MAC ⤸
```

causes all files with the filename extension of .MAC to be typed. The command

```
. TYPE * . * ↵
```

causes all files to be typed.

2.6.3 The DELETE Command

Having finished with a file, you may erase it from your disk area with the DELETE command. Multiple files may be deleted in one command string by separating the files with commas. For example,

```
. DELETE LINEAR ↵
```

and

```
. DELETE CHANGE.F4, SINE.REL ↵
```

are both legal commands. The asterisk convention discussed in section 2.6.2 may also be used with the DELETE command.

2.6.4 The RENAME Command

The names of one or more files on your disk area may be changed with the RENAME command. The old filename on the right and the new filename on the left are separated by an equal (=) sign. In renaming more than one file, each pair of filenames (new=old) is separated by commas. For example, the command

```
. RENAME SALES.CBL=GROSS.CBL,FILE2.F4=FILE1.F4 ↵
```

changes the name of file GROSS.CBL to SALES.CBL and file FILE1.F4 to FILE2.F4. The old filename no longer appears in your directory; instead the new filenames appear containing exactly the same data as in the old files. The asterisk convention may again be used. For example, the command

```
. RENAME    * . F4= *  ↵
```

causes all files with no filename extension to have the extension .F4.

2.7 TRANSLATING, LOADING, EXECUTING, DEBUGGING PROGRAMS

At this point you know how to get on the system, how to create and edit a source file of a program, and how to list your source file on the Teletype. The program has not been executed. This only happens after it has been translated into the binary machine language understandable to the computer and loaded into core memory. More often than not the program must be debugged.

2.7.1 The COMPILE Command

This command has as its argument one or more filenames separated by commas. It causes each command to be processed (translated) if necessary by the appropriate processor (translator). It is considered necessary to process a file if no .REL file of the source file exists, or if the .REL file was created before the last time the source file was edited. If the .REL file is up-to-date, no translation is done. The appropriate processor is determined by examining the extension of the file. The following shows which processor is used for various extensions.

. MAC	MACRO assembler
. F4	FORTRAN IV compiler
. CBL	COBOL compiler
. REL	No processing is done
other than above, or null	"Standard processor"

The standard processor is used to translate programs with null or nonstandard extensions. The standard processor is FORTRAN at the beginning of the command string, but may be changed by use of various switches (See the PDP- 10 Reference Handbook, Communication With the Monitor). Although it is not necessary to indicate the extension of a file in the COMPILE command string, the standard processor can be disregarded if all source files are kept with the appropriate extension.

When the appropriate translator has translated the source file, there is a file on your disk area with the filename extension .REL and the same filename as the source file. This file is where the translator stores the results of its translation and is called the <u>relocatable binary</u> of the program. The program is now translated into binary machine language, but is still on the disk. Since the disk is used for storage and not for execution, a copy of the binary program must be loaded into core memory to form a <u>core image</u>. The core memory of the computer is used for execution; it is like a scratch pad. The COMPILE command does not generate a core image, but the following three commands do.

2.7.2 The LOAD Command

The LOAD command performs the same operations as the COMPILE command and in addition causes the Linking Loader to be run. The Linking Loader is a resident program that takes the specified REL files, links them together, and generates a core image. The LOAD command does not cause execution of the program.

2.7.3 The EXECUTE Command

This command performs the functions of the LOAD command and also begins execution of the loaded programs , if no translation or loading errors are detected. The compiled program is now in core memory and running, and what happens next depends on the program. More than likely, the program is not returning the correct answers, and you now enter the magic world of program debugging.

2.7.4 The DEBUG Command

This command prepares for the debugging of a program in addition to performing the functions of the COMPILE and LOAD commands. DDT, the Dynamic Debugging Technique program (see the DDT section in the PDP- 10 Reference Handbook), is loaded into core memory first, followed by the program. Upon completion of loading, DDT is started rather than the program. A command to DDT may then be issued to begin the program execution. This command should be used by the experienced programmer familiar with DDT. The above four commands have extended command forms discussed in the PDP- 10 Reference Handbook.

The following is an example showing the compilation and execution of a FORTRAN main program and subroutine. The login procedure is not shown.

```
.  CREATE   MAIN.F4 ⤸          CREATE a disk file
*I ⤸                           Command to LINED to begin inserting on
                               line 10, incrementing by 10
00010    TYPE 69 ⤸             Statements of the FORTRAN main program
00020 69  FORMAT ('  THIS IS THE MAIN PROGRAM') ⤸
00030     CALL SUB1 ⤸
00040     END ⤸
00050     $                    Altmode ends the insert
*E ⤸                           LINED command to end the edit
* ↑C                           Return to the Monitor
.  CREATE SUB1.F4 ⤸            Create a disk file for the subroutine
*I ⤸                           Begin inserting at line 10 incrementing by 10
```

```
00010       SUBROUTINE SUBR ↵    Statements of the FORTRAN Subroutine
00020       TYPE 105 ↵
00030    105  FORMAT (' THIS IS SUB1') ↵
00040       RETURN ↵
00050       $                 Altmode ends the insert
* E ↵                         LINED command to end the edit.
* ↑C                          Return to Monitor
. EXECUTE MAIN.F4,SUBI,F4 ↵  Request execution of the programs created
FORTRAN:   MAIN.F4            FORTRAN reports its progress
FORTRAN:    SUB1.F4
LOADING
000001 UNDEFINED GLOBALS
SUB1     000152               There is no subroutine named SUB1
?
LOADER    3K   CORE           This includes the space for the loader.
? EXECUTION DELETED           No execution was done
EXIT
↑C
. EDIT ↵                      Ask to edit SUB1.F4, filename need not be
                              mentioned since it was the last file edited.
* P10,20 ↵                    Type lines 00010 and 00020 on the Teletype.
00010     SUBROUTINE SUBR
00020     TYPE 105
* I10 ↵                       Insert a new line 10
00010'   SUBROUTINE SUB1 ↵
* E ↵
* ↑C
. EXECUTE MAIN.F4,SUB1.F4 ↵  Request execution
FORTRAN:    SUBI.F4           Only the subroutine is recompiled since only
                              it has been edited.
LOADING                       Both MAIN and SUB1 are loaded
LOADER  3K CORE
EXECUTION
THIS IS THE MAIN PROGRAM      Execution  begins
THIS IS SUB1
EXIT
↑C                            Execution ends
.
```

2.8 GETTING INFORMATION FROM THE SYSTEM

There are several monitor commands that are used to obtain information from the system. Three commands useful at this point are discussed in this section, and additional commands are discussed in Book 7, Advanced Monitor Commands.

2.8.1 The PJOB Command

If you have forgotten the job number assigned to you at login time, you may use the PJOB command to obtain it. The system responds to this command by typing out your assigned job number. For example,

```
.PJOB ⤸
17
```

2.8.2 The DAYTIME Command

This command gives the date followed by the time of day. The time is presented in the following format:

```
hh:mm
```

where hh represents the hours and mm represents the minutes. For example,

```
.DAYTIME ⤸
17-JUNE-70   14:37
```

2.8.3 The TIME Command

The TIME command produces three lines of typeout. The first line is the total running time since the last TIME command was typed. The second line is the total running time since you logged in. The third line is used for accounting purposes. The time is presented in the following format:

```
hh:mm:ss.hh
```

where hh represents the hours, mm the minutes, and ss.hh the seconds to the nearest hundredth. For example,

```
.TIME )
52.45
02:29.95
KILO-CORE-SEC=57
```

In the first two lines, you are told that you have been running 52.45 seconds since the last time you typed the TIME command, and a total of 2 minutes and 29.95 since you logged in. The third line of typeout is used by your installation for accounting and is the integrated product of running time and core size. See the PDP-10 Reference Handbook, Communicating With the Monitor.

2.9 LEAVING THE SYSTEM

Now that you know how to log into the system and create and run a program, you might be wondering how you leave the system. You have to tell the system you are leaving, and you do this by the KJOB command.

2.9.1 The KJOB Command

The KJOB command is your way of saying goodbye to the system. Many things happen when you type the command. The job number assigned to you is released and your Teletype is now free for another user. An automatic TIME command is performed. In addition, if you have any files on your disk area, the monitor responds with

```
CONFIRM:
```

and you have several options available to you. By typing a carriage-return after the CONFIRM: message, the monitor lists the options available. For example, the following typeout occurs by responding to the confirm message with a carriage return.

```
TYPE ↑C TO ABORT LOG-OUT; OR
TYPE ONE OF THE FOLLOWING ( AND CAR RET):
K   TO KILL JOB AND DELETE ALL UNPROTECTED FILES;
L   TO LIST YOUR DISK DIRECTORY;  OR
I   TO INDIVIDUALLY SAVE AND DELETE FILES AS FOLLOWS:
              AFTER EACH FILE NAME IS LISTED, TYPE:
              P   TO SAVE AND PROTECT,
                    S  TO SAVE WITHOUT PROTECTING, OR
                    CAR RETURN ONLY TO DELETE.
      CONFIRM:
```

You may now use the options available. If K was used as the option, the following is a sample of what is outputted to your Teletype.

```
JOB 33, USER [27,560] LOGGED OFF TTY34    1317 20-FEB-70
DELETED ALL 2 FILES (INCLUDING UFD, 3. DISK BLOCKS)
RUNTIME 0 MIN, 00.29 SEC
```

Remember that the CONFIRM message is typed only if there are files on your disk area. If there are no files on your disk area, the typeout would look like the following:

```
. KJOB ⤸
JOB 17, USER [27,3201] LOGGED OFF TTY17  1317 20-FEB-70
RUNTIME 0 MIN, 00.29 SEC
```

2.10 HOW TO LIVE WITH THE TELETYPE

On the Teletype, there is a special key marked CTRL called the Control Key. If this key is held down and a character key is depressed, the Teletype types what is known as a control character rather than the character printed on the key. In this way, more characters can be used than there are keys on the keyboard. Most of the control characters do not print on the Teletype, but cause special functions to occur, as described in the following sections.

There are several other special keys that are recognized by the system. The system constantly monitors the typed characters and, most of the time, sends the characters to the program being executed. The important characters not passed to the program are also explained in the following sections. (See also the PDP-10 Reference Handbook, Communicating With the Monitor.)

2.10.1 Control - C

Control - C (↑C) interrupts the program that is currently running and takes you back to the monitor. The monitor responds to a control - C by typing a period on your Teletype, and you may then type another monitor command. For example, suppose you are running a program in BASIC, and you now decide you want to leave BASIC and run a program in AID. When BASIC requests input from your Teletype by typing an asterisk, type control - C to terminate BASIC and return to the monitor You may now issue a command to the monitor to initialize AID (.R AID). If the program is not requesting input from your Teletype (i.e., the program is in the middle of execution) when you type control - C, the program is not stopped immediately. In this case, type control - C twice in a row to stop the execution of the program and return control to the monitor. If you wish to continue at the same place that the program was interrupted, type the monitor command CONTINUE. As an example, suppose you want the computer to add a million numbers and to print the square root of the sum. Since you are charged by the amount of processing time your program uses, you want to make sure your program does not take an unreasonable amount of processing time to run. Therefore, after the computer has begun execution of your program, type control - C twice to interrupt your program. You are now communicating with the monitor and may issue the monitor command TIME to find out how long your program has been running. If you wish to continue your program, type CONTINUE and the computer begins where it was interrupted.

2.10.2 The RETURN Key

This key causes two operations to be performed: (1) a carriage-return and (2) an automatic line-feed. This means that the typing element returns to the beginning of the line (carriage-return) and that the paper is advanced one line (line-feed). Commands to the monitor are terminated by depressing this key.

2.10.3 The RUBOUT Key

The RUBOUT key permits correction of typing errors. Depressing this key once causes the last character typed to be deleted. Depressing the key n times causes the last n characters typed to be deleted. RUBOUT does not delete characters beyond the previous carriage-return, line-feed, or altmode. Nor does RUBOUT function if the program has already processed the characters you wish to delete.

The monitor types the deleted characters, delimited by backslashes. For example, if you were typing CREATE and go as far as CRAT, you can correct the error by typing two RUBOUTS and then the correct letters. The typeout would be

```
CRAT\TA\EATE
```

Notice that you typed only two RUBOUTS, but \TA\ was printed. This shows the deleted characters, but in reverse order.

2.10.4 Control - U

Control - U (↑U) is used if you have completely mistyped the current line and wish to start over again. Once you type a carriage-return, the command is read by the computer, and line-editing features can no longer be used on that line. Control - U causes the deletion of the entire line, back to the last carriage-return, line-feed, or altmode. The system responds with a carriage-return, line-feed so you may start again.

2.10.5 The ALTMODE Key

The ALTMODE key, which is labeled ALTMODE, ESC, or PREFIX, is used as a command terminator for several programs, including TECO and LINED. Since the ALTMODE is a non-printing character, the Teletype prints an ALTMODE as a dollar sign ($).

2.10.6 Control - O

Control - O (↑O) tells the computer to suppress Teletype output. For example, if you issue a command to type out a 100 lines of text and then decide that you do not want the type-out, type control - O to stop the output. Another command may then be typed as if the typeout had terminated normally.

2.10.7 Control - B

Control - B (↑B) affects the printing of Teletype output in one of two ways depending on your Teletype; it either restores printing of the characters or suppresses double printing of the characters. Suppose that when you begin typing on the Teletype, you notice that the characters you are typing are not printing on the Teletype paper. Type control - B to restore the printing of the characters. On the other hand, suppose you receive double printing of your typeins. To suppress this double printing, type control - B.

2.10.8 Control - F

This control character is needed only for the KSR37 Teletype. This key changes the way lower case characters are handled by the system. Normally, the system converts all lower case letters to upper case. Since the KSR37 Teletype is capable of transmitting both lower and upper case letters, control - F is used to permit the entry of lower case letters.

Book 3

Conversational Programming With BASIC

CONTENTS

CHAPTER 1 INTRODUCTION		Page
1.1	Example of Basic Program	3-9
1.2	Discussion of the Program	3-10
1.3	Fundamental Concepts of Basic	3-13
1.3.1	Arithmetic Operations	3-13
1.3.2	Mathematical Functions	3-14
1.3.3	Numbers	3-15
1.3.4	Variables	3-15
1.3.5	Relational Symbols	3-15
1.4	Summary	3-16
1.4.1	LET Statement	3-16
1.4.2	READ and DATA Statements	3-16
1.4.3	PRINT Statement	3-17
1.4.4	GO TO Statement	3-18
1.4.5	IF - THEN Statement	3-18
1.4.6	ON - GO TO Statement	3-19
1.4.7	END Statement	3-19
CHAPTER 2 LOOPS		
2.1	FOR and NEXT Statements	3-22
2.2	Nested Loops	3-23
2.3	Summary	3-23
2.3.1	FOR and NEXT Statements	3-23
CHAPTER 3 LISTS AND TABLES		
3.1	The Dimension Statement (DIM)	3-26
3.2	Example	3-27
3.3	Summary	3-28
3.3.1	The DIM Statement	3-28
CHAPTER 4 HOW TO RUN BASIC		
4.1	Gaining Access to Basic	3-29
4.2	Entering the Program	3-30

		Page
4.3	Executing the Program	3-31
4.4	Correcting the Program	3-31
4.5	Interrupting the Execution of the Program	3-31
4.6	Leaving the Computer	3-32
4.7	Example of Basic Run	3-32
4.8	Errors and Debugging	3-34
4.8.1	Example of Finding and Correcting Errors	3-34

CHAPTER 5 FUNCTIONS AND SUBROUTINES

5.1	Functions	3-37
5.1.1	The Integer Function (INT)	3-37
5.1.2	The Random Number Generating Function (RND)	3-38
5.1.3	The RANDOMIZE Statement	3-39
5.1.4	The Sign Function (SGN)	3-40
5.1.5	The Define User Function (DEF) and Function End Statement (FNEND)	3-40
5.2	Subroutines	3-41
5.2.1	GOSUB and RETURN Statements	3-41
5.2.2	Example	3-42

CHAPTER 6 MORE SOPHISTICATED TECHNIQUES

6.1	More About the Print Statement	3-45
6.2	Input Statement	3-48
6.3	Stop Statement	3-49
6.4	Remarks Statement (REM)	3-49
6.5	Restore Statement	3-49

CHAPTER 7 VECTORS AND MATRICES

7.1	MAT Instruction Conventions	3-52
7.2	MAT C = ZER, MAT C = CON, MAT C = IDN	3-52
7.3	MAT Print A, B, C	3-53
7.4	MAT Input V and the NUM Function	3-54
7.5	MAT B = A	3-55
7.6	MAT C = A + B and MAT C = A - B	3-55

CONTENTS (Cont)

		Page
7.7	MAT C = A * B	3-55
7.8	MAT C = TRN(A)	3-55
7.9	MAT C = (K) * A	3-55
7.10	MAT C = INV(A) and the DET Function	3-55
7.11	Examples of Matrix Programs	3-56
7.12	Simulation of N-Dimensional Arrays	3-57

CHAPTER 8 ALPHANUMERIC INFORMATION (STRINGS)

8.1	Reading and Printing Strings	3-59
8.2	String Conventions	3-60
8.3	Numeric and String Data Blocks	3-61
8.4	Accessing Individual Characters	3-61

CHAPTER 9 EDIT AND CONTROL 3-65

CHAPTER 10 DATA FILE CAPABILITY

10.1	FILES Command	3-67
10.2	SCRATCH Command	3-68
10.3	WRITE Command	3-68
10.4	RESTORE Command	3-69
10.5	INPUT Command	3-69
10.6	IF END Command	3-70

WHY BASIC? BASIC is a problem-solving language that is easy to learn and conversational, and has wide application in the scientific, business, and educational communities. It can be used to solve both simple and complex mathematical problems from the user's Teletype® and is particularly suited for time-sharing.

In writing a computer program, it is necessary to use a language or vocabulary that the computer recognizes. Many computer languages are currently in use, but BASIC is one of the simplest of these because of the small number of clearly understandable and readily learned commands that are required, its easy application in solving problems, and its practicality in an evolving educational environment.

BASIC is similar to other programming languages in many respects; and is aimed at facilitating communication between the user and the computer in a time-sharing system. As with most programming languages, BASIC is divided into two sections:

a. Elementary statements that the user must know to write simple programs, and
b. Advanced techniques needed to efficiently organize complicated problems.

As a BASIC user, you type in a computational procedure as a series of numbered statements by using common English syntax and familiar mathematical notation. You can solve almost any problem by spending an hour or so learning the necessary elementary commands. After becoming more experienced, you can add the advanced techniques needed to perform more intricate manipulations and to express your problem more efficiently and concisely. Once you have entered your statements via the Teletype, simply type in RUN or RUNNH. These commands initiate the execution of your program and return your results almost instantaneously.

SPECIAL FEATURES OF BASIC - BASIC incorporates the following special features:

a. Matrix Computations - A special set of 13 commands designed exclusively for performing matrix computations.

b. Alphanumeric Information Handling - Single alphabetic or alphanumeric strings or vectors can be read, printed, and defined in LET and IF...THEN statements. Individual characters within these strings can be easily accessed by the user. Conversion can be performed between characters and their ASCII equivalents. Tests can be made for alphabetic order.

® Teletype is the registered trademark of Teletype Corporation.

c. Program Control and Storage Facilities - Facilities are included that store programs or data on a mass storage device (e.g., disk or DECtape) and later retrieve them for execution. You, as the user, can also input programs from the standard low-speed Teletype paper tape reader as well as from the high-speed paper tape reader at the PDP-10 site.

d. Program Editing Facilities - An existing program can be edited by adding or deleting lines, by renaming the program, or by resequencing the line numbers. The user can combine two programs into a single program and request a listing of his program, either in whole or in part, on his Teletype or on a high-speed line printer.

e. Formatting of Output - Controlled formatting of Teletype output includes tabbing, spacing, and printing columnar headings.

f. Documentation and Debugging Aids - Documenting programs by the insertion of remarks within procedures enables recall of needed information at some later date and is invaluable in situations in which the program is shared by other users. Debugging of programs is aided by the typeout of meaningful diagnostic messages which pinpoint syntactical and logical errors detected during execution.

CHAPTER 1
INTRODUCTION

This chapter introduces the user to PDP-10 BASIC and to its restrictions and characteristics. The best introduction lies in beginning with a BASIC program and discussing each step completely.

1.1 EXAMPLE OF A BASIC PROGRAM

The following example is a complete BASIC program, named LINEAR, that can be used to solve a system of two simultaneous linear equations in two variables

$$ax + by = c$$
$$dx + ey = f$$

and then used to solve two different systems, each differing from the above system only in the constants c and f. If ae - bd is not equal to 0, this system can be solved to find that

$$x = \frac{ce - bf}{ae - bd} \quad \text{and} \quad y = \frac{af - cd}{ae - bd}$$

If ae - bd = 0, there is either no solution or there are many, but there is no unique solution. Study this example carefully and then read the commentary and explanation. (In most cases the purpose of each line in the program is self-evident.)

```
10      READ A,B,D,E
15      LET G=A*E-B*D
20      IF G=0 THEN 65
30      READ C,F
37      LET X=(C*E-B*F)/G
42      LET Y=(A*F-C*D)/G
55      PRINT X,Y
60      GO TO 30
65      PRINT "NO UNIQUE SOLUTION"
70      DATA 1,2,4
80      DATA 2,-7,5
85      DATA 1,3,4,-7
90      END
```

NOTE

All statements are terminated by pressing the RETURN key (represented in this text by the symbol ↵). The RETURN key echoes as a carriage return, line feed.

1.2 DISCUSSION OF THE PROGRAM

Each line of the program begins with a line number of 1 to 5 digits that serves to identify the line as a statement. A program is made up of statements, most of which are instructions to the computer. Line numbers serve to specify the order in which these statements are to be performed. Before the program is run, BASIC sorts out and edits the program, putting the statements into the order specified by their line numbers; thus, the program statements can be typed in any order, as long as each statement is prefixed with a line number indicating its proper sequence in the order of execution. Each statement starts after its line number with an English word which denotes the type of statement. Spaces have no significance in BASIC, except in messages which are printed out, as in line number 65 above. Thus, spaces may or may not be used to modify a program and make it more readable.

With this preface, the above example can be followed through step-by-step.

```
        1Ø        READ A,B,D,E
```

The first statement, 10, is a READ statement and must be accompanied by one or more DATA statements. When the computer encounters a READ statement while executing a program, it causes the variables listed after the READ to be given values according to the next available numbers in the DATA statements. In this example, we read A in statement 10 and assign the value 1 to it from statement 70 and, similarly, with B and 2, and with D and 4. At this point, the available data in statement 70 has been exhausted, but there is more in statement 80, and we pick up from it the value 2 to be assigned to E.

```
        15        LET G=A*E-B*D
```

Next, in statement 15, which is a LET statement, a formula is to be evaluated. [The asterisk (*) is used to denote multiplication.] In this statement, we compute the value of AE - BD, and call the result G. In general, a LET statement directs the computer to set a variable equal to the formula on the right side of the equal sign.

```
        2Ø        IF G=Ø THEN 65
```

If G is equal to zero, the system has no unique solution. Therefore, we next ask, in line 20, if G is equal to zero.

```
65      PRINT "NO UNIQUE SOLUTION"
70      DATA 1,2,4
80      DATA 2,-7,5
85      DATA 1,3,4,-7
90      END
```

If the computer discovers a "yes" answer to the question, it is directed to go to line 65, where it prints NO UNIQUE SOLUTION. Since DATA statements are not executable statements, the computer then goes to line 90 which tells it to END the program.

```
30      READ C,F
```

If the answer is "no" to the question "Is G equal to zero?", the computer goes to line 30. The computer is now directed to read the next two entries, -7 and 5, from the DATA statements (both are in statement 80) and to assign them to C and F, respectively. The computer is now ready to solve the system

$$x + 2y = -7$$
$$4x + 2y = 5$$

```
37      LET X=(C*E-B*F)/G
42      LET Y=(A*F-C*D)/G
```

In statements 37 and 42, we instruct the computer to compute the value of X and Y according to the formulas provided, using parentheses to indicate that C*E - B*F is calculated before the result is divided by G.

```
55      PRINT X,Y
60      GO TO 30
```

The computer prints the two values X and Y in line 55. Having done this, it moves on to line 60 where it is reverted to line 30. With additional numbers in the DATA statements, the computer is told in line 30 to take the next one and assign it to C, and the one after that to F. Thus,

$$x + 2y = 1$$
$$4x + 2y = 3$$

As before, it finds the solutions in 37 and 42, prints them out in 55, and then is directed in 60 to revert to 30.

In line 30, the computer reads two more values, 4 and -7, which it finds in line 85. It then proceeds to solve the system

$$x + 2y = 4$$
$$4x + 2y = -7$$

and print out the solutions. Since there are no more pairs of numbers in the DATA statement available for C and F, the computer prints OUT OF DATA IN 30 and stops.

If line number 55 (PRINT X, Y) had been omitted, the computer would have solved the three systems and then told us when it was out of data. If we had omitted line 20, and G were equal to zero, the computer would print DIVISION BY ZERO IN 37 and DIVISION BY ZERO IN 42. Had we omitted statement 60 (GO TO 30), the computer would have solved the first system, printed out the values of X and Y, and then gone to line 65, where it would be directed to print NO UNIQUE SOLUTION.

The particular choice of line numbers is arbitrary as long as the statements are numbered in the order the machine is to follow. We would normally number the statements 10, 20, 30, ..., 130, so that later we can insert additional statements. Thus, if we find that we have omitted two statements between those numbered 40 and 50, we can give them any two numbers between 40 and 50 -- say 44 and 46. Regarding DATA statements, we need only put the numbers in the order that we want them read (the first for A, the second for B, the third for D, the fourth for E, the fifth for C, the sixth for F, the seventh for the next C, etc.). In place of the three statements numbered 70, 80, and 85, we could have written the statement:

```
75        DATA 1,2,4,2,-7,5,1,3,4,-7
```

or, more naturally,

```
70        DATA 1,2,4,2
75        DATA -7,5
80        DATA 1,3
85        DATA 4,-7
```

to indicate that the coefficients appear in the first data statement and the various pairs of right-hand constants appear in the subsequent statements.

The program and the resulting run is shown below as it appears on the Teletype.

```
10        READ A,B,D,E
15        LET G=A*E-B*D
20        IF G=0 THEN 65
30        READ C,F
37        LET X=(C*E-B*F)/G
42        LET Y=(A*F-C*D)/G
55        PRINT X,Y
60        GO TO 30
65        PRINT "NO UNIQUE SOLUTION"
70        DATA 1,2,4
80        DATA 2,-7,5
85        DATA 1,3,4,-7
90        END
RUN
```

(continued on next page)

```
LINEAR          11:03          10/19/69

4               -5.50000
0.666667        0.166667
-3.66667        3.83333
OUT OF DATA IN 30
```

NOTE

Remember to terminate all statements by pressing the RETURN key.

After typing the program, we type the word RUN, followed by a carriage return to direct the computer to execute the program. Note that the computer, before printing out the answers, printed the name LINEAR which we gave to the problem (refer to paragraph 4.1) and the time and date of the computation. The message OUT OF DATA IN 30, may be ignored here. However, in some instances, it indicates an error in the program.

1.3 FUNDAMENTAL CONCEPTS OF BASIC

BASIC can perform many operations such as adding, subtracting, multiplying, dividing, extracting square roots, raising a number to a power, and finding the sine of an angle measured in radians.

1.3.1 Arithmetic Operations

The computer performs its primary function (that of computation) by evaluating formulas similar to those used in standard mathematical calculation, with the exception that all BASIC formulas must be written on a single line. Five arithmetic operations can be used to write a formula.

Symbol	Example	Meaning
+	A + B	add B to A
-	A - B	subtract B from A
*	A * B	multiply B by A
/	A / B	divide A by B
↑	X ↑ 2	find X^2

If we type A + B * C ↑ D, the computer first raises C to the power D, multiplies this result by B, and then adds the resulting product to A. We must use parentheses to indicate any other order. For example, if it is the product of B and C that we want raised to the power D, we must write A + (B * C) ↑ D; or if we want to multiply A + B by C to the power D, we write (A + B) * C ↑ D. We could add A to B, multiply their sum by C, and raise the product to the power D by writing ((A + B) * C) ↑ D. The order of precedence is summarized in the following rules.

a. The formula inside parentheses is evaluated before the parenthesized quantity is used in computations.

b. In the absence of parentheses in a formula, BASIC performs exponentiations first, multiplications and divisions second, and additions and subtractions third.

c. In the absence of parentheses in a formula involving only multiplications and divisions, BASIC performs the operations from left to right, in the order that they are read.

d. In the absence of parentheses in a formula involving only additions and subtractions, BASIC performs the operations from left to right, in the order that they are read.

The rules tell us that the computer, faced with A - B - C, (as usual) subtracts B from A, and then C from their difference; faced with A/B/C, it divides A by B, and that quotient by C. Given A ↑ B ↑ C, the computer raises the number A to the power B and takes the resulting number and raises it to the power C. If there is any question about the precedence, put in more parentheses to eliminate possible ambiguities.

1.3.2 Mathematical Functions

In addition to these five arithmetic operations, BASIC can evaluate certain mathematical functions. These functions are given special three-letter English names.

Function	Interpretation	
SIN (X)	Find the sine of X	
COS (X)	Find the cosine of X	X interpreted as
TAN (X)	Find the tangent of X	an angle measured
COT (X)	Find the cotangent of X	in radians
ATN (X)	Find the arctangent of X	
EXP (X)	Find e raised to the X power (e^X)	
LOG (X)	Find the natural logarithm of X (ln X)	X interpreted
ABS (X)	Find the absolute value of X ($\lvert X \rvert$)	as a
SQR (X)	Find the square root of X ($\sqrt{X}$)	number

Five other functions are also available in BASIC: INT, RND, SGN, NUM, and DET; these are reserved for explanation in Chapters 5 and 7. In place of X, we may substitute any formula or any number in parentheses following any of these functions. For example, we may ask the computer to find $\sqrt{4 + X^3}$ by writing SQR (4 + X ↑ 3), or the arctangent of $3X - 2e^X + 8$ by writing ATN (3 * X - 2 * EXP (X) + 8). If the value of $(\frac{5}{6})^{17}$ is needed, the two-line program can be written:

```
10      PRINT(5/6)↑17
20      END
```

and the computer finds the decimal form of this number and prints it out.

1.3.3 Numbers

A number may be positive or negative and it may contain up to eight digits, but it must be expressed in decimal form (i.e., 2, -3.675, 12345678, -.98765432, and 483.4156). The following are not numbers in BASIC: 14/3 and SQR(7). The computer can find the decimal expansion of 14/3 or SQR(7), but we may not include either in a list of DATA. We gain further flexibility by using the letter E, which stands for: times ten to the power. Thus, we may write .0012345678 as .12345678E-2 or 12345678E-10 or 1234.5678E-6. We do not write E7 as a number, but write 1E7 to indicate that it is 1 that is multiplied by 10^7.

1.3.4 Variables

A numerical variable in BASIC is denoted by any letter, or by any letter followed by a single digit. (See Chapter 8 for alphanumeric string variables.) Thus, the computer interprets E7 as a variable, along with A, X, N5, 10, and O1. A variable in BASIC stands for a number, usually one that is not known to the programmer at the time the program is written. Variables are given or assigned values by LET and READ statements. The value so assigned does not change until the next time a LET or READ statement is encountered with a value for that variable. However, all variables are set equal to 0 before a RUN. Consequently, it is only necessary to assign a value to a variable when a value other than 0 is required.

Although the computer does little in the way of correcting during computation, it sometimes helps if an absolute value hasn't been indicated. For example, if you ask for the square root of -7 or the logarithm of -5, the computer gives the square root of 7 along with an error message stating that you have asked for the square root of a negative number, or it gives the logarithm of 5 along with the error message that you have asked for the logarithm of a negative number.

1.3.5 Relational Symbols

Six other mathematical symbols of relation are used in IF-THEN statements where it is necessary to compare values. An example of the use of these relation symbols was given in the sample program LINEAR.

Any of the following six standard relations may be used:

Symbol	Example	Meaning
=	A = B	A is equal to B
<	A < B	A is less than B
<=	A<= B	A is less than or equal to B
>	A > B	A is greater than B
>=	A>= B	A is greater than or equal to B
<>	A<>B	A is not equal to B

1.4 SUMMARY

Several elementary BASIC commands have been introduced in our discussions. In describing each of these commands, a line number is assumed, and brackets are used to denote a general type. For example, [variable] refers to any variable.

1.4.1 LET Statement

The LET statement is used when computations must be performed. This command is not of algebraic equality, but a command to the computer to perform the indicated computations and assign the answer to a certain variable. Each LET statement is of the form:

LET [variable] = [formula]

Generally, several variables may be assigned the same value by a single LET statement. Examples of assigning a value to a single variable are given in the following two statements:

```
100     LET X=X+1
259     LET W7=(W-X4↑3)*(Z-A/(A-B)-17)
```

Examples of assigning a value to more than one variable are given in the following statements:

5Ø	LET X=Y3=A(3,1)=1	The variables X, Y3, and A(3,1) are assigned the value 1.
9Ø	LET W=Z=3*X-4*X↑2	The variables W and Z are assigned the value 3*X-4X ↑ 2

1.4.2 READ and DATA Statements

READ and DATA statements are used to enter information into the computer. We use a READ statement to assign to the listed variables those values which are obtained from a DATA statement. Neither statement is used without the other. A READ statement causes the variables listed in it to be given in order, the next available numbers in the collection of DATA statements. Before the program is run, the computer takes all of the DATA statements in the order they appear and creates a large data block. Each time a READ statement is encountered anywhere in the program, the data block supplies the next available number or numbers. If the data block runs out of data, the program is assumed to be finished and we get an OUT OF DATA message.

Since we have to read in data before we can work with it, READ statements normally occur near the beginning of a program. The location of DATA statements is arbitrary, as long as they occur in the correct order. A common practice is to collect all DATA statements and place them just before the END statement.

Each READ statement is of the form:

READ [sequence of variables]

Each DATA statement is of the form:

DATA [sequence of numbers]

```
150     READ X,Y,Z,X1,Y2,Q9
330     DATA 4,2,1.7
340     DATA 6.734E-3,-174.321,3.1415927

234     READ B(K)
263     DATA 2,3,5,7,9,11,10,8,6,4

10      READ R(I,J)
440     DATA -3,5,-9,2.37,2.9876.-437.234E-5
450     DATA 2.765, 5.5576, 2.3789E2
```

Remember that only numbers are put in a DATA statement, and that 15/7 and SQRT(3) are formulas, not numbers. Refer to Chapter 3 for a discussion of the subscripted variables.

1.4.3 PRINT Statement

The common uses of the PRINT statement are:

a. to print out the result of some computations

b. to print out verbatim a message included in the program

c. a combination of the two

d. to skip a line.

The following are examples of a type a.:

```
100     PRINT X,SQR(X)
135     PRINT X,Y,Z, B*B-4*A*C, EXP(A-B)
```

The first example prints X, and a few spaces to the right, the square root of X. The second prints five different numbers:

$$X, Y, Z, B^2 -4AC, \text{and } e^{A-B}$$

The computer computes the two formulas and prints up to five numbers per line in this format.

The following are examples of type b.:

```
100     PRINT "NO UNIQUE SOLUTION"
430     PRINT "X VALUE", "SINE", "RESOLUTION"
500     PRINT X,M,D
```

Line 100 prints the sample statement, and line 430 prints the three labels with spaces between them. The labels in 430 automatically line up with the three numbers called for in PRINT statement 500.

The following is an example of type c.:

```
150     PRINT "THE VALUE OF X IS" X
30      PRINT "THE SQUARE ROOT OF" X, "IS" SQR(X)
```

If the first has computed the value of X to be 3, it prints out: THE VALUE OF X IS 3. If the second has computed the value of X to be 625, it prints out: THE SQUARE ROOT OF 625 IS 25.

The following is an example of type d.:

```
250     PRINT
```

The computer advances the paper one line when it encounters this command.

1.4.4 GO TO Statement

The GO TO statement is used when we want the computer to unconditionally transfer to some command other than the next sequential command. In the LINEAR problem, we direct the computer to go through the same process for different values of C and F with a GO TO statement. This statement is in the form of GO TO [line number].

```
150     GO TO 75
```

1.4.5 IF - THEN Statement

The IF - THEN statement is used to transfer conditionally from the sequential order of commands according to the truth of some relation. It is sometimes called a conditional GO TO statement. Each IF - THEN statement is of the form:

IF [formula] [relation] [formula] THEN [line number]

The following are two examples of this statement:

```
40      IF SIN(X)<=M THEN 80
20      IF G=0 THEN 65
```

If the first asks if the sine of X is less than or equal to M, and skips to line 80 if so. The second asks if G is equal to 0, and skips to line 65 if so. In each case, if the answer to the question is no, the computer goes to the next line.

1.4.6 ON - GO TO Statement

The IF - THEN statement allows a two-way fork in a program; the ON - GO TO statement allows a many-way switch. For example:

```
80      ON X GO TO 100,200,150
```

This condition causes the following to occur:

If X = 1, the program goes to line 100,
If X = 2, the program goes to line 200,
If X = 3, the program goes to line 150

In other words, any formula may occur in place of X, and the instruction may contain any number of line numbers, as long as it fits on a single line. The value of the formula is computed and its integer part is taken. If this is 1, the program transfers to the line whose number is first on the list; if its integer part is 2, the program transfers to the line whose number is the second one, etc. If the integer part of the formula is below 1, or larger than the number of line numbers listed, an error message is printed. To increase the similarity between the ON - GO TO and IF - THEN instructions, the instruction

```
75      IF X>5 THEN 200
```

may also be written as:

```
75      IF X>5 GO TO 200
```

Conversely, THEN may be used in an ON - GO TO statement.

1.4.7 END Statement

Every program must have an END statement, and it must be the statement with the highest line number in the program.

```
999     END
```

CHAPTER 2
LOOPS

We are frequently interested in writing a program in which one or more portions are executed a number of times, usually with slight variations each time. To write the simplest program in which the portion of the program to be repeated is written just once, we use a loop. A loop is a block of instructions that the computer executes repeatedly until a specified terminal condition is met.

The programs which use loops can be best illustrated and explained by using two versions of a program for the simple task of printing out a table of the positive integers 1 through 100 together with the square root of each. Without a loop, the first program is 101 lines long and reads

```
10      PRINT 1,SQR(1)
20      PRINT 2,SQR(2)
30      PRINT 3,SQR(3)
        . . . . . . . .
990     PRINT 99,SQR(99)
1000    PRINT 100,SQR(100)
1010    END
```

With the following program example, using one type of loop, we can obtain the same table with far fewer lines of instructions (5 instead of 101):

```
10      LET X=1
20      PRINT X,SQR(X)
30      LET X=X+1
40      IF X<=100 THEN 20
50      END
```

Statement 10 gives the value of 1 to X and initializes the loop. In line 20, both 1 and its square root are printed. Then, in line 30, X is increased by 1, to a value of 2. Line 40 asks whether X is less than or equal to 100; an affirmative answer directs the computer back to line 20, where it prints 2 and $\sqrt{2}$ and goes to 30. Again, X is increased by 1, this time to 3, and at 40 it goes back to 20. This process is repeated -- line 20 (print 3 and $\sqrt{3}$), line 30 (X = 4), line 40 (since 4 < 100, go back to line 20), etc. -- until the loop has been traversed 100 times. Then, after it has printed 100 and its square root, X becomes 101. The computer now receives a negative answer to the question in line 40 (X is greater than 100, not less than or equal to it), does not return to 20 but moves on to line 50, and ends the program. All loops contain four characteristics:

a. initialization (line 10)
b. the body (line 20)
c. modification (line 30)
d. an exit test (line 40)

2.1 FOR AND NEXT STATEMENTS

BASIC provides two statements to specify a loop: the FOR statement and the NEXT statement.

```
10        FOR X=1 TO 100
20        PRINT X,SQR(X)
30        NEXT X
50        END
```

In line 10, X is set equal to 1, and a test is set up, like that of line 40 above. Line 30 carries out two tasks: X is increased by 1, and the test is carried out to determine whether to go back to 20 or to go on. Thus, lines 10 and 30 take the place of lines 10, 30, and 40 in the previous program.

Note that the value of X is increased by 1 each time BASIC goes through the loop. If we want a different increase, we could specify it by writing the following:

```
10        FOR X=1 TO 100 STEP 5
```

and the computer would assign 1 to X on the first time through the loop, 6 to X on the second time, 11 on the third , and 96 on the last time. Another step of 5 would take X beyond 100, allowing the program to proceed to the end after printing 96 and its square root. The STEP may be positive or negative, and we could have obtained the first table, printed in reverse order, by writing line 10 as follows:

```
10        FOR X=100 TO 1 STEP-1
```

In the absence of a STEP instruction, a step-size of +1 is assumed.

More complicated FOR statements are allowed. The initial value, the final value, and the step-size may all be formulas of any complexity. For example, if N and Z have been specified earlier in the program, we could write the following:

```
FOR X=N+7*Z TO (Z-N)/3 STEP(N-4*Z)/10
```

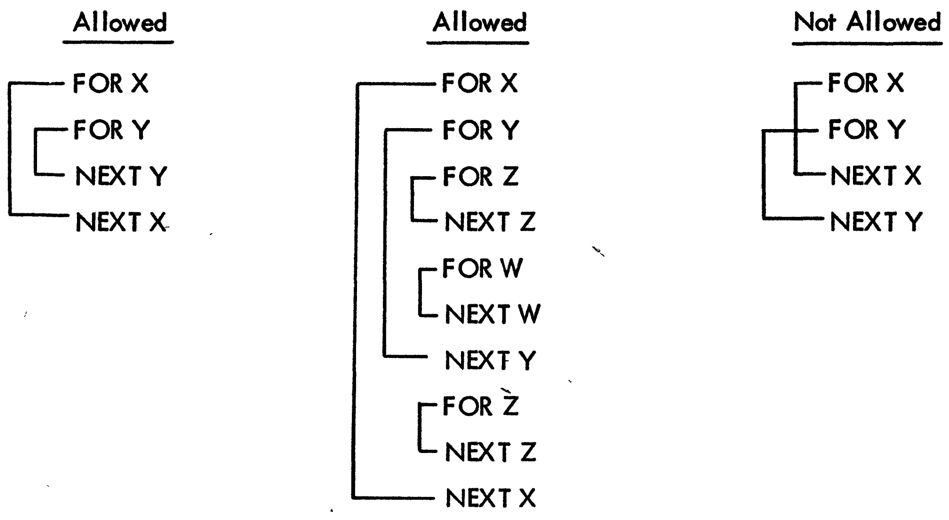
For a positive step-size, the loop continues as long as the control variable is less than or equal to the final value. For a negative step-size, the loop continues as long as the control variable is greater than or equal to the final value.

If the initial value is greater than the final value (less than the final value for negative step-size), the body of the loop is not performed at all, but the computer immediately passes to the statement following the NEXT. The following program for adding up the first n integers gives the correct result 0 when n is 0.

```
10      READ N
20      LET S=Ø
30      FOR K=1 TO N
40      LET S=S+K
50      NEXT K
60      PRINT S
70      GO TO 1Ø
90      DATA 3,1Ø,Ø
99      END
```

2.2 NESTED LOOPS

Nested loops (loops within loops) can be expressed with FOR and NEXT statements. They must be nested and not crossed as the following skeleton examples illustrate:



2.3 SUMMARY

By making use of a loop, the programmer can direct the computer to execute portions of a program many times. This is a concise technique for writing a program, and saves the programmer much type-in time. In describing the instructions used to specify a loop, a line number is assumed and brackets are used to denote a general type.

2.3.1 FOR and NEXT Statements

Every FOR statement is of the following form:

FOR [variable] = [formula] TO [formula] STEP [formula]

Most commonly, the expressions are integers and the STEP is omitted. In the latter case, a step-size of +1 is assumed. The accompanying NEXT statement is simple in form, but the variable must be precisely the same one as that following FOR in the FOR statement. Its form is as follows:

NEXT [variable]

```
30        FOR X=0 TO 3 STEP D
80        NEXT X

120       FOR X4=(17+COX(Z))/3 TO 3* SQR(10) STEP 1/4
235       NEXT X4

240       FOR X=8 TO 3 STEP -1

456       FOR J=-3 TO 12 STEP 2
```

Note that the step-size may be a variable (D), a formula (1/4), a negative number (-1), or a positive number (2). In lines 120 and 235, the successive values of X4 are .25 apart, in increasing order. In line 240, the successive values of X will be 8, 7, 6, 5, 4, 3. In line 456, on successive trips through the loop, J will take on values -3, -1, 1, 3, 5, 7, 9, and 11. If the initial, final, or step-size values are given as formulas, these formulas are evaluated upon entering the FOR statement. The control variable can be changed in the body of the loop; it should be noted that the exit test always uses the latest value of this variable. If 50 FOR Z = 2 TO -2 is written, without a negative step-size, the body of the loop is not performed and the computer proceeds to the statement immediately following the corresponding NEXT statement.

CHAPTER 3
LISTS AND TABLES

In addition to the ordinary variables used by BASIC, variables can be used to designate the elements of a list or a table. Many occasions arise where a list or a table of numbers is used over and over, and, since it is inconvenient to use a separate variable for each number, BASIC allows the programmer to designate the name of a list or table by a single letter.

Lists are used when we might ordinarily use a single subscript, as in writing the coefficients of a polynomial (a_0, a_1, a_2, ..., a_n). Tables are used when a double subscript is to be used, as in writing the elements of a matrix (bi,j). The variables used in BASIC consist of a single letter, which is the name of the list or table, followed by the subscript in parentheses. Thus,

A(0), A(1), A(2),..., A(N)

represents the coefficients of a polynomial, and

B(1,1), B(1,2), ..., B(N,N)

represents the elements of a matrix.

The single letter denoting a list or a table name may also be used without confusion to denote a simple variable. However, the same letter may not be used to denote both a list and a table in the same program because BASIC recognizes a list as a special kind of table having only one column. The form of the subscript is flexible: A list item B(I + K) may be used, or a table item Q(A(3,7), B-C) may be used.

We can enter the list A(0), A(1), ..., A(10) into a program by the following lines:

```
10      FOR I=0 TO 10
20      READ A(I)
30      NEXT I
40      DATA 2,3,-5,2.2,4,-9,123,4,-4,3
```

3.1 THE DIMENSION STATEMENT (DIM)

BASIC automatically reserves room for any list or table with subscripts of 10 or fewer. However, if we want larger subscripts, we must use a DIM statement. This statement indicates to the computer that the specified space is to be allowed for the list or table. For example, the instruction

```
10      DIM A(15)
```

reserves 16 spaces for list A (A(0), A(1), A(2),...,A(15)). The instruction

```
20      DIM Y(10,15)
```

reserves 176 spaces for matrix Y (10 + 1 rows * 15 + 1 columns). Space may be reserved for more than one list and/or table with a single DIM statement by separating the entries with commas, as shown in the following example:

```
30      DIM A(100),B(20,30),C(25)
```

A DIM statement is not executed; therefore, it may appear on any line before the END statement. However, the best place to put it is at the beginning so that it is not forgotten. If we enter a table with a subscript greater than 10, without a DIM statement, BASIC gives an error message, telling us that we have a subscript error. This condition can be rectified by entering a DIM statement with a line number less than the line number of the END statement.

A DIM statement is normally used to reserve additional space, but in a long program that requires many small tables, it may be used to reserve less space for tables in order to have more space for the program. When in doubt, declare a larger dimension than you expect to use, but not one so large that there is no room for the program. For example, if we want a list of 15 numbers entered, we may write the following:

```
10      DIM A(25)
20      READ N
30      FOR I=1 TO N
40      READ A(I)
50      NEXT I
60      DATA 15
70      DATA 2,3,5,7,11,13,17,19,23,29,31,37,41,43,47
```

Statements 20 and 60 could have been eliminated by writing 30 as FOR I = 1 TO 15 but the program as typed allows for the lengthening of the list by changing only statement 60, as long as the list does not exceed 25 and there is sufficient data.

We could enter a 3-by-5 table into a program by writing the following:

```
10      FOR I=1 TO 3
20      FOR J=1 TO 5
30      READ B(I,J)
40      NEXT J
50      NEXT I
60      DATA 2,3,-5,-9,2
70      DATA 4,-7,3,4,-2
80      DATA 3,-3,5,7,8
```

Again, we may enter a table with no DIM statement: BASIC then handles all the entries from B(0,0) to B(10,10).

3.2 EXAMPLE

Below are the statements and run of a problem which uses both a list and a table. The program computes the total sales of five salesmen, all of whom sell the same three products. The list, P, gives the price per item of the three products and the table, S, tells how many items of each product each man sold. Product 1 sells for $1.25 per item, product 2, for $4.30 per item, and product 3, for $2.50 per item; also, salesman 1 sold 40 items of the first product, 10 of the second, 35 of the third, and so on. The program reads in the price list in lines 40 through 80, using data in lines 910 through 930. The same program could be used again, modifying only line 900 if the prices change, and only lines 910 through 930 to enter the sales in another month. This sample program does not need a DIM statement, because the computer automatically reserves enough space to allow all subscripts to run from 0 to 10.

NOTE

Since spaces are ignored, statements may be indented for visual identity of the various loops within the program.

```
10      FOR I=1 TO 3
20        READ P(I)
30      NEXT I
40      FOR I=1 TO 3
50        FOR J=1 TO 5
60      READ S(I,J)
70          NEXT J
80        NEXT I
90      FOR J=1 TO 5
100       LET S=0
110       FOR I=1 TO 3
120         LET S=S+P(I)*S(I,J)
130       NEXT I
140       PRINT "TOTAL SALES FOR SALESMAN"J,"$"S
150     NEXT J
900     DATA 1.25,4.30,2.50
910     DATA 40,20,37,29,42
920     DATA 10,16,3,21,8
930     DATA 35,47,29,16,33
999     END
```

```
READY
RUN
SALES1   11:06    10/20/69
TOTAL SALES FOR SALESMAN 1 $180.500
TOTAL SALES FOR SALESMAN 2 $211.300
TOTAL SALES FOR SALESMAN 3 $131.650
TOTAL SALES FOR SALESMAN 4 $166.550
TOTAL SALES FOR SALESMAN 5 $169.400
```

3.3 SUMMARY

Because the number of simple variable names is limited, BASIC allows a programmer to use lists and tables to increase the number of problems that can be programmed easily and concisely. A single letter is used for the name of the list or table, and the subscript that follows is enclosed in parentheses. The subscripts may be explicitly stated or may be any legal expression.

Lists and tables are called subscripted variables, and simple variables are called unsubscripted variables. Usually, you can use a subscripted variable anywhere that you use an unsubscripted variable. However, the variable mentioned immediately after FOR in the FOR statement and after NEXT in the NEXT statement must be an unsubscripted variable. The initial, terminal, and step values may be any legal expression.

3.3.1 The DIM Statement

To enter a list or a table with a subscript greater than 10, a DIM statement is used to retain sufficient space, as in the following examples:

```
20      DIM H(35)
35      DIM Q(5, 25)
```

The first example enables us to enter list H with 36 items (H(0), H(1), ..., H(35)). The second reserves space for a table of 156 items (5 + 1 rows * 25 + 1 columns).

CHAPTER 4
HOW TO RUN BASIC

After learning how to write a BASIC program, we must learn how to gain access to BASIC via the Teletype so that we can type in a program and have the computer solve it. Steps required to communicate with the monitor must first be performed. These steps are fully explained in the PDP-10 Reference Handbook and the PDP-10 System User's Guide.

4.1 GAINING ACCESS TO BASIC

Once the monitor has responded with a period to indicate that it is ready to receive a monitor command, type in the following command:

```
.R BASIC
```

This command establishes contact with the BASIC CUSP (Commonly Used System Program). BASIC responds with the following:

```
NEW OR OLD--
```

Type in:

```
NEW
```

if you are going to create a new program. BASIC responds with the following:

```
NEW FILE NAME--
```

After you type in the name of your program, BASIC checks to make sure that the name does not already exist. If you want to work with a previously created program that you saved on a storage device (disk or DECtape), type in the following:

```
OLD
```

BASIC then asks for the name of the old program, as follows:

```
OLD FILE NAME--
```

Respond by typing in the name of your old file. If your old file is stored on a directory device other than the disk, you must type in the device name as in the following example:

```
OLD FILE NAME--DTA6:SAMPLE
```

BASIC retrieves the file named SAMPLE from DECtape 6 and replaces the current contents of user core with the file SAMPLE. The disk may be specified as the device on which the old program is stored, but this is not necessary because the disk is the device used when no device is specified. For example, the following statements are equivalent:

```
OLD FILE NAME--DSK:TEST1

OLD FILE NAME--TEST1
```

Program names can be any combination of letters and digits up to and including six characters in length. Characters other than letters and digits can be used, but * , ; / $ are to be avoided. In previous chapters we have used program names such as LINEAR and SALES 1. If you recall an old program from storage, you must use exactly the same name you assigned to it when it was saved.

4.2 ENTERING THE PROGRAM

After you type in your filename (whether it is old or new), BASIC responds with the following:

```
READY
```

You can begin to type in your program. Make sure that each line begins with a line number containing no more than five digits and containing no spaces or nondigit characters. Also, be sure to start at the beginning of the Teletype line for each new line. Press the RETURN key upon completion of each line.

If, in the process of typing a statement, you make a typing error and notice it before you terminate the line, you can correct it by pressing the RUBOUT key once for each character to be erased, going backward until the character in error is reached. Then continue typing, beginning with the character in error. The following is an example of this correcting process:

```
10      PRNIT\TIN\INT 2,3
```

NOTE

The RUBOUT key echoes as a backslash (\), followed by the deleted characters and a second backslash.

Also, to delete the entire line being typed, you can depress the ALTMODE key (if a Teletype Model 35 is used), the ESC key (if a Teletype Model 33 is used), or the PREFIX key (if a Teletype Model 37 is used).

4.3 EXECUTING THE PROGRAM

After typing the complete program (do not forget to end with an END statement), type RUN or RUNNH, followed by the RETURN key. BASIC types the name of the program, the time of day, the current date (unless RUNNH is specified), and then it analyzes the program. If the program can be run, BASIC executes it and, via PRINT statements, types out any results that were requested. The typeout of results does not guarantee that the program is correct (the results could be wrong), but it does indicate that no grammatical errors exist (e.g., missing line numbers, misspelled words, or illegal syntax). If errors of this type do exist, BASIC types a message (or several messages) to you. A list of these diagnostic messages, with their meanings, is given in Appendix B.

4.4 CORRECTING THE PROGRAM

If you receive an error message typeout informing you, for example, that line 60 is in error, the line can be corrected by typing in a new line 60 to replace the erroneous one. If the statement on line 110 is to be eliminated from your program, it is accomplished by typing the following:

```
110 ⤸
```

If you wish to insert a statement between lines 60 and 70, type a line number between 60 and 70 (e.g., 65), followed by the statement.

4.5 INTERRUPTING THE EXECUTION OF THE PROGRAM

If the results being typed out seem to be incorrect and you want to stop the execution of your program, type ↑O (hold down CTRL key and at the same time type O) to suppress the typeout, or type ↑C twice, as indicated in the following example:

```
↑C    { Stops execution of your program, and
↑C    { Returns control to Monitor
```

If you typed ↑C, the monitor responds with a period and waits for you to type a monitor command. If you wish to reinitialize, type either of the following:

`.START` or `.REENTER`

BASIC responds with the following:

```
          READY
```

whereupon you can modify or add statements and/or type RUN. If you wish to continue at the point where you interrupted the execution, type the following:

```
          .CONT
```

4.6 LEAVING THE COMPUTER

When you wish to leave the computer, type the following

```
          ↑C
```

The monitor responds with a period. Then type the following:

```
          .KJOB
```

The monitor responds with the following:

```
          CONFIRM:
```

If you simply want to get off the machine and delete all files you may have created, type the following:

```
          K
```

Other options available following the typeout of CONFIRM: are listed for you if you respond to the CONFIRM: message with a carriage return (RETURN key) only. The monitor then lists all options available, along with the response required to request each option.

4.7 EXAMPLE OF BASIC RUN

The following is a simple example of the use of BASIC under a time-sharing monitor:

```
          ↑C                                         GO TO MONITOR LEVEL
          .LOGIN                                     REQUEST LOGIN
          JOB 7     DEC PDP-10 #40 4561H PR          MONITOR TYPES OUT YOUR ASSIGNED
                                                     JOB NUMBER, THE CURRENT VERSION
                                                     NUMBER OF THE MONITOR

          #27,20                                     MONITOR REQUESTS YOUR PROJECT-
                                                     PROGRAMMER NUMBER; TYPE IT IN
```

```
PASSWORD:
```

MONITOR REQUESTS YOUR PASSWORD; TYPE IT IN; IT WILL NOT ECHO BACK

```
Ø927 29-OCT-69  TTY3

↑C
```

MONITOR TYPES OUT THE TIME OF DAY, THE CURRENT DATE, YOUR TELETYPE UNIT NUMBER, ↑C, AND A PERIOD

```
.R BASIC
```

INSTRUCT MONITOR TO BRING BASIC INTO CORE AND START ITS EXECUTION

```
NEW OR OLD--NEW
```

BASIC ASKS WHETHER NEW OR OLD PROGRAM IS TO BE RUN

```
NEW FILE NAME--SAMPLE
```

BASIC ASKS FOR NEW FILENAME

```
READY
```

BASIC IS NOW READY TO RECEIVE STATEMENTS

```
1Ø      FOR N=1 TO 7

2Ø      PRINT N, SQR(N)

3Ø      NEXT N

4Ø      PRINT "DONE"

5Ø      END
```

TYPE IN STATEMENTS

```
RUN
```

RUN PROGRAM

```
SAMPLE  11:14  1Ø/2Ø/69

1             1

2             1.41421

3             1.732Ø5

4             2

5             2.236Ø7

6             2.44949

7             2.64575

DONE

↑C

.KJOB

CONFIRM:K

JOB 7, USER 27, 2Ø OFF TTY3 AT Ø93Ø ON 29-OCT-69

FILES DELETED: Ø, FILES SAVED: Ø, RUNTIME Ø MIN, Ø1 SEC
```

4.8 ERRORS AND DEBUGGING

Occasionally, the first run of a new problem is free of errors and gives the correct answers, but, more commonly, errors are present and have to be corrected. Errors are of two types: errors of form (grammatical errors) which prevent the running of the program, and logical errors in the program which cause the computer to produce wrong answers or no answers at all.

Errors of form cause error messages to be printed, and the various types of error messages are listed and explained in Appendix B. Logical errors are more difficult to uncover, particularly when the program gives answers which seem to be nearly correct. In either case, after the errors are discovered, they can be corrected by changing lines, by inserting new lines, or by deleting lines from the program. As indicated previously, a line is changed by typing it correctly with the same line number; a line is inserted by typing it with a line number between those of two existing lines; and a line is deleted by typing its line number and pressing the RETURN key. Note that you can insert a line only if the original line numbers are not consecutive integers. For this reason, most programmers begin by using arbitrary line numbers that are multiples of five or ten.

These corrections can be made either before or after a run. Since BASIC sorts out lines and arranges them in order, a line may be retyped out of sequence. Simply retype the offending line with its original line number.

4.8.1 Example of Finding and Correcting Errors

We can best illustrate the process of finding the errors (bugs) in a program and correcting (debugging) them by an example. Consider the problem of finding that value of X between 0 and 3 for which the sine of X is a maximum, and ask the machine to print out this value of X and the value of its sine. Although we know that $\pi/2$ is the correct value, we use the computer to test successive values of X from 0 to 3, first using intervals of .1, then of .01, and finally of .001. Thus, we ask the computer to find the sine of 0, of .1, of 2, of .3..., of 2.8, of 2.9, and of 3, and to determine which of these 31 values is the largest. It does so by testing SIN(0) and SIN(.1) to see which is larger, and calling the larger of these two numbers M. It then picks the larger of M and SIN(.2) and calls it M. This number is checked against SIN(.3). Each time a larger value of M is found, the value of X is "remembered" in X0. When it finishes, M will have been assigned to the largest value. It then repeats the search, this time checking the 301 numbers 0, .01, 02, .03, ..., 2.98, 2.99, and 3, finding the sine of each, and checking to see which has the largest sine. At the end of each of these three searches, we want the computer to print three numbers: the value X0 which has the largest sine, the sine of that number, and the interval of search.

Before going to the Teletype, we write a program such as the following:

```
10        READ D
20        LET X0=0
30        FOR X=0 TO 3 STEP D
40        IF SIN(X)<=M THEN 100
50        LET X0=X
60        LET M=SIN(X0)
70        PRINT X0,X,D
80        NEXT X0
90        GO TO 20
100       DATA .1,.01,.001
110       END
```

The following is a list of the entire sequence on the Teletype with explanatory comments on the right side:

```
NEW OR OLD--NEW
NEW FILE NAME--MAXSIN
READY
10        READ D
20        LWR X0=0
30        FOR X=0 TO 3 STEP D
40        IF SINE\E\(X)<=M THEN 100
50        LET X0=X
60        LET M=SIN(X)
70        PRINT XO,X,D
80        NEXT T\T\X0
90        GO TO 20
20        LET X0=0
100       DATA.1.,.01,.001
110       END
RUN

MAXSIN  11:35   10/20/69


ILLEGAL VARIABLE IN 70
NEXT WITHOUT FOR IN 80
FOR WITHOUT NEXT IN 30

70        PRINT X0,X,D
40        IF SIN(X)<=M THEN 80
80        NEXT X
RUN

MAXSIN  11:36   10/20/69

0.1       0.1       0.1
0.2       0.2       0.1
0.3

20
RUN

MAXSIN  11:37            10/20/69

UNDEFINED LINE NUMBER 20 IN 90
```

Note the use of the RUBOUT key (echoes as a \) to erase a character in line 40 (which should have started IF SIN (X), etc.) and in line 80.

We discover that LET was mistyped in line 20, and we correct it after 90.

After receiving the first error message, we inspect line 70 and find that we used XO for a variable instead of X0. The next two error messages relate to lines 30 and 80 having mixed variables. These are corrected by changing line 80.

Both of these changes are made by retyping lines 70 and 80. In looking over the program, we also discover that the IF-THEN statement in 40 directed the computer to a DATA statement and not to line 80 where it should go. This is obviously incorrect. We are having every value of X printed, so we direct the machine to cease operations by typing ↑C twice even while it is running. We notice that SIN(0) is compared with M on the first time through the loop, but we had assigned a value to X0 but not to M. However, we recall that all variables are set equal to zero before a RUN; therefore, line 20 is unnecessary.

```
90        GO TO 10
RUN

MAXSIN    11:43                   10/20/69

0.1       0.1       0.1
0.2       0.2       0.1
0.3
```

Line 90, of course, sent us back to line 20 to repeat the operation and not back to line 10 to pick up a new value for D. We retype line 90 and then type RUN again.

We are about to print out the same table as before. Each time that it goes through the loop, it is printing out X0, the current value of X, and the interval size.

```
70
85        PRINT X0,M,D
5         PRINT "X VALUE","SIN",RESOLUTION"
RUN

MAXSIN    11:44    10/20/69

ILLEGAL VARIABLE IN 5

5         PRINT "X VALUE","SINE","RESOLUTION"
RUN


MAXSIN    11:47    10/20/69
```

We rectify this condition by moving the PRINT statement outside the loop. Typing 70 deletes that line, and line 85 is outside of the loop. We also realize that we want M printed, not X. We also decide to put in headings for the columns by a PRINT statement.

There is an error in our PRINT statement: no left quotation mark for the third item.

Retype line 5, with all of the required quotation marks.

```
X VALUE             SINE                RESOLUTION
1.60                0.999574            0.1
1.57                1.                  0.01
1.57099             1.                  0.001
OUT OF DATA IN 10

LIST

MAXSIN   11:48    10/20/69
5        PRINT "X VALUE","SINE","RESOLUTION"
10       READ D
30       FOR X=0 TO 3 STEP D
40       IF SIN(X)<=M THEN 80
50       LET X0=X
60       LET M=SIN(X)
80       NEXT X
85       PRINT X0,M,D
90       GO TO 10
100      DATA .1,.01,.001
110      END

READY
SAVE
READY
```

These are the desired results. Of the 31 numbers (0, .1, .2, .3,..., 2.8, 2.9, 3), it is 1.6 which has the largest sine, namely .999574; this is true for finer subdivisions.

Having changed so many parts of the program, we ask for a list of the corrected program.

The program is saved for later use.

A PRINT statement could have been inserted to check on the machine computations. For example, if M were checked, we could have inserted 65 PRINT M, and seen the values.

CHAPTER 5
FUNCTIONS AND SUBROUTINES

5.1 FUNCTIONS

Occasionally, you may want to calculate a function, for example, the square of a number. Instead of writing a small program to calculate this function, BASIC provides 14 functions as part of the language, 9 of which are described in Chapter 1. Three of the remaining functions are described here, and the last two are described in Chapter 7.

The desired function is called by a three-letter name. The value to be used is expressed explicitly or implicitly in parentheses and follows the function name. The expression enclosed in parentheses is the argument of the function, and it is evaluated and used as indicated by the function name. For example:

```
15      LET B=SQR(4+X↑3)
```

indicates that the expression (4 + X ↑3) is to be evaluated and then the square root taken.

5.1.1 The Integer Function (INT)

The INT function appears in algebraic notation as [X] and returns the greatest integer of X that is less than or equal to X. For example:

INT (2.35) = 2
INT (-2.35) = -3
INT (12) = 12

One use of this function is to round numbers to the nearest integer by asking for INT (X + .5). For example:

INT (2.9 + .5) = INT (3.4) = 3

rounds 2.9 to 3. Another use is to round to any specific number of decimal places. For example:

INT (X * 10 ↑ 2 + .5) / 10 ↑ 2

rounds X correct to two decimal places and

INT (10 * X ↑ D + .5) / 10 ↑ D

rounds X correct to D decimal places.

5.1.2 The Random Number Generating Function (RND)

The RND function produces random numbers between 0 and 1. This function is used to simulate events that happen in a somewhat random way. RND does not need an argument.

If we want the first 20 random numbers, we can write the program shown below and get 20 six-digit decimals.

```
10      FOR L=1 TO 20
20      PRINT RND,
30      NEXT L
40      END
RUN

RANDOM  13:24   10/20/69

0.406533        0.88445         0.681969        0.939462        0.253358
0.863799        0.880238        0.638311        0.602898        0.990032
0.863799        0.897931        0.628126        0.613262        0.303217
5.00548 E-2     0.393226        0.680219        0.632246        0.668218
```

NOTE

This is a sample run of random numbers. The format of the PRINT statement is discussed in Chapter 6.

```
RUN

RANDOM  13:25   10/20/69

0.406533       0.88445        0.681969        0.939462        0.253358
0.863799
```

A second RUN gives exactly the same random numbers as the first RUN; this is done to facilitate the debugging of programs. If we want 20 random one-digit integers, we could change line 20 to read as follows:

```
20      PRINT INT(10*RND),
RUN
```

We would obtain the following:

```
RANDOM   13:26   10/20/69

4        8        6        9        2
8        8        6        6        9
5        8        6        6        3
0        3        6        6        6
```

To vary the type of random numbers (20 random numbers ranging from 1 to 9, inclusive), change line 20 as follows:

```
20       PRINT INT(9*RND +1);
RUN

RANDOM   13:28   10/20/69

4  8  7  9  3  8  8  6  6  9  6  6  3  1  4  7  6  7
```

To obtain random numbers which are integers from 5 to 24, inclusive, change line 20 to the following:

```
20       PRINT INT(20*RND +5);
RUN

RANDOM   13:28   10/20/69

13  22  18  23  10  22  22  17  17  24  16  22  17  17  11  6  12  18
17  18
```

If random numbers are to be chosen from the A integers of which B is the smallest, call for INT (A*RND+B).

5.1.3 The RANDOMIZE Statement

As noted when we ran the first program of this chapter twice, we got the same numbers in the same order each time. However, we get a different set with the RANDOMIZE statement, as in the following program:

```
5        RANDOMIZE
10       FOR L=1 TO 20
20       PRINT INT(10*RND);
30       NEXT L
40       END
RUN

RNDNOS   13:32   10/20/69

1  9  4  2  1  1  6  6  3  8  4  9  8  6  5  8  6  2  6  0


RUN

RNDNOS   13:33   10/20/69

1  1  4  6  6  6  0  5  3  8  4  0  8  1  0  5  1  8  0  1
```

RANDOMIZE (RANDOM) resets the numbers in a random way. For example, if this is the first instruction in a program using random numbers, then repeated RUNs of the program produce different results. If the instruction is absent, then the official list of random numbers is obtained in the usual order. It is suggested that a simulated model should be debugged without this instruction so that one always obtains the same random numbers in test runs. After the program is debugged, and before starting production runs, you insert the following:

```
        1         RANDOM
```

5.1.4 The Sign Function (SGN)

The SGN function is one which assigns the value 1 to any positive number, 0 to zero, and -1 to any negative number. Thus, SGN (7.23) = 1, SGN (0) = 0, and SGN (-.2387) = -1. For example, the following statement:

```
        50        ON SGN(X)+2 GO TO 100,200,300
```

transfers to 100 if $X < 0$, to 200 if $X = 0$, and to 300 if $X > 0$.

5.1.5 The Define User Function (DEF) and Function End Statement (FNEND)

In addition to the 14 functions BASIC provides, you may define up to 26 functions of your own with the DEF The name of the defined function must be three letters, the first two of which are FN, e.g., FNA, FNB,..., FNZ. Each DEF statement introduces a single function. For example, if you repeatedly use the function $e^{-X^2} + 5$, introduce the function by the following:

```
        30        DEF FNE(X)=EXP(-X↑2)+5
```

and call for various values of the function by FNE (.1), FNE (3.45), FNE (A+2), etc. This statement saves a great deal of time when you need values of the function for a number of different values of the variable.

The DEF statement may occur anywhere in the program, and the expression to the right of the equal sign may be any formula that fits on one line. It may include any combination of other functions, such as those defined by different DEF statements; it also can involve other variables besides those denoting the argument of the function.

Each defined function may have zero, one, two, or more variables as in the following example:

```
        10        DEF FNB(X,Y)=3*X*Y-Y↑3
        105       DEF FNC(X,Y,Z,W)=FNB(X,Y)/FNB(Z,W)
        530       DEF FNA=3.1416*R↑2
```

In the definition of FNA, the current value of R is used when FNA occurs. Similarly, if FNR is defined by the following:

```
70        DEF FNR(X)=SQR(2+LOG(X)-EXP(Y*Z)*(X+SIN(2*Z)))
```

you can ask for FNR(2.7), and give new values to Y and Z before the next use of FNR.

The method of having multiple line DEFs is illustrated by the "max" function shown below. Using this method, the possibility of using IF...THEN as part of the definition is a great help as shown in the following example:

```
10        DEF FNM(X,Y)
20        LET FNM=X
30        IF Y<= X THEN 50
40        LET FNM=Y
50        FNEND
```

The absence of the equals sign (=) in line 10 indicates that this is a multiple line DEF. In line 50, FNEND terminates the definition. The expression FNM, without an argument, serves as a temporary variable for the computation of the function value. The following example defines N-factorial:

```
10        DEF FNF(N)
20        LET FNF=1
30        FOR K=1 TO N
40        LET FNF=K*FNF
50        NEXT K
60        FNEND
```

Any variable which is not an argument of FN in a DEF loop has its current value in the program. Multiple line DEFs may not be nested and there must not be a transfer from inside the DEF to outside its range, or vice versa.

5.2 SUBROUTINES

When you have a procedure that is to be followed in several places in your program, the procedure may be written as a subroutine. A subroutine is a self-contained program which is incorporated into the main program at specified points. A subroutine differs from other control techniques in that the computer remembers where it was before it entered the subroutine, and it returns to the appropriate place in the main program after executing the subroutine.

5.2.1 GOSUB and RETURN Statements

Two new statements, GOSUB and RETURN, are required with subroutines. The subroutine is entered from any point in the main program with a GOSUB statement. This statement is similar to a GO TO statement; however, with a GOSUB statement, the computer remembers where it was prior to the transfer. Following is an example of the GOSUB statement:

```
        90       GOSUB 210
```

where 210 is the line number of the first statement in the subroutine. The last line in the subroutine is a RETURN statement which directs the computer to the statement following the GOSUB from which it transferred. For example:

```
        350      RETURN
```

returns to the next highest line number greater than the GOSUB call.

Subroutines may appear anywhere in the main program, and care should be taken to make certain that the computer enters only through a GOSUB statement and exits via a RETURN statement.

5.2.2 Example

A program for determining the greatest common divisor (GCD) of three integers, using the Euclidean Algorithm, illustrates the use of a subroutine. The first two numbers are selected in lines 30 and 40, and their GCD is determined in the subroutine, lines 200 through 310. The GCD just found is called X in line 60; the third number is called Y, in line 70; and the subroutine is entered from line 80 to find the GCD of these two numbers. This number is, of course, the greatest common divisor of the three given numbers and is printed out with them in line 90.

A GOSUB inside a subroutine to perform another subroutine is called a nested GOSUB. It is necessary to exit from a subroutine only with a RETURN statement. You may have several RETURNs in the subroutine, as long as exactly one of them will be used.

```
10       PRINT "A", "B", "C", "GCD"
20       READ A, B, C
30       LET X=A
40       LET Y=B
50       GOSUB 200
60       LET X=G
70       LET Y=C
80       GOSUB 200
90       PRINT A,B,C,G
100      GO TO 20
110      DATA 60,90,120
120      DATA 38456,64872,98765
130      DATA 32,384,72
200          LET Q=INT(X/Y)
210          LET R=X-Q*Y
220          IF R=0 THEN 300
230          LET X=Y
240          LET Y=R
250          GO TO 200
```

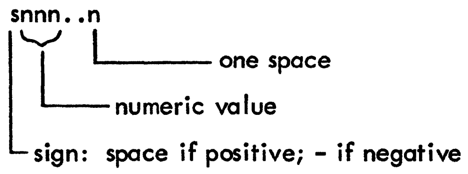
```
300          LET G=Y
310          RETURN
320      END
RUN


GCD3NO             13:38    10/20/69

A        B         C        GCD

60       90        120      30
39456    64872     98765    1
32       384       72       8


OUT OF DATA IN 20
```

CHAPTER 6
MORE SOPHISTICATED TECHNIQUES

The preceding chapters have covered the essential elements of BASIC. At this point, you are in a position to write BASIC programs and to input these programs to the computer via your Teletype. The commands and techniques discussed so far are sufficient for most programs. This chapter and remaining ones are for a programmer who wishes to perform more intricate manipulations and to express programs in a more sophisticated manner.

6.1 MORE ABOUT THE PRINT STATEMENT

The PRINT statement permits a greater flexibility for the more advanced programmer who wishes to have a different format for his output. The Teletype line is divided into 5 zones of 14 spaces each. A comma is a signal to move to the next print zone or, if the fifth print zone has just been filled, to move to the first print zone of the next line. If a label (expression in quotes) is followed by a semicolon, the label is printed with no space after it. If a variable is followed by a semicolon, its value is printed in the following format:



When you type in the following program:

```
1Ø        FOR I=1 TO 15
2Ø        PRINT I
3Ø        NEXT I
4Ø        END
```

the Teletype prints 1 at the beginning of a line, 2 at the beginning of the next line, and, finally, 15 on the fifteenth line. But, by changing line 20 to read as follows:

```
2Ø        PRINT I,
```

the numbers are printed in the zones, reading as follows:

```
1         2         3         4         5
6         7         8         9         1Ø
11        12        13        14        15
```

If you want the numbers printed in this fashion, but compressed, change line 20 by replacing the comma with a semicolon as in the following example:

```
2Ø        PRINT I;
```

The following results are printed:

```
1  2  3  4  5  6  7  8  9  1Ø  11  12  13  14  15
```

A label inside quotation marks is printed as it appears, and the end of a PRINT statement signals a new line, unless a comma or semicolon is the last symbol. Thus, the following instruction:

```
5Ø        PRINT X, Y
```

prints two numbers and then returns to the next line, while the instruction:

```
5Ø        PRINT X, Y,
```

prints these two values and does not return. The next number to be printed appears in the third zone, after the values of X and Y in the first two zones.

Since the end of a PRINT statement signals a new line,

```
25Ø       PRINT
```

causes the Teletype to advance the paper one line, to put a blank line for vertical spacing of your results, or to complete a partially filled line.

```
5Ø        FOR M=1 TO N
11Ø       FOR J=Ø TO M
12Ø       PRINT B(M,J);
13Ø       NEXT J
14Ø       PRINT
15Ø       NEXT M
```

This program prints B(1,0) and next to it B(1,1). Without line 140, the Teletype would go on printing B(2,0), B(2,1), and B(2,2) on the same line, and then B(3,0), B(3,1), etc. After the Teletype prints the B(1,1) value corresponding to M = 1, line 140 directs it to start a new line; after printing the value of B(2,2) corresponding to M = 2, line 140 directs it to start another new line, etc.

The following instructions:

```
50        PRINT "TIME-"; "SHAR"; "ING";
51        PRINT "ON"; "THE"; "PDP-1Ø"
```

cause the printing of the following:

```
TIME-SHARING ON THE PDP-1Ø
```

Formatting of output can be controlled even further by use of the function TAB, in the form TAB(n), where n is the desired print position (0 through 74).

Insertion of TAB(17) causes the Teletype to move to column 17, as if a tab had been set there. For this purpose, the positions on a line are numbered from 0 through 71, and 72 is assumed to be the 0 position on the next line.

More precisely, TAB may contain any formula as its argument. The value of the formula is computed, and its integer part is taken. This, in turn, is treated modulo 75, to obtain a value from 0 through 74, as indicated above. The Teletype is then moved forward to this position (unless it has already passed this position, in which case the TAB is ignored). For example, inserting the following line in a loop

```
55        PRINT X; TAB(12); Y; TAB(27); Z
```

causes the X values to start in column 0, the Y values in column 12, and the Z values in column 27.

The following rules are used to interpret the printed results:

a. If a number is an integer, the decimal point is not printed. If the integer contains more than eight digits, it is printed in the format as follows.

```
n.nnnnnEp
|  |   |
|  |   └──── E (Exponent) followed by p (power of 10)
|  └──── next five digits
└── first digit
```

For example, 32,437,580,259 is written as 3.24376E+10

b. For any decimal number, no more than six significant digits are printed.

c. For a number less than 0.1, the E notation is used, unless the entire significant part of the number can be printed as a 6-digit decimal number. Thus, 0.03456 indicates that the number is exactly .0345600000, while 3.45600E-2 indicates that the number has been rounded to .0345600.

d. Trailing zeros after the decimal point are not printed.

The following program, in which powers of 2 are printed out, demonstrates how numbers are printed.

```
10      FOR N=-5 TO 30
20      PRINT 2↑N;
30      NEXT N
40      END
RUN

POWERS  11:54   10/20/69


3.12500E-2  6.25000E-2  0.125  0.25  0.5  1  2  4  8  16  32  64  128
256.512 1024  2048  4096  8192  16384  32768  65536  131072  262144
524288 1048576  2097152  4194304  8388608  16777216  33553332
67108864 1.34218 E+8  2.68435 E+8  5.36871 E+8  1.07374 E+9
```

6.2 INPUT STATEMENT

At times, during the running of a program, it is desirable to have data entered. This is particularly true when one person writes the program and saves it on the storage device as a library program (refer to SAVE command, Chapter 9), and other persons use the program and suppy their own data. Data may be entered by an INPUT statement, which acts as a READ but accepts numbers of alphanumeric data from the Teletype keyboard. For example, to supply values for X and Y into a program, type the following:

```
40      INPUT X,Y
```

prior to the first statement which uses either of these numbers. When BASIC encounters this statement, it types a question mark. The user types two numbers, separated by a comma, and presses the RETURN key, and BASIC continues the program. No number can be longer than 8 digits.

Frequently, an INPUT statement is combined with a PRINT statement to make sure that the user knows what the question mark is asking for. You might type in the following statement:

```
20      PRINT "YOUR VALUES OF X,Y, AND Z ARE";
30      INPUT X,Y,Z
```

and BASIC types out the following:

```
YOUR VALUES OF X,Y, AND Z ARE?
```

Without the semicolon at the end of line 20, the question mark would have been printed on the next line.

Data entered via an INPUT statement is not saved with the program. Therefore, INPUT should be used only when small amounts of data are to be entered, or when necessary during the running of the program.

6.3 STOP STATEMENT

STOP is equivalent to GOTO xxxxx, where xxxxx is the line number of the END statement in the program. For example, the following two program portions are exactly equivalent:

```
250     GO TO 999              250     STOP
        .........                      ....
340     GO TO 999              340     STOP
        .........                      ....
999     END                    999     END
```

6.4 REMARKS STATEMENT (REM)

REM provides a means for inserting explanatory remarks in the program. BASIC completely ignores the remainder of that line, allowing you to follow the REM with directions for using the program, with identifications of the parts of a long program, or with any other information. Although what follows REM is ignored, its line number may be used in a GOTO or IF-THEN statement as in the following:

```
100     REM INSERT IN LINES 900-998.  THE FIRST
110     REM NUMBER IS N, THE NUMBER OF POINTS.  THEN
120     REM THE DATA POINTS THEMSELVES ARE ENTERED, BY
200     REM THIS IS A SUBROUTINE FOR SOLVING EQUATIONS


        ......
300     RETURN
        ......
520     GOSUB 200
```

A second method for adding comments to a program consists of placing an apostrophe (') at the end of the line, and following it by a remark. Everything following the ' is ignored except when the line ends in a string (refer to Chapter 8).

6.5 RESTORE STATEMENT

The RESTORE statement permits READing the data in the DATA statements of a program more than once. Whenever RESTORE is encountered in a program, BASIC restores the data block pointer to the first number. A subsequent READ statement then starts reading the data all over again. However, if the desired data is preceded by code numbers or parameters, superfluous READ statements should be used to pass over these numbers. As an example, the following program portion reads the data, restores the data block to its original state, and reads the data again. Note the use of line 570 (READ X) to pass over the value of N, which is already known.

```
100     READ N
110     FOR I=1 TO N
120     READ X
        ......
```

```
200      NEXTI
         ......
560      RESTORE
570      READ X
580      FOR I=1 TO N
590      READ X
         ......
700      DATA.....
710      DATA.....
```

CHAPTER 7
VECTORS AND MATRICES

Operations on lists and tables occur frequently; therefore, a special set of 13 instructions for matrix computations, all of which are identified by the starting word MAT, is used. These instructions are not necessary and can be replaced by combinations of other BASIC instructions, but use of the MAT instructions results in shorter programs that run much faster.

The MAT instructions are as follows:

MAT READ a, b, c	Read the three matrices, their dimensions having been previously specified.
MAT c = ZER	Fill out c with zeros.
MAT c = CON	Fill out c with ones.
MAT c = IDN	Set up c as an identity matrix.
MAT PRINT a, b, c	Print the three matrices. (Semicolons can be used immediately following any matrix which you wish to have printed in a closely packed format.)
MAT INPUT v	Call for the input of a vector.
MAT b = a	Set the matrix b equal to the matrix a.
MAT c = a + b	Add the two matrices a and b.
MAT c = a - b	Subtract the matrix b from the matrix a.
MAT c = a * b	Multiply the matrix a by the matrix b.
MAT c = TRN(a)	Transpose the matrix a.
MAT c = (k) * a	Multiply the matrix a by the number k. The number, which must be in a parentheses, may also be given by a formula.
MAT c = INV (a)	Invert the matrix a.

7.1 MAT INSTRUCTION CONVENTIONS

The following convention has been adopted for MAT instructions: while every vector has a component 0, and every matrix has a row 0 and a column 0, the MAT instructions ignore these. Thus, if we have a matrix of dimension M-by-N in a MAT instruction, the rows are numbered 1, 2, ..., M, and the columns 1, 2, ..., N.

The DIM statement may simply indicate what the maximum dimension is to be. Thus, if we write the following:

```
DIM M(20,35)
```

M may have up to 20 rows and up to 35 columns. This statement is written to reserve enough space for the matrix; consequently, the only concern at this point is that the dimensions declared are large enough to accommodate the matrix. However, in the absence of DIM srarements, all vectors may have up to 10 components and matrices up to 10 rows and 10 columns. This is to say that in the absence of DIM statements, this much space is automatically reserved for vectors and matrices on their appearance in the program. The actual dimension of a matrix may be determined either when it is first set up (by a DIM statement) or when it is computed. Thus the following

```
10      DIM M(20,7)
        -----
50      MAT READ M
```

reads a 20-by-7 matrix for M, while the following:

```
50      MAT READ M(17,30)
```

reads a 17-by-30 matrix for M, provided sufficient space has been saved for it by writing

```
10      DIM M(20,35)
```

7.2 MAT C = ZER, MAT C = CON, MAT C = IDN

The following three instructions:

MAT M = ZER	(sets up matrix M with all components equal to zero)
MAT M = CON	(sets up matrix M with all components equal to one)
MAT M = IDN	(sets up matrix M as an identity matrix)

act like MAT READ as far as the dimension of the resulting matrix is concerned. For example,

```
MAT M = CON(7,3)
```

sets up a 7-by-3 matrix with 1 in every component, while in the following:

MAT M = CON

sets up a matrix, with ones in every component, and of 10-by-10 dimension (unless previously given other dimensions). It should be noted, however, that these instructions have no effect on row and column zero. Thus, the following instructions:

```
10      DIM M(20,7)
20      MAT READ M(7,3)
.......
35      MAT M=CON
70      MAT M=ZER(15,7)
.......
90      MAT M=ZER(16,10)
```

first read in a 7-by-3 matrix for M. Then they set up a 7-by-3 matrix of all 1s for M (the actual dimension having been set up as 7-by-3 in line 20). Next they set up M as a 15-by-7 all-zero matrix. (Note that although this is larger than the previous M, it is within the limits set in 10.) An error message results because of line 90. The limit set in line 10 is (20 + 1) x (7 + 1) = 168 components, and in 90 we are calling for (16 + 1) x (10 + 1) = 187 components. Thus, although the zero rows and columns are ignored in MAT instructions, they play a role in determining dimension limits. For example,

```
90      MAT M=ZER(25,5)
```

would not yield an error message.

Perhaps it should be noted that an instruction such as MAT READ M(2,2) which sets up a matrix and which, as previously mentioned, ignores the zero row and column, does, however, affect the zero row and column. The redimensioning which may be implicit in an instruction causes the relocation of some numbers; therefore, they may not appear subsequently in the same place. Thus, even if we have first LET M(1,0) = M(2,0) = 1, and then MAT READ M(2,2), the values of M(1,0) and M(2,0) now are 0. Thus when using MAT instructions, it is best not to use row and column zero.

7.3 MAT PRINT A, B, C

The following instruction:

MAT PRINT A, B; C

causes the three matrices to be printed with A and C in the normal format (i.e., with five components to a line and each new row starting on a new line) and B closely packed.

Vectors may be used in place of matrices, as long as the above rules are observed. Since a vector like V(I) is treated as a column vector by BASIC, a row vector has to be introduced as a matrix that has only one row, namely row 1. Thus,

```
DIM X(7), Y(0,5)
```

introduces a 7-component column vector and a 5-component row vector.

If V is a vector, then

```
MAT PRINT V
```

prints the vector V as a column vector.

```
MAT PRINT V,
```

prints V as a row vector, five numbers to the line, while

```
MAT PRINT V;
```

prints V as a row vector, closely packed.

7.4 MAT INPUT V AND THE NUM FUNCTION

The following instruction:

```
MAT INPUT V
```

calls for the input of a vector. The number of components in the vector need not be specified. Normally, the input is limited by its having to be typed on one line. However, by ending the line of input with an ampersand (&) before the carriage return, the machine asks for more input on the next line. Note that, although the number of components need not be specified, if we wish to input more than 10 numbers, we must save sufficient space with a DIM statement. After the input, the function NUM equals the number of components, and V(1), V(2), ..., V(NUM) become the numbers inputted, allowing variable length input. For example,

```
5       LET S=Ø
1Ø      MAT INPUT V
2Ø      LET N=NUM
3Ø      IF N=Ø THEN 99
4Ø      FOR I=1 TO N
45      LET S=S+V(I)
5Ø      NEXT I
6Ø      PRINT S/N
7Ø      GO TO 5
99      END
```

allows the user to type in sets of numbers, which are averaged. The program takes advantage of the fact that zero numbers may be inputted, and it uses this as a signal to stop. Thus, the user can stop by simply pushing RETURN on an input request.

7.5 MAT B = A

This instruction sets up B to be the same as A and, in doing so, dimensions B to be the same as A, provided that sufficient space has been saved for B.

7.6 MAT C = A + B AND MAT C = A - B

For these instructions to be legal, A and B must have the same dimensions, and enough space must be saved for C. These statements cause C to assume the same dimensions as A and B. Instructions such as MAT A = A ± B are legal; the indicated operation is performed and the answer stored in A. Only a single arithmetic operation is allowed; therefore, MAT D = A + B - C is illegal but may be achieved with two MAT instructions.

7.7 MAT C = A * B

For this instruction to be legal, it is necessary that the number of columns in A be equal to the number of rows in B. For example, if matrix A has dimension L-by-M and matrix B has dimension M-by-N, then C = A * B has dimension L-by-N. It should be noted that while MAT A = A + B may be legal, MAT A = A * B is self-destructive because, in multiplying two matrices, we destroy components which would be needed to complete the computation. MAT B = A * A is, of course, legal provided that A is a "square" matrix.

7.8 MAT C = TRN(A)

This instruction lets C be the transpose of the matrix A. Thus, if matrix A is an M-by-N matrix, C is an N-by-M matrix.

7.9 MAT C = (K) * A

This instruction allows C to be the matrix A multiplied by the number K (i.e., each component of A is multiplied by K to form the components of C). The number K, which must be in parentheses, may be replaced by a formula. MAT A = (K) * A is legal.

7.10 MAT C = INV(A) AND THE DET FUNCTION

This instruction allows C to be the inverse of A. (A must be a "square" matrix.) The function DET is available after the execution of the inversion, and it will equal the determinant of A. This condition enables the user to decide whether the determinant was large enough for the inverse to be meaningful. In particular, attempting to invert a singular matrix does not cause the program to stop, but DET is set equal to 0. Of course, the user may actually want the determinant of a matrix, and he may obtain it by inverting the matrix and then noting what value DET has.

7.11 EXAMPLES OF MATRIX PROGRAMS

The first example reads in A and B in line 30 and, in so doing, sets up the correct dimensions. Then, in line 40, A + A is computed and the answer is called C. This automatically dimensions C to be the same as A. Note that the data in line 90 results in A being 2-by-3 and in B being 3-by-3. Both MAT PRINT formats are illustrated, and one method of labeling a matrix print is shown.

```
10      DIM A(20,20), B(20,20), C(20,20)
20      READ M,N
30      MAT READ A(M,N),B(N,N)
40      MAT C=A+A
50      MAT PRINT C;
60      MAT C=A*B
70      PRINT
75      PRINT "A*B=",
80      MAT PRINT C
90      DATA 2,3
91      DATA 1,2,3
92      DATA 4,5,6
93      DATA 1,0,-1
94      DATA 0,-1,-1
95      DATA -1,0,0
99      END
RUN

MATRIX  13:48   10/20/69

2       4       6
8       10      12
A*B=

-2      -2      -3
-2      -5      -9
```

The second example inverts an n-by-n Hilbert matrix:

```
1       1/2      1/3 . . .    1/n
1/2     1/3      1/4 . . .    1/n + 1
1/3     1/4      1/5 . . .    1/n + 2
 .       .        . . . .
 .       .        . . . .
 .       .        . . . .
1/n     1/n + 1   1/n + 2     1/2n-1
```

Ordinary BASIC instructions are used to set up the matrix in lines 50 to 90. Note that this occurs after correct dimensions have been declared. A single instruction then results in the computation of the inverse, and one more instruction prints it. Because the function DET is available after an inversion, it is taken advantage of in line 130, and is used to print the value of the determinant of A. In this example, we have supplied 4 for N in the DATA statement and have made a run for this case:

```
5       REM THIS PROGRAM INVERTS AN N-BY-N HILBERT MATRIX
10       DIM A(20,20),B(20,20)
20       READ N
30       MAT A=CON(N,N)
50       FOR I=1 TO N
60       FOR J=1 TO N
70       LET A(I,J)=1/(I+J-1)
80       NEXT J
90       NEXT I
100      MAT B=INV(A)
115      PRINT "INV(A)="
120      MAT PRINT B;
125      PRINT
130      PRINT "DETERMINANT OF A=" DET
190      DATA 4
199      END
RUN

HILMAT   13:52    10/20/69

INV(A)=

16.0001          -120.001         240.003          -140.002
-120.001         1200.01          -2700.03         1680.02
240.003          -2700.03         6480.08          -4200.05
-140.002         1680.02          -4200.05         2800.03

DETERMINANT OF A=1.65342 E-7
```

A 20-by-20 matrix is inverted in about 0.5 seconds. However, the reader is warned that beyond n = 7, the Hilbert matrix cannot be inverted because of severe round-off errors.

7.12 SIMULATION OF N-DIMENSIONAL ARRAYS

Although it is not possible to create n-dimensional arrays in BASIC, the method outlined below does simulate them. The example is of a three-dimensional array, but it has been written in such a way that it could be easily changed to four dimensions or higher. We use the fact that functions can have any number of variables, and we set up a 1-to-1 correspondence between the components of the array and the components of a vector which equals the product of the dimensions of the array. For example, if the array has dimensions 2, 3, 5, then the vector has 30 components. A multiple line DEF could be used in place of the simple DEF in line 30 if the user wished to include error messages. The printout is in the form of two 3-by-5 matrices.

```
10       DIM V(1000)
20       MAT READ D(3)
30       DEF FNA(I,J,K)=((I-1)*D(2)+(J-1))*D(3)+K
50       FOR I=1 TO D(1)
60       FOR K=1 TO D(3)
80       LET V(FNA(I,J,K))=I+2*J+K↑2
90       PRINT V(FNA(I,J,K)),
100      NEXT K
110      NEXT J
112      PRINT
115      PRINT
120      NEXT I
900      DATA 2,3,5
999      END
RUN

3-ARRAY 08:07    10/27/69
4        7        12       19       28
6        9        14       21       30
8        11       16       23       32

5        8        13       20       29
7        10       15       22       31
9        12       17       24       33
```

CHAPTER 8
ALPHANUMERIC INFORMATION (STRINGS)

In previous chapters, we have dealt only with numerical information. However, BASIC also processes alphabetic and alphanumeric information. A string is a sequence of characters, each of which is a letter, a digit, a space, or some other printable character.

Variables may be introduced for simple strings and string vectors, but not for string matrices. Any simple variable, followed by a dollar sign ($), stands for a string; e.g., A$ and C7$. A vector variable, followed by $, denotes a list of strings; e.g., V$(n) where n is the nth string in the list. For example, V$(7) is the seventh string in the list V.

8.1 READING AND PRINTING STRINGS

Strings may be read and printed. For example:

```
10      READ A$, B$, C$
20      PRINT C$; B$; A$
30      DATA ING,SHAR,TIME-
40      END
```

causes TIME-SHARING to be printed. The effect of the semicolon in the PRINT statement is consistent with that discussed in Chapter 6; i.e., with alphanumeric output, the semicolon causes close packing whether that output is in quotes or is the value of a variable. Commas and TABs may be used as in any other PRINT statement. The loop:

```
70      FOR I=1 TO 12
80      READ M$(I)
90      NEXT I
```

reads a list of 12 strings.

In place of the READ and PRINT, corresponding MAT instructions may be used for lists. For example, MAT PRINT M$; causes the members of the list to be printed without spaces between them. We may also use INPUT or MAT INPUT. After a MAT INPUT, the function NUM equals the number of strings inputted.

As usual, lists are assumed to have no more than 10 elements; otherwise, a DIM statement is required. The following statement:

```
10      DIM M$(20)
```

saves space for 20 strings in the M$ list.

In the DATA statements, numbers and strings may be intermixed. Numbers are assigned only to numerical variables, and strings only to string variables. Strings in DATA statements are recognized by the fact that they start with a letter. If they do not, they must be enclosed in quotes. The same requirement holds for a string containing a comma. For example:

```
90      DATA 10,ABC,5,"4FG","SEPT. 22, 1968",2
```

The only convention on INPUT is that a string containing a comma or starting with a non-alphanumeric character must be enclosed in quotes.

With a MAT INPUT, a string containing a comma or an ampersand (&) must be enclosed in quotes. The following example shows the correct format for a response to a MAT INPUT:

```
"MR. & MRS. SMITH", MR. JONES
```

8.2 STRING CONVENTIONS

In employing the three methods of inputting string information into a program (DATA, INPUT or MAT INPUT), leading blanks are ignored unless the string, including the blanks, is enclosed in quotes. Strings (in quotes) or string variables may occur in LET and IF-THEN statements. The following two examples are self-explanatory:

```
10      LET Y$="YES"
20      IF Z7$="YES" THEN 200
```

The relation "<" is interpreted as "earlier in alphabetic order." The other relational symbols work in a similar manner. In any comparison, trailing blanks in a string are ignored, as in the following:

```
"YES" = "YES   "
```

We illustrate these possibilities by the following program, which reads a list of strings and alphabetizes them:

```
10        DIM L$(50)
20        READ N
30        MAT READ L$(N)
40        FOR I=1  TO N
50        FOR J=1  TO N-I
60        IF L$(J)<=L$(J+1)  THEN 100
70        LET A$=L$(J)
80        LET L$(J)=L$(J+1)
90        LET L$(J+1)=A$
100       NEXT J
110       NEXT I
120       MAT PRINT L$
900       DATA 5,ONE,TWO,THREE,FOUR,FIVE
999       END
```

Omitting the $ signs in this program serves to read a list of numbers and to print them in increasing order.

A rather common use is illustrated by the following:

```
330       PRINT "DO YOU WISH TO CONTINUE";
340       INPUT A$
350       IF A$="YES" THEN 10
360       STOP
```

8.3 NUMERIC AND STRING DATA BLOCKS

Numeric and string data are kept in two separate blocks, and these act independently of each other. RESTORE retains both the numerical data and the string data. RESTORE* retains only the numerical data and RESTORE $ only the string data.

8.4 ACCESSING INDIVIDUAL CHARACTERS

In BASIC, it is very easy to obtain the individual digits in a number by using the function INT. It is possible to obtain the individual characters in a string with the instruction CHANGE. The use of CHANGE is best illustrated with the following examples.

```
5         DIM A(65)
10        READ A$
15        CHANGE A$ TO A
20        FOR I=0 TO A(0)
25        PRINT A(I);
30        NEXT I
40        DATA ABCDEFGHIJKLMNOPQRSTUVWXYZ
45        END
RUN

CHANGE    13:55    10/20/69

26   65   66   67   68   69   70   71   72   73   74   75   76   77   78   79   80   81
82   83   84   85   86   87   88   89   90
```

In line 15, the instruction CHANGE A$ TO A has caused the vector A to have as its zero component the number of characters in the string A$ and, also, to have certain numbers in the other components. These numbers are the BASIC code numbers for the characters appearing in the string (e.g., A(1) is 65 - the BASIC code number for A).

The BASIC code for the printable characters is as follows:

Character	BASIC Code No. (Decimal)	Character	BASIC Code No. (Decimal)
Space			
^	32	@	64
!	33	A	65
"	34	B	66
#	35	C	67
$	36	D	68
%	37	E	69
&	38	F	70
'	39	G	71
(	40	H	72
)	41	I	73
*	42	J	74
+	43	K	75
,	44	L	76
—	45	M	77
.	46	N	78
/	47	O	79
0	48	P	80
1	49	Q	81
2	50	R	82
3	51	S	83
4	52	T	84
5	53	U	85
6	54	V	86
7	55	W	87
8	56	X	88
9	57	Y	89
:	58	Z	90
;	59	[	91
<	60	\	92
=	61	]	93
>	62	↑	94
?	63	←	95

Additional symbols useful on output are as follows:

LF (line feed)	10
CR (carriage return)	13

The above list is not complete; there are 128 characters numbered 0 through 127.

The other use of CHANGE is illustrated by the following:

```
10      FOR I=0 TO 5
15      READ A(I)
20      NEXT I
25      DATA 5,65,66,67,68,69
30      CHANGE A TO A$
35      PRINT A$
40      END
```

This program prints ABCDE because the numbers 65 through 69 are the code numbers for A through E.

Before CHANGE is used in the vector-to-string direction, we must give the number of characters which are to be in the string as the zero component of the vector. In line 15, A(0) is read as 5. The following is a final example:

```
5       DIM V(128)
10      PRINT "WHAT DO YOU WANT THE VECTOR V TO BE";
20      MAT INPUT V
30      LET V(0)=NUM
40      CHANGE V TO A$
50      PRINT A$
60      GO TO 10
70      END
RUN

EXAMPLE          13:59    10/20/69

WHAT DO YOU WANT THE VECTOR V TO BE? 40,32,45,60,45,89,90
( -<-YZ
WHAT DO YOU WANT THE VECTOR V TO BE? 32,33,34,35,36,37,38,39,40,41,42,&
? 43,44,45,46,47,48,49,50
 !"#$%&'()*+,-./012
WHAT DO YOU WANT THE VECTOR V TO BE? 4
```

Note that in this example we have used the availability of the function NUM after a MAT INPUT to find the number of characters in the string which is to result from line 40. Giving the input "4" on the last request obtains the response EOT (end of transmission), which turns off the Teletype.

CHAPTER 9

EDIT AND CONTROL

Several commands for editing BASIC programs and for controlling their execution enable you, for example, to:

a. delete lines

b. list the program

c. change or resequence line numbers with set increments

d. save programs on a file-structured storage device (disk or DECtape)

e. replace old programs on the storage device with new programs

f. call in programs from the storage device.

These commands are summarized as follows:

Command	Action
DELETE n	Delete line number n and the contents of the line from user core.
DELETE n,m	Delete line numbers n through m from user core.
LENGTH	Print length of source program (expressed as the number of characters).
LIST	List program with heading.
LIST n	List program with heading, beginning at line number n.
LIST n,m	List program with heading, from line number n through m.
LISTNH LISTNH n LISTNH n,m	Same as LIST, but with heading suppressed.
NEW	BASIC asks for new program name and checks to make certain that it does not already exist.
OLD	BASIC asks for program name and replaces current contents of user core with existing program of that name from the storage device.
RENAME filename	Change name of program currently in user core.
REPLACE	Replace old file of current name with contents of user core.
RUN	Compile and run program currently in core.
RUNNH	Same as RUN, but with heading suppressed.

Command	Action
SAVE	Save the contents of user core as file whose filename is current program name and whose extension is .BAS[†].
SAVE filename	Save user core as filename .BAS[†].
SCRATCH	Delete all program statements from user core.
RESEQUENCE n	Change line numbers to n, n + 10, ...
RESEQUENCE n,,k	Change line numbers to n, n + k, ... Commas are necessary as argument delimiters.
RESEQUENCE n, f, k	Change line numbers from line f upward to n, n + k, ... f must not be greater than n.
SYSTEM	Exit to Monitor.
WEAVE filename	Read program statements from the file named filename.BAS (existing statements in user core are replaced by new statements having same line numbers).
↑C	To stop a running program and enter Monitor level, type ↑C twice.
↑O	To suppress output (typeout), type ↑O.

[†]SAVE commands do not overwrite an existing file of the same name (use REPLACE instead).

CHAPTER 10
DATA FILE CAPABILITY

The data file capability allows information to be written onto the disk for immediate or semipermanent storage. The user can save this information until the disk is refreshed, or he can utilize PIP (Peripheral Interchange Program) to save it permanently on DECtape or paper tape.

With each BASIC program, a user can save up to nine files, each with a different filename. The filename is assigned by the user and must follow the filename rules described in Chapter 4. The extension is assigned by the BASIC compiler and is set to .BAS. (All BASIC files have this extension.) The current date and time are placed into the file directory along with the project-programmer numbers. The file protection key is set to the standard protection when the file is created, indicating protection-protection and write-protection against all users except the owner of the file.

10.1 FILES COMMAND

The FILES command specifies what files are to be read or written. The command format appears as follows:

FILES name1, name2, ..., name9

where name1, name2, ..., name9 are filenames. The filenames may be separated by commas or semicolons. This command may come after any executable command, but must precede any command that is associated with creating a new file or referencing an old file.

I/O channels are assigned consecutively, starting with channel 1, to the files. The names are positional, where name1 corresponds to software channel 1, name2 to software channel 2, and so forth, up to name9 to software channel 9. Since the filenames are positional, commas must precede filenames that are not sequentially ordered, for example:

```
10      FILES ,,S,,R,T
```

indicates that input or output is desired on channel 3 (filename S), channel 5 (filename R), and channel 6 (filename T).

Files are in either read or write mode and are assumed to be initially in read mode. An error message is given if an attempt is made to read a file which does not exist or to read a file which is being written. Examples of the FILES command are as follows:

```
FILES   AAA
FILES   X,Y,Z
FILES   .D,F
```

10.2 SCRATCH COMMAND

The SCRATCH command opens a file for writing. More than one channel may be referenced. The command format is as follows:

```
SCRATCH      #M,#N, #P or
SCRATCH      #M,N,P
```

where M, N, and P are channel specifiers. The # must precede the first channel specifier, but need not be duplicated for subsequent channel specifiers in the same command. This command must be used prior to the writing of a file. Examples of the SCRATCH command are as follows:

```
SCRATCH      #1
SCRATCH      #1, #4
SCRATCH      #1, 4
```

10.3 WRITE COMMAND

The WRITE command causes data to be output to the disk on the specified channel. The data may be an area of storage previously dimensioned, or any information appearing in a PRINT statement. The format of I/O to the disk is the same as the format to the Teletype. The command format is as follows:

```
WRITE        #N, (sequence of variables)
```

where N is a channel specifier. When writing a file, BASIC inserts line numbers, starting with 10 and incrementing by 10. After each line number, BASIC inserts the letter D to separate the line number from the data. When reading a file, BASIC recognizes the nondigit character (D) following the line numbers, and ignores it. Examples of the WRITE command are as follows:

```
WRITE        #2, A(1)
WRITE        #6, Z$
```

The following is an example of the storage of the sines of 1-10 radians in file RRR:

```
10      DIM A(10)
20      FILES RRR
30      SCRATCH #1
40      FOR I=1 TO 10
50      A(I)=SIN(I)
60      WRITE #1,A(I)
70      NEXT I
80      END
```

10.4 RESTORE COMMAND

The RESTORE command opens a file for reading. More than one channel may be referenced. The command format is as follows:

```
RESTORE     #M, #N, #P  or
RESTORE     #M,N,P
```

where M, N, and P are channel specifiers. The main use of the RESTORE command is to reread a file or read a file that has just been written. If the first function in the program is to read an already existing file, the command is not necessary. In other words, if a different program created the files, then a new program does not need the RESTORE command in order to read the files; this is because files are initially in read mode. Examples of the RESTORE command are as follows:

```
RESTORE     #1, #2
RESTORE     #1,2
```

10.5 INPUT COMMAND

The INPUT command causes data to be input from the disk on the specified channel into the specified area. Each command can reference only one channel and, therefore, only one file. The command format is as follows:

```
INPUT       #N, (sequence of variables)
```

where N is a channel specifier. The READ and INPUT commands are equivalent when data files are read from the disk, and the PRINT and WRITE commands are equivalent when data files are written on the disk. Examples of the INPUT command are as follows:

```
INPUT       #2, A(i)
INPUT       #6, Z$
INPUT       #3, B(K)
```

The following example demonstrates access to previously stored values in the file RRR:

```
10        DIM A(10)
20        FILES RRR
30        FOR I=1 TO 10
40        INPUT #1,A(I)
50        PRINT A(I)
60        NEXT I
70        END
```

```
RUNN H

0.841471
0.909297
0.14112
-0.756802
-0.958924
-0.279416
0.656987
0.989358
0.412119
-0.544021
```

10.6 IF END COMMAND

This command provides control in a program when an End-of-File is detected during input from the disk. The command format is as follows:

$$\text{IF END \#N} \begin{Bmatrix} \text{THEN} \\ \text{GO TO} \end{Bmatrix} \text{(line number)}$$

where N is a channel specifier. The line number must follow the rules discussed in Chapter 1. Either THEN or GO TO is acceptable. Examples of the IF END command are as follows:

IF END #1 GO TO 160
IF END #4 THEN 435

APPENDIX A
SUMMARY OF BASIC STATEMENTS

A.1 ELEMENTARY BASIC STATEMENTS

The following subset of the BASIC command repertoire includes the most commonly used commands and is sufficient for solving most problems.

DATA [data list]	DATA statements are used to supply one or more numbers or alphanumeric strings to be accessed by READ statements. READ statements, in turn, assign the next available data in the DATA string to the variables listed. Numeric and alphanumeric data are kept in separate tables and must be accessed by separate READ statements; however, they both may be entered in the same DATA statement.
READ [sequence of variables]	
PRINT [arguments]	Types the values of the specified arguments which may be variables, text, or format control characters.
LET [variable] = [formula]	Assigns the value of the formula to the specified variable.
GO TO [line number]	Transfers control to the line number specified and continues execution from that point.
IF [formula] [relation] [formula] $\left\{\begin{matrix}\text{THEN}\\ \text{GO TO}\end{matrix}\right\}$ [line number]	If the stated relationship is true, then transfers control to the line number specified; if not, continues in sequence.
FOR [variable] = [formula$_1$] TO [formula$_2$] STEP [formula$_3$]	Used for looping repetitively through a series of steps. The FOR statement initializes the variable to the value of formula$_1$ and then performs the following steps until the NEXT statement is encountered. The NEXT statement increments the variable by the value of formula$_3$. (If omitted, the increment value is assumed to be +1.) The resultant value is then compared to the value of formula$_2$. If variable $<$formula$_2$, control is sent back to the step following the FOR statement and the sequenc of steps is repeated; eventually, when variable $\geq$ formula$_3$, control continues in sequence at the step follow ing NEXT.
NEXT [variable]	

ON [x] $\left\{\begin{array}{l}\text{GO TO}\\ \text{THEN}\end{array}\right\}$ [line number$_1$,] [line number$_2$,] .... [line number$_n$]	If the integer portion of x = 1, transfers control to line number$_1$, if x = 2, to line number$_2$, etc. [x] may be a formula.
DIM [variable] (subscript)	Enables the user to enter a table or array with a subscript greater than 10 (i.e., more than 10 items).
END	Last statement to be executed in the program, and must be present.

A.2 ADVANCED BASIC STATEMENTS

GOSUB [line number] Subroutine $\left\{\begin{array}{l}\text{[line number]}\\ \quad\vdots\\ \quad\vdots\\ \text{RETURN}\end{array}\right.$	Simplifies the execution of a subroutine at several different points in the program by providing an automatic RETURN from the subroutine to the next sequential statement following the appropriate GOSUB (the GOSUB which sent control to the subroutine).
INPUT [variable(s)]	Causes typeout of a ? to the user and waits for user to respond by typing the value(s) of the variable(s).
STOP	Equivalent to GO TO [line number of END statement].
REM	Permits typing of remarks within the program. The insertion of short comments following any BASIC statement is accomplished by preceding such comments with an apostrophe (').
RESTORE	Sets pointer back to beginning of string of DATA values.

A.3 MATRIX INSTRUCTIONS

NOTE

The word "vector" may be substitued for the word "matrix" in the following explanations.

MAT READ a, b, c	Read the three matrices, their dimensions having been previously specified.
MAT c = ZER	Fill out c with zeros.
MAT c = CON	Fill out c with ones.
MAT c = IDN	Set up c as an identity matrix.
MAT PRINT a, b, c	Print the three matrices.
MAT INPUT v	Input a vector.

MAB b = a	Set matrix b = matrix a.
MAT c = a + b	Add the two matrices, a and b.
MAT c = a - b	Subtract matrix b from matrix a.
MAT c = a * b	Multiply matrix a by matrix b.
MAT c = TRN(a)	Transpose matrix a.
MAT c = (k) * a	Multiply matrix a by the number k. (k, which must be in parentheses, may also be given by a formula.)
MAT c = INV(a)	Invert matrix a.

A.4 DATA FILE COMMANDS

FILES [sequence of filenames]	Specifies the files that are to be read or written.
SCRATCH [sequence of channel specifiers]	Opens a file for writing.
RESTORE [sequence of channel specifiers]	Opens a file for reading.
WRITE [channel specifier] [sequence of variables]	Causes data to be output to the disk on the specified channel.
INPUT [channel specifier] [sequence of variables]	Causes data to be input to the disk on the specified channel into the specified area.
IF END [channel specifier] $\left\{\begin{matrix}\text{THEN}\\\text{GO TO}\end{matrix}\right\}$ [line number]	Provides control when an end-of-file is detected during input from the disk.

A.5 FUNCTIONS

In addition to the common arithmetic operators of addition (+), subtraction (-), multiplication (*), division (/), and exponentiation (↑), BASIC includes the following elementary functions:

SIN (x)	COT (x)	LOG (x)
COS (x)	ATN (x)	ABS (x)
TAN (x)	EXP (x)	SQR (x)

Some advanced functions include the following:

INT (x)	Find the greatest integer not greater than x.
RND	Generate random numbers between 0 and 1. The same set of random numbers can be generated repeatedly for purposes of program testing and debugging. The statement RANDOMIZE can be used to cause the generation of new sets of random numbers.

SGN (x)	Assign a value of 1 if x is positive, 0 if x is 0, or -1 if x is negative.

Two special functions used with matrix computations are as follows:

(NUM)	Equals number of components following an INPUT.
DET	Equals the determinant of a matrix after inversion.

The user can also define his own functions by use of the DEFine statement. For example,

```
[line number]   DEF    FNC(x) = SIN (x) + TAN(x) - 10
```

(Define the user function FNC as the formula SIN(x) + TAN(x) - 10.)

NOTE

DEFine statements may be extended onto more than one line; all other statements are restricted to a single line.

APPENDIX B
BASIC DIAGNOSTIC MESSAGES

Most messages typed out by BASIC are self-explanatory. BASIC diagnostic messages are divided into three categories:

a. Command errors
b. Compilation errors
c. Execution errors

Following is a complete list of these messages and their meanings.

Command Errors

Message	Explanation
NNN IN LINE MMM	During a RESEQUENCE command, line MMM was found to contain undefined line number NNN.
COMMAND ERROR (LINE NUMBERS MAY NOT EXCEED 99999)	The given RESEQUENCE command is not executed for that reason.
DELETE COMMAND MUST SPECIFY WHICH COMMANDS TO DELETE	A DELETE command had no arguments.
NO ROOM IN DIRECTORY	Could not enter a file to SAVE or REPLACE it.
FILE NOT SAVED	A file which was requested did not exist.
MISSING LINE NUMBER FOLLOWING LINE NNN	During a WEAVE or OLD command, a line without a line number was found in the file. The line is thrown away.
NO SUCH DEVICE	The INIT of a device failed.
DUPLICATE FILE NAME, REPLACE OR RENAME	User tried to SAVE file that exists.
COMMAND ERROR (YOU MAY NOT OVER-WRITE LINES OR CHANGE THEIR ORDER)	The given RESEQUENCE command would have changed the order of lines in the file. The command is ignored.
WHAT?	Catchall command error.

Compilation Errors

Message	Explanation
NO END INSTRUCTION	
STRING VECTOR IS 2-DIM ARRAY	The user managed to do this error despite many other checks.
UNDEFINED FUNCTION -- fn*	The actual function name, not fn*, is typed
FOR WITHOUT NEXT IN NNN	
NO DATA	Program contains READ but not DATA.
DATA NOT IN CORRECT FORM	Incorrect number or string data in: DATA statement, TTY input, TTY MAT input, or MAT READ following an input error. User is invited to retype line.
NESTED DEF IN NNN	DEF within multiline DEF.
FUNCTION DEFINED TWICE IN NN	
VARIABLE DIMENSIONED TWICE IN NN	
END IS NOT LAST IN NN	
FNEND BEFORE DEF IN NN	FNEND occurs, but not in a function DEF.
FNEND BEFORE NEXT IN NN	A FOR occurred in a DEF, but its NEXT did not.
UNDEFINED LINE NUMBER NN IN MM	In line MM, NN is used as a line number. Line number NN does not exist.
ILLEGAL LINE REFERENCE IN NN	BASIC syntax reauired an integer, but user typed something else; e.g., GO TO A.
ILLEGAL LINE REFERENCE MM IN NN	In line NN, line MM was referred to illegally because: a. Line MM is a REM b. The first character in line MM is an apostrophe ('). c. One of the lines NN or MM is inside a function; the other is not inside that function.
ILLEGAL RELATION IN NN	Incorrect IF relation.
MIXED STRINGS AND NUMBERS IN NN	Line NN illegally contains a string variable or literal because: a. No element of this statement may be a string. b. All elements must be strings, but some were not.
NEXT WITHOUT FOR IN NN	
ILLEGAL CONSTANT IN NN	
INCORRECT NUMBER OF ARGUMENTS IN NN	A function was used with the wrong number of arguments.
USE VECTOR, NOT ARRAY IN NN	A letter previously defined as a two-dimensional array is now used in MAT input or CHANGE.
ILLEGAL INSTRUCTION IN NN	First three letters not recognizable; implicit LET is impossible.
ILLEGAL VARIABLE IN NN	
ILLEGAL FORMULA IN NN	Syntax error in an arithmetic formula.

Compilation Errors (Cont)

Message	Explanation
ILLEGAL CHARACTER IN NN	A meaningless character; e.g., DIM #(1).
ILLEGAL FORMAT IN NN	Catchall for other syntax errors.
SYSTEM ERROR	An I/O error, or the UUO mechanism drops a bit, or something similar to those errors.
DELETED	User has ended a line with <ALTMODE> OR <CONTROL -X><RETURN>. The line is ignored.
OUT OF ROOM	Can't get more core to make room for: a. More compilation space. b. Maximum space for all the vectors and arrays. c. Space to store another string during execution.
TOO MANY FILES	A maximum of nine files may be read or written in a program.
ILLEGAL DSK READ IN LINE N	File was never accessed for reading.
FILE NEVER ESTABLISHED-REFERENCED IN LINE N	File was not referenced in a FILES command.
FILE NOT FOUND BY RESTORE COMMAND IN LINE N	File was never written.
ILLEGAL DSK WRITE IN LINE N	File was not opened for writing.
FAILURE ON ENTRY IN LINE N	Channel is not available for SCRATCH command.
EOF IN LINE N	An attempt was made to read data from a file after all data had been read.
BAD DATA INTO LINE N	Input data is not in correct form.

Execution Errors

Message	Explanation
SUBROUTINE OR FUNCTION CALLS ITSELF IN NN	FNA is defined in terms of FNB which is defined in terms of FNA, or a similar situation with FUNCTIONS or GOSUBS.
ON EVALUATED OUT OF RANGE IN NN	The value of the ON index was <1 or > the number of branches.
OVERFLOW IN NN UNDERFLOW IN NN DIVISION BY ZERO IN NN	These all work according to specifications whenever the APRENB UUO is implemented.
RETURN BEFORE GOSUB IN NN	
NOT ENOUGH INPUT--ADD MORE	
INPUT DATA NOT IN CORRECT FORM--RETYPE LINE	
OUT OF DATA IN NN	
TOO MANY ELEMENTS--RETYPE LINE	

Message	Explanation
IMPOSSIBLE VECTOR LENGTH IN NN	In a CHANGE (to string) statement, the zeroth element of the number vector was negative or exceeded its maximum dimension.
NON-ASCII CHAR SEEN IN NN	In a change operation of the same kind, one of the other elements of the number vector was negative or exceeded octal 177.
NO ROOM FOR STRING IN NN	In a CHANGE A$ TO A, the number of characters in A$ exceeds the maximum size of A.
LOG OF NEGATIVE NUMBER IN NN	
SQRT OF NEGATIVE NUMBER IN NN	
TAN OF P1/2 OR COTAN OF ZERO IN NN	
EXP TOO LARGE IN NN	
ZERO TO A NEGATIVE POWER IN NN	
ABSOLUTE VALUE RAISED TO POWER IN NN	
DIMENSION ERROR IN NN	
LOG OF ZERO IN NN	

Book 4

Conversational Programming With AID

CONTENTS

CHAPTER 1
INTRODUCTION

1.1	Gaining Access to the AID System	4-9
1.2	Terminating AID	4-9
1.3	AID Language	4-10
1.3.1	Rules of Form	4-11
1.3.2	Arithmetic Accuracy and Notations	4-11
1.4	Teletype Consoles	4-12
1.5	Correction of Typing Errors	4-13

CHAPTER 2
STEPS AND PARTS

2.1	Direct Steps	4-15
2.2	Indirect Steps	4-15
2.3	Parts	4-15

CHAPTER 3
IDENTIFIERS (VARIABLES)

3.1	Defining an Identifier by a Value (SET and DEMAND Commands)	4-17
3.2	Defining Identifiers by Formulas (LET Command)	4-19
3.2.1	Arithmetic Formulas	4-19
3.2.2	Boolean Expressions (Propositions) Defined by the LET Command	4-20
3.2.3	User Functions Defined by the LET Command	4-20
3.3	Identifier References	4-20
3.4	Indexed Identifiers (Arrays)	4-21

CHAPTER 4
ARITHMETIC OPERATORS, FUNCTIONS, PROPOSITIONS and ITERATIONS

4.1	Arithmetic Operators	4-23
4.2	AID Functions	4-25
4.3	User-Defined Functions	4-29
4.3.1	Examples of User-Defined Functions	4-30
4.4	Propositions	4-31
4.4.1	Conditional Expressions	4-32

CONTENTS (continued)

4.5	Iterative Clauses (Ranges)	4-34
4.5.1	Series of Values	4-34
4.5.2	Incrementation	4-34
4.5.3	Combinations	4-34

CHAPTER 5
AID VERBS

5.1	CANCEL	4-37
	Description	4-37
	Parenthetical CANCEL (CANCEL)	4-37
	Diagnostic Messages	4-38
5.2	DELETE	4-39
	Description	4-39
5.3	DEMAND	4-41
	Description	4-41
	DEMAND.....AS "Any Text" Option	4-41
5.4	DISCARD	4-45
	Description	4-45
	Diagnostic Messages	4-45
5.5	DO	4-46
	Description	4-46
	TIMES Option	4-46
	Range Option (FOR Clause)	4-46
	IF Clause	4-48
	Parenthetical DO (DO......)	4-48
	Diagnostic Messages	4-50
5.6	DONE	4-51
	Description	4-51
	Diagnostic Messages	4-52
5.7	FILE	4-53
	Description	4-53
	Diagnostic Messages	4-53
5.8	FORM	4-54
	Description	4-54

CONTENTS (continued)

5.8 (cont)	Specific Notations	4-54
	Multiple Results on a Single Line	4-55
	Interspersing Text with Results	4-55
	Report-Type Headings	4-56
	Diagnostic Messages	4-56
5.9	GO	4-57
	Description	4-57
	Diagnostic Messages	4-58
5.10	IF Clause	4-59
	Description	4-59
5.11	LET	4-60
	Description	4-60
	Arithmetic Formulas	4-60
	Boolean Expressions (Propositions)	4-60
	User Functions	4-60
5.12	LINE	4-62
	Description	4-62
5.13	PAGE	4-63
	Description	4-63
5.14	QUIT	4-64
	Description	4-64
5.15	RECALL	4-65
	Description	4-65
	Diagnostic Messages	4-65
5.16	RESET TIMER	4-66
	Description	4-66
5.17	SET	4-67
	Description	4-67
5.18	STOP	4-68
	Description	4-68
5.19	TO	4-69
	Description	4-69
	Diagnostic Messages	4-70

CONTENTS (continued)

5.20	TYPE	4-71
	Description	4-71
	Combined TYPE Commands	4-71
	IN FORM Option	4-71
5.21	USE	4-75
	Description	4-75
	Diagnostic Messages	4-75

APPENDIX A
TABLES

A-1	A Glossary of AID Terms	4-78
A-2A	AID Command Summary	4-80
A-2B	File Command Subset	4-84
A-3	AID Character Set	4-85
A-4	AID Diagnostic Messages	4-87

TABLES

4-1	AID Arithmetic Operators	4-23
4-2	AID Functions	4-26

PREFACE

AID (for Algebraic Interpretive Dialogue) is a PDP-10 adaptation of language elements of JOSS[1], the well-known computing service program developed by The RAND Corporation under contract to the United States Air Force.

The system is designed to assist scientists and engineers in solving complex numerical problems. The language is direct and relatively easy to learn. No previous programming experience is needed, either to understand this manual or to use AID at a Teletype console. Commands are typed in the form of imperative English sentences and mathematical expressions are typed, for the most part, in standard notation.

Digital Equipment Corporation is grateful to The RAND Corporation for permission to adapt the language processors of JOSS for the PDP-10. We also wish to express our appreciation to the programmers and computer scientists who developed the system, and to The RAND Corporation and E. P. Gimble of the Air Force's Sacramento Air Materiel Area for the use of RAND publications in the preparation of this AID manual.

[1] JOSS is the trademark and service mark of The RAND Corporation for its computer program and services using that program.

CHAPTER 1
INTRODUCTION

AID is available on all PDP-10 systems and provides each user with a personal computing service, interacting with the user and responding to commands expressed in a simple language via the user's Teletype. AID has proven to be easy and convenient to use in solving both simple and complex numerical problems.

AID is device independent. It provides the user with a facility to create external files for storage of subroutines and data for subsequent recall and use. For accessibility and speed, such files are normally stored on directory devices such as disk or DECtape; however, files may be stored on any retrievable medium such as magnetic tape.

AID runs in approximately 11 K of core memory (with 1K of user data area) and expands to 14K of core (with 4K of user data area) as required. Note that AID will not run on a 16K machine if Monitor occupies more than 5K of core.

1.1 GAINING ACCESS TO THE AID SYSTEM

To gain access to AID, the user must first gain access to the Monitor. In the case of all but the disk monitors, this is accomplished simply by typing ↑C (hold down the CTRL key while striking C). In the case of the PDP-10/50 Monitor, the user must log in (see either the Time-Sharing Monitors: 10/40, 10/50 manual or the PDP-10 System Users Guide, found in the PDP-10 Reference Handbook, order code AIW.

When access to the Monitor is gained, and the Monitor has responded with a period (.), the user types

```
.R AID↵
```

When AID is loaded into core, it responds with the message

```
AID (revision date) AT YOUR SERVICE...
*
```

The asterisk (*) indicates that AID is ready to accept a command from the user.

1.2 TERMINATING AID

AID is terminated and control is returned to Monitor level by typing

↑C↵ (hold down the CTRL key and strike C)

AID can then be re-entered by typing

.CONT↵[1]

or killed by either typing

.KJOB↵

or running another program.

1.3 AID LANGUAGE

Table A-2 contains all AID commands and functions. Each command occupies a single line and is terminated by a carriage return. A period at the end of a command is optional. A command can be entered as a direct command (to be executed immediately) or as an indirect command (to be stored for later execution). Variables in commands are represented by single alphabetic letters, A through Z and a through z, called identifiers. Entire routines can be stored as a series of indirect commands to be executed in a specific order. An expression is defined as one number or identifier (or a combination of numbers and/or identifiers and/or AID functions) which is reducible to a number when AID is called upon to use it. The standard mathematical operators can be expressed in AID as follows:

! !	absolute value (equivalent to the mathematical symbol | |)
[ ]	brackets[2]
( )	parentheses[2]
+	addition
-	subtraction
*	multiplication[3]
/	division
↑	exponentiation[4] (x^3 = x↑3)

The order of precedence for these operations is conventional:

a) ! !, [ ], and ( ) from the innermost pair to the outermost pair

b) ↑ (exponentiation)

c) * (multiplication) and / (division) from left to right within each term

d) + (addition) and - (subtraction).

[1] If AID was performing an iterative process when interrupted by the previously typed ↑C, execution proceeds automatically following the CONT command to Monitor.

If it is desired to have AID halt before continuing, type RE-ENTER instead of CONT. In this case, AID will execute the next step of the interrupted process, type the message "I'M AT STEP m.n" and halt. To continue, type GO.

[2] Brackets and parentheses can be used interchangeably in pairs.

[3] The ampersand (&) will perform multiplication also, however when returning the result, AID will type an asterisk (*) in place of the ampersand.

[4] The up arrow for exponentiation is typed by striking the ↑, N key with SHIFT on Teletype Models 33 and 35. On Teletype Model 37 strike the ~, ∧ without SHIFT. In either case do not use the CTRL key for exponentiation.

Examples:

$a/3*c \quad = \quad \left(\frac{a}{3}\right)c$ not $\frac{a}{3c}$ (left-to-right rule)

$x/y↑3 \quad = \quad \frac{x}{y^3}$ not $\left(\frac{x}{y}\right)^3$ (order of precedence)

Boolean expressions composed of arithmetic statements using the operators

= (equal to), # ($\neq$ not equal to), <= ($\leq$ less than or equal to), >= ($\geq$greater than or equal to), <(less than), >(greater than).

and the negation

not

and connected in turn by logical operators

and, or (inclusive)

are handled by AID.

1.3.1 Rules of Form

a. Only one step (command) can be typed per line and only one line can be used for each step.

b. Each step begins with a verb and terminates with a carriage return. A period at the end of a step is optional.

c. Words, variables (identifiers), and numerals can neither abut each other nor contain embedded spaces; spaces cannot appear between an identifier (when it appears in an array, a formula, or a function) and its associated grouped operators and arguments. Otherwise, spaces can be used freely.

d. When operating via the Teletype Model 37 in upper and lower case mode (entered by typing ↑F - CTRL F), the initial letter of the first word of each command may be typed in upper case. All other letters within the command must be in lower case (with the exception of letters in a character string enclosed by quotation marks, or in the case of identifiers of the range A through Z).

Examples:

Step Number (indirect)	Verb	Arguments	Modifiers
1) *1.23	Type	a, a↑2, a↑3	in form 1 if a>0.↵
2) *1.4	Do	part 2	for c = 5(10)100.↵

1.3.2 Arithmetic Accuracy and Notations

All results are rounded to the nine most significant digits.

All results with a value of less than 10^6 and equal to or greater than .001 are typed by AID in fixed point notation.

Examples:

```
a)  *Type 1/3+2↵
         1/3+2    =              2.33333333
b)  *Type 100**3↵
         100**3   =              1*10↑6
c)  *Type 1/4*1↵
         1/4*1    =               .25
d)  *Type cos(2.5)↵
         cos(2.5) =              -.801143616
```

All other results are typed in <u>scientific</u> notation.

Examples:

```
a)  *Type     365*24*60*60↵
    365*24*60*60  =              3.1536*10↑7       (Read as 3.1536 times 10 raised
                                                   to the 7th power)
b)  *Type (.0005)*   (17)*(.01)↵
    (.0005)*   (17)*(.01)   =    8.5*10↑(-5)       (Read as 8.5 times 10 raised to
                                                   the minus 5th power)
```

1.4 TELETYPE CONSOLES

A Teletype console is the link between the user and AID. A PDP-10 system may be equipped with any one of three Teletype models, Model 33, Model 35, or Model 37. The essential difference between Model 37 and Models 33 and 35 is that Model 37 has both upper and lower case letters[1] (with special characters occupying other keys), while Models 33 and 35 have upper case letters only (typed without use of the SHIFT key - some of the special characters occupy what are normally the upper case positions on the letter keys)[2].

Command examples shown in this publication use both upper and lower case letters (rules governing capitalization are similar to those of the English language, the initial letter of the first word of a command is capitalized). Thus, commands can be typed on the Model 37 exactly as shown on the following pages, while only upper case characters can be typed on Models 33 and 35.

Table A-3 lists the AID character set, corresponding standard mathematical symbols, corresponding JOSS symbols, and the method used to obtain each character on the various Teletype models.

---

[1] The system accepts only upper case letters (typed without use of the SHIFT key) from Teletype Model 37 unless ↑F (CTRL F) is first typed, in which case both upper case and lower case letters are recognized.

[2] The difference between Model 33 and Model 35 is that Model 33 does not have TAB, FORM, or VT (Vertical Tab) mechanisms.

1.5 CORRECTION OF TYPING ERRORS

If the user should make an error while typing a command to AID, he can correct it by one of two methods; (1) he can strike the RUBOUT key once for each character to be erased and then type the correct data, or (2) he can type an asterisk followed by a carriage return to delete the entire line, and type the line over.

1)	`*Type VEC\CEV\VECTOR CALCULATION"↵`	User omitted the quotation mark before the V; he erases C, E, and V by striking the RUBOUT key three times (deleted characters are printed between \\), types the missing quotation mark, and continues.

2)	`*Type "VECTOR CALCULATION*↵`
	`*Type "VECTOR CALCULATION"↵`
		User realizes that he has omitted a space between the e and the quotation mark; he decides to delete the line and retype it.

Should the user type an incorrect command, AID, when it attempts to interpret it, will respond with the message

```
EH ?
*
```

NOTE

When indirect commands are entered, AID merely checks the validity of the step number; the validity of the command is not checked until it is called upon for execution.

CHAPTER 2
STEPS AND PARTS

A user requests AID functions by typing single-line commands called steps. The user can enter a step whenever AID responds with an asterisk (*) typeout. Each step is terminated by a carriage return (↵). Steps can be entered in two ways: (1) as direct steps, or (2) as indirect steps.

2.1 DIRECT STEPS

A direct step is interpreted and executed by AID immediately (following the terminating carriage return typed by the user).

`*Type 2+2 ↵`	User types direct step.
`2+2 = 4`	AID responds immediately by interpreting and executing the step.
`*`	

Direct steps are performed only once each time they are typed, and must be retyped each time the user desires to execute them.

2.2 INDIRECT STEPS

An indirect step is entered by preceding the step with a numeric label containing both an integer and a decimal portion (1.1, 2.53). By preceding a step with a numeric label, the user signals to AID that the step is not to be executed immediately, but is to be stored for later execution as part of a routine. AID files away labeled steps in sequence according to the numeric value of the label or step number. Thus, a step number can be used to indicate that a step is to be inserted into, deleted from, or substituted in a series of previously entered indirect steps. Step numbers can contain a maximum of nine significant digits.

`*1.1 Type "X VALUES" ↵`
`*1.2 Type x ↵`
`*1.3 Type x*2, x/2, x↑2 ↵`
	User types in a 3-step routine for later execution.

`*1.15 Set x=3 ↵`	User inserts a step between steps 1.1 and 1.2 by assigning it a number which falls between these two step numbers.
`*1.3 Type x*2, x/2, x↑2, x↑3 ↵`	User changes step 1.3 by substituting a new step having the same step number.

`*Delete step 1.2 ↵`
`*`
	User deletes step 1.2.

2.3 PARTS

Steps are organized into parts according to the integer portion of their step numbers. All steps with step numbers containing the same value in their integer portion belong to the same part. Thus, all of the steps in the previous example can be referred to as <u>part</u> 1.

```
*Type part 1↵                       User requests AID to type all steps in part 1.
1.1    Type "X VALUES"
1.15   Set x=3
1.3    Type x*2, x/2,x ↑2, x ↑3
*Do part 1↵                         User requests AID to interpret and execute (i.e.,
                                    DO) all steps in part 1.
X VALUES
         x*2 =    6
         x/2 =    1.5
         x↑2 =    9
         x↑3 =   27

*
```

Steps and parts are units which may be entered, changed, deleted, typed out, executed, or filed in (and later recalled from) a file stored on some retrievable I/O medium (e.g., disk or DECtape). In addition, they are available in core storage as stored routines for repetitive execution.

Examples:

a) *Type step 1.1↵
b) *Type part 1↵
c) *Do step 2.3↵
d) *Do part 4↵
e) *File step 3.65 as item 4↵
f) *File part 3 as item 2↵

All steps or parts can be referred to collectively (except by DO).

Examples:

a) *Type all steps↵
b) *Type all parts↵
c) *File all steps as item 8↵
d) *File all parts as item 9↵

CHAPTER 3
IDENTIFIERS (VARIABLES)

An identifier (variable) is used in expressions to represent a variable quantity. In AID, identifiers are represented by single alphabetic characters to which arithmetic or logical values have been assigned. On Teletype Models 33 and 35 (and Model 37 in the upper case mode), 26 unique identifiers are available. However, when the Model 37 Teletype is operated in the upper/lower case mode, 52 unique identifiers (A through Z, and a through z) can be used.

3.1 DEFINING AN IDENTIFIER BY A VALUE (SET AND DEMAND COMMANDS)

A fixed value can be assigned to an identifier by typing

```
*Set x = value
```

NOTE

When this command is typed as a direct command, the verb (SET) may be omitted, e.g.,

```
*x = 95
```

In a SET command, the single-character identifier on the left of the equals sign (=) is not a number, but an identifier being defined (or redefined). The value or expression on the right of the equals sign is a numeric value (or truth value) and must always, if a numeric expression, be immediately reducible to a number.

Examples:

a)	*x = 10 ↵	
b)	*Set x =3.5 ↵	
c)	*y = cos (25)+2 ↵	cos is a standard function provided by AID (see Chapter 4).
d)	*Set a = sqrt(20)+5 ↵	sqrt is also a standard AID function.
e)	*m = false ↵	m is set equal to the value false.

The SET command is a convenient way to shorten a lengthy expression by using identifiers to represent its parts.

Example:

$$\frac{\left(5 + \frac{34}{73}\right)^{2} + \frac{42 - \sqrt{50}}{19}}{\left(5 + \frac{34}{73}\right)}$$

This expression can be simplified and solved as follows.

```
*a = 5+34/73 ↵
*b = [42-sqrt(50)]/19 ↵
*Type (a↑2+b)/a ↵
(a↑2+b)/a =          5.80209589
```

Common algebraic functions (e.g., sqrt, cos, sin, log) are provided in AID for use in expressions (see Chapter 4).

Example:

Define the value of pi (π)

```
*p = arg(-1, 0) ↵                   arg is the AID function of a rectangular coordi-
                                    nate point (see Table 4-2).
*Type p ↵
        p =                         3.14159265
*Type p*36↑2 ↵                      Calculate the area of a circle having a radius
                                    of 36.
    p*36↑2 =                      4071.50407
```

An identifier can also be set to a value, to be typed in by the user prior to execution of the associated routine. This is accomplished by using the DEMAND command, which can be used indirectly only. Execution of a DEMAND command causes a typeout of the specified variable, which is followed by the value to be used, typed by the user.

Example:

```
*1.1 Demand x ↵
*1.2 Demand y ↵
*1.3 Type x*y, (x↑2)*(y↑2) ↵
*Do part 1 ↵
          x =       *4 ↵            AID requests value for x.  User responds by typing
                                    in 4.
          y =       *6 ↵            AID requests value for y.  User responds by typing
                                    in 6.
        x*y =       24
(x↑2)*(y↑2) =      576
```

An identifier can be set to a range of values by the "DO...FORx=1st-value (increment) last-value" command (see "DO", Chapter 5). When this form of the DO command is used, the series of steps is executed repetitively for each requested value, beginning with 1st-value and incrementing it by "increment" following each repetition until "last-value" is reached.[1]

[1] As described in Section 4.5, the range given for a variable can be greatly expanded beyond this simple format. For example,

Do part 1 for x = 1,2,3(2)25(i)2↑t(k)200,500.

In this example, part 1 is performed for x = 1, 2, 3, then in increments of 2 up through 25, then in increments of i up through the value of 2^t, then in increments of k up through 200, and 500.

Example:

```
*1.1  Type x,x↑2, x↑3 ↵
*Do part 1 for x = 2(2)10 ↵          Directs AID to perform step 1.1 for values of x, be-
                                     ginning with a value of 2 and incrementing this value
                                     by 2 until 10 is reached in a series of repetitive
                                     executions.

         x   =        2              AID types out results.
         x↑2 =        4
         x↑3 =        8
         x   =        4
         x↑2 =       16
         x↑3 =       64
         x   =        6
         x↑2 =       36
         x↑3 =      216
         x   =        8
         x↑2 =       64
         x↑3 =      512
         x   =       10
         x↑2 =      100
         x↑3 =     1000
*
```

3.2 DEFINING IDENTIFIERS BY FORMULAS (LET COMMAND)

3.2.1 Arithmetic Formulas

AID can be told how to calculate the value of an identifier rather than associating the identifier with a fixed value. This is done with the LET command. The use of LET causes the identifier on the left of the equals sign to be set to the formula (not necessarily a numeric value) on the right of the equals sign (=).

```
*Let d = sqrt(a) + b + c ↵
*Type formula d ↵
    d:  sqrt(a) + b + c              Note that AID associates a formula, not a numeric
*                                    value, with the identifier d.
```

In the above example, the formula for d is an expression reducible to a number, but this value is not calculated until d is called for. However, before d can be calculated, the user must supply values for all variables in the formula associated with d.

```
*Type d ↵
Error in formula d:  a = ??          User has assigned no value to a.
*a = 9↵
*b = 5↵
*c = 8↵
*Type d ↵
    d =       16
*
```

3.2.2 Boolean Expressions (Propositions) Defined by the LET Command

A second use of the LET command is to define an identifier as being equivalent to the value (true or false) of a proposition, i.e., a Boolean expression composed of arithmetic and logical statements using common relational operators (e.g., =, >, <), the logical negation (not), and logical operators (and, or).

Example:

```
*Set a = true ↵
*b = false↵
*Let c = a and b ↵
*Type c ↵
        c =          false
```

Propositions are discussed in detail in Chapter 4.

3.2.3 User Functions Defined by the LET Command

AID provides many of the common algebraic and geometric functions (sqrt - square root, cos - cosine, log - logarithm, etc.). AID functions are specified in expressions by using the appropriate function mnemonic (sqrt = square root).

A third use of the LET command is to equate an identifier to a user-defined function. Once defined, a user function can be used the same as an AID function.

Example:

```
*Let a(b,c) = (b↑2)+(2*b*c)+(c↑2) ↵          Defines the user function, a.[1]
*Type a(4,10) ↵
        a(4,10) =             196
```

Both AID functions and user-defined functions are discussed further in Chapter 4.

3.3 IDENTIFIER REFERENCES

In addition to an identifier in a formula referring to its associated value or formula, it can also be used to delete, type, or file that value or formula.

Examples:

a)	*Delete a ↵	Delete a and its associated value.
b)	*Delete formula b ↵	Delete b and its associated formula.
c)	*Type c ↵	Type the value of c.
d)	*Type formula d ↵	Type the formula associated with d.
e)	*File e as item 1 ↵	Store e and its associated value on the currently open file (see "FILE") as item 1.

All current identifiers and their associated values or formulas can be referred to collectively.

1 In the function a(b,c), b and c are dummy arguments and do not conflict with variables of the same letter outside of the formula (b and c can be used as identifiers elsewhere).

Examples:

```
a)  *Type all values ↵
b)  *Type all formulas ↵
c)  *Delete all values ↵
d)  *Delete all formulas ↵
e)  *File all values as item 3 ↵
f)  *File all formulas as item 4 ↵
```

3.4 INDEXED IDENTIFIERS (ARRAYS)

Values may be organized into vectors and arrays by using indexed letters for identifiers. The letters may then be used to refer to the arrays. Identifiers defined by formulas may not be indexed. The index or subscript is enclosed in parentheses immediately following the identifier.

Example:

```
*x(1)   =  12 ↵
*x(2)   =   4 ↵
*x(3)   =   6 ↵
*Type x(1), x(2), x(3), x(1)*x(2)*x(3) ↵
    x(1)      =  12
    x(2)      =   4
    x(3)      =   6
x(1)*x(2)*x(3)= 288
```

```
*Type x ↵
    x(1)   =   12
    x(2)   =    4
    x(3)   =    6
```

x refers to all indexed x's; thus, a nonindexed identifier cannot coexist with the same identifier indexed.

Multiple subscripts can be specified for an identifier to create a multidimensional array. An identifier can be indexed by one to ten subscripts, and each subscript may have an integer value in the range -250 through +250.

Examples:

```
a)  *x(1) = 6 ↵
b)  *a(1,2) = 10 ↵
c)  *c(100,50,67) = 130 ↵
```

An individual identifier can be used in only one way at any one time and redefinition deletes any previous definitions. Thus, the definition of an identifier with n dimensions deletes all definitions of the same identifier having other than n dimensions.

Example:

```
*x = 5 ↵
*Type x ↵
    x  = 5
```

The identifier x (unindexed - 0 dimension) is defined as equal to 5.

```
*x(1) = 10 ↵
*x(2) = 20 ↵
*Type x ↵
     x(1)  =  10
     x(2)  =  20
```

The identifier x is now redefined with one dimension (subscript); the unindexed x is deleted.

```
*x(1,1) = 33 ↵
```

The identifier x is now redefined as describing a two-dimensional array; x's having other dimensions are deleted.

```
*Type x ↵
     x(1,1) = 33
*x(1,2) = 44 ↵
*x(2, 1)= 55 ↵
```

Additional x values having the same number of subscripts as the previously defined x are entered; no deletions occur.

```
*Type x ↵
     x(1,1)  =  33
     x(1,2)  =  44
     x(2,1)  =  55
```

NOTE

Undefined elements of the x array in this example can be set to a value of 0 by the use of the command

Let x be sparse.

```
*Type x(2,2) ↵
x(2,2) =  ???
*Let x be sparse ↵
*Type x (2,2) ↵
     x(2,2) =      0
*Type x ↵
     x(1,1) =    33
     x(1,2) =    44
     x(2,1) =    55
x is sparse.
*
```

CHAPTER 4 ARITHMETIC OPERATORS, FUNCTIONS, PROPOSITIONS, AND ITERATIONS

4.1 ARITHMETIC OPERATORS

As discussed in Chapter 1, paragraph 1.3, "AID Language", all standard arithmetic operators can be expressed in AID. These are presented in Table 4-1, in their order of precedence.

Table 4-1
AID Arithmetic Operators

Standard Designation	AID Symbology	Meaning
| x |	! x !	Absolute value of x
[ ]	[ ]	1st level grouping[1]
( )	( )	Second level grouping[1]
x^e	x↑e	The value "x" raised to the power of "e".
a · b, (a)(b), or a X b	a*b	Multiply a times b.
a/b or $\frac{a}{b}$	a/b	Divide a by b.
a + b	a+b	Add a to b.
a - b	a-b	Subtract b from a.

1. Within nested pairs of brackets (or parentheses), the order of evaluation is from the innermost pair outward.

Examples:

```
a)  *x = -5↵
    *y = +2↵
    *Type x+y ↵
          x+y =               -3
    *Type  !x+y! ↵
          !x+y!  =             3
    *Type x+y↑2-15 ↵
       x+y↑2-15 =            -16
    * Type  (x+y)↑2-15 ↵
      (x+y)↑2-15 =            -6
    *Type (x+y)↑(2-15) ↵
    (x+y)↑(2-15) =            -6.27225472*10↑(-7)
```

b)
```
*x  = sqrt(16)+sqrt(9)↲
*y  = sqrt[16+sqrt(9)]↲
*z  = sqrt[sqrt(16+9)]↲
*Type x↲
          x     =              7
*Type y↲
          y     =              4.35889894
*Type z↲
          z     =              2.23606798
```

c) <u>Computing simple interest</u>

r = rate of interest per year (in %)
t = time (in years)
p = principal

$$i = \frac{(p)(r)(t)}{100}$$

```
*Let i = (p*r*t)/100↲
*p = 1000↲
*r = 6↲
*t = 3↲
*Type i↲
          i     =          180
```

d) <u>Computing total accumulated principal and compound interest</u>

a = accumulated principal and interest, compounded annually.
r, t, and p are the same as above.
$a = p(1+r/100)^t$

```
*Let a = p*(1+r/100) ↑ t↲
*p = 1000↲
*r = 6↲
*t = 3↲
*Type a↲
          a     =          1191.016
```

e) <u>Formula for a catenary curve</u>

$$y = \frac{a}{2}\left(e^{\frac{x}{a}} + e^{-\frac{x}{a}}\right)$$

where a is a constant, and
e is Euler's number

```
*Let m  = x/a↲
*Let n  = 0-m↲  (optional)
*Let e  = 2.71828183↲
*Let y  = (a/2)*[(6↑m)+(e↑(0-m))]↲

                 or

*Let y  = (a/2)*[(e↑ m)+(e↑ n)]↲
*1.1 Type y↲
*a = -3↲
*Do part 1 for x = 1(1)5↲
          y     =          -3.1682156
          y     =          -3.69172674
          y     =          -4.62924191
          y     =          -6.08589753
          y     =          -8.22504851
```

Execute part 1 for y with values of x beginning with 1 and incremented by 1 until 5 is reached.

4.2 AID FUNCTIONS

Many common algebraic and geometric functions are provided by AID for use in expressions. Two of the most commonly used functions are

sqrt	SQUARE ROOT
log	NATURAL LOGARITHM

Examples:

```
sqrt(10)
log(x*y)
```

Note that the argument for a function is enclosed in parentheses and immediately follows the function mnemonic.

Table 4-2 lists AID functions in alphabetic order. The symbols x and y represent any expression reducible to a number and are the arguments of the function. The variable i is a dummy variable and does not affect any real identifier denoted by the same alphabetic character.

Table 4-2
AID Functions

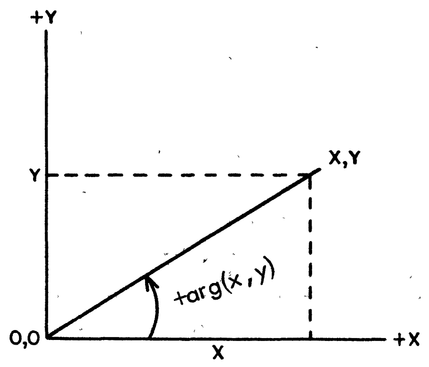
Function	Description
arg(x,y) 	The ARGUMENT function takes two arguments (x,y) and computes the angle between the +x axis of the x,y plane and the line joining point 0,0 and point x,y. The result is in radians arg(x,y) The value of arg (0,0) is 0. The range of arg is 0 through 2π or $-\pi$ through π.
cos(x)	The COSINE function requires one argument, assumed to be in radians. $\mid x \mid$ must be < 100.
dp(x)	The DIGIT PART function. dp(13456.5432) = 1.34565432
exp(x)	The EXPONENTIAL function: e^x , where e is Euler's number (2.718281828). The argument (x) must fulfill the requirement that $e^x<10^{100}$(i.e., x must be less than 230.25851). If $e^x<10^{-99}$, the result is 0.
first(i=range...: i proposition)	The FIRST function requires two arguments: (a) an iterative clause (see paragraph 4.5) and (b) a proposition containing i as an index. The result is the first value of index i to satisfy the proposition.
fp(x)	FRACTION PART function. fp(13456.5432) = .5432
ip(x)	INTEGER PART function. ip(13456.5432) = 13456

Table 4.2 (Cont)
AID Functions

Function	Description
log(x)	NATURAL LOGARITHM function. The argument (x) must be greater than zero.
max(i=range...:...i expression...)	The MAXIMUM function requires two arguments: (a) an iterative clause (see paragraph 4.5), and (b) an expression containing a function of i. The expression is computed iteratively for each value of i, and the result (largest value) is typed out.[1]
min(i=range...:...i expression...)	The MINIMUM function requires two arguments: (a) an iterative clause (see paragraph 4.5), and (b) an expression containing a function of i. The expression is computed iteratively for each value of i, and the result (smallest value) is typed out.[1]
prod(i=range...:...i expression..)	The PRODUCT function requires the same two types of arguments as the MAXIMUM, MINIMUM and SUM functions. The expression is computed iteratively for each value of i, and the result (product of all the iterations) is typed out.[1]
sgn(x)	The SIGNUM function. The value of a signum function of an argument greater than zero is +1, of an argument equal to zero is 0, of an argument less than zero is -1.
sin(x)	The SINE function requires one argument, assumed to be in radians. $\mid x \mid$ must be <100.
sqrt(x)	The SQUARE ROOT function. The argument (x) must be equal to or greater than zero.
sum(i=range...:..i expression..)	The SUM function requires the same two types of arguments as the MAXIMUM, MINIMUM, and PRODUCT functions. The expression is computed iteratively for each value of i, and the result (sum of all iterations) is typed out.[1]

[1]The iterative clause and i function can, in all of these cases, be replaced by a simple series of values for i.
Example: max(5,-4.3,y,x↑2)
See Section 4.5.

Table 4-2 (Cont)
AID Functions

Function	Description
tv (proposition)	The TRUTH VALUE function requires one argument, a proposition, and converts this argument into a numeric value: 1, if the proposition is true; 0, if the proposition is false.
xp(x)	The EXPONENT PART function. xp(13456.5432) = 4 i.e., 13456.5432 = 1.34565432*10 ↑ 4

Examples:

```
          *a   = 10↵
          *b   = 12↵
          *c   = -2.5↵
          *d   = 100↵
          *e   = 1.325↵
          *f   = 10.435↵
          *i   = 25↵

      a)  *Let z    = sum(i=0(10)100:i*2)↵
          *Type z↵
                   z =             1100
      b)  *Type sum (a,b,c,d,e,f,i,z)↵
          sum (a,b,c,d,e,f,i,z)  =  1256.26

      c)  *Let z = prod (i=1(1)5:i ↑ 2)↵
          *Type z↵
                   z =             14400

      d)  *Let z  = max(i=-15(1)15:(i ↑ 2)-(-5*i))↵
          *Type z↵
                   z =               300

      e)  *Let z = min (i=-15(1)15:(i ↑ 2)-(-5*i))↵
          *Type z↵
                   z =                -6

      f)  *Type min(a,b,c,d,e,f,i)↵
          min(a,b,c,d,e,f,i) =        -2.5

      g)  *Type arg(-1,0)↵
              arg(-1,0) =              3.14159265

      h)  *Type arg(c,a)↵
              arg(c,a) =               1.81577499

      i)  *Type sin(10)↵
              sin(10) =                -.544021111

      j)  *Type cos(a)↵
              cos(a) =                 -.839071529

      k)  *Type sin((a*e)-i)↵
          sin((a*e)-i) =                .728664976
```

NOTE

The i in (a) is a dummy variable and in no way relates to the i in (b). The latter is an identifier and refers to the variable and defined above.

```
l)   *Type exp(.346) ↵
       exp(.346)  =                 1.41340262
m)   *Type dp(e+f) ↵
          dp(e+f)  =                1.176
n)   *Type fp(e+f) ↵
          fp(e+f)  =                 .76
o)   *Type ip(e+f) ↵
          ip(e+f)  =               11
p)   *Type log(650) ↵
         log(650)  =                6.47697236
q)   *Type log(e+f) ↵
         log(e+f)  =                2.46470394
r)   *Type sgn(d-(b↑2)+i) ↵
     sgn(d-(b↑2)+i) =              -1
s)   *m = -5↵
     *n = 3↵
     *Type tv((m>=n) or (m=0) or (m<0) and (m>-4)) ↵
     tv((m>=n) or (m=0) or (m<0) and (m>-4)) = 0
t)   *Type sqrt(a+b+c+d+e) ↵
     sqrt(a+b+c+d+e) =                 10.9920426
u)   *1.1 Let a(x) = x↑2-20 ↵
     *Do step 1.1 for x=1(1)30 ↵                        Set up a table (or array) of 30 items
                                                        calculated according to the formula
                                                        given in step 1.1.

     *Type a(25) ↵
          a(25) =                605
     *Type first(I=1(1)30:a(I)=0) ↵
     first(I=1(1)30:a(I)=0) = ???                       No such value found in table.
     *Type first(I=1(1)30:a(I)>700) ↵
     first(I=1(1)30:a(I)>700) =     27
     *Type a(27) ↵
          a(27) =                709
```

4.3 USER-DEFINED FUNCTIONS

Functions not included in AID can easily be defined for repetitive use.

As discussed in Chapter 3, the LET command is used to equate an identifier to some user-defined function. Following this function identifier, up to ten dummy arguments (enclosed as a group in parentheses) can be specified; these are replaced by actual arguments when the function is to be used. Dummy arguments are also represented by single alphabetic characters, but the use of a letter as a dummy in no way affects the use of that same letter as an identifier. Following the dummy arguments, an equals sign and the expression representing the user function are typed.

f(a,b,c,....) = expression

f = function identifier (any single alphabetic character)

(a,b,c,....) = dummy arguments (also single alphabetic characters)

expression = the arithmetic expression representing the user function

Arguments supplied for functions can themselves be functional.

4.3.1 Examples of User-Defined Functions

```
*Let a(b,c) = sqrt(b*c) + b↑2 + c↑2 ↵
```

Define the user function a, with two dummy arguments b and c, as being equivalent to the formula

sqrt(b*c)+b↑2+c↑2

```
*Type a(120.555,32.076) ↵
a(120.555,32.076) =           15624.5624
```

Use the newly-defined function by specifying two actual arguments in place of the dummy arguments, b and c.

```
*Type a ↵
    a(b,c): sqrt(b*c) + b↑2 + c↑2
```

Note that a is equated to the formula, not a value, since a alone is not an expression.

```
*Type formula a ↵
    a(b,c):  sqrt(b*c) + b↑2 + c↑2
```

Same typeout.

```
*Type a(b,c) ↵
          b = ???
```

No values have been specified for the identifiers (not dummy arguments), b and c.

```
*b = (4↑6)/9↵
*c = 5.23↵
*Type a(b,c) ↵
     a(b,c) =                207202.264
```

Many common functions can be defined as user functions, as shown below.

Tangent

```
*Let T(a) = sin(a)/cos(a) ↵
*Type T(10) ↵
       T(10) =                    .648360828
```

Arc cos

```
*Let F(a) =arg(a, sqrt(1-a↑2)) ↵
*Type F(.10) ↵
     F(.10) =                    1.47062891
```

Arc cot

```
*Let C(a) = arg(a,1) ↵
*Type C(10) ↵
       C(10) =                    .0996686522
```

Arc csc

```
*Let S(a) = arg(sqrt(1-1/a↑2),1/a) ↵
*Type S(10) ↵
       S(10) =                    .100167421
```

Arc sec

```
*Let K(a) = arg(1/a,sqrt(1-1/a↑2)) ↵
*Type K(10) ↵
       K(10)  =                  1.47062891
```

Arc sin

```
*Let N(a) = arg(sqrt(1-a ↑ 2),a)↵
*Type N(.10)↵
     N(.10) =                    .100167421
```

Arc tan

```
*Let T(a) = arg(1,a)↵
*Type T(10)↵
     T(10) =                    1.47112767
```

Log to base 10

```
*Let L(a) = log(a)/log(10)↵
*Type L(25.38)↵
     L(25.38) =                 1.40449162
```

Derivative of a function of a variable

```
*Let D(a) = (F(a+.005)-F(a-.005))/.01↵
*Let F(a) = 3*a ↑ 3-4*a ↑ 2+2*a+5↵
*Type D(4)↵
     D(4) =                  114
```

4.4 PROPOSITIONS

As discussed in Chapter 3, propositions are Boolean expressions composed of arithmetic or logical statements using the relational operators.

= (equal), # (not equal), >(greater than), <(less than),

>= (greater than or equal to), <= (less than or equal to)

the negation

not

and the logical operators

and

or

A proposition has either of two possible values: true or false.

Example:

```
*x = true↵
*y = false↵
*Let z = (x) and (y) or (x) and (100 >sqrt(959))↵
*Type z↵
     z =                     true
```

The order of execution within a proposition is:

a) evaluation of expressions
b) ( ) Within nested pairs of parentheses, the order of evaluation is from the innermost pair outward.
c) relational operations

d) not

e) and

f) or

A series of relational operations is assumed to be an and chain if no logical operator intervenes.

a=b>c<d is equivalent to a=b and b>c and c<d

The truth value (tv) function (see "AID Functions") converts the value of a proposition to a numeric value (true = 1, false = 0) and allows it to be used as an expression, since it is then reducible to a numeric value.

```
*Set x = true ↵
*Let y = (x) and (sqrt(100)>sqrt(30*5-20)) ↵
*Type tv(y) ↵
        y =                       0
*Type 24+tv(x) ↵
   24+tv(x)    =                  25
```

4.4.1 Conditional Expressions

A conditional expression allows an expression (e.g., a variable) to have different values depending upon which of a number of conditions is true. It is composed of a series of clauses separated by semicolons, with each clause made up of a proposition followed by a colon followed by an expression. The entire conditional expression must be enclosed by parentheses or brackets.

(proposition:expression; proposition:expression;....)

Example:

Express the function:

```
If x>0, C(x) = x↑2;
if x=0, C(x) =0;
if x<0, C(x) = x.
```

```
*Let C(x) = (x>0:x↑2;x=0:0;x<0:x) ↵
*Type C(5), C(-10), C(0), C(10) ↵
          C(5) =              25
        C(-10) =             -10
          C(0) =               0
         C(10) =             100
```

If the last expression is to be true for all cases which do not satisfy any of the stated conditions, the expression can be typed without a preceding proposition. For example, in the case above, the user could have typed:

```
*Let C(x) = (x>0:x↑2;x=0:0;x) ↵
```

NOTE

Every possible combination of the variable must be provided for, either by explicitly stating a conditional expression and a proposition for it, or by simply specifying a terminating expression to be executed for all cases which do not satisfy any of the explicitly stated propositions. If this provision is not made, and an unprovided-for condition occurs, AID responds with the message

ERROR IN FORMULA X

A conditional expression can be used to perform a table lookup for all items whose values satisfy one or more conditions.

Example:

```
*1.1 Set A(x) = x↑2 + x*.5 - 5*x ꝺ
```

Generates a 35-item table.

```
*Do step 1.1 for x = 1(1)35 ꝺ
*Type A(20) ꝺ
        A(20) =          310
*Type A(3) ꝺ
         A(3) =          -4.5
*Let F(x) = (x<0:x;x>700:x;fp(x)>0 and x>300:x; ← ) ꝺ
```

Find all values which are (1) less than zero; (2) greater than 700; or (3) greater than 300 and have a fractional part which is nonzero. If x is none of these, perform a line feed, carriage return (indicated by the ← symbol).

```
*1.1 Type F(A(i)) ꝺ
*Do step 1.1 for i = 1(5)35 ꝺ
```

Test every fifth item in the table.

```
     F(A(i)) =            -3.5


     F(A(i)) =            346.5

     F(A(i)) =            821.5
     F(A(i)) =           1067.5
```

Values in tested items which do not satisfy any of the three propositions result in line feed/carriage return (because of the terminating ← symbol in the conditional expression).

```
*Let E(x) = (fp(x/2)=0:x; ← ) ꝺ
```

Find all even-numbered values in the table.

```
*1.1 Type E(A(i)) ꝺ
*Do step 1.1 for i = 1(1)35 ꝺ
```

Test every item in the table.

```
     E(A(i)) =            -2

     E(A(i)) =            28

     E(A(i)) =            90

     E(A(i)) =           184

     E(A(i)) =           310

     E(A(i)) =           468

     E(A(i)) =           658

     E(A(i)) =           880
```

4.5 ITERATIVE CLAUSES (RANGES)

Iterative clauses are used with the DO command and with the functions FIRST, MAX, MIN, PROD, and SUM. In both cases, the iterative clause specifies a range of values to be acted upon by the command or function.

4.5.1 Series of Values

One format of an iterative clause lists the individual values which make up the range:

$$n, n_1, n_2, n_3, \ldots\ldots.$$

For example,

Do part m for x = range Do part 1 for A = 1, M, 100, 50, -25, x↑3	Part 1 will be executed for each of the individual values of A.
Type sum(x=range) Type sum(A = -4.6, M*N, 240.5,C)	The SUM function is performed on all values listed and the result (the SUM of all values) typed out.

4.5.2 Incrementation

The range of values for a variable can also be expressed as a first value, an incremental value, and an ending value. As a result, the variable values range from the first value upward in steps of the specified increment until the ending value is reached. The ending value is always taken as the last value in the range, even though the incremental steps may not hit it exactly.

The general form of an incremental iterative clause is:

x = first-value(increment)ending-value

For example:

Do step m.n for x = range Do step 2.3 for A = 1(2)12	Step 2.3 will be executed for each individual value of A: 1,3,5,7,9,11,12
Type sum(x=range) Type sum(A = -50(B)C)	The SUM of all values of A, as indicated by the range, is calculated. This range begins with -50 and continues in increments of B until C is reached.

4.5.3 Combinations

A range can be expressed as a combination of value series and increments.

x = a(b)c,d,e(f)g(h)i,j,k

The range of x values begins with a, continues in increments of b through c, then d, e, then in increments of f until g is reached, then in increments of h until i is reached, then j and k.

For example,

Do part 3 for W = 20(Y)50(50)500, 1000(100)Z

Part 3 will be performed for all values of W, beginning with 20, continuing in increments of Y through 50, then continuing in increments of 50 through 500, and from 1000 in increments of 100 through Z.

Type sum(W = A(30)B, C, 800(D)1500)

The SUM function will be applied to all values of W, beginning with A and continuing in increments of 30 through B, then C, followed by 800 through 1500 in increments of D.

CHAPTER 5
AID VERBS

This chapter contains a summary of AID verbs, their command formats, optional features, and examples of usage. Some of these verbs (e.g., TYPE, DO) have appeared frequently in examples in previous chapters; others (e.g., LET, SET) have already been described extensively and are included here only as a review.

Some verb descriptions include diagnostic messages which are associated with a specific command or group of commands. A complete list of diagnostic messages can be found in Table A-4.

5.1 CANCEL

DESCRIPTION

The CANCEL command cancels a currently stopped (interrupted) process, if the user does not desire to resume execution.[1]

CANCEL is the antithesis of the GO command.

CANCEL also releases any immediate memory currently assigned to the interrupted execution.

NOTE

The CANCEL command does not, however, delete any commands, formulas, variables, etc., associated with the interrupted process.

The CANCEL command can be given directly only.

Parenthetical CANCEL (CANCEL)

Typing

(CANCEL)

cancels any currently stopped process that was initiated by a parenthetical DO.

[1] An interrupted process is automatically cancelled whenever the user types a direct DO command to initiate another part or step.

EXAMPLE

```
*1.1 Let x = .....↵
 :
*2.10 Type....↵
*Do part 1 ↵
Error at step 2.5: illegal set of values for iteration.
*Cancel. ↵                         User does not desire to correct step and resume
                                   execution.
*
```

DIAGNOSTIC MESSAGES

DON'T GIVE THIS COMMAND INDIRECTLY	The CANCEL command can be given directly only (with no step number preceding it).

DELETE

DESCRIPTION

The DELETE command erases a step, part, form, value, or formula from immediate storage and frees that storage for some other use. This command should be used frequently to delete routines, tables, and other items which are no longer needed. By doing this, unnecessary waste of storage and possible storage overflow can be avoided.

Delete a

Delete identifier a and its associated value(s) from immediate storage. If identifier a is a subscripted variable, the entire a array is deleted.

Delete a(b,..)

Delete the particular array item, a(b,..), and its associate value from immediate storage.

Delete step m.n

Delete the step numbered m.n.

Delete part m

Delete part m (all steps having numbers whose integer portion is m).

Delete formula a

Delete the formula associated with a.

Delete form m

Delete form m. (See "FORM" and "TYPE" for an explanation of forms.)

$$\text{Delete all}\left\{\begin{array}{l}\text{values}\\ \text{steps}\\ \text{parts}\\ \text{formulas}\\ \text{forms}\end{array}\right\}$$

Delete all entries of the named type.

Delete all

Delete all entries.

Several individual DELETE commands can be combined into one. For example,

Delete x, form 3, formula b, all parts.

EXAMPLES

```
*1.1 Type "STEP A"                                  Type entries into immediate storage.
*1.2 Type a+b in form 1
*1.3 To step 2.1
*2.1 Type a↑2+b↑2
*2.2 Type "END"
*Let a = 10+b
*b = 25
*Form 1:
*←←←←←←•←←
*Do part 1                                          Test routine.
STEP A
    60.00
    a↑2+b↑2 =                 1850
END
*Delete b                                           Delete identifier b and its associated
*Do part 1                                          value.  Attempt to use b.
STEP A
Error at step 1.2 (in formula A):  b = ???
*Delete step 2.1                                    Delete step 2.1.
*Do step 2.1                                        Attempt to execute it.
I can't find the required step.
*Delete a                                           Delete formula a.
*Type a                                             Attempt to type it.
a = ???
*Delete all                                         Delete remaining entries.
*Type all                                           Test that all have been deleted.
*
```

DEMAND

DESCRIPTION

DEMAND causes AID to type out a request for a user-supplied value during execution of a routine. The DEMAND command can be given indirectly only.

Demand a.

AID types out a request for the value of a.

```
*1.1 Demand a
*1.2 Type.....
*Do part 1
            a =        *(user types value here)
```

Demand a(b,..).

AID types out request for the value of the subscripted variable a(b,..).

```
*1.1 Demand M(3,5,7)
*1.2 Type.....
*Do part 1
    M(3,5,7) =         *(user types value here)
```

DEMAND.....AS "ANY TEXT" OPTION

Demand a as "any text".

AID types "any text" to request a value for a.

```
*1.1 Demand p as "NUMBER OF SAMPLES WANTED"
*1.2 Type....
*Do part 1

     NUMBER OF SAMPLES WANTED =        *(user types value here)
```

Demand a(b,..) as "any text".

AID types "any text" to request a value for the subscripted variable a(b,..).

```
*1.1 Demand y(3) as "MAXIMUM SPEED"
*1.2 Type.....↵
*Do part 1 ↵
    MAXIMUM SPEED =          *(user types value here)↵
```

Depending upon the use of the variable specified, values requested by a DEMAND command can be entered in the form of

a) a numeric expression (e.g., a number in fixed or floating point notation, or an identifier representing a numeric value),

b) a formula, or

c) a Boolean value (true or false).

NOTE

Only one variable can be specified in each DEMAND command.

EXAMPLES

a)

```
*x=25 ↵
*y=50.25↵
*z=16.4↵
*1.1 Type "CONVERSION OF POUNDS TO KILOGRAMS"
*1.2 Demand a ↵
*1.3 Type a,b in form 1 ↵
*1.4 To step 1.2 ↵
*Let b=.45359*a ↵
*Form 1:↵
*
 ←←←←←←.←←←←←POUNDS = ←←←←←←.←←←←←KILOGRAMS↵
*Do part 1 ↵
CONVERSION OF POUNDS TO KILOGRAMS
              a = *25.8↵
      25.8000 POUNDS =      11.7026 KILOGRAMS
              a = *100.543↵
     100.5430 POUNDS =      45.6053 KILOGRAMS
              a = *5567.98↵
    5567.9800 POUNDS =    2525.5801 KILOGRAMS
              a = *↵                          Carriage return by user
I'm at step 1.2.                              terminates iterations.

*1.2 Demand a as "POUNDS"
*Do part 1 ↵
CONVERSION OF POUNDS TO KILOGRAMS
        POUNDS = *25.8↵
      25.8000 POUNDS =      11.7026 KILOGRAMS
        POUNDS = *↵
I'm at step 1.2.
*
```

b)
```
*Let a = m and n
*Let b = m or n
*1.1 Demand m
*1.2 Demand n
*1.3 Type tv(a) in form 1
*1.4 Type tv(b) in form 2
*1.5 To step 1.1
*Form 1:
*    Logical AND:    ←
*Form 2:
*    Logical OR:     ←
*Do part 1
            m = *true
            n = *false
     Logical AND:  0
     Logical OR:   1

            m = *false
            n = *false
     Logical AND:  0
     Logical OR:   0
            m = *not true
            n = *not false
     Logical AND:  0
     Logical OR:   1
            m = *
I'm at step 1.1.
*
```

Carriage return terminates iterations.

c)
```
*1.1 Do part 2 for B = 1(1)3
*1.2 Type A
*2.1 Do part 3 for C = 1(1)5
*3.1 Demand A(B,C)
*3.2 Set A(B,C) = sqrt(A(B,C))
*Do part 1
          A(1,1) = *30
          A(1,2) = *65
          A(1,3) = *4
          A(1,4) = *50
          A(1,5) = *43.55677
          A(2,1) = *32
          A(2,2) = *1
          A(2,3) = *45.99
          A(2,4) = *29
          A(2,5) = *22.3333
          A(3,1) = *56.77
          A(3,2) = *66.7777
          A(3,3) = *99
          A(3,4) = *100
          A(3,5) = *1234.33
          A(1,1) =               5.47722558
          A(1,2) =               8.06225775
          A(1,3)                 2
          A(1,4) =               7.07106781
```

```
A(1,5) =          6.5997553
A(2,1) =          5.65685425
A(2,2) =          1
A(2,3) =          6.78159273
A(2,4) =          5.38516481
A(2,5) =          4.7258121
A(3,1) =          7.53458692
A(3,2) =          8.17176236
A(3,3) =          9.94987437
A(3,4) =         10
A(3,5) =         35.1330329
```

DISCARD

DESCRIPTION

DISCARD deletes an item from the external storage file currently in use.

Discard item m (code).

Erase item #m (where m can be in the range 1 through 25) from the currently open external storage file and make the item available for some other use.

Immediate storage is not affected in any way.

Code is optional for documentation purposes only and is ignored by AID; however, code, if used, cannot exceed five characters in length.

EXAMPLE

```
*Discard item 20 ↳
Done.
```

Item 20 of the external storage file currently in use has been cleared successfully, as evidenced by the AID message, DONE.. Item 20 can now be used for storing some other data, via the FILE command.

DIAGNOSTIC MESSAGES

I CAN'T FIND THE REQUIRED ITEM.	The specified item cannot be found in the currently open file. Either the wrong file is open, or the item number is incorrect.
ITEM NUMBER MUST BE POSITIVE INTEGER <=25.	An invalid item number was given.
YOU HAVEN'T TOLD ME WHAT FILE TO USE	A DISCARD command was attempted before an external storage file was opened via a USE command.

DO

5.5

DESCRIPTION

The DO command executes an indirect step or part. DO is completed when either (1) in a noniterative operation, the last step in the sequence has been completed, or (2) in an iterative operation, the last iteration has been completed.[1] If the DO command is a direct step, control returns to the user at the completion of the DO; if the DO command is indirect, control returns to the step following the DO. If the step or part being executed contains imbedded DO or TO commands, they are executed normally.

Do step m.n

Execute step #m.n and return control as described above.

TIMES Option

Do step m.n, p times

Execute step #m.n the number of times specified by integer p and return control as described above. Note that a comma must immediately follow the step number.

RANGE Option (FOR Clause)

Do step m.n for x = range

Execute step m.n iteratively for each specified value of x as indicated by range (see Section 4.5).

When the range is satisfied, control is returned as described above.

Do step 1.2 for x = 1(1)5	Execute step 1.2 iteratively, beginning with an initial x value of 1 and incrementing x by 1 prior to each iteration until the maximum value of 5 is reached.
Do step 5.25 for a = -10.25(.25)4.50	Execute step 5.25 iteratively, beginning with an initial a value of -10.25 and incrementing a by the value .25 each time until the maximum value of 4.50 is reached.
Do step 1.3 for m = 1,-2.5,100,-43.666	Execute step 1.3 for each of the four specified m values.

[1] Remember that steps are always executed according to the numerical sequence of their step numbers, regardless of the order in which the steps were originally entered.

Do step 10.6 for p = 200, -30.667, -2.3(.1)1.9, 5.75	Execute step 10.6 for three values of p (200, -30.667, and 5.75) and for a range of values of p (-2.3 through 1.9, in increments of .1).
Do step 1.3 for m = 1(4)26(5)50(25)155	Perform step 1.3 iteratively for m = 1 to 26 in increments of 4 26 to 50 in increments of 5 50 to 150 in increments of 25. Thus, the values of m will be 1,5,9, 13, 17, 21, 25, 26, 31, 36, 41, 46, 50, 75, 100, 125, 150, and 155 for the 18 iterations of step 1.3.

Do part m

Execute part m (all steps having the value m as the integer portion of their step number). All steps are executed in numeric sequence; any jump (via a DO or TO) to a step which is outside part m is handled correctly. Control is returned as described above.

Do part m, p times

Execute part m the number of times specified by integer p and return control as described above.

Do part m for x = range

Execute part m iteratively for each specified value of x in the same manner as described under "Do step m.n for x = range".

The FOR clause of a DO command is interpreted only once, at the point where the DO command is encountered; therefore, if a variable specified within the FOR clause is changed during execution of the DO-initiated routine, the change has no effect on the performance of the FOR clause. The number of iterations performed and the setting of the variable at the beginning of each iteration is the same as if no modification of the variable were performed by the routine.

```
*1.1 Do part 2 for x=1(1)5 Δ
*2.1 Type x Δ
*2.2 Set x = x+100 Δ
*2.3 Type x Δ
*Do part 1 Δ
                x =            1
                x =          101
                x =            2
                x =          102
                x =            3
                x =          103
                x =            4
                x =          104
                x =            5
                x =          105
```

Note that, when the FOR clause is used, the end-range value is hit exactly. For example, given the DO command

Do part 1 for x = 1(3)14.5

iterations will be performed for x = 1, 4, 7, 10, 13, and 14.5.

IF Clause

The IF clause (q.v.) when appended to a DO command is also interpreted only once (when the DO command is encountered) and has no effect once execution of the DO has begun. Thus, even though the DO-initiated routine might perform some action which would make the IF condition no longer satisfied, once execution has begun it continues to its normal termination.

```
*x = 20 ↵
*1.1 Do part 2, 3 times if x>0 ↵
*2.1 Set x = x-50 ↵
*2.2 Type x ↵
*Do part 1 ↵                                    At the start, x = 20.
              x =               -30       x is now<0, but iteration continues.
              x =               -80
              x =              -130
```

Parenthetical DO (DO......)

The parenthetical DO command is used to initiate execution of a step or part, while another process is waiting to continue after a STOP or other type of interrupt, without cancelling that other process.

NOTE

A normal DO command automatically cancels any currently stopped process.

The parenthetical DO command includes all the options of the normal DO command. Its general format is:

(Do ....)

a) (Do part 3)

b) (Do step 1.4 for x = 5(5)25)

The parenthetical DO command is commonly used to execute a step or part to test its validity; thus, this command is primarily a debugging aid.

Any stopped process which was originally initiated by a parenthetical DO can be cancelled by a parenthetical CANCEL command.

Examples of parenthetical DOs can be found under "GO" in this chapter.

EXAMPLES

```
a)  *1.1 Type "A" ↵
    *1.2 Type "D" ↵
    *1.3 Type "F" ↵
    *1.4 Type "J" ↵
```

```
*1.25 Type "E" ↵
*2.1 Type "B" ↵
*2.2 Type "C" ↵
*3.1 Type "G"↵
*3.2 Type "H"↵
*3.3 Type "I"↵
*Do part 1 ↵
A
D
E
F
J
*1.15 Do part 2
*Do part 1 ↵
A
B
C
D
E
F
J
*1.35 To part 3 ↵
*Do part 1
A
B
C
D
E
F
G
H
I


*
```

Note that no return is made to step 1.4. (the TO command does not return control).

b)
```
*Let I = sqrt (a↑2+b↑2)↵
*1.1 Do part 2 for b = 1 (3)9↵
*2.1 Type a, b, I in form 1↵
*Form 1:↵
*    a = ←←←  b = ←←←  c = ←←←←←.←←←←↵
*Do part 1 for a = 1 (2)12 ↵
    a =     1   b =     1     c =     1.4142
    a =     1   b =     4     c =     4.1231
    a =     1   b =     7     c =     7.0711
    a =     1   b =     9     c =     9.0554
    a =     3   b =     1     c =     3.1623
    a =     3   b =     4     c =     5.0000
    a =     3   b =     7     c =     7.6158
    a =     3   b =     9     c =     9.4868
    a =     5   b =     1     c =     5.0990
    a =     5   b =     4     c =     6.4031
    a =     5   b =     7     c =     8.6023
    a =     5   b =     9     c =    10.2956
    a =     7   b =     1     c =     7.0711
    a =     7   b =     4     c =     8.0623
```

```
     a=   7   b=   7   c=    9.8995
     a=   7   b=   9   c=   11.4018
     a=   9   b=   1   c=    9.0554
     a=   9   b=   4   c=    9.8489
     a=   9   b=   7   c=   11.4018
     a=   9   b=   9   c=   12.7279
     a=  11   b=   1   c=   11.0454
     a=  11   b=   4   c=   11.7047
     a=  11   b=   7   c=   13.0384
     a=  11   b=   9   c=   14.2127
     a=  12   b=   1   c=   12.0416
     a=  12   b=   4   c=   12.6491
     a=  12   b=   7   c=   13.8924
     a=  12   b=   9   c=   15.0000
*
```

c)
```
*Delete all. ↵
*Let A = (B↑2)/4*sqrt(3) ↵
*1.1 Type A, 2*A, ← ↵
*1.2 Stop if A>100 ↵
*2.1 Type B ↵
*Do part 1 for B=10(25)100 ↵
           A =                43.3012703
         2*A =                86.6025406
           A =               530.440561
         2*A =              1060.88112

Stopped by step 1.2.
*Type A ↵
           A =               530.440561
*1.0 Stop if A>100 ↵
*Do part 1 for B=10(25)100 ↵
           A =                43.3012703
         2*A =                86.6025406

Stopped by step 1.
*
```

DIAGNOSTIC MESSAGES

I CAN'T FIND THE REQUIRED STEP.	An incorrect step number has been specified; no such step number exists.
I CAN'T FIND THE REQUIRED PART.	An incorrect part number has been specified; no such part number exists.

DONE

DESCRIPTION

The DONE command skips execution of the remaining steps of a part during the current iteration. This command can be given indirectly only. It is usually given conditionally.

Done (unconditional)

Normally used only as a temporary step (during the testing of a routine) when performing a partial execution.

Done if ...... (conditional)

Used to skip execution of the remaining steps of a part when certain conditions (specified in the IF clause) are met.

EXAMPLE

```
*Let A = B↑2+2*B+10 ↵
*Let C = A↑2+2*A*B+B↑2 ↵
*1.1 Type A,B,C  in form 1 ↵
*1.2 Type A*B ↵
*1.3 Type A*C,← ↵
*Form 1:↵
*←←←←←←•←←←←←  ←←←←←←•←←←←←  ←←←←←←•←←←←←
                                            ↵
*Do part 1 for B = 1(1)4 ↵
     13.0000          1.0000      196.0000
       A*B =          13
       A*C =        2548

     18.0000          2.0000      400.0000
       A*B =          36
       A*C =        7200

     25.0000          3.0000      784.0000
       A*B =          75
       A*C =       19600


     34.0000          4.0000     1444.0000
       A*B =         136
       A*C =       49096
*1.15 Done ↵
```

Insert temporary premature termination step following 1.1.

```
*Do part 1 for B=1(1)4 ↵
     13.0000          1.0000      196.0000
     18.0000          2.0000      400.0000
     25.0000          3.0000      784.0000
     34.0000          4.0000     1444.0000
*1.15 Done if B>2 ↵
*Do part 1 for B=1(1)4 ↵
     13.0000          1.0000      196.0000
       A*B =          13
       A*C =        2548

     18.0000          2.0000      400.0000
       A*B =          36
       A*C =        7200
```

Change unconditional DONE to conditional.

```
                25.0000        3.0000      784.0000
                34.0000        4.0000     1444.0000
   *
```

DIAGNOSTIC MESSAGES

DON'T GIVE THIS COMMAND DIRECTLY	The DONE command must only be given <u>indirectly</u> (preceded by a step number).

5.7 FILE

DESCRIPTION

FILE stores an item in the external storage file currently in use. Immediate storage is not affected in any way.

```
        ⎧ a             ⎫
        ⎪ a(b,...)      ⎪
        ⎪ form m        ⎪
        ⎪ step m        ⎪
        ⎪ part m        ⎪
        ⎪ formula f     ⎪
        ⎪ all steps     ⎪
File   <  all parts      >  as item n (code)
        ⎪ all formulas  ⎪
        ⎪ all forms     ⎪
        ⎪ all values    ⎪
        ⎩ all           ⎭
```

Store the specified information as item n (where n can be in the range 1 through 25) in the currently open external storage file. Code is optional for documentation only and has no meaning to AID; however, code, if used, must not exceed five characters in length.

EXAMPLE

```
*File all parts as item 5↵
Done.
*
```

All parts existing in immediate storage are stored on the currently open external storage file as item 5. Successful execution of the command is evidenced by the AID response, DONE.. Item contents can be retrieved by the RECALL command.

DIAGNOSTIC MESSAGES

ITEM NUMBER MUST BE POSITIVE INTEGER <=25.	An invalid item number was given.
PLEASE DISCARD THE ITEM OR USE A NEW ITEM NUMBER.	The specified item is already occupied; no change in either immediate or external storage occurs.
PLEASE LIMIT ID'S TO 5 LETTERS AND/OR DIGITS	Code exceeds five characters in length.
YOU HAVEN'T TOLD ME WHAT FILE TO USE.	A FILE command was attempted before an external storage file was opened via a USE command.

NOTE

Only 22 items are allowed, if DECtape is used for external storage.

FORM

5.8

DESCRIPTION

FORM is used to edit typeouts of results for purposes of readibility, e.g., to (1) specify that results be typed in a specific notation, (either scientific or fixed point), (2) specify that multiple results are to be printed on a single line, usually to conserve space, (3) intersperse text with results, and (5) produce report-type headings.

The elements which can be typed in a form are:

a) Numeric values, including variables, $ (line counter), TIME, TIMER, and SIZE.

1. Type -23.466 in form 1
2. Type a, b, c in form 2
3. Type $ in form 3
4. Type TIMER in form 4

b) Propositional values (TRUE and FALSE). Both of these values must be provided with an integral form field containing at least five character positions.

Type F in form 5 (where F is a proposition).

c) ← (indicating a blank field).

Type a, b, ← , f in form 6.

Forms are entered as two lines:

*Form n:↵	n identifies the specific form and must be an integer.
* user types actual format here.	

Once a form is defined, it can be used by specifying it in a TYPE command.

Type .... <u>in form n</u>

Specific Notations

Fixed-point notation is specified by a series of left arrows, one for each digit position and one for a sign (if any). If less integer places appear in the form than in the result, the error message I CAN'T EXPRESS THE VALUE IN YOUR FORM is typed; if less decimal places appear in the form than in the result, rounding occurs. A period is used to indicate the decimal point position.

```
←←←←.←←←
-345.667
←←←←.←←←←←←
-345.666667
```

At least seven periods must appear in a scientific notation form.

```
                    .......
                   -3.3-01
```

Reducing the number of periods in a scientific notation form reduces the number of fraction digits appearing in the result; these digits are dropped after rounding.

Multiple Results on a Single Line

More than one result can be typed on a single line through the use of the FORM command. Such a technique might be used to conserve space, increase output speed, and/or cause results to appear under previously typed column headings.

```
*Form 1:
*   ←←•←  ←←←←←←←←•←  ←←←←←←←←•←←  ←←←←←←←•←←←
* 1.1 Type a, a↑2, a↑3, a↑4  in form 1
*Do step 1.1 for a = 10(.5)15
    10.0       100.0      1000.00      10000.000
    10.5       110.3      1157.63      12155.063
    11.0       121.0      1331.00      14641.000
    11.5       132.3      1520.88      17490.063
    12.0       144.0      1728.00      20736.000
    12.5       156.3      1953.13      24414.063
    13.0       169.0      2197.00      28561.000
    13.5       182.3      2460.38      33215.063
    14.0       196.0      2744.00      38416.000
    14.5       210.3      3048.63      44205.063
    15.0       225.0      3375.00      50625.000
*
```

Interspersing Text with Results

A form can be used to intersperse explanatory text with typed results.

```
*Form 2:
* If a = ←←←•←     then a↑2 = ←←←←←←•←←←←
*2.1  Type a, a↑2 in form 2
*Do part 2 for a = 10(.5)12
    If a = 10.0     then a↑2 =     100.0000
    If a = 10.5     then a↑2 =     110.2500
    If a = 11.0     then a↑2 =     121.0000
    If a = 11.5     then a↑2 =     132.2500
    If a = 12.0     then a↑2 =     144.0000
*
```

Report-Type Headings

A form containing only text can be used to generate columnar headings.

```
*Form 3:ȡ
*     a           a↑2          a↑3           a↑4
*1.1 Type form 3
*1.2 Do step 1.3 for a = 10(.5)12ȡ
*1.3 Type a, a↑2, a↑3, a↑4 in form 1ȡ
*Form 1:
*    ←←.←   ←←←←←←.←←←←   ←←←←←←.←←←←←   ←←←←←←.←←←←←ȡ
*Do part 1ȡ
      a           a↑2          a↑3           a↑4
     10.0      100.0000     1000.0000    10000.0000
     10.5      110.2500     1157.6250    12155.0625
     11.0      121.0000     1331.0000    14641.0000
     11.5      132.2500     1520.8750    17490.0625
     12.0      144.0000     1728.0000    20736.0000
     12.0      144.0000     1728.0000    20736.0000
*
```

DIAGNOSTIC MESSAGES

FORM NUMBER MUST BE INTEGER AND 1<=FORM<10↑9.	Form numbers must be integers in the range 1 through 10^9-1.
I CAN'T EXPRESS THE VALUE IN YOUR FORM.	A value cannot be expressed in the format given (the value is too large).
I CAN'T FIND THE REQUIRED FORM.	The specified form does not exist; the form number is incorrect.
I HAVE TOO MANY VALUES FOR THE FORM.	The TYPE command specifies more elements to be typed than there are fields in the form.

DESCRIPTION

GO continues execution of a currently stopped (interrupted) process.

GO is the antithesis of the CANCEL command.

The GO command is normally used to continue execution after control has been returned to the user via a STOP command.

Go

The GO command must be given directly only.

EXAMPLE

```
*Let v=p*(r↑2)h↵
*p=3.142↵
*1.1 Do part 2 for h=.5(.5)3 ↵
*1.2 Stop↵
*2.1 Type r, h, v in form 1↵
*3.1 Type r↑2↵
*3.2 Delete step 1.2↵
*Form 1:
*  ←←←  ←←←.←  ←←←←←.←←←←  ↵
*Do part 1 for r=1(1)3 ↵
Error at step 2.1 (in formula v):  eh?

*Type formula v↵
        v:   p*(r↑2)h                                Multiplication symbol was omitted.
*Let v=p*(r↑2)*h ↵                                   Correct formula.
*Go↵                                                 Execute GO to continue.
        1       .5          1.5710
        1      1.0          3.1420
        1      1.5          4.7130
        1      2.0          6.2840
        1      2.5          7.8550
        1      3.0          9.4260

Stopped by step 1.2                                  STOP command at step 1.2 is
*(Do part 3)↵                                        encountered.
        r↑2 =               1
Done.  I'm ready to go from step 1.2, although I can't find it.
*Go↵
        2       .5          6.2840                   Execute part 3 via a parenthetical
        2      1.0         12.5680                   DO; then GO to continue.
        2      1.5         18.8520
        2      2.0         25.1360
        2      2.5         31.4200
        2      3.0         37.7040
        3       .5         14.1390
        3      1.0         28.2780
        3      1.5         42.4170
        3      2.0         56.5560
        3      2.5         70.6950
        3      3.0         84.8340
*Go
I have nothing to do.
*
```

DON'T GIVE THIS COMMAND INDIRECTLY.	The GO command can be given directly only (with no step number preceding it).
I HAVE NOTHING TO DO.	When the GO command was given, no process was in a stopped or interrupted status. Control returns to the user and AID waits for a new command.

IF CLAUSE

DESCRIPTION

The IF clause can be appended to any command (except the short SET command) to make that command conditional; (the command is executed only if the proposition following the word IF is satisfied).

Verb (arguments) IF proposition

EXAMPLES

a)	*1.1 Set b = 50 if a>100 ↵	Set b equal to 50 if, and only if, a is greater than 100; otherwise leave the value of b undisturbed.
b)	*3.3 To part 5 if fp(d)=0 ↵	Transfer control to part 5 if, and only if, d is an integer; otherwise, continue in sequence.
c)	*2.9 Do part 3 if tv(f)=1 ↵	Execute part 3 if the truth value (tv) of proposition f is equal to 1; otherwise, continue in sequence.

LET

5.11

DESCRIPTION

LET defines arithmetic formulas, Boolean expressions (propositions), and user functions. The formula, expression, or function with which an identifier is associated is re-evaluated each time that identifier appears during execution of a routine.

Arithmetic Formulas

The LET command can be used to tell AID how to calculate the value of an identifier (versus associating the identifier with a fixed value, as with the SET command). LET causes the identifier on the left of the equals sign to be set to the formula on the right of the equals sign.

```
a)  *Let v = p*(r↑2)*L ↵
    *Set p = 3.1416 ↵
b)  *Let L = w*h ↵
```

Boolean Expressions (Propositions)

LET can also be used to equate an identifier to the value (true or false) of a proposition (a Boolean expression) composed of arithmetic and logical statements using common relational operators (e.g., =,<, >), the logical negation (not), and logical operators (and, or).

```
a)  *Let a = true ↵
b)  *Let c = a and b or c or d ↵
c)  *Let y = x and y or (sqrt(100) <sqrt(z)) ↵
```

User Functions

LET has a third use, that of defining a user function.

```
a)  *Let a(b,c) = b*c ↵
b)  *Let v(R,H) = p(R↑2)*H ↵
```

User functions, once defined, are used in exactly the same manner as AID functions.

```
a)  *Type a(12,30) ↵
b)  *Let m = v(f,g)*d ↵
```

EXAMPLES

A more complete discussion of these three uses of LET, including examples, can be found in Chapter 3.

SPECIAL LET COMMAND - LET S BE SPARSE

Let s be sparse   where s is a subscripted letter.

Declares undefined array elements to have zero value; such elements require no space in immediate storage.

Example

```
*x(1,2)=55đ
*x(1,5)=43đ
*x(1,10)=60đ
*x(2,4)=77đ
*Type x(1,10)đ
      x(1,10) =        60
*Type x(1,3)đ
x(1,3) = ???
*Let x be sparseđ                  Set all undefined x array items to
*Type x(1,3)đ                      zero.
      x(1,3) =          0
*Type x(2,1)đ
      x(2,1) =          0
*Type xđ
      x(1,2) =         55
      x(1,5) =         43          Only those elements which have
    x(1,10) =          60          been explicitly defined are typed,
      x(2,4) =         77          followed by a message reminding
x is sparse.                       the user that he has defined x as
*                                  sparse.
```

Although an array may be defined as sparse, at least one element in the array must be given an explicit value (so that AID will know the dimensions of the array) before any attempt is made to refer to an item within the array.

```
*Let d be sparseđ
*Type d(1,3,5)đ
d(1,3,5) = ???
*Type dđ
d = ???
*d(2,4,6) = 20đ
*Type d(1,3,5)đ
     d(1,3,5) =         0
Type dđ
     d(2,4,6) =        20
d is sparse.
*
```

LINE

5.12

DESCRIPTION

The LINE command advances the Teletype page one line.

Line.

The LINE command is often given conditionally:

2.4 Line if fp($/5) = 0

EXAMPLE

```
*1.1 Type "VOLUME CALCULATION"  ↵
*1.2 Line ↵
*1.3 Type a, b, c, a*b*c  ↵
*a=3 ↵
*b=5 ↵
*c=12 ↵
*Do part 1 ↵
VOLUME CALCULATION
                                        AID advances paper form one line.
         a =          3
         b =          5
         c =         12
     a*b*c =        180
```

NOTE

The steps above perform essentially the same process as the command:

```
*1.1 Type "VOLUME CALCULATION"  ↵
*1.2 Type ←,a,b,c,a*b*c ↵
```

DESCRIPTION

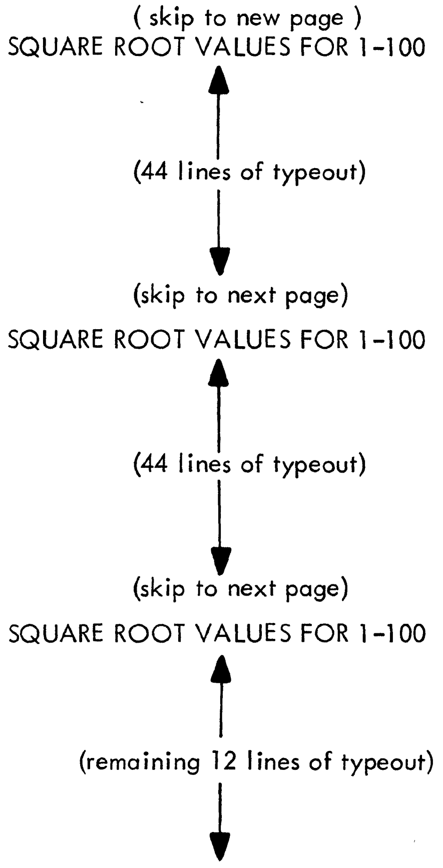
PAGE advances the Teletype paper form to the top of the next page.

Page.

The PAGE command can be used in conjunction with the $ symbol, which represents the current line count, (the number of lines printed thus far on the current page). AID allows for a maximum of 54 lines per page.

EXAMPLE

```
*1.0 Page
*1.1 Type"SQUARE ROOT VALUES FOR 1 - 100"
*1.2 Do part 2 for a = 1(1)100
*2.1 Type a, sqrt(a)
*2.2 Do part 3 if $>45
*3.1 Page
*3.2 Do step 1.1
*Do part 1
```



```
         ( skip to new page )
SQUARE ROOT VALUES FOR 1-100

        (44 lines of typeout)

         (skip to next page)
SQUARE ROOT VALUES FOR 1-100

        (44 lines of typeout)

         (skip to next page)
SQUARE ROOT VALUES FOR 1-100

   (remaining 12 lines of typeout)
```

QUIT

5.14

DESCRIPTION

QUIT skips execution of the remaining steps of a part and satisfies the DO command for that part by cancelling any further iterations.

The QUIT command is usually given conditionally.

Quit. (unconditional)

Normally used only as a temporary step (inserted for the purpose of testing a portion of a routine) when performing a partial execution.

Quit if...... (conditional)

Used to skip execution of the remaining steps of a part (and any further iterations of the part by the current DO command) when certain conditions are present.

EXAMPLE

```
*Let A = B↑2+2*B+10↲
*Let C = A↑2+2+A+B+B↑2↲
*1.1 Type A,B,C↲
*1.2 Type A*B↲
*1.3 Type A*C↲
*Do part 1 for B = 5(5)15↲
            A  =            45
            B  =             5
            C  =          2102
          A*B  =           225
          A*C  =         94590
            A  =           130
            B  =            10
            C  =         17142
          A*B  =          1300
          A*C  =             2.22846*10↑6
            A  =           265
            B  =            15
            C  =         70732
          A*B  =          3975
          A*C  =             1.874398*10↑7
*1.15 Quit if A >100↲
*Do part 1 for B = 5(5)15↲
            A  =            45
            B  =             5
            C  =          2102
          A*B  =           225
          A*C  =         94590
            A  =           130
            B  =            10
            C  =         17142
*
```

RECALL

DESCRIPTION

RECALL reads an item, previously stored by a FILE command, from the currently open external storage file into immediate storage. The contents of the item then exist both on the external file and in immediate storage. All steps, identifiers, forms, etc., which were in immediate storage before the RECALL command was given remain unchanged, with the exception of those which are redefined by the recalled item.

*Recall item m (code)	Read in item #m (where m can be in the range 1 through 25) from the currently open external storage file. Code is optional for documentation purposes only and is ignored by AID; however, code, if used, cannot exceed five characters in length.

EXAMPLE

```
*Recall item 23
Done.
*
```

The contents of item 23 of the currently open file are read into immediate storage. Successful execution of the RECALL command is evidenced by the AID response, DONE..

DIAGNOSTIC MESSAGES

I CAN'T FIND THE REQUIRED ITEM.	The specified item cannot be found in the currently open file. Either the wrong file is open, the item number is incorrect, or the item was never filed.
ITEM NUMBER MUST BE POSITIVE INTEGER <=25.	An invalid item number was given.
PLEASE LIMIT ID'S TO 5 LETTERS AND/OR DIGITS.	Code exceeds five characters in length.
YOU HAVEN'T TOLD ME WHAT FILE TO USE.	A RECALL command was attempted before an external storage file was opened via a USE command.

RESET TIMER

5.16

DESCRIPTION

Resets TIMER[1] to zero.

Reset timer

TIMER is a counter used by AID to keep track of the amount of central processor time spent by the user in running AID. This cumulative running time can be obtained at any point by typing the request Type timer.. Each time the user wishes to reset the timer and to begin timing a new operation, he types Reset timer..

[1]The least significant digit of TIMER is frequently used to supply pseudo-random decimal numbers.

5.17 SET

DESCRIPTION

SET defines an identifier as equivalent to a fixed value. This value is calculated once and the result is then used whenever the identifier appears in a calculation.

*Set x = expression or value	If an expression, the expression must be immediately reducible to a numeric value.

NOTE

When the SET command is typed as a direct command, the verb (SET) may be omitted. This form is called a short SET command.

a) *Set a = 20 ↵

b) *a = 20↵

c) *Set a = sqrt(20)+43.5↑2 ↵

d) *Set d = true

e) *f = false

EXAMPLES

A more complete discussion of the use of SET commands, including additional examples, can be found in Chapter 3.

STOP

5.18

DESCRIPTION

The STOP command temporarily halts the current process at the point where the STOP command appears and returns control to the user. The stopped process can be resumed by typing GO. If the user does not desire to continue the process, he types CANCEL.

The STOP command can be given indirectly only.

Stop (unconditional)

Normally used only as a temporary step (during the testing of a routine) when performing a partial execution.

Stop if .... (conditional)

Used to temporarily halt execution and return control to the user when certain conditions (specified in the IF clause) are met.

EXAMPLE

```
*Let B=16-C
*Let A=3.17568/B
*1.1 Stop if B=0
*1.2 Type A,B,C in form 1
*Form 1:
*    ..............   ..............    ..............
*Do part 1 for C=-4(4)24
     1.58784000-01    2.00000000 01      -4.00000000 00
     1.98480000-01    1.60000000 01       0
     2.64640000-01    1.20000000 01       4.00000000 00
     3.96960000-01    8.00000000 00       8.00000000 00
     7.93920000-01    4.00000000 00       1.20000000 01
Stopped by step 1.1
*Type C
             C =      16
*Type B
             B =       0
*
```

STOP command prevents attempt to divide by 0 in formula A.

TO

DESCRIPTION

TO discontinues the sequential execution of the part currently being executed and transfers control to another step or part. When the new part is finished, the direct command which initiated the execution is satisfied.

The TO command can be given indirectly only.

$$\underline{\text{To}} \begin{Bmatrix} \text{part m} \\ \text{step m.n} \end{Bmatrix}$$

EXAMPLE

```
*1.1 Demand G                                   Demand gross pay for week.
*1.2 Demand T                                   Demand total FICA year-to-date.
*1.3 To part 2 if T>=560                        $560 = maximum deduction/year.
*1.4 Let d = G*.046                             d = current deduction.
*1.5 To part 3 if (T+d)>560
*1.6 Type "DEDUCTIONS"
*1.7 Type d,d+T,←,←
*2.1 Type "NO DEDUCTION REQUIRED"
*2.2 Let d = 0
*2.3 To step 1.6
*3.1 Let d = 560-T
*3.2 To step 1.6
*Do part 1, 4 times.
             G = *125.00
             T = *340.00
DEDUCTIONS
             d =              5.75
           d+T =            345.75


             G = *350.00
             T = *545.00
DEDUCTIONS
             d =             15
           d+T =            560


             G = *103.45
             T = *559.04
DEDUCTIONS
             d =               .96
           d+T =            560


             G = *300.00
             T = *565.00
NO DEDUCTION REQUIRED
DEDUCTIONS
             d =              0
           d+T =            565


*
```

DON'T GIVE THIS COMMAND DIRECTLY.	The TO command must only be given indirectly (preceded by a step number).

5.20

TYPE

DESCRIPTION

Types out the specified information on the user's console.

Type

n	where n is numeric value or expression
s	where s is a subscripted variable
s(a,b,..)	
p	where p is a proposition
"any text"	
←	← represents a null item (a blank field when typing in form x, or a blank line)
form m	
step m.n	
part m	
formula f	
f(n,...)	function of (n,...)
f(p)	function (tv) of a proposition
all steps	
all parts	
all formulas	
all forms	
all values	
all	
time	current time of day in minutes since midnight
timer	processor time used (see RESET TIMER)
size	amount of immediate storage being used
item-list	item list of currently open external storage file
users	always returns an answer of 0..

Combined TYPE Commands

Several individual TYPE commands (except for TYPE "any text" or TYPE ITEM-LIST) can be combined into one command.

*Type all parts, 1243, formula D, form 5

Each entry, however, is still typed on a separate line.

IN FORM Option

Output editing can be performed by appending the IN FORM ... option to a TYPE command. See "FORM". Note that only certain types of entries can be typed in forms.

EXAMPLE

```
                                                   Variables
*b = 20 △
*c = 30 △
*d = 111.333 △
*Let a(b,c) = (b↑2)*(c↑2) △                        User functions
*Let f(b) = b/2 △
*e(1) = 16 △                                       Subscripted variables
*e(2) = 25 △
*e(3) = 35 △
*Let g = h or i and j △                            Propositions
*h = true △
*i = false △
*j = false △
*Let k = b*c*d △                                   Formulas
*Let m = k/2*sqrt(k) △
*Form 1: △                                         Forms
*   .........  POUNDS IS  .........  OUNCES △
*Form 2: △
*   POUNDS     OUNCES
*1.1 Type b,c △                                    Parts and steps
*1.2 Type d,e △
*1.3 To part 2 △
*2.1 Type form 2 △
*2.2 Type d/e(1), d in form 1 △
*3.1 Type k,m △
*3.2 Type a(e(1),e(2)) △
*Do part 1 △
              b =             20
              c =             30
              d =             111.333
           e(1) =             16
           e(2) =             25
           e(3) =             35
    POUNDS      OUNCES
    6.958 00 POUNDS IS         1.113 02  OUNCES
*Type e △                                          A command to type the values of a
           e(1) =             16                   subscripted letter results in typeouts
           e(2) =             25                   of all values.
           e(3) =             35
*Type a(3,6) △
         a(3,6) =            324
*Type a △
         a(b,c):       (b↑2)*(c↑2)
*Type g △
              g =            true
*Type tv(g) △
          tv(g) =              1
*Type form 1 △
    ......... POUNDS IS .........  OUNCES
*Type step 1.3 △
1.3 To part 2.
*Type formula k △
              k:  b*c*d
*Type all steps △
```

```
1.1 Type b,c.
1.2 Type d,e.
1.3 To part 2.
2.1 Type form 2.
2.2 Type d/e(1), d in form 1.
3.1 Type k,m.
3.2 Type a(e(1),e(2)).
*Type all formulas Δ
        a(b,c):  (b↑2)*(c↑2)
             g:  h or i and j
             k:  b*c*d
             m:  k/2*sqrt(k)
*Type all forms Δ
Form 1:

    .........  POUNDS IS  .........  OUNCES
Form 2:
    POUNDS     OUNCES
*Type all values Δ
           b =              20
           c =              30
           d =             111.333
           h =             true
           i =             false
           j =             false
        e(1) =              16
        e(2) =              25
        e(3) =              35
*Type all Δ
1.1 Type b,c.
1.2 Type d,e.
1.3 To part 2.
2.1 Type form 2.
2.2 Type d/e(1), d in form 1.
3.1 Type k,m.
3.2 Type a(e(1),e(2)).
Form 1:
    .........  POUNDS IS  .........  OUNCES
Form 2:
    POUNDS     OUNCES
        a(b,c):  (b↑2)*(c↑2)
             g:  h or i and j
             k:  b*c*d
             m:  k/2*sqrt(k)
           b =              20
           c =              30
           d =             111.333
           h =             true
           i =             false
           j =             false
        e(1) =              16
        e(2) =              25
        e(3) =              35
*Type time
         time:  1453                          24-hour time
```

```
*Type timer

        timer =              mm ss.ss
```

The total central processor time utilized by the user thus far.

mm = minutes
ss.ss = seconds to the nearest hundredth

```
*Type size
        size:                64
```

The number of "cells" of immediate storage currently occupied by the user's work area. In a 13K environment, approximately 1900 such "cells" are available for this purpose.

```
*Use file 110 (DSK)
Roger.
*File all formulas as item 1 (FMULA)
Done.
*File all forms as item 2 (FORM)
Done.
*File all values as item 3 (VALUE)
Done.
*File all steps as item 10 (STEP)
Done.
*Type item-list
          ITEM-LIST
ITEM                     DATE
  1                     mm/dd/yy
  2                     mm/dd/yy
  3                     mm/dd/yy
 10                     mm/dd/yy
*
```

DATE is the creation date.

USE

DESCRIPTION

USE makes an external storage file[1] available for use. The external file thus addressed remains open for use until another USE command is given or until the AID program is terminated.

USE file filename (device)

where	filename	is a positive integer, greater than 0 and less than or equal to 2750, which identifies the particular file on the device.
	device	is the device name (logical or physical) of the device containing the file. Device can be one of the following.

DSK	disk
DTAn	DECtape, where n is the drive number and can be in the range 0 through 7.

or any logical device name assigned to either of the above device types. If device is omitted, DSK is assumed. Magnetic tape may also be used, provided the user positions the the tape directly before the desired file before issuing a RECALL or FILE command; the TYPE ITEM-LIST and DISCARD commands have no meaning for magnetic tape.

Once a file has been opened by a USE command, all DISCARD, FILE, RECALL, and TYPE ITEM-LIST commands are assumed to refer to that file.

EXAMPLE

```
*Use file 103 (DSK) ↵                    (or Use file 103)
Roger.↵
*
```

Makes file 103, on disk, available for use. Successful execution of the command is evidenced by the AID response ROGER.

DIAGNOSTIC MESSAGES

FILE NUMBER MUST BE POSITIVE INTEGER <=2750.	Filename is less than 0 or greater than 2750.

[1]External storage files created by AID are in a special AID (8-bit) character format, not ASCII.

APPENDIX A
TABLES

This appendix contains five tables:

A-l	A Glossary of AID Terms
A-2A	AID Command Summary
A-2B	File Command Subset
A-3	AID Character Set
A-4	AID Diagnostic Messages

Table A-1.
A Glossary of AID Terms

← Indicates a blank field, when typed in conjunction with a FORM; otherwise, a blank line.

$ A special symbol which refers to the line counter kept by AID. This symbol can be used in TYPE commands, or it can be tested within a conditional command to cause a line feed (LINE) or advance to next head of form (PAGE) at appropriate points.

Conditional expression	A series of clauses separated by semicolons, with each clause made up of a proposition followed by a colon and then by an expression. The entire conditional expression must be enclosed in parentheses or brackets.
External storage	See "File".
Expression	An arithmetic formula, Boolean proposition, or function.
File	A peripheral storage medium (usually disk or DECtape) used for the preservation of user subroutines, values, etc. Files are manipulated by the DISCARD, FILE, and RECALL commands.
Form	A user-specified format for editing output via the TYPE command.
Formula	An arithmetic expression defined by the user via the LET command.
Function	An arithmetic or Boolean function provided by AID (see Table 4-2) or defined by the user via the LET command.
Identifier	A single alphabetic character associated with a variable (via the LET or SET commands) and then used to access the current value of the variable.
Immediate storage	Core storage work area. In a 13K environment, approximately 4K of core is available to the user for his steps, values, tables, etc.
Item-list	The file directory associated with an external storage file. The user can obtain a listing of the directory of the currently open file by typing the command TYPE ITEM-LIST.
Part	A series of indirect steps, the step numbers of which have the same integral value.
SIZE	A noun which can be specified in a TYPE command to obtain a typeout of the amount of core "cells" used. (1 cell = approx. 2 core words)

Table A-1 (Cont)
A Glossary of AID Terms

Step	Any one-line command typed by the user. A step can be direct (executed immediately), in which case no step number precedes it, or indirect (to be executed later), in which case it is preceded by a step number.
TIME	A noun which can be specified in a TYPE command to obtain a typeout of the time of day (in 24-hour format).
TIMER	A noun which can be specified in a TYPE command to obtain a typeout of the amount of central processor time spent thus far by the user during the current AID run.
Variable	An element defined by either the LET or SET commands which is associated with some value. It is referred to by an associated tag called an identifier.

Table A-2A
AID Command Summary

Command Format	Type	Description
CANCEL	D, O	Cancels a currently stopped process when the user does not desire to resume execution.
(CANCEL)	D, O	Cancels a currently stopped process which was initiated by a parenthetical DO.
DELETE $\left\{\begin{array}{l}\text{L}\\ \text{S}\\ \text{S(m,n)}\\ \text{form m}\\ \text{step m.n}\\ \text{part m}\\ \text{formula f}\\ \text{all steps}\\ \text{all parts}\\ \text{all formulas}\\ \text{all forms}\\ \text{all values}\\ \text{all}\end{array}\right\}$	O	Erases the specified item from immediate storage and frees the space occupied by it for some other use. Several DELETE commands can be combined into one.
DEMAND $\left\{\begin{array}{l}\text{L}\\ \text{S(m,n)}\\ \text{L as "any text"}\\ \text{S(m,n) as "any}\\ \text{ text"}\end{array}\right\}$	I, O	Causes AID to type out a message requesting the user to supply a value for the specified item. Only one variable can be specified in each DEMAND command.
DISCARD ITEM m(code).	F	Deletes item #m from the external storage file currently in use. (Code) is optional.
DO $\left\{\begin{array}{l}\text{step m.n}\\ \text{step m.n, p times}\\ \text{step m.n for L=range}\\ \text{part m}\\ \text{part m, p times}\\ \text{part m for L=range}\end{array}\right\}$	O	Executes an indirect step or part. If the DO command is a direct step, control returns to the user at the completion of the DO; if an indirect step, control returns to the step following the DO.
(DO ....same as above..)		Initiates a new execution without cancelling the currently stopped process.
DONE	I, O	Skips execution of the remaining steps of a part during the current iteration.

Table A-2A (Cont)
AID Command Summary

Command Format	Type	Description
FILE $\left\{\begin{array}{l}\text{L}\\ \text{S}\\ \text{S(m,n)}\\ \text{form m}\\ \text{step m.n}\\ \text{part m}\\ \text{formula f}\\ \text{all steps}\\ \text{all parts}\\ \text{all formulas}\\ \text{all forms}\\ \text{all values}\\ \text{all}\end{array}\right\}$ AS ITEM n (code)	F	Stores the specified item in the external storage file currently open. Immediate storage is not affected in any way . (code) is optional.
FORM m: ....++++ Text	O	Defines a format to be used in editing typeouts for purposes of readability. ++++++.+++++ fixed point notation (up to nine digit positions plus the decimal point) .... scientific notation (minimum of seven positions maximum of fourteen) text any text to be included in the line; not enclosed in quotation marks unless they are part of the text.
GO	D,O	Continues execution of a currently stopped process; opposite of the CANCEL command.
IF Clause Verb....IF proposition.	M	Can be appended to any command (except the abbreviated-SET command) to make the command conditional; the command is executed only if the proposition is true.
LET $\left\{\begin{array}{l}\text{L = m}\\ \text{L = formula}\\ \text{F(L) = m}\\ \text{F(L) = proposition}\end{array}\right\}$	O	Defines arithmetic formulas, Boolean expressions (propositions), and user functions and associates them with identifiers. The formula, expression, or function with which an identifier is associated is re-evaluated each time the identifier appears during an execution.
LET S be sparse	S	Sets undefined array elements to zero.

Table A-2A (Cont)
AID Command Summary

Command Format	Type	Description
LINE	O	Advances the Teletype paper form one line.
PAGE	O	Advances the Teletype paper form to the top of the next page.
QUIT	O	Skips execution of the remaining steps of a part and satisfies the DO command for that part by cancelling any further iterations. Usually given conditionally.
RECALL ITEM m (code)	F	Reads an item, previously stored by a FILE command, from the currently open external storage file into immediate storage. (Code) is optional and is for documentation only.
RESET TIMER	S	Resets TIMER to zero.
SET $\left\{\begin{array}{l}\text{L=m}\\ \text{L=proposition}\\ \text{s(m,n)=m}\\ \text{S(m,n)=proposition}\end{array}\right\}$	O	Defines an identifier as equivalent to a fixed value, which is calculated once and then used whenever the identifier appears. A short form of the SET command, where the word SET is omitted, can be used if the command is direct.
STOP	I, O	Temporarily halts the current process at the point where the STOP command appears and returns control to the user. The stopped process can be resumed by typing GO.
TO $\left\{\begin{array}{l}\text{part m}\\ \text{step m.n}\end{array}\right\}$	I, O	Discontinues the sequential execution of the part currently being executed and transfers control to another step or part; when the new part is finished, the direct command which initiated the execution is satisfied.

Table A-2A (Cont)
AID Command Summary

Command Format	Type	Description
TYPE $\left\{\begin{array}{l}\text{m}\\ \text{S}\\ \text{S(m,n)}\\ \text{proposition}\\ \text{"any text"}\\ \leftarrow\\ \text{form m}\\ \text{step m.n}\\ \text{part m}\\ \text{formula f}\\ \text{F(x)}\\ \text{F(proposition)}\\ \text{all steps}\\ \text{all parts}\\ \text{all formulas}\\ \text{all forms}\\ \text{all values}\\ \text{all}\\ \text{time}\\ \text{timer}\\ \text{size}\\ \text{item-list}\end{array}\right\}$	O S S S F	Types out the specified information on the user's console. Several individual TYPE commands may be combined into one (except for TYPE "any text" or TYPE ITEM-LIST). The command Type ... in form n causes the listed items to be typed out in the format specified by form n. n can be a numeric value (for example, form 3) or it can be a numeric formula (for example, form (2*x-y)).
USE FILE filename (device)	F	Makes an external storage file available for use. The external file thus addressed remains open for use (by DISCARD, FILE, RECALL, and TYPE ITEM-LIST commands) until another USE command is given or the AID program is terminated.

Command Format Symbology

L = letter
S = subscripted letter.
m, n, p = numeric values.
f = formula.
F = function.
range = an interative sequence or series of values.

Type Symbology

D = Can be given directly only
I = Can be given indirectly only.
O = Operational command.
F = File command.
S = Special command.

Table A-2B
File Command Subset

Command Format	Section Reference	Description
DISCARD ITEM m (code)	5.4	Deletes item #m from the external storage file currently in use. (Code) is optional.
FILE { L / S / S(m,n) / form m / step m.n / part m / formula f / all steps / all parts / all formulas / all forms / all values / all } AS ITEM n (code)	5.7	Stores the specified item in the external storage file currently open. Immediate storage is not affected in any way. (Code) is optional.
RECALL ITEM m (code)	5.15	Reads item #m, previously stored by a FILE command, from the currently open external storage file into immediate storage. (Code) is optional and is for documentation only.
TYPE ITEM-LIST	5.20	Obtains a typeout of the directory of the currently open external storage file.
USE FILE filename (device)	5.21	Makes an external storage file available for use. The external storage file thus addressed remains open for use (by DISCARD, FILE, RECALL, and TYPE ITEM-LIST commands) until another USE command is given or the AID program is terminated.

Table A-3
AID Character Set

Standard Math Symbol	AID Symbol	Typing Method		JOSS Symbol	Notes
		Model 37	Models 33 and 35		
	A through Z	Strike appropriate key with SHIFT.	Strike appropriate key; no SHIFT.	A through Z	
	a through z	Strike appropriate key without SHIFT.	Not available; use upper-case letters.	a through z	
	0,1 through 9	Strike appropriate key; no SHIFT.	Strike appropriate key; no SHIFT.	0,1 through 9	
Operators: | | (absolute)	! !	Strike the !, 1 key with SHIFT.	Strike the !, 1 key with SHIFT.	| |	
[ ] (brackets)	[ ]	Strike appropriate keys.	[Strike K with SHIFT] Strike M with SHIFT	[ ]	
( ) (parentheses)	( )	(Strike the (,8 key with SHIFT.) Strike the),9 key with SHIFT.	(Strike the (,8 key with SHIFT.) Strike the),9 key with SHIFT.	( )	
x^e (exponent)	x↑e	Strike the ~, ∧ key; no SHIFT.	Strike the ↑,N key with SHIFT.	*	
/ (divide)	/	Strike the ?,/ key; no SHIFT.	Strike the ?,/ key; no SHIFT.	/	
· (multiplication)	*	Strike the *,: key with SHIFT.	Strike the *,: key with SHIFT.	·	
+ (addition)	+	Strike the +,; key with SHIFT.	Strike the +,; key with SHIFT.	+	
- (subtraction)	-	Strike the =,- key; no SHIFT.	Strike the =,- key; no SHIFT.	-	
Boolean Expressions: = (equal)	=	Strike the =,- key with SHIFT.	Strike the =,- key with SHIFT.	=	
≠ (not equal)	#	Strike the #,3 key with SHIFT.	Strike the #,3 key with SHIFT.	≠	

Table A-3 (Cont)
AID Character Set

Standard Math Symbol	AID Symbol	Typing Method: Model 37	Typing Method: Models 33 and 35	JOSS Symbol	Notes
≤ (equal to or less than)	< = (2 characters)	Strike the <, < key; then strike the =, -key with SHIFT.	Strike the <, .key with SHIFT; then strike the =,- key with SHIFT.	≤	
≥ (equal to or greater than)	> = (2 characters)	Strike the >, > key; then strike the =,-key with SHIFT.	Strike the >, . key with SHIFT; then strike the =,- key with SHIFT.	≥	
	RUBOUT (types back as deleted characters between \\.	Strike DELETE key to erase each preceding character in error; then type correctly. Example: TPE\\EP\\YPE PART 1.	Strike RUBOUT key to erase each preceding character in error; then type correctly.	BACK-SPACE and type over	Used to correct typing errors.
null item	←	Strike the _'— key with or without SHIFT.	Strike O key with SHIFT.	_ (underscore)	
	$ (current line number)	Strike the $,4 key with SHIFT.	Strike the $,4 key with SHIFT.	$	
	* (cancel entire line)	Strike the *,: key with SHIFT.	Strike the *,: key with SHIFT.	*	

Table A-4
AID Diagnostic Messages

Message	Meaning
x = ???	A value has not been supplied by the user for variable x.
DONE.	Signals completion of a File command (DISCARD, FILE, RECALL).
DONE. I'M READY TO GO $\left\{\begin{array}{l}\text{AT}\\\text{FROM}\\\text{IN}\end{array}\right\}$ STEP m.n.	...AT STEP m.n... Task was suspended by an interruption or error during the interpretation of an indirect step. ...FROM STEP m.n... Task was suspended by a stopping command. ...IN STEP m.n... Task was suspended during an indirectly initiated DO command. AID resumes execution whenever the user types Go.
DONE. I'M READY TO GO $\left\{\begin{array}{l}\text{AT}\\\text{FROM}\\\text{IN}\end{array}\right\}$ STEP m.n, ALTHO I CAN'T FIND IT.	Same as above, except that the step at which AID is prepared to resume can no longer be found in immediate storage. Possibly, a direct command (or a routine initiated by a parenthetical DO) has deleted the step in the interim. Upon receipt of a GO command from the user, AID will attempt to resume at the step following the missing step.
DON'T GIVE THIS COMMAND $\left\{\begin{array}{l}\text{DIRECTLY}\\\text{INDIRECTLY}\end{array}\right\}$	This command can be given only indirectly (TO, DONE, STOP, DEMAND) or only directly (CANCEL, GO).
EH?	The previously entered line is incorrect. Indirect commands: The step number was incorrectly typed. Direct LET commands: LET x portion is incorrect. Other direct commands: A space was omitted. The terminating period was omitted. The command is not legitimate. An expression is incorrectly written. To continue, retype the command correctly.

Table A-4 (Cont)
AID Diagnostic Messages

Message	Meaning
ERROR AT STEP m.n: EH?	The step number is correct, but the command is incorrect. a. Request a typeout of the step in error. b. Check for the errors listed under "Eh ?". c. Retype the command correctly. d. Type GO. to continue.
ERROR AT STEP m.n:	The step in error refers to a nonexistent step or part.
I CAN'T FIND THE REQUIRED $\left\{\begin{array}{l}\text{STEP}\\ \text{FORM}\\ \text{PART}\\ \text{FORMULA}\end{array}\right\}$.	Correct the error and type GO. to continue.
ERROR AT STEP m.n: (IN FORMULA x): z = ???	The variable z has not been assigned a value by the user. Check for any other errors, define variable z correctly, and type GO. to continue.
ERROR IN FORMULA x: EH?	(Following a direct command in which x was used) The form of the expression for x is in error. a. Request a typeout of formula x. b. Check for the errors listed under "Eh ?". c. Formula x may be correctly written, but the definition of one or more identifiers is not consistent with their use in formula x.
FILE NUMBER MUST BE POSITIVE INTEGER <=2750.	The filename of a USE command must not be greater than the value 2750.
FORM NUMBER MUST BE INTEGER AND 1 <= FORM < 10 ↑ 9.	Form numbers must be integers in the range 1 through 10^9-1.
I CAN'T EXPRESS THE VALUE IN YOUR FORM.	A value cannot be expressed in the format specified by the FORM (e.g., the value is too large to specify in fixed point notation). To correct, follow the steps given under "I HAVE TOO MANY VALUES FOR THE FORM."
I CAN'T FIND THE REQUIRED $\left\{\begin{array}{l}\text{FORM}\\ \text{ITEM}\\ \text{PART}\\ \text{STEP}\end{array}\right\}$.	Either the element has never been defined or has been deleted.

Table A-4 (Cont)
AID Diagnostic Messages

Message	Meaning
I CAN'T MAKE OUT YOUR FIELDS IN THE FORM.	The fields in the form specified were typed in such a way that AID cannot distinguish their beginning or ending. Possibly, there are either no fields in the form or two or more are run together with no intervening space.
I HAVE AN ARGUMENT <=0 FOR LOG.	The argument for the LOG function must be greater than 0.
I HAVE A NEGATIVE ARGUMENT FOR SQRT.	Square root arguments must be positive.
I HAVE A NEGATIVE BASE TO A FRACTIONAL POWER.	An attempt was made to raise a negative value to a fractional power. For example, Type (-y)↑(1/2).
I HAVE AN OVERFLOW.	Some number has exceeded 9.99999999.10↑99 in magnitude.
I HAVE A ZERO DIVISOR.	An attempt was made to divide by 0.
I HAVE NOTHING TO DO.	The user has typed GO., but there is no currently stopped process which can be continued.
I HAVE TOO FEW VALUES.	An insufficient number of arguments have been supplied for a function.
I HAVE TOO MANY VALUES FOR THE FORM.	There are not enough fields in the form to receive all the values to be typed. a. Type the form and the values. b. Check for errors. c. Change either the TYPE command or the FORM to make them compatible and then type GO. to continue.
I HAVE ZERO TO A NEGATIVE POWER.	An attempt was made to raise 0 to a negative power.
ILLEGAL SET OF VALUES FOR ITERATION.	An error has been detected in a range clause of a function or a DO command, such that the ending value can never be reached (e.g., the increment is 0).
I'M AT STEP m.n.	When the user responds to a DEMAND-produced request (x= *) with a carriage return only, AID types back this message.

Table A-4 (Cont)
AID Diagnostic Messages

Message	Meaning
INDEX VALUE MUST BE INTEGER AND !INDEX! <250	All index values (subscripts) must be integral and must have an absolute value of <250.
I NEED INDIVIDUAL VALUES FOR A FORM.	A command was given to type a subscripted variable in a form (e.g., Type B in form 1, where B is a subscripted variable). Individual values only can be specified for TYPE....IN FORM n, commands.
I RAN OUT OF SPACE.	User's immediate memory is filled due to one of the following errors. a. Endless loops because of DO commands or because DO was typed instead of TO. b. Unlimited recursive definition. c. Variable x defined in terms of y, and variable y defined in terms of x via LET command. d. Program is too large for available memory; use TYPE SIZE command to determine how much immediate storage has been used. File commands can be used to store parts of the routine and execute them one at a time.
I RAN OUT OF FILE SPACE.	DECtape directory is full (limit = 22 items).
ITEM NUMBER MUST BE <=25.	The item number in file commands (DISCARD, FILE, RECALL) must be less than or equal to 25.
NUMBER-OF-TIMES MUST BE INTEGER AND >= 0.	The value specified in the TIMES clause of a DO command must be a positive integer.
PART NUMBER MUST BE INTEGER AND 1 < =PART< 10↑9.	Part numbers must be integers and in the range 1 through 10^9-1.
PLEASE DELETE THE ITEM OR USE A NEW ITEM NUMBER.	The user has attempted to FILE information into an item which already exists on the currently open external storage file. The user must either DISCARD the item prior to filing the new information or use a different item number in the FILE command.
PLEASE KEEP !X!<100 FOR SIN(X) AND COS(X).	Arguments for the SINE and COSINE functions must be less than 100.
PLEASE LIMIT ID'S TO 5 LETTERS AND/OR DIGITS.	Filename in a USE file command or code in a DISCARD, FILE, or RECALL command exceeds five characters in length or contains special characters.

Table A-4 (Cont)
AID Diagnostic Messages

Message	Meaning
PLEASE LIMIT LINES TO 78 UNITS (CHECK MARGIN STOPS) SAY AGAIN:	User typeins are limited to single-line, 78-character strings.
PLEASE LIMIT NUMBERS TO 9 SIGNIFICANT DIGITS.	Numeric values are limited to nine significant digits.
PLEASE LIMIT NUMBER OF INDICES TO 10.	The number of subscripts following an identifier cannot exceed 10.
PLEASE LIMIT NUMBER OF PARAMETERS TO TEN.	The number of arguments for a function is limited to 10.
PLEASE LIMIT STEP LABELS TO 9 SIGNIFICANT DIGITS.	Step numbers can be up to nine digits in length.
REVOKED. I RAN OUT OF SPACE.	See "I RAN OUT OF SPACE."
ROGER.	Signals successful completion of a USE file command.
SOMETHING'S WRONG. I CAN'T ACCESS THE FILES.	A system I/O error (or other type of AID error) has occurred. Begin again.
SOMETHING'S WRONG. TRY AGAIN.	AID has found something unusual in its internal records or has received contradictory signals from its I/O routine. Begin again.
SORRY. SAY AGAIN:	A transmission error occurred on the previous typein. This message is preceded by the erroneous line with # symbols typed where the failure occurred. Retype the line.
STEP NUMBER MUST SATISFY 1<= STEP< 10↑9.	Step numbers must be in the range 1 through 10^9-1.
STOPPED BY STEP m.n.	Process has been temporarily halted by a STOP command at step m.n.
YOU HAVEN'T TOLD ME WHAT FILE TO USE.	The user has issued a DISCARD, FILE, RECALL, or TYPE ITEM-LIST command before he has given a USE file command.

Book 5

Programming In FORTRAN

CONTENTS

		Page
CHAPTER 1	INTRODUCTION TO THE FORTRAN LANGUAGE	
1.1	Line Format	5-15
1.1.1	Statement Number Field	5-15
1.1.2	Line Continuation Field	5-15
1.1.3	Statement Field	5-16
1.1.4	Comment Line	5-16
1.2	Character Set	5-17
CHAPTER 2	CONSTANTS, VARIABLES, AND EXPRESSIONS	
2.1	Constants	5-19
2.1.1	Integer Constants	5-19
2.1.2	Real Constants	5-19
2.1.3	Double Precision Constants	5-20
2.1.4	Octal Constants	5-20
2.1.5	Complex Constants	5-20
2.1.6	Logical Constants	5-21
2.1.7	Literal Constants	5-21
2.2	Variables	5-22
2.2.1	Scalar Variables	5-22
2.2.2	Array Variables	5-22
2.3	Expressions	5-24
2.3.1	Numeric Expressions	5-24
2.3.2	Logical Expressions	5-26
CHAPTER 3	THE ARITHMETIC STATEMENT	
3.1	General Description	5-29
CHAPTER 4	CONTROL STATEMENTS	
4.1	GO TO Statement	5-31
4.1.1	Unconditional GO TO Statements	5-31
4.1.2	Computer GO TO Statements	5-31
4.1.3	Assigned GO TO Statement	5-32
4.2	IF Statement	5-32

CONTENTS (Cont)

		Page
4.2.1	Numerical IF Statements	5-33
4.2.2	Logical IF Statements	5-33
4.3	DO Statement	5-34
4.4	CONTINUE Statement	5-36
4.5	PAUSE Statement	5-36
4.6	STOP Statement	5-37
4.7	END Statement	5-37

CHAPTER 5 DATA TRANSMISSION STATEMENTS

5.1	Nonexecutable Statements	5-39
5.1.1	FORMAT Statement	5-39
5.1.2	NAMELIST Statement	5-49
5.2	Data Transmission Statements	5-51
5.2.1	Input/Output Lists	5-52
5.2.2	Input/Output Records	5-53
5.2.3	PRINT Statement	5-53
5.2.4	PUNCH Statement	5-54
5.2.5	TYPE Statement	5-54
5.2.6	WRITE Statement	5-54
5.2.7	READ Statement	5-56
5.2.8	REREAD Statement	5-57
5.2.9	ACCEPT Statement	5-58
5.3	Device Control Statements	5-58
5.4	Encode and Decode Statements	5-59

CHAPTER 6 SPECIFICATION STATEMENTS

6.1	Storage Specification Statements	5-62
6.1.1	DIMENSION Statement	5-62
6.1.2	COMMON Statement	5-64
6.1.3	EQUIVALENCE Statement	5-65
6.1.4	EQUIVALENCE and COMMON	5-66
6.2	Data Specification Statements	5-66
6.2.1	DATA Statement	5-66
6.2.2	BLOCK DATA Statement	5-68

CONTENTS (Cont)

Page

6.3 Type Declaration Statements 5-68

6.3.1 IMPLICIT Statement 5-68

CHAPTER 7 SUBPROGRAM STATEMENTS

7.1 Dummy Identifiers 5-71

7.2 Library Subprograms 5-71

7.3 Arithmetic Function Definition Statement 5-71

7.4 FUNCTION Subprograms 5-72

7.4.1 FUNCTION Statement 5-72

7.5 SUBROUTINE Subprograms 5-74

7.5.1 SUBROUTINE Statement 5-74

7.5.2 CALL Statement 5-75

7.5.3 RETURN Statement 5-75

7.6 BLOCK DATA Subprograms 5-76

7.6.1 BLOCK DATA Statement 5-76

7.7 EXTERNAL Statement 5-76

7.8 Summary of PDP-10 FORTRAN IV Statements 5-77

CHAPTER 8 LIB40

8.1 The FORTRAN Operating System 5-83

8.1.1 FORSE. 5-83

8.1.2 I/O Conversion Routines 5-84

8.1.3 FORTRAN UUOs 5-85

8.2 Science Library and FORTRAN Utility Subprograms 5-86

8.2.1 FORTRAN IV Library Functions 5-86

8.2.2 FORTRAN IV Library Subroutines 5-90

CHAPTER 9 SUBPROGRAM CALLING SEQUENCES

9.1 Macro Subprograms Called by FORTRAN Main Programs 5-95

9.1.1 Calling Sequences 5-95

9.1.2 Returning of Answers 5-96

9.1.3 Use of Accumulators 5-96

9.1.4 Examples of Subprogram Linkage 5-96

CONTENTS (Cont)

		Page
9.2	Macro Main Programs Which Reference FORTRAN Subprograms	5-103
9.2.1	Calling Sequences	5-103
9.2.2	Returning of Answers	5-103
9.2.3	Example of Subprogram Linkage	5-104
CHAPTER 10	ACCUMULATOR CONVENTIONS FOR MAIN PROGRAMS AND SUBPROGRAMS	
10.1	Locations	5-111
10.2	Accumulators	5-111
10.2.1	Accumulators 0 and 1	5-111
10.2.2	Accumulators 2 Through 15	5-112
10.2.3	Accumulators 16 and 17	5-112
10.3	UUOS	5-112
10.4	Subprograms Called by JSA 16, Address	5-112
10.5	Subprograms Called by PUSHJ 17, Address	5-112
10.6	Subprograms Called by UUOS	5-113
CHAPTER 11	SWITCHES AND DIAGNOSTICS	
11.1	FORTRAN Switches and Diagnostics	5-115
CHAPTER 12	RELATED FEATURES OF THE PDP-10	
12.1	ASCII Character Set	5-125
12.2	PDP-10 Word Formats	5-126
12.3	FORTRAN Input/Output	5-127
12.3.1	Logical and Physical Peripheral Device Assignments	5-128
12.3.2	DECtape and Disk Usage	5-128
12.3.3	Magnetic Tape Usage	5-130
12.4	PDP-10 Instruction Set	5-132
APPENDIX A	THE SMALL FORTRAN IV COMPILER	

ILLUSTRATIONS

		Page
1-1	Typical FORTRAN Coding Form	5-16
2-1	Array Storage	5-23
4-1	Nested DO Loops	5-35

TABLES

2-1	Types of Resultant Subexpressions	5-25
3-1	Allowed Assignment Statements	5-30
5-1	Magnitude of Internal Data	5-41
5-2	Numeric Field Codes	5-42
5-3	Device Control Statements	5-59
8-1	I/O Conversion Routine	5-84
8-2	FORTRAN UUOs	5-85
8-3	FORTRAN IV Library Functions	5-87
8-4	FORTRAN IV Library Subroutines	5-90
10-1	Accumulator Conventions for PDP-10 FORTRAN IV Compiler and Subprograms	5-113
11-1	FORTRAN Compiler Switch Options	5-115
11-2	FORTRAN Compiler Diagnostics (Command Errors)	5-116
11-3	FORTRAN Compiler Diagnostics (Compilation Errors)	5-117
11-4	FORTRAN Operating System Diagnostics (Execution Errors)	5-120
12-1	ASCII Character Set	5-125
12-2	PDP-10 FORTRAN IV Standard Peripheral Devices	5-127
12-3	Device Table for FORTRAN IV	5-129

PREFACE

This is a reference manual describing the specific statements and features of the FORTRAN IV language for the PDP-10. It is written for the experienced FORTRAN programmer who is interested in writing and running FORTRAN IV programs alone or in conjunction with MACRO-10 programs in the single-user or time-sharing environment. Familiarity with the basic concepts of FORTRAN programming on the part of the user is assumed. PDP-10 FORTRAN IV conforms to the requirements of the USA Standard FORTRAN.

The FORTRAN compiler translates source programs written in the FORTRAN IV language into the machine language of the PDP-10. This translated version of the FORTRAN program exists as a retrievable, relocatable binary file on some storage device. All relocatable binary filenames have the extension .REL if they reside on a directory-oriented device (disk or DECtape). Binary files may also be created by the MACRO-10 assembler (see Chapter 9)[1].

In order for the FORTRAN program to be processed, the Linking Loader must load the relocatable binary file into core memory. Also loaded are any relocatable binary files found in the FORTRAN library (LIB40) which are necessary for the program's execution. Within the FORTRAN source program, the library files may be called explicitly, such as SIN, in the statement

```
          X = SIN(Y)
```

or implicitly, such as FLOUT., the floating-point to ASCII conversion routine, which is implied in the following statements.

```
              PRINT    3,X
      3       FORMAT(1X,F4.2)
```

A FORTRAN main program and its FORTRAN and/or MACRO-10 subprograms may be compiled or assembled separately and then linked together by the Linking Loader at load time. The core image may then be saved on a storage device. When saved on a directory storage device, these files have the extension .SAV in a multiprogramming Monitor system and .SVE in a single-user Monitor system.

The Time-Sharing Monitors act as the interface between the user and the computer so that all users are protected from one another and appear to have system resources available to themselves. Several user programs are loaded into core at once and the Time-Sharing Monitors schedule each program to run for a certain length of time. All Monitors direct data flow between I/O devices and user programs, making the programs device independent, and overlap I/O operations concurrently with computations.

In a multiprogramming system, all jobs reside in core and the scheduler decides which of these jobs should run. In a swapping system, jobs can exist on an external storage device (usually disk) as well as in core. The scheduler

[1]For further information on the MACRO-10 assembler, see the MACRO-10 ASSEMBLER manual, DEC-10-AMZA-D.

decides not only which job is to run but also when a job is to be swapped out onto the disk or brought back into core.

The number of users that can be handled by a given size time-sharing configuration is further increased by using the reentrant user-programming capability. This means that a sequence of instructions may be entered by more than one user job at a time. Therefore, a single copy of a reentrant program may be shared by a number of users at the same time to increase system economy. The FORTRAN compiler and operating system are both reentrant.

SECTION I

The PDP-10 FORTRAN IV Language

The seven chapters of this section deal with the PDP-10 FORTRAN IV language. Included in these chapters are the language elements of FORTRAN IV and the five categories of FORTRAN IV statements (arithmetic, control, input/output, specification, and subprogram).

CHAPTER 1
INTRODUCTION TO THE FORTRAN LANGUAGE

The term FORTRAN IV (FORmula TRANslation) is used interchangeably to designate both the FORTRAN IV language and the FORTRAN IV translator or compiler. The FORTRAN IV language is composed of mathematical-form statements constructed in accordance with precisely formulated rules. FORTRAN IV programs consist of meaningful sequences of FORTRAN statements intended to direct the computer to perform the specified operations and computations.

The FORTRAN IV compiler is itself a computer program that examines FORTRAN IV statements and tells the computer how to translate the statements into machine language. The compiler runs in a minimum of 9K of core. The program written in FORTRAN IV language is called the source program. The resultant machine language program is called the object program. Digital's small FORTRAN compiler, which runs in 5.5K of core, is virtually identical to the larger compiler, except for differences explained in Appendix 2. Operating procedures and diagnostic messages for both compilers are explained in the PDP-10 System Users Guide (DEC-10-NGCC-D).

1.1 LINE FORMAT

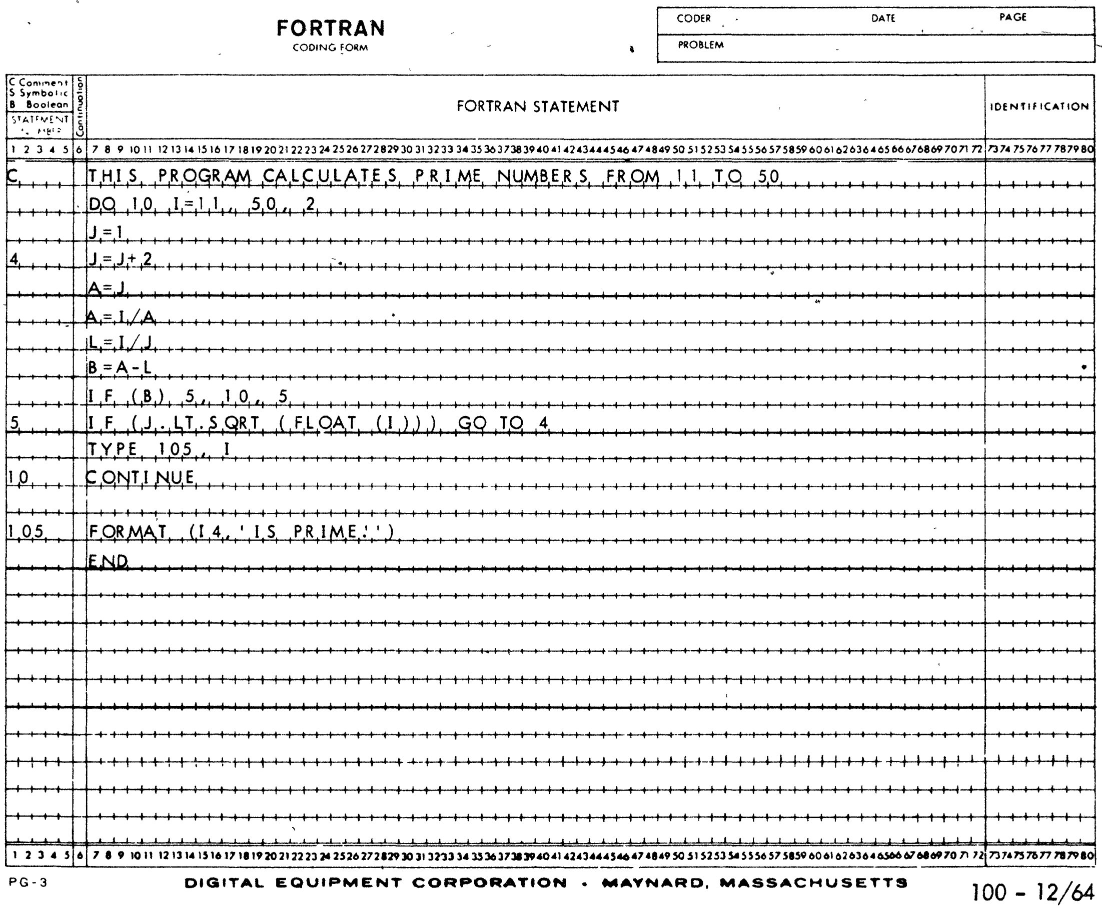
Each line of a FORTRAN program consists of three fields: statement number field, line continuation field, and statement field. A typical FORTRAN program is shown in Figure 1-1.

1.1.1 Statement Number Field

A statement number consists of from one to five digits in columns 1-5. Leading zeros and all blanks in this field are ignored. Statement numbers may be in any order and must be unique. Any statement referenced by another statement must have a statement number. For source programs prepared on a teletypewriter, a horizontal tab may be used to skip to the statement field. This is the only place a tab is not treated as a space.

1.1.2 Line Continuation Field

If a FORTRAN statement is so large that it cannot conveniently fit into one statement field, the statement fields of up to 19 additional lines may be used to specify the complete statement. Any line which is not continued, or the first line of a sequence of continued lines, must have a blank or zero in column 6. Continuation lines must



FORTRAN
CODING FORM

CODER DATE PAGE
PROBLEM

C Comment / S Symbolic / B Boolean / STATEMENT NUMBER | Continuation | FORTRAN STATEMENT | IDENTIFICATION

```
C     THIS PROGRAM CALCULATES PRIME NUMBERS FROM 11 TO 50
      DO 10 I=11, 50, 2
      J=1
4     J=J+2
      A=J
      A=I/A
      L=I/J
      B=A-L
      IF (B) 5, 10, 5
5     IF (J.LT.SQRT (FLOAT (I))) GO TO 4
      TYPE 105, I
10    CONTINUE

105   FORMAT (I4,' IS PRIME.')
      END
```

PG-3 DIGITAL EQUIPMENT CORPORATION • MAYNARD, MASSACHUSETTS 100 - 12/64

Figure 1-1 Typical FORTRAN Coding Form

have a character other than blank or zero in column 6. If a continuation line is desired when a TAB is used in the statement number field, a digit from 1 to 9 must immediately follow the TAB.

1.1.3 Statement Field

Any FORTRAN statement, as described in later sections, may appear in the statement field (columns 7-72). Except for alphanumeric data within a FORMAT statement, DATA statement, or literal constant, blanks (spaces) and TABS are ignored and may be used freely for appearance purposes.

1.1.4 Comment Line

Any line which starts with the letter C in column 1 is interpreted as a line of comments. Comment lines are printed onto any listings requested but are otherwise ignored by the compiler. Columns 2-72 may be used in any format for comment purposes. A comment line must not immediately precede a continuation line.

1.2 CHARACTER SET

The following characters are used in the FORTRAN IV language:

Blank	0	@	P
!	1	A	Q
"	2	B	R
#	3	C	S
$	4	D	T
%	5	E	U
&	6	F	V
'	7	G	W
(	8	H	X
)	9	I	Y
*	:	J	Z
+	;	K	↑
,	<	L	
-	=	M	
.	>	N	
/	?	O	

NOTE

ASCII characters greater than Z (132_8) are replaced by the error character "↑". See Chapter 12 for the internal representation of these characters.

CHAPTER 2
CONSTANTS, VARIABLES, AND EXPRESSIONS

The rules for defining constants and variables and for forming expressions are described in this chapter.

2.1 CONSTANTS

Seven types of constants are permitted in a FORTRAN IV source program: integer or fixed point, real or single-precision floating point, double-precision floating point, octal, complex, logical, and literal.

2.1.1 Integer Constants

An integer constant consists of from one to eleven decimal digits written without a decimal point. A negative constant must be preceded by a minus sign. A positive constant may be preceded by a plus sign.

```
    Examples:     3
                  +10
                  -528
                  8085
```

An integer constant must fall within the range $-2^{35}+1$ to $2^{35}-1$. When used for the value of a subscript or as an index in a DO statement, the value of the integer is taken as modulo 2^{18}.

2.1.2 Real Constants

Real constants are written as a string of decimal digits including a decimal point. A real constant may consist of any number of digits but only the leftmost 9 digits appear in the compiled program. Real constants may be given a decimal scale factor by appending an E followed by a signed integer constant. The field following the letter E must not be blank, but may be zero.

```
    Examples:     15.
                   0.0
                    .579
                  -10.794
                   5.0E3(i.e., 5000.)
                   5.0E+3(i.e., 5000)
                   5.0E-3(i.e., 0.005)
```

A real constant has precision to eight digits. The magnitude must lie approximately within the range 0.14×10^{-38} to 1.7×10^{38}. Real constants occupy one word of PDP-10 storage.

2.1.3 Double Precision Constants

A double precision constant is specified by a string of decimal digits, including a decimal point, which are followed by the letter D and a signed decimal scale factor. The field following the letter D must not be blank, but may be zero.

```
Examples:      24.671325982134D0
                3.6D2 (i.e., 360.)
                3.6D-2 (i.e., .036)
                3.0D0
```

Double precision constants have precision to 16 digits. The magnitude of a double precision constant must lie approximately between 0.14×10^{-38} and 1.7×10^{38}. Double-precision constants occupy two words of PDP-10 storage.

2.1.4 Octal Constants

A number preceded by a double quote represents an octal constant. An octal constant may appear in an arithmetic or logical expression or a DATA statement. Only the digits 0-7 may be used and only the last twelve digits are significant. A minus sign may precede the octal number, in which case the number is negated. A maximum of 12 octal digits are stored in each 36-bit word.

```
Examples:      "7777
               "-31563
```

2.1.5 Complex Constants

FORTRAN IV provides for direct operations on complex numbers. Complex constants are written as an ordered pair of real constants separated by a comma and enclosed in parentheses.

```
Examples:      (.70712, -.70712)
               (8.763E3,2.297)
```

The first constant of the pair represents the real part of the complex number, and the second constant represents the imaginary part. The real and imaginary parts may each be signed. The enclosing parentheses are part of the constant and always appear, regardless of context. Each part is internally represented by one single-precision floating point word. They occupy consecutive locations of PDP-10 storage.

FORTRAN IV arithmetic operations on complex numbers, unlike normal arithmetic operations, must be of the form:

$$A \pm B = a_1 \pm b_1 + i(a_2 \pm b_2)$$

$$A*B = (a_1b_1 - a_2b_2) + i(a_2b_1 + a_1b_2)$$

$$A/B = \frac{(a_1b_1 + a_2b_2)}{b_1^2 + b_2^2} + i\,\frac{(a_2b_1 - a_1b_2)}{b_1^2 + b_2^2}$$

where $A = a_1 + ia_2$, $B = b_1 + ib_2$, and $i = \sqrt{-1}$.

2.1.6 Logical Constants

The two logical constants, .TRUE. and .FALSE., have the internal values -1 and 0, respectively. The enclosing periods are part of the constant and always appear.

Logical constants may be entered in DATA or input statements as signed octal integers (-1 and 0). Logical quantities may be operated on in either arithmetic or logical statements. Only the sign is tested to determine the truth value of a logical variable.

2.1.7 Literal Constants

A literal constant may be in either of two forms:

a. A string of alphanumeric and/or special characters enclosed in single quotes; two adjacent single quotes within the constant are treated as one single quote.

b. A string of characters in the form

$nHx_1x_2\ldots x_n$

where $x_1x_2\ldots x_n$ is the literal constant, and n is the number of characters following the H.

Literal constants may be entered in DATA statements or input statements as a string of up to 5 7-bit ASCII characters per variable (10 characters if the variable is double-precision or complex). Literal constants may be operated on in either arithmetic or logical statements.

NOTE

Literal constants used as subprogram arguments will have a zero word as an end-of-string indicator.

Examples:
```
CALL SUB ('LITERAL CONSTANT')
'DONT''T'
5HDON'T
A = 'FIVE' + 42
B = (5HABCDE .AND. ''376)/2
```

2.2 VARIABLES

A variable is a quantity whose value may change during the execution of a program. Variables are specified by name and type. The name of a variable consists of one or more alphanumeric characters, the first one of which must be alphabetic. Only the first six characters are interpreted as defining the variable name. The type of variable (integer, real, logical, double precision, or complex) 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 a fixed point (integer) variable; any other first letter indicates a floating-point (real) variable. Variables of any type may be either scalar or array variables.

2.2.1 Scalar Variables

A scalar variable represents a single quantity.

Examples:
```
A
G2
POPULATION
```

2.2.2 Array Variables

An array variable represents a single element of an n dimensional array of quantities. The variable is denoted by the array name followed by a subscript list enclosed in parentheses. The subscript list is a sequence of integer expressions, separated by commas. The expressions may be of any form or type providing they are explicitly changed to type integer when each is completely evaluated. Each expression represents a subscript, and the values of the expressions determine the array element referred to. For example, the row vector A_I would be represented by the subscripted variable A(J), and the element, in the second column of the first row of the square matrix A, would be represented by A(1,2). Arrays may have any number of dimensions.

Examples:
```
Y(1)
STATION (K)
A (3* K+2, I, J-1)
```

The three arrays above (Y, STATION, and A) would have to be dimensioned by a DIMENSION, COMMON, or type declaration statement prior to their first appearance in an executable statement or in a DATA or NAMELIST statement. (Array dimensioning is discussed in Chapter 6).

1-Dimensional Array A(10)

A(1)	A(2)	A(3)	A(4)	A(5)	A(6)	A(7)	A(8)	A(9)	A(10)

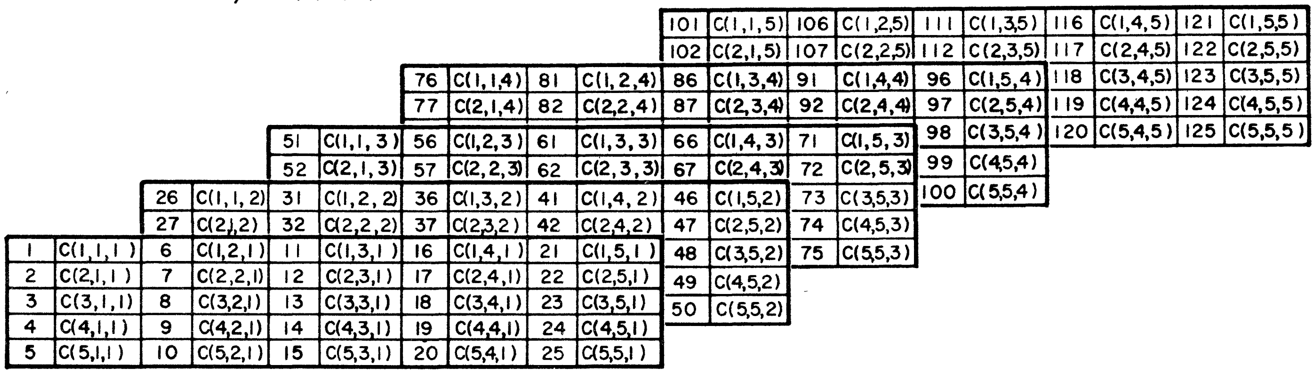
CONSECUTIVE STORAGE LOCATIONS ——————

2-Dimensional Array B(5,5)

1	B(1,1)	6	B(1,2)	11	B(1,3)	16	B(1,4)	21	B(1,5)
2	B(2,1)	7	B(2,2)	12	B(2,3)	17	B(2,4)	22	B(2,5)
3	B(3,1)	8	B(3,2)	13	B(3,3)	18	B(3,4)	23	B(3,5)
4	B(4,1)	9	B(4,2)	14	B(4,3)	19	B(4,4)	24	B(4,5)
5	B(5,1)	10	B(5,2)	15	B(5,3)	20	B(5,4)	21	B(5,5)

B(3,1) IS THE THIRD STORAGE WORD IN SEQUENCE
B(3,4) IS THE EIGHTEENTH STORAGE WORD IN SEQUENCE

3-Dimensional Array C(5,5,5)



101	C(1,1,5)	106	C(1,2,5)	111	C(1,3,5)	116	C(1,4,5)	121	C(1,5,5)
102	C(2,1,5)	107	C(2,2,5)	112	C(2,3,5)	117	C(2,4,5)	122	C(2,5,5)
						118	C(3,4,5)	123	C(3,5,5)
						119	C(4,4,5)	124	C(4,5,5)
						120	C(5,4,5)	125	C(5,5,5)

76	C(1,1,4)	81	C(1,2,4)	86	C(1,3,4)	91	C(1,4,4)	96	C(1,5,4)
77	C(2,1,4)	82	C(2,2,4)	87	C(2,3,4)	92	C(2,4,4)	97	C(2,5,4)
								98	C(3,5,4)
								99	C(4,5,4)
								100	C(5,5,4)

51	C(1,1,3)	56	C(1,2,3)	61	C(1,3,3)	66	C(1,4,3)	71	C(1,5,3)
52	C(2,1,3)	57	C(2,2,3)	62	C(2,3,3)	67	C(2,4,3)	72	C(2,5,3)
								73	C(3,5,3)
								74	C(4,5,3)
								75	C(5,5,3)

26	C(1,1,2)	31	C(1,2,2)	36	C(1,3,2)	41	C(1,4,2)	46	C(1,5,2)
27	C(2,1,2)	32	C(2,2,2)	37	C(2,3,2)	42	C(2,4,2)	47	C(2,5,2)
								48	C(3,5,2)
								49	C(4,5,2)
								50	C(5,5,2)

1	C(1,1,1)	6	C(1,2,1)	11	C(1,3,1)	16	C(1,4,1)	21	C(1,5,1)
2	C(2,1,1)	7	C(2,2,1)	12	C(2,3,1)	17	C(2,4,1)	22	C(2,5,1)
3	C(3,1,1)	8	C(3,2,1)	13	C(3,3,1)	18	C(3,4,1)	23	C(3,5,1)
4	C(4,1,1)	9	C(4,2,1)	14	C(4,3,1)	19	C(4,4,1)	24	C(4,5,1)
5	C(5,1,1)	10	C(5,2,1)	15	C(5,3,1)	20	C(5,4,1)	25	C(5,5,1)

C(1,3,2) is the 36th storage word in sequence.
C(1,1,5) is the 101st storage word in sequence.

Figure 2-1 Array Storage

Arrays are stored in increasing storage locations with the first subscript varying most rapidly and the last subscript varying least rapidly. For example, the 2-dimensional array B(I,J) is stored in the following order: B (1,1), B (2,1),..., B (I,1),B (1,2),B (2,2),...,B (I,2),...,B (I,J).

2.3 EXPRESSIONS

Expressions may be either numeric or logical. To evaluate an expression, the object program performs the calculations specified by the quantities and operators within the expression.

2.3.1 Numeric Expressions

A numeric expression is a sequence of constants, variables, and function references separated by numeric operators and parentheses in accordance with mathematical convention and the rules given below.

The numeric operators are +, -, *, /, **, denoting, respectively, addition, subtraction, multiplication, division, and exponentiation.

In addition to the basic numeric operators, function references are also provided to facilitate the evaluation of functions such as sine, cosine, and square root. A function is a subprogram which acts upon one or more quantities, called arguments, to produce a single quantity called the function value. Function references are denoted by the identifier, which names the function (such as SIN, COS, etc.), followed by an argument list enclosed in parentheses:

```
identifier(argument, argument, ..., argument)
```

At least one argument must be present. An argument may be an expression, an array identifier, a subprogram identifier, or an alphanumeric string.

Function type is given by the type of the identifier which names the function. The type of the function is independent of the types of its arguments. (See Chapter 7, Section 7.4.1.1.)

A numeric expression may consist of a single element (constant, variable, or function reference):

```
2.71828
Z(N)
TAN(THETA)
```

Compound numeric expressions may be formed by using numeric operations to combine basic elements:

```
X+3.
TOTAL/A
TAN(PI*M)
(X+3.) -(TOTAL/A) * TAN (PI*M)
```

Compound numeric expressions must be constructed according to the following rules:

a. With respect to the numeric operators +, -, *, /, any type of quantity (logical, octal, integer, real, double precision, complex or literal) may be combined with any other, with one exception: a complex quantity cannot be combined with a double precision quantity.

The resultant type of the combination of any two types may be found in Table 2-1. The conversions between data types will occur as follows:

(1) A literal constant will be combined with any integer constant as an integer and with a real or double word as a real or double word quantity. (Double word refers to both double precision and complex.)

(2) An integer quantity (constant, variable, or function reference) combined with a real or double word quantity results in an expression of the type real or double word respectively; e.g., an integer variable plus a complex variable will result in a complex subexpression. The integer is converted to floating point and then added to the real part of the complex number. The imaginary part is unchanged.

(3) A real quantity (constant, variable, or function reference) combined with a double word quantity results in an expression that is of the same type as the double word quantity.

(4) A logical or octal quantity is combined with an integer, real, or double word quantity as if it were an integer quantity in the integer case, or a real quantity in the real or double word case (i.e., no conversion takes place).

b. Any numeric expression may be enclosed in parentheses and considered to be a basic element.

```
(X+Y)/2
(ZETA)
(COS(SIN(PI*M)+X))
```

Table 2-1
Types of Resultant Subexpressions

+,-,*,/		Type of Quantity				
		Real	Integer	Complex	Double Precision	Logical, Octal, or Literal
Type of Quantity	Real	Real	Real	Complex	Double Precision	Real
	Integer	Real	Integer	Complex	Double Precision	Integer
	Complex	Complex	Complex	Complex	Not Allowed	Complex
	Double Precision	Double Precision	Double Precision	Not Allowed	Double Precision	Double Precision
	Logical, Octal, or Literal	Real	Integer	Complex	Double Precision	Logical, Octal, or Literal

c. Numeric expressions which are preceded by a + or – sign are also numeric expressions:

```
+X
-(ALPHA*BETA)
-SQRT(-GAMMA)
```

d. If the precedence of numeric operations is not given explicitly by parentheses, it is understood to be the following (in order of decreasing precedence):

Operator	Explanation
**	numeric exponentiation
*and/	numeric multiplication and division
+and-	numeric addition and subtraction

In the case of operations of equal hierarchy, the calculation is performed from left to right.

e. No two numeric operators may appear in sequence. For instance:

```
X*-Y
```

is improper. Use of parentheses yields the correct form:

```
X*(-Y)
```

By use of the foregoing rules, all permissible numeric expressions may be formed. As an example of a typical numeric expression using numeric operators and a function reference, the expression for one of the roots of the general quadratic equation:

$$\frac{-b+\sqrt{b^2-4ac}}{2a}$$

would be coded as:

```
(-B+SQRT(B**2-4.*A*C))/(2.*A)
```

2.3.2 Logical Expressions

A logical expression consists of logical constants, logical variables, logical function references, and arithmetic expressions, separated by logical operators or relational operators. Logical expressions are provided in FORTRAN IV to permit the implementation of various forms of symbolic logic. Logical constants are defined by arithmetic statements, which are described in Chapter 3. Logical variables and functions are defined by the LOGICAL statement, described in Chapter 6. Binary variables may be represented by the logical constants .TRUE. and .FALSE., which must always be written with enclosing periods. Logical masks may be represented by using octal constants. The result of a logical expression has a logical value (i.e., either true or false) and therefore, only uses one word.

2.3.2.1 Logical Operators - The logical operators, which include the enclosing periods and their definitions, are as follows, where P and Q are logical expressions:

.NOT.P	Has the value .TRUE. only if P is .FALSE., and has the value .FALSE. only if P is .TRUE.
P.AND.Q	Has the value .TRUE. only if P and Q are both .TRUE., and has the value .FALSE. if either P or Q is .FALSE.
P.OR.Q	(Inclusive OR) Has the value .TRUE. if either P or Q is .TRUE., and has the value .FALSE. only if both P and Q are .FALSE.
P.XOR.Q	(Exclusive OR) Has the value .TRUE. if either P or Q but not both are .TRUE., and has the value .FALSE. otherwise.
P.EQV.Q	(Equivalence) Has the value .TRUE. if P and Q are both .TRUE. or both .FALSE., and has the value .FALSE. otherwise.

Logical operators may be used to form new variables, for example,

```
X = Y.AND.Z
E = E.XOR. "400000000000
```

2.3.2.2 Relational Operators - The relational operators are as follows:

Operator	Relation
.GT.	greater than
.GE.	greater than or equal to
.LT.	less than
.LE.	less than or equal to
.EQ.	equal to
.NE.	not equal to

The enclosing periods are part of the operator and must be present.

Mixed expressions involving integer, real, and double precision types may be combined with relationals. The value of such an expression will be .TRUE. or .FALSE..

The relational operators .EQ. and .NE. may also be used with COMPLEX expressions. (Double word quantities are equal if the corresponding parts are equal.)

A logical expression may consist of a single element (constant, variable, function reference, or relation):

```
.TRUE.
X.GE.3.14159
```

Single elements may be combined through use of logical operators to form compound logical expressions, such as:

```
TVAL.AND.INDEX
BOOL(M).OR.K.EQ.LIMIT
```

Any logical expression may be enclosed in parentheses and regarded as an element:

```
(T.XOR.S).AND.(R.EQV.Q)
CALL PARITY ((2.GT.Y.OR.X.GE.Y).AND.NEVER)
```

Any logical expression may be preceded by the unary operator .NOT. as in:

```
.NOT.T
.NOT.X+7.GT.Y+Z
BOOL(K).AND..NOT.(TVAL.OR.R)
```

No two logical operators may appear in sequence, except in the case where .NOT. appears as the second of two logical operators, as in the example above. Two decimal points may appear in sequence, as in the example above, or when one belongs to an operator and the other to a constant.

When the precedence of operators is not given explicitly by parentheses, it is understood to be as follows (in order of decreasing precedence):

```
**
*,/
+,-
.GT.,.GE.,.LT.,.LE.,.EQ.,.NE.
.NOT.
.AND.
.OR.
.EQV., .XOR.
```

For example, the logical expression

```
.NOT.ZETA**2+Y*MASS.GT.K-2.OR.PARITY.AND.X.EQ.Y
```

is interpreted as

```
(.NOT.(((ZETA**2)+(Y*MASS)).GT.(K-2))).OR.(PARITY.AND.(X.EQ.Y))
```

CHAPTER 3
THE ARITHMETIC STATEMENT

3.1 GENERAL DESCRIPTION

One of the key features of FORTRAN IV is the ease with which arithmetic computations can be coded. Computations to be performed by FORTRAN IV are indicated by arithmetic statements, which have the general form:

```
A=B
```

where A is a variable, B is an expression, and = is a replacement operator. The arithmetic statement causes the FORTRAN IV object program to evaluate the expression B and assign the resultant value to the variable A. Note that the = sign signifies replacement, not equality. Thus, expressions of the form:

```
A=A+B and
A=A*B
```

are quite meaningful and indicate that the value of the variable A is to be replaced by the result of the expression to the right of the = sign.

```
Examples:      Y=1*Y
               P=.TRUE.
               X(N)=N*ZETA(ALPHA*M/PI)+(1.,-1.)
```

Table 3-1 indicates which type of expression may be equated to each type of variable in an arithmetic statement. D indicates that the assignment is performed directly (no conversion of any sort is done); R indicates that only the real part of the variable is set to the value of the expression (the imaginary part is set to zero); C means that the expression is converted to the type of the variable; and H means that only the high-order portion of evaluated expression is assigned to the variable.

The expression value is made to agree in type with the assignment variable before replacement occurs. For example, in the statement:

```
THETA=W*(ABETA+E)
```

if THETA is an integer and the expression is real, the expression value is truncated to an integer before assignment to THETA.

Table 3-1
Allowed Assignment Statements

Variable	Expression				
	Real	Integer	Complex	Double Precision	Logical, Octal, or Literal Constant
Real	D	C	R,D	H,D	D
Integer	C	D	R,C	H,C	D
Complex	D,R,I	C,R,I	D	H,D,R,I	D,R,I
Double Precision	D,H,L	C,H,L	R,D,H,L	D	D,H,L
Logical	D	D	R,D	H,D	D

D	- Direct Replacement
C	- Conversion between integer and floating point
R	- Real only
I	- Set imaginary part to 0
H	- High order only
L	- Set low order part to 0

CHAPTER 4
CONTROL STATEMENTS

FORTRAN compiled programs normally execute statements sequentially in the order in which they were presented to the compiler. However, the following control statements are available to alter the normal sequence of statement execution: GO TO, IF, DO, PAUSE, STOP, END, CALL, RETURN. CALL and RETURN are used to enter and return from subroutines.

4.1 GO TO STATEMENT

The GO TO statement has three forms: unconditional, computed, and assigned.

4.1.1 Unconditional GO TO Statements

Unconditional GO TO statements are of the form:

GO TO n

where n is the number of an executable statement. Control is transferred to the statement numbered n. An unconditional GO TO statement may appear anywhere in the source program, except as the terminal statement of a DO loop.

4.1.2 Computed GO TO Statements

Computed GO TO statements have the form:

GO TO $(n_1, n_2, \ldots, n_k)$, i

where $n_1, n_2, \ldots, n_k$ are statement numbers, and i is an integer expression.

This statement transfers control to the statement numbered $n_1, n_2, \ldots, n_k$ if i has the value 1, 2, ..., k, respectively. If i exceeds the size of the list of statement numbers or is less than one, execution will proceed to the next executable statement. Any number of statement numbers may appear in the list. There is no restriction on other uses for the integer variable i in the program.

In the example

```
        GO TO (20,10,5),K
```

the variable K acts as a switch, causing a transfer to statement 20 if K=1, to statement 10 if K=2, or to statement 5 if K=3.

A computed GO TO statement may appear anywhere in the source program, except as the terminal statement of a DO loop.

4.1.3 Assigned GO TO Statement

Assigned GO TO statements have two equivalent forms:

GO TO k

and

GO TO k, $(n_1, n_2, n_3, \ldots)$

where k is a nonsubscripted integer variable and $n_1, n_2, \ldots n_k$ are statement numbers. Any number of statement numbers may appear in the list. Both forms of the assigned GO TO have the effect of transferring control to the statement whose number is currently associated with the variable k. This association is established through the use of the ASSIGN statement, the general form of which is:

ASSIGN i TO k

If more than one ASSIGN statement refers to the same integer variable name, the value assigned by the last executed statement is the current value.

```
Examples:      ASSIGN 21 TO INT              ASSIGN 1000 TO INT
                      .                              .
                      .                              .
                      .                              .
                  GO TO INT              GO TO INT, (2,21,1000,310)
```

An assigned GO TO statement may appear anywhere in the source program, except as the terminal statement of a DO loop.

4.2 IF STATEMENT

IF statements have two forms in FORTRAN IV: numerical and logical.

4.2.1 Numerical IF Statements

Numerical IF statements are of the form:

IF (expression) n_1, n_2, n_3

where n_1, n_2, n_3 are statement numbers. This statement transfers control to the statement numbered n_1, n_2, n_3 if the value of the numeric expression is less than, equal to, or greater than zero, respectively. All three statement numbers must be present. The expression may not be complex.

Examples:

```
IF (ETA) 4,7,12
IF (KAPPA-L (10)) 20,14,14
```

4.2.2 Logical IF Statements

Logical IF statements have the form:

IF (expression)S

where S is a complete statement. The expression must be logical. S may be any executable statement other than a DO statement or another logical IF statement (see Chapter 2, Section 2.3.2). If the value of the expression is .FALSE., control passes to the next sequential statement. If value of the expression is .TRUE., statement S is executed. After execution of S, control passes to the next sequential statement unless S is a numerical IF statement or a GO TO statement; in these cases, control is transferred as indicated. If the expression is .TRUE. and S is a CALL statement, control is transferred to the next sequential statement upon return from the subroutine.

Numbers are present in the logical expression:

```
IF (B)Y=X*SIN(Z)
W=Y**2
```

If the value of B is .TRUE., the statements Y=X*SIN(Z) and W=Y**2 are executed in that order. If the value of B is .FALSE., the statement Y=X*SIN(Z) is not executed.

Examples:

```
IF (T.OR.S)X=Y+1
IF (Z.GT.X(K)) CALL SWITCH (S,Y)
IF (K.EQ.INDEX) GO TO 15
```

NOTE

Care should be taken in testing floating point numbers for equality in IF statements as rounding may cause unexpected results.

4.3 DO STATEMENT

The DO statement simplifies the coding of iterative procedures. DO statements are of the form:

DO n i=m_1,m_2,m_3

where n is a statement number, i is a nonsubscripted integer variable, and m_1,m_2,m_3 are any integer expressions. If m_3 is not specified, it is understood to be 1.

The DO statement causes the statements which follow, up to and including the statement numbered n, to be executed repeatedly. This group of statements is called the range of the DO statement. The integer variable i of the DO statement is called the index. The values of m_1,m_2, and m_3 are called, respectively, the initial, limit, and increment values of the index.

A zero increment (m_3) is not allowed. The increment m_3 may be negative if $m_1 \geq m_2$. If $m_1 \leq m_2$, the increment m_3 must be positive. The index variable can assume legal values only if $(m_2-m_i)*m_3 \geq 0$. (m_i is the current value of the index variable m_1.)

Examples:

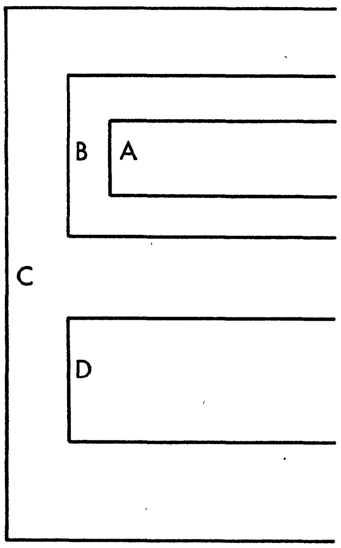
Form	Restriction
DO 10 I=1,5,2	
DO 10 I=5,1,-1	
DO 10 I=J,K,5	J≤K
DO 10 I=J,K,-5	J≥K
DO 10 L=I,J,-K	I≤J,K<0 or I≥J,K>0
DO 10 L=I,J,K	I≤J,K>0 or I≥J,K>0

Initially, the statements of the range are executed with the initial value assigned to the index. This initial execution is always performed, regardless of the values of the limit and increment. After each execution of the range, the increment value is added to the value of the index and the result is compared with the limit value. If the value of the index is not greater than the limit value, the range is executed again using the new value of the index. When the increment value is negative, another execution will be performed if the new value of the index is not less than the limit value.

After the last execution of the range, control passes to the statement immediately following the range. This exit from the range is called the normal exit. Exit may also be accomplished by a transfer from within the range.

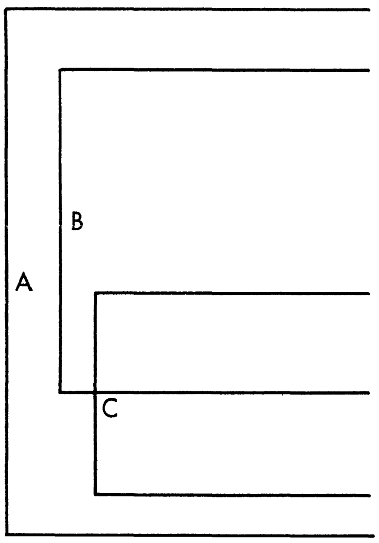
The range of a DO statement may include other DO statements, provided that the range of each contained DO statement is entirely within the range of the containing DO statement. That is, the ranges of two DO statements must intersect completely or not at all. A transfer into the range of a DO statement from outside the range is not allowed.

Valid DO Loop Nest



Control must not pass from within loop A or loop B into loop D, or from loop D into loop A or loop B.

Invalid DO Loop Nest



Loop C is not fully within the range of loop B even though it is within the range of loop A.

Figure 4-1 Nested DO Loops

Within the range of a DO statement, the index is available for use as an ordinary variable. After a transfer from within the range, the index retains its current value and is available for use as a variable. The value of the index variable becomes undefined when the DO loop it controls is satisfied. The values of the initial, limit, and increment variables for the index and the index of the DO loop, may not be altered within the range of the DO statement.

The range of a DO statement must not end with a GO TO type statement or a numerical IF statement. If an assigned GO TO statement is in the range of a DO loop, all the statements to which it may transfer must be either in the range of the DO loop or all must be outside the range. A logical IF statement is allowed as the last statement of the range. In this case, control is transferred as follows. The range is considered ended when, and if, control would normally pass to the statement following the entire logical IF statement.

As an example, consider the sequences:

```
  DO 5 K = 1,4
5 IF(X(K).GT.Y(K))Y(K) = X(K)
6 ...
```

Statement 5 is executed four times whether the statement Y(K) = X(K) is executed or not. Statement 6 is not executed until statement 5 has been executed four times.

```
Examples:     DO 22 L = 1,30
              DO 45 K = 2,LIMIT,-3
              DO 7 X = T,MAX,L
```

4.4 CONTINUE STATEMENT

The CONTINUE statement has the form:

```
        CONTINUE
```

This statement is a dummy statement, used primarily as a target for transfers, particularly as the last statement in the range of a DO statement. For example, in the sequence:

```
        DO 7 K = START,END
           .
           .
           .
        IF (X(K))22,13,7
           .
           .
   7    CONTINUE
```

a positive value of X(K) begins another execution of the range. The CONTINUE provides a target address for the IF statement and ends the range of the DO statement.

4.5 PAUSE STATEMENT

The PAUSE statement enables the program to incorporate operator activity into the sequence of automatic events. The PAUSE statement assumes one of three forms:

```
        PAUSE
        PAUSE n
        PAUSE 'xxxxx'
```

where n is an unsigned string of six or less octal digits, and 'xxxxx' is a literal message.

Execution of the PAUSE statement causes the message or the octal digits, if any, to be typed on the user's teletypewriter. Program execution may be resumed (at the next executable FORTRAN statement) from the console by typing "G," followed by a carriage return. Program execution may be terminated by typing "X," followed by carriage return.

```
Example:      PAUSE 167
              PAUSE 'NOW IS THE TIME'
```

4.6 STOP STATEMENT

The STOP statement has the forms:

```
STOP      or
STOP n
```

where n is an unsigned string of one to six octal digits.

The STOP statement terminates the program and returns control to the monitor system. (Termination of a program may also be accomplished by a CALL to the EXIT or DUMP subroutines.)

4.7 END STATEMENT

The END statement has the form:

```
END
```

The END statement informs the compiler to terminate compilation and must be the physically last statement of the program.

CHAPTER 5
DATA TRANSMISSION STATEMENTS

Data transmission statements are used to control the transfer of data between computer memory and either peripheral devices or other locations in computer memory. These statements are also used to specify the format of the output data. Data transmission statements are divided into the following four categories.

a. Nonexecutable statements that enable conversions between internal form data within core memory and external form data (FORMAT), or specify lists of arrays and variables for input/output transfer (NAMELIST).

b. Statements that specify transmission of data between computer memory and I/O devices: READ, WRITE, PRINT, PUNCH, TYPE, ACCEPT.

c. Statements that control magnetic tape unit mechanisms: REWIND, BACKSPACE, END FILE, UNLOAD, SKIP RECORD.

d. Statements that specify transmission of data between series of locations in memory: ENCODE, DECODE.

5.1 NONEXECUTABLE STATEMENTS

The FORMAT statement enables the user to specify the form and arrangement of data on the selected external medium. The NAMELIST statement provides for conversion and input/output transmission of data without reference to a FORMAT statement.

5.1.1 FORMAT Statement

FORMAT statements may be used with any appropriate input/output medium or ENCODE/DECODE statement. FORMAT statements are of the form:

$$n \text{ FORMAT } (S_1, S_2, \ldots S_n / S_1^1, S_2^1, \ldots, S_n^1 / \ldots)$$

where n is a statement number, and each S is a data field specification.

FORMAT statements may be placed anywhere in the source program. Unless the FORMAT statement contains only alphanumeric data for direct input/output transmission, it will be used in conjunction with the list of a data transmission statement.

Slashes are used to specify unit records, which must be one of the following:

a. A tape or disk record with a maximum length corresponding to a line buffer (135 ASCII characters).

b. A punched card with a maximum of 80 characters.

c. A printed line with a maximum of 72 characters for a Teletype® and either 120 or 132 characters for the line printer.

During transmission of data, the object program scans the designated FORMAT statement. If a specification for a numeric field is present (see Section 5.2.1 of this chapter) and the data transmission statement contains items remaining to be transmitted, transmission takes place according to the specifications. This process ceases and execution of the data transmission statement is terminated as soon as all specified items have been transmitted. Thus, the FORMAT statement may contain specifications for more items than are specified by the data transmission statement. Conversely, the FORMAT statement may contain specifications for fewer items than are specified by the data transmission statement.

The following types of field specifications may appear in a FORMAT statement: numeric, numeric with scale factors, logical, alphanumeric. The FORMAT statement also provides for handling multiple record formats, formats stored as data, carriage control, skipping characters, blank insertion, and repetition. If an input list requires more characters than the input device supplies for a given unit record, blanks are supplied.

5.1.1.1 Numeric Fields - Numeric field specification codes designate the type of conversion to be performed. These codes and the corresponding internal and external forms of the numbers are listed in Table 5-2.

The conversions are specified by the forms:

1.	Dw.d	
2.	Ew.d	
3.	Fw.d	
4.	Iw	
5.	Ow	
6.	Gw.d	(for real or double precision)
	Gw	(for integer or logical)
	Gw.d,Gw.d	(for complex)

respectively. The letter D, E, F, I, O, or G designates the conversion type; w is an integer specifying the field width, which may be greater than required to provide for blank columns between numbers; d is an integer specifying the number of decimal places to the right of the decimal point or, for G conversion, the number of significant digits. (For D, E, F, and G input, the position of the decimal point in the external field takes precedence over the value of d in the format.)

® Teletype is a registered trademark of Teletype Corporation.

For example,

```
FORMAT (I5,F10.2,D18.10)
```

could be used to output the line,

```
bbb32bbbb-17.60bbb.5962547681D+03
```

on the output listing.

The G format is the general format code that is used to transmit real, double precision, integer, logical, or complex data. The rules for input depend on the type specification of the corresponding variable in the data list. The form of the output conversion also depends on the individual variable except in the case of real and double-precision data. In these cases the form of the output conversion is a function of the magnitude of the data being converted. The following table shows the magnitude of the external data, M, and the resulting method of conversion.

Table 5-1
Magnitude of Internal Data

Magnitude of Data	Resulting Conversion
$0.1 \leq M < 1$	F(w-4).d, 4x
$1 \leq M < 10$	F(w-4).(d-1), 4x
.	.
.	.
.	.
$10^{d-2} \leq M < 10^{d-1}$	F(w-4). 1, 4x
$10^{d-1} \leq M < 10^{d}$	F(w-4). 0, 4x
All others	Ew.d

The field width w should always be large enough to include spaces for the decimal point, sign, and exponent. In all numeric field conversions if w is not large enough to accommodate the converted number, the excess digits on the left will be lost; if the number is less than w spaces in length, the number is right-adjusted in the field.

Table 5-2
Numeric Field Codes

Conversion Code	Internal Form	External Form
D	Binary floating point double-precision	Decimal floating point with D exponent
E	Binary floating point	Decimal floating point with E exponent
F	Binary floating point	Decimal fixed point
I	Binary integer	Decimal integer
O	Binary integer	Octal integer
G	One of the following: single precision binary floating point, binary integer, binary logical, or binary complex	Single precision decimal floating point integer, logical (T or F), or complex (two decimal floating point numbers), depending upon the internal form

5.1.1.2 Numeric Fields with Scale Factors - Scale factors may be specified for D, E, F, and G conversions. A scale factor is written nP where P is the identifying character and n is a signed or unsigned integer that specifies the scale factor.

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

$$\text{external number} = (\text{internal number})*\ 10^{(\text{scale factor})}$$

For D, E, and G (external field not decimal fixed point) conversions, 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)
```

corresponds to the line

```
bb26.451bbbb-0.41321E-01
```

then the statement

```
FORMAT (-1PF8.3,2PE16.5)
```

might correspond to the line

```
bbb2.645bbb-41.32157E-03
```

In input operations, F type (and G type, if the external field is decimal fixed point) conversions are the only types affected by scale factors.

When no scale factor is specified, it is understood to be zero. However, once a scale factor is specified, it holds for all subsequent D, E, F, and G type conversions within the same format unless another scale factor is encountered. The scale factor is reset to zero by specifying a scale factor of zero. Scale factors have no effect on I and O type conversions.

5.1.1.3 Logical Fields - Logical data can be transmitted in a manner similar to numeric data by use of the specification:

```
Lw
```

where L is the control character and w is an integer specifying the field width. The data is transmitted as the value of a logical variable in the input/output list.

If on input, 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 will be stored.

On output, w minus 1 blanks followed by T or F will be output if the value of the logical variable is true or false, respectively.

5.1.1.4 Variable Field Width - The D, E, F, G, I, and O conversion types may appear in a FORMAT statement without the specification of the field width (w) or the number of places after the decimal point (d). In the case of input, omitting the w implies that the numeric field is delimited by any character which would otherwise be illegal in the field, in addition to the characters -, +, ., E, D, and blank provided they follow the numeric field. For example, input according to the format

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

might appear on the input medium as

```
-10,3/15.621-.0016E-10,777.
```

In this case, commas delimit the numeric fields, blanks may also be used as field delimiters. On output, omitting the w has the following effect:

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

5.1.1.5 Alphanumeric Fields - Alphanumeric data can be transmitted in a manner similar to numeric data by use of the form Aw, where A is the control character and w is the number of characters in the field. The alphanumeric characters are transmitted as the value of a variable in an input/output list. The variable may be of any type. For the sequence:

```
    READ 5,V
  5 FORMAT (A4)
```

causes four characters to be read and placed in memory as the value of the variable V.

Although w may have any value, the number of characters transmitted is limited by the maximum number of characters which can be stored in the space allotted for the variable. This maximum depends upon the variable type. For a double precision variable the maximum is ten characters; for all other variables, the maximum is five characters. If w exceeds the maximum, the leftmost characters are lost on input and replaced with blanks on output. If, on input, w is less than the maximum, blanks are filled in to the right of the given characters until the maximum is reached. If, on output, w is less than the maximum, the leftmost w characters are transmitted to the external medium. Since for complex variables each word requires a separate field specification, the maximum value for w is 5. For example,

```
    COMPLEX C
    ACCEPT 1, C
  1 FORMAT (2A5)
```

could be used to transmit ten alphanumeric characters into complex variable C.

5.1.1.6 Alphanumeric Data Within Format Statements - Alphanumeric data may be transmitted directly into or from the format statement by two different methods: H-conversion, or the use of single quotes.

In H-conversion, the alphanumeric string is specified by the form nH. H is the control character and n is the number of characters in the string counting blanks. For example, the format in the statement below can be used to print PROGRAM COMPLETE on the output listing.

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

The statement

```
FORMAT (16HPROGRAM COMPLETE)
```

causes ROGRAM COMPLETE to be printed.

Referring to this format in a READ statement would cause the 17 characters to be replaced with a new string of characters.

The same effect is achieved by merely enclosing the alphanumeric data in quotes. The result is the same as in H-conversion; on input, the characters between the quotes are replaced by input characters, and, on output, the characters between the quotes (including blanks) are written as part of the output data. A quote character within the data is represented by two successive quote marks. For example, referring to:

```
FORMAT (' DON''T')
```

with an output statement would cause DON'T to be printed. Referring to

```
FORMAT ('DON''T')
```

causes ON'T to be printed. The first character referenced by the FORMAT statement for output is interpreted as a carriage control character (see 5.1.1.13).

5.1.1.7 Mixed Fields - An alphanumeric format field may be placed among other fields of the format. For example, the statement:

```
FORMAT (I5,7H FORCE=F10.5)
```

can be used to output the line:

```
bbb22bFORCE=bb17.68901
```

The separating comma may be omitted after an alphanumeric format field, as shown above.

5.1.1.8 Complex Fields - Complex quantities are transmitted as two independent real quantities. The format specification consists of two successive real specifications or one repeated real specification. For instance, the statement:

```
FORMAT (2E15.4,2(F8.3,F8.5))
```

could be used in the transmission of three complex quantities.

5.1.1.9 Repetition of Field Specifications - Repetition of a field specification may be specified by preceding the control character D, E, F, I, O, G, L, or A by an unsigned integer giving the number of repetitions desired. For example:

```
FORMAT (2E12.4,3I5)
```

is equivalent to:

```
FORMAT (E12.4,E12.4,I5,I5,I5)
```

5.1.1.10 Repetition of Groups - A group of field specifications may be repeated by enclosing the group in parentheses and preceding the whole with the repetition number. For example:

```
FORMAT (2I8,2(E15.5,2F8.3))
```

is equivalent to:

```
FORMAT (2I8,E15.5,2F8.3,E15.5,2F8.3)
```

5.1.1.11 Multiple Record Formats - To handle a group of input/output records where different records have different field specifications, a slash is used to indicate a new record. For example, the statement:

```
FORMAT (3O8/I5,2F8.4)
```

is equivalent to

```
FORMAT (3O8)
```

for the first record and

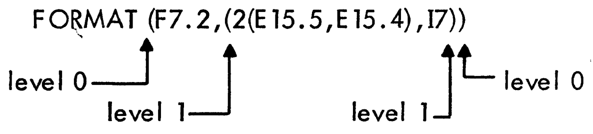
```
FORMAT (I5,2F8.4)
```

for the second record.

The separating comma may be omitted when a slash is used. When n slashes appear at the end or beginning of a format, n blank records may be written on output or records skipped on input. When n slashes appear in the middle of a format, n-1 blank records are written or n-1 records skipped.

Both the slash and the closing parenthesis at the end of the format indicate the termination of a record. If the list of an input/output statement dictates that transmission of data is to continue after the closing parenthesis of the format is reached, the format is repeated starting with that group repeat specification terminated by the last right parenthesis of level one or level zero if no level one group exists.

Thus, the statement



causes the format

```
F7.2,2(E15.5,E15.4),I7
```

to be used on the first record, and the format

```
2(E15.5,E15.4),I7
```

to be used on succeeding records.

As a further example, consider the statement

```
FORMAT (F7.2/(2(E15.5,E15.4),I7))
```

The first record has the format

```
F7.2
```

and successive records have the format

```
2(E15.5,E15.4),I7
```

5.1.1.12 Formats Stored as Data - The ASCII character string comprising a format specification may be stored as the values of an array. Input/output statements may refer to the format by giving the array name, rather than the statement number of a FORMAT statement. The stored format has the same form as a FORMAT statement excluding the word "FORMAT." The enclosing parentheses are included.

As an example, consider the sequence:

```
      DIMENSION SKELETON (2)
      READ 1, (SKELETON(I), I = 1,2)
    1 FORMAT (2A4)
      READ SKELETON,K,X
```

The first READ statement enters the ASCII string into the array SKELETON. In the second READ statement, SKELETON is referred to as the format governing conversion of K and X.

5.1.1.13 Carriage Control - The first character of each ASCII record controls the spacing of the line printer or Teletype. This character is usually set by beginning a FORMAT statement for an ASCII record with 1Ha, where a is the desired control character. The line spacing actions, listed below, occur before printing:

Character		Effect
	space	skip to next line with a FORM FEED after every 60 lines
0	zero	skip a line
1	one	form feed - go to top of next page
+	plus	suppress skipping - will overprint line
*	asterisk	skip to next line with no FORM FEEDS
-	minus	skip 2 lines
2	two	skip to next 1/2 of page
3	three	skip to next 1/3 of page
/	slash	skip to next 1/6 of page
.	period	skip to next 1/20 of page
,	comma	skip to next 1/30 page

A $ (dollar sign) as a format field specification code suppresses the carriage return at the end of the line.

5.1.1.14 Spacing - Input and output can be made to begin at any position within a FORTRAN record by use of the format code

Tw

where T is the control character and w is an unsigned integer constant specifying the character position in a FORTRAN record where the transfer of data is to begin. When the output is printed, w corresponds to the (w-1)th print position. This is because the first character of the output buffer is a carriage control character and is not printed. It is recommended that the first field specification of the output format be 1x, except where a carriage control character is used.

For example,

```
2 FORMAT (T50, 'BLACK'T30, 'WHITE')
```

would cause the following line to be printed

```
Print Position 29        Print Position 49
↓                        ↓
   WHITE                    BLACK
```

For input, the statement

```
1 FORMAT(T35,'MONTH')
  READ (3,1)
```

cause the first 34 characters of the input data to be skipped, and the next 5 characters would replace the characters M, O, N, T, and H in storage. If an input record containing

```
ABCbbbXYZ
```

is read with the format specification

```
10 FORMAT (T7,A3,T1,A3)
```

then the characters XYZ and ABC are read, in that order.

5.1.1.15 Blank or Skip Fields - Blanks may be introduced into an output record or characters skipped on an input record by use of the specification nX. The control character is X; n is the number of blanks or characters skipped and must be greater than zero. For example, the statement

```
FORMAT (5Hˬ STEPI5, 10X2HY=F7.3)
```

may be used to output the line

```
bSTEPbbb28bbbbbbbbbbY=b-3.872
```

5.1.2 NAMELIST Statement

The NAMELIST statement, when used in conjunction with special forms of the READ and WRITE statements, provides a method for transmitting and converting data without using a FORMAT statement or an I/O list. The NAMELIST statement has the form

$$\text{NAMELIST}/X_1/A_1,A_2,\ldots,A_i/X_2/B_1,B_2,\ldots,B_i\ldots/X_m/C_1,C_2,\ldots C_n$$

where the X's are NAMELIST names, and the A's, B's, and C's are variable or array names.

Each list or variable mentioned in the NAMELIST statement is given the NAMELIST name immediately preceding the list. Thereafter, an I/O statement may refer to an entire list by mentioning its NAMELIST name. For example:

```
NAMELIST/FRED/A,B,C/MARTHA/D,E
```

states that A, B, and C belong to the NAMELIST name FRED, and D and E belong to MARTHA.

The use of NAMELIST statements must obey the following rules:

a. A NAMELIST name may not be longer than six characters; it must start with an alphabetic character; it must be enclosed in slashes; it must precede the list of entries to which it refers; and it must be unique within the program.

b. A NAMELIST name may be defined only once and must be defined by a NAMELIST statement. After a NAMELIST name has been defined, it may only appear in READ or WRITE statements. The NAMELIST name must be defined in advance of the READ or WRITE statement.

c. A variable used in a NAMELIST statement cannot be used as a dummy argument in a subroutine definition.

d. Any dimensioned variable contained in NAMELIST statement must have been defined in a DIMENSION statement preceding the NAMELIST statement.

5.1.2.1 Input Data For NAMELIST Statements - When a READ statement refers to a NAMELIST name, the first character of all input records is ignored. Records are searched until one is found with a $ or & as the second character immediately followed by the NAMELIST name specified. Data is then converted and placed in memory until the end of a data group is signaled by a $ or & either in the same record as the NAMELIST name, or in any succeeding record as long as the $ or & is the second character of the record. Data items must be separated by commas and be of the following form:

$$V=K_1,K_2,\ldots,K_n$$

where V may be a variable name or an array name, with or without subscripts. The K's are constants which may be integer, real, double precision, complex (written as (A, B) where A and B are real), or logical (written as T for true and F for false). A series of J identical constants may be represented by J*K where J is an unsigned integer and K is the repeated constant. Logical and complex constants must be equated to logical and complex variables, respectively. The other types of constants (real, double precision, and integers) may be equated to any other type of variable (except logical or complex), and will be converted to the variable type. For example, assume A is a two-dimensional real array, B is a one-dimensional integer array, C is an integer variable, and that the input data is as follows:

```
 $FRED A(7,2)=4, B=3,6*2.8, C=3.32$
 ↑
 Column 2
```

A READ statement referring to the NAMELIST name FRED will result in the following: the integer 4 will be converted to floating point and placed in A(7,2). The integer 3 will be placed in B(1) and the floating point number 2.8 will be placed in B(2), B(3),..., B(7). The floating point number 3.32 will be converted to the integer 3 and placed in C.

5.1.2.2 Output Data For NAMELIST Statements - When a WRITE statement refers to a NAMELIST name, all variables and arrays and their values belonging to the NAMELIST name will be written out, each according to its type. The complete array is written out by columns. The output data will be written so that:

a. The fields for the data will be large enough to contain all the significant digits.

b. The output can be read by an input statement referencing the NAMELIST name.

For example, if JOE is a 2x3 array, the statement

```
NAMELIST/NAM1/JOE,K1,ALPHA
WRITE (u,NAM1)
```

generate the following form of output.

```
 Column 2
 ↓
 $NAM1
 JOE = -6.75,        .234E-04,         68.0,
       -17.8,             0.0,    -.197E+07,
  K1 = 73.1,        ALPHA=3,$
```

5.2 DATA TRANSMISSION STATEMENTS

The data transmission statements accomplish input/output transfer of data that may be listed in a NAMELIST statement or defined in a FORMAT statement. When a FORMAT statement is used to specify formats, the data transmission statement must contain a list of the quantities to be transmitted. The data appears on the external media in the form of records.

5.2.1 Input/Output Lists

The list of an input/output statement specifies the order of transmission of the variable values. During input, the new values of listed variables may be used in subscript or control expressions for variables appearing later in the list. For example:

```
READ 13,L,A(L),B(L+1)
```

reads a new value of L and uses this value in the subscripts of A and B.

The transmission of array variables may be controlled by indexing similar to that used in the DO statement. The list of controlled variables, followed by the index control, is enclosed in parentheses. For example,

```
READ 7, (X(K),K=1,4),A
```

is equivalent to:

```
READ 7, X(1),X(2),X(3),X(4),A
```

As in the DO statement, the initial, limit, and increment values may be given as integer expressions:

```
READ 5, N, (GAIN(K),K=1,M/2,N)
```

The indexing may be compounded as in the following:

```
READ 11, ((MASS(K,L),K=1,4),L=1,5)
```

The above statement reads in the elements of array MASS in the following order:

```
MASS(1,1), MASS(2,1),...,MASS(4,1),MASS(1,2),...,MASS(4,5)
```

If an entire array is to be transmitted, the indexing may be omitted and only the array identifier written. The array is transmitted in order of increasing subscripts with the first subscript varying most rapidly. Thus, the example above could have been written:

```
READ 11, MASS
```

Entire arrays may also be designated for transmission by referring to a NAMELIST name (see description of NAMELIST statement).

5.2.2 Input/Output Records

All information appearing on external media is grouped into records. The maximum amount of information in one record and the manner of separation between records depends upon the medium. For punched cards, each card constitutes one record; on a teletypewriter a record is one line, and so forth. The amount of information contained in each ASCII record is specified by the FORMAT reference and the I/O list. For magnetic tape binary records, the amount of information is specified by the I/O list.

Each execution of an input or output statement initiates the transmission of a new data record. Thus, the statement

```
READ 2, FIRST,SECOND,THIRD
```

is not necessarily equivalent to the statements

```
READ 2, FIRST
READ 2, SECOND
READ 2, THIRD
```

since, in the second case, at least three separate records are required, whereas, the single statement

```
READ 2, FIRST,SECOND,THIRD
```

may require one, two, three, or more records depending upon FORMAT statement 2.

If an input/output statement requests less than a full record of information, the unrequested part of the record is lost and cannot be recovered by another input/output statement without repositioning the record.

If an input/output list requires more than one ASCII record of information, successive records are read.

5.2.3 PRINT Statement

The PRINT statement assumes one of two forms

```
PRINT f, list
PRINT f
```

where f is a format reference.

The data is converted from internal to external form according to the designated format. If the data to be transmitted is contained in the specified FORMAT statement, the second form of the statement is used.

```
Examples:        PRINT 16,T,(B(K),K=1,M)
                 PRINT F106,SPEED,MISS
```

In the second example, the format is stored in array F106.

5.2.4 PUNCH Statement

The PUNCH statement assumes one of two forms

```
PUNCH f, list
PUNCH f
```

where f is a format reference.

Conversion from internal to external data forms is specified by the format reference. If the data to be transmitted is contained in the designated FORMAT statement, the second form of the statement is used.

```
Examples:        PUNCH 12,A,B(A),C(B(A))
                 PUNCH 7
```

5.2.5 TYPE Statement

The TYPE statement assumes one of two forms

```
TYPE f, list
TYPE f
```

where f is a format reference.

This statement causes the values of the variables in the list to be read from memory and listed on the user's teletypewriter. The data is converted from internal to external form according to the designated format. If the data to be transmitted is contained in the designated FORMAT statement, the second form of the statement is used.

```
Examples:        TYPE 14,K,(A(L),L=1,K)
                 TYPE FMT
```

5.2.6 WRITE Statement

The WRITE statement assumes one of the following forms

```
WRITE (u,f) list
WRITE(u,f)
WRITE(u,N)
WRITE(u) list
WRITE(u#R,f) list
```

where u is a unit designation, f is a format reference, N is a NAMELIST name, and R is a record number where I/O is to start.

The first form of the WRITE statement causes the values of the variables in the list to be read from memory and written on the unit designated in ASCII form. The data is converted to external form as specified by the designated FORMAT statement.

The second form of the WRITE statement causes information to be read directly from the specified format and written on the unit designated in ASCII form.

The third form of the WRITE statement causes the names and values of all variables and arrays belonging to the NAMELIST name, N, to be read from memory and written on the unit designated. The data is converted to external form according to the type of each variable and array.

The fourth form of the WRITE statement causes the values of the variables in the list to be read from memory and written on the unit designated in binary form.

The fifth form of the WRITE statement allows the FORTRAN programmer to access fixed-length records in a disk file directly. This eliminates sequential writing of data in order to access one or more records within a file. The file must be defined properly and the record from which the writing is desired must be specified. The file whose records are to be accessed is defined as follows.

```
CALL DEFINE FILE ( D, S, V, F, Pj, Pg)
```

D	=	data set (device)
S	=	size of records in the file in characters (ASCII) or words (binary)
V	=	associated variable which initially contains the length of the file that would be accessed next if the program were to continue I/O sequentially
*F	=	file name.ext defined first in DATA statement
*Pj	=	project number
*Pg	=	programmer number

Output begins when the random WRITE is specified in the correct format. The arguments designated by an asterisk (*) may be zero. This implies a default filename and/or user's project and programmer numbers; for example,

```
CALL DEFINE FILE (3,80,NX,0,0,0)
```

5.2.7 READ Statement

The READ statement assumes one of the following forms

```
READ f, list
READ f
READ(u,f) list
READ(u,f)
READ(u,N)
READ(u)list
READ(u#R,f) list
READ(u,f,END=C, ERR=d) list
READ(u,f,END=C) list
READ(u,f, ERR=d) list
```

where f is a format reference, u is a unit designation, N is a NAMELIST name, R is a record number where I/O is to start, C is a statement number to which control is transferred upon encountering an end-of-file, and d is the statement number to which control is transferred upon encountering an error condition on the input data.

The first form of the READ statement causes information to be read from cards and put in memory as values of the variables in the list. The data is converted from external to internal form as specified by the referenced FORMAT statement.

Example: READ 28,Z1,Z2,Z3

The second form of the READ statement is used if the data read from cards is to be transmitted directly into the specified format.

Example: READ 10

The third form of the READ statement causes ASCII information to be read from the unit designated and stored in memory as values of the variables in the list. The data is converted to internal form as specified by the referenced FORMAT statement.

Example: READ(1,15)ETA,P1

The fourth form of the READ statement causes ASCII information to be read from the unit designated and transmitted directly into the specified format.

Example: READ(N,105)

The fifth form of the READ statement causes data of the form described in the discussion of input data for NAMELIST statements to be read from the unit designated and stored in memory as values of the variables or arrays specified.

Example: READ(2,FRED)

The sixth form of the READ statement causes binary information to be read from the unit designated and stored in memory as values of the variables in the list.

Example: READ(M)GAIN,Z,AI

The seventh form of the READ statement allows random access of fixed-length records in a disk file. The file whose records are to be read is defined by the DEFINE FILE call where the arguments are the same as described in Section 5.2.5 of this chapter.

Example:
```
DOUBLE PRECISION FIL
DIMENSION A(6)
DATA FIL/'FILE.ONE'/
CALL DEFINE FILE (4,30,NV,FIL,"11,"23)
READ (4#54,5)A
```

This example reads the 54th record from FILE.ONE on the disk area belonging to programmer [11,23] into the list variables A(1) through A(6).

The eighth form of the READ statement causes control to be transferred if an end-of-file or error condition is encountered on the input data. The arguments END=c and ERR=d are optional and may or may not be included. If an end-of-file is encountered, control transfers to the statement specified by END=c. If an END parameter is not specified, I/O on that device terminates and the program halts with an error message to the user's TTY. If an error on input is encountered, control transfers to the statement specified by ERR=d. If an ERR=d parameter is not specified, the program halts with an error message to the user's TTY.

Example:
```
        READ (7,7,END=888, ERR=999) A
        .
        .
        .
    888 (control transfers here if an end-of-file is encountered)
        .
        .
        .
    999 (control transfers here if an error on input is encountered)
```

5.2.8 REREAD Statement

The reread feature allows a FORTRAN program to reread information from the last used input file. The format used during the reread need not correspond to the original read format, and the information may be read as many times as desired.

a. To reread from an input device, the following coding would be used:

```
READ (16,100)A
    :
    :
REREAD 105,A
```

The REREAD 105,A statement causes the last input device used to be reread according to format statement 105. The original read format and a subsequent reread format need not be the same.

b. The reread feature cannot be used until an input from a file has been accomplished. If the feature is used prematurely, an error message will be generated.

c. Information may be reread as many times as desired using either the same or a new format statement each time.

d. The reread feature must be used with some forethought and care since it rereads from the last input file used, i.e.:

The following example will reread from the file on Device No. 10, not Device No. 16:

```
READ (16,100)A
    :
    :
READ (10,200)B
    :
    :
REREAD 110,A
```

5.2.9 ACCEPT Statement

The ACCEPT statement assumes one of two forms

```
ACCEPT f, list
ACCEPT f
```

where f is a format reference.

This statement causes information to be input from the user's teletypewriter and put in memory as values of the variables in the list. The data is converted to internal form as specified by the format. If the transmission of data is directly into the designated format, the second form of the statement is used.

```
Examples:     ACCEPT 12,ALPHA,BETA
              ACCEPT 27
```

5.3 DEVICE CONTROL STATEMENTS

Device control statements and their corresponding effects are listed in Table 5-3.

Table 5-3
Device Control Statements

Statement	Effect
BACKSPACE u	Backspaces designated tape one ASCII record or one logical binary record.
END FILE u	Writes an end-of-file.
REWIND u	Rewinds tape on designated unit.
SKIP RECORD u	Causes skipping of one ASCII record or one logical binary record.
UNLOAD u	Rewinds and unloads the designated tape.

5.4 ENCODE AND DECODE STATEMENTS

ENCODE and DECODE statements transfer data, according to format specifications, from one section of user's core to another. No peripheral equipment is involved. DECODE is used to change data in ASCII format to data in another format. ENCODE changes data of another format into data in ASCII format.

The two statements are of the form

```
ENCODE(c,f,v),L(1),...,L(N)
DECODE(c,f,v),L(1),...,L(N)
```

where

c = the number of ASCII characters
f = the format statement number
v = the starting address of the ASCII record referenced
L(1),...,L(N) = the list of variables.

Example: Assume the contents of the variables to be as follows:

A(1)	contains the floating-point binary number 300.45
A(2)	contains the floating-point binary number 3.0
J	contains the binary integer value 1.
B	is a four-word array of indeterminate contents
C	contains the ASCII string 12345

```
        DO 2 J = 1,2
        ENCODE (16,10,B) J, A(J)
 10     FORMAT (IX,2HA(,I1,4H) = ,F8.2)
        TYPE 11,B
 11     FORMAT (4A5)
  2     CONTINUE
        DECODE (4, 12, C) B
```

```
12      FORMAT (3F1.0,IX,F1.0)
        TYPE 13,B
13      FORMAT (4F5.2)
        END
```

Array B can contain 20 ASCII characters. The result of the ENCODE statement after the first iteration of the DO loop is:

B(1)	A(1)
B(2)	=
B(3)	300.4
B(4)	5

Typed as

A(1) = 300.45

The result after the second iteration is:

B(1)	A(2)
B(2)	=
B(3)	3.0
B(4)	

Typed as

A(2) = 3.0

The result of the DECODE statement is to extract the digits 1, 2, and 3 from C and convert them to floating-point binary values and store them in B(1), B(2), and B(3). Then skip the next character (4) and extract the digit 5 from C, convert it to a floating-point binary value, and store it in B(4).

CHAPTER 6
SPECIFICATION STATEMENTS

Specification statements allocate storage and furnish information about variables and constants to the compiler. Specification statements may be divided into three categories, as follows:

a. Storage specification statements: DIMENSION, COMMON, and EQUIVALENCE.

b. Data specification statements: DATA and BLOCK DATA.

c. Type declaration statements: INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, SUBSCRIPT INTEGER, and IMPLICIT.

By extending the USA Standard in regard to specification statements, PDP-10 FORTRAN IV allows the following statements to be used anywhere in the program, provided that the variables they specify appear in executable statements only after the particular specification statement. The specification statement must not appear in the range of a DO loop.

DIMENSION statement
EXTERNAL statement (described in Chapter 7)
COMMON statement
EQUIVALENCE statement
Type declaration statements
DATA statement

A sample program that incorporates these statements follows.

```
        DOUBLE PRECISION D
        DIMENSION Y(10), D(5)
        Y(1) = -1.0
        INTEGER XX(5)
        Y(2) = ABS(Y(1))
        DATA XX/1,2,3,4,5
        DO 10 I = 3,7
10      Y(I) = XX(I-2)
        COMMON Z
        Z=Y(1)*Y(2)/(Y(3) + Y(5))
        END
```

Only IMPLICIT statements and arithmetic function definition statements (described in Chapter 7) must appear in the program before any executable statement.

In addition, arrays must be dimensional before being referenced in a NAMELIST, EQUIVALENCE, or DATA statement. DOUBLE PRECISION and COMPLEX arrays must be declared before they are dimensioned.

6.1 STORAGE SPECIFICATION STATEMENTS

6.1.1 DIMENSION Statement

The DIMENSION statement is used to declare identifiers to be array identifiers and to specify the number and bounds of the array subscripts. The information supplied in a DIMENSION statement is required for the allocation of memory for arrays. Any number of arrays may be declared in a single DIMENSION statement. The DIMENSION statement has the form

DIMENSION $S_1, S_2, \ldots, S_n$

where S is an array specification.

Each array variable appearing in the program must represent an element of an array declared in a DIMENSION statement, unless the dimension information is given in a COMMON or TYPE statement. Dimension information may appear only once for a given variable.

Each array specification gives the array identifier and the minimum and maximum values which each of its subscripts may assume in the following form:

```
identifier(min/max,min/max,...,min/max).
```

The minima and maxima must be integers. The minimum must not exceed the maximum. For example, the statement

```
DIMENSION EDGE(-1/1,4/8)
```

specifies EDGE to be a two-dimensional array whose first subscript may vary from -1 to 1 inclusive, and the second from 4 to 8 inclusive.

Minimum values of 1 may be omitted. For example,

```
NET(5, 10)
```

is interpreted as:

```
NET(1/5,1/10)
```

```
Examples:     DIMENSION FORCE(-1/1,0/3,2,2,-7/3)
              DIMENSION PLACE(3,3,3),JI(2,2/4),K(256)
```

Arrays may also be declared in the COMMON or type declaration statements in the same way:

```
          COMMON X(10,4),Y,Z
          INTEGER A(7,32),B
          DOUBLE PRECISION K(-2/6,10)
```

6.1.1.1 Adjustable Dimensions - Within either a FUNCTION or SUBROUTINE subprogram, DIMENSION and TYPE statements may use integer variables in an array specification, provided that the array name and variable dimensions are dummy arguments of the subprogram. The actual array name and values for the dummy variables are given by the calling program when the subprogram is called. The variable dimensions may not be altered within the subprogram (i.e., typing the array DOUBLE PRECISION or COMPLEX after it has been dimensioned) and must be less than or equal to the explicit dimensions declared in the calling program.

```
Example:      SUBROUTINE SBR(ARRAY,M1,M2,M3,M4)
              DIMENSION ARRAY (M1/M2,M3/M4)
                        .
                        .
                        .
              DO 27 L=M3,M4
              DO 27 K=M1,M2
                        .
                        .
                        .
         27   ARRAY(K,L)=VALUE
                        .
                        .
                        .
              END
```

The calling program for SBR might be:

```
          DIMENSION A1(10,20),A2(1000,4)
                    .
                    .
                    .
          CALL SBR(A1,5,10,10,20)
                    .
                    .
                    .
          CALL SBR(A2,100,250,2,4)
                    .
                    .
                    .
          END
```

6.1.2 COMMON Statement

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 the data of its subprograms may share a common storage area.

The common area may be divided into separate blocks which are identified by block names. A block is specified as follows:

```
/block identifier/identifier,identifier,...,identifier
```

The identifier enclosed in slashes is the block name. The identifiers which follow are the names of the variables or arrays assigned to the block and are placed in the block in the order in which they appear in the block specification. A common block may have the same name as a variable in the same program.

The COMMON statement has the general form

```
COMMON/BLOCK1/A,B,C/BLOCK2/D,E,F/...
```

where BLOCK1,BLOCK2,... are the block names, and A,B,C,... are the variables to be assigned to each block. For example, the statement

```
COMMON/R/X,Y,T/C/U,V,W,Z
```

indicates that the elements X,Y, and T are to be placed in block R in that order, and that U,V,W, and Z are to be placed in block C.

Block entries are linked sequentially throughout the program, beginning with the first COMMON statement. For example, the statements

```
COMMON/D/ALPHA/R/A,B/C/S
COMMON/C/X,Y/R/U,V,W
```

have the same effect as the statement

```
COMMON/D/ALPHA/R/A,B,U,V,W/C/S,X,Y
```

One block of common storage, referred to as blank common, may be left unlabeled. Blank common is indicated by two consecutive slashes. For example,

```
COMMON/R/X,Y//B,C,D
```

indicates that B, C, and D are placed in blank common. The slashes may be omitted when blank common is the first block of the statement.

```
COMMON B,C,D
```

Storage allocation for blocks of the same name begins at the same location for all programs executed together. For example, if a program contains

```
COMMON A,B/R/X,Y,Z
```

as its first COMMON statement, and a subprogram has

```
COMMON/R/U,V,W//D,E,F
```

as its first COMMON statement, the quantities represented by X and U are stored in the same location. A similar correspondence holds for A and D in blank common.

Common blocks may be any length provided that no program attempts to enlarge a given common block declared by a previously loaded program.

Array names appearing in COMMON statements may have dimension information appended if the arrays are not declared in DIMENSION or type declaration statements. For example,

```
COMMON ALPHA,T(15,10,5),GAMMA
```

specifies the dimensions of the array T while entering T in blank common. Variable dimension array identifiers may not appear in a COMMON statement, nor may other dummy identifiers. Each array name appearing in a COMMON statement must be dimensioned somewhere in the program containing the COMMON statement.

6.1.3 EQUIVALENCE Statement

The EQUIVALENCE statement causes more than one variable within a given program to share the same storage location. The EQUIVALENCE statement has the form

$$\text{EQUIVALENCE}(V_1,V_2,\ldots),(V_k,V_{k+1},\ldots),\ldots$$

where the V's are variable names.

The inclusion of two or more references in a parenthetical list indicates that the quantities in the list are to share the same memory location. For example,

```
EQUIVALENCE(RED,BLUE)
```

specifies that the variables RED and BLUE are stored in the same location.

The relation of equivalence is transitive; e.g., the two statements,

```
EQUIVALENCE(A,B),(B,C)
EQUIVALENCE(A,B,C)
```

have the same effect.

The subscripts of array variables must be integer constants.

Example: `EQUIVALENCE(X,A(3),Y(2,1,4)),(BETA(2,2),ALPHA)`

6.1.4 EQUIVALENCE and COMMON

Identifiers may appear in both COMMON and EQUIVALENCE statements provided the following rules are observed.

a. No two quantities in common may be set equivalent to one another.

b. Quantities placed in a common block by means of EQUIVALENCE statements may cause the end of the common block to be extended. For example, the statements

```
COMMON/R/X,Y,Z
DIMENSION A(4)
EQUIVALENCE(A,Y)
```

causes the common block R to extend from X to A(4), arranged as follows:

```
X
Y A(1)     (same location)
Z A(2)     (same location)
  A(3)
  A(4)
```

c. EQUIVALENCE statements which cause extension of the start of a common block are not allowed. For example, the sequence

```
COMMON/R/X,Y,Z
DIMENSION A(4)
EQUIVALENCE(X,A(3))
```

is not permitted, since it would require A(1) and A(2) to extend the starting location of block R.

6.2 DATA SPECIFICATION STATEMENTS

The DATA statement is used to specify initial or constant values for variables. The specified values are compiled into the object program, and become the values assumed by the variables when program execution begins.

6.2.1 DATA Statement

The data to be compiled into the object program is specified in a DATA statement. The DATA statement has the form

DATA list/d_1,d_2,.../,list/d_k,d_{k+1},.../,...

where each list is in the same form as an input/output list, and the d's are data items for each list.

Indexing may be used in a list provided the initial, limit, and increment (if any) are given as constants. Expressions used as subscripts must have the form

$c_1 * i \pm c_2$

where c_1 and c_2 are integer constants and i is the induction variable. If an entire array is to be defined, only the array identifier need be listed. Variables in COMMON may appear on the lists only if the DATA statement occurs in a BLOCK DATA subprogram. (See Chapter 7, Section 7.6)

The data items following each list correspond one-to-one with the variables of the list. Each item of the data specifies the value given to its corresponding variable. Data items may be numeric constants, alphanumeric strings, octal constants, or logical constants. For example,

```
DATA ALPHA, BETA/5, 16.E-2/
```

specifies the value 5 for ALPHA and the value .16 for BETA.

Alphanumeric data is packed into words according to the data word size in the manner of A conversion; however, excess characters are not permitted. The specification is written as nH followed by n characters or is imbedded in single quotes.

Octal data is specified by the letter O or the character ", followed by a signed or unsigned octal integer of one to twelve digits.

Logical constants are written as .TRUE.,.FALSE., T, or F.

Example:
```
DATA NOTE,K/4HFOOT, O-7712/
DATA QUOTE/'QUOTE'/
```

Any item of the data may be preceded by an integer followed by an asterisk. The integer indicates the number of times the item is to be repeated. For example,

```
DATA(A(K),K=1,20)/61E2, 19*32E1/
```

specifies 20 values for the array A; the value 6100 for A(1); the value 320 for A(2) through A(20).

6.2.2 BLOCK DATA Statement

The BLOCK DATA statement has the form:

```
BLOCK DATA
```

This statement declares the program which follows to be a data specification subprogram. Data may be entered into labeled or blank common.

The first statement of the subprogram must be the BLOCK DATA statement. The subprogram may contain only the declarative statements associated with the data being defined.

Example:

```
BLOCK DATA
COMMON/R/S,Y/C/Z,W,V
DIMENSION Y(3)
COMPLEX Z
DATA Y/1E-1,2*3E2/,X,Z/11.877D0,(-1.41421,1.41421)/
END
```

Data may be entered into more than one block of common in one subprogram.

6.3 TYPE DECLARATION STATEMENTS

The type declaration statements INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, IMPLICIT, and SUBSCRIPT INTEGER are used to specify the type of identifiers appearing in a program. An identifier may appear in only one type statement. Type statements may be used to give dimension specifications for arrays.

The explicit type declaration statements have the general form

```
type identifier,identifier,identifier...
```

where type is one of the following:

```
INTEGER,REAL,DOUBLE PRECISION,COMPLEX,LOGICAL,
SUBSCRIPT INTEGER
```

The listed identifiers are declared by the statement to be of the stated type. Fixed-point variables in a SUBSCRIPT INTEGER statement must fall between -2^{27} and 2^{27}.

6.3.1 IMPLICIT Statement

The IMPLICIT statement has the form

IMPLICIT $\text{type}_1(a_1,a_2,\ldots),\ldots,\text{type}_2(a_3,a_4,\ldots)$

where type represents INTEGER, REAL, LOGICAL, COMPLEX, or DOUBLE PRECISION, and $a_1 a_2$,... represent single alphabetic characters, each separated by commas, or a range of characters (in alphabetic sequence) denoted by the first and last characters of the range separated by a minus sign (e.g., (A-D)).

This statement causes any program variable which is not mentioned in a type statement, and whose first character is one of those listed in the IMPLICIT statement, to be classified according to the type appearing before the list in which the character appears. As an example, the statement

```
IMPLICIT REAL(A-D,L,N-P)
```

causes all variables starting with the letters A through D,L, and N through P to be typed as real, unless they are explicitly declared otherwise.

The initial state of the compiler is set as if the statement

```
IMPLICIT REAL(A-H,O-Z), INTEGER(I-N)
```

were at the beginning of the program. This state is in effect unless an IMPLICIT statement changes the above interpretation; i.e., identifiers, whose types are not explicitly declared, are typed as follows.

a. Identifiers beginning with I, J, K, L, M, or N are assigned interger type.
b. Identifiers not assigned integer type are assigned real type.

If the program contains an IMPLICIT statement, this statement will override throughout the program the implicit state initially set by the compiler. No program may contain more than one IMPLICIT declaration for the same letter.

CHAPTER 7
SUBPROGRAM STATEMENTS

FORTRAN subprograms may be either internal or external. Internal subprograms are defined and may be used only within the program containing the definition. The arithmetic function definition statement is used to define internal functions.

External subprograms are defined separately from (i.e., external to) the programs that call them, and are complete programs which conform to all the rules of FORTRAN programs. They are compiled as closed subroutines; i.e., they appear only once in the object program regardless of the number of times they are used. External subprograms are defined by means of the statements FUNCTION and SUBROUTINE.

7.1 DUMMY IDENTIFIERS

Subprogram definition statements contain dummy identifiers, representing the arguments of the subprogram. They are used as ordinary identifiers within the subprogram definition and indicate the sort of arguments that may appear and how the arguments are used. The dummy identifiers are replaced by the actual arguments when the subprogram is executed.

7.2 LIBRARY SUBPROGRAMS

The standard FORTRAN IV library for the PDP-10 includes built-in functions, FUNCTION subprograms, and SUBROUTINE subprograms, listed and described in Chapter 8. Built-in functions are open subroutines; that is, they are incorporated into the object program each time they are referred to by the source program. FUNCTION and SUBROUTINE subprograms are closed subroutines; their names derive from the types of subprogram statements used to define them.

7.3 ARITHMETIC FUNCTION DEFINITION STATEMENT

The arithmetic function definition statement has the form:

```
identifier(identifier,identifier,...)=expression
```

This statement defines an internal subprogram. The entire definition is contained in the single statement. The first identifier is the name of the subprogram being defined.

Arithmetic function subprograms are single-valued functions with at least one argument. The type of the function is determined by the type of the function identifier.

The identifiers enclosed in parentheses represent the arguments of the function. These are dummy identifiers; they may appear only as scalar variables in the defining expression. Dummy identifiers have meaning and must be unique only within the defining statement. Dummy identifiers must agree in order, number, and type with the actual arguments given at execution time.

Identifiers, appearing in the defining expression, which do not represent arguments are treated as ordinary variables. The defining expression may include external functions or other previously defined arithmetic statement functions.

All arithmetic function definition statements must precede the first executable statement of the program.

Examples:
```
SSQR(K)=K*(K+1)*(2*K+1)/6
ACOSH(X)=(EXP(X/A)+EXP(-X/A))/2
```

In the last example above, X is a dummy identifier and A is an ordinary identifier. At execution time, the function is evaluated using the current value of the quantity represented by A.

7.4 FUNCTION SUBPROGRAMS

A FUNCTION subprogram is a single-valued function that may be called by using its name as a function name in an arithmetic expression, such as FUNC(N), where FUNC is the name of the subprogram that evaluates the corresponding function of the argument N. A FUNCTION subprogram begins with a FUNCTION statement and ends with an END statement. It returns control to the calling program by means of one or more RETURN statements.

7.4.1 FUNCTION Statement

The FUNCTION statement has the form:

```
FUNCTION identifier(argument,argument,...)
```

This statement declares the program which follows to be a FUNCTION subprogram. The identifier is the name of the function being defined. This identifier must appear as a scalar variable and be assigned a value during execution of the subprogram which is the function value.

Arguments appearing in the list enclosed in parentheses are dummy arguments representing the function argument. The arguments must agree in number, order, and type with the actual arguments used in the calling program. FUNCTION subprogram arguments may be expressions, alphanumeric strings, array names, or subprogram names.

Dummy arguments may appear in the subprogram as scalar identifiers, array identifiers, or subprogram identifiers. A function must have at least one dummy argument. Dummy arguments representing array names must appear within the subprogram in a DIMENSION statement, or one of the type statements that provide dimension information. Dimensions given as constants must equal the dimensions of the corresponding arrays in the calling program. In a DIMENSION statement, dummy identifiers may be used to specify adjustable dimensions for array name arguments. For example, in the statement sequence:

```
          FUNCTION TABLE(A,M,N,B,X,Y)
                       .
                       .
                       .
          DIMENSION A(M,N),B(10),C(50)
```

The dimensions of array A are specified by the dummies M and N, while the dimension of array B is given as a constant. The various values given for M and N by the calling program must be those of the actual arrays which the dummy A represents. The arrays may each be of different size but must have two dimensions. The arrays are dimensioned in the programs that use the function.

Dummy dimensions may be given only for dummy arrays. In the example above the array C must be given absolute dimensions, since C is not a dummy identifier. A dummy identifier may not appear in an EQUIVALENCE statement in the FUNCTION subprogram.

A function must not modify any arguments which appear in the FORTRAN arithmetic expression calling the function. Modification of implicit arguments from the calling program, such as variables in COMMON and DO loop indexes, is not allowed. The only FORTRAN statements not allowed in a FUNCTION subprogram are SUBROUTINE, BLOCK DATA, and another FUNCTION statement.

7.4.1.1 Function Type - The type of the function is the type of identifier used to name the function. This identifier may be typed, implicitly or explicitly, in the same way as any other identifier. Alternatively, the function may be explicitly typed in the FUNCTION statement itself by replacing the word FUNCTION with one of the following.

```
          INTEGER FUNCTION
          REAL FUNCTION
          COMPLEX FUNCTION
          LOGICAL FUNCTION
          DOUBLE PRECISION FUNCTION
```

For example, the statement

```
COMPLEX FUNCTION HPRIME(S,N)
```

is equivalent to the statements

```
FUNCTION HPRIME(S,N)
COMPLEX HPRIME
```

Examples:

```
FUNCTION MAY(RANGE,EP,YP,ZP)
COMPLEX FUNCTION COT(ARG)
DOUBLE PRECISION FUNCTION LIMIT(X,Y)
```

7.5 SUBROUTINE SUBPROGRAMS

A SUBROUTINE subprogram may be multivalued and can be referred to only by a CALL statement. A SUBROUTINE subprogram begins with a SUBROUTINE statement and returns control to the calling program by means of one or more RETURN statements.

7.5.1 SUBROUTINE Statement

The SUBROUTINE statement has the form:

```
SUBROUTINE identifier(argument,argument,...)
```

This statement declares the program which follows to be a SUBROUTINE subprogram. The first identifier is the subroutine name. The arguments in the list enclosed in parentheses are dummy arguments representing the arguments of the subprogram. The dummy arguments must agree in number, order, and type with the actual arguments used by the calling program.

SUBROUTINE subprograms may have expressions, alphanumeric strings, array names, and subprogram names as arguments. The dummy arguments may appear as scalar, array, or subprogram identifiers.

Dummy identifiers which represent array names must be dimensioned within the subprogram by a DIMENSION or type declaration statement. As in the case of a FUNCTION subprogram, either constants or dummy identifiers may be used to specify dimensions in a DIMENSION statement. The dummy arguments must not appear in an EQUIVALENCE or COMMON statement in the SUBROUTINE subprogram.

A SUBROUTINE subprogram may use one or more of its dummy identifiers to represent results. The subprogram name is not used for the return of results. A SUBROUTINE subprogram need not have any argument at all.

Examples:

```
SUBROUTINE FACTOR(COEFF,N,ROOTS)
SUBROUTINE RESIDU(NUM,N,DEN,M,RES)
SUBROUTINE SERIES
```

The only FORTRAN statements not allowed in a function subprogram are FUNCTION, BLOCK DATA, and another SUBROUTINE statement.

7.5.2 CALL Statement

The CALL statement assumes one of two forms:

```
CALL identifier
CALL identifier (argument,argument,...,argument)
```

The CALL statement is used to transfer control to SUBROUTINE subprogram. The identifier is the subprogram name.

The arguments may be expressions, array identifiers, alphanumeric strings or subprogram identifiers; arguments may be of any type, but must agree in number, order, type, and array size (except for adjustable arrays, as discussed under the DIMENSION statement) with the corresponding arguments in the SUBROUTINE statement of the called subroutine. Unlike a function, a subroutine may produce more than one value and cannot be referred to as a basic element in an expression.

A subroutine may use one or more of its arguments to return results to the calling program. If no arguments at all are required, the first form is used.

Examples:

```
CALL EXIT
CALL SWITCH(SIN,2.LE.BETA,X**4,Y)
CALL TEST(VALUE,123,275)
```

The identifier used to name the subroutine is not assigned a type and has no relation to the types of the arguments. Arguments which are constants or formed as expressions must not be modified by the subroutine.

7.5.3 RETURN Statement

The RETURN statement has the form:

```
RETURN
```

This statement returns control from a subprogram to the calling program. Normally, the last statement executed in a subprogram is a RETURN statement. Any number of RETURN statements may appear in a subprogram.

7.6 BLOCK DATA SUBPROGRAMS

A BLOCK DATA subprogram is a data specification subprogram and is used to enter initial values into variables in COMMON for use by FORTRAN subprograms and MACRO-10 main programs (see Chapter 9). No executable statements may appear in a BLOCK DATA subprogram.

7.6.1 BLOCK DATA Statement

The BLOCK DATA statement has the form:

```
BLOCK DATA
```

This statement declares the program which follows to be a data specification subprogram and it must be the first statement of the subprogram (see Chapter 6, Section 6.2.2).

7.7 EXTERNAL STATEMENT

FUNCTION and SUBROUTINE subprogram names may be used as the actual arguments of subprograms. Such subprogram names must be distinguished from ordinary variables by their appearance in an EXTERNAL statement. The EXTERNAL statement has the form:

```
EXTERNAL identifier, identifier,...,identifier
```

This statement declares the listed identifiers to be subprogram names. Any subprogram name given as an argument to another subprogram must have previously appeared in an external declaration in the calling program (i.e., as an identifier in an EXTERNAL or CALL statement or as a function name in an expression).

Example:

```
EXTERNAL SIN,COS
       .
       .
CALL TRIGF(SIN,1.5,ANSWER)
       .
       .
CALL TRIGF(COS,.87,ANSWER)
       .
       .
END

SUBROUTINE TRIGF(FUNC,ARG,ANSWER)
       .
       .
ANSWER = FUNC(ARG)
       .
       .
RETURN
END
```

To reference external variables from a MACRO-10 program by name, place the variables in named COMMON. Use the name of the variable as the name of the COMMON block:

```
COMMON /A/A /B/B (13) /C C(6,7)
```

7.8 SUMMARY OF PDP-10 FORTRAN IV STATEMENTS

CONTROL STATEMENTS

General Form	Section References
ASSIGN i to m	4.1.3
CALL name $(a_1, a_2, \ldots)$	7.5.2
CONTINUE	4.4
DO i $m=m_1, m_2, m_3$	4.3
GO TO i	4.1.1
GO TO m	4.1.3
GO TO m, $(i_1, i_2, \ldots)$	4.1.3
GO TO $(i_1, i_2, \ldots)$, m	4.1.2
IF $(e_1) i_1, i_2, i_3$	4.2.1
IF (e_2)s	4.2.2
PAUSE	4.5
PAUSE j	4.5
PAUSE 'h'	4.5
RETURN	7.5.3
STOP	4.6
END	4.7

DATA TRANSMISSION STATEMENTS

General Form	Section References
ACCEPT f	5.2.9
ACCEPT f, list	5.2.9
BACKSPACE unit	5.3
DECODE (n,f,v)list	5.4
END FILE unit	5.3
ENCODE (n,f,v)list	5.4
FORMAT (g)	5.1.1
PRINT f	5.2.3
PRINT f, list	5.2.3

General Form	Section References
PUNCH f	5.2.4
READ f	5.2.7
READ f, list	5.2.7
READ (unit,f)	5.2.7
READ (unit,f)list	5.2.7
READ (unit)list	5.2.7
READ (unit,name$_1$)	5.2.7
READ (unit #R,f)list	5.2.7
READ (unit,f,END=c,ERR=d)list	5.2.7
READ (unit,f,END=c)list	5.2.7
READ (unit,f,ERR=d)list	5.2.7
REREAD f,list	5.2.8
REWIND unit	5.3
SKIP RECORD unit	5.3
TYPE f	5.2.5
TYPE f,list	5.2.5
WRITE (unit,f)	5.2.6
WRITE (unit,f)list	5.2.6
WRITE (unit)list	5.2.6
WRITE (unit,name$_1$)	5.2.6
WRITE (unit #R,f)list	5.2.6
UNLOAD unit	5.3

SPECIFICATION STATEMENTS

General Form	Section References
COMMON $a(n_1,n_2,\ldots),b(n_3,n_4,\ldots),\ldots$	6.1.2
COMPLEX $a(n_1,n_2,\ldots),b(n_3,n_4,\ldots),\ldots$	6.3
DATA $t,u,\ldots/k_1,k_2,k_3,\ldots/$ $v,w,\ldots/k_4,k_5,k_6,\ldots/\ldots$	6.2.1
DIMENSION $a(n_1,n_2,\ldots),b(n_1,n_2,\ldots),\ldots$	6.1.1
DOUBLE PRECISION $a(n_1,n_2,\ldots),b(n_3,n_4,\ldots),\ldots$	6.3
EQUIVALENCE $(a(n_1,\ldots),b(n_2,\ldots),\ldots),\ldots$ $(c(n_3,\ldots),d(n_4,\ldots),\ldots),\ldots$	6.1.3
EXTERNAL y,z,...	7.7
IMPLICIT $type_1(1_1-1_2),type_2(1_3-1_4),\ldots$	6.3.1

General Form	Section Reference
INTEGER $a(n_1,n_2,...),b(n_3,n_4,...),...$	6.3
LOGICAL $a(n_1,n_2,...),b(n_3,n_4,...),...$	6.3
NAMELIST $/name_1/a,b,.../name_2/c,d,...$	5.1.2
REAL $a(n_1,n_2,...)b(n_3,n_4,...),...$	6.3
SUBSCRIPT INTEGER $a(n_1,n_2,...),b(n_3,...),...$	6.3

ARITHMETIC STATEMENT FUNCTION DEFINITION

General Form	Section Reference
name(a,b,...)=e	7.3

NOTE:

$a_1,a_2,...$	are expressions
a,b,c,d	are variable names
c	is the statement number to which control is transferred upon encountering an end-of-file
d	is the statement number to which control is transferred upon encountering an error condition on the input data.
e	is an expression
e_1	is a noncomplex expression
e_2	is a logical expression
f	is a format number
g	is a format specification
'h'	is an alphanumeric string
$i,i_1,i_2,...$	are statement numbers
j	is an integer constant
$k_1,k_2,...$	are constants of the general form j*k where k is any constant
$l_1,l_2,...$	are letters
list	is an input/output list
m	is an integer variable name
m_1,m_2,m_3	are integer expressions
$n_1,n_2,...$	are dimension specifications
n	are the number of ASCII characters
name	is a subroutine or function name

$name_1$, $name_2$	are NAMELIST names
#R	is a record number where I/O begins
s	is a statement (not DO or logical IF)
t,u,v,w	are variable names or input/output lists
$type_1$, $type_2$, ...	are type specifications
unit	is an integer variable or constant specifying a logical device number
v	is the starting address of the ASCII record referenced
y,z	are external subprogram names

SECTION II
THE RUN TIME SYSTEM

The five chapters of this section contain information on LIB40, SUBPROGRAM calling sequences, accumulator usage, compiler switches and diagnostic messages, and characteristics of the PDP-10 from a FORTRAN programmer's point of view.

CHAPTER 8
LIB40

LIB40 is a single file which contains all of the programs in the FORTRAN library. It is composed of three groups of programs:

(1) The FORTRAN Operating System.
(2) Science Library.
(3) FORTRAN Utility Subprograms.

8.1 THE FORTRAN OPERATING SYSTEM

The system programs in the FORTRAN Operating System act as the interface between the user's program and the PDP-10. All of these programs are invisible to the user's program. The FORTRAN Operating System is loaded automatically from LIB40 and resides in the user's core area along with the user's main programs and any library functions and subroutines that his programs reference.

8.1.1 FORSE.

FORSE. is the main program of the FORTRAN Operating System and is loaded whenever a FORTRAN main program is in core. The primary functions of FORSE. are

a. FORMAT statement processing,
b. Dispatching of all UUOs, and
c. Control of I/O devices at runtime.

8.1.1.1 FORMAT Processing - FORSE. assumes that all FORMAT statements are syntactically correct since the syntax of each statement is checked by the compiler. FORSE. scans the FORMAT statements and performs the indicated I/O operations. FORSE. invokes the required conversion routine to actually do data conversion. The conversion routine that is used is a function of the conversion indicated in the FORMAT statement and of the data type of the element in the I/O list.

8.1.1.2 UUO Dispatching - Some UUOs are handled minimally by FORSE. (NLIN, NLOUT, MTOP), but the others are handled almost entirely within FORSE.

8.1.1.3 I/O Device Control - FORSE. executes the required carriage control of output devices that are physical listing devices (LPT, TTY) and stores the carriage control character at the beginning of each line if the output is going to a retrievable medium for deferred listing. When listings are deferred, the appropriate switch in PIP can be used to list the file and execute the required carriage control.

8.1.1.4 Additional Functions of FORSE. - FORSE. is responsible for the following:

a. Control of REREAD and ENCODE/DECODE features.

b. Interaction with EOFTST and READ (unit,f,END=C)list to handle end-of-file testing.

c. Control of the assignment of devices to software channels.

d. Control of the handling of filenames for I/O associated with directory devices.

e. Control of the opening and closing of data files.

f. Control the handling of the functions associated with the MAGDEN, BUFFER, IBUFF, OBUFF, DEFINE FILE, TRAPS, and RELEASE subroutines.

8.1.2 I/O Conversion Routines

The I/O conversion routines convert data from internal PDP-10 format to external format or vice versa. The calls to these routines are implied by FORMAT and data transfer statements in the FORTRAN source program. The routines reside as relocatable binary files in LIB40. REL.

Table 8-1
I/O Conversion Routines

Routine	Description
ALPHI.	Alphanumeric ASCII input conversion
ALPHO.	Alphanumeric ASCII output conversion
DIRT.	Double precision input conversion
DOUBT.	Double precision output conversion
FLIRT.	Floating point input conversion
FLOUT.	Floating point output conversion
INTI.	Integer input conversion
INTO.	Integer output conversion
LINT.	Logical input conversion
LOUT.	Logical output conversion

Table 8-1 (Cont)
I/O Conversion Routines

Routine	Description
BINWR.	Binary I/O
OCTI.	Octal input conversion
OCTO.	Octal output conversion
NMLST.	Namelist

8.1.3 FORTRAN UUOs

Operation codes 000 through 077 in the PDP-10 are programmed operators, sometimes referred to as UUO's (Unimplemented User Operators) since from a hardware point of view their function is not prespecified. Some of these op-codes trap to the Monitor and the rest trap to the user program. FORTRAN UUO's trap to the FORTRAN Operating System UUO Handler and are then processed.

Table 8-2
FORTRAN UUOs

UUO	Op Code	Meaning
RESET.	015	Resets all devices, clears tables and flags.
IN.	016	Initializes device for formatted input, does a LOOKUP.
OUT.	017	Initializes device for formatted output, does an ENTER.
DATA.	020	Converts one data element from external to internal format or vice versa depending upon whether input or output is being done. Actual data transfer takes place.
FIN.	021	Terminates data transfer statements.
RTB.	022	Initializes device for unformatted input, similar to IN.
WTB.	023	Initializes device for unformatted output, similar to OUT.
MTOP.	024	Performs Magtape operations, rewind, rewind and unload, backspace, end file, skip, write blank record.
SLIST.	025	Converts entire arrays from external to internal format or vice versa depending upon whether input or output is being done. Actual data transfer takes place.
INF.	026	IFILE. Sets up input filename, similar to IN. but with specified filename.
OUTF.	027	OFILE. Sets up output filename, similar to OUT. but with specified filename.
RERED.	030	REREAD. Reread last record.
NLI.	031	Namelist input.

Table 8-2 (Cont)
FORTRAN UUOs

UUO	Op Code	Meaning
NLO.	032	Namelist output.
DEC.	033	DECODE.
ENC.	034	ENCODE.

8.2 SCIENCE LIBRARY AND FORTRAN UTILITY SUBPROGRAMS

The Science Library and FORTRAN Utility Subprograms extend the capabilities of the FORTRAN language. These subprograms are called explicitly by the user. The subprograms include the built-in FORTRAN math functions and the user-called utility subroutines which provide optional I/O capabilities and control of and information about the program's environment. The optional I/O capabilities and environmental control are achieved by the subroutines from interactions with the FORTRAN Operating System.

8.2.1 FORTRAN IV Library Functions

This section contains descriptions of all standard function subprograms provided with the FORTRAN IV library for the PDP-10. These functions are called by using the function mnemonic as a function name in an arithmetic expression.

Table 8-3
FORTRAN IV Library Functions

Function	Mnemonic	Definition	Number of Arguments	Type of		Storage (Decimal)	External Calls
				Argument	Function		
Absolute value:							
Real	ABS	$\lvert arg \rvert$	1	Real	Real	11	
Integer	IABS	$\lvert arg \rvert$	1	Integer	Integer	11	
Double precision	DABS	$\lvert arg \rvert$	1	Double	Double	8	
Complex to real	CABS	$c=(x^2+y^2)^{1/2}$	1	Complex	Real	22	SQRT
Conversion:							
Integer to real	FLOAT		1	Integer	Real	9	
Real to integer	IFIX	Result is largest integer $\leq a$	1	Real	Integer	12	
Double to real	SNGL		1	Double	Real	16	
Real to double	DBLE		1	Real	Double	5	
Complex to real (obtain real part)	REAL		1	Complex	Real	4	
Complex to real (obtain imaginary part)	AIMAG		1	Complex	Real	5	
Real to complex	CMPLX	$c=Arg_1+i*Arg_2$	2	Real	Complex	5	
Truncation:							
Real to real	AINT	Sign of arg * largest integer $\leq \lvert arg \rvert$	1	Real	Real	12	
Real to integer	INT		1	Real	Integer	13	
Double to integer	IDINT		1	Double	Integer	18	
Remaindering:							
Real	AMOD	The remainder when Arg 1 is divided by Arg 2	2	Real	Real	28	ERROR., TRAPS
Integer	MOD		2	Integer	Integer	6	
Double precision	DMOD		2	Double	Double	79	
Maximum Value:							
	AMAX0	$Max(Arg_1, Arg_2, \ldots)$	≥ 2	Integer	Real	31	FLOAT
	AMAX1			Real	Real		
	MAX0			Integer	Integer		
	MAX1			Real	Integer		IFIX
	DMAX1			Double	Double	22	
Minimum Value:							
	AMIN0	$Min(Arg_1, Arg_2, \ldots)$	≥ 2	Integer	Real	31	FLOAT
	AMIN1			Real	Real		
	MIN0			Integer	Integer		
	MIN1			Real	Integer		IFIX
	DMIN1			Double	Double	22	

Table 8-3 (Cont)
FORTRAN IV Library Functions

Function	Mnemonic	Definition	Number of Arguments	Type of Argument	Type of Function	Storage (Decimal)	External Calls
Transfer of Sign:							
Real	SIGN	$\{\text{Sgn}(\text{Arg}_2)*\lvert\text{Arg}_1\rvert\}$	2	Real	Real	18	
Integer	ISIGN		2	Integer	Integer	18	
Double precision	DSIGN		2	Double	Double	11	
Positive Difference:							
Real	DIM	$\{\text{Arg}_1-\text{Min}(\text{Arg}_1,\text{Arg}_2)\}$	2	Real	Real	7	
Integer	IDIM		2	Integer	Integer	10	
Exponential:							
Real	EXP	$\{e^{\text{Arg}}\}$	1	Real	Real	59	ERROR.
Double	DEXP		1	Double	Double	201	
Complex	CEXP		1	Complex	Complex	73	EXP, SIN, COS, ALOG, ERROR.
Logarithm:							
Real	ALOG	$\log_e$ (Arg)	1	Real	Real	54	ERROR.
	ALOG10	$\log_{10}$ (Arg)	1	Real	Real	54	ERROR.
Double	DLOG	$\log_e$ (Arg)	1	Double	Double	188	
	DLOG10	$\log_{10}$ (Arg)	1	Double	Double	188	
Complex	CLOG	$\log_e$ (Arg)	1	Complex	Complex	56	ALOG, ATAN2, SQRT, ERROR.
Square Root:							
Real	SQRT	$(\text{Arg})^{1/2}$	1	Real	Real	44	ERROR.
Double	DSQRT	$(\text{Arg})^{1/2}$	1	Double	Double	89	
Complex	CSQRT	$c=(x+i\,y)^{1/2}$	1	Complex	Complex	82	SQRT
Sine:							
Real (radians)	SIN	$\{\sin(\text{Arg})\}$	1	Real	Real	71	
Real (degrees)	SIND		1	Real	Real	71	
Double (radians)	DSIN		1	Double	Double	218	
Complex	CSIN		1	Complex	Complex	84	SIN, SINH, COSH, ALOG, EXP
Cosine:							
Real (radians)	COS	$\{\cos(\text{Arg})\}$	1	Real	Real	71	
Real (degrees)	COSD		1	Real	Real	71	
Double (radians)	DCOS		1	Double	Double	218	
Complex	CCOS		1	Complex	Complex	84	SIN, SINH, COSH, ALOG, EXP

Table 8-3 (Cont)
FORTRAN IV Library Functions

Function	Mnemonic	Definition	Number of Arguments	Type of Argument	Type of Function	Storage (Decimal)	External Calls
Hyperbolic:							
Sine	SINH	sinh (Arg)	1	Real	Real	53	EXP, ERROR.
Cosine	COSH	cosh (Arg)	1	Real	Real	11	EXP, ERROR.
Tangent	TANH	tanh (Arg)	1	Real	Real	46	EXP
Arc - sine	ASIN	asin (Arg)	1	Real	Real	37	ATAN, SQRT, ERROR.
Arc - cosine	ACOS	acos (Arg)	1	Real	Real	39	ATAN, SQRT, ERROR.
Arc tangent							
Real	ATAN	atan (Arg)	1	Real	Real	53	
Double	DATAN	atan (Arg)	1	Double	Double	192	
quotient of two arguments	ATAN2	atan (Arg_1/Arg_2)	2	Real	Real	39	ATAN, ERROR., TRAPS
	DATAN2	atan (Arg_1/Arg_2)	2	Double	Double	65	DATAN, ERROR.
Complex Conjugate	CONJG	Arg=X+iY, C=X-iY	1	Complex	Complex	6	
Random Number	RAN	result is a random number	1	Integer, Real, Double, or Complex	Real	32	
	CHANG	converts sign magnitude numbers to 2's complement and vice versa.	1	Real	Real	8	

8.2.2 FORTRAN IV Library Subroutines

This section contains descriptions of all standard subroutine subprograms provided within the FORTRAN IV library for the PDP-10. These subprograms are closed subroutines and are called with a CALL statement.

Table 8-4
FORTRAN IV Library Subroutines

Subroutine Name	Effect
BUFFER	Allows the programmer to specify buffering for a device at one of fifteen levels. CALL BUFFER (unit*, in/out, number) where in/out is 1 for input buffering only, 2 for output buffering only, or 3 for both, and number is the level of buffering ($1 \leq$ number ≤ 15). If number is not specified, 2 is assumed. In calls to two entries in BUFFER, IBUFF and OBUFF, the programmer can specify a non-standard buffer size if the records in his data files exceed standard buffer sizes set by the Monitor. (See Table 12-1.) The programmer cannot change buffer sizes for the disk; IBUFF and OBUFF are designed primarily for Magtape. CALL IBUFF (d,n,s) where d is the device number, n is the number of buffers, and s is the size of buffer.
CHAIN	Reads a segment of coding (Chain file) into core and links it to a program already residing in core. CALL CHAIN (type,device,file) where type is 0 (the next Chain file is read into core immediately above the permanent resident area) or type is 1 (the next Chain file is read into core immediately above the FORTRAN IV program which marks the end of the removable resident). Device is 0,1,2,... FORTRAN IV logical device number (Chain files can be stored on DSK, MTA, or DTA only) corresponding to the device where the Chain file can be found. File is 0 for reading the next file from the selected magnetic tape or 1,2,... for the number of the magnetic tape unit where the Chain file is located.
DATE	Places today's date as left-justified ASCII characters into a dimensioned 2-word array. CALL DATE (array) where array is the 2-word array. The date is in the form dd-mmm-yy

*For explanation, see page 5-80

Table 8-4 (Cont)
FORTRAN IV Library Subroutines

Subroutine Name	Effect
DATE (cont)	where dd is a 2-digit day (if the first digit is 0, it is converted to a blank), mmm is a 3-digit month (e.g., MAR), and yy is a 2-digit year. The date is stored in ASCII code, left-justified in the two words.
DUMP	Causes particular portions of core to be dumped and is referred to in the following form: CALL DUMP $(L_1,U_1,F_1,\ldots,L_n,U_n,F_n)$ where L_i and U_i are the variable names which give the limits of core memory to be dumped. Either L_i or U_i may be upper or lower limits. F_i is a number indicating the format in which the dump is to be performed: 0=octal, 1=real, 2=integer, and 3=ASCII. If F is not 0,1,2,3, the dump is in octal. If F_n is missing, the last section is dumped in octal. If U_n and F_n are missing, an octal dump is made from L to the end of the job area. If L_n, U_n, and F_n are missing, the entire job area is dumped in octal. The dump is terminated by a call to EXIT.
EOF1(unit*)	Skips one end-of-file terminator when found and returns the value TRUE if an end-of-file was found and FALSE if it was not found. Subsequent terminators produce an error message.
EOFC(unit*)	Skips more than one end-of-file terminators when found and returns the value TRUE if an end-of-file was found or FALSE if it was not found.
ERRSET	Allows the user to control the typeout of execution-time arithmetic error messages, ERRSET is called with one argument in integer mode. CALL ERRSET(N) Typeout of each type of error message is suppressed after N occurances of that error message. IF ERRSET is not called, the default value of N is 2.
EXIT	Returns control to the Monitor and, therefore, terminates the execution of the program.
IFILE	Performs LOOKUPs for files to be read from DECtape and disk. CALL IFILE(unit*,filnam) where filnam is a filename consisting of five ASCII characters.

*For explanation, see page 5-80

Table 8-4 (Cont)
FORTRAN IV Library Subroutines

Subroutine Name	Effect
ILL	Sets the ILLEG flag. If the flag is set and an illegal character is encountered in floating-point/double-precision input, the corresponding word is set to zero. CALL ILL
LEGAL	Clears the ILLEG flag. If the flag is set and an illegal character is encountered in floating-point/double-precision input, the corresponding word is set to zero. CALL LEGAL
MAGDEN	Allows specification of magnetic tape density and parity. CALL MAGDEN(unit*,density,parity) where density is the tape density desired (200=200 bpi, 556=556 bpi, or 800=800 bpi) and parity is the tape parity desired (0=odd, 1=even). Even parity is intended for use with BCD-coded tapes only.
OFILE	Performs ENTERs for files to be written on DECtape and disk. CALL OFILE (unit*,filnam) where filnam is a filename consisting of five ASCII characters.
PDUMP	Is referred to in the following form: CALL PDUMP(L_1,U_1,F_1,...,L_n,U_n,F_n) where the arguments are the same as those for DUMP. PDUMP is the same as DUMP except that control returns to the calling program after the dump has been executed.
RELEAS	Closes out I/O on a device initialized by the FORTRAN Operating System and returns it to the uninitialized state. CALL RELEAS (unit*)
SAVRAN	SAVRAN is called with one argument in integer mode. SAVRAN sets its argument to the last random number (interpreted as an integer) that has been generated by the function RAN.
SETRAN	SETRAN has one argument which must be a non-negative integer $< 2^{31}$. The starting value of the function RAN is set to the value of this argument, unless the argument is zero. In this case, RAN uses its normal starting value.

*For explanation, see page 5-80

Table 8-4 (Cont)
FORTRAN IV Library Subroutines

Subroutine Name	Effect
SLITE(i)	Turns sense lights on or off. i is an integer expression. For $1 \leq i \leq 36$ sense light i will be turned on. If i=0, all sense lights will be turned off.
SLITE(i,j)	Checks the status of sense light i and sets the variable j accordingly and turns off sense light i. If i is on, j is set to 1; and if i is off, j is set to 2.
SSWTCH(i,j)	Checks the status of data switch $i(0 \leq i \leq 35)$ and sets the variable j accordingly. If i is set down, j is set to 1; and, if i is up, j is set to 2.
TIME	Returns the current time in its argument(s) in left-justified ASCII characters. If TIME is called with one argument, CALL TIME(X) the time is in the form hh : mm where hh is the hours (24-hour time) and mm is the minutes. If a second argument is requested, CALL TIME(X,Y) the first argument is returned as before and the second has the form ss.t where ss is the seconds and t is the tenths of a second.

CHAPTER 9

SUBPROGRAM CALLING SEQUENCES

This chapter describes the conventions used in writing MACRO subprograms which can be called by FORTRAN IV programs, and FORTRAN subprograms which can be linked to MACRO main programs. The reader is assumed to be familiar with the following texts:

> MACRO-10 Assembler (DEC-10-AMZA-D)
> Section 2.5.8 "Linking Subroutines"
> Figure 7-1, "Sample Program, CLOG"
>
> Time-Sharing Monitors: 10/40, 10/50 (DEC-T9-MTZA-D)
> Section 3.2.2 "Loading Relocatable Binary Files"
>
> Science Library and FORTRAN Utility Subprograms (DEC-10-SFLE-D)
>
> How to Use This Manual - FORTRAN calling sequences

9.1 MACRO SUBPROGRAMS CALLED BY FORTRAN MAIN PROGRAMS

9.1.1 Calling Sequences

The FORTRAN calling sequence, in the main program, for a subroutine is

FORTRAN Code	MACRO Code (Generated by Compiler)
CALL subprog (adr_1, adr_2,...)	JSA 16, subprog ARG $code_1$, adr_1 ARG $code_2$, adr_2 ⋮

where

subprog	is the name of the subprogram
adr_1, adr_2,...	are the addresses of the arguments
$code_1$, $code_2$	are the accumulator fields of the ARG instructions which indicate the type of argument being passed to the subprogram. These codes are as follows:

0	Integer argument	4	Octal argument
1	Unused	5	Hollerith argument
2	Real argument	6	Double-precision argument
3	Logical argument	7	Complex argument

An example of a FORTRAN calling sequence for a subroutine and the MACRO-10 coding generated by the compiler is given below.

FORTRAN Code	MACRO Code
CALL PROG1 (REAL,INT)	JSA 16, PROG1
	ARG 02, REAL
	ARG 00, INT

The MACRO code generated by the compiler is the same for subroutines and functions; however, the FORTRAN code is different.

9.1.2 Returning of Answers

A subroutine returns to its answers in specified locations in the main program. These locations are often given as argument names or as variable names.

A function returns its answer in accumulator 0 (if a single word result) or in accumulators 0 and 1 (if a double-precision or complex result). A function may also return its answer in specified locations (given by argument names in the CALL) or variable names; in any event, however, it must return an answer in accumulator 0 (or accumulators 0 and 1).

A MACRO subprogram access COMMON by declaring as external common block names for labelled COMMON and by declaring .COMM. as external for blank common. A common block name always refers to the same core location as the first element following the block name in a COMMON statement. MACRO subprograms may refer to the remainder of the variables in the common block through additive globals.

9.1.3 Use of Accumulators

For accumulator usage, see Chapter 10, Accumulator Conventions for PDP-10 Main Programs and Subprograms.

9.1.4 Examples of Subprogram Linkage

Three examples of subprogram linkage, one of a subroutine, one of a function subprogram, and one of a FORTRAN main program and MACRO subprogram both referencing COMMON, are given below.

9.1.4.1 Example of a Subroutine Linkage - The coding of the subroutine in this example is followed by the calling sequence.

```
ENTRY       SUBA
SUBA:       0
            MOVE        1,@0(16)        ;GET FIRST ARGUMENT
            IMULI       1, 12           ;MULTIPLY BY 10
            MOVEM       1,@0(16)        ;RETURN RESULT IN ARGUMENT
            JRA         16, 1(16)       ;RETURN TO MAIN PROGRAM
```

FORTRAN Calling Sequence	MACRO Code (Generated by Compiler)
CALL SUBA(INT)	JSA 16, SUBA ARG 00, INT

9.1.4.2 Example of a Function Subprogram Linkage - The coding of the function subprogram in this example is followed by the calling sequence.

```
ENTRY       FNC
FNC:        0
            MOVE        00,@0(16)       ;PICK UP FIRST ARGUMENT
            MOVE        01,@1(16)       ;PICK UP SECOND ARGUMENT
            IMUL        00, 01          ;MULTIPLY BOTH ARGUMENTS
                                        ;RESULT IN AC0
            JRA         16, 2(16)       ;RETURN WITH ANSWER IN AC0
```

FORTRAN Calling Sequence	MACRO Code (Generated by Compiler)
X = FNC (I, 10)	JSA 15, FNC ARG 00, I ARG 00, CONST.

9.1.4.3 Example of a FORTRAN Main Program and a MACRO Subprogram Both Referencing COMMON

```
T           F40         V013        28-NOV-69    12:24

                                                              DIMENSION A(5),B(3,4),C(3)
1M          BLOCK       0
                                                              COMMON C

                                                              COMMON/A/A/B/B/D/D

                                                              A(2)=B(2,3)+C(3)+D
            MOVE        02,D
            FADR        02,B+7
            FADR        02,C+2
            MOVEM       02,A+1
                                                              CALL SUB2
            JSA         16,SUB2
                                                              END

            JSA         16,EXIT
MAIN.%      RESET,      00,0
            JRST        1M

COMMON
C           /.COMM./                0
A           /A/         0
B           /B/         0
D           /D/         0

SUBPROGRAMS

FORSE.
JOBFF
SUB2
EXIT

SCALARS

D           0

ARRAYS
```

```
A           0
B           0
C           0

MAIN.       ERRORS DETECTED: 0

 2K CORE USED
```

```
.MAIN       MACRO.V36    12:23   28-NOV-69

                                                      EXTERNAL .COMM.,A,B,D
                                                      ENTRY      SUB2
            000000     000000     000000     SUB2:    0
            000001     200000     000002              MOVE       0,A+2          ;GET A(3)
            000002     202000     000003              MOVEM      0,B+3          ;STORE IN B(1,2)
            000003     200000     000000              MOVE       0,.COMM.       ;GET C
            000004     202000     000000              MOVEM      0,D            ;STORE IN D
            000005     267716     000000              JRA        16,(16)        ;RETURN TO FORTRAN PROGRAM
                                                      END                       ;END

NO ERRORS DETECTED

PROGRAM BREAK IS 000006

            SYMBOL TABLE

A                      000000  EXT           B         000000  EXT      D         000004' EXT
SUB2                   000000' INT           .COMM.    000003' EXT
```

```
003466 IS THE  PROGRAM BREAK

STORAGE MAP

MAIN.          000140         000035

MAIN.          000146
.COMM.         000150
A              000153
B              000160
D              000174

.MAIN          000175         000006

SUB2           000175

JOBDAT         000203         000000

FORSE.         000203         002374

BUFCA.         001624
BUFHD.         002337
CHINN.         001121
CLOS.          002002
CLOSI.         002000
CLROU.         001763
CLRSY.         001770
DADDR.         002276
DATA.          000000
DEPOT.         001004
DEVIC.         002477
DEVNO.         002172
DYNDV.         002212
DYNND.         002356
ENDLN.         001047
EOFFL.         002205
EOFTS.         001214
EOL.           002275
FI.            001112
FIN.           000000
FMTBG.         002274
FMTEN.         002273
FNCTN.         001751
IORTR.         000334
LOOK.          002034
MTOP.          000000
MTPZ.          002030
NLI.           000000
NLO.           000000
FORSE.         000203
IIB.           001141
IN.            000000
INF.           000000
INP.           002007
INPDV.         002203
NXTCR.         001162
NXTLN.         001172
ONLY1.         002204
OUT.           000000
OUTF.          000000
OUTT.          002013
OVFLS.         002202
PAKFL.         002176
RERDV.         002501
RERED.         000000
RESET.         000000
RIN.           000245
RTB.           000000
SESTA.         002020
SETOU.         001755
SLIST.         000000
STAT.          001774
TCNT1.         002506
TCNT2.         002507
TEMP.          002232
TNAM1.         002133
TANM2.         002132
TPNTR.         002505
TYPE.          002504
UUOH.          001234
WAIT.          002024
WTB.           000000
XIO.           000424

ERROR.         002577         000431
```

```
BPHSE.      002777
DEVER.      002667
DPRER.      002767
DUMER.      003041
ENDTP.      002772
ERROR.      002577
ILLCH.      002634
ILLMG.      003007
ILRED.      003025
ILUUO.      003051
INIER.      002654
LISTB.      002737
LOGEN.      002627
MSNG.       002707
NMLER.      003020
NOROM.      002720
PARER.      003034
QTY1        003170
REDER.      002746
TBLER.      002700
UUOM        003067
WLKER.      002731

EXIT        003230      000002

EXIT        003230
EXIT.       003231

IOADR.      003232      000014

IOADR.      003232

DALPHI      003246      000002

ALPHI.      003246

DALPHO      003250      000002

ALPHO.      003250

DDIRT       003252      000002

DIRT.       003252

DDOUBT      003254      000002

DOUBT.      003254

DFLIRT      003256      000002

FLIRT.      003256

DFLOUT      003260      000002

FLOUT.      003260

DINTI       003262      000002

INTI.       003262

DOCTI       003264      000002

OCTI.       003264

DINTO       003266      000002

INTO.       003266

DOCTO       003270      000002

OCTO.       003270

DLINT       003272      000002

LINT.       003272

DLOUT       003274      000002

LOUT.       003274

DNMLST      003276      000003
```

```
DELIM.      003300
NMLST.      003276

DTFMT       003301      000002

TFMT.       003301

DBINWR      003303      000002

BINDT.      003303

BINEN.      003303
BINWR.      003303
INPT.       003303

DTPFCN      003305      000002

TPFCN.      003305

DEVTB.      003307      000123

DATTB.      003363
DEVLS.      003344
DEVND.      003352
DEVTB.      003307
DVTOT.      000035
MBFBG.      003352
MTABF.      003353
MTACL.      003421
NEG1.       000005
NEG2.       000007
NEG3.       000003
NEG5.       000002
TABP1.      003363
TABPT.      003362

PDLST.      003432      000025

PDLST.      003432

ILL         003457      000007

ILL         003457
```

```
ILLEG.      003465
LEGAL       003462

LOADER 3K CORE
3+3K MAX  1225 WORDS FREE
```

9.2 MACRO MAIN PROGRAMS WHICH REFERENCE FORTRAN SUBPROGRAMS

9.2.1 Calling Sequences

The MACRO code which calls the FORTRAN subprogram should be the same as that produced by the FORTRAN IV compiler when it calls a subroutine. That is:

<u>MACRO Code</u>

```
JSA 16, subprog
ARG code1, adr1
ARG code2, adr2
```

where

subprog	is the name of the subprogram
adr_1, adr_2,...	are the addresses of the arguments
$code_1$, $code_2$	are the accumulator fields of the ARG instruction which indicate the type of argument being passed to the subprogram. These codes are as follows:

0	Integer argument
1	Unused
2	Real argument
3	Logical argument
4	Octal argument
5	Hollerith argument
6	Double-precision argument
7	Complex argument

Both subroutines and functions are called in this manner.

9.2.2 Returning of Answers

A FORTRAN subroutine returns its answers in specified locations in the main program. These locations may be given as variable names in COMMON or as argument names.

A FORTRAN function returns its answer in accumulator 0, if a single word result, or in accumulators 0 and 1, if a double-precision or complex result. A function may also return its answer in specified locations given by argument names in the CALL, or variable names in COMMON; in any event, however, it must return an answer in accumulator 0 (or accumulators 0 and 1).

If it is desired to reference a common block of data in both the MACRO main program and the FORTRAN subprogram, it is necessary to set up the common area first by loading a FORTRAN BLOCK DATA program before the MACRO main program and the FORTRAN subprogram.

9.2.3 Example of Subprogram Linkage

The following is an example of a FORTRAN subroutine being called by a MACRO main program. Both programs reference common data. Read and write statements have been omitted for simplification. Because the FORTRAN operating system, FORSE., sets up I/O channels at run time, the MACRO programmer must be sure not to initialize a device on a channel that FORSE. will then try to use, unless he releases the device before FORSE. is called. FORSE. initializes the first device encountered in the user program on software channel 1, the second on channel 2, etc.

It is possible to release a device from its associated channel in a FORTRAN program by a call to the subroutine RELEAS. Channels one through seventeen are available for I/O. If a FORTRAN user wishes to write MACRO programs which do I/O, he may use either FORTRAN UUO's or the channel numbers less than or equal to seventeen but greater than the largest number used by FORSE.

The FORTRAN RESET. UUO should be the first instruction executed in any program which accesses FORTRAN subroutines. For this reason the FORTRAN operating system, which contains the FORTRAN UUO handler routine, must be declared external in the MACRO main program. This causes FORSE. to be loaded. In general, any program in the FORTRAN library referenced in a MACRO program must be declared external. This results in the searching of LIB40 by the Linking Loader and loading the referenced program.

```
BLKDTA.F4        F40      V016     22-JAN-70          15:46

1M       BLOCK   0                                    BLOCK DATA

                                                      COMMON/A/A/B/B/C/C

                                                      COMMON D

                                                      DIMENSION A(5),B(2,3)

                                                      END

DAT.     BLOCK   0

COMMON
A        /A/        0
B        /B/        0
C        /C/        0
D        /.COMM./            0

SUBPROGRAMS

JOBFF

SCALARS

C        0
D        0

ARRAYS

A        0
B        0

DAT.     ERRORS DETECTED:  0

2K CORE USED
```

```
.MAIN    MACRO.V40 16:05 22-JAN-70
START .MAC

                                          ENTRY      START
                                          EXTERNAL              .COMM.,A,B,C,ARGS,FORSE.,EXIT.
      000000   015000   000000   START:   RESET.     00,0                 ;DO FORTRAN UUO RESET, FOUND IN FORSE.
      000001   200000   000000            MOVE       0,A                  ;GET A(1)
      000002   202000   000000            MOVEM      0,B                  ;STORE IN B(1,1)
      000003   200000   000000            MOVE       0,C                  ;GET C
      000004   202000   000000            MOVEM      0,.COMM.             ;STORE IN D
      000005   200040   000002            MOVE       1,A+2                ;GET A(3)
      000006   202040   000005            MOVEM      1,B+5                ;STORE IN B(2,3)
      000007   266700   000000            JSA        16,ARGS              ;GO TO FORTRAN SUBROUTINE ARGS
                                          JSA        16,EXIT.             ;EXIT. FORTRAN EXIT ROUTINE WHICH PRINTS
      000010   266700   000000                                            ;OUT SUMMARIES AND ALSO CALLS MONITOR
                                                                          ;LEVEL EXIT UUO. USER HAS OPTION TO USE
                                                                          ;EITHER

                                          END        START                ;END

NO ERRORS DETECTED

PROGRAM BREAK IS 000011

START    .MAC      SYMBOL TABLE

A                  000001' EXT        ARGS        000007' EXT      B          000002' EXT
C                  000003' EXT        EXIT.       000010' EXT      FORSE.     000000  EXT
START              000000' ENT        .COMM.      000004' EXT
```

```
1M          BLOCK   0                                   SUBROUTINE ARGS

                                                        COMMON/A/A/B/B/C/C

                                                        COMMON D

                                                        DIMENSION A(5),B(2,3)

                                                        A(1)=B(1,1)+C+D
            MOVE    02,C
            FADR    02,D
            FADR    02,B
            MOVEM   02,A
                                                        RETURN
            JRST    2M
                                                        END

            JRST    2M
ARGS%       ARG     00,0
            MOVEM   15,TEMP.
            MOVEM   16,TEMP.+1
            JRST    1M
2M          MOVE    15,TEMP.
            MOVE    16,TEMP.+1
            JRA     16,0(16)

COMMON
A           /A/     0
B           /B/     0
C           /C/     0
D           /.COMM./        0

SCALARS

ARGS        17
C           0
D           0
```

```
ARRAYS

A          0
B          0

ARGS      ERRORS DETECTED:  0
2K CORE USED




003471 IS THE LOW SEGMENT BREAK
.MAIN     STORAGE MAP        16:06      22-JAN-70
STARTING ADDRESS 000155 PROG .MAIN FILE START
DAT.      000140     000015

          DAT.         000140     A              000140     B              000145     C              000153
          .COMM.       000154

.MAIN     000155       000011

          START        000155

ARGS      000166       000020

          ARGS         000174

JOBDAT    000206       000000

FORSE.    000206       002374

          BUFCA.       001627     BUFHD.         002342     CHINN.         001124     CLOS.          002005
          CLOSI.       002003     CLROU.         001766     CLRSY.         001773     DADDR.         002301
          DATA.        000000     DEPOT.         001007     DEVIC.         002502     DEVNO.         002175
          DYNDV.       002215     DYNND.         002361     ENDLN.         001052     EOFFL.         002210
          EOFTS.       001217     EOL.           002300     FI.            001115     FIN.           000000
          FMTBG.       002277     FMTEN.         002276     FNCTN.         001754     FORSE.         000206
          IIB.         001144     IN.            000000     INF.           000000     INP.           002012
          INPDV.       002206     IORTR.         000337     LOOK.          002037     MTOP.          000000
          MTPZ.        002033     NLI.           000000     NLO.           000000     NXTCR.         001165
          NXTLN.       001175     ONLY1.         002207     OUT.           000000     OUTF.          000000
          OUTT.        002016     OVFLS.         002205     PAKFL.         002201     RERDV.         002504
```

```
            RERED.     000000     RESET.     000000     RIN.       000250     RTB.       000000
            SESTA.     002023     SETOU.     001760     SLIST.     000000     STAT.      001777
            TCNT1.     002511     TCNT2.     002512     TEMP.      002235     TNAM1.     002136
            TNAM2.     002135     TPNTR.     002510     TYPE.      002507     UUOH.      001237
            WAIT.      002027     WTB.       000000     XIO.       000427

ERROR.      002602     000431

            BPHSE.     003002     DEVER.     002672     DPRER.     002772     DUMER.     003044
            ENDTP.     002775     ERROR.     002602     ILLCH.     002637     ILLMG.     003012
            ILRED.     003030     ILUUO.     003054     INIER.     002657     LISTB.     002742
            LOGEN.     002532     MSNG.      002712     NMLER.     003023     NOROM.     002723
            PARER.     003037     QTY1       003173     REDER.     002751     TBLER.     002703
            UUOM       003072     WLKER.     002734

EXIT        003233     000002

            EXIT       003233     EXIT.      003234

IOADR.      003235     000014

            IOADR.     003235

DALPHI      003251     000002

            ALPHI.     003251

DALPHO      003253     000002

            ALPHO.     003253

DDIRT       003255     000002

            DIRT.      003255

DDOUBT      003257     000002

            DOUBT.     003257

DFLIRT      003261     000002

            FLIRT.     003261

DFLOUT      003263     000002

            FLOUT.     003263

DINTI       003265     000002

            INTI.      003265
```

```
DOCTI     003267    000002
          OCTI.     003267
DINTO     003271    000002
          INTO.     003271
DOCTO     003273    000002
          OCTO.     003273
DLINT     003275    000002
          LINT.     003275
DLOUT     003277    000002
          LOUT.     003277
DNMLST    003301    000003
          DELIM.    003303    NMLST.    003301
DTFMT     003304    000002
          TFMT.     003304
DBINWR    003306    000002
          BINDT.    003306    BINEN.    003306    BINWR.    003306    INPT.     003306
DTPFCN    003310    000002
          TPFCN.    003310
DEVTB.    003312    000123
          DATTB.    003366    DEVLS.    003347    DEVND.    003355    DEVTB.    003312
          DVTOT.    000035    MBFBG.    003355    MTABF.    003356    MTACL.    003424
          NEG1.     000005    NEG2.     000007    NEG3.     000003    NEG5.     000002
          TABP1.    003366    TABPT.    003365
PDLST.    003435    000025
          PDLST.    003435
ILL       003462    000007
          ILL       003462    ILLEG.    003470    LEGAL     003465

LOADER 3K CORE
3+3K MAX 1222 WORDS FREE
```

CHAPTER 10
ACCUMULATOR CONVENTIONS FOR MAIN PROGRAMS AND SUBPROGRAMS

10.1 LOCATIONS

Locations specified in the calling sequence for a FORTRAN subprogram may be either required locations or defined locations. A required location is a memory location whose address is specified in the calling sequence for a subprogram. For example, X is a required location in the calling sequence

```
JSA 16, SQRT
ARG     X
```

A defined location is a memory location whose address is specified in the definition of a calling sequence. The location does not appear in the calling sequence. For example in the calling sequence

```
MOVEI 16, MEMORY
PUSHJ  17, DFAS.0
```

MEMORY is required, and AC0, AC1, and AC2 are defined by DFAS.0.

10.2 ACCUMULATORS

10.2.1 Accumulators 0 and 1

When used for subprograms called by JSA, accumulators 0 and 1 may be used at any time without restoring their original contents. These accumulators cannot be required locations. A FORTRAN function returns its answer in accumulator 0 (if a single word result) or in accumulators 0 and 1 (if a double-precision or complex result). A function may also return its answer in specified locations (given by argument names in the CALL) or variable names; in any event, an answer must be returned either in accumulator 0 or in accumulators 0 and 1.

When used for subprograms called by PUSHJ 17, adr, accumulators 0 and 1 may have their contents destroyed. Some subprograms by their definition return an argument in accumulator 0 or 1.

10.2.2 Accumulators 2 Through 15

Accumulators 2 through 15 must not be destroyed by FORTRAN functions, but may be destroyed by FORTRAN subroutines. (Presently subroutines must preserve the contents of accumulator 15.) The contents of these accumulators must not be destroyed by subprograms called by PUSHJ unless the definition of the subroutines requires it.

10.2.3 Accumulators 16 and 17

Accumulator 16 should be used only for JSA-JRA subprogram calls unless the definition of the subprogram sequence requires otherwise. The contents of accumulator 16 may be destroyed by subprograms called by PUSHJ 17, adr.

Accumulator 17 must be used only for pushdown list operations.

10.3 UUOS

User UUO's are not considered subprograms and may not change any locations except those required for input and the contents of accumulators 0 or 1.

10.4 SUBPROGRAMS CALLED BY JSA 16, ADDRESS

The calling sequence is

```
JSA 16, address
ARG     adr1
ARG     adr2
   .
   .
   .
ARG     adrN
```

where each ARG adrN corresponds to one argument of the subprogram.

There may or may not be arguments. If there are arguments, they must be in accumulators 2 through 15. Subroutines called with the FORTRAN CALL statement may, by definition, return an argument in accumulator 0 or 1. Subprograms that are FORTRAN functions (such as SIN or SQRT) may destroy the contents of accumulators 0 and 1. Results are returned in accumulator 0 for single word results and accumulators 0 and 1 for double word results.

10.5 SUBPROGRAMS CALLED BY PUSHJ 17, ADDRESS

See section 10.2. In addition, three consecutive accumulators are required for double-precision addition, subtraction, multiplication, and division operations. The contents of the third accumulator may be destroyed. The "to memory" modes also leave the answer in the defined accumulators. The two arguments of the double-precision operation cannot be in the same accumulators. Complex addition, subtraction, multiplication, and division operations do not destroy locations except those required for the answer and accumulator 16. The two arguments of the complex operation must not be in the same accumulator.

10.6 SUBPROGRAMS CALLED BY UUOS

Subprograms called by UUO's may change the contents of accumulators 0 and 1 only.

Table 10-1
Accumulator Conventions for
PDP-10 FORTRAN IV Compiler and Subprograms

Subprogram Called By: Accumulators	JSA		PUSHJ	UUO
	Functions	Subroutines		
0, 1	1) May be destroyed. 2) May not be used to pass arguments. 3) A result must be returned in 0 or 0 and 1.	1) May be destroyed. 2) May not be used to pass arguments. 3) Results must not be returned.	1) May be destroyed. 2) May be used to pass arguments if the subprogram is defined with an argument in 0 or 0 and 1. 3) Results may be returned if the subprogram is so defined.	1) May be destroyed. 2) May be used to pass arguments except as defined. 3) Results must not be returned.
2-15	1) Must be preserved. 2) Arguments may be passed. 3) Results may be returned if required by calling sequence.	1) May be destroyed. 2) Arguments may be passed. 3) Results must not be returned.	1) Must be preserved unless the definition of subprogram forces results to be returned. 2) Arguments may be passed. 3) Results may be returned if the subprogram is so defined.	1) Must be preserved. 2) Arguments may be passed. 3) Results must not be returned.
16 Reserved for JSA-JRA Operations (except as noted for PUSHJ)	1) Must be preserved. 2) May not be used to pass arguments. 3) Results must not be returned.	1) Must be preserved. 2) May not be used to pass arguments. 3) Results must not be returned.	1) Is destroyed. 2) Used for argument address. 3) Results must not be returned.	1) Must be preserved. 2) May not be used to pass arguments. 3) Results must not be returned.
17 Reserved for Pushdown List Operations	1) Must be preserved. 2) May not be used to pass arguments. 3) Results must not be returned.	1) Must be preserved. 2) May not be used to pass arguments. 3) Results must not be returned.	1) Must be preserved. 2) May not be used to pass arguments. 3) Results must not be returned.	1) Must be preserved. 2) May not be used to pass arguments. 3) Results must not be returned.

CHAPTER 11
SWITCHES AND DIAGNOSTICS

11.1 FORTRAN SWITCHES AND DIAGNOSTICS

Table 11-1
FORTRAN Compiler Switch Options

Switch	Meaning
A[†]	Advance magnetic tape reel by one file.
B[†]	Backspace magnetic tape reel by one file.
C[†]	Generate a CREF-type cross-reference listing. (DSK:CREF.TMP assumed if no list-dev specified)
	Complement: Do not produce cross-reference information (standard procedure).
D	List error message codes only.
	Complement: List complete error message.
E	Print an octal listing of the binary program produced by the compiler in addition to the symbolic listing output.
	Complement: Do not produce octal listing (standard procedure).
M	Complement: Include MACRO coding in the output listing.
	Eliminate the MACRO coding from the output listing (standard procedure).
N	Suppress output of error messages on the Teletype.
	Complement: Output error messages on TTY (standard procedure).
S	If the compiler is running on the PDP-10, produce code for execution on the PDP-6 and vice-versa.
T[†]	Skip to the logical end of the magnetic tape reel.
W[†]	Rewind the magnetic tape reel.
Z[†]	Zero the DECtape directory.

[†]Switches A through C and T, W, and Z must immediately follow the device name or filename.ext to which the individual switch applies.

Table 11-2
FORTRAN Compiler Diagnostics
(Command Errors)

Message	Meaning
?BINARY OUTPUT ERROR dev:filename.ext	An output error has occurred on the device specified for the binary program output.
?CANNOT FIND dev:filename.ext	Filename.ext cannot be found on this device.
?DEVICE INPUT ERROR for command string	Device error occurred while attempting to read Monitor command file.
IMPROPER IO FOR DEVICE dev:	An input device is specified for output (or vice versa) or an illegal data mode was specified (e.g., binary output to TTY).
?INPUT DATA ERROR dev:filename.ext	A read error has occurred on the source device.
?x IS A BAD SWITCH	This specified switch is not recognizable.
?x IS AN ILLEGAL CHARACTER	A character in a command string typein is not recognizable (e.g., FORM-FEED).
? dev: IS NOT AVAILABLE	Either the device does not exist or it has been assigned to another job.
LINKAGE ERROR	Input device error while doing Dump Mode I/O, or not enough core was available to execute the newly loaded program.
? LINKAGE ERROR FOR dev:prog.ext	Specified dev:prog.ext appears in a ! Monitor command string, but cannot be run for some reason.
?LISTING OUTPUT ERROR dev:filename.ext	An output error has occurred on the device specified for the binary program output.
?CANNOT USE dev:filename.ext	The directory on dev: DTAn is full and cannot accept filename.ext as a new file, or a protection failure occurred for a DSK output file, or an illegal filename has been used.
?NOT ENOUGH CORE FOR LINKAGE	Not enough core available to load (with dump mode I/O) the program specified in a ! Monitor command string.
?SYNTAX ERROR IN COMMAND STRING	A syntax error has been detected in a command string typein (e.g., the ← has been omitted).
?X SWITCH ILLEGAL AFTER LEFT ARROW	Cannot change machine type with a file or clear source directory.
?X SWITCH ILLEGAL AFTER FIRST STANDARD FILE	Cannot clear directory after start of compilation (Batch Mode).
?X SWITCH, NO LISTING FILE	A CREF listing requires a listing file.
?INSUFFICIENT CORE - COMPILATION TERMINATED	The compiler has insufficient table space to compile the program.

Table 11-3
FORTRAN Compiler Diagnostics
(Compilation Errors)

Message		Meaning
I-1	DUPLICATED DUMMY VARIABLE IN ARGUMENT STRING	A dummy variable (identifier) may appear only once in any one argument set representing the arguments of a subprogram. (See Section 7.3)
I-2	ARRAY NAME ALREADY IN USE	Any attempt to re-dimension a variable or redefine a scalar as an array is illegal. (See Section 6.1.1)
I-3	ATTEMPT TO REDEFINE VARIABLE TYPE	Once a variable has been defined as either complex, double precision, integer, logical or real it may not be defined again. (See Sections 2.2, 6.3)
I-4	NOT A VARIABLE FORMAT ARRAY	The variable which contains the FORMAT specification read-in at object time must be a diminsioned variable, i.e., an array. (See Section 5.1.1)
I-5	NAME ALREADY USED AS NAMELIST NAME	After a NAMELIST name has been defined, it may appear only in READ or WRITE statements and may not be defined again. (See Section 5.1.2)
I-6	DUPLICATED NAMELIST NAME	A NAMELIST name has already been used as a scalar array or global dummy argument. (See Section 5.1.2)
I-7	A NAME APPEARS TWICE IN AN EXTERNAL STATEMENT	A subprogram name has been declared EXTERNAL more than once. (See Section 7.7)
I-10	SUBPROGRAM NAME ALREADY IN USE	A subprogram name has appeared in another statement as a scalar or array variable, arithmetic function statement name, or COMMON block name. (See Section 7.5)
I-11	DUMMY ARGUMENT IN DATA STATEMENT	Dummy arguments may not appear in DATA statements. (See Section 6.2.1)
I-12	NOT A SCALAR OR ARRAY	The variable defining the starting address for an ENCODE/DECODE statement must be a scalar or an array. (See Section 5.4)
		The I/O unit name of a READ/WRITE statement is not a scalar or array. (See Sections 5.2.6, 5.2.7)
		An attempt to ASSIGN a label number to a variable that is not a scalar or array. (See Sections 2.2)
		An attempt to GO TO through a variable that is not a scalar or array. (See Section 4.1)
I-13	ILLEGAL USE OF DUMMY	Dummy arguments may be used with functions or subprograms only. (See Sections 7.4.1, 7.5.1)
I-14	ILLEGAL DO LOOP PARAMETER	The DO index must be a non-subscripted integer variable while the initial, limit and increment values of the index must be an integer expression - the index may not be zero. (See Section 4.3)
I-15	I/O VARIABLES MUST BE SCALARS OR ARRAYS	Referencing data in an I/O statement other than scalars or arrays is illegal. (See Section 5.2)

Table 11-3 (Cont)
FORTRAN Compiler Diagnostics
(Compilation Errors)

Message		Meaning
S-1	SYNTAX	Indicates an error in the format of the statement referenced.
S-2	ILLEGAL USE OF DO-LOOP	Control may not transfer into the range of a DO from any statement outside its range. (See Section 4.3)
S-3	ILLEGAL FIELD SPECIFICATION	The field width or decimal specification in a FORMAT statement must be integer. The number of Hollerith characters in an H specification must be equal to the number specified. (See Sections 5.1.1.1, 5.1.1.6)
S-4	SCALAR VARIABLE — MAY NOT BE SUBSCRIPTED	An undimensioned variable (a scalar variable) is being illegally subscripted. (See Section 2.2.1)
S-5	ILLEGAL TYPE SPECIFICATION	The type of constant specified is illegal or mispelled. (See Section 2.1)
S-6	ARGUMENT IS NOT SINGLE LETTER	Arguments in parentheses must be single letters in IMPLICIT statement. (See Section 6.3.1)
S-7	'NAMELIST' NOT FOLLOWED BY "/"	The first character following NAMELIST must be /. (See Section 5.1.2)
S-10	ILLEGAL CHARACTER-LINE DELIMITER EXPECTED	The requirements for a complete FORTRAN statement have been satisfied; any additional characters other than a line delimiter are illegal. A carriage return-line feed is a line delimiter. (See Section 1.1)
S-11	A NUMBER WAS EXPECTED	Only arrays which are subprogram arguments can have adjustable dimensions. (See Section 6.1.1.1)
S-12	ILLEGAL USE OF IMPLIED DO LOOP	Implied DO loops in I/O statements must be nested properly. An undefined index variable was used in defining a DO loop. (See Sections 4.3, 5.2.1)
S-13	ATTEMPT TO USE AN ARRAY AS A SCALAR	Variables may be either scalar or array but not both. Variables appearing in a DIMENSION statement must be subscripted when used. (See Section 2.2)
S-14	ARRAY NOT SUBSCRIPTED	See S-13
S-15	ILLEGAL USE OF AN ARITHMETIC FUNCTION NAME	Arithmetic function definition statement name is being used without arguments (i.e., as a scalar) in an arithmetic expression. (See Section 7.3)
S-16	ILLEGAL CHARACTER DETECTED — DELIMITER EXPECTED	A /, or other delimiter is missing.
O-1	BLOCK DATA NOT SEPARATE PROGRAM	Block Data must exist as a separate program. (See Sections 6.2.2, 7.6)
O-2	SUBROUTINE IS NOT A SEPARATE PROGRAM	A subroutine following a main program or another subroutine subprogram may have no statement between it and the preceding programs END statement and must begin with a SUBROUTINE statement. The previous program must have been terminated properly. (See Section 7.5)

Table 11-3 (Cont)
FORTRAN Compiler Diagnostics
(Compilation Errors)

Message		Meaning
O-3	STATEMENT OUT OF PLACE	The IMPLICIT specification statement and any arithmetic function definition statement must appear before any executable statement. (See Chapter 6)
A-1	MINIMUM VALUE EXCEEDS MAXIMUM VALUE	Minimum value of an array exceeds the maximum value specified. (See Section 6.1.1)
A-2	ATTEMPT TO ENTER A VARIABLE INTO COMMON TWICE	A variable name may appear in COMMON statement only once. (See Section 6.1.2)
A-3	ATTEMPT TO EQUIVALENCE A DUMMY ARGUMENT	Dummy argument identifiers of subprograms may not appear in EQUIVALENCE statements in that subprogram. (See Sections 6.1.3, 7.1)
A-4	NOT A CONSTANT OR DUMMY ARGUMENT	Only constant and dummy arguments may be used as arguments in dimension statements. (See Section 7.4.1)
M-1	TOO MANY SUBSCRIPTS	An array variable appears with more subscripts than specified. (See Sections 2.2.2, 6.1.1)
M-2	NOT ENOUGH SUBSCRIPTS	An array variable appears with too few subscripts. (See Sections 2.2.2, 6.1.1)
M-3	CONSTANT OVERFLOW	Too many significant digits in the formation of a constant or the exponent is too large. (See Section 2.1)
M-4	ILLEGAL 'IF' ARGUMENT	Logical IF or DO statement adjacent to a logical IF statement, or illegal expression within a logical IF statement. (See Sections 4.2.2, 4.3)
M-5	ILLEGAL CONVERSION IMPLIED	Attempt to mix double precision and complex data in the same expression. (See Section 2.3.1)
M-6	NUMBER TOO LARGE	Illegal statement label. (See Section 1.1.1)
M-7	UNTERMINATED HOLLERITH STRING	A missing single quote or fewer than n characters following an "nH" specification. (See Section 5.1.1.6)
M-10	ILLEGAL DO LOOP CLOSE	Illegal statement terminating a DO loop. (See Section 4.3)
M-11	VARIABLES AND DATA DO NOT MATCH	Incorrect number of constants supplied for a DATA statement. (See Section 6.2.1)
M-12	NON-INTEGER PARAMETER IN 'DO' STATEMENT	DO statement parameters must be integers. (See Section 4.3)
M-13	NON-INTEGER SUBSCRIPT	Array subscripts must be integer constants, variables, or expressions. (See Section 4.3)
M-14	ILLEGAL COMPARISON OF COMPLEX VARIABLES	The only comparison allowed of complex variables is .NE. or .EQ. (See Sections 2.2, 2.3)
M-15	TOO MANY CONTINUATION CARDS	More than 19 continuation cards. (See Section 1.1.2)
M-16	NON-INTEGER I/O UNIT OR CHARACTER COUNT	The I/O unit variable of a READ/WRITE statement, or the character count variable of an ENCODE/DECODE statement, is not an integer variable. (See Sections 5.2.6, 5.2.7, 5.4)

Table 11-3 (Cont)
FORTRAN Compiler Diagnostics
(Compilation Errors)

Message	Meaning
EXCESSIVE COUNT	The number specified is greater than the maximum possible number of characters in a statement.
OPEN DO LOOPS	The list of statements are specified in DO statements but not defined.
UNDEFINED LABELS	The list of labels that do not appear in the label field.
MULTIPLY DEFINED LABELS	The list of labels that appeared more than once in the label field.
ALLOCATION ERRORS	The list of EQUIVALENCEd COMMON variables which have attempted to extend the beginning of a COMMON block.

Table 11-4
FORTRAN Operating System Diagnostics
(Execution Errors)

Message	Meaning
?DEVICE dev: NOT AVAILABLE	FORSE. tried to initialize a device which either does not exist or has been assigned to another job.
?DEVICE NUMBER n IS ILLEGAL	A nonexistent device number was selected.
?DOUBLE PRECISION OVER OR UNDERFLOW	An overflow or underflow error occurred while adding, subtracting, multiplying, or dividing two double-precision numbers.
END OF FILE ON dev:	A premature end of file has occurred on an input device.
?END OF TAPE ON dev:	The end of tape marker has been sensed during input or output.
?FILE NAME filename.ext NOT ON DEVICE dev:	Filename.ext cannot be found in the directory of the specified device.
?ILLEGAL CHARACTER, x, IN FORMAT	The illegal character x is not valid for a FORMAT statement.
?ILLEGAL CHARACTER, x, IN INPUT STRING	The illegal character x is not valid for this type of input.
?ILLEGAL MAGNETIC TAPE OPERATION, TAPE dev:	An attempt was made to skip a record after performing output on a magnetic tape.
?ILLEGAL PHYSICAL RECORD COUNT, TAPE dev:	FORSE. has encountered an inconsistency in the physical record count on a magnetic tape.
?ILLEGAL USER UUO uuu AT USER loc	An illegal user UUO to FORSE. was encountered at location loc.
?INPUT DEVICE ERROR ON dev:	A data transmission error has been detected in the input from a device.

Table 11-4 (Cont)
FORTRAN Operating System Diagnostics
(Execution Errors)

Message	Meaning
?MORE THAN 15 DEVICES REQUESTED	Too many devices have been requested.
?NAMELIST SYNTAX ERROR	Improper mode of I/O (octal or Hollerith), incorrect variable name.
?NO ROOM FOR FILE filename.ext ON DEVICE dev:	There is no room for the file in the directory of the named device or no room on the device.
program name NOT LOADED	A dummy routine was loaded instead of the real one. Generally, this error occurs when a loaded program is patched to include a call to a library program which was not called by the original program at load time.
?OUTPUT DEVICE ERROR ON dev:	A data transmission error has been detected during output to a device.
?PARITY ERROR ON dev:	A parity error has been detected.
?REREAD EXECUTED BEFORE FIRST READ	A reread was attempted before initializing the first input device.
?TAPE RECORD TOO SHORT ON UNIT n	The data list is too long on a binary tape READ operation.
?dev: WRITE PROTECTED	The device is WRITE locked.

NOTE

With the exception of the messages ILLEGAL USER UUO uuu AT USER loc and ENCODE/DECODE ERROR, all messages are followed by a second message

LAST FORTRAN I/O AT USER LOC adr

Several arithmetic error conditions can occur during execution time.

a. Overflow - An attempt was made to create either a positive number greater than the largest representable positive number or a negative number greater in magnitude than the most negative representable number (in the appropriate mode).

Example: For I an integer,

$377777777777 < I < 400000000000$ (octal)

b. Underflow - An attempt was made to create either a positive non-zero number smaller than the smallest representable positive non-zero number or a negative number smaller in magnitude than the negative number whose magnitude is the smallest representable.

Example: For X a real non-zero number,

$777400000000 < X < 000400000000$

c. Divide Check - An attempt was made to divide by zero.

d. Improper Arguments for LIB40 math routines - For example, an attempt was made to find the arc sine of an argument greater than 1.0.

When overflow, underflow, or divide check errors occur in the user's FORTRAN program, the Monitor calls the LIB40 routine OVTRAP. This routine replaces the resulting numbers, if the numbers are floating point, with either zero in the case of underflow or ± the largest representable number in the cases of overflow and divide check. OVTRAP does not affect numbers in integer mode.

Overflow, underflow, and divide check errors occurring in LIB40 math routines are handled differently from when they occur in the user's program: only if the final answer from a routine is in error is an error condition considered to exist. If the answer is floating point, it is set to the appropriate value as for user program errors. Integer answers are handled in various ways. (See the Science Library and FORTRAN Utility Subprograms, DEC-10-SFLE-D.)

When an error condition occurs in a user program or in a final answer from a LIB40 math routine, an error message is typed. Presently there are eight distinct error messages.

Error Message No.	Error Message
1	INTEGER OVERFLOW PC=nnnnnn
2	INTEGER DIVIDE CHECK PC=nnnnnn
3	FLOATING OVERFLOW PC=nnnnnn
4	FLOATING UNDERFLOW PC=nnnnnn
5	FLOATING DIVIDE CHECK PC=nnnnnn
6	ATTEMPT TO TAKE SQRT OF NEGATIVE ARG
7	ACOS OF ARG > 1.0 IN MAGNITUDE
8	ASIN OF ARG > 1.0 IN MAGNITUDE

NOTE

nnnnnn = location at which the error occurred.

After two typeouts of a particular error message, further typeout of that error message is suppressed. At the end of execution, a summary listing the actual number of times each error message occurred is typed out. If the user wishes to permit more than two typeouts for each error message, he may do so by calling the routine ERRSET at the beginning of the executable part of his main program. ERRSET accepts one argument in integer mode. This argument is the number of typeouts that are permitted for each error message before suppression occurs. This routine is used to obtain the PC information which would otherwise be lost. Alternatively, because of the slowness of the Teletype output, the user may wish to suppress typeout of the messages entirely. This can be done by calling ERRSET with an argument of zero. Suppression of typeout can also be accomplished during execution by typing ↑O on the Teletype.

Error messages and the summary are output to the Teletype (or the output device when running BATCH), regardless of the device assignments that have been made.

The treatment of overflow, underflow, and divide check errors in MACRO programs (those that are loaded with OVTRAP) can, to a certain extent, be manipulated by the user. (See OVTRAP in the Science Library and FORTRAN Utility Subprogram manual.)

CHAPTER 12

RELATED FEATURES OF THE PDP-10

12.1 ASCII CHARACTER SET

Table 12-1
ASCII Character Set

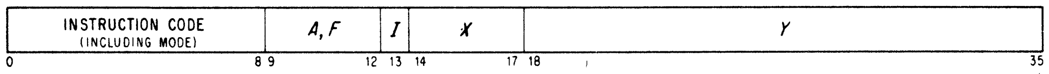
SIXBIT	Character	ASCII 7-Bit†	SIXBIT	Character	ASCII 7-Bit†	Character	ASCII 7-Bit†
00	Space	040	40	@	100	\	140
01	!	041	41	A	101	a	141
02	"	042	42	B	102	b	142
03	#	043	43	C	103	c	143
04	$	044	44	D	104	d	144
05	%	045	45	E	105	e	145
06	&	046	46	F	106	f	146
07	'	047	47	G	107	g	147
10	(	050	50	H	110	h	150
11	)	051	51	I	111	i	151
12	*	052	52	J	112	j	152
13	+	053	53	K	113	k	153
14	,	054	54	L	114	l	154
15	-	055	55	M	115	m	155
16	.	056	56	N	116	n	156
17	/	057	57	O	117	o	157
20	0	060	60	P	120	p	160
21	1	061	61	Q	121	q	161
22	2	062	62	R	122	r	162
23	3	063	63	S	123	s	163
24	4	064	64	T	124	t	164
25	5	065	65	U	125	u	165
26	6	066	66	V	126	v	166
27	7	067	67	W	127	w	167
30	8	070	70	X	130	x	170
31	9	071	71	Y	131	y	171
32	:	072	72	Z	132	z	172
33	;	073	73	[	133	{	173
34	<	074	74	\	134	|	174
35	=	075	75	]	135	}	175
36	>	076	76	↑	136	~	176
37	?	077	77	←	137	Delete	177

†FORTRAN IV also accepts the following control codes in 7-bit ASCII:

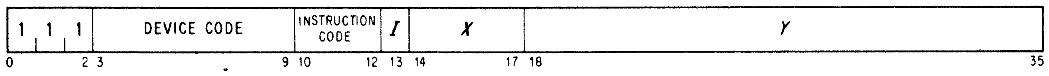
Horizontal Tab	011	Carriage Return	015
Line Feed	012	Form Feed	014

12.2 PDP-10 WORD FORMATS

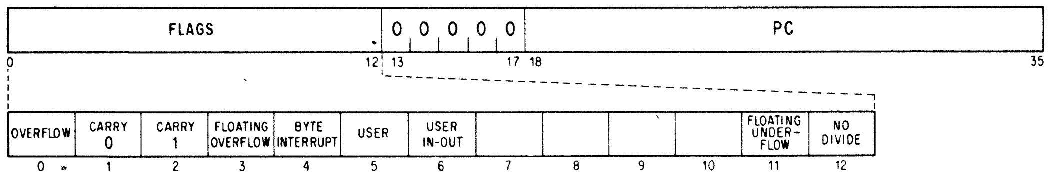
BASIC INSTRUCTIONS



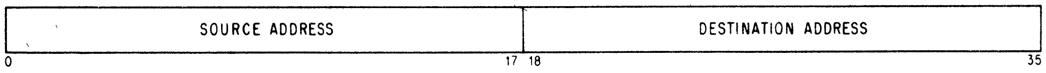
IN-OUT INSTRUCTIONS



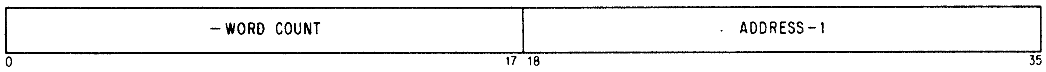
PC WORD



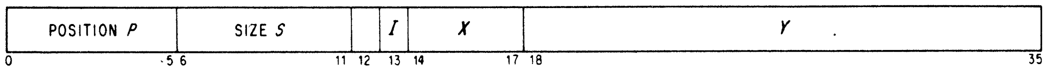
BLT POINTER {XWD}



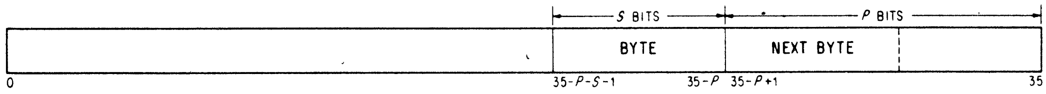
BLKI/BLKO POINTER, PUSHDOWN POINTER, DATA CHANNEL CONTROL WORD {IOWD}



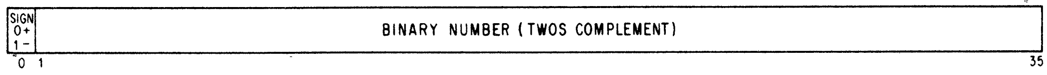
BYTE POINTER



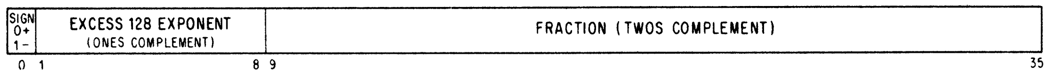
BYTE STORAGE



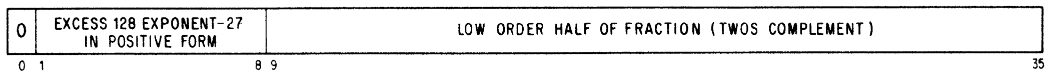
FIXED POINT OPERANDS



FLOATING POINT OPERANDS


SIGN 0+ 1–
EXCESS 128 EXPONENT (ONES COMPLEMENT)
FRACTION (TWOS COMPLEMENT)
0 1
8 9
35

LOW ORDER WORD IN DOUBLE LENGTH FLOATING POINT OPERANDS


0
EXCESS 128 EXPONENT-27 IN POSITIVE FORM
LOW ORDER HALF OF FRACTION (TWOS COMPLEMENT)
0 1
8 9
35

12.3 FORTRAN INPUT/OUTPUT

In addition to the arithmetic functions, the PDP-10 FORTRAN IV library (LIB40) contains several subprograms which control FORTRAN IV I/O operations at runtime. The I/O subprograms are compatible with the PDP-10 Monitors.

In general FORTRAN IV I/O is done with double buffering unless the user has either specified otherwise through calls to IBUFF and OBUFF or is doing random access I/O to the disk. In these cases, single buffers are used. The standard buffer sizes for the devices normally available to the user are given in Table 12-2. Note that the devices and buffer sizes are determined by the Monitor and may be changed by a particular installation. Also a user may specify buffer sizes for magtape operations through the use of IBUFF and OBUFF.

The logically first device in a FORTRAN program is initialized on software I/O channel one, the second on software I/O channel two, and so forth. Software I/O channel 0 is reserved for error message and summary output. The SIXBIT name of the device that is initialized on channel N can be found in a dynamic device table at location DYNDV. + N. A device may be initialized for input and output on the same I/O channel. Devices are initialized only once and are released through either the CALL [SIXBIT/EXIT/] executed at the end of every FORTRAN program or the LIB40 subroutine RELEAS.

Table 12-2
PDP-10 FORTRAN IV Standard Peripheral Devices

Name	Mnemonic	Input/Output		Buffer Size In Words	Operation
		Formatted	Unformatted		
Card Punch	CDP	Yes	Yes	26	WRITE
Card Reader	CDR	Yes	Yes	28	READ
Disk (includes disk packs and drums)	DSK	Yes	Yes	128	READ/WRITE
DECtapes	DTA	Yes	Yes	127	READ/WRITE
Line Printer	LPT	Yes	No	26	WRITE
Magtape	MTA	Yes	Yes	128	READ/WRITE
Plotter	PLT	Yes	Yes	36	WRITE
Paper Tape Punch	PTP	Yes	Yes	33	WRITE
Paper Tape Reader	PTR	Yes	Yes	33	READ
Pseudo Teletype	PTY	Yes	No	17	READ/WRITE
Teletype - User	TTY	Yes	No	17	READ/WRITE
Teletype - Console	CTY	Yes	No	17	READ/WRITE

12.3.1 Logical and Physical Peripheral Device Assignments

Logical and physical device assignments are controlled by either the user at runtime or a table called DEVTB. The first entry in DEVTB. is the length of the table. Each entry after the first is a sixbit ASCII device name. The position in the table of the device name corresponds to the FORTRAN logical number for that device. For example, in Table 12-3, magnetic tape 0 is the 16th entry in DEVTB. Therefore, the statement

```
WRITE (16, 13)A
```

refers to magnetic tape 0. The last five entries in DEVTB. correspond to the special FORTRAN statements READ, ACCEPT, PRINT, PUNCH, and TYPE. Any device assignments may be changed by reassembling DEVTB.

If the user gives the Monitor command

```
ASSIGN DSK 16
```

prior to the running of his program, a file named FOR16.DAT would be written on the disk. Similarly, the Monitor command

```
ASSIGN LPT 16
```

causes output to go to the line printer.

12.3.2 DECtape and Disk Usage

12.3.2.1 Binary Mode - In binary mode, each block contains 127 data words, the first of which is a record control word of the form:

w	n

where w is the word count specifying the number of FORTRAN data words in the block (126 for a full block) and n is 0 in all but the last block of a logical record, in which case n is the number of blocks in the logical record. (A logical record contains all the data corresponding to one READ or WRITE statement.)

12.3.2.2 ASCII Mode - In ASCII mode, blocks are packed with as many full lines (a line is a unit record as specified by a format statement) as possible. Lines always begin with a new word. If a line terminates in the middle of a word, the word is filled out with null characters and the next line begins with the next word. Lines are not split across blocks.

Table 12-3
Device Table for FORTRAN IV

```
TITLE     DEVTB  V.017
SUBTTL     1-APR-69

ENTRY      DEVTB.,DEVND.,DEVLS.,DVTOT.
ENTRY      MTABF.,MBFBG.,TABPT.,TABP1.
ENTRY      MTACL.,DATTB.,NEG1.,NEG2.,NEG3.,NEG5.
P=17

DEVTB.:    EXP       DEVND.-.        ;NO. OF ENTRIES
                                     ;LOGICAL#/FILENAME/DEVICE
           SIXBIT    .DSK.           ; 1 FOR01.DAT     DISC
CORPOS:    SIXBIT    .CDR.           ; 2 FOR02.DAT     CARD READER
LPTPOS:    SIXBIT    .LPT.           ; 3 FOR03.DAT     LINE PRINTER
           SIXBIT    .CTY.           ; 4 FOR04.DAT     CONSOLE TELETYPE
TTYPOS:    SIXBIT    .TTY.           ; 5 FOR05.DAT     USER TELETYPE
           SIXBIT    .PTR.           ; 6 FOR06.DAT     PAPER TAPE READER
PTPPOS:    SIXBIT    .PTP.           ; 7 FOR07.DAT     PAPER TAPE PUNCH
           SIXBIT    .DIS.           ; 8 FOR08.DAT     DISPLAY
           SIXBIT    .DTA1.          ; 9 FOR09.DAT     DECTAPE
           SIXBIT    .DTA2.          ; 10 FOR10.DAT
           SIXBIT    .DTA3.          ; 11 FOR11.DAT
           SIXBIT    .DTA4.          ; 12 FOR12.DAT
           SIXBIT    .DTA5.          ; 13 FOR13.DAT
           SIXBIT    .DTA6.          ; 14 FOR14.DAT
           SIXBIT    .DTA7.          ; 15 FOR15.DAT
           SIXBIT    .MTA0.          ; 16 FOR16.DAT    MAGNETIC TAPE
           SIXBIT    .MTA1.          ; 17 FOR17.DAT
           SIXBIT    .MTA2.          ;  18 FOR18.DAT
           SIXBIT    .FORTR.         ; 19 FORTR.DAT    ASSIGNABLE DEVICE, FORTR

           SIXBIT    .DSK0.          ; 20 FOR20.DAT    DISK
           SIXBIT    .DSK1.          ; 21 FOR21.DAT
           SIXBIT    .DSK2.          ; 22 FOR22.DAT
           SIXBIT    .DSK3.          ; 23 FOR23.DAT
           SIXBIT    .DSK4.          ; 24 FOR24.DAT
           SIXBIT    .DEV1.          ; 25 FOR25.DAT    ASSIGNABLE DEVICES
           SIXBIT    .DEV2.          ; 26 FOR26.DAT
           SIXBIT    .DEV3.          ; 27 FOR27.DAT
           SIXBIT    .DEV4.          ; 28 FOR28.DAT
DEVLS.:    SIXBIT    .DEV5.          ; 29 FOR29.DAT      V.006
           SIXBIT    .REREAD.        ;        -6       REREAD
           SIXBIT    .CDR.           ;        -5       READ
           SIXBIT    .TTY.           ;        -4       ACCEPT
           SIXBIT    .LPT.           ;        -3       PRINT
           SIXBIT    .PTP.           ;        -2       PUNCH
DEVND.:    SIXBIT    .TTY.           ;        -1       TYPE
```

12.3.2.3 File Names - File names may be declared for DECtapes or the disk through the use of the library subprograms IFILE and OFILE. In order to make an entry of the file name FILE1 on unit u, the following statement could be used:

```
CALL OFILE (u,FILE1)
```

Similarly, the following statements might be used to open the file, RALPH, for reading:

```
RALPH=5HRALPH
CALL IFILE(u,RALPH)
```

After writing a file, the END FILE u statement must be given in order to close the current file and allow for reading or writing another file or for reading or rewriting the same file. If no call to IFILE or OFILE has been given before the execution of a READ or WRITE referencing DECtape or the disk the file name FORnn.DAT is assumed where nn is the FORTRAN logical number used in the I/O statement that references device nn.

The FORTRAN programmer can make logical assignments such that each device has its own unique file as intended, but each can be on the DSK. In order to use the devices available, the programmer can make assignments at run time and assign the DSK to those not available.

For example, the FORTRAN logical device numbers, e.g., 1 = DSK, 2 = CDR, 3 = LPT, are used in the file name. The written file names are FOR01.DAT, FOR02.DAT, etc. The same is true for READ. For example, a WRITE (3, 1) A, B, C, in the FORTRAN program generates the file name FOR03.DAT on the DSK if the DSK has been assigned LPT or 3 prior to running the program. (Note: REREAD rereads from the file belonging to the device last referenced in a READ statement, not FOR-6.DAT, as usual.) The programmer must, of course, realize his own mistake in assigning the DSK as the TTY in the case that FORSE tries to type out error messages or PAUSE messages.

More than one DSK File may be accessed, without making logical assignments at runtime, by using logical device numbers 1, and 20 through 24 in the FORTRAN program. Logical device number 19 refers to logical device FORTR which must be assigned at runtime and accesses file name FORTR.DAT to maintain compatibility with the past system of default file name FORTR.DAT. In all cases when the operating system fails to find a file specified, an attempt will be made to read from file FORTR.DAT as before.

The magnetic tape operation REWIND is simulated on DECtape or the disk. Thus, a program which uses READ, WRITE, END FILE, and REWIND for magnetic tape need only have the logical device number changed or assigned to a MTA at runtime in order to perform the proper input/output sequences on DECtape or the disk.

12.3.3 Magnetic Tape Usage

Magnetic tape and disk/DECtape I/O are different in the following ways. When a READ is issued, a record is read in for both magnetic tape and disk. If a WRITE is then issued, the next sequential record is written on magnetic tape but not on disk. When one or more READs have been executed on a disk file and a WRITE is issued, the file is closed prior to the WRITE and then reopened. The WRITE causes the writing over of the first logical record in that file.

12.3.3.1 Binary Mode - The format of binary data on magnetic tape is similar to that for DECtape except that the physical record size depends on the magnetic tape buffer size assigned in the Time-Sharing Monitor or by IBUFF/OBUFF (see Section 8.2.2). Normally, the buffer size is set at either 129 or 257 words so that either 128 or 256 word records are written (containing a control word and 127 or 255 FORTRAN data words).

The first word, control word, of each block in a binary record contains information used by the operating system. The left half of the first word contains the word count for that block. The right half of the first word contains a null character except for the last block in a logical record. In this case, the right half of the first word contains the number of blocks in the logical record.

12.3.3.2 ASCII Mode - The format for ASCII data is the same as that used on DECtape.

12.3.3.3 Backspacing and Skipping Records - Both the BACKSPACE u and SKIP RECORD u statements are executed on a logical basis for binary records and on a line basis for ASCII records.

a. Binary Mode - Both BACKSPACE and SKIP RECORD space magnetic tape physically over one (1) logical record; i.e., the result of one WRITE (u) statement.

b. ASCII Mode - ASCII records are packed, that is WRITE (u, f) statements do not cause physical writing on the tape until the output buffers are full or a BACKSPACE, END FILE, or REWIND command is executed by the program. BACKSPACE and SKIP RECORD on ASCII record space over one (1) line.

c. BACKSPACE and SKIP RECORD following WRITE ASCII commands.

   (1) BACKSPACE closes the tape, writes 2 EOF's (tapemark) and backspaces over the last line.

   (2) SKIP RECORD cannot be used during a WRITE operation. This is an input function only.

MOV { E | e Negative | e Magnitude | e Swapped } — { to AC | Immediate to AC | to Memory | to Self }

Half word { Right | Left } to { Right | Left } { no effect | Ones | Zeros | Extend sign } — { to AC | Immediate to AC | to Memory | to Self }

BLock Transfer

EXCHange AC and memory

use present pointer | Increment pointer } and { LoaD Byte into AC | DePosit Byte in memory }

Increment Byte Pointer

PUSH down | POP up } { ~ | and Jump }

SET to { Zeros | Ones | Ac | Memory | Complement of Ac | Complement of Memory } — to { AC | AC Immediate | Memory | Both }

AND | inclusive OR } { ~ | with Complement of Ac | with Complement of Memory | Complements of Both } — to { AC | AC Immediate | Memory | Both }

Inclusive OR | eXclusive OR | EQuiValence } — to { AC | AC Immediate | Memory | Both }

SKIP if memory | JUMP if AC } — if { never | Less | Equal | Less or Equal | Always | Greater | Greater or Equal | Not equal }

Add One to | Subtract One from } { memory and Skip | AC and Jump } if { never | Less | Equal | Less or Equal | Always | Greater | Greater or Equal | Not equal }

Compare Ac { Immediate | with Memory } and skip if AC — { never | Less | Equal | Less or Equal | Always | Greater | Greater or Equal | Not equal }

Add One to Both halves of AC and Jump if { Positive | Negative }

ADD | SUBtract | • MULtiply | Integer MULtiply | DIVide | Integer DIVide } — { ~ | Immediate | to Memory | to Both }

Floating AdD | Floating SuBtract | Floating MultiPly | Floating DiVide } — { ~ | Long | to Memory | to Both }; and Round — { ~ | Immediate | to Memory | to Both }

Floating SCale

Double Floating Negate

Unnormalized Floating Add

Arithmetic SHift | Logical SHift | ROTate } { ~ | Combined }

Jump { to SubRoutine | and Save Pc | and Save Ac | and Restore Ac | if Find First One | on Flag and CLear it | on OVerflow (JFCL 10,) | on CaRrY 0 (JFCL 4,) | on CaRrY 1 (JFCL 2,) | on CaRrY (JFCL 6,) | on Floating OVerflow (JFCL 1,) | and ReSTore | and ReSTore Flags (JRST 2,) | and ENable PI channel (JRST 12,) }

HALT (JRST 4,)

eXeCuTe

DATA | BLocK } — { In | Out }

CONditions — { In | Out }; — in and Skip if { all masked bits Zero | some masked bit One }

Test AC { with Direct mask | with Swapped mask | Right with E | Left with E } { No modification | set masked bits to Zeros | set masked bits to Ones | Complement masked bits } and skip { never | if all masked bits Equal 0 | if Not all masked bits equal 0 | Always }

APPENDIX A
THE SMALL FORTRAN IV COMPILER

This compiler runs in 5.5K of core, and to the user, is identical to the large compiler, with the exception of the following language differences. Operating procedures are given in the Systems User's Guide (DEC-10-NGCC-D).

Language Differences

The IMPLICIT, DATA, and NAMELIST statements are not recognized; constant strings are not collapsed (for example, A=5*3 will not be treated as A=15).

Book 6

Demonstration Programs

The following demonstration programs illustrate the flexibility of the PDP-10 software system. Each demonstration is aimed at a specific class of user and should be studied with this in mind.

Demonstration #1 is an elementary FORTRAN main program and subroutine. It is intended for a beginning programmer interested in creating and editing files with LINED. The subroutine calculates the sum of the square of twenty numbers and returns the answer to the main program. The main program prints this answer on the Teletype. A bug is found in the subroutine and corrected with elementary LINED commands.

Demonstration #2 is a FORTRAN program which fits a curve to a set of points by the method of least squares. LINED is used to edit and debug the program.

This demonstration was created for the applications programmer or the engineer with some mathematical background. There are four sections within the program; each section can be examined separately without detracting from the reader's comprehension.

Demonstration #3 is an advanced example of the procedure used for creating a FORTRAN main program and a MACRO-10 subprogram. It is intended primarily for the experienced systems programmer.

The two programs are inputted using TECO (Text Editor and Corrector) and then translated and executed together. A listing file is created for each program in case a bug is encountered during execution. Since a bug is found, the listings, along with DDT (the Dynamic Debugging Technique program), are used to debug the program. The erroneous program is then corrected with TECO and saved on the disk for later use. For a complete description of DDT, see the PDP-10 Reference Handbook.

Demonstration #4 is an advanced demonstration of a FORTRAN subroutine that is used to return a random number, and a FORTRAN testing program that tests the accuracy of the subroutine. This demonstration is intended for the experienced programmer familiar with DDT and TECO.

The two programs are inputted using TECO and then executed together. Errors are detected by the FORTRAN compiler and corrected with advanced TECO commands. Execution is again attempted, but now the program returns incorrect results. Cross-reference listings, along with advanced DDT commands, are used to debug the program. The program is then permanently corrected with TECO and saved on the disk for future use.

Demonstration #1

```
 ↑C

.LOGIN
JOB 1Ø      4SP74G
#27,235
PASSWORD:
1525     Ø5-MAR-7Ø      TTY25
PLEASE DELETE ANY FILES THAT ARE NOT NEEDED FROM YOUR
DISK AREA...



.CREATE MAIN.F4


*I
ØØØ1Ø          TYPE 69
ØØØ2Ø    69    FORMAT(' THIS PROGRAM PRINTS THE SUM OF THE SQUARES')
ØØØ3Ø          CALL SUB1(ISUM)
ØØØ4Ø          TYPE 7Ø,ISUM
ØØØ5Ø    7Ø    FORMAT(' THE SUM OF THE SQUARES IS',3X,I5)
ØØØ6Ø          END
ØØØ7Ø    $


*E
*↑C



.CREATE SUB1.F4


*I
ØØØ1Ø          SUBROUTINE SUBR(J)
ØØØ2Ø          J=Ø
ØØØ3Ø    .     DO 1ØØ I=1,2Ø
ØØØ4Ø          J=J+I**2
ØØØ5Ø     1ØØ  CONTINUE
ØØØ6Ø          RETURN
ØØØ7Ø    $

*E
*↑C



.EXECUTE MAIN.F4,SUB1.F4
FORTRAN:  MAIN.F4
FORTRAN:  SUB1.F4
LOADING

ØØØØØ1 UNDEFINED GLOBALS
```

Establish communication with the monitor by typing ↑C while depressing the CTRL key. Begin the login procedure by typing the monitor command LOGIN followed by a carriage-return. The monitor responds with your job number and the monitor name and version number. The login program requests your identification by typing the number sign (#). Type in your project programmer numbers, followed by a carriage-return. The login program requests your password. Type it in; to maintain password security, it is not printed. If your identification matches the identification stored in the system, the monitor responds with the time, date, Teletype number, message of the day (if any), and a period.

CREATE a new disk file with LINED. Call the new file MAIN.F4.

Command to LINED to insert line numbers starting with 10 and incrementing by 10.

Statements of the FORTRAN main program.

The Altmode ends the insert.

Command to LINED to end the creation of the file and to write the file on the disk.

Return to monitor.

CREATE a disk file for the subroutine. Call the file SUB1.F4.

Command to LINED to insert line numbers starting with 10 and incrementing by 10.

Statements of the FORTRAN subroutine.

Altmode ends the insert.

LINED command to end the creation of the file and to write the file on the disk.

Return to monitor.

Request execution of the two programs created.

FORTRAN reports its progress.

```
SUB1        000156
?

LOADER 4K CORE
?EXECUTION DELETED

EXIT
↑C

.EDIT SUB1.F4
*P10
00010            SUBROUTINE SUBR(J)
*I10
00010'           SUBROUTINE SUB1(J)
00020   $


*E
*↑C

.EXECUTE MAIN.F4,SUB1.F4
FORTRAN:  SUB1.F4
LOADING

LOADER 4K CORE
EXECUTION

THIS PROGRAM PRINTS THE SUM OF THE SQUARES
THE SUM OF THE SQUARES IS    2870
EXIT
↑C

.KJOB
CONFIRM: K
JOB 10, USER [27,235]  LOGGED OFF TTY25    1540  5-MAR-70
DELETED ALL 6 FILES (INCLUDING UFD, 7. DISK BLOCKS)
RUNTIME 0 MIN, 04.95 SEC
```

There is no subroutine named SUB1

This includes the space for the Loader. No execution was done.

Ask to edit SUB1.F4

Type line 10 on the Teletype.

Insert a new line 10.

Request execution. Only the subroutine is recompiled since it has been edited.

Both MAIN and SUB1 are loaded.

Execution begins.

Execution ends.

Issue KJOB command to log off the system. Answer the CONFIRM: message with K and a carriage-return if you wish to delete all files you have created. See the following demonstrations for other options available with the KJOB command.

Demonstration #2

```
↑C
.LOGIN
JOB 7    DEC PDP-1Ø #4Ø 4561H PR

#27,2Ø
PASSWORD:


Ø927  29-OCT-69   TTY3




.CREATE SAMPLE.F4
*I10
```

Introduction

Given a set of n sample points $[(x_1,y_1),(x_2,y_2)\ldots(x_n,y_n)]$ this demonstration program calculates the coefficients of the quadratic equation, $a_1+a_2x+a_3x^2)$ such that the value of a_1 a_2,a_3 minimizes the expression

$$\sum_{i=1}^{n}[(a_1 + a_2x_i + a_3x_i^2) - y_i]^2$$

This method of minimizing a_1,a_2,a_3 is called <u>least squares.</u>

The program logically divides into four sections. Section 1 sets up a set of simultaneous equations (the normal equations); section 2 solves the equations for a_1, a_2, a_3. The polynomial approximation $(a_1+a_2x+a_3x^2)$ and the sample points are graphed in section 3. In section 4, a subroutine evaluates 3x3 determinates.

The programmer logs into the system by typing LOGIN (abbreviated LOG); the monitor responds by typing the job number assigned to the programmer and a version number of the monitor. Following the printout of a # symbol, the programmer types his project-programmer number. The monitor types PASSWORD: and awaits an entry. The programmer then types his password (echo typeout suppressed). If the password and project-programmer number match correctly with the password and project-programmer number stored in the system, the monitor types out the time, the date and the number of the Teletype. The monitor may also type out some other information (the message of the day) before awaiting a command.

The monitor command: CREATE SAMPLE.F4
calls in LINED and opens file SAMPLE for creation. The extension F4 marks the file as FORTRAN. The monitor responds to the CREATE command by typing an * indicating that a LINED instruction is now to be typed. The LINED command I10 will begin line numbering at 10, incrementing by 10 for each new line.

An explanation of the mathematical calculafions is given for each section. Note: not all variables which appear in the explanation appear in the program. When the same variables occur in both the explanation and program, they occur in lower case in the explanation and, by necessity, occur in upper case in the program.

```
C         THIS PROGRAM FITS A QUADRATIC POLYNOMIAL
C         ('THE APPROXIMATION POLYNOMIAL') TO A SET OF
C         SAMPLE POINTS BY THE METHOD OF LEAST SQUARES.
C         ***SECTION 1***
C         THE ELEMENTS OF Z,X,IX,TEMP ARE INITIALIZED
C         TO ZERO BY THE DIMENSION STATEMENT.
          DIMENSION Z(3,4),X(9,4),IX(9,4),A(4),TEMP(3)
          COMMON Z
          TYPE 500
C         N IS THE NUMBER OF SAMPLE POINTS. N CAN NOT
C         BE GREATER THAN 9 (THIS RESTRICTION IS IMPOSED
C         BY FORMAT 510). FORMAT 500 WILL BE TYPED ON
C         THE TELETYPE TO INDICATE WHEN THE USER IS
C         TO ENTER THE DATA FOR N.
          ACCEPT 510,N
C         FORMAT 520 WILL BE TYPED ON THE TELETYPE TO
C         INDICATE WHEN THE USER IS TO ENTER THE SAMPLE
C         POINTS.
          TYPE 520
          ACCEPT 530,(X(I,2),X(I,4),I=1,N)
          DO 10 I=1,N
C         THE TWO STATEMENTS BELOW ROUND THE X AND Y
C         COORDINATES OF THE SAMPLE POINTS TO THE
C         NEAREST INTEGER. IX(I,1) AND IX(I,2) WILL
C         BE USED FOR GRAPHING.
          IX(I,1)=IFIX(X(I,2)+.5)
          IX(I,2)=IFIX(X(I,4)+.5)
          X(I,1)=1
10        X(I,3)=X(I,2)**2
          DO 40 I=1,3
          DO 30 K=1,N
          DO 20 L=1,4
20        Z(I,L)=Z(I,L)+X(K,L)*X(K,I)
30        CONTINUE
40        CONTINUE
500       FORMAT(1X,'NUM. OF POINTS;FORMAT(I1);2<N<10'/)
510       FORMAT(I1)
520       FORMAT(1X,'POINTS;FORMAT(9(F4.1,F4.1,3X))'/)
530       FORMAT(9(F4.1,F4.1,1X))
```

SECTION 1

The array x is set up as the augmented matrix[1] of the equations

$$a_1 + a_2x_1 + a_3x_1^2 = y_1$$
$$a_1 + a_2x_2 + a_3x_2^2 = y_2$$
$$\vdots$$
$$a_1 + a_2x_n + a_3x_n^2 = y_n$$

where (x_1, y_1) is the first sample point, (x_2, y_2) is the second sample point.... The unknowns are a_1, a_2, and a_2.

Three normal equations are needed to fit a quadratic equation. They are:

$$a_1 \sum_{i=1}^{n} 1 + a_2 \sum_{i=1}^{n} x_i + a_3 \sum_{i=1}^{n} x_i^2 = \sum_{i=1}^{n} y_i$$

$$a_1 \sum_{i=1}^{n} x_i + a_2 \sum_{i=1}^{n} x_i^2 + a_3 \sum_{i=1}^{n} x_i^3 = \sum_{i=1}^{n} y_i x_i$$

$$a_1 \sum_{i=1}^{n} x_i^2 + a_2 \sum_{i=1}^{n} x_i^3 + a_3 \sum_{i=1}^{n} x_i^4 = \sum_{i=1}^{n} y_i x_i^2$$

The normal equations are calculated from the x array and stored in the z array.

[1]An augmented matrix is a matrix of a set of equations which have been altered to set the right hand side of the equations equal to zero.

$3a - 4b = 9$ (Altered) $3a - 4b - 9 = 0$
$5c + 6d = 10$ (to) $5c - 6d - 10 = 0$

The matrix is: $\begin{pmatrix} 3 & -4 \\ 5 & 6 \end{pmatrix}$. The augmented matrix is: $\begin{pmatrix} 3 & -4 & -9 \\ 5 & 6 & -10 \end{pmatrix}$

```
C         ***SECTION 2***
C         SUBROUTINE DETR CALCULATES THE DETERMINATE OF
C         ARRAY X. THE VALUE OF THE DETERMINANT IS
C         STORED IN THE ARGUMENT OF DETR.
          CALL DETR(DET)
          DO 70 J=1,3
          DO 50 I=1,3
C         COLUMN J OF ARRAY Z(I,J) IS TEMPORARILY STORED
C         IN ARRAY TEMP. COLUMN J IS REPLACED BY COLUMN 4.
C         THE DETERMINANT OF THE NEW Z MATRIX IS CALCULATED
          TEMP(I)=Z(I,J)
50        Z(I,J)=Z(I,4)
          CALL DETR(DET1)
          DO 60 I=1,3
C         THE DATA IN TEMP IS RESTORED TO COLUMN J.
60        Z(I,J)=TEMP(I)
70        A(J)=DET1/DET
          TYPE 700,(A(J),J=1,3)
700       FORMAT(1X,'A(1)=',F8.5,3X,'A(2)=',F8.5,3X,'A(3)=',F8.5)
```

SECTION 2

The normal equations are of the form:

$$a_1 z_{11} + a_2 z_{12} + a_3 z_{13} = z_{14}$$

$$a_1 z_{21} + a_2 z_{22} + a_3 z_{23} = z_{24}$$

$$a_1 z_{31} + a_2 z_{32} + a_3 z_{33} = z_{34}$$

where:

$$z_{11} = 1$$

$$z_{12} = z_{21} = \sum_{i=1}^{n} x_i$$

$$z_{13} = z_{22} = z_{31} = \sum_{i=1}^{n} x_i^2$$

$$z_{23} = z_{32} = \sum_{i=1}^{n} x_i^3$$

$$z_{33} = \sum_{i=1}^{n} x_i^4$$

$$z_{14} = \sum_{i=1}^{n} y_i$$

$$z_{24} = \sum_{i=1}^{n} y_i x_i$$

$$z_{34} = \sum_{i=1}^{n} y_i x_i^2$$

Cramer's method is used to solve the simultaneous equations. By Cramer's rule

$$a_1 = \frac{\begin{vmatrix} z_{14} & z_{12} & z_{13} \\ z_{24} & z_{22} & z_{23} \\ z_{34} & z_{32} & z_{33} \end{vmatrix}}{\text{det}} \qquad a_2 = \frac{\begin{vmatrix} z_{11} & z_{14} & z_{13} \\ z_{21} & z_{24} & z_{23} \\ z_{31} & z_{34} & z_{33} \end{vmatrix}}{\text{det}}$$

```
C       ***SECTION 3***
C       THE APPROXIMATION POLYNOMIAL AND THE SAMPLE
C       POINTS ARE GRAPHED BETWEEN THE RANGE:
C       -2Ø=<X COORDINATE=<+2Ø
C       -29=<Y COORDINATE=<+29
C       ALL VALUES ARE ROUNDED TO THE NEAREST INTEGER.
C       THE Y AXIS IS PRINTED HORIZONTALLY.
        DO 18Ø IL=-2Ø,2Ø
C       IL INDEXES THE X COORDINATE AND CONTROLS
C       SPACING ALONG THE X AXIS.
        IF (IL) 9Ø,8Ø,9Ø
C       IF IL(THE X COORDINATE) EQUALS Ø THEN THE Y
C       AXIS IS PRINTED.
8Ø      TYPE 54Ø
        GO TO 1ØØ
9Ø      TYPE 55Ø
C       CALCULATE THE Y COORDINATE OF THE APPROXIMATION
C       POLYNOMIAL FOR THE VALUE OF IL. ROUND YCO TO
C       THE NEAREST INTEGER.
1ØØ     YCO=A(1)+A(2)*IL +A(3)*IL**2
        IYCO=IFIX(YCO+.5)
C       CHECK IF IYCO IS WITHIN THE RANGE OF THE GRAPH.
        MTEMP=IABS(IYCO)
        IF (MTEMP-3Ø) 11Ø,13Ø,13Ø
C       SPACE THE PAPER TO THE CORRECT PRINT POSTION
C       AND TYPE THE POINT ON THE APPROXIMATION
C       POLYNOMIAL (REPRESENTED BY A +).
11Ø     DO 12Ø IY=Ø, 3Ø+(IYCO-1)
12Ø     TYPE 56Ø
        TYPE 57Ø
C       CHECK IF THERE EXISTS A SAMPLE POINT WHOSE
C       X COORDINATE EQUALS IL.
13Ø     DO 14Ø J=1,N
        IF(IX(J,1)-IL) 14Ø,15Ø,14Ø
14Ø     CONTINUE
        GO TO 18Ø
        MTEMP=IX(J,2)
15Ø     IF (MTEMP-3Ø) 16Ø,16Ø,18Ø
C       CHECK IF THE X COORDINATE OF THE SAMPLE
C       POINT IS WITHIN THE RANGE OF THE GRAPH.
C       SPACE THE PAPER TO THE CORRECT PRINT POSTION
C       AND TYPE THE SAMPLE POINT (REPRESENTED BY A *).
16Ø     DO 17Ø IY=Ø,3Ø+(IX(J,2)-1)
17Ø     TYPE 56Ø
        TYPE 58Ø
18Ø     CONTINUE
54Ø     FORMAT(16X'-Y...........................+Y')
55Ø     FORMAT(31X,'.')
C       '+' AS THE FIRST 3 CHARACTERS IN A FORMAT
C       STATEMENT INDICATES THAT THE FORMAT CONTAINING
C       THE CHARACTERS IS TYPED ON THE SAME LINE
C       AS THE PREVIOUS FORMAT.
56Ø     FORMAT (1H+,1H ,$)
C       $ AS THE LAST CHARACTER OF A FORMAT SUPPRESSES
C       THE CARRIAGE RETURN.
57Ø     FORMAT (1H+,1H+)
```

$$a_3 = \frac{\begin{vmatrix} z_{11} & z_{12} & z_{14} \\ z_{21} & z_{22} & z_{24} \\ z_{31} & z_{32} & z_{34} \end{vmatrix}}{\text{det}} \qquad \text{det} = \begin{vmatrix} z_{11} & z_{12} & z_{13} \\ z_{21} & z_{22} & z_{23} \\ z_{31} & z_{32} & z_{33} \end{vmatrix}$$

The $| \; |$ symbol denotes the determinate. The subroutine DETR calculates the determinate of array z; since the array is stored in the common area (by the COMMON statement), the subroutine DETR will evaluate the determinate of the current value of z as calculated in the main program.

SECTION 3

The approximation polynomial and the sample points are graphed. The x axis is printed vertically and the y axis is printed horizontally.

To control vertical spacing of the graph, the 0 print position of the line is used[1]. Any character placed in position 0 will not be printed; however, certain characters in position 0 control the line feed, i.e., a + in position 0 suppresses the line feed.

To control horizontal spacing a DO-loop is used. A TYPE statement which prints one space is placed within a DO-loop. The loop spaces to the correct position for typing a character on the graph. The format which types the space is:

```
FORMAT (1H+,1H ,$)
```

The dollar sign ($) suppresses the carriage return so that the spaces will be printed in consecutive print positions on one line. Without the dollar sign, each space would be printed in print position 1 of the line. (Recall that a + suppresses the line feed.)

[1]The Teletype line is 72 characters wide (print position 0 – 71).

```
C          ***SECTION 4***
C          DETR CALCULATES THE DETERMINATE USING MINORS.
           SUBROUTINE DETR(DEE)
           COMMON Z
           DIMENSION Z(3,4),B(4)
           DEE=Ø
           DO 3ØØ I=1,3
           K=1
           DO 25Ø M=1,3
C          DELETE THE COLUMN AND ROW OF THE SCALAR MULTIPLIER
C          ,Z(1,I). STORE THE ELEMENTS OF THE 2X2
C          DETERMINANTS IN B(1),B(2),B(3),AND B(4)
           IF(M-I) 2ØØ,25Ø,2ØØ
2ØØ        B(K)=Z(2,M)
           B(K+2)=Z(3,M)
           K=2
25Ø        CONTINUE
C          THE VALUE OF THE DETERMINANT OF ARRAY Z IS
C          STORED IN DEE.
3ØØ        DEE=(-1)**(1+I)*Z(1,I)*(B(1)*B(4)-B(3)*B(2))+DEE
           RETURN
           END
```

SECTION 4

Subroutine DETR expands by minors to calculate the determinate of array Z.

The following equality holds:

$$\begin{vmatrix} z_{11} z_{12} z_{13} \\ z_{21} z_{22} z_{23} \\ z_{31} z_{32} z_{33} \end{vmatrix} = (-1)^{1+1} z_{11} \begin{vmatrix} z_{22} z_{23} \\ z_{32} z_{33} \end{vmatrix} + (-1)^{1+2} z_{12} \begin{vmatrix} z_{21} z_{23} \\ z_{31} z_{33} \end{vmatrix} + (-1)^{1+3} z_{13} \begin{vmatrix} z_{21} z_{22} \\ z_{31} z_{32} \end{vmatrix}$$

Notice that the new 2 x 2 determinates are formed by deleting the column and row containing the scalar multiplier, i.e., if z_{11} is a scalar multiplier, then row 1 and column 1 are deleted from the new system to form

$$\begin{vmatrix} z_{22} & z_{23} \\ z_{32} & z_{33} \end{vmatrix}$$

The quantity (- 1) is raised to the sum of the row and column numbers which contain the scalar multiplier, i.e., z_{11} is contained in row 1 and column 1 so that the term

$$z_{11} \begin{vmatrix} z_{22} & z_{23} \\ z_{32} & z_{33} \end{vmatrix}$$

is multiplied by $(-1)^{1+1} = 1$.

The program uses array b as temporary storage for elements of the 2 x 2 determinate, i.e., for

$$\begin{vmatrix} z_{22} & z_{23} \\ z_{32} & z_{33} \end{vmatrix} \qquad \begin{array}{ll} B(1) = z_{22} & B(3) = z_{32} \\ B(2) = z_{23} & B(4) = z_{33} \end{array}$$

```
Ø1370    $
*I1141
Ø1141    58Ø      FORMAT(1H+,1H*)
Ø1151"   $
*E
*↑C




.EXECUTE SAMPLE.F4
FORTRAN:  SAMPLE.F4
       Ø1170           SUBROUTINE DETR(DEE)
                                      ↑

                      Ø-2 SUBROUTINE IS NOT A SEPARATE PROGRAM
      Ø1180            COMMON Z
                                 ↑

                      A-2 ATTEMPT TO ENTER A VARIABLE INTO COMMON TWICE
       Ø1190           DIMENSION Z(3,4),B(4)
                                    ↑

                      I-2 ARRAY NAME ALREADY IN USE
       Ø128Ø    2ØØ    B(K)=Z(2,M)
                        ↑

                      S-1  SYNTAX
       Ø129Ø           B(K+2)=Z(3,M)
                        ↑

                      S-1  SYNTAX
MAIN. ERRORS DETECTED: 5

? TOTAL ERRORS DETECTED:5
LOADING

ØØØØØ2 UNDEFINED GLOBALS

B       ØØØ556
DETR    ØØØ276
?

LOADER 5K CORE
?EXECUTION DELETED

EXIT
↑C
```

This system of determinants reduces further:

$$(-1)^{1+1} z_{11} \begin{vmatrix} z_{22} z_{23} \\ z_{32} z_{33} \end{vmatrix} + (-1)^{1+2} z_{12} \begin{vmatrix} z_{21} z_{23} \\ z_{31} z_{33} \end{vmatrix} + (-1)^{1+3} z_{13} \begin{vmatrix} z_{21} z_{22} \\ z_{31} z_{32} \end{vmatrix}$$

$$= z_{11} (z_{22} z_{33} - z_{32} z_{23}) - z_{12} (z_{21} z_{33} - z_{31} z_{23}) + z_{13} (z_{21} z_{32} - z_{31} z_{22})$$

To return to LINED command mode, the programmer types an ALTMODE ($) and LINED returns a * . Upon reviewing the listing, the programmer discovers that a line has been omitted after 1140 . A new line, line 1141, is inserted for the omitted coding. After line 1141 is typed, LINED responds with line number 1151. The double quote after the line number 1151 indicates a pre-existing line has been skipped (line 1150). The programmer closes the file by typing an E. A control C (↑C) returns the program to monitor level.

The Monitor command: EXECUTE SAMPLE.F4 initiates loading and execution of SAMPLE.F4. Five errors are detected in SAMPLE.F4, however, terminating the program before execution. The errors are generated by the omission of an END statement. This omission has led the FORTRAN compiler to consider the main program and the subroutine as one program; hence the messages: "ATTEMPT TO ENTER VARIABLE INTO COMMON TWICE ", "UNDEFINED GLOBALS", and "ARRAY ALREADY IN USE." Since the DIMENSION statement is in error, any use of array b generates additional errors.

```
.EDIT SAMPLE.F4
*I1145
Ø1145           END
Ø1155"  $
*E
*↑C

.EXECUTE
FORTRAN: SAMPLE.F4
LOADING

LOADER 5K CORE
EXECUTION

NUM. OF POINTS;FORMAT(I1);2<N<1Ø
4

POINTS;FORMAT(9(F4.1,F4.1,1X))
Ø6.ØØ6.Ø -6.Ø-6.Ø -6.ØØ6.Ø Ø6.Ø-6.Ø

A(1)= Ø.ØØØØØ  A(2)= Ø.ØØØØØ  A(3)= Ø.ØØØØØ
                                     +
                                     +
                                     +
                                     +
                                     +
                                     +
                              ↑O

EXIT
↑C


.EDIT SAMPLE.F4
*I445,5
ØØ445           IF (DET) 2ØØ,19Ø,2ØØ
ØØ45Ø'  2ØØ     DO 7Ø J=1,3
ØØ455   $
*I1142,1
Ø1142           STOP
Ø1143   19Ø     TYPE 59Ø
Ø1144   59Ø     FORMAT(1X,'PROGRAM TERMINATED,DETERMINATE EQUALS Ø')
Ø1145'  $
*E
*↑C

.EXECUTE
FORTRAN:  SAMPLE.F4
LOADING

LOADER 5K CORE
EXECUTION

NUM. OF POINTS;FORMAT(I1);2<N<1Ø
4

POINTS;FORMAT(9(F4.1,F4.1,1X))
Ø6.ØØ6.Ø -6.Ø-6.Ø -6.ØØ6.Ø Ø6.Ø-6.Ø

PROGRAM TERMINATED,DETERMINATE EQUALS Ø
EXIT
↑C
```

The errors can be corrected using LINED. The Monitor command EDIT SAMPLE.F4 calls in LINED and reopens the pre-existing file, SAMPLE.F4. The END statement is inserted on line 1145, the file is closed, and the job is returned to Monitor level. The Monitor command EXECUTE initiates loading and execution of the file named in the previous EXECUTE command (in this case SAMPLE.F4).

The printing of the graph is terminated prematurely by the programmer by typing a control O (↑O).

Examination of the program reveals that division by 0 was attempted. For this data set, DET = 0 (line 560). Therefore, array a was not changed by the division; A(1), A(2), A(3) remain equal to 0. A return is made to LINED level to insert a test for DET = 0. The LINED command I445,5 types line 445 and then increments line number 445 by 5. Similarly I1142,1 types line 1142 and then increments the line number by 1.

The program is executed with the same data set and this time is terminated because DET = 0.

```
•EXECUTE
FORTRAN:  SAMPLE•F4
4
LOADING

LOADER 5K CORE
EXECUTION

NUMBER OF POINTS;FORMAT(I1);2<NUMBER<1Ø

SAMPLE POINTS;FORMAT(9(F4•1,F4,1,1X))
Ø2•1Ø7•8 Ø6•7Ø5•5 -8•9Ø9•8 -1•Ø-1•Ø

A(1)= 2•76598   A(2)= Ø•Ø3ØØ7   A(3)= Ø•Ø8547
                              •
                              •
                              •                          +
                              •                       +
                              •                     +
                              •                  +
                              •               +
                              •              +
                              •            +
                              •          +
                              •        +*
                              •       +
                              •      +
                              •     +
                              •    +
                              •   +
                              •  +
                              •  +
                              •  +
                             *•  +
                -Y•••••••••••••••••••••••••••+Y
                              •  +
                              •  +    *
                              •   +
                              •   +
                              •    +
                              •     +
                              •     *+
                              •       +
                              •         +
                              •           +
                              •            +
                              •              +
                              •                +
                              •                  +
                              •                    +
                              •                       +
                              •                          +
                              •
                              •
                              •

EXIT
↑C
```

Another execution of the program is made with different data set. Since DET $\neq 0$, the graph is printed.

```
.KJOB
CONFIRM:
TYPE ↑C TO ABORT LOG-OUT; OR
TYPE ONE OF THE FOLLOWING (AND CAR RET):

K  TO KILL JOB AND DELETE ALL UNPROTECTED FILES;
L  TO LIST YOUR DISK DIRECTORY; OR,
I  TO INDIVIDUALLY SAVE AND DELETE FILES AS FOLLOWS:

        AFTER EACH FILE NAME IS LISTED, TYPE:
        P  TO SAVE AND PROTECT,
        S  TO SAVE WITHOUT PROTECTING, OR
        CAR RET ONLY TO DELETE.

CONFIRM: I

SAMPLE.REL   <Ø55>   6. BLKS   :P
SAMPLE.BAK   <Ø55>   1Ø. BLKS  :
SAMPLE.F4    <Ø55>   1Ø. BLKS  :P
JOB 7, USER 27, 2Ø OFF TTY3 AT 29-OCT-69
DELETED 1 FILES
SAVED 2 FILES (INCLUDING UFD, 28. DISK BLOCKS)
RUNTIME Ø MIN, 2Ø.29 SEC
.
```

The programmer logs off the Teletype with the Monitor command: KJOB. The Monitor asks for confirmation, CONFIRM: The programmer types a carriage return causing the Monitor to type an explanation and then ask again for a confirmation. Since the programmer typed an I, the Monitor responds with SAMPLE.REL <055 > 6. BLKS:

SAMPLE.REL is the relocatable binary file of the source program SAMPLE.F4. SAMPLE.REL uses six disk blocks (6. BLKS:). The octal number in angular brackets is the protection code. Protection code 055 means the file is sharable (the file can be read from or written on by a programmer with any project-programmer number). Typing a P following the colon changes the protection to 455 meaning that only a programmer logged in under the present project - programmer number can write on the file.

After the programmer types the P and a carriage return, the Monitor prints:

```
SAMPLE.BAK < 055 > 10. BLKS:
```

SAMPLE.BAK is a backup file. The backup file is the next-to-last closed file. In this case, the backup file is the FORTRAN file without the addition of lines 445, 1142, 1143, 1144 and the change of line 450. The programmer deletes file SAMPLE.BAK.

The Monitor types

```
SAMPLE.F4 <055 > 10. BLKS:
```

By typing a P, the programmer saves and protects the current FORTRAN source file. The Monitor then types the terminating message which includes the total number of disk blocks saved (The User File Directory, UFD, uses two blocks).

```
↑C

.LOGIN
JOB 14   4SP74G #40PROD
#27,560
PASSWORD:
0859     23-FEB-70        TTY11

.MAKE MODULO

*IC      ROUTINE TO TYPE IN TWO INTEGERS N1,N2
C        AND TO TYPE OUT N1 MODULO N2.

1        TYPE 5
5        FORMAT(' TYPE N1 AND N2'//)
         ACCEPT 10,N1,N2
10       FORMAT(2I)
         M=MOD(N1,N2)
         TYPE 20,N1,N2,M
20       FORMAT(1X,I5,' MOD 'I5,' IS 'I5//)
         GO TO 1
         END
$$
*EX$$

EXIT
↑C

.MAKE MACSUB.MAC

*I       ;MOD SUBROUTINE WITH STANDARD FORTRAN CALLING SEQUENCE
         ;FORTRAN STATEMT M=MOD(N1,N2)
         ;RESULTS IN CALL TO MOD IN THE FORM
         ;        JSA 16,MOD
         ;        ARG 0,N1
         ;        ARG 0,N2
         ;THUS UPON ENTRY TO MOD, AC16 POINTS TO N1
         ;THE REMAINDER OR "MOD" FUNCTION IS RETURNED IN AC0
         ;AS THE RESULT OF DIVIDING N1 (IN AC17) BY N2.

ENTRY MOD         ;ENTRY POINT FOR LOADER
MOD:     0
         MOVEM 17,SAV17   ;SAVE AC17
         MOVE 17,@(16)    ;PICK UP N1
         IDIV 17,@(16)    ;DIVIDE BY N2 (REMAINDER IN AC0)
         MOVE 17,SAV17    ;RESTORE AC17
         JRA 16,2(16)     ;RETURN TO CALLING PROGRAM
SAV177: 0
         END
$$
*JSSTATEM$IEN$0LT$$
         ;FORTRAN STATEMENT M=MOD(N1,N2)
*16LT$$
SAV177: 0
*5C1D0LT$$
SAV17:  0
*EX$$

EXIT
↑C
```

Place the Teletype in monitor command mode (↑C) and log into the system by typing LOGIN, followed by the prescribed "login" information for your particular system. The monitor responds with time, date, and Teletype number.

MAKE MODULO calls in TECO to create the file MODULO for the FORTRAN IV source program. The text of the program is preceded by the TECO insert command I. The two ALTMODEs ($$) signify the termination of the text to be inserted. It is now possible to edit the text if a typing error is discovered. Since no typing errors were made, type EX$$ to write the file onto your disk area and return control to the monitor.

MAKE MACSUB.MAC creates the file MACSUB.MAC with TECO for the MACRO-10 subprogram. Two errors were made while typing the program. These are corrected with the TECO command strings just before the EX$$ command. The first command string searches for the characters STATEM, then the characters EN are inserted and the corrected line is typed out. The next command string types out the sixteenth line counting from the line previously typed. The third command string deletes the extra 7 and the corrected line is typed out. The EX$$ command then writes the file on your disk area and returns control to the monitor.

```
.EXECUTE/L MODULO,MACSUB.MAC
FORTRAN:  MODULO
MACRO: .MAIN
LOADING

LOADER 4K CORE
EXECUTION



TYPE N1 AND N2

10 3

   10 MOD        3 IS     0

TYPE N1 AND N2

12 8

   12 MOD        8 IS     0

TYPE N1 AND N2

↑C

.TYPE MODULO.LST,MACSUB.LST


MODULO   F40      V013     23-FEB-70        9:14    PAGE 1


                                            C       ROUTINE TO TYPE IN TWO INT
EGERS N1,N2

                                            C       AND TO TYPE OUT N1 MODULO
N2.

1P       BLOCK    0

                                            1       TYPE 5
1M       MOVEI    01,5P
         OUT.     01,777777
         FIN.     00,0

                                            5       FORMAT(' TYPE N1 AND N2'//

)
5P       JRST     2M

         ASCII    (' TY
         ACSII    PE N1
         ASCII     AND
         ASCII    N2'//
         ASCII    )
2M       BLOCK    0
```

The EXECUTE/L MODULO,MACSUB.MAC command causes the system to create binary machine-language files for both the FORTRAN IV main program and the MACRO-10 subprogram. These binary files are then loaded together and execution is begun. Since no extension was specified for the file MODULO, the FORTRAN compiler is the processor used. However, the file MACSUB.MAC has the extension .MAC, thus the MACRO-10 assembler is used for this file. The /L causes listing files to be created for both source files. These listing files may be used if a bug is encountered during execution. The system acknowledges its progress as each step is executed and also types out the total core space needed for loading.

The program seems always to give an answer of zero. The next step is to find the bug and correct it.

The command TYPE MODULO.LST,MACSUB.LST causes the listing files for both the main program and the subprogram to be typed out on the Teletype.

```
                                                     ACCEPT 10,N1,N2
        MOVEI   01,10P
        IN.     01,777774
        DATA.   00,N1
        DATA.   00,N2
        FIN.    00,0

                                              10     FORMAT(2I)
10P     JRST    3M

        ASCII   (2I)
3M      BLOCK   0

                                                     M=MOD(N1,N2)
        JSA     16,MOD
        ARG     00,N1
        ARG     00,N2
        MOVEM   00,M

                                                     TYPE 20,N1,N2,M
        MOVEI   01,20P
        OUT.    01,777777
        DATA.   00,N1
        DATA.   00,N2
        DATA.   00,M
        FIN.    00,0
                                              20     FORMAT(1X,I5,' MOD '15,1' I
S 'I5//)
```

```
20P     JRST    4M

        ASCII   (1X,I
        ASCII   5,' M
        ASCII   OD ' I
        ASCII   5,' I
        ASCII   S 'I5
        ASCII   //)
4M      BLOCK   0

                                                       GO TO 1
        JRST    1P
```

```
                                                       END
        JSA     16,EXIT
MAIN.%  RESET.  00,0
        JRST    1M
```

Demonstration Program #3 continues on next page

```
SUBPROGRAMS

FORSE.
JOBFF
INTO.
INTI.
MOD
EXIT

SCALARS

N1        45
N2        46
M         47



.MAIN     MACRO.V40 09:14 23-FEB-70 PAGE 1
MACSUB.MAC
                                              ;MOD SUBROUTINE WITH STANDARD FO
RTRAN CALLING SEQUENCE
                                              ;FORTRAN STATEMENT M=MOD(N1,N2)
                                              ;RESULTS IN CALL TO MOD IN THE F
ORM
                                              ;        JSA 16,MOD
                                              ;        ARG 0,N1
                                              ;        ARG 0,N2
                                              ;THUS UPON ENTRY TO MOD, AC16 PO
INTS TO N1
                                              ;THE REMAINDER OR "MOD" FUNCTION
 IS RETURNED IN AC0
                                              ;AS THE RESULT OF DIVIDING N1 (I
N AC17) BY N2.

                                       ENTRY MOD        ;ENTRY POINT FOR LOADER
       000000  000000  000000  MOD:    0
       000001  202740  000006'         MOVEM 17,SAV17   ;SAVE AC17
       000002  200776  000000          MOVE 17,@(16)    ;PICK UP N1
       000003  230776  000000          IDIV 17,@(16)    ;DIVIDE BY N2 (R
EMAINDER IN AC0)
       000004  200740  000006'         MOVE 17,SAV17    ;RESTORE AC17
       000005  267716  000002          JRA 16,2(16)     ;RETURN TO CALLI
NG PROGRAM
       000006  000000  000000  SAV17:  0
                                       END
NO ERRORS DETECTED

PROGRAM BREAK IS 000007




.MAIN    MACRO.V40 09:14 23-FEB-70 PAGE 2
MACSUB.MAC       SYMBOL TABLE

MOD              000000' ENT              SAV17            000006'
```

The error seems to be here. In line 000003 a 1 has been omitted between the @ and the (16). A patch is inserted in the program with DDT to check its validity.

```
.DEBUG MODULO,MACSUB.MAC
LOADING

LOADER 4K CORE
EXECUTION



.MAIN$: MOD+3/  IDIV 17,@0(16)  IDIV 17,@1(16)
$G


TYPE N1 AND N2

10 3

   10 MOD       3 IS     1

TYPE N1 AND N2

12 8

   12 MOD       8 IS     4

TYPE N1 AND N2

43762 10822

43762 MOD 10822 IS    474

TYPE N1 AND N2

↑C

.TECO MACSUB.MAC

*JSIDIV 17,@$I1$0LT$$
        IDIV 17,@1(16)  ;DIVIDE BY N2 (REMAINDER IN AC0)
*EX$$

EXIT
↑C


.KJOB
CONFIRM: I
MODULO        <055>   1. BLKS  :S
MACSUB.BAK    <055>   1. BLKS  :
MODULO.LST    <055>   2. BLKS   DELETED
MODULO.REL    <055>   1. BLKS  :S
MACSUB.REL    <055>   1. BLKS  :S
MACSUB.LST    <055>   2. BLKS   DELETED
MACSUB.MAC    <055>   1. BLKS  :S
JOB 14, USER [27,560] LOGGED OFF TTY11  0924 23-FEB-70
DELETED 4 FILES (6. DISK BLOCKS)
SAVED 4 FILES (INCLUDING UFD, 6. DISK BLOCKS)
RUNTIME 0 MIN, 05.54 SEC
```

To load DDT with the programs , type DEBUG MODULO,MACSUB.MAC. Note that the two source files are not translated again since they have not been modified since the previous translation occurred.

Before execution of the program is begun, the system transfers control to DDT. .MAIN$: opens the symbol table for the MACRO-10 subprogram. (.MAIN was automatically assigned by MACRO since no name was specified.) DDT acknowledges the command with a TAB. Then type MOD+3/ to examine the contents of location MOD+3. After DDT responds with the erroneous instruction, simply retype the instruction correctly. The $G then starts the program with the correction included.

The program now returns correct results! It is now necessary to return to TECO to permanently correct the program.

The command TECO MACSUB.MAC opens the already existing file for editing. A command string is then given which inserts the 1 in the proper place, and the correction is typed out.

It is now necessary to log off the system. Type KJOB to accomplish this. When the system types CONFIRM: , type I to individually save or delete your files. The MACSUB.BAK file is a backup file which was created when the TECO MACSUB.MAC command was given. This file contains the file as it was before the command was executed. Since this file is no longer needed, only a carriage return is typed to delete it. The listing files are automatically deleted by the system. An S is typed after the colon to save all the other files on the disk for future use. The system then completes the logout procedure by typing out the number of files deleted and saved, their length in disk blocks, and the total runtime of your job.

Demonstration #4 (Advanced)

```
.LOG
JOB 7    4SP74G
#27,560
PASSWORD:
0902     25-FEB-70       TTY3
SYS 40 WILL BE DOWN FROM 1800-2400 TUE FEB 24......

.MAKE RANDOM

*         SUBROUTINE RANDOM (ARG)
C         RANDOM NUMBER GENERATING SUBROUTINE
C         RETURNS A SINGLE PRECISION FLOATING POINT (REAL)
C         PSEUDO-RANDOM NUMBER UNIFORMLY DISTRIBUTED OVER <0,1>.
C
C         THIS SUBROUTINE ATTEMPTS TO DUPLICATE THE ASSEMBLY
C         LANGUAGE PROGRAM RANDOM USED IN THE PDP-10 REFERENCE HANDBOOK
C         DEMONSTRATION #2
          EQUIVALENCE (T,I)
          INTEGER RNUMB,MAGIC,IT(3)
          DATA RNUMB/1/
          MAGIC = 5**15
C         GET THE TIME IN ASCII HH:MM SS.T
          CALL TIME (I,ARG)
C         USE TIME TO SELECT 1-4 ITERATIONS
          DECODE (5,1,ARG) (IT(K),K=1,3)
1         FORMAT (1X,2I1,1X,I1)
          K=((( IT(1) + 15) / IT(2) ) + IT(3) ) .AND. "3
          DO 2 I=K,0,-1
C         MULTIPLICATIVE RANDOM NUMBER GENERATOR
2         RNUMB = RNUMB * MAGIC
C         CONVERT TO FLOATING POINT FORMAT IN THE RANGE <0,1>,
C         BY CLEARING THE EXPONENT FIELD WITH A "SHIFT" AND SETTING
C         THE EXPONENT TO 0 (EXCESS 200).
          I = ( RNUMB /256 ) .OR. "20000000000
C         NORMALIZE AND STORE RESULT
          ARG = T + 0.0
          RETURN
          END
$$
*HT$$
          SUBROUTINE RANDOM (ARG)
C         RANDOM NUMBER GENERATING SUBROUTINE
C         RETURNS A SINGLE PRECISION FLOATING POINT (REAL)
C         PSEUDO-RANDOM NUMBER UNIFORMLY DISTRIBUTED OVER <0,1>.
C
C         THIS SUBROUTINE ATTEMPTS TO DUPLICATE THE ASSEMBLY
C         LANGUAGE PROGRAM RANDOM USED IN THE PDP-10 REFERENCE HANDBOOK
C         DEMONSTRATION #2.
          EQUIVALENCE (T,I)
          INTEGER RNUMB,MAGIC,IT(3)
          DATA RNUMB/1/
          MAGIC = 5**15
C         GET THE TIME IN ASCII HH:MM SS.T
          CALL TIME (I,ARG)
C         USE TIME TO SELECT 1-4 ITERATIONS
          DECODE (5,1,ARG) (IT(K),K=1,3)
1         FORMAT(1X,2I1,1X,I1)
          K=((( IT(1) + 15 ) / IT(2) ) + IT(3) ) .AND. "3
          DO 2 I=K,0,-1
C         MULTIPLICATIVE RANDOM NUMBER GENERATOR
2         RNUMB = RNUMB * MAGIC
```

LOG (abbreviation for LOGIN) starts the process of logging in to the system. Type the appropriate "login" information for your particular system, and it will respond with the time, date, Teletype number, and the message of the day (if there is a message to be given).

MAKE RANDOM creates the file RANDOM with TECO for the FORTRAN random number generating subroutine. The TECO insert command (I) is not needed since the first character to be input is a TAB. The TAB causes it, along with the rest of the text up to the two ALTMODEs ($$), to be inserted into the buffer.

HT$$ causes TECO to type the entire buffer on the Teletype. This is used to check the previously inserted text for errors.

```
C       CONVERT TO FLOATING POINT FORMAT IN THE RANGE <Ø,1>
C       BY CLEARING THE EXPONENT FIELD WITH A "SHIFT" AND SETTING
C       THE EXPONENT TO Ø (EXCESS 2ØØ).
        I = ( RNUMB /256 ) .OR. "2ØØØØØØØØØØ
C       NORMALIZE AND STORE RESULT
        ARG = T + Ø.Ø
        RETURN
        END
*EX$$

EXIT
↑C

.MAKE TESTIT







*I1Ø    TYPE 1
1       FORMAT(' COMPUTE THE AVERAGE OF ',5X,'.
         9PSEUDO-RANDOM NUMBERS')
        TYPE 2
2       FORMAT(1H+,T25,$)
        ACCEPT 3,I
3       FORMAT(I5)
        T=Ø.Ø
        DO 4 K=1,I
        CALL RANDOM (R)
4       T=T+R
        T=T/I
        TYPE 5,T
5       FORMAT(1X,'IS: ',F)
        GO TO 1Ø
        END
$$
*HT$$
1Ø      TYPE 1
1       FORMAT(' COMPUTE THE AVERAGE OF ',5X,'.
         9PSEUDO-RANDOM NUMBERS')
        TYPE 2
2       FORMAT(1H+,T25,$)
        ACCEPT 3,I
3       FORMAT(I5)
        T=Ø.Ø
        DO 4 K=1,I
        CALL RANDOM (R)
4       T=T+R
        T=T/I
        TYPE 5,T
5       FORMAT(1X,'IS: ',F)
        GO TO 1Ø
        END
*EX$$

EXIT
↑C
```

The TECO command EX$$ causes the file to be ended and the contents of the buffer to be read onto your disk area. It then returns control to the monitor.

MAKE TESTIT calls in TECO to open a file for the FORTRAN testing program. The program calls the subroutine to get random numbers. It then averages a specified number of random numbers and, since the random numbers are confined to the range (0,1), the average should be close to 0.5 if the numbers are truly random.

The TECO insert command (I) causes TECO to insert the text of the program up to the two ALTMODEs into the buffer.

The command HT$$ causes the whole buffer to be typed onto your Teletype. This is useful to check for typing mistakes.

EX$$ causes the file to be closed and read onto your disk area. It then returns control to the monitor.

```
.EXECUTE TESTIT,RANDOM




FORTRAN:  TESTIT
********         1        FORMAT(' COMPUTE THE AVERAGE OF ',5X,'.
********                                                       ↑
********                  M-7 UNTERMINATED HOLLERITH STRING
********                    9PSEUDO-RANDOM NUMBERS')
********                    ↑
********                  S-1 SYNTAX
MAIN.   ERRORS DETECTED: 2

? TOTAL ERRORS DETECTED: 2
FORTRAN:  RANDOM
LOADING

LOADER 5K CORE
?EXECUTION DELETED

EXIT
↑C

.TECO TESTIT

*S5X,'.$$
*2R8<1A=$C>$$
39
46
13
1Ø
9
32
57
8Ø


*ØLCDØL-TT$$
1       FORMAT(' COMPUTE THE AVERAGE OF ',5X,'.
        9PSEUDO-RANDOM NUMBERS')



*EG$$
FORTRAN:        TESTIT
LOADING

LOADER 5K CORE
EXECUTION

COMPUTE THE AVERAGE OF ØØØ1Ø.PSEUDO-RANDOM NUMBERS
```

Type EXECUTE TESTIT ,RANDOM to call in the FORTRAN IV compiler to create binary machine-language files (these will be given the extension .REL). The EXECUTE command also causes the two relocatable binary files to be loaded and then executed, if there are no compiler detected errors.

The FORTRAN compiler has detected errors in the main (testing) program. It types out the lines the errors occur in and an arrow at the particular point the error is detected. The total errors for the main program are then typed out. The compiler then translates the file RANDOM and does not detect any errors. The total core requirement for loading is then typed out. The execution is deleted since there were compiler detected errors. Control is then returned to the monitor.

The error messages returned by the FORTRAN compiler seem to indicate that there may be an illegal character in one or both of the two lines which were typed out by the compiler. To check this assumption, type TECO TESTIT which opens the already existing file TESTIT for editing. This command has the added effect of renaming the file, as it was before editing, TESTIT.BAK.. The first command string to TECO searches for the characters 5X,'.. The second string causes the decimal equivalents of the ASCII code of eight consecutive characters, starting with the character ' , to be typed on the Teletype. In this way, it should be possible to see if there is a character among these eight (even a non-printing one) that should not be there.

The 32 is a space, and for this line to be a continuation line it should not be there. Type the TECO command string shown to delete the space and type the corrected lines.

The command EG$$ causes TECO to perform the functions of an EX$$ and then to execute the last compile class command given to the monitor. In this case, it was EXECUTE TESTIT,RANDOM. Since RANDOM has not been changed, it is not recompiled.

Execution of the program is begun and it types the line COMPUTE THE AVERAGE OF PSEUDO-RANDOM NUMBERS. The program will then space over to the blank spaces and wait for you to enter the number of random numbers it is to average. In this case 10 random numbers are averaged. Strike the return key after the number has been entered, and the program should give you the average and type the first line again.

```
INTEGER DIVIDE CHECK    PC=ØØØ267

INTEGER DIVIDE CHECK    PC=ØØØ267



INTEGER OVERFLOW        PC=ØØØ276

INTEGER OVERFLOW        PC=ØØØ276



IS:          Ø.ØØØØØØØ
COMPUTE THE AVERAGE OF ↑C    .PSEUDO-RANDOM NUMBERS


.DEBUG /CREF/COMPILE TESTIT(,,M),RANDOM(,,M)



FORTRAN:  TESTIT
FORTRAN:  RANDOM
LOADING

LOADER 7K CORE
EXECUTION


XXX     ↑C


.SAVE DSK TEST
JOB SAVED
↑C


.CREF
```

The INTEGER DIVIDE CHECK messages occur when the seconds portion of the time used in the program is an even multiple of 10. This causes IT(2) to be 0 and the divide check results when the program attempts to divide by IT(2). When this happens, the result is as if the divide never occurred, thus the results obtained are still accurate.

The INTEGER OVERFLOW messages occur when RNUMB is multiplied by MAGIC (MAGIC = 5^{15}). The fact that these overflows occur does not affect the results, since the purpose is just to obtain a random number. Both the INTEGER DIVIDE CHECK and INTEGER OVERFLOW messages are suppressed after they are outputted twice.

The average is returned as 0 when it should be close to 0.5 . There must be another bug in the program. When the program requests the number of random numbers to average, type Control-C (↑C) to return to the monitor.

The DEBUG/CREF/COMPILE TESTIT(,,M),RANDOM(,,M) command does several different operations. First it loads DDT along with the two programs to make it possible to debug with DDT. The /CREF causes a cross-referenced listing to be produced to help in finding the bug. /COMPILE must be used for forced compilation of the programs. Ordinarily the source files are compiled only if they have a creation time newer than the binary machine-language files. The /COMPILE (called a switch) overrides this feature and ensures the creation of the listing files. If it was not used the CREF files would not be produced. The (,,M) causes the listings to contain the MACRO expansion code which is sometimes helpful in debugging.

After the monitor types EXECUTION, it transfers control to DDT. This happens because DEBUG was used instead of EXECUTE.

Strike the RUBOUT (or DELETE) key to make certain that DDT was entered. In DDT, the RUBOUT key echoes (types back) as XXX. Now type ↑C to return to the monitor to get the listings.

So that DDT can be reentered with the greatest possible ease, it is necessary to save the core area on the disk. If this is not done, another DEBUG command must be given. The command SAVE DSK TEST writes a copy of the core area on the disk and gives it the filename TEST.SAV. The monitor types JOB SAVED and leaves the Teletype in monitor mode.

Type CREF to print the cross-reference listings on the line printer.

```
.GET DSK TEST
JOB SETUP
↑C

.DDT




MAIN.$: R,,4P$$1B









T,,4P+4$2B
$$F
$G

COMPUTE THE AVERAGE OF ØØØ1Ø.PSEUDO-RANDOM NUMBERS
$1B>>4P R/       .171Ø5693E-33

INTEGER OVERFLOW         PC=ØØ3552
$1B>>4P R/       .265Ø2884E-34

INTEGER OVERFLOW         PC=ØØ3552
$1B>>4P R/       .12697389E-33
$1B>>4P R/       .18615497E-33
$1B>>4P R/       .17992393E-33
$1B>>4P R/       .174Ø8278E-33
$1B>>4P R/       .15958281E-33
$1B>>4P R/       .372Ø1131E-34
$1B>>4P R/       .16486226E-33

INTEGER DIVIDE CHECK     PC=ØØ3543
$1B>>4P R/       .14481467E-33
$2B>>4P+4        T/       .13711563E-32    $P

IS:        Ø.ØØØØØØØ
COMPUTE THE AVERAGE OF ↑C    .PSEUDO-RANDOM NUMBERS

.DDT


RANDOM$:         $B
RNUMB,,2P+4$$1B
$$C
$G
```

GETDSK TEST loads the core image from the disk into core. The monitor types JOB SETUP and returns the Teletype to monitor mode.

The command DDT is typed to reenter the program in DDT. After this command is executed it is as if the monitor had just typed EXECUTION and transferred control to DDT. It is now possible to debug the program with the aid of DDT and the listings from the line printer.

MAIN.$: opens the symbol table for the FORTRAN main program, in this case TESTIT. DDT responds to each command with a TAB. By examining the listings carefully, it is possible to find a place to stop the program to find out what some of the random numbers are it is getting from the subroutine. Location 4P (assigned by the compiler) is the correct point to examine the random numbers. The command R,,4P$$1B tells DDT to type the contents of R (address of the random number) at location 4P, assign this as the first breakpoint, and continue program execution without stopping. If only one ALTMODE ($) were used, the program would stop after typing the contents of R.

At location 4P+4, the total of the random numbers can be examined by typing out the contents of T. The command T,,4P+4$2B instructs DDT to type the contents of T at location 4P+4, assign this as the second breakpoint, and stop the program after the typeout. $$F tells DDT to output the contents of the specified locations as floating point numbers. The $G then starts execution of the program.

The ten random numbers are extremely small and not even close to the range of zero to one. This explains why an average of zero was returned. The calculated average was too small to print using an F format. The bug must be somewhere in the random number generating subroutine. After DDT types the total, type $P to proceed with the program. When it asks for the number of random numbers, type ↑C to return to the monitor.

Reenter DDT by typing the command DDT. Open the symbol table for the random number generating routine by typing RANDOM$: . $B is typed to clear the previous breakpoints. To examine the random numbers produced by RANDOM, type RNUMB,,2P+4$$1B. This tells DDT to type the contents of RNUMB at location 2P+4, assign this as the first breakpoint, and continue the program execution without stopping. This should simply produce random digits in address RNUMB. Type $$C to cause DDT to type the numbers as octal constants. $G starts execution of the program.

```
COMPUTE THE AVERAGE OF 00010.PSEUDO-RANDOM NUMBERS

INTEGER OVERFLOW           PC=003552
$1B>>2P+4          RNUMB/  103231,,730141

INTEGER OVERFLOW           PC=003552
$1B>>2P+4          RNUMB/  353364,,630365
$1B>>2P+4          RNUMB/  2032,,151761
$1B>>2P+4          RNUMB/  273526,,272275
$1B>>2P+4          RNUMB/  105623,,334015
$1B>>2P+4          RNUMB/  52227,,117021
$1B>>2P+4          RNUMB/  136431,,262241
$1B>>2P+4          RNUMB/  317344,,302511
$1B>>2P+4          RNUMB/  366123,,235065
$1B>>2P+4          RNUMB/  254721,,447775

IS:        0.0000000
COMPUTE THE AVERAGE OF ↑C   .PSEUDO-RANDOM NUMBERS

.DDT

T,,2P+10$$1B
$G

COMPUTE THE AVERAGE OF 00010.PSEUDO-RANDOM NUMBERS

INTEGER OVERFLOW           PC=003552

INTEGER OVERFLOW           PC=003552
$1B>>2P+10        T/       20403,,354047
$1B>>2P+10        T/       20402,,262376
$1B>>2P+10        T/       20142,,231053
$1B>>2P+10        T/       20755,,531405
$1B>>2P+10        T/       20300,,643450
$1B>>2P+10        T/       20126,,233254
$1B>>2P+10        T/       20254,,522723
$1B>>2P+10        T/       20315,,115224

INTEGER DIVIDE CHECK       PC=003543

$1B>>2P+10        T/       20316,,622373
$1B>>2P+10        T/       20240,,644566

IS:        0.0000000
COMPUTE THE AVERAGE OF ↑C   .PSEUDO-RANDOM NUMBERS

.DDT


CONST./ 20000,,0           200000,,0
./      200000,,0          $B        $G

COMPUTE THE AVERAGE OF 00010.PSEUDO-RANDOM NUMBERS

INTEGER OVERFLOW           PC=003552

INTEGER OVERFLOW           PC=003552

IS:        0.5106965
COMPUTE THE AVERAGE OF 00100.PSEUDO-RANDOM NUMBERS

IS:        0.5007578
COMPUTE THE AVERAGE OF ↑C   .PSEUDO-RANDOM NUMBERS
```

The numbers appear to be random. The bug must be farther on in the program. Type ↑C to return to the monitor.

Type DDT to return to DDT. It is unnecessary to open the symbol table of RANDOM since it was opened the previous time DDT was used. The command T,,2P+10$$1B causes DDT to type the contents of T at location 2P+10, assign this as breakpoint one, and continue execution without stopping. It is not necessary to clear the breakpoints since breakpoint one has been reassigned. T at location 2P+10 should contain the unnormalized random number. Each number should begin with 200, as that is the exponent used. Type $G to start the program.

The bug is here! Each number begins with 020 instead of 200; therefore, a zero must have been omitted in the exponent in the program. Type ↑C to return to the monitor and reenter DDT by typing DDT.

Type CONST./ to examine the contents of that location. DDT types 20000,,0. There should be another zero in the left half of the word. Type 200000,,0 to correct the error. DDT changes the contents of CONST. to the numbers just typed. Next type ./ to again examine the contents of CONST. DDT responds with the corrected value. $B clears the breakpoints and $G starts the program.

The answer is close to 0.5 for 10 random numbers and closer for 100 random numbers. The program is correct.

```
.TECO RANDOM



*S"2$-T$$
C       THE EXPONENT TO Ø (EXCESS 2ØØ).
        I = ( RNUMB /256 ) .OR. "2*IØ$ØLT$$
        I = ( RNUMB /256 ) .OR. "2ØØØØØØØØØØØ
*EX$$

EXIT
↑C



.DIR/F


DIRECTORY 27,56Ø        1Ø2Ø    25-FEB-7Ø

RANDOM.BAK
TESTIT.BAK
TESTIT.REL
RANDOM.REL
TESTIT
TESTIT.CRF
RANDOM.CRF
TEST  .SAV
RANDOM


.K
CONFIRM: I
RANDOM.BAK    <Ø55>   2. BLKS   :
TESTIT.BAK    <Ø55>   1. BLKS   :
TESTIT.REL    <Ø55>   2. BLKS   :
RANDOM.REL    <Ø55>   2. BLKS   :
TESTIT        <Ø55>   1. BLKS   :S
TESTIT.CRF    <Ø55>   4. BLKS    DELETED
RANDOM.CRF    <Ø55>   6. BLKS    DELETED
TEST  .SAV    <Ø55>   45. BLKS   :
RANDOM        <Ø55>   2. BLKS   :S
JOB 7, USER [27,56Ø]  LOGGED OFF TTY3     1Ø23  25-FEB-7Ø
DELETED 7 FILES (62. DISK BLOCKS)
SAVED 2 FILES (INCLUDING UFD, 5. DISK BLOCKS)
RUNTIME Ø MIN, 26.52 SEC
```

The command TECO RANDOM opens the already existing file RANDOM for editing. The old file is renamed RANDOM.BAK. The command string S"2$-T$$ causes TECO to search for the characters "2 which are unique in the program and are located at the beginning of the octal constant used to insert the exponent. TECO then types the previous line and the current line up to the position following the characters searched for. TECO signals its readiness for another command string by typing an asterisk. The second command string inserts a zero and types the corrected line. The EX$$ causes the file to be closed and read onto your disk area with the corrections included. Control is then returned to the monitor.

Type DIR/F to obtain a directory of your disk area. The /F causes a shortened form of the directory to be printed. The length of each file and the creation date are omitted.

The two files with extensions .BAK are back-up files. They contain the files, specified by file names, as they were before editing was done with TECO. The .REL files are the binary machine-language files created by the FORTRAN compiler. The files with the extension .CRF are the CREF listing files.

Type K (abbreviation for KJOB) to log off the system. When the monitor types CONFIRM :, respond with an I to individually save or delete your files. The two source files are saved by typing an S, the .CRF files are automatically deleted by the system, and the other files are deleted by typing only a carriage return. The system then responds with the rest of the logout information.

Book 7

Advanced Monitor Commands

The system controls many peripheral devices, such as Teletypes, magnetic tape drives, DECtape drives, card readers and punches, line printers, papertape readers and punches, and disks. The monitor is responsible both for allocating these peripheral devices, as well as other system resources (e.g., core memory), and for maintaining a pool of such available resources from which you can draw.

Each device controlled by the system has a physical name associated with it. The physical name is unique. It consists of three letters and zero to three numerals specifying a unit number. The following table lists the physical names associated with various peripheral devices.

Table 7-1
Peripheral Devices

Device	Physical Name
Teletype	TTY0, TTY1, . . . , TTY77
Console Teletype	CTY
Paper Tape Reader	PTR
Paper Tape Punch	PTP
Plotter	PLT
Line Printer	LPT
Card Reader	CDR
Card Punch	CDP
DECtape	DTA0, DTA1, ..., DTA7
Magnetic Tape	MTA0, MTA1, ..., MTA7
Disk	DSK

You may also give each device a logical device name. The logical device name is an alias, and the device can be referred to either by this alias or by the physical name. The logical name consists of one to six alphanumeric characters of your choice. The reason for logical device names is that in writing a program you may use arbitrarily selected device names (logical device names) that can be assigned to the most convenient physical devices at runtime. However, care should be exercised in assigning logical device names because these names have priority over physical device names. For example, if a DECtape is assigned the logical name DSK, then all of your programs attempting to use the disk via the physical name DSK end up using the DECtape instead. It is wise not to give any device the logical name DSK because certain monitor commands (such as the COMPILE commands) make extensive use of special features that the disk has but other devices do not have. The following examples show the use of logical and physical device names.

.ASSIGN DTA,ABC ⤸	Assign a DECtape the logical name ABC.
.ASSIGN MTA1,XYZ ⤸	Assign magnetic tape drive #1 the logical name XYZ.
.ASSIGN PTR,FOO ⤸	Assign the papertape reader the logical name FOO.

In order to use most peripheral devices, you must assign the desired device to your job. You may assign a device either by a program or from the console. The first kind of assignment occurs when you write a program that uses a particular device. When the program begins using the device, it is assigned to you on a temporary basis and released from you when your program has finished with it. The second kind of assignment occurs when you explicitly assign the device by means of the ASSIGN monitor command. This is a permanent assignment that is in effect until the device is released by a DEASSIGN or FINISH monitor command or by your logging off the system.

When you assign a device to your job, the monitor associates your job number with that device. This means that no other user may use the device while you are using it. The only exception is the disk, which is accessible by all users. When you assign the disk, you are allocated a fraction of the disk, not the entire unit. When you deassign a device or kill your job, the device is returned to the monitor's pool of available resources.

7.1 COMMANDS TO ALLOCATE SYSTEM RESOURCES

7.1.1 The ASSIGN Command

The ASSIGN command is used to assign a peripheral device on a permanent basis for the duration of your job or until you explicitly deassign it. This command must have as an argument the legal physical device name (see Table 7-1) of the device you wish to assign. For example, if you want to assign a DECtape drive to your job, type

```
.ASSIGN DTA ↵
```

The monitor responds with the message

```
DTAn ASSIGNED
```

where n is the unit number of the DECtape drive assigned to your job. If all drives are in use, the monitor responds with

```
NO SUCH DEVICE
```

and you must wait until a drive becomes available. You may assign a specific DECtape drive as follows:

```
.ASSIGN DTA3 ↵
```

The monitor responds with

```
DTA3 ASSIGNED
```

if the drive is available, or

```
ALREADY ASSIGNED TO JOB n
```

if job n is using DECtape drive #3.

The ASSIGN command may also have, as an optional argument, a logical device name following the physical device name. The logical device name may be used in place of the physical device name in all references to the device. For example, if you want to use DECtape drive #1 and have it named SAMPLE, type the command

```
.ASSIGN DTA1 SAMPLE ↵
```

If DECtape drive #1 is free, the monitor responds with

```
DTA1 ASSIGNED
```

and stores the logical name you typed. You may then refer to the DECtape by the name SAMPLE until you explicitly release the device or kill your job.

Logical names can be very useful. Suppose you write a program that uses DECtape drive #5 and refers to it by its physical name (DTA5). When you run your program, you find that DECtape drive #3 is the only drive available. Instead of rewriting your program to use DEC-tape drive #3, type

```
.ASSIGN DTA3 DTA5 ↵
```

Thereafter, whenever your program refers to DTA5, it is actually referring to DTA3. Since logical device names are strictly your own, they are different from the logical names of other users. The following is an example using physical and logical device names.

```
. ASSIGN DTA,NAME ↵               Assign a DECtape drive the logical name
                                  NAME.
DEVICE DTA4 ASSIGNED              DECtape drive #4 has been assigned.
.ASSIGN DTA,LINE ↵                Find another DECtape drive;
                                  assign the logical name LINE.
NO SUCH DEVICE                    All DECtape drives are in use.
.ASSIGN PTP,NAME ↵                Reserve paper tape punch.
LOGICAL NAME ALREADY IN USE
                                  Paper tape punch is assigned but NAME
DEVICE PTP ASSIGNED               still refers to DTA4 only.

.ASSIGN DTA3,LINE ↵               Request DECtape drive #3 and give it the
                                  logical name LINE
ALREADY ASSIGNED TO JOB7          Another user (job 7) has DTA3.
```

7.1.2 The DEASSIGN Command

The DEASSIGN command is used to release one or more devices currently associated with your job. This command may have as an argument a physical or logical device name. If an argument is given, the specified devices are released. If an argument is not specified, all devices assigned to your job are released. When devices are released, they are returned to the monitor's pool of available resources for use by other users. The DEASSIGN command does not affect any temporary assignments your job may have for devices.

7.1.3 The REASSIGN Command

The REASSIGN command allows you to give a device assigned to you to another user without having the device returned to the monitor's pool of available resources. Two arguments are required with this command: the name of the device being reassigned and the job number of the user who is receiving the device. For example, suppose you have finished with DECtape drive #6 and the person who is job 10 wants it. Type the command

```
.REASSIGN DTA6 10 )
```

This deassigns DECtape drive #6 from your job and assigns it to job 10, just as if you had typed

```
.DEASSIGN DTA6 )
```

and job 10 had typed

```
.ASSIGN DTA6 )
```

immediately thereafter. All devices except a Teletype console can be reassigned.

7.1.4 The FINISH Command

The FINISH command is used to prematurely terminate a program that is being executed while preserving as much output as possible. If this command is not used, part or all of the output file may be lost. The FINISH command may be followed by a physical or logical device name, in which case any input or output currently in progress in relation to that device is terminated. If no device is specified, input or output is terminated on all devices assigned to your job. The monitor responds to this command by terminating output, closing the file, and releasing the device for use by others.

This command could be used if you were generating an assembly listing of a program on your disk area and decided that you wanted only the first part of the listing, not the entire listing. Type

```
↑C
.FINISH DSK )
```

and the monitor completes the writing of your listing and releases the disk.

7.1.5 The CORE Command

The CORE command allows you to modify the amount of core assigned to your job. The command is followed by a decimal number representing the total number of 1K blocks (1024 word blocks) that you want the program to have from this point on. For example, if you want the program to have 8K blocks of core, type

```
.CORE 8 ↩
```

and the monitor gives the program 8K blocks, if available. If you request additional core and there is none available, the monitor responds with an error message. If the CORE command is followed by the decimal number 0, your program disappears from core because you are requesting 0K blocks of core. If the decimal number following the command is omitted, the monitor types out (1) the total number of 1K blocks you have, (2) the maximum you can request, and (3) the amount of core not assigned to any user.

7.2 COMMANDS TO MANIPULATE TELETYPE ASSIGNMENTS

7.2.1 The TALK Command

The TALK command allows you to type directly to another user's console. This means that everything you type appears on his Teletype and everything he types appears on yours. If his console is not both talking to the monitor and positioned at the left margin, you get a BUSY message. This command is especially useful when you are at a remote Teletype and wish to direct people at the computer site to mount a DECtape for you. When you finish talking to the other console, type control - C to get back to the monitor.

7.2.2 The DETACH Command

The DETACH command causes your Teletype to be disconnected from your program and released to control another job. This means that, while your program is disconnected, you may log in again, receive a new job number, and do something else. The job that was detached from your Teletype is said to be a background job. This means that it is not under control of any user's console. If your background job attempts to type something to the Teletype, it is stopped, for there is no Teletype attached to it.

7.2.3 The ATTACH Command

The ATTACH command allows you to attach a console to a background, or detached job. You must specify the number of the job to which you wish to attach. If you are the owner of the detached job, your console is immediately detached from your current job and attached to your detached job. After this command is executed, the console is in communication with the monitor. If the job you just attached to happens to be running, type CONTINUE without affecting the status of the job.

If you are not the owner of the detached job, you must also specify the project-programmer number of the owner. The project-programmer number must be enclosed in square brackets (e.g. [27,400]) for this command to work. If the job whose job number you typed is already attached to a Teletype, you cannot attach and the monitor responds with

```
TTYn ALREADY ATTACHED
```

where n is the number of the Teletype attached to the job. Observe that only one Teletype can be attached to a job at any time. (For a demonstration on attaching and detaching jobs, see the PDP-10 Reference Handbook.)

7.3 COMMANDS TO REQUEST LINE PRINTER OUTPUT

In Book 2, the TYPE command for listing source files on your Teletype was discussed. In addition, there are three commands that may be used to list files on the line printer: LIST, CREF, and DIRECTORY.

7.3.1 The LIST Command

The LIST command causes the contents of the specified source files to be typed on the line printer. Headings at the top of each page tell the page number, the name of the file, and the date of printing. You may list files from disk or DECtape. For example, the command

```
.LIST TEST.F4 ⤸
```

causes your disk file TEST.F4 to be listed on the line printer. You may list multiple files by separating the filenames by commas. For example, if you want to list three files, SAMPLE.BAS, FILEX.F4,MAIN.F4, from DECtape drive #4, type the command

```
.LIST DTA4: SAMPLE.BAS,FILEX.F4,MAIN.F4 ⤸
```

The asterisk convention discussed in Book 2 may be used with this command. For example,

```
.LIST *.F4 ⤸
```

causes all files with the filename extension .F4 to be listed.

7.3.2 The CREF Command

The CREF command is used to list a certain type of file called a cross-reference file. This command is an invaluable aid in program debugging. If a COMPILE, LOAD, EXECUTE, or DEBUG command string (see Book 2) has a /CREF switch, the command string generates an expanded listing that includes (1) the original code as it appears in the file, (2) the octal values the code represents, (3) the relative locations into which the octal values go, (4) a list of all the symbols your program uses, and (5) the numbers of the lines on which each symbol appears. This is called a cross-reference listing. To print this listing file, you must call in a special cross-reference lister with the CREF command. All the cross-reference listing files you have generated since the last CREF command are printed on the line printer. The file containing the names of the cross-reference listing files is then deleted so that subsequent CREF commands will not list them again.

7.3.3 The DIRECTORY Command

When a DTAn: argument is specified with the DIRECTORY command, the directory of DECtape n is typed on the Teletype. (See Book 2 for a discussion of the DIRECTORY command when no argument is specified.) For example, the command

```
.DIRECTORY DTA2 ⮐
```

types the directory of DECtape drive #2 on the Teletype.

Besides having optional device arguments, this command has two switch options. The first switch option is /F. Including /F in the command string causes the short form of the directory to be listed on the Teletype. The short form of the directory consists of the names of your files and the length of each file in PDP-10 disk blocks. (The long form of the directory also lists the creation dates of each file.) The second switch option is /L. Including /L in the command string causes the output of the directory to go to the line printer rather than to the Teletype. For example, the command

```
.DIRECTORY /L ⮐
```

lists your directory of your disk area on the line printer. The line printer is assigned to you on a temporary basis and is released when the output is finished.

7.4 COMMANDS TO MANIPULATE CORE IMAGES

By using one of the following commands, you can load core image files (see Book 2 for the definition of a core image file) from disk, DECtapes, and magnetic tapes into core and then later save the core images. These files can be retrieved and controlled from the user's console. Files on disk and DECtape are called by filename, and if you have any files on magnetic tape, you must position the tape to the beginning of the file.

7.4.1 The SAVE Command

The SAVE command causes your current core image to be saved on the specified device with the specified filename. This command must be followed by several arguments. First, you must tell the monitor the device on which you want to save the core image. A colon may follow the device name. Second, you must give a name to the core image file. If the filename extension is not specified, the monitor designates one. You may specify the amount of core in which you want your file saved by specifying a decimal number to represent the number of 1K blocks. For example, if you want to save your core image on DECtape drive #2, give it the name SALES, and allow 12K of core for storage, type

```
.SAVE DTA2: SALES 12 ⤸
```

A file called SALES is created and your core image is stored in it. If you list your DECtape directory, the length of the file is slightly over 12,000 words. After you use this command, you cannot continue executing the program. The program can be restarted only from the beginning.

7.4.2 The RUN Command

The RUN command allows you to run programs you previously saved on the disk, DECtape, or magnetic tape. This command reads the core image file from a storage device and starts its execution. You must specify the device containing the core image file and the name of that file. The file must have been saved previously with a SAVE command. If the file is not a saved program, the monitor responds with an error message. If the core image file you want to execute is on another user's disk area, you must specify his project-programmer number, enclosed in square brackets. Again, you may specify the amount of core to be assigned to the program if different from the minimum core needed to load the program or from the core argument of the SAVE command.

7.4.3 The R Command

The R command is a special form of the RUN command. This command runs programs (or CUSPs) that are part of the system, rather than programs that are your own. The R command is the usual way to run a CUSP that does not have a direct monitor command associated with it. For example, the only way to run BASIC and AID is by the commands

```
        .R  BASIC ↵
and     .R  AID ↵
```

A device name or a project programmer number may not be specified for this command.

7.4.4 The GET Command

The GET command is the same as the RUN command except that it does not start the program; it merely generates a core image and exits. The monitor types

```
        JOB SETUP
```

and is ready to accept another command.

7.5 COMMANDS TO START A PROGRAM

7.5.1 The START Command

The START command begins execution of the program at its starting address, the location specified within the file, and is valid only if you have a core image. This command allows you to specify another starting address by typing the octal address after the command. Normally, to start a program, type

```
        .START ↵
```

but to start a program at the specified octal location 347, type

```
        .START 347 ↵
```

A GET command followed by a START command is equivalent to a RUN command.

7.5.2 The HALT (↑C) Command

Typing ↑C stops your program and takes you back to the monitor. The program "remembers" at what point it was interrupted so that it may subsequently be continued. After typing ↑C, you may type any commands that do not affect the status of your program (e.g., PJOB, DAYTIME, RESOURCES) and still be able to continue the execution of the program with a CONTINUE command. However, continuing is impossible if you issue any command that runs a new program, such as a RUN or R command.

7.5.3 The CONTINUE Command

If you stop your program by a HALT (↑C) command, you may resume execution from the point at which it was interrupted by typing the CONTINUE command. You may continue the program only if you exit by typing control - C. If the program exited on an error condition of some sort, the monitor does not let you continue. It types

```
CAN'T CONTINUE
```

if you try. However, you may continue your program if it has halted and given the typeout

```
HALT AT USER n
```

7.6 COMMANDS TO GET INFORMATION FROM THE SYSTEM.

7.6.1 The RESOURCES Command

The RESOURCES command types out a list of all the available devices (except Teletypes) and the number of free blocks on the disk. A disk block is 128 PDP-10 words in length. For example,

```
. RESOURCES ⤸
22100. BLKS,PTY1,CDR,PTR,MTA1,CDP,PLT
```

At the time of this command, there were 22100 free disk blocks available in addition to six devices.

7.6.2 The SYSTAT Command

The SYSTAT command produces a summary of the current status of the system and may be typed without logging in. Included in the summary is a list of the jobs currently logged in, along with their project-programmer numbers, program names being run, and runtime. The following is a partial example of SYSTAT. More information is contained in this program and can be obtained by running SYSTAT.

```
UPTIME  Ø1:26:4Ø, 38% NULL TIME = IDLE+LOST = 26% + 12%

JOB     WHO      WHERE   WHAT    SIZE    STATE   RUNTIME

1       **,**    DET     PIP     1K      ↑C SW   ØØ:12:Ø1
2       2,5      TTY4    MACRO   4K      ↑C SW   ØØ:Ø1:22
3       11,554   TTY3    MACRO   12K     RN      ØØ:Ø5:Ø5
4       4Ø,633   TTY6    PIP     1K      TT      ØØ:ØØ:32
5       13,575   TTY2Ø   COBOL   15K     TT SW   ØØ:ØØ:19
6       3Ø,637   TTY24   DEVCHK  1K      TT SW   ØØ:ØØ:52
7       1,2      TTY1    OPFILE  3K      IO      ØØ:ØØ:2Ø
8       **,**    TTY27   LOGOUT  1K      MQ      ØØ:ØØ:24
9       2Ø,521   TTY13   SYSTAT  1K      ↑C SW   ØØ:ØØ:58
1Ø      2Ø,623   TTY23   PIP     1K      AU      ØØ:11:11
11      1,2      TTY2    PRINTR  2K      IO      ØØ:ØØ:39
12      **,**    TTY33   LOGIN   1K      SL      ØØ:ØØ:ØØ
13      **,**    TTY14   LOGOUT  1K      MQ      ØØ:Ø2:25
14      **,**    DET     UFILE   2K      ↑C SW   ØØ:ØØ:Ø9
15      1Ø4,334  TTY43   TECO    2K      ↑C SW   ØØ:ØØ:ØØ
16      7Ø,54    TTY15   STACK   2K      TT SW   ØØ:Ø1:33
17      4Ø,65    TTY1Ø   TECO    2K      RN      ØØ:Ø2:42
18      4Ø,64    TTY7    PRINT   1K      MQ      ØØ:ØØ:12
19      **,**    DET     SYSTAT  1K      ↑C SW   ØØ:ØØ:ØØ
20      11,7Ø    TTY21   MONGEN  3K      TT      ØØ:ØØ:Ø1
21      **,**    DET     CHKPRT  ØK      ↑C      ØØ:ØØ:Ø1
22      11,131   TTY37   HMAC    6K      TT SW   ØØ:ØØ:42
23      13,12    TTY5    BATCH   4K      SL      ØØ:ØØ:58
24      **,**    DET     PLEASE  1K      ↑C SW   ØØ:ØØ:Ø1
```

7.7 ADDITIONAL MONITOR COMMANDS

In order to present the complete set of monitor commands, the following must be included. These commands are special run control and system administration commands and are discussed in the PDP-10 Reference Handbook, Communicating With the Monitor, Chapter 2.

Table 7-2

Additional Monitor Commands

Command	Explanation
DDT	This command is used in debugging programs and allows DDT to assume control of the execution of your program in various ways. See the PDP-10 Reference Handbook, Communicating with the Monitor, for further information.

Table 7-2 (Cont.)

Command	Explanation
REENTER	Similar to the DDT command. This command causes the program to start at an alternate specified entry point. The alternate entry point must be set by you or your program.
E	This command allows you to examine locations in your core area. The argument adr is required the first time you use this command. By specifying an address, the contents of that location are typed. If an argument is not specified, the contents of the location following the previously specified address are typed out.
D	This command allows you to deposit information in your core area. You must specify octal values to be deposited in the left half and right half of the location. The address of the location in which information is to be deposited may be specified. If the address is not specified, the information is deposited in the last location examined or the last location in which information was deposited.
SSAVE	This command is the same as the SAVE command except that the program will be sharable when it is loaded with the GET command. To indicate sharability, the high segment is written with the extension .SHR. A subsequent GET causes the high segment to be sharable.
CSTART CCONT	These commands are identical to the START and CONT commands except that you are able to continue talking to the monitor. The START and CONT allow you to communicate only with the CUSP that is in execution. Monitor commands may now be executed while the job is running.
SCHEDULE n	This command is legal only from the operator's console. It changes the scheduled use of the system depending on n, where n is: 0 = regular time-sharing 1 = no further logins allowed 2 = no further logins allowed from remote Teletypes
ASSIGN SYS:dev	This command changes the system device to device "dev". The user must be logged in under [1,1] or [1,2] .
DETACH dev	This command assigns the device "dev" to job 0, thus making it unavailable. The user must be logged in under [1,1].

Table 7-2 (Cont.)

Command	Explanation
ATTACH dev	This command returns a detached device to the monitor's pool of available resources. The user must be logged in under [1,1].
CTEST	This command is used to test extensions made to the COMPIL CUSP.

7.8 MONITOR ERROR MESSAGES

The following table contains a summary of the error messages the system can issue. The convention used in the summary is that

dev	represents any legal device name.
file.ext	represents any legal filename and filename extension.
adr	represents a user address.
n	represents a job number or device unit number

Table 7-3

Monitor Diagnostic Messages

Message	Meaning
The typein is typed back followed by ? ↵	The monitor encountered an incorrect character, such as a letter in a numeric argument. The incorrect character appears immediately before the ? For example, .CORE ABC ↵ CORE A? ↵
ADDRESS CHECK FOR DEVICE dev AT USER adr	The monitor checked a user address and found it to be too large or too small; in other words, the address lies outside the bounds of your program.
ALREADY ASSIGNED TO JOB n	The device is already assigned to another user's job (job n).
BAD DIRECTORY FOR DEVICE DTAn	The system cannot read or write the DECtape directory without getting some kind of error. Many times this error occurs when you try to write on a write-locked tape or use a virgin tape.

Table 7-3 (Cont)

Message	Meaning
BUSY	The console addressed is not communicating with the monitor or is not positioned at the left margin. (The operator's console is never busy.)
CAN'T ATTACH TO JOB	The project-programmer number specified is not that of the owner of the desired job.
CAN'T CONTINUE	The job was terminated due to a monitor detected error and cannot be continued.
COMMAND ERROR	General catch-all error response for the COMPILE commands. The syntax of the command is in error and the command cannot be deciphered.
dev: ASSIGNED	The device has been successfully assigned to your job.
DEVICE CAN'T BE REASSIGNED	A user's Teletype cannot be reassigned or an attempt was made to reassign a device that your job is still using.
DEVICE dev OK?	The device is temporarily disabled. The line printer may be turned off or out of paper. For mag tapes, there may be no tape mounted or the switch is in LOCAL. You should correct the situation and then proceed (retry the operation) by typing CONTINUE.
DEVICE dev WASN'T ASSIGNED	The device isn't currently assigned to your job and cannot be deassigned or reassigned by your job.
DEVICE NOT AVAILABLE	Specified device could not be initialized because someone else is using it.
DIRECTORY FULL	The directory of the device is full and no more files can be added.
DISK NOT AVAILABLE	The device DSK: could not be initialized.
ERROR IN JOB n	A fatal error occurred in your job or in the monitor while servicing your job. This typeout usually precedes a 1-line description of the error.
EXECUTION DELETED	A program is prevented from being executed because of errors detected during assembly, compilation, or loading. Loading is performed, but the Loader exits to the monitor without starting execution.

Table 7-3 (Cont.)

Message	Meaning
FILE IN USE OR PROTECTED	A temporary command file could not be entered in your UFD (user's file directory).
HALT AT USER adr	Your program executed a HALT instruction at adr. Typing CONTINUE resumes execution at the effective address of the HALT instruction.
HUNG DEVICE dev	If a device does not respond within a certain period of time when it is referenced, the system decides that the device is not functioning and outputs this message.
ILLEGAL DATA MODE FOR DEVICE dev AT USER adr	The data mode specified for a device in your program is illegal, such as dump mode for Teletype.
ILLEGAL UUO AT USER adr	An illegal UUO was executed at user location adr.
ILL INST. AT USER adr	An illegal operation code was encountered in your program.
ILL MEM REF AT USER adr	An illegal memory reference was made by your program at adr or adr + 1.
INCORRECT RETRIEVAL INFORMATION	The retrieval pointers for a file are not in the correct format; the file is unreadable.
INPUT DEVICE dev CANNOT DO OUTPUT	Output was attempted on a device that can only do input, such as the card reader.
INPUT ERROR	I/O error occurred while reading a temporary command file from the disk. File should be rewritten.
?INVALID ENTRY - TRY AGAIN	An illegal project-programmer number or password was entered and did not match identification in system.
I/O TO UNASSIGNED CHANNEL AT USER adr	An attempt was made to do an OUTPUT, INPUT, OUT, or IN to a device that your program has not initialized.
JOB NEVER WAS INITIATED	An attempt was made to attach to a job that has not been initialized.
JOB SAVED	The output is completed.
LINKAGE ERROR	I/O error occurred while reading a CUSP from device SYS: .

Table 7-3 (Cont.)

Message	Meaning
LOGICAL NAME ALREADY IN USE DEVICE dev ASSIGNED	You previously assigned this logical name to another device. The device is assigned but the logical name is not.
LOGIN PLEASE ?	A command that requires you to be logged in has been typed to the monitor; it cannot be accepted until you log in.
LOOKUP AND ENTER HAVE DIFFERENT NAMES	An attempt was made to read and write a file on the disk. However, the LOOKUP and ENTER UUO's have specified different names on the same user channel. This message does not indicate a DECtape error.
MASS STORAGE DEVICE FULL	The storage disk is full. Users must delete unneeded files before the system can proceed.
NESTING TOO DEEP	The @ construction exceeds a depth of nine and may be due to a loop of @ command files.
nK OF CORE NEEDED	There is insufficient free core to load the file.
n1K BLOCKS OF CORE NEEDED	The user's current core allocation is less than the contents of JOBFF.
NON-EX MEM AT USER adr	Usually due to an error in the monitor.

NON-RECOVERABLE DISC READ ERROR;
 	
NON-RECOVERABLE DISC WRITE ERROR;
	The monitor encountered an error while reading or writing a critical block in the disk file structure (e.g., the MFD or the SAT table). If this condition persists, the disk must be reloaded using FAILSAFE, after the standard location for the MFD and SAT table has been changed using the monitor once-only dialogue.

NO CORE ASSIGNED	No core was allocated when the GET command was given and no core argument was specified in the GET.
NO START ADR	Starting address or reenter address is zero because you failed to specify the address.
NO SUCH DEVICE	The device name does not exist or all devices of this type are in use.

Table 7-3 (Cont.)

Message	Meaning
NO SUCH FILE- file.ext	Specified file could not be found. May be a source file or a file required for operation of the COMPILE commands.
NOT A DUMP FILE	The file is not a core image file.
NOT A JOB	The job number is not assigned to any currently running job.
NOT ENOUGH CORE	System cannot supply enough core to use as buffers or to read in a CUSP.
NOT ENOUGH FREE CORE IN MONITOR	The monitor has run out of free core for assigning disk data blocks and monitor buffers.
NOT FOUND	The program file requested could not be found on the system device or the specified device.
OUT OF BOUNDS	The specified adr is not in your core area, or the high segment is write-protected and you do not have privileges to the file which initialized the high segment.
OUTPUT DEVICE dev CANNOT DO INPUT	An attempt was made to input from an output device, such as input from the line printer.
OUTPUT ERROR	I/O error occurred while writing a temporary command file on disk.
PC EXCEEDS MEMORY BOUND AT USER adr	Your program made an illegal transfer outside its bounds.
PROCESSOR CONFLICT	Use of + construction has resulted in a mixture of source languages.
PLEASE KJOB OR DETACH	Attempt was made to attach a job when you already have a job initialized at that Teletype.
SWAP READ ERROR	A consistent checksum error has been encountered when checksumming locations JOBDAC through JOBDAC+74 of the Job Data Area during swapping.
TOO FEW ARGUMENTS	A command has been typed, but necessary arguments are missing.
TOO MANY NAMES or TOO MANY SWITCHES	Command string complexity exceeds table space in the COMPIL CUSP.

Table 7-3 (Cont)

Message	Meaning
TRANSMISSION ERROR	During a SAVE, GET, or RUN command, the system received parity errors from the device, or was unable to read your file in some other way. This can be as simple as trying to write on a write-locked tape.
TTYn ALREADY ATTACHED	Job number is erroneous and is attached to another console, or another user is attached to the job.
UNRECOGNIZABLE SWITCH	An ambiguous or undefined word followed a slash (/).
UUO AT USER adr	This message accompanies many error messages and indicates the location of the UUO that was the last instruction your program executed before the error occurred.

Book 8

Utility Programs

BATCH

CHAIN

LINED

TECO

BATCH

CONTENTS

Page

CHAPTER 1 INTRODUCTION 8-7

CHAPTER 2 BATCH PROCESSING UNDER THE PDP-10 MONITORS

2.1 Description 8-9

2.2 Batch Equipment Requirements 8-11

2.3 Object Job Status 8-12

2.4 User-Control Levels 8-13

CHAPTER 3 PROGRAMMER CONTROL OF THE PDP-10 BATCH PROCESSOR

3.1 Control Cards Acceptable as Batch Input 8-16

3.1.1 Control Card Notation 8-16

3.1.2 Control Card Descriptions 8-16

3.2 Loading User Programs 8-28

CHAPTER 4 DRIVER (A CORE DUMP PROGRAM)

4.1 General Description 8-29

4.2 Equipment Requirements 8-29

4.3 Implementation of Driver 8-30

4.4 Using Driver 8-30

APPENDIX A BATCH EXAMPLES 8-33

APPENDIX B TABLES 8-41

ILLUSTRATIONS

2-1 Batch Processing within the PDP-10 Time-Sharing System 8-10

2-2 Lines of Control and Communication Between Batch and the Batch-Controlled Object Job 8-11

A-1 Sample Control Cards for Three Jobs Processed by Batch 8-38

TABLES

3-1 Batch User Diagnostic Messages 8-28

B-1 Batch Control Cards 8-41

CHAPTER 1
INTRODUCTION

The PDP-10 Batch Processor (Batch), running under any of the PDP-10 Monitors, supervises the sequential execution of a series of jobs. In a time-sharing environment, Batch performs this function as one of the users of the system, thus allowing normal access by other users. Batch maintains constant communication with the operator and allows him to interrupt, skip, repeat, or prematurely terminate one or more of the jobs in the series at any time. Chapter 2 contains a general description of Batch operation. Chapter 3 describes the Batch control cards that must be present in the input to Batch. The Appendices contain supplementary information on Batch, including examples, and control card and Teletype command summaries.

Two programs are used in conjunction with Batch-Stack and Driver. Stack is used to transfer job files to the Batch input device and stack them there for subsequent input to Batch, transfer output job files created by a Batch run from the Batch output device to some other device (usually the line printer), list job file directories, delete job files, and list directories with selective file deletion or transfer. As described in Chapter 4, the DRIVER provides a core dump of the user's core area.

CHAPTER 2
BATCH PROCESSING UNDER THE PDP-10 MONITORS

2.1 DESCRIPTION

The PDP-10 Batch Processor, Batch, is a subsystem which exists as one of many Commonly Used Systems Programs (cusps) in the PDP-10 System Library. Batch is called in by the operator from his console Teletype by the Monitor command "R BATCH" in the same manner as F40 (the FORTRAN Compiler), Macro-10 (the Macro Assembler), or PIP (the Peripheral Interchange Program), or any of the other cusps. The Batch Processor is not an integral part of the Monitor and therefore occupies no core storage when it is not being run. There is only one version of Batch; this version runs under all Monitors.

Batch is a command interpreter and job supervisor; it does not contain its own versions of assemblers, compilers or loaders. Rather, the PDP-10 Batch Processor calls in and controls the execution of the same standard cusps that an on-line user would utilize in unbatched execution of jobs from a console Teletype. The input/output handler of the Batch Processor uses Programmed Operators (UUOs) which trap to the Monitor; thus, the actual I/O is handled by routines that exist within the Monitor, thereby enabling Batch to be run as a time-shared job in only 3K of core.

When Batch is being run from a user's console Teletype (TTY) as a time-shared job, with a job number n, it sequentially initializes and executes a series of time-shared jobs (with job numbers m_1, m_2, etc.) in parallel with itself. Batch accomplishes this by utilizing a "simulated Teletype" (called a pseudo-Teletype (PTY)). A PTY is a mechanism by which one job (a control job) can control another job (an object job). Batch uses a PTY to type its commands to the object job. Thus, the Batch control job runs concurrently with the time-shared, batch-processed object job. Since the Monitor "sees" the control job (which is running Batch) and the object job (which is being run by Batch) as two separate and distinct time-shared jobs, memory protection and relocation ensures the complete protection of the Batch Processor area of core as well as that of the Time-Sharing Monitor.[1] Consequently, there is no

[1]This holds true only if the hardware configuration includes the memory protection and relocation package, i.e., a time-sharing system only.

way in which an error in a user program being run under Batch can inadvertently interfere with the Batch Processor itself. The fact that the two jobs are being run in parallel by the Time-Sharing Monitor makes it possible for the operator to communicate directly with Batch via his console Teletype without any interruption of the job being run by the Batch Processor; i.e., that portion of Batch which interprets console commands can operate independently from that portion which controls the execution of the object job (unless, of course, the console command specifically requests an interruption or termination of the current user's job being run). See Figure 2-2.

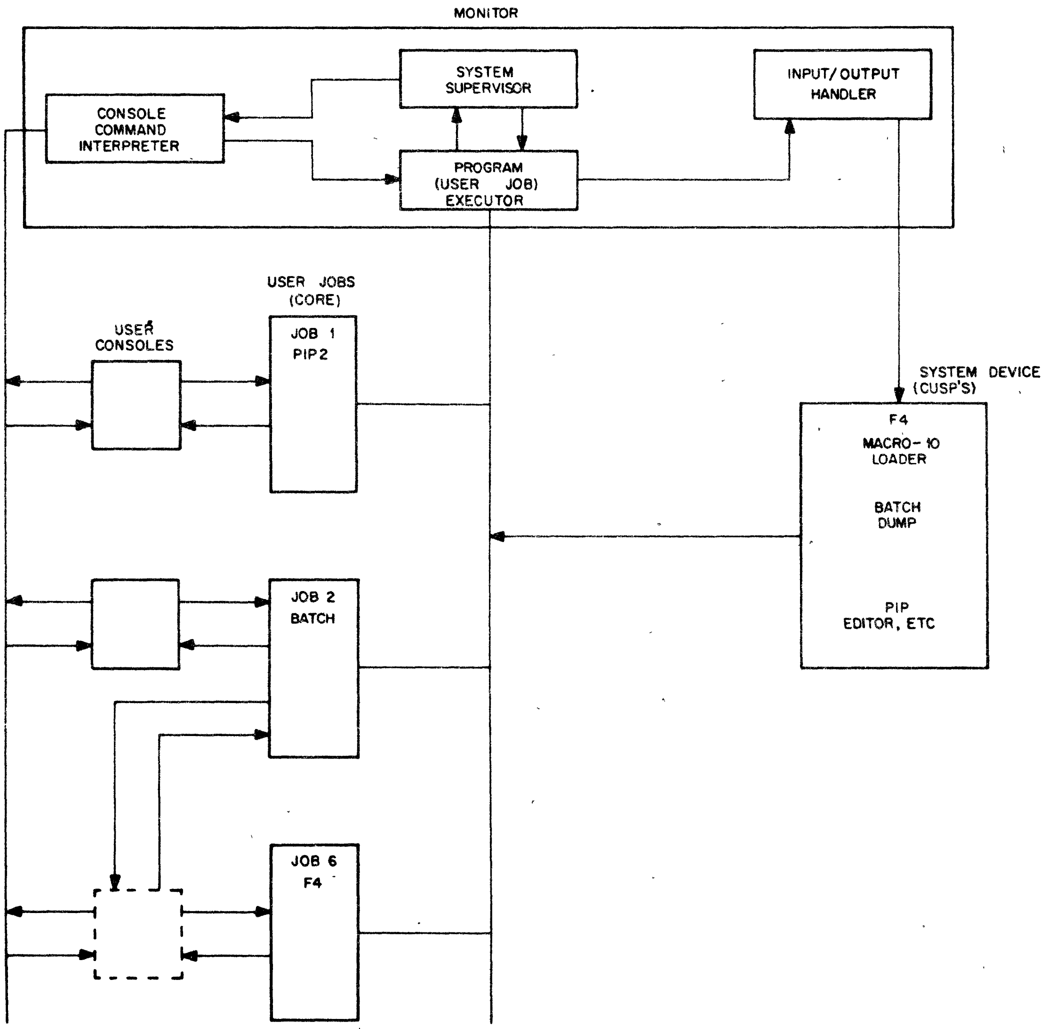


Figure 2-1 Batch Processing within the PDP-10 Time-Sharing System

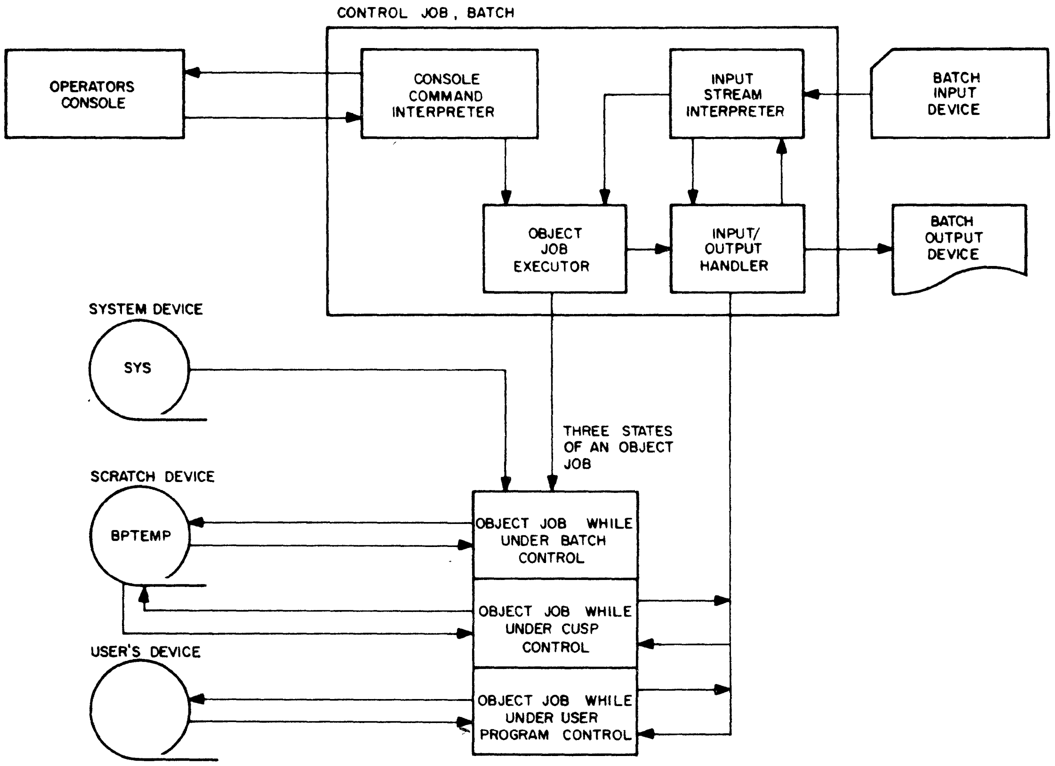


Figure 2-2 Lines of Control and Communication Between Batch and the Batch-Controlled Object Job

2.2 BATCH EQUIPMENT REQUIREMENTS

The minimum utilities required for batch processing within a PDP-10 system are: one Batch input device (contains the input stack of jobs to be run), one Batch output device (for any assembler or compiler listings), one scratch device (BPTEMP)[1], the shared system device (SYS; contains the systems programs), 3K of core for the Batch program, and sufficient core available to Batch to run the required cusps or

[1]This scratch device can be accessed by programs running under Batch by use of the device name BPTEMP:. Assume that it is file-structured.

specified user programs. The input device may be any device capable of input within the installation's system (card reader, magnetic tape, DECtape, paper tape reader, or disk). The Batch output, or listing device, may be any output device within the PDP-10 system configuration (line printer, paper tape punch, magnetic tape, DECtape, or disk); in a time-sharing system, a console Teletype other than the operator's can be used as the listing device. The scratch device can be any retrievable device (magnetic tape, DECtape, or disk) and is assigned to the control job with the logical name BPTEMP before Batch is started. The system device (logical name, SYS) is a directory device specified at system-build time, and shared by all system users; it is kept write protected and cannot be uniquely assigned to any one job, although any job can initialize and input from it. Because of the disk file structure in the PDP-10 systems, a disk may be initialized as "different" devices concurrently; therefore, in a system which includes a disk in its configuration, Batch may be run with that disk as its input, output, scratch and system device, all initialized at the same time. Other devices, including this disk, that a particular job may require must be assigned by control cards in the Batch input stream.

The traditional approach to Batch processing is that all Batch control cards relating to a job to be run appear together with the job on the Batch input device, and the Batch output messages directly concerning the job appear with the job's output on the Batch output device. Other messages may go to the operator's console. In this mode of operation, the user's job should address the Batch input device and Batch output device; these devices may be assigned as any device and the user's job will still run without modification.

However, there is no restriction which states that the user must operate in this way. The Batch control cards and the user's job (e.g., program, data, etc.) may be placed on two or more separate devices.

In most of the discussions in this manual, the first mode of operation is implied, but in actuality only the Batch control cards <u>must</u> appear on the Batch input device, and only the Batch messages concerning the user's job <u>must</u> appear on the Batch output device.

2.3 OBJECT JOB STATUS

Each object job comprises a file in the input stack. Each such job is separately logged in by Batch, and at the end of each job, Batch performs a KJOB. The job number assigned by the system to the object job is typed out following the typeout of that job's $JOB card.

The object job, in which all job execution is done, is always in one of three possible states determined by where control of the object job resides (see Figure 2-2); utilization of the above-mentioned devices varies between each of these states. The object job is under sole Batch control whenever job execution is being controlled; i.e., whenever Batch is bringing a cusp into the object job's core area, whenever compilations, assemblies, or other processes are initialized or terminated, or whenever a postmortem dump is performed. All control-card command execution is done while the object job is under Batch control. Batch can always bring the object job back to its own control, regardless of the object job's state.

When a system program, such as the FORTRAN Compiler, has been started as the time-shared object job by Batch, that program is in control of the job. The compiler may ask for input from the Batch input device, do output to the Batch output device, or perform I/O to devices specified on the control card that initialized its execution. Relocatable binary files produced by the assembler and compiler, or any other intermediate files, are placed on the scratch device, BPTEMP, for subsequent input for another process -- usually a loading process prior to running the now executable user program.

When a user program is executed (execution automatically follows compilation or assembly unless the control card includes a command to suppress execution or there has been an error while compiling or assembling), the object job is running under that program's control. The program can perform input and output on the Batch input and output devices or on any device previously assigned by a control card. When a user program exits normally (with a CALL EXIT), control reverts to Batch which then processes the next control card request; when the exit is due to a monitor-detected error, Batch will execute a postmortem core dump, if requested, by calling most of the dump routine from the system device. If the previously specified maximum running time for the user program is exceeded, control also reverts to Batch.

Ultimate control of the object job always remains with Batch, since it can interrupt the execution of any program which is being run at any time. Batch does the timing of user programs being executed, and interprets commands that the operator may type on the console. These commands could interrupt, terminate, or alter the execution of a series of user jobs, or they could specify new parameters to Batch without interruption of the existing sequence, or even without slowing down object-job execution.

2.4 USER-CONTROL LEVELS

There are two levels of control that a user exercises over Batch; these levels are determined by the form of the commands input to Batch. These commands are either in the form of "control cards" (see Chapter 3) which are prepared and arranged before Batch run time, or they are console commands typed to Batch at run time. The remainder of this section is primarily of interest to the programmer and discusses the lever of control that is exercised over the processor by various types of control-card commands.

CHAPTER 3
PROGRAMMER CONTROL OF THE PDP-10 BATCH PROCESSOR

The programmer controls the execution of his job by Batch through control cards which he prepares prior to run time. A control card is hereby defined as any ASCII string of characters from any legal input device, where that string is terminated by the ASCII characters, RETURN and LINE-FEED, and initialized with an ASCII $ (dollar sign) as its first character. We shall refer to control cards as if they were cards, and refer to positions on these cards as columns. If the input device happens to be a card reader, the end of the card itself determines the termination of the ASCII character string; however, in every case (except the KEY card), column 1 must contain the dollar sign.

The format of control cards requires that the control card type be identified by its first word, which must begin in column 2; this card type is usually an instruction or category of commands which may or may not take a series of arguments. The fields of the various arguments are delimited by spaces (except for the name field on the $JOB card, which must be delimited by a comma, since it may contain spaces) and they need not be delimited by specific columns; leading spaces before arguments are ignored by Bat :h, so absolute positioning is flexible - the relative positioning of arguments, however, must be as specified below.

If a directory device is used as the Batch input device, each user's job is assumed to be a separate file with a name, IJOBxy.abc, where xy represents the ASCII characters 01 through 99 and .abc represents any desired extension (omission of .abc implies a null extension). "xy" must run consecutively, with no missing numbers between the first job (IJOB01) and the last job (IJOBnn). Stack "stacks" files on a directory device, giving them consecutive names as just described (see Chapter 5). Batch runs jobs in the input stack as follows. First, Batch searches for the job IJOB01 and initializes it. At the end of IJOB01, Batch searches for IJOB02, etc. Batch continues operating in this sequential manner until after running IJOBn, a LOOKUP on IJOBn+1 fails; at this point, Batch assumes that it has reached the end of the input stack and types "END OF BATCH."

When Batch is executed with a directory device as its output device, Batch creates filenames in the form OJOBxy.abc, where the xy.abc string in the output files always corresponds to that of the IJOBxy.abc files.

3.1 CONTROL CARDS ACCEPTABLE AS BATCH INPUT

3.1.1 Control Card Notation

In the following control-card command descriptions, these notation conventions are used.

Parameters in uppercase letters indicate the use of that particular ASCII string in the field indicated on the control card; e.g., NOGO suppresses program execution.

Lowercase letters indicate an ASCII string to be substituted; e.g., "name" indicates a variable name.

Lowercase terms with # as last character indicate a quantitative argument; e.g., "time#" indicates maximum execution time.

Parentheses ( ) indicate that the enclosed argument is optional.

Braces { } indicate a choice of formats.

3.1.2 Control Card Descriptions

3.1.2.1 $JOB Card - The $JOB card must be the first card for a given user's application, or user job. The user job may consist of any number of specific tasks, compilations, assemblies or executions which are placed prior to the $EOJ (end of job) card, which terminates the job, causing Batch to look for the next $JOB card in the stack. The format of the $JOB card is as follows.

```
1
$JOB name, core# time# proj# ,prog# (NOGO) (DUMP)
```

name	User's name or other information; must be delimited by a comma and cannot exceed 25 characters, including blanks.

core #	This quantity is the user's estimate of the maximum number of blocks (1024_{10}, or 2000_8 words per block) of core memory required to process his job. This quantity is expressed in decimal. This estimate must include the 1K of locations reserved by the Monitor as the job-data area in every runable user program, and also the 1K locations required by the core dump DRIVER which is loaded by Batch starting at user location 140. All programs and/or cusps within a user job will be run in the amount of core requested on the $JOB card, whether that particular program or cusp requires it or not; this is necessary to prevent another time-shared user from acquiring core needed to complete a specific Batch user job.
time#	This quantity is the user's estimate of the maximum amount of time (in seconds) that his job should require to run to completion. This figure should reflect use of processor time only, since input/output is handled by the Monitor; furthermore, if the environment is time-sharing, only the running time of the user is measured by Batch, not the real-time processor use of any other time-shared jobs. At program run time, if Batch finds that the specified time has been exceeded, a message is typed to the operator on the Batch console teletype, MAX TIME EXCEEDED; however, the user job continues uninterrupted for 8 seconds before skipping to the next user job. During this period, the operator may type TIMEOK to extend the program's execution until its normal exit (or until the operator terminates it with a console command, see Chapter 4).
proj#	The proj#, prog# are applicable only in systems that have a LOGIN feature; this includes all systems with a mass storage device (disk or drum) in their configuration. The project number of a user is the first part of the number that he types when logging in for on-line utilization of the PDP-10 system. The project-programmer number sequence identifies the user file directory (UFD) of that particular user on the disk. Once Batch has logged in the user under his number, all references to the disk with unspecified UFD by any part of the user's program or by $ASSIGN card will be a reference through that user's own file directory. The proj# can be connected to the prog# with a comma (ASCII code =054) or a slash (ASCII code =057).
prog#	This argument is the second part of the project-programmer number; every user of a system should have his own unique programmer number; the project number is usually common to several programmers. The significance of the project-programmer number in a disk system is treated in more detail in the PDP-10 Time-Sharing Monitors manual.

NOGO — If the argument NOGO is included after the variable arguments on the $JOB card, the object program produced by assembly and load or compilation and load will not be executed; this feature might be used in conjunction with a $SAVE or $SSAVE cards in case the user wishes to run the program on-line at some later time.

DUMP — If the character string DUMP is included on the $JOB card, a postmortem dump is performed on the user core image following any object program termination other than a CALL EXIT. If the DUMP argument is not specified, however, the operator may still initialize a core dump by typing the DUMP command on his console Teletype at any time (e.g., upon receiving a MAX TIME EXCEEDED message).

3.1.2.2 <u>KEY Card</u> - The KEY card must be the second card in the user job on systems that have a LOGIN feature. The key is a string of six or less ASCII characters and is associated with the previously mentioned project-programmer number. To be logged in, the key must be correct; otherwise, the user's job will not be executed by BATCH. The format of the KEY card is as follows:

```
1
key
```

NOTE

The ASCII dollar sign ($) is not used as the first character on this card.

Examples of the $JOB card and KEY card combination:

```
1
$JOB EXAMPLE, 20 60 99,99 NOGO
PASWRD

$JOB EXAMPLE2, 21 59 100,100 DUMP
SECRET
```

3.1.2.3 <u>$ASSIGN Card</u> - The $ASSIGN card is required if the user wishes to reference any device other than the Batch input and output device or SYS, the system device. If the user wants to perform I/O on the Batch input/output devices, regardless of what these devices are assigned to by the operator at run time, he does not use the $ASSIGN card; instead he performs I/O with ACCEPT and TYPE statements in FORTRAN, or with "TTY" initialized as his input and output device if his source program is in

Macro. A user program may also use the Batch scratch device, BPTEMP, without an $ASSIGN card; in fact, the symbolic unit (or logical name) specified on an $ASSIGN card can be any logical name except BPTEMP and SYS. With the exception of DSK, the physical device specified cannot define a device already specified as the Batch input or Batch output device (see "NOTES," 4.1.3).

The function of the $ASSIGN card is to set up a correspondence between a symbolic unit used in the user's program and a particular physical device designated at run time; if the device is a tape, it also associates the label appearing on the physical tape with the logical name and the tape drive. The format of the $ASSIGN card is as follows.

```
1
$ASSIGN label symu (PROTECT) (NOREW)
```

label — This is the argument by which the user specifies the physical device he wants to reference in his program by the logical name, symu; it also specifies the physical device that subsequent control cards will reference as symu.

If label is a nonsharable physical device (e.g., CDR, LPT) and is not available at user run time, Batch will type a message, "USER NEEDS DEVICE label." to the operator and then wait for a response before proceeding. The operator should attempt to assign the required device to Batch, perhaps after waiting for the current user of the device to finish with it and release it. When the operator types "CONT", Batch will attempt to INIT the device. If this attempt succeeds, Batch will reassign the device to the object job and continue execution. If Batch cannot INIT the device, it will print an error message on the output device and skip to the next user's job.

If the user assigns a sharable physical device (e.g., DTA, MTA), the label must begin with either D (for DECtape)[1] or M (for magtape) followed by some word or number that identifies the reel to be mounted. For example, the serial number might be used to indicate to the operator which of the tapes given him by the user should be mounted at this time. A label beginning with a D or M will cause Batch to type "MOUNT TAPE label WRITE PROTECTED" or "MOUNT TAPE label WRITE ENABLED" (the presence or absence of the PROTECT argument determines which of these two messages is typed). The operator then mounts the specified tape on an available drive and types the device name and number on the console. (NOTE: The drive selected must not be that of either the Batch input or Batch output device.)

[1]However, note that the label "DSK" assigns disk to the symu.

```
$ASSIGN TAPEIN D1984 PROTECT
```

This $ASSIGN card causes the following typeouts at run time.

```
MOUNT TAPE D1984 WRITE PROTECTED
```

(Operator mounts DECtape with serial number of 1984 on an available DECtape drive, for example, drive 5)

```
DTA5: ↵
OK
```

Operator types in device name and number, followed by a colon.

Batch responds with message indicating that device was successfully initialized.

If, for some reason, the drive on which the operator has just mounted the tape is not available (i.e., it is assigned to another time-shared job), the message "physdev NOT AVAILABLE" (where physdev is the physical device typed in) is typed to the operator followed immediately by the "MOUNT TAPE" message previously described. As soon as successful assignment is made, Batch types "OK" to the operator and waits for his "CONTINUE" command before resuming processing. If no drives are available or if the operator wishes to cease processing of the job, "END" can be typed to abort the job.

symu	This argument can consist of six or fewer alphanumeric characters (no spaces or punctuation can appear in this string). This is the symbolic unit which must be the name that the user's program employs to reference the device.
PROTECT	This argument is described under "label" above.
NOREW	This argument causes magtape to bypass rewind.

The $ASSIGN control card

$ASSIGN label symu

is equivalent to the Monitor command

.ASSIGN label symu

if symu is equal to DSK or does not begin with an ASCII D or M. In the latter case (D or M), the operator must provide Batch with an actual DECtape drive or magtape drive.

3.1.2.4 $TAPE Card - The $TAPE card causes Batch to dynamically perform certain device-dependent operations on DECtapes or magnetic tapes that have previously been assigned a symbolic-unit name with an $ASSIGN card. The control card format is as follows.

```
1
$TAPE symu op1 op2 op3 ...
```

symu	This logical symbolic unit must have been specified with an $ASSIGN card; if it was not, the message "UNIT symu NOT ASSIGNED, CARD IGNORED" is output on the Batch listing device, where symu is the unassigned logical name which was specified on the $TAPE card. If symu was assigned to a nontape device, the operations specified will be no-ops, but no message will be printed.
opn	This argument may be any of the following 2-character tape operation codes; the operations are executed in the order in which they appear on the card and their number may be variable. Operations that do not pertain to the device specified (e.g., zero directory command for magnetic tape) are considered to be no-ops.

DZ	Zero directory of DECtape symu.
MA	Advance one file on magnetic tape symu.
MW	Rewind magnetic tape symu to load point, or rewind DECtape to end zone at front of tape.
MB	Backspace magnetic tape symu one file.
MT	Advance magnetic tape symu to logical end of tape (double end of file).

3.1.2.5 $MAC, $F4, and $CBL Cards - The $MAC card executes a Macro-10 assembly; the $F4 card executes a FORTRAN compilation at the user's source program; similarly, the $CBL card executes a COBOL compilation of the user's source program. Then (if execution is not suspended), the users program is loaded and executed with the timing control specified on the $JOB card. The action that Batch takes with the relocatable binary program produced by either assembly or compilation is independent of whether the compiler or assembler produced it. The format of the cards is identical; and the conditions under which the relocatable binary program is to be handled by Batch may be specified on other control cards regardless of whether the source program was FORTRAN, Macro-10, or COBOL. The format of the cards is as follows.

$$\left\{\begin{array}{l}1\\ \$MAC\\ \$F4\\ \$CBL\end{array}\right\} \text{prgname (B: symu2) (L: symu3)} \left\{\begin{array}{l}\text{(S: symu1)}\\ \text{(S: symu1: prgname1,prgname2, ...}\\ \quad \text{....symux:prgnamey)}\end{array}\right\}$$

prgname	This argument is a program name of six or less characters. If the control card has specified any directory output device, this will be the name of the file which will be produced by the compilation or assembly. If a directory output device has been specified and no name is given here, the name TEMP will be assigned. If NOGO was specified on the $JOB card, the file produced by assembly or compilation will not be passed to the Loader.
B: L: S:	These optional arguments specify that the user wishes to specify his own source, binary or listing device, respectively. The symu following the B, L, or S must have been assigned previously with a $ASSIGN card. The first of the "S" format above assumes the file prgname if symu1 is a directory device. The second of the two "S" formats above describes the case in which a file other than prgname or several source files are to be compiled or assembled into one binary program. B:* or L:* indicates that binary output or listing is to be suppressed.

If no source device is specified, the Batch input device is assumed: if no listing device is specified, the Batch output device is assumed; and if no binary device has been specified, the intermediate relocatable binary file is output on the Batch scratch tape, BTEMP, from which it is loaded by the Loader prior to execution (note that BPTEMP is zeroed following each job). Since the Macro-10 Assembler is two-pass, whenever the source is unspecified, the following file is copied from the Batch input device onto BPTEMP prior to assembly; in this case, if some binary device has not been specified[1], Batch will request that the operator mount additional scratch device(s).

Note that "TTY" specified as source or listing device results in the source or listing being input or output on the Batch input or output device.

3.1.2.6 <u>$CREF Card</u> - The $CREF card performs all the functions of the $MAC card and, in addition, produces a cross-reference assembly listing. It is provided for the user who wishes to obtain a special cross-reference listing of his Batch assembly. In the Macro listing produced, there is a decimal sequence number printed leftmost on each line of the output, regardless of whether that line is an instruction, comment, or even blank. At the end of the listing, after the symbol table, every symbol referenced in the program is listed alphabetically; each symbol heads a horizontal list of sequence numbers which identify every line in which the given symbol is referenced.

[1]Specification could be by $BIN card prior to this card.

Note that, according to the following format description of the $CREF card, the listing produced is printed on the Batch output device.

```
1
$CREF    prgname   (B:symu2)   { (S:symu1)                                      }
                               { (S:symu1:prgnam1, prgnam2,...,symux:prgnam3)  }
```

B:
S:	Since the cross-reference procedure requires an intermediate "scratch" listing device, Batch utilizes BPTEMP for this purpose. Since the $MAC card already requires a binary device (in addition to BPTEMP) if no source device is specified, the $CREF card requires that if neither a binary nor a source device is specified, the operator will be requested to mount two additional scratch devices at assembly time.

3.1.2.7 <u>$LDR Card</u> - The $LDR card will cause the Loader to load relocatable binary files from a device previously assigned with an $ASSIGN card or from the system device SYS or from BPTEMP. The format for this card is as follows. Note that every program which has been assembled or complied (via $MAC,$F4, $CBL and $CREF cards) is automatically loaded. Thus, the $LDR card should be used only for programs which already exist in binary format and have not been assembled or compiled during this Batch run.

```
1
$LDR  symu: prgnam1, prgname2,...   (LIB)
```

symu	Argument is described above.
prgnam	These must be the file names of relocatable binary files; they are specified only if symu is a directory device.
LIB	If this string appears, every program on the file is loaded in library search mode (see description of Loader, <u>PDP-10 System User's Guide</u>).

3.1.2.8 <u>$EXLDR Card</u>

Card format:

```
1
$EXLDR (MAP)
```

This card is optional, and is to be used only when no more source programs follow. When this card is read by Batch, the Loader is brought into core and is used to load all programs previously compiled or assembled, or previously specified by $LDR cards.

MAP	If this appears on the card, a storage map is generated for this loading operation.

Any $LDR cards that follow the $EXLDR card do not bring the Loader in again, but continue to load with the Loader that is already in the user's area.

If the $EXLDR card is missing, the Loader is called and loading is triggered by the $EOJ or $EOF card.

3.1.2.9 $BIN Card

Card format:

```
1
$BIN symu
```

symu	The logical name, symu (previously specified with an $ASSIGN card), specifies an assigned device which becomes the permanent output device for all relocatable binary files. Once this card has been read, all files normally written by Batch on BPTEMP are written on the specified device.

NOTE

Make certain that the device is not write protected when the $ASSIGN card specifying it is read.

3.1.2.10 $SAVE Card and $SSAVE Card

Card format:

```
1
$SAVE symu:prgname (core#)
$SSAVE symu:prgname (core#)
```

These cards cause Batch to save the core image of the user's area on the symu device specified in the argument field; if the device has a directory, the saved file is assigned the name "prgname." The constraints on the symu device are the same as for the $BIN card above; i.e., it must not be write protected. The $SAVE card and the $SSAVE card serve the same function except that the $SSAVE cause the stored program to be sharable.

core#   This argument specifies the number of 1K core-image locations which Batch is to save on the specified device; however, if the program in core requires more space than specified in the argument, it will be saved in the amount of space required. This argument takes precedence over the core argument specified on the $JOB card for the specific execution of the $SAVE (or $SSAVE card) card only, with all following operations being executed in the amount of core specified on the $JOB card. If no core# argument is specified on the $SAVE (or $SSAVE card) card, the program is saved according to Monitor conventions.

3.1.2.11 $RUN Card

Card format:

```
1
$RUN symu:prgname
```

This card causes Batch to read in and execute the program "prgname" from the previously assigned symu; this device may be write protected. The device can also be SYS or BPTEMP. Note that BTEMP is zeroed at the end of each job.

prgname   This must be a previously saved .SAV file or a system cusp. If the program "prgname" expects input from TTY, these desired prgname inputs must be on the Batch input device, following the $RUN card; they should not have dollar signs in column 1. In this case, an $EOF card should be used to terminate the prgname input stream.

NOTE

Programs which run in DDT submode where communication between Batch and the program run by Batch is via Teletype (in DDT submode) cannot be specified (e.g., TECO, DDT).

3.1.2.12 $GET Card

Card format:

```
1
$GET symu:prgname
```

This card causes Batch to read in the program "prgname" from the previously assigned symu. The arguments and functions of this card are similar to those of the $RUN card in all respects except that the program is not executed.

3.1.2.13 $START Card

Card format:

```
1
$START
```

The $START card causes Batch to initiate execution of a program that has been called in with the previous $GET card, or that has been linked by the loader.

3.1.2.14 $EOF Card - This card has no argument field.

The $EOF card creates an internal end-of-file to a preceding input stream on the Batch input device. If NOGO was not specified on the $JOB card, the $EOF card triggers the execution of programs that have been loaded in core, or before execution triggers the loading of programs that have been assembled or compiled previously under control of $MAC, $F4, $CBL or $CREF cards. If the previously compiled or loaded programs require data from the Batch input device (ACCEPT statements or INPUT UUOs in Macro from TTY), the data must appear on the cards immediately following the $EOF card on the Batch input device.

3.1.2.15 $EOJ Card - This card has no argument field.

The $EOJ card specifies the end of a user's job. If no $EOF card has been issued previously, the $EOJ card triggers the same actions as far as loading and execution of programs by Batch as the $EOF card. The $EOJ card, however, specifies the end of the input stream for the current job.

NOTE

> This card is mandatory for the end of the job; however, if an end of file (EOF) on the input device occurs (an EOF appears after each input job, except when the Batch input device is card or paper tape), Batch simulates the $EOJ card.

3.1.2.16 $DUMP Card

Card format:

```
1
$DUMP
```

The $DUMP card inserted at any point in the user's data specifies that a dump is to be taken when the job reaches this point.

3.1.2.17 $* Card - Write to operator and wait for reply.

Card format:

```
1
$*  comment
```

Columns 4 through 80 of this card are typed on the operator's console, and Batch waits for operator action, e.g., CONT or END.

3.1.2.18 $** Card - Write to operator without reply.

Card format:

```
1
$**  comment
```

Columns 4 through 80 of this card are typed on the operator's console, and Batch continues without operator intervention.

3.1.2.19 $PAUSE Card

Card format:

```
1
$PAUSE
```

The $PAUSE card suspends Batch operation until operator intervention.

3.2 LOADING USER PROGRAMS

Batch will run Loader whenever it encounters a $EXLDR, $EOF, $EOJ, $SSAVE or $SAVE control card. Batch loads programs in the order they were encountered on the $LDR, $F4, $CBL and $MAC cards, so that programs which previously existed in binary form may be loaded prior to (that is, into numerically lower addresses than) programs which have been translated during this Batch run.

If loading was initiated by a $EXLDR card, Batch will then load any programs specified by $LDR cards and finish loading when it encounters a $EOF, $EOJ, $SAVE or $SSAVE card.

Table 3-1
Batch User Diagnostic Messages

Message	Meaning
BEGIN EXECUTION	$EOJ card has been processed.
CAN ONLY COPY 1 FILE ONTO DISK	Cannot copy onto BPTEMP for MACRO assembly.
CAN'T COMPILE OR ASSEMBLE AFTER A $EXLDR CARD	The input deck is out of order.
DRIVER NOT LOADED - CAN'T DUMP	
******JOB ABORTED	The input job was terminated during runtime by operation intervention.
JOB CARD BAD	A $JOB card field has been omitted or is in improper format.
MAX TIME EXCEEDED - JOB KILLED	Job has exceeded time on job cards.
NO EXECUTION	An error or a NOGO has resulted in no execution of a previously loaded program.
NO LOAD - PROGRAM NOT SAVED	Trying to load a non-saved program.
******NOT AVAILABLE	Batch has received a "?" from the monitor on a $ASSIGN card indicating the device is not available.
SCRATCH DEVICE DIRECTORY FULL	Could not enter a file name into the directory.
UNIT UUU NOT ASSIGNED, CARD IGNORED	The unit named on the card has not been specified on a $ASSIGN card.
UNRECOGNIZABLE CARD, IGNORED	An illegal control card has been introduced into the Batch stream.
UNRECOGNIZABLE FIELD ON CARD, CARD IGNORED	A field on the card is in improper format.

CHAPTER 4

DRIVER (A CORE DUMP PROGRAM)

4.1 GENERAL DESCRIPTION

Driver is a PDP-10 system program used in conjunction with the Batch Processor (BATCH).

The major function of Driver is to provide a dump of the user's core area, when the Batch Processor encounters a $DUMP card in the Batch input deck, a DUMP option on the job card, or a DUMP command typed in by the Batch operator.

Driver performs all of the following functions when it is invoked. These functions are performed sequentially as listed.

(1) A dump of the ACs in both octal and decimal is performed.

(2) A dump of selected monitor status information, obtained by utilization of the GETTAB UUO is performed. The standard Driver outpus the job status word (JBTSTS), job relocation and protection (JBTADR), high segment table (JBTSGN), and the user program name (JBTPRG).

(3) A dump of specific job data locations is performed. The standard Driver outputs the following job information in this format.

JOBUUO	JOB41	JOBERR	JOBENB	JOBREL
JOBBLT	JOBHCU	JOBDDT	JOBHRL	JOBSYM
JOBUSY	JOBSA	JOBFF	JOBREN	JOBAPR
JOBCNI	JOBTPC	JOBOPC	JOBCHN	JOBCOR
JOBVER				

(4) A dump from the end of Driver up to JOBFF in octal, SIXBIT, and ASCII is performed.

NOTE

Because Driver is written as a series of MACRO calls, it is relatively simple for each installation to modify or extend the information output on the dump. For further information, consult the Driver source file.

4.2 EQUIPMENT REQUIREMENTS

The Batch Processor operator's console is required for control of Driver. Driver requires one output device (usually the line printer) although intermediary output devices, such as MTA, can be used.

4.3 IMPLEMENTATION OF DRIVER

The $EXLDR card automatically loads Driver into core before any user programs.[1] When a dump is requested (either by the $DUMP card or by operator intervention) the Batch Processor uses the E (EXAMINE) command to look at location 140. If location 140 contains "DRIVER" in SIXBIT, an ST (START) 141 is used to invoke the dump. If location 140 contains a value other than SIXBIT "DRIVER", the Batch Processor outputs a diagnostic to the user's output, and continues with the Batch note job stream.

4.4 USING DRIVER

Once invoked by either the $DUMP card or by operator intervention, Driver performs a core dump of the Batch job area without any operator intervention; no further operator actions are required for this Batch job.

[1] If the $EXLDR card is omitted, Driver can be loaded with a $LDR card or a $RUN SYS LOADER card.

```
                                                  *********** CORE DUMP ***********
 ACS                                                      OCTAL
00-07 000000000004 777700004371 000011002166 000000006457 000000007247 000000000000 777250007247 000600000012
10-17 000000000012 000000000042 047000000012 040332471016 000000000006 004555002367 777754004235 777750000632
                                                          DECMAL
00-07 +4           -16774919    +2360438     +3375        +3751        +0           -90173785    +100663306
10-17 +10          +34          +5234491402  +4352274958  +6           +632554743   -5240675     -6291046

 MONITOR STATUS INFO

JBTSTS 440004000006 JBTADR 007777104002 JBTSGN 000000000000 JBTPRG LOADER

JOB DATA INFO

JOBUUO 000404007255 JOB41  264000002107 JOBERR 000000000000 JOBENB 000000000000 JOBREL 000000007777
JOBBLT 310000000301 JOBHCU 000000000000 JOBDDT 000000000000 JOBHRL 000000000000 JOBSYM 777250007250
JOBUSY 000000007250 JOBSA  004271001054 JOBFF  000000004371 JOBREN 000000000000 JOBAPR 000000000000
JOBCNI 000000000000 JOBTPC 000000000000 JOBOPC 310000001052 JOBCHN 001056000000 JOBCOR 004271000000
JOBVER 000000000017

 ADDR                     OCTAL                                  SIXBIT          ASCII

001050 440600000001 440700000000 254000001052 266700004033 *D8   ID'     5@  (J6W  @|@ *H      H       *      -\    ↑
001054 015000000000 254000001052 321440001067 200000003260 *!H     5@  (JI,@ (W0   IP* *       *      42          X↑
001060 202000003262 607440000010 254000001067 230140003270 *00   |RP\@  (5@   (W0!@ |X* * @  YAR     *            \↑
001064 202147000001 200140003272 202147000002 135140003340 *01G   |0|@ |Z01G  "*|@ |@* * F8          ] F8    &    P↑
001070 271140003040 205000350700 202000003034 550543000000 *7)@ 8@0H =' 00    8<MXC    * *,&     !   Q@ @     Z     ↑
001074 505540000700 554013000000 306000254000 254000001110 *HM@  ' M@*    8P 5@ 5@   )(* *Q6    @[ X    1@ X *     S↑
001100 370000000013 554003000000 336700000000 201000000373 *?     *M@#    |P      0(   #[* *>      [        7@          ]↑
001104 270000000013 202540003033 202540003334 354000000000 *7     *05@ 8|05@ |<=@      * *,        V         V   NI     ↑
001110 550020000013 542000003034 202540003035 630440003342 *M 0   *L0    8<05@ 8=S5@ |B* *Z       X@        V    F     Q↑
001114 476000003036 541240000000 403600003017 621440100000 *GP   8>L*@     @>    8/R,@(   * *0@     Y*      @X      D2     ↑
001120 541440000000 403040000006 505240000000 200100000013 *L,@    @8@   &HJ@     0!      ** *X2      @B      Q@           ↑
001124 202100003332 134100000013 311540003034 254000001177 *01    ||*A    *9-@ 8<5@    )** * D   M          26      *    ?↑
001130 620100000100 640100000040 200140000002 231140000007 *R!    ! T!    @0!@   "3)@    !* *D      H                &8    ↑
001134 135144001470 254023001511 663440000001 254000001142 **ID  ,X5@3  -)V<@    |5@   )B* * &      *      $LR      *    1↑
001140 200200003332 202200003333 275100000020 221040000012 *0"    ||02    ||7I     02(@    ** *       M H   M/$      $"     ↑
001144 270040000002 254000001123 671240000001 621440000001 *7 @    "5@    )3W*@   |R,@    |* *,        *      )N*      D2     ↑
001150 201040000000 254000001123 261740000001 261740000013 *0(@     5@    )36/@   |6/@    ** * "      '*     ),>      ,>     ↑
001154 607440000002 303600000001 254000001166 627440000001 *P\@    "8>    |5@   )VR\@    |* *AR      @X      *     |ER     ↑
001160 254000001164 200040003333 202040003334 304000000000 *5@    )T0 @ ||00@  |<8@         * **        |     M B    N1     ↑
001164 202540003334 202540003033 201040000000 344600001123 *05@  |<05@ 8|0(@   ' <F    )3* * V     N V      "       9    )↑
001170 260740001346 541440000000 367600001214 667440000002 *6'@ *FL,@     >*    *,V\@   "* *,      SX2     =X    FMR     ↑
001174 321600001177 333017777777 254000001214 260740001346 *!,    )-|8/---5@   *,6'@ *F* *48    ?6@*       F,       S↑
001200 260740001721 550000003010 301002000002 274740003343 *6'@  /1M     8(8(     "7G@ |C* *,      HZ       0       /     G↑
001204 661440100000 200540003334 621440000001 603440010000 *V,@(   0X@  |<R,@    |P<@|    * *L2             ND2     @R     ↑
001210 254000001463 201000003060 251000003077 254000002103 *5@   ,S0(    8P5(    8-5@  1#* **              *        *     |↑
001214 373017777777 254000001220 274740003343 254000001117 *?8/---5@    *07G@ |C5@    )/* *>@*      H/      0*      !*
001220 200557000000 344600001117 661440200000 621440000001 *0X0     <F    )/V,@0   R,@    |* *   X   9     'L2      D2     ↑
001224 254000001123 260740001346 260740001721 254000001120 *5@    )36'@  *F6'@  /15@    )0* **       ),      S,      H*    (↑
001230 201040000204 621440000001 661440001000 621440000001 *0(@  "$R,@    |V,@  ( R,@    |* * "     BD2      L2      D2     ↑
001234 322040001120 550100003034 275113000000 221100000005 *10@   )0M|    8<7|*    2)     X* *4R     (Z      /$X     $$    ↑
001240 325440001252 134000000013 362100001177 607440001000 *:L@  *J*@     *>1    )-P\@ ( * *52     U       <D     ?AR    ↑
```

NOTE

SIXBIT output is enclosed between asterisks (*)
ASCII output is enclosed between up-arrows (↑).

APPENDIX A
BATCH EXAMPLES

This appendix contains a collection of examples showing the use of Batch control cards and operational procedures.

Example 1

```
$JOB     BATCH1,     10   20   100,101
DEMO1
$F4      TEMP
(FORTRAN PROGRAM STATEMENTS)
$EOF
(INPUT DATA)
$EOJ
```

On running the above sequence of cards under control of Batch, the following actions occur.

a. The contents of the $JOB card, followed by the job number assigned by the system to the object job, is printed on the operator's Teletype.

b. The FORTRAN compiler produces a file, TEMP.REL, on BPTEMP.

c. The $EOF card triggers the loading of TEMP.REL and starts the job.

d. The $EOJ card marks the end of the job and the end of the input data.

Example 2

```
$JOB     BATCH2,     10   10   100,101
DEMO1
$ASSIGN   DTA1        DOUT
$F4      PROGRM
         DIMENSION    II(1000)
C        DTA 1   = LOGICAL DEVICE NO.9
         INTEGER G
         DATA G/9/
         DO   1  J=1,  1000
1        II (J) = J
         FILE = 4HFILE
         CALL   OFILE  (G,FILE)
```

```
                    WRITE  (G)    II
                    END FILE G
                    STOP
                    END
        $EOJ
```

On running the above job, the following actions occur.

a. The contents of the $JOB card, followed by the assigned job number, is printed on the operator's Teletype.

b. The message

MOUNT TAPE DOUT WRITE ENABLED

is printed on the operator's Teletype. The operator can specify the actual DECtape (DOUT) drive on which he has mounted a DECtape reel by typing

DTAn:⤸

Batch replies with

```
OK
*
```

and, when user types CONTINUE, proceeds with the running of the job.

c. The FORTRAN compiler produces a file, PROGRM.REL, on BPTEMP, and the $EOJ card performs the functions of the $EOF card (triggers the loading of PROGRM.REL and starts the job).

d. The execution of the program will create a file, FILE.DAT, on DECtape DOUT.

Example 3

In the preceding examples, the storage map produced by the Loader was written on the output device. This map can be suppressed by using the following sequence of control cards.

```
$JOB        BATCH3,    10     1     100,101
DEMO1
$ASSIGN     DREL       DTA
$F4         MAIN

(FORTRAN PROGRAM STATEMENTS)

$LDR DTA:SUBR1,SUBR2
$EXLDR   NOMAP
$EOF

(INPUT DATA)

$EOJ
```

The DECtape DREL containing the .REL files SUBR1 and SUBR2 will be requested by Batch.

```
MOUNT TAPE DREL WRITE ENABLED
```

After the compilation, the Loader is brought in and executed by the $EXLDR card. Note that the file MAIN.REL is not specified, as doing so would result in multiply defined symbols. The option NOMAP on the $EXLDR card suppresses the writing out of the storage map. The $EOF card begins the execution of the loaded program. The $EOJ card terminates the input data and the job.

Example 4

To save a program on a DECtape, the following sequence of control cards can be used.

```
$JOB        BATCH,   10   1   100,101   NOGO
DEMO1
$ASSIGN     DSAV     DTA
$F4.        PROG
(FORTRAN PROGRAM STATEMENTS)

$SAVE   DTA:FORSAV
$EOJ
```

Changing DSAV on the $ASSIGN card to DSK would result in saving the job on the disk in the user's (100,101) disk area.

Example 5

To suppress the printing of the Macro code generated by the FORTRAN compiler, the following sequence of control cards can be used.

```
$JOB      BATCH    12    100,101
DEMO1
$RUN    SYS:F40
BPTEMP:COMP,TTY:/M←TTY:
(FORTRAN PROGRAM STATEMENTS)

$LDR.   BPTEMP:COMP
$EXLDR   NOMAP
$EOF

(INPUT DATA)

$EOJ
```

The following facts should be noted.

a. Switch /M suppresses the printing of the Macro code.

b. BPTEMP may be used without assigning it.

c. COMP must be specified on the $LDR card, because the compilation is not done under control of a $F4 card.

d. The $EOF card begins execution of the loaded program.

Example 6

The use of the Macro-10 assembler by means of the $MAC or $CREF card is completely analogous to the use of the $F4 card.

If one wants to run the assembler by means of a $RUN card, it should be noted that the assembler makes two passes and, consequently, the source statements must be entered twice; this introduces a problem where card input is involved. However, by means of PIP (Peripheral Interchange Program), this problem can be alleviated. The control cards used are given below.

```
$JOB    BATCH6,         10    40    100,101
DEMO1
$RUN    SYS:PIP
BPTEMP:SOURCE←TTY:

(MACRO PROGRAM STATEMENTS)

$EOF
$TAPE BPTEMP MW
$RUN    SYS:MACRO
BPTEMP:MACBIN,TTY:←BPTEMP:SOURCE
$EOJ
```

Example 7

This example shows the combination of a FORTRAN program and a Macro-10 program in one job.

```
$JOB        BATCH7,    10    1    100,101
DEMO1
            $F4
            DIMENSION KK(40)
            ACCEPT  1,KK
            DO  2 I=1,40
            K=KK(I)
            CALL MACRO(K,L)
2           TYPE 3,K,L
            STOP
1           FORMAT(40I2)
3           FORMAT(1X,2I7)
            END
```

```
$EOF
$MAC      SUB
          ENTRY MACRO
MACRO:    Z
          MOVEM    1,SAVE
          MOVE     1,@(16)       ;LOAD K
          LSH      1,1
          MOVEM    1,@1(16)      ;STORE 2*K IN L
          MOVE     1,SAVE
          JRA      16,2(16)
SAVE:     Z
          END
$EXLDR    MAP
$EOF
Ø1Ø2.........................................3940
$EOJ
```

Example 8

This example contains a diagram of sample control cards for three jobs to be executed under Batch (see Figure A-1) and a sample dialogue produced on the operator's console Teletype while these three jobs are being run.

Following this is the dialogue on the operator's console Teletype produced while the batch in Figure A-1 is run. Underlined strings indicate computer typeouts.

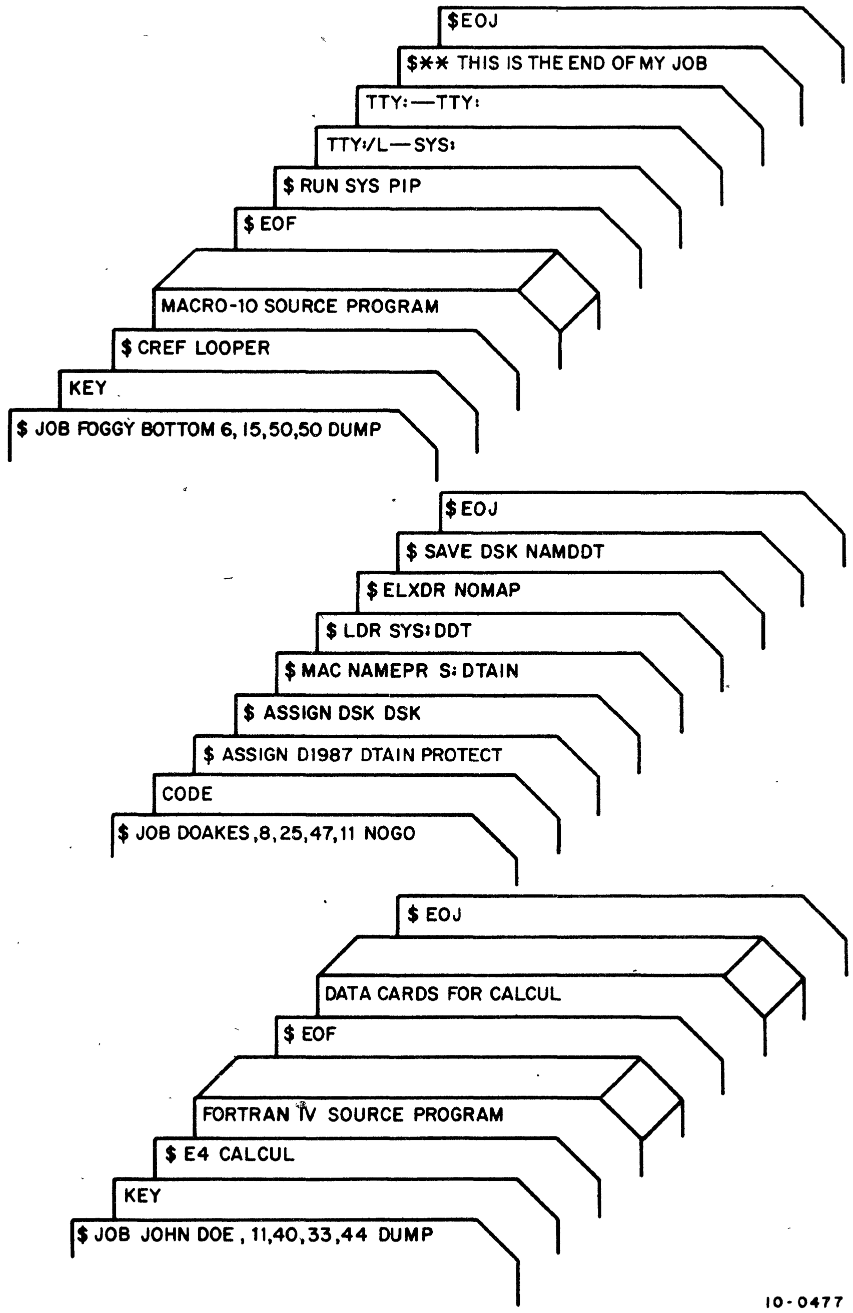


Figure A-1 Sample Control Cards for Three Jobs Processed by Batch

```
                                          (Operator assigns scratch device)
.ASSIGN MTA1 BPTEMP⤸                      (Monitor command to run Batch)
.R BATCH⤸                                 (Card reader input)
*IN CDR:⤸                                 (Line printer output)
*OUT LPT:⤸                                (Skip 0 jobs; start with first)
*SK 0⤸


$JOB JOHN DOE,  11  40  33,44     DUMP
RUNNING JOB 3
  (Compilation and Execution)             ($JOB card always printed on TTY)
RUN TIME - 32 SECS.
$JOB JOE DOAKES,  8  30  47,11 NOGO


RUNNING JOB 3
MOUNT TAPE D1984 WRITE PROTECTED
DTA3⤸


DTA3 NOT AVAILABLE
MOUNT TAPE D1984 WRITE PROTECTED

DTA5⤸
OK
*CONT⤸                                    (Operator mounts tape before typing CONT command)
CONTINUING                                (Execution suppressed)
RUN TIME - 0 SECS.
$JOB FOGGY BOTTOM,  6  70  50,50 DUMP
RUNNING JOB 5
PLEASE MOUNT A SCRATCH TAPE
DTA4⤸

                                          (Operator mounts tape, write enabled, before typing
OK                                        CONT)


*CONT⤸
CONTINUING
PLEASE MOUNT A SCRATCH TAPE                    (Batch needs 2 extra tapes)
MTA0⤸

OK                                        (Operator mounts magnetic tape number 0 write en-
*CONT⤸                                    abled)
CONTINUING                                (Program execution exceeded time limit; e.g., in-
MAX. TIME EXCEEDED                        finite loop)


DUMP⤸                                     (Operator commands core dump on output device,
                                          LPT, and termination of user job)

RUN TIME - 67 SECS.
END OF BATCH                              (At this point operator can run another batch or exit
*                                         with a CTRL C command)
```

APPENDIX B

TABLES

Table B-1
Batch Control Cards

Format	Function
$ASSIGN label symu (PROTECT) (NOREW)	To reference any devices other than the Batch input and output device or system (SYS) device (see 3.1.2.3).
$BIN symu	To specify the previously assigned device to be used to output all relocatable binary files (see 3.1.2.9).
$CBL prgname (B:symu2) (L:symu3) (S:symu1)	Executes a COBOL compilation of user's source program and executes the resultant object program with timing control specified in the $JOB card (3.1.2.9).
$CREF prgname (B:symu2) (S:symu1)	To cause the assembly of a Macro-10 symbolic program and the generation of a special cross-reference listing (see 3.1.2.6).
$DUMP	Inserted at any point in the user's data to produce dump at this point (3.1.2.16).
$EOF	Creates an internal end of file to a preceding input stream on the Batch input device and triggers the running of programs loaded into core and the running of programs assembled (or compiled) but not yet loaded (see 3.1.2.14).
$EOJ	Signals the end of the user's job. If no $EOF card precedes the $EOJ card, the $EOJ card triggers the loading and running actions described above (see 3.1.2.15).
$EXLDR (MAP)	Used only when no more source programs follow; brings the loader into core and loads all programs previously assembled or compiled or specified by preceding $LDR cards (see 3.1.2.8).

Table B-1 (Cont)
Batch Control Cards

Format	Function
$F4 prgname (B:symu2) (L:symu3) (S:symu1)	Executes a FORTRAN compilation of the user's source program and executes the resultant object program with the timing control specified in the $JOB card (see 3.1.2.4).
$GET symu: prgname	Directs Batch to load the program "prgname" from the previously assigned symu. Similar to $RUN card except that the loaded program is not executed (see 3.1.2.12).
$JOB name, core# time# proj#, prog# (NOGO) (DUMP)	The first card of a user job. All compilations, executions, and other tasks requested between this card and an $EOJ card are considered to be part of this job (see 3.1.2.1).
Key	Must follow $JOB card (3.1.2.2) and password associated with Project Programmer number.
$LDR symu:prgname1, prgnam2,...(LIB)	Calls the loader to load relocatable binary files from a device previously assigned by a $ASSIGN card or from the system device or from BPTEMP (see 3.1.2.7).
$MAC prgname (B:symu2) (L:symu3) (S:symu1)	Executes a Macro-10 assembly of the user's source program and executes the resultant object program with the timing control specified in the $JOB card (see 3.1.2.5).
$PAUSE	Suspends Batch until operator intervention (3.1.2.19).
$RUN symu:prgname	Directs Batch to load the program "prgname" from the previously assigned symu (or SYS or BPTEMP) and then execute it (see 3.1.2.11).
$SAVE symu:prgname (core#) $SSAVE symu: prgname (core#)	Directs Batch to save the core image of the user's area on the symu device specified and, if symu is a directory device, assign the name "prgname" (see 3.1.2.10). SSAVE card marks the"prgname" as shareable.
$START	Causes Batch to initiate execution of a program which has been called in with previous $GET card, or which has been linked by loader, (3.1.2.13).

Table B-1 (Cont)
Batch Control Cards

Format	Function
$TAPE symu op1 op2 op3... DZ - Zero DECtape directory. MA - Advance mag tape on file. MW - Rewind mag tape to load point. MB - Backspace mag tape one file. MT - Advance mag tape to logical end of tape.	Directs Batch to dynamically perform certain device-dependent operations on DECtapes or magnetic tapes which have previously been assigned a symbolic-unit name with an $ASSIGN card (see 3.1.2.3).
$*comment	Comment Card. Directs Batch to type out contents of columns 4 through 80 of this card and wait for operator to type CONTINUE or END (see 3.1.2.17).
$**comment	Comment Card. Directs Batch to type out contents of columns 4 through 80 of this card; Batch continues without operator intervention (3.1.2.18).

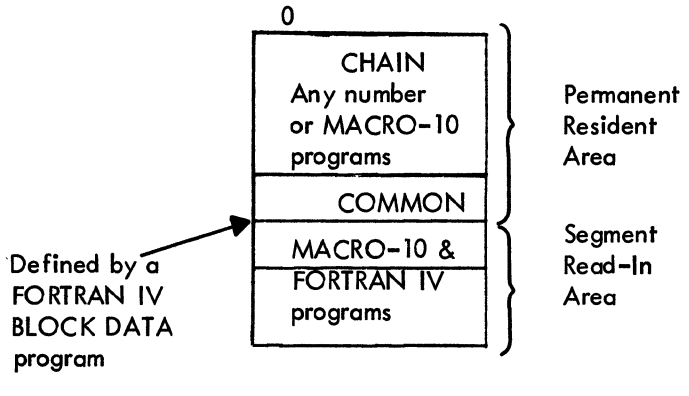
CHAIN

1. ABSTRACT

The CHAIN Program is the prime component of a system that allows users to deal with FORTRAN IV programming applications which would produce programs too large to fit or to load into the amount of core available. Special switches in the LOADER allow the user to create CHAIN files, consisting of complete programs and subroutines, which can be read into core and executed as they are needed. CHAIN itself, a subroutine called by the user, provides a standardized method of reading the successive segments of coding (CHAIN files) into core and linking them to the programs already residing in core.

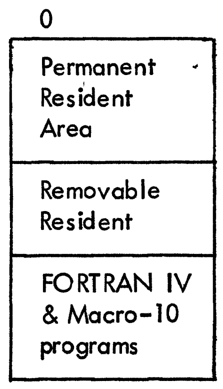
2. PDP-10 CHAIN JOB IMPLEMENTATION

A COMMON area in lower core is set aside for the transmission of data between successive CHAIN files. This area in lower core is known as the Permanent Resident Area, and remains in core at all times. The Permanent Resident Area contains CHAIN and part of LIB40 and may contain any number of Macro-10 programs followed by the FORTRAN IV BLOCK DATA program which defines COMMON. The BLOCK DATA program defines the end of the Permanent Resident Area. There can be only one BLOCK DATA program in any CHAIN job.

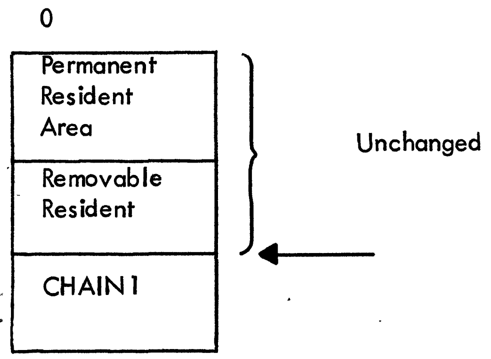


For any CHAIN job the Permanent Resident Area assumes a fixed length. This area is unaffected by the process of reading in successive CHAIN files. The Macro-10 programs residing in the Permanent Resident Area should be programs used by more than one CHAIN file (e.g., DDT or CHAIN). The remainder of core, above the Permanent Resident Area, is known as the Segment Read-in Area. It is in this area that various CHAIN files are read into core and executed. The Segment Read-in Area may contain a mixture of FORTRAN IV and Macro-10 programs. Within this area the user may define a Removable Resident Area. This area is directly above the Permanent Resident Area. It may contain any number of Macro-10 programs followed by one FORTRAN IV program. When the user makes a call to CHAIN, he may specify that the next CHAIN file be read in either directly after the Permanent Resident Area or directly after the Removable Resident Area. Thus the user may leave his Removable Resident in core while several CHAIN files are being read in and executed, and then he may read over it with another file.

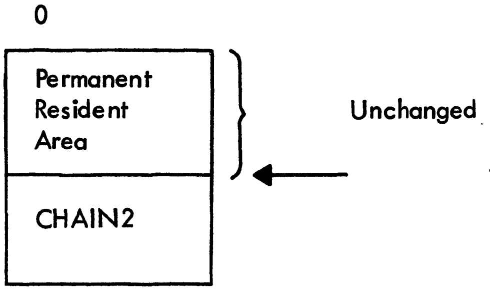
A CHAIN job beginning in this state:



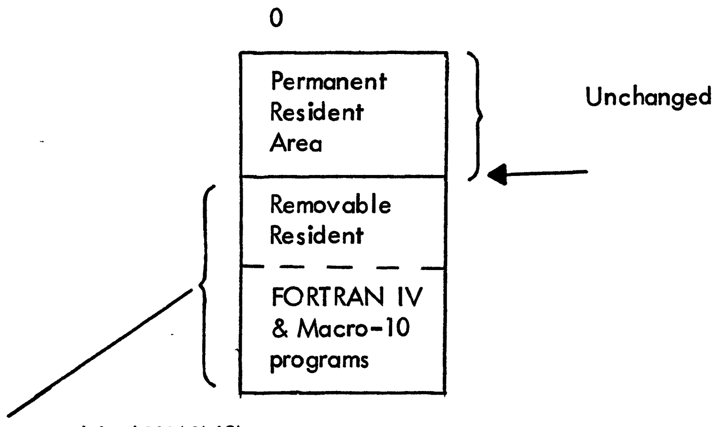
Reads in a new CHAIN file and becomes



Then reads a CHAIN file that overlays the Removable Resident and becomes:



Then reads in a CHAIN file that restores the Removable Resident:



CHAIN file just read in (CHAIN3)

Arrows point to that location in core which is the lower bound into which the file has just been read.

By reading successive CHAIN files into the area above the Removable Resident, the Removable Resident may be kept in core as long as desired. The Removable Resident may be removed at any time by reading a CHAIN file into core immediately above the Permanent Resident Area. The Removable Resident may be restored at any time by reading a new CHAIN file containing a copy of the Removable Resident into core immediately above the Permanent Resident Area.

3. LOADING CHAIN

CHAIN will be loaded from SYS by the Loader. This loading <u>must</u> be followed by a library search of LIB40 before the loading of block data so that essential Macro programs are loaded in the PRA.

4. CALLING SEQUENCE

The call to CHAIN is of the form:

CALL CHAIN (TYPE, DEVICE, FILE)

where: TYPE = 0 reads the next CHAIN file into core immediately above the Permanent Resident Area (COMMON).

TYPE = 1 reads the next CHAIN file into core immediately above the FORTRAN IV program which marks the end of the Removable Resident.

where: DEVICE = 1, 2,. . . is an I / O device number corresponding to the device where the CHAIN file may be found. For example, when reading a CHAIN file from the disk, DEVICE = 1. NOTE: CHAIN files may be stored in DSK, MTA, or DTA only.

where: FILE = 1, 2,... is the number of the magnetic tape file where the CHAIN file is located.

or: FILE = 0 tells CHAIN to read the next file from the selected magnetic tape.

or: FILE = 'ASCII filename' if the CHAIN file is to be read from DTA or DSK.

NOTE: File names are limited to five or fewer letters according to the FORTRAN IV convention.

EXAMPLE: If a CHAIN job wishes to read in a CHAIN file named SEG4 from the disk, and it wishes to read that file into the area above the Removable Resident, the call to CHAIN would be:

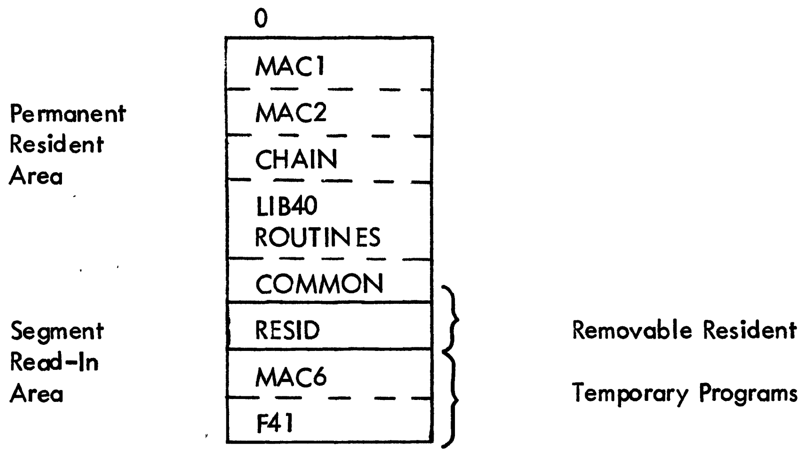
CALL CHAIN (1,1,'SEG4')

5. CREATION OF CHAIN FILES

Using LOADER[1], the creation of CHAIN files is a two-step operation.

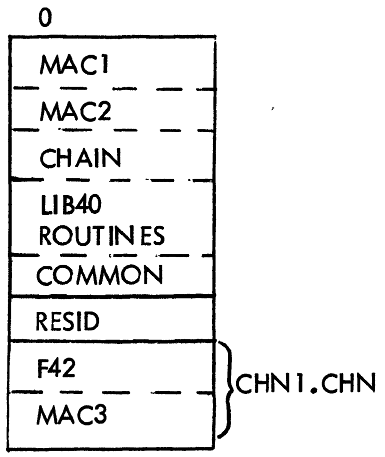
First, one uses LOADER to create an image of core as it will look when the CHAIN file being created is read in at run time. For instance, consider the CHAIN job illustrated below:

[1]Version 51 or later of the LOADER is necessary.



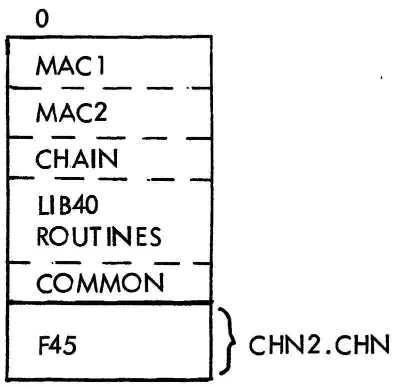
Stage I

A call to CHAIN of the form CALL CHAIN(1,1,'CHN1') would produce:



Stage II

A call to CHAIN of the form CALL CHAIN(0,1,'CHN2') would produce:



Stage III

The first step in the creation of the file CHN1 would be to input to LOADER all the REL files corresponding to the programs which will be in core at STAGE III. In this case, the command to LOADER would be:

```
*DSK:MAC1,MAC2,CHAIN/FDSK:COMMON,RESID/FDSK:F42,MAC3
```

where F42 and MAC3 constitute the basis of the CHAIN file to be created. The /F switch is used to force the loading of library programs so that they will be placed in lower core where they can be properly and permanently linked to the resident programs which call them. Commas must not follow a /F switch because the LOADER tries to load LIB40 twice.

Second, a command is given that will cause the remaining library routines to be loaded and the completed CHAIN file, consisting of everything above the Removable Resident, to be written out onto the specified output device. In the case of the example above, the command might be:

```
*DSK:CHN1←/R $
```

where "DSK:CHN1←" instructs LOADER to write the CHAIN file named CHN1.CHN on the disk. /R instructs LOADER that the CHAIN file consists of everything above the Removable Resident, and $ (ALTMODE key) instructs LOADER to first load the necessary library routines, and then write out the CHAIN file.

> NOTE: LOADER puts all library routines referenced only by the programs in a CHAIN file into core above the last user program MAC3 in the example), and they are written out as part of the CHAIN file.

Similarly, the CHAIN file CHN2 would have been created by giving to LOADER the command:

```
*DSK:MACl,MAC2,CHAIN/FDSK:COMMON,F45
```

where F45 (and the library routines it references) constitutes the body of the CHAIN file. The subsequent command:

```
*DSK:CHN2←/C $
```

would instruct LOADER to do the necessary library search and write out the CHAIN file CHN2.CHN starting with the program after COMMON.

The original core image in a CHAIN job (see Stage I above) is a LOADER-created SAV mode core image. In this case, it would be created by using the LOADER command string:

```
*DSK:MAC1,MAC2,CHAIN/FDSK:COMMON,RESID/FDSK:MAC6,F41 $
```

In summary, when one creates CHAIN files or the original core image used in a CHAIN job, the Permanent Resident Area must be constructed in exactly the same way each time LOADER is run. The same rule applies to the creation of Removable Residents, when they exist. To this end, the /F switch of the LOADER should be used to force the uniform loading of library routines referenced by programs in the Permanent Resident Area (PRA) or in the Removable Resident (RR). The /F switch should be given immediately after the last program in the PRA and the RR.[1] When an /F switch is set, LOADER loads the previously specified file, then enters library search mode and scans the FORTRAN library loading the required programs only. The /F switch terminates a specification; therefore, it must not be followed by a comma as this also terminates a specification causing the LOADER to load the last file specified again. Also note that the user must explicitly load CHAIN into his PRA.

6. PROGRAMMING CONSIDERATIONS

a. When a call to CHAIN is successful, a new CHAIN file will be read in and execution begins at the starting address of the CHAIN file which has just been inputted. This starting address is governed by the usual LOADER rules and switches. This allows the user to specify different starting addresses for each link that he loads. As part of its binary output, the FORTRAN IV Compiler produces a starting address for each main program that it compiles, and this information is used by LOADER to tell where to start a loaded program at run time. Similarly, the END statement in a Macro-10 program can be used to specify a starting address. When loading more than one program, LOADER accepts the starting address of the last program loaded which has a starting address as the starting address of the whole group of programs unless the /I switch to LOADER is used. (See LOADER writeup for details.)

b. CHAIN releases all I/O devices in use at the time the call to CHAIN is made. Thus I/O started by the programs of one CHAIN file cannot be completed by the programs of another CHAIN file. In utilizing file-oriented devices, data must be organized into separate files for each CHAIN link. It is recommended that the user release all channels after the return from CHAIN. (This is done automatically by the first FORTRAN main program in each CHAIN file.)

[1] The library program loaded after the FORTRAN IV program which defines the end of the RR will be written out as part of the created CHAIN file.

c. When using the optional Removable Resident, the length of the Removable Resident may not be changed during a particular CHAIN job. If a Removable Resident is removed and then replaced, it must be replaced by a Removable Resident of the same length.,

d. It is not necessary that all CHAIN files be the same size. However, the user must assign to his job enough core to accommodate the largest CHAIN file.

While CHAIN is loading a new link, three errors may occur:

1. The device specified in the call to CHAIN is not available. This causes the message:

   DEVICE xxx NOT AVAILABLE - CHAIN

   followed by a call to EXIT.

2. The filename specified with DECtape or disk input could not be found. This causes the message:

   FILE xxx NOT FOUND - CHAIN

   followed by a call to EXIT.

3. An input error occurred while loading the actual dump file. Since CHAIN performs this input from the accumulators, its space is limited.

If a read error has occurred, CHAIN will execute a "Halt" instruction in location 14, an d the Monitor will print a message on the Teletype.

7. PROGRAM DESCRIPTION

When CHAIN is called, it does a RELEASE on all I/O devices. If the specified input device is magnetic tape, the tape is positioned as specified in the call to CHAIN, otherwise a LOOKUP is done on the appropriate directory device. A CHAIN file contains an image of the appropriate portion of upper core. It also contains five preceding words which update information in the JOBDATA area (JOB41, JOBSA, JOBSYM, JOBDDT, and length of CHAIN file). Thus five words at the end of the BLOCK DATA area (COMMON) or the Removable Resident are used while the new CHAIN file is being read into core. The actual input of the new CHAIN file is done by a portion of coding placed in the accumulators by CHAIN. This coding in the accumulators does the input of the CHAIN file, updates JOB41, JOBSA, JOBSYM, and JOBDDT and length of CHAIN file, and restores the five words to the area just below the CHAIN file that has been read in. Control is then transferred to the starting address for the CHAIN file just read in. However, the contents of all accumulators except 17 are altered by CHAIN.

When CHAIN is called, it does not know the length of the incoming CHAIN file. This lack of information forces CHAIN to INPUT the file into the whole of available core. This lengthens the time required to complete the reading of a CHAIN file which is stored on DECtape or disk and which is smaller than the amount of core available. This happens because input from these devices is not completed until sufficient data is transmitted to fill the amount of core specified with the INPUT commands[1]. It is to the user's advantage to load all CHAIN files into core images of approximately equal length so as to minimize the time required to input CHAIN files from DSK or DTA.

With this version of CHAIN(V50) and LOADER(V51), the removable resident area, when used, must remain a constant length. It is not possible to read in a link that destroys the original removable resident and then read in another link that replaces the removable resident with one of a different length. This will be fixed in further versions of CHAIN and LOADER by preserving one more word from the Job Data Area, JOBCHN.

[1] However, after input is complete, CHAIN knows how much data should have been read and checks for too little data transferred.

LINED

LINED
A LINE EDITOR FOR PDP-10 DISK FILES

LINED is a line editor for disk files. It is used to create and edit source program files which are written on disk in ASCII code with line sequence numbers appended. LINED has the ability to reference any line at any time without the user having to close and reopen the file. LINED is a reentrant program and loads in 2K pure and 2K impure segments of core.

NOTE

In this document, computer typeouts are indicated by underscoring. The symbol ↲ represents the RETURN key. The symbol $ represents the ALTMODE key.

1. MONITOR COMMANDS

The MONITOR commands CREATE and EDIT may be used to select a file for editing with LINED. A temporary disk file, called ###EDT.TMP, is created for passing the commands to LINED.

1.1 The CREATE Command

The CREATE command calls in LINED and opens the specified new disk file for editing. The CREATE command is of the form:

```
. CREATE filename.ext ↲
```

1.2 The EDIT Command

The EDIT command calls in LINED and opens the specified existing disk file for editing. The EDIT command is of the form:

```
. EDIT filename.ext ↲
```

2. LINED COMMANDS

LINED indicates its readiness to receive commands by typing an asterisk. At this time LINED is said to be in command mode. The user may then type in the following LINED commands.

2.1 Inserting or Replacing a Line

```
*Innnnn
nnnn  aaaa......a
nnnxx  $
*
```

Insert or replace the following typed line at line number nnnnn of the currently open file; nnnnn can be specified as a line sequence number or a point (.), or it can be omitted entirely. A point refers to the last line which was typed, or the last line deleted, or the last line inserted. If nnnnn is omitted, it is assumed to be 10.

When LINED has typed a line sequence number, the program enters text mode. In the text mode, characters typed by the user are understood to be text for the insertion. Following the user's typein of the line to be inserted, LINED types out the next sequential line number (nnnnn+10) following which the user presses the ALTMODE key (sometimes labeled PREFIX or ESC) to terminate the insert process and return to LINED command level.

If there already exists a line at nnnnn, it will be replaced. A single quote following the line number indicates that insertion at this line number will cause the existing line to be replaced.

2.2 Inserting Multiple Lines

```
*Innnnn,iiiii
nnnnn   aaaaa......a
nnnxx   bbbbb......b
   .
   .
nnnyy  $
*
```

Insert the following typed lines, beginning at line number nnnnn (which can be specified as either a line number or a point) of the currently open file. Each time a line is entered, nnnnn is increased by the specified increment, iiiii. If iiiii is omitted, it is assumed to be 10 (if iiiii has never been specified previously), or the previous increment specified.

If nnnnn is omitted, it is assumed to be 10, and the result becomes the line number of the next insertion. Type ALTMODE on the line following the last insertion to return to LINED command mode. LINED then awaits another command.

A double quote following a line number indicates that the increment specified for the current insert instruction has resulted in an existing line being skipped.

2.3 Deleting a Line

```
*Dnnnnn
```

Delete a line number nnnnn from the currently open file; nnnnn can be specified as either a line sequence number or a point

2.4 Deleting Multiple Lines

*Dmmmmm,nnnnn	Delete lines mmmmm through nnnnn from the currently open file; mmmmm must be less than nnnnn. Either mmmmm or nnnnn may be specified as a point as long as mmmmm is less than nnnnn.

2.5 Printing a Line

*Pnnnnn	Print line nnnnn on the user's Teletype; nnnnn can be specified as either a line sequence number or a point. Typing ALTMODE following a typeout will cause the next sequential line to be printed.

2.6 Printing Multiple Lines

*Pmmmmm,nnnnn	Print lines mmmmm through nnnnn of the currently open file; mmmmm must be less than nnnnn. Either mmmmm or nnnnn may be specified as a point as long as mmmmm is less than nnnnn.

2.7 Closing the Current File

E ↲	Closes the current file and returns to LINED command mode. At this point, the user may either open another file or type ↑C to return to Monitor level to assemble, list, and/or save his file on a permanent storage device (e.g., DECtape).

2.8 Examples of Command Sequence

Example 1

```
.CREATE FILEA                 RUN LINED AND OPEN FILE FILEA

*I1Ø                          BEGIN INSERTING LINES AT LINE NUMBER
ØØØ1Ø THE PROGRAM             1Ø INCREMENTING BY 1Ø.
ØØØ2Ø IS INSERTED
ØØØ3Ø HERE
  .
  .
  .
ØØ35Ø($)                      RETURN CONTROL TO LINED COMMAND
*E                            MODE BY TYPING ($) . CLOSE FILE FILEA
*↑C                           BY TYPING AN E. TYPING A ↑C RE-
.                             TURNS TO THE MONITOR COMMAND LEVEL.
```

Example 2

```
.EDIT FILEA                      RUN LINED AND OPEN EXISTING FILE
                                 FILEA
*P1Ø,3Ø                          PRINT LINES 10 THROUGH 30
ØØØ1Ø  THE PROGRAM               PRINTOUT
ØØØ2Ø  IS INSERTED
ØØØ3Ø  HERE
*I2Ø                             INSERT LINE 20
ØØØ2Ø IS PLACED
ØØØ3Ø ($)
*D3Ø                             DELETE LINE 30
*P 1Ø,3Ø                         PRINT LINES 10 THROUGH 30
ØØØ1Ø  THE PROGRAM               PRINTOUT
ØØØ2Ø  IS PLACED
*E                               TYPE E TO CLOSE FILE FILEA
*↑C                              TYPING A   ↑C  RETURNS JOB TO MONITOR
.                                CONTROL LEVEL.
```

3. AUXILIARY COMMANDS

These Auxiliary Commands provide an alternate method of calling LINED and opening files. In most cases, auxiliary commands can be replaced by the monitor instructions CREATE and EDIT (Section 1).

3.1 R LINED

LINED can be called in from the system device by typing

```
.R LINED ↵
*
```

LINED responds with an asterisk to indicate its readiness to receive a command.

3.2 Initializing a File for Processing

S filename.ext ↵	Select an existing disk file, filename.ext, for editing.
S filename.ext ($)	Select (create) a new disk file for editing, calling it filename.ext.

4. LINED CONVENTIONS AND RESTRICTIONS

The following conventions and restrictions should be noted.

a. Files are written with the installation standard protection. See Book 1 for explanation of protected files.

b. When in insert mode, typing ALTMODE following the printout of the next insertion line sequence number causes a returned to LINED command level. Typing ALTMODE to terminate a line of text to be inserted causes the text line to be ignored.

```
Ø0Ø1Ø  LINE OF TEXT
ØØØ2Ø  $                        Returns to LINED command level
*


ØØØ1Ø  LINE OF TEXT  $    Line is ignored
*
```

c. LINED assumes that all blocks in a disk file have an integral number of lines (i.e., each block begins with a sequence number and no line is split between blocks). This will always be the case with files which have been created and edited only with LINED; however, if sequence numbers have been removed, say by TECO, they may be restored by using PIP switch /A (see PDP-10 Reference Handbook.)

d. LINED files can be resequenced using PIP switch /A (see PDP-10 Reference Handbook).

e. Line number 0 is illegal and cannot be used.

f. Lines can be edited in any order; however, editing lines by ascending line numbers reduces file access time.

5. ERROR HANDLING

When an error is detected, LINED types a message and returns the user to LINED command level (indicated by the output of an * on the Teletype). Some errors are fatal and cause control to return to the monitor. Error messages for LINED are given in Table 1.

Table 1
LINED Error Messages

Message	Meaning
?FAU *	Filename Already in Use. The filename specified for a newly created file already exists on the disk. Retype command correctly.
?ILC*	ILlegal Command. Illegal syntax or other error in command string.

Table 1 (Cont.)

Message	Meaning
?NCF*	Not Current File. The filename in an "S filename" command could not be found on the disk.
?NFO*	No File Open. No "S filename" command preceded this command string.
?NLN*	Nonexistent Line Number. A Print or Delete command refers to a nonexistent line sequence number.
	NOTE The following are internal system errors.
?CCL*	CCL error. Error while referencing CCL comand file.
?COR*	No core available for data segment.
?DCE*	Device directory full.
?DDE*	Device Data Error. Read or write failure on disk.
?UNA*	Unit Not Available. The disk is not available.

6. IMPLEMENTATION

The following explanation is intended to help the user to understand how LINED works so that he may use it more effectively.

Lines of text are stored in a 1000-word working buffer. Each line has a 1-word header containing two items. The left half contains the sequence number of the line, and the right half contains the number of words (including the word containing the line header) needed to store the line of text. Thus, to find the beginning of the next line of text, it is necessary to simply take the address of the current line header and add the word count of the current line.

Several pointer words are used to keep track of the lines in the working buffer. WRTLST contains the sequence number of the highest line in the buffer. SN contains the sequence number of the line currently being handled in a command.

When LINED discovers that SN is greater than WRTLST, it knows that the line being sought has already passed through the working buffer. This line is not directly accessible, because there is no way to read a disk file backwards. Consequently, it is necessary for LINED to close the file and then reopen it. This process of going from the current position of the file to the end of the file, from there to the beginning of the file, and finally to the line being sought is accomplished as follows:

a. To close the file, all remaining text must be passed through the working buffer to the temporary output file (called ###LIN.TMP). This is done by giving the subroutine FNDLIN (which finds a line whose sequence number is SN) the highest possible sequence number - 99999.

b. Next, the original file is renamed to ###TMP.TMP, the temporary output file is renamed to the original filename and the original file (###TMP.TMP) is renamed to name.BAK (same name as original with an extension of BAK).

c. FNDLIN is then given the sequence number being sought, and LINED continues with the original command.

TECO

CHAPTER 1

INTRODUCTION TO TECO

TECO, a very powerful text editor, enables the advanced PDP-10 user to edit any ASCII text with a minimum of effort. All editing can be accomplished by using only a few simple commands; or the user may select any of a large set of sophisticated commands such as character string searching, command repetition, conditional commands, programmed editing, and text block movement. In this description of TECO only the basic commands are described. If the user requires information about the more advanced uses of TECO, he can refer to the TECO section of the PDP-10 Reference Handbook.

TECO is a character-oriented editor. One or more characters in a line can be modified without retyping the rest of the line. Any sort of document can be edited: programs written in FORTRAN, COBOL, MACRO-10, or any other language; memoranda; specifications; and other types of arbitrarily formatted text. TECO does not require that line numbers or any other extraneous information be associated with the text.

1.1 GENERAL OPERATING PROCEDURE

TECO operates on ASCII data files. A file is an ordered set of data on some peripheral device. In the case of TECO, a data file is some type of document. An input file may be a named file on disk or DECtape, a file on magnetic tape, a deck of punched cards, or a punched paper tape. An output file can be written onto any of these same devices. The input file for a given editing operation is the file to which the user wishes to make changes. If the user is using TECO to create a new file, there is no input file. The output file is either the newly created file or the edited version of the input file. An output file is not required if the user wishes merely to examine a file without making any changes.

In general, the process of editing proceeds as follows. The user first specifies the file he wishes to edit and then reads in a "page" of text. A page is normally an amount of text that is intended for a single sheet of paper. Form feeds are used to separate a document into pages. On input, TECO interprets form feeds as end-of-page indicators. It is not required, however, that a document be so divided into pages. If a form feed is not encountered, TECO simply reads as much text as will reasonably fit into its editing buffer. For the purposes of this document, the word page is used to mean the segment of text in TECO's editing buffer.

When a page has been read into the buffer, the user can modify this text by using the various editing commands. When he has finished editing the page, he outputs it and reads in the next page. This process continues until, after the last page has been output, the user closes the output file. If there are several pages where no editing is required, there are commands which may be used to skim over them.

1.2 INITIALIZATION

The two main uses of TECO are (1) to create a new disk file, and (2) to edit an existing disk file. These are the only uses of TECO described in this document. In particular, the use of TECO with devices other than disk is not described. The beginner can get around this limitation by using PIP to transfer files to and from disk. (Refer to Book 6 in the PDP-10 Reference Handbook for information about PIP.)

The two main uses of TECO are so common that there are direct monitor commands to initialize TECO for executing them. The command

```
.MAKE    filename.ext↵
```

is used to initialize TECO for creating a new disk file. Filename.ext is the name that the user gives to the new file. The filename can be from one to six alphanumeric characters. This is followed (optionally) by a period (.) and a filename extension of from one to three alphanumeric characters. The most commonly used filename extensions are:

.F4	for FORTRAN source programs
.CBL	for COBOL source programs
.MAC	for MACRO-10 source programs

The MAKE command opens a new disk file to receive output from TECO and gives it the name specified by the user. Once the file has been opened it is then actually created by using the insert and output commands, which are explained in sections 2.5 and 2.6 of this document.

The command

```
.TECO    filename.ext↵
```

is used to initialize TECO for editing an existing disk file, named filename.ext. The filename and filename extension must be exactly the same as those of the file that is to be edited. The TECO command opens the specified file for input by TECO and opens a new file, with a temporary name, for output of the edited version. When output of the new version is completed, the original version of the file is automatically renamed filename.BAK, and the newly edited version is given the name of the original file. The filename extension .BAK is used for backup files.

After TECO has been initialized for a particular job, it responds by typing an asterisk (*) . The asterisk indicates that TECO is ready to accept commands; it is typed at the beginning of TECO's operation and at the completion of execution of every command string.

Examples:

. MAKE EARNNG.F4 ↵ * _	This command initializes TECO for creation of a new disk file called EARNNG.F4. The extension .F4 is used because the file is to be a FORTRAN source file.
. TECO LIB40.MAC ↵ * _	This command initializes TECO for editing the existing disk file LIB40.MAC. At the completion of editing, TECO automatically changes the name of the original version of LIB40.MAC to LIB40.BAK and gives the name LIB40.MAC to the new version.

NOTE

The TECO command cannot be used to edit a file which has the filename extension .BAK. To edit a backup file the user must first rename the backup file. For example, to edit the file LIB40.BAK the user should proceed as follows:

```
. RENAME    LIB40.OLD=LIB40.BAK ↵
.TECO     LIB40.OLD ↵
*
_
```

1.3 SPECIAL SYMBOLS USED IN THIS DOCUMENT

Symbol	Character Represented	Comment
↵	Carriage Return	Whenever the RETURN key is typed, TECO automatically appends a line feed to the carriage return.
Ⓢ	Altmode	On most Teletypes, the altmode key is labeled "ALTMODE", but on some

Symbol	Character Represented	Comment
		it is labeled "ESC" or "PREFIX". Since the altmode is a non-printing character, TECO indicates that it has received an altmode type-in by echoing a dollar sign ($).
↑C	Control C	This character is typed by typing the letter C while holding down the CTRL key. Other control characters are represented in similar fashion.
(FORM)	Form Feed	Form feed is typed by typing ↑F (control F).
↓	Line Feed	This symbol is used only when a line feed is explicitly typed. It is not used for the line feed which is automatically assumed when a carriage return is typed.
→|	Tab	Tab is typed by typing ↑I (control I).
△	Space	This symbol is used occasionally for emphasis.
(RO)	Rubout	This key is used to nullify a character erroneously typed in a command string. Its use is explained fully in Section 1.5.

1.4 GENERAL COMMAND STRING SYNTAX

TECO commands are usually given by typing the one- or two- letter name of the command. However, many of the commands take arguments. Some typical examples are shown below, to give the reader an idea how TECO commands look. These commands are fully explained later in the manual.

```
L
PW
ISAMPLE ($)
3K
```

TECO commands may be given one at a time. However, it is usually more convenient to type, in a single command string, several commands that form a logical group. An example of a command string is shown below.

$\underline{*}$ YIHEADING$\textcircled{\$}$NTAG:$\textcircled{\$}$2LT$\textcircled{\$}\textcircled{\$}$

A command string may be typed after TECO indicates its readiness by printing an asterisk . Command strings are formed by merely writing one command after another. Command strings are terminated by typing two consecutive altmodes.

Execution of the command string begins only after the double altmode has been typed. At that point each command in the string is executed in turn, starting at the left. When all commands in the string have been executed, TECO prints another asterisk, indicating its readiness to accept another command.

If some command in the string cannot be executed because of a command error, execution of the command string stops at that point, and an error message is printed. Commands preceding the bad command are executed. The bad command and those following it are not executed.

1.5 ERASING COMMANDS

Typographical errors, if discovered while typing a command string, may be "erased" by use of the rubout key. This process is best explained by an example.

$\underline{*}$ 3LKILEIF ERICXON

After typing this much of the command string, the user discovers that he has misspelled the name "Ericson." To nullify his error, he types three successive rubouts. As he does this, TECO responds by retyping the characters which are being rubbed out.

$\underline{*}$ 3LKILEIF ERICXON $\textcircled{\text{RO}}$ $\underline{\text{N}}$$\textcircled{\text{RO}}$ $\underline{\text{O}}$$\textcircled{\text{RO}}$ $\underline{\text{X}}$

Of course, rubout is a non-printing character so the actual line looks like this:

$\underline{*}$ 3LKILEIF ERICXON$\underline{\text{NOX}}$

Once he has rubbed out the bad character, the user continues the command string from the last correct character.

```
* 3LKILEIF ERICXONNOXSON $ 0LT $$
```

The actual function of the rubout character is to delete the last typed character in the command string. Consequently, if the bad character is not the last in the string, all characters back to that point must be deleted. Rubout characters do not enter the command string.

An entire command string may be erased, if it has not yet been terminated, by typing two successive ↑G (control G) characters.

Example:

```
* 3LKILIEF ERICXON ↑G↑G
```

↑G ↑G causes the entire command string to be rejected. TECO types a new asterisk and awaits a new command.

1.6 COMMAND ARGUMENTS

There are two types of arguments for TECO commands. Some commands require numeric arguments and some require alphanumeric (text) arguments.

Numeric arguments, and also all numeric type-outs by TECO, are decimal integers. Numeric arguments always precede the command to which they apply. A typical example of a command taking a numeric argument is the command to delete three characters: "3D".

Alphanumeric arguments are textual arguments meant to be interpreted as ASCII code by TECO. Alphanumeric arguments always follow the command to which they apply, and they must always be terminated by an altmode. Examples of alphanumeric arguments are (1) text to be inserted, and (2) character strings to be searched for.

Example:

```
*ISOMETHING $$
```

The argument is "SOMETHING".

As shown in the above example, the altmode used to terminate an alphanumeric argument may also serve as one of the two altmodes necessary to terminate a command string. Any ASCII character except null, altmode, and rubout may be included in an alphanumeric argument.

CHAPTER 2

TECO COMMANDS

2.1 INPUT COMMANDS

The Y (yank) command first clears the editing buffer and then reads the next page of the input file into the buffer.

A single Y command is automatically performed by the command

```
.TECO    filename.ext ⤶
```

so that when editing with this command the first page of the input file is automatically read in before TECO prints the first asterisk.

The Y command may be used to delete entire pages of a file, since the editing buffer is completely cleared before the input is performed.

The A (append) command reads in the next page of the input file without clearing the current contents of the editing buffer. This command is used to combine several pages of a document. When the A command is used, the form feed separating the page already in the buffer and the page to be read in is removed. Thus after the A command the two pages are combined into one.

If the editing buffer does not have enough room to accommodate an A command which has been given, TECO automatically expands its buffer and then executes the A command. The user is notified of this action by a message of the following form

```
[3K CORE]
```

If sufficient core is not available to allow buffer expansion, the user is notified by an error message.

NOTE

On either an A or a Y command the form feed terminating the page to be read in is not actually read into the buffer. It is removed on input and a single form feed is appended to the end of the buffer when the buffer is output.

Examples:

```
.TECO  REPORT.CBL↵
*
```

This command, as part of the process of initializing TECO for editing the disk file REPORT.CBL, automatically clears the buffer and then reads in the first page of the file.

```
*Y$$
*
```

This command deletes the entire contents of the buffer and then reads in the next page of the input file.

```
*AA$$
*
```

Read the next two pages of the input file into the buffer, combining them with the page already in the buffer.

```
*A$$
[4K CORE]
*
```

The buffer is expanded as required by the A command. In most cases this message need be of no concern to the user. It is important only if the system is low on core and does not have swapping capability.

2.2 BUFFER POINTER POSITIONING

Since TECO is a character-oriented editor, it is very important that the user understand the concept of the buffer pointer. The position of the buffer pointer determines the effect of many of the editing commands. For example, insertion and deletion always take place at the current position of the buffer pointer.

The buffer pointer is simply a movable position indicator. It is always positioned between two characters in the editing buffer, or before the first character in the buffer, or after the last character in the buffer. It is never positioned " on " a particular character, but rather before or after the character. The pointer may be moved forward or backward over any number of characters.

The J command moves the buffer pointer to the beginning of the buffer, i.e., to the position immediately before the first character in the buffer.

The ZJ command moves the pointer to the end of the buffer, i.e., to the position following the last character in the buffer.

The C command advances the pointer over one character in the buffer. The C command may be preceded by a (decimal) numeric argument. The command nC moves the pointer forward over n characters. (The pointer cannot be advanced beyond the end of the buffer.)

The R command moves the pointer backward over one character in the buffer. This command may also be preceded by a numeric argument. The command nR moves the pointer backward over n characters. (The pointer cannot be moved backward beyond the beginning of the buffer.)

The L command is used to advance the buffer pointer or move it backward, on a line-by-line basis. The L command takes a numeric argument, which may be positive, negative, or zero, and is understood to be one (1) if omitted.

The action of the L command with various arguments is best explained in a more concrete way. Suppose the buffer pointer is positioned at the beginning of line b, or at some position within line b.

The command L, or 1L, advances the pointer to the beginning of line b+1, i.e., to the position following the line feed which terminates line b.

The command nL, where $n > 0$, advances the pointer to the beginning of line b+n.

The command 0L moves the pointer to the beginning of line b. If the pointer is already at the beginning of line b, nothing happens.

The command -L moves the pointer back to the beginning of line b-1.

The command -nL moves the pointer back to the beginning of line b-n.

NOTE

After execution of a Y command, the buffer pointer is always positioned before the first character in the buffer. (The Y command automatically executes an implicit J command.) The A command does not change the position of the buffer pointer.

In examples, the position of the buffer pointer is often represented in this manual by the symbol ↑ just below the line of text.

Examples:

`*J3L$$` `*`	The J command moves the pointer to the beginning of the first line in the buffer. The 3L command then moves it to the beginning of the fourth line.
`*ZJ-2L$$` `*`	This moves the pointer to the beginning of the next to last line in the buffer.
`*L4C$$` `*`	Advance the pointer to the position following the fourth character in the next line.
`*0L2R$$` `*`	0L moves the pointer back to the beginning of the line it is currently on. Then 2R moves it back over the carriage return-line feed pair which terminates the preceding line.

`ABCDEF`
`  ↑`
	In this example of text stored in the buffer, the position of the buffer pointer is shown to be between B and C.

2.3 TEXT TYPE-OUT

Various parts of the text in the buffer can be typed out for examination. This is done by use of the T command. Just what is typed out depends on the position of the buffer pointer and the argument given. The T command never moves the buffer pointer.

The command T types out everything from the buffer pointer through the next line feed. Thus, if the pointer is at the beginning of a line, the command T causes that line to be typed out. If the pointer is in the middle of a line, T causes the portion of the line following the pointer to be typed.

The command nT (n > 0) is used to type out n lines, i.e., everything from the buffer pointer through the nth line feed following it.

The command 0T types out everything from the beginning of the current line up to the buffer pointer. This is useful for determining the position of the pointer.

The command HT types out the entire contents of the buffer.

The user, especially one new to TECO, should use the T command often, to make sure the buffer pointer is where he thinks it is.

During execution of any T command, the user may stop the Teletype output by typing the ↑O (control O) character. This command causes TECO to finish execution of the command string, omitting all further type-out. The ↑O command does not carry over to the next command string.

Examples:

```
* 0LT $$
ENTIRE LINE TYPED
*
```

This command string is used to move the pointer back to the beginning of a line and then type out the entire line. It is frequently used after insertion and search commands.

```
* 0TT $$
ENTIRE LINE TYPED
*
```

This command string causes the entire line to be typed without moving the pointer. It is useful after insertion and search commands when it is not convenient to move the pointer back to the beginning of the line.

```
* 2T $$
EF
GHIJKL
*
```

If the buffer contains the text below with the pointer between D and E,

```
ABCD‸EF↵↓
GHIJKL↵↓
MNOPQR↵↓
```

this command causes the typeout shown.

"ABCD" is not typed because these characters precede the pointer. MNOPQR is not typed because these characters follow the second line feed.

2.4 DELETION COMMANDS

Characters are deleted individually by using the D command. The command D deletes the character immediately following the buffer pointer. The command nD, where n > 0, deletes the n characters immediately following the pointer. The commands -D and -nD delete the character or the n characters, respectively, which immediately precede the buffer pointer.

Lines are deleted using the K command. The K command may be preceded by a numeric argument, which is understood to be 1, if omitted. The command nK (n> 0) deletes everything from the current position of the pointer through the nth line-feed character following the pointer. The command HK deletes the entire contents of the buffer.

At the conclusion of a D or K command the buffer pointer is positioned between the characters which precede and follow the deletion.

Examples:

The editing buffer contains the following three lines of text, and the pointer is positioned between the G and H.

ABCDEFG‸HIJKLM↵↓
NOPQRSTUVWXYZ↵↓
1234567890↵↓

* 4D $$	Delete HIJK.
*	
* -D $$	Delete G.
*	
* -3D $$	Delete EFG.
*	
* 7D $$	Delete HIJKLM↵ but do not delete the line feed at the end of the first line.
*	
* K $$	Delete HIJKLM↵↓.
*	Since the carriage return and line feed at the end of the first line are deleted, the text in the buffer after this command would be: ABCDEFGNOPQRSTUVWXYZ↵↓ 1234567890↵↓
* 2K10D $$	This would leave the buffer containing only ABCDEFG↵↓.
*	
* 0LK $$	This is the command string that is required to kill (delete) the entire first line.
*	
* L2K $$	This kills the last two lines.
*	
* HK $$	Kill the entire buffer.
*	

2.5 INSERTION COMMAND

The only insertion command is the I command. The ASCII text that is to be inserted into the buffer is typed immediately after the letter I. The text to be inserted is terminated by an altmode.

Any ASCII character except null, altmode, and rubout may be included in the text to be inserted. Specifically, spaces, tabs, carriage returns, form feeds, line feeds, and control characters are all allowed. If a carriage return is typed in an insertion, it is automatically followed by a line feed.

The text to be inserted is placed in the buffer at the position of the buffer pointer, i.e., between the characters. At the conclusion of the insertion command the buffer pointer is positioned at the end of the insertion.

Any number of lines may be inserted with a single I command. For the user's protection, however, no more than 10 to 20 lines should be inserted with each I command.

Examples:

If the buffer contains ABCD↑EF↵↓ with the pointer between D and E, the command

```
*IXYZ $$                 produces  ABCDXYZ↑EF↵↓
*

*I↵                      produces  ABCD↵↓
$$                                 ↑EF↵↓
*

*I↓                      produces  ABCD↓
  $$                                 ↑EF↵↓
*

*3RI△$ 4CI△$$            produces  A△BCDE△↑F↵↓
*

*I(FORM)                 This command is used to separate the page in
$$                       the buffer into two pages. Both pages, however,
*                        remain in the buffer. They are not actually
                         separated until output.

*JILINE ONE↵             This example shows insertion of several lines of
LINE TWO↵                text at the beginning of the buffer.
LINE THREE↵
$$
*
```

```
*KI↲
$$
*
```

This is the command string used to delete the tail of a line without removing the carriage return-line feed at the end of the line. If the buffer contains

```
ABCD↲↓
 ↑
EFGH↲↓
```

This command will produce

```
AB↲↓
↑EFGH↲↓
```

2.6 OUTPUT COMMANDS

The command P causes (1) the entire contents of the editing buffer to be output to the output file and (2) an implicit Y command to be performed which reads in the next page of the input file. This command is used after editing of a given page is complete and the user is ready to move on to the next page.

The P command may be used with a positive numeric argument to skim over several pages. Specifically, the nP command causes the n consecutive pages of the input file, starting with the page in the editing buffer, to be output, and then the n+1st page to be yanked in.

The PW command merely outputs the page currently in the editing buffer. It does not clear the buffer, it does not read in any more text, and it does not move the buffer pointer. This command is used when creating a new file. It is also used to output the last page of a file.

If the buffer is empty, the PW and P commands have no effect.

The EF command must be used to close the output file after all output to it is complete. EF is normally used after the PW command which outputs the last page of the file.

Examples:

```
*PWEF$$
*
```

This is the command string usually used to close out a file when the last page of the file is in the buffer.

```
*PT$$
FIRST LINE
*
```

This command string outputs the current page, reads in the next page, and then types the first line of the new page.

```
*8P$$
*
```

If, for example, page 6 of a document is in the editing buffer, this command causes pages 6 through 13 of the document to be output, one after the other, and then reads in page 14.

2.7 SPECIAL EXIT COMMANDS

The EX command is used to conclude an editing job with a minimum of effort. Its use is best shown by an example.

Suppose the user is editing a 30-page file and suppose that the last actual change to the file is made on page 10. At this point the user gives the command

```
*EX$$
.
```

In this case the action performed by TECO is equivalent to the command string 20PPWEF, with an automatic return to the monitor at the end. Thus, the action of TECO is (1) to rapidly move all the rest of the input file on to the output file, (2) close the output file, and (3) to return control to the monitor.

The EG command is even more efficient. This command performs exactly the same functions as the EX command, but after that it causes re-execution of the last COMPILE, LOAD, EXECUTE, or DEBUG command attempted before TECO was called.

For example, suppose the user gives the command

```
.COMPILE PLOT.F4↵
```

to request compilation of a FORTRAN source program, but the compiler discovers errors in the code. The user would then call TECO to correct these errors:

```
.TECO PLOT.F4↵
*
```

When all the errors are edited, the user exits from TECO with the command

```
*EG$$
```

This causes the COMPILE command to be executed again on the file PLOT.F4, after TECO has finished output of the file.

Any TECO job may be aborted by using the standard return-to-monitor command: ↑C↑C (control C typed twice). However, if this command is typed before the output file is closed, the output file is lost.

If no input or output operations are in progress a single ↑C is sufficient to exit from TECO to the monitor. In such a case, the user may reenter TECO without destroying the job he was previously executing. This is illustrated in the following example.

```
.TECO    SOURCE.MAC↵ A TECO job is started.
*ICOMMENTS$$
*↑C                  The user exits to perform a few simple monitor
                     commands.
.DEASSIGN LPT↵
.DAYTIME↵
24-FEB-70    10:34
.REE↵                The user reenters TECO. The previous buffer
*                    is still intact.
```

2.8 SEARCH COMMANDS

In many cases the simplest way to position the buffer pointer is by using a character string search. A search command causes TECO to scan through the text until a specified string of characters is found, and then to position the pointer at the end of this string. There are two main search commands.

The S command is used to search for a character string within the editing buffer. The string to be searched for is specified as an alphanumerical argument following the S command. This argument must be terminated by an altmode. The character string to be searched for may contain any ASCII character except null, altmode, or rubout.

The S command may be preceded by a numerical argument $n > 1$. This argument is used to search for the nth occurrence of a character string. Thus a 2S command searches for the second occurrence of the particular character string, skipping the first occurrence. If n is omitted, $n = 1$ is assumed.

Execution of the S command begins at the position of the buffer pointer and continues to the end of the buffer. If the specified character string is not found in this range, an error message is printed and the buffer pointer is set to the beginning of the buffer.

Examples:

```
*SA→|B$$
*
```

This causes the pointer to be positioned after the B in the first occurrence of the string A - tab - B past the current position of the pointer.

```
*J2SNAME$$
*
```

This causes the pointer to be positioned after the second occurrence of the string "NAME" in the buffer.

```
*S20↵
TAG:$0LT$$
TAG:    REST OF LINE
*
```

This moves the pointer to the position just following the colon in the string "20↵↓TAG:", then repositions the pointer to the beginning of the line (just before the "TAG:") and types out the entire line starting with "TAG:".

Warning: When attempting a search it is very easy to overlook an occurrence of the search string preceding the one which the user desires. For example, he may want to move the pointer after the word "AND" but erroneously position it after a preceding occurrence of a word like "THOUSAND" . For this reason the user, especially the novice, is strongly urged to execute a T command to ascertain the position of the pointer after each search command.

Example:

```
*SWORD$0TT$$
FORMAT(1X,'WORD')
*I△WORD2$$
*
```

Here the user wishes to insert "△WORD2" after "WORD". He wisely types out the line to make sure he is at the right place, before inserting "WORD2".

The other principle search command is the N command. The difference is that an S search ends at the end of the current buffer, whereas an N search does not. An N search begins like an S search, but if the character string is not found in the current buffer, an automatic P command is executed. The current page is outputted, the next page read in, and the search continued on the new page. This process continues until either the string is found or the input file is exhausted.

If the N search does find the specified character string, the pointer is positioned at its end.

If the string is not found, an error message is generated. In this case the user caused himself a fair amount of delay. If an N search fails, the user must close the file with an EX command, then reopen it and try the N search again with a character string that can be found. The user is strongly urged to be careful when typing search character strings. Remember also that a search string must be terminated with an altmode.

Example:

```
* NSTRING - 3D $$
? 35
* EX $$
. TECO  filename.ext↵
* NSTRING $- 3D $$
*
```

Here the user meant to search for the character string "STRING", and to delete the last three characters of the string. However, he forgot to terminate the search string with an altmode and this caused the unsatisfied search request error message (?35).

CHAPTER 3

ERROR MESSAGES

When TECO encounters an illegal command or a command that for any other reason cannot be executed, a numeric error message is printed on the user's Teletype. Such messages are of the form ?nn where nn is a two-digit decimal integer that refers to the following table of error messages.

When an error message is generated, the command to which it refers is not executed, the remainder of the command string is ignored, and TECO returns to the idle state by typing an asterisk and awaiting a new command string.

The novice user is especially warned that there are a great many TECO commands that have not been described in this introductory material. Almost every letter of the alphabet and many of the special characters have meanings as TECO commands. Hence, the user should be careful when typing command strings. The beginner should probably stick to relatively short command strings.

In the following table, all TECO error messages are listed, even though some of them refer to the more advanced commands not described in this manual. Error messages referring to the advanced commands will probably be encountered by the user of this introductory material only if he has typed an unintended command letter.

The complete set of TECO commands is fully described in the TECO section of the PDP-10 Reference Handbook. Since most editing can be done using only the basic commands covered in this introductory material, most users should be able to get along without the more advanced description for some time. The novice should gain complete mastery of the basic commands before attempting to use any of the advanced commands.

Table 3-1

TECO Error Messages

nn	Meaning
1	TECO attempted to read commands beyond the terminating ⓈⓈ . This error is probably due to an unterminated @I or @S, or to an unsatisfied O command.
2	Error on output device; file closed.
3	An attempt was made to supply more than two arguments to a command, either by the use of two commas or by "H ,".
4	Too many right parentheses.
5	= command with no argument.
6	U command with no argument.
7	Q, U, X, or G command specifies an illegal Q-register (i.e., other than A through Z or 0 through 9).
8	In an X command, the second argument is not greater than the first.
9	In a G command, the Q -register does not contain text.
10	In a G command, the data in the Q -register is not in correct form (this is an internal error).
11	In an Ec command (e.g., ER, EW, EF, etc.), c is illegal.
12	File not found on LOOKUP.
13	Blank file name specified for directory device.
14	Proj-Prog number specified does not have a UFD.
15	Protection failure on DSK.
16	File cannot be accessed because it is currently being written.
17	LOOKUP or ENTER returned error type 6 (not defined).
18	LOOKUP or ENTER returned error type 7 (no device).
19	Directory full on ENTER.
20	Requested I/O device not available.
21	Not assigned.
22	EW command between an EB command and its EF.
23	EM command given when no input file is open.
24	nEM command, where n is not in the range from 1 to 16.
25	Internal error: EF and EB, but no input file open.
26	Illegal character in filename.
27	Illegal character in project-programmer number.
28	Attempt to read an input page when no file has been opened for input.

nn	Meaning
29	I/O error on input device.
30	Attempt to output a page when no file has been opened for output.
31	Two arguments supplied for an L command.
32	Attempt to move the pointer beyond the page.
33	A 2-argument command has its second argument less than the first argument.
34	Attempt to search for too long a character string.
35	Search command did not find the requested string.
36	In an M command, the Q-register does not contain text.
37	In an M command, the data in the Q-register is not in correct form (this is an internal error).
38	Unmatched right angle bracket.
39	; encountered when not in an iteration.
40	" command with no numeric argument, or "x where x is not a G, L, E, N, or C.
41	This is the number typed out at the end of the ? command's dump of the command string in error. Refer to the number of the original error.
42	A character has been encountered as a command which is not defined.
43	Control D command when DDT is not loaded with TECO.
44	Not enough core available from the monitor.
45	Rename command with a name which is blank or one which is already in use. Presumably due to a fault in the EB command.
46	Numeric argument should not be used with EX or EG commands.
47	Using EB command or TECO command with a file having the extension .BAK is illegal.
48	ER, EW, EZ, and EB commands may not be used with device TTYn, where TTYn is the user's console or any other attached user's console.
49	n <...> where n=0 is illegal.

Appendices

Index

pdp10 user's bookshelf

A Bibliography of PDP-10 Programming Documents — APRIL, 1970

To solve several customer problems, the PDP-10 Product Line has initiated a new documentation system. PDP-10 software information is now being printed in two handbooks and a series of notebooks. The handbooks and the notebooks will contain essentially the same material; the notebooks, however, will be updated more frequently with insertable pages.

Two handbooks "PDP-10 Timesharing Handbook" and PDP-10 Reference Handbook" now duplicate software manuals. These handbooks are easy to handle and store, and to assure availability, they are printed in large quantities. Revision and reprinting of the handbooks is done every six months. Customers will receive twenty copies of each handbook with the signing of the purchase order for the PDP-10 and twenty more copies with the delivery of the machine.

Each customer will also receive, free of charge, two copies of the PDP-10 software notebooks—a multiple volume set of manuals in the 8½ by 11" format. The notebooks are printed on high quality paper allowing the customer to effectively reproduce the material. Technical accuracy is maintained with quarterly update pages to the notebooks.

Since most PDP-10 software information is presently included in the handbook, it is no longer necessary to print separate manuals. Therefore, except when information is not contained in the handbooks, individual manuals can no longer be ordered. This Bookshelf serves to indicate in which handbook manuals are now located. If the manuals are not located in either handbook, the order number and price are given.

Available manuals and additional copies of the handbooks may be obtained from Digital Sales Offices or by sending a written request (with check or money order) to Program Library, Digital Equipment Corporation, Maynard.

PDP-10 Reference Handbook

This handbook is a comprehensive volume of information for experienced programmers, systems analysts, and engineers who are interested in writing and operating assembly-language programs in the PDP-10 time-sharing environment. Included in the handbook are four manuals (System Reference Manual, MACRO-10 Assembler, Time-Sharing Monitors, and DDT-10), editor programs (Editor, LINED and TECO), and utility programs (LOADER, PIP and TENDMP). A sample LOGIN procedure, a convenient summary of monitor commands, and a comprehensive index/glossary make this handbook a valuable reference for the person interested in assembly-language programming.
Order No. AIW $5.00

PDP-10 Timesharing Handbook

This is a tutorial document intended primarily for students, scientists, engineers, and financial analysts who have no experience in programming. It contains an introduction to timesharing and an explanation of the elementary and advanced monitor commands. Included are the three reference manuals BASIC, AID, and FORTRAN, as well as procedural descriptions of Batch, CHAIN, LINED, and TECO. The four demonstration programs in Book 6 enhance the tutorial aspect of the handbook.
Order No. AKW $5.00

PDP-10 System Reference Manual

An indexed programmer's handbook that describes the PDP-10 processor and the basic instruction repertoire. Following an introduction to the PDP-10's central processor structure, general word format, memory characteristics, and assembler source-programming conventions, this manual presents the specific instruction format, mnemonic and octal op codes, functions, timing formulas, and examples of each of the basic instructions. Several helpful appendices, including mnemonic op code tables, algorithms and timing charts, complete the manual. Contained in PDP-10 Reference Handbook.

Time-Sharing Monitors:

A complete guide to the use of the PDP-10's two powerful, real-time, multiprogramming, time-sharing Monitors. All Monitors schedule multiple-user time sharing of the system, allocate facilities to programs, accept input from and direct output to all system I/O devices, and relocate and protect user programs in storage. This manual details user interaction with the Monitors, from both a programming and operating viewpoint, and contains several quick-reference tables of commonly used Monitor commands and parameters, as well as examples of user coding. Contained in PDP-10 Reference Handbook.

AID (Algebraic Interpretive Dialogue)

A 'hands-on' guide to the use of AID at the Teletype. AID, a PDP-10 version of JOSS[1], is an on-line system which provides each user with a personal computing service utilizing a conversational algebraic language. This manual describes the use of the Teletype, the syntax and general rules governing the AID language, and each of the AID commands, with appropriate examples. Contained in PDP-10 Time-sharing Handbook.

Single-User Monitor System **Revision September, 1969**

A complete guide to the use of the Single-User Monitor, which performs fast job-to-job sequencing, provides I/O service for all standard devices, and is upward compatible with the Time-Sharing systems.
Order No. DEC-10-MKZA-D $2.00

Batch Processor (Batch) and Job Stacker (Stack)

An indexed manual containing all information required to prepare and run user jobs under control of the Batch Processor in either a single-user or time-sharing environment. (Batch supervises the sequential execution of a series of jobs with a minimum of operator attention, yet allows the operator to interrupt, skip, repeat, or prematurely terminate one or more of the jobs in the series at any time.) Job Stacker is used in conjunction with Batch to (1) transfer job files to the Batch input device and stack them there for subsequent input to Batch, (2) transfer Batch output job files from the Batch output device to some other device, (3) list job file directories, (4) delete job files, and (5) list directories with selective file deletion or transfer. Contained in PDP-10 Time-Sharing Handbook.

System User's Guide **Revision August, 1969**

A fact-filled operations guide designed for handy reference at the user's Teletype. Contains the basics of Teletype usage and complete operating procedures for all Commonly Used Systems Programs (CUSPs). Includes complete write-ups on DECtape Editor, BASIC, LINED, and Linking Loader. A typical chapter includes a brief description of the program, its operating environment, initialization procedures, command string formats, special switches, diagnostic messages, and indepth examples. The manual is tab-indexed for the user's convenience.
Order No. DEC-10-NGCC-D $10.00

[1]JOSS is a trademark and service mark of the RAND Corporation for its computer program and services using that program.

COBOL LANGUAGE **August, 1969**

A reference manual designed to aid the user in writing COBOL programs for the PDP-10. Each COBOL language element is accorded a detailed treatment that explains and demonstrates its use in a variety of programming contexts. The four major divisions of a COBOL program and their conventional formats are clearly described and effectively illustrated. Other subjects given extended coverage in this manual are the COBOL library, COBOL reserved words, and the CALL procedure. Each chapter contains numerous examples of the efficient use of the components of a COBOL program. Indexed.

Order No. DEC-10-KC1A-D $6.00

FORTRAN IV

This manual describes statements and features of FORTRAN IV on the PDP-10. Includes descriptions of library functions, calling library subroutines from the Science Library, and the FORTRAN IV operating System. An appendix contains language differences for those using the small (5.5K) PDP-10 FORTRAN Compiler. Contained in PDP-10 Time-Sharing Handbook.

Science Library and Fortran Utility Subprograms
Revision March, 1969

A general reference manual covering Science Library arithmetic function and utilicy subprograms and FORTRAN IV nonmathematical utility subprograms. A functional description followed by the calling sequence, list of external subprograms called, entry points, and subprogram length, is given for each subprogram. In addition, the type of argument(s) and result, a description of the algorithm used, and a discussion of the accuracy of the algorithm are given for each function. Appendices contain information on error analyses, double-precision format and input conversion, a bibliography, and average run times.

Order No. DEC-10-SFLE-D $4.00

TECO (Text Editor and Corrector)
Minor Revision, August, 1969

This programmer's reference manual describes the powerful context editor for the PDP-10. Editing is done on a character, line or variable character string basis. Describes more than 30 commands for inserting, deleting, appending, searching for, and displaying text. Contained in PDP-10 Reference Handbook.

BASIC

A valuable guide to the BASIC® commands needed for a efficient expression of scientific, business, and educational problems. The manual contains complete tutorial explanations of these additional features: (1) matrix computations; (2) alphanumeric information handling; (3) program control and storage facilities; (4) program editing capabilities; (5) formatting of Teletype output; and (6) documentation and debugging aids. Contained in PDP-10 Time-Sharing Handbook.

® Registered: Trustees of Dartmouth College

PIP (Peripheral Interchange Program)

Explains how PIP is used to transfer data files between standard peripheral devices. Shows how command strings are written, describes switches available for optional functions, techniques for handling file directories, error messages and other features. Contained in PDP-10 Reference Handbook.

MACRO-10 Assembler

The programmer's reference manual for the PDP-10 assembly system. Explains format of statements, use of pseudo-operations, and coding of macro instructions which make MACRO-10 one of the most powerful assemblers available. Contained in PDP-10 Reference Handbook.

PDP-10 Reference Card **Revision November, 1969**

A handy pocket-sized guide to instruction mnemonics, hardware and software (Monitor system) word formats, and instruction codes.

Order No. DEC-10-J00B-D $0.25

DDT-10 (Dynamic Debugging Technique)

This reference manual describes the dynamic debugging program used for on-line checkout and testing of MACRO-10 and FORTRAN programs. The commands of DDT are grouped so that they can be used easily and effectively by both the uninitiated user and the experienced programmer. Included in the appendices is an informative summary of all DDT functions. Contained in PDP-10 Reference Handbook.

The following supplementary documents are also available from the Program Library.

PDP-10 DECtape Copy Program (COPY)	DEC-10-RPTA-D	1.00
FORTRAN IV Software Maintenance Memos	DEC-10-KF1A-D	1.00
Linking Loader V.27	DEC-10-LLZA-D	1.00
FORTRAN IV Utility Subprograms (RELEAS, MAGDEN, BUFFER, IFILE, and OFILE)	DEC-10-FIYB-D	1.00
S68OI-DC68A Data Line Scanner for PDP-10	DEC-10-FWVA-D	1.00
Program Logic Manual for the PDP-10 Time-Sharing Monitors	DEC-10-MRZA-D-(L)	4.00

APPENDIX B

PDP-10 SOFTWARE

Table B-1 lists PDP-10 system programs and the documentation pertaining to them. For a description of each of the documents, refer to the Bookshelf in Appendix A of this handbook.

Table B-1
PDP-10 Software

Software	Documentation	Document Order No.
AID	PDP-10 Timesharing Handbook (Book 4) PDP-10 Software Notebook System User's Guide	AKW DEC-10-SYZA-D DEC-10-NGCC-D
BASIC	PDP-10 Timesharing Handbook (Book 3) PDP-10 Software Notebook	AKW DEC-10-SYZA-D
Batch	PDP-10 Timesharing Hardbook (Book 8) PDP-10 Software Notebook	AKW DEC-10-SYZA-D
BINCOM	PDP-10 Reference Handbook (Book 6) PDP-10 Software Notebook System User's Guide	AIW DEC-10-SYZA-D DEC-10-NGCC-D
CHAIN	PDP-10 Timesharing Handbook (Book 8) PDP-10 Software Notebook	AKW DEC-10-SYZA-D
CHKPNT	PDP-10 Software Notebook (System Manager's Guide)	DEC-10-SYZA-D
COBOL	COBOL Language PDP-10 Software Notebook	DEC-10-KC1A-D DEC-10-SYZA-D

Table B-1 (Cont.)

Software	Documentation	Document Order No.
COMPIL	PDP-10 Reference Handbook (Book 2) PDP-10 Software Notebook	AIW DEC-10-SYZA-D
COPY	PDP-10 DECtape Copy Program (COPY) PDP-10 Software Notebook	DEC-10-RPTA-D DEC-10-SYZA-D
CREF	PDP-10 Reference Handbook (Book 5) PDP-10 Software Notebook	AIW DEC-10-SYZA-D
DDT	PDP-10 Reference Handbook (Book 5) PDP-10 Software Notebook System User's Guide	AIW DEC-10-SYZA-D DEC-10-NGCC-D
DRIVER	PDP-10 Timesharing Handbook (Book 8) PDP-10 Software Notebook	AKW DEC-10-SYZA-D
DSKLST	PDP-10 Software Notebook (System Manager's Guide)	DEC-10-SYZA-D
EDITOR	PDP-10 Reference Handbook (Book 4) PDP-10 Software Notebook	AIW DEC-10-SYZA-D
FAILSAFE	PDP-10 Software Notebook (System Manager's Guide)	DEC-10-SYZA-D
FILDDT	PDP-10 Software Notebook (System Manager's Guide)	DEC-10-SYZA-D
FORTRAN	PDP-10 Timesharing Handbook (Book 5) PDP-10 Software Notebook System User's Guide	AKW DEC-10-SYZA-D DEC-10-NGCC-D
FUDGE2	PDP-10 Reference Handbook (Book 6) PDP-10 Software Notebook	AIW DEC-10-SYZA-D

Table B-I (Cont.)

Software	Documentation	Document Order No.
GLOB	PDP-10 Reference Handbook (Book 6) PDP-10 Software Notebook	AIW DEC-10-SYZA-D
LINED	PDP-10 Reference Handbook (Book 4) PDP-10 Timesharing Handbook (Book 8) PDP-10 Software Notebook	AIW AKW DEC-10-SYZA-D
LOADER	PDP-10 Reference Handbook (Book 3) PDP-10 Software Notebook	AIW DEC-10-SYZA-D
LOGIN	PDP-10 Reference Handbook (Book 3) PDP-10 Software Notebook (System Manager's Guide)	AIW DEC-10-SYZA-D
LOGOUT	PDP-10 Reference Handbook (Book 3) PDP-10 Software Notebook (System Manager's Guide)	AIW DEC-10-SYZA-D
MACRO	PDP-10 Reference Handbook (Book 2) PDP-10 Software Notebook System User's Guide	AIW DEC-10-SYZA-D DEC-10-NGCC-D
MONEY	PDP-10 Software Notebook (System Manager's Guide)	DEC-10-SYZA-D
MONGEN	PDP-10 Software Notebook (System Manager's Guide)	DEC-10-SYZA-D
MONITOR	PDP-10 Reference Handbook (Book 3) PDP-10 Software Notebook	AIW DEC-10-SYZA-D
PIP	PDP-10 Reference Handbook (Book 3) PDP-10 Software Notebook	AIW DEC-10-SYZA-D

Table B-1 (Cont.)

Software	Documentation	Document Order No.
PIP1	PDP-10 Reference Handbook (Book 3) PDP-10 Software Notebook	AIW DEC-10-SYZA-D
PRINT	PDP-10 Software Notebook (System Manager's Guide)	DEC-10-SYZA-D
PRINTR	PDP-10 Software Notebook (System Manager's Guide)	DEC-10-SYZA-D
REACT	PDP-10 Software Notebook (System Manager's Guide)	DEC-10-SYZA-D
SRCCOM	PDP-10 Reference Handbook (Book 6) PDP-10 Software Notebook System User's Guide	AIW DEC-10-SYZA-D DEC-10-NGCC-D
STACK	PDP-10 Software Notebook	DEC-10-SYZA-D
SYSTAT	PDP-10 Reference Handbook (Book 3) PDP-10 Software Notebook	AIW DEC-10-SYZA-D
TECO	PDP-10 Reference Handbook (Book 4) PDP-10 Timesharing Handbook (Book 8) System User's Guide PDP-10 Software Notebook	AIW AKW DEC-10-NGCC-D DEC-10-SYZA-D
TENDMP	PDP-10 Reference Handbook (Book 6) PDP-10 Software Notebook	AIW DEC-10-SYZA-D

INDEX

Page numbers refer to the book and the page number within the book; i.e., 3-42 is the 42nd page of Book 3.

A command, TECO, 8-75
A format, 5-44
ABS function, 3-14, 3-73
Absolute value, 3-14, 3-73, 4-10, 4-23
ACCEPT statement, 5-58, 5-77
Access time, I-10
Access to BASIC, 3-29, 3-32
Accumulator conventions, 5-111
Accuracy of AID, 4-11
Addition, 4-10, 4-23
Adjustable dimensions, 5-63
Allocating system resources
  ASSIGN monitor command, 7-4
  CORE monitor command, 7-8
  DEASSIGN monitor command, 7-6
  FINISH monitor command, 7-7
  REASSIGN monitor command, 7-7
Alphanumeric fields, 5-44
ALPHI. routine, 5-84
ALPHO. routine, 5-84
ALTMODE key, 2-20
Argument, FORTRAN, definition, 5-24
Arithmetic error conditions, FORTRAN, 5-121
Arithmetic Formulas, AID, 4-19, 4-78
Arithmetic function definition statement 5-71, 5-79
Arithmetic operations, BASIC, 3-13
Arithmetic operations on complex numbers, FORTRAN, 5-21
Arithmetic statement, FORTRAN, 5-29
Array dimensioning, 5-22, 5-62
Arrays, 4-21, 5-22
ASCII character set, 5-125
ASCII mode
  DECtape, 5-128
  disk, 5-128
  magnetic tape, 5-131
$Assign Card, Batch control card, 8-20
  PROTECT, 8-20, 8-21
ASSIGN monitor command, 7-4
ASSIGN statement, 5-32, 5-77
ASSIGN SYS monitor command, 7-15
Assigned GO TO statement, 5-32, 5-77
Asterisk convention, 2-11
Asynchronous design, 1-6
ATN function, 3-14, 3-73
ATTACH monitor command, 7-8, 7-16
Augmented matrix, 6-11

Background job, 7-8
BACKSPACE statement, 5-39, 5-59, 5-77
Batch
  diagnostic messages, 8-28
  equipment requirements, 8-11
  examples, 8-33, 8-39
  introduction, 8-7
  processing under PDP-10 monitors, 8-7
  processor, 8-7
Batch control cards
  #Tape, 8-21
  $*Card, 8-27
  $**Card, 8-28
  $ASSIGN Card, 8-20
  $BIN, 8-24
  $CBL, 8-21
  $CREF, 8-23
  $DUMP, 8-27
  $EOF, 8-26
  $EOJ, 8-26
  $EXCDR, 8-23
  $F4, 8-21
  $GET, 8-25
  $JOB, 8-16
  KEY, 8-18
  $LDR, 8-23
  $MAC, 8-21

$PAUSE, 8-27
$RUN, 8-25
$SAVE, 8-24
$SSAVE, 8-24
$START, 8-26
Batch processing, 1-3
$BIN Card, Batch control card, 8-24
Binary mode
DECtape, 5-128
disk, 5-128
magnetic tape, 5-131
BINWR. routine, 5-85
Blank common, 5-64
Blank fields, 5-49
Blank records, 5-47
BLOCK DATA statement, 5-68, 5-76
BLOCK DATA subprogram, 5-76
Block identifier, 5-64
Block name, 5-64
Block of words, 1-10
Boolean expressions, 4-11, 4-20, 4-31
Brackets, AID, 4-10, 4-23
BTEMP, 8-11, 8-12, 8-19
Buffer pointer positioning, TECO, 8-76
Buffer sizes of peripheral devices, 5-127
BUFFER subroutine, 5-90

C TECO command, 8-77
CALL statement, 5-75, 5-77
CANCEL command, 4-37, 4-80
$*Card, Batch control card, 8-27
$**Card, Batch control card, 8-28
Card notation, Batch,
# , 8-16
braces, 8-16
lower case letters, 8-16
parameters, 8-16
parentheses, 8-16
Cards acceptable as Batch input
card description, 8-16
card notation, 8-16
Carriage control, FORTRAN, 5-45, 5-48
$CBL Card, Batch control card, 8-21.
CCONT monitor command, 7-15
CHAIN calling sequence, 8-50
Chain files, 5-90, 8-47
CHAIN job implementation, 8-47
CHAIN program description, 8-54
CHAIN programming considerations, 8-53
CHAIN subroutine, 5-90, 8-47
CHANGE instruction, 3-61, 3-63
Character set, AID, 4-85
Character set, ASCII, 5-125
Character set, FORTRAN, 5-17

Clock, 1-5
Closed subroutines, 5-71
COBOL compilation, 8-21
Coding form, FORTRAN, 5-16
Command
direct, definition, 4-10
indirect, definition, 4-10
Command summary, AID, 4-80
Comment line, FORTRAN, 5-16
COMMON area, CHAIN, 8-47
Common block, FORTRAN, 5-64
COMMON statement, 5-64, 5-66, 5-78
Common storage, FORTRAN, 5-64
COMPILE monitor command, 2-13
Complex constants, FORTRAN, 5-20
Complex fields, 5-46
Complex subexpression, 5-25
COMPLEX, type declaration
statement, 5-68, 5-78
Compound expressions, FORTRAN
logical, 5-28
numeric, 5-24
Compute bound, 1-6
Computed GO TO statement, 5-31, 5-77
Conditional expressions, 4-32, 4-78
Conditional GO TO statement
see IF-THEN
Constants, FORTRAN
complex, 5-20
double precision, 5-20
integer, 5-19
literal, 5-21
logical, 5-21
octal, 5-20
real, 5-19
Context switching, 1-5
CONTINUE monitor command, 3-32, 4-10, 7-13
CONTINUE statement, 5-36, 5-77
Control commands, BASIC, 3-65
Control statements, FORTRAN, 5-31, 5-77
CALL, 5-75
CONTINUE, 5-36
DO, 5-34
END, 5-37
GO TO, 5-31
IF, 5-32
PAUSE, 5-36
RETURN, 5-75
STOP, 5-37
CONTROL-C, 2-18, 3-31, 3-66, 4-9
CONTROL-F, 2-20, 4-11, 4-12
Core image, 2-3, 2-13
CORE monitor command, 7-8

Correcting typing errors
  AID, 4-13
  BASIC, 3-30, 3-31, 3-34
  Monitor, 2-19
  TECO, 8-73
Cramer's method, 6-13
CREATE monitor command, 2-7, 8-59
Creating files
  CREATE monitor command, 2-7
  MAKE monitor command, 2-9
Creation of CHAIN files, 8-50
$CREF card, Batch control card, 8-23
  cross reference assembly listing, 8-23
CREF monitor command, 7-10
Cross-reference listing, 7-10, 8-23
COS function, 3-14, 3-73
Cosine, 3-14, 3-73
COT function, 3-14, 3-73
Cotangent, 3-14, 3-73
CSTART monitor command, 7-15
CTEST monitor command, 7-16

D format, 5-40, 5-44
D LINED command, 8-61
D monitor command, 7-15
D TECO command, 8-79
Data block, 3-16, 3-61
Data file capability, BASIC, 3-67, 3-73
Data line multiplexor, 1-8
Data line scanner, 1-8
Data record, 5-53
Data set, 1-8
DATA statement, BASIC, 3-11, 3-12, 3-13, 3-16, 3-71
DATA statement, FORTRAN, 5-66, 5-78
Data specification statements, 5-61
  DATA, 5-66, 5-78
  BLOCK DATA, 5-68, 5-76
Data specification subprogram, 5-68
Data transmission statements, 5-39, 5-77
  ACCEPT, 5-58
  DECODE, 5-59
  ENCODE, 5-59
  PRINT, 5-53
  PUNCH, 5-54
  READ, 5-56
  REREAD, 5-57
  TYPE, 5-54
  WRITE, 5-54
DATA. UUO, 5-85
DATE subroutine, 5-90
DAYTIME monitor command, 2-16
DDT monitor command, 7-14
DDT submode, 8-25
DEASSIGN monitor command, 7-6
DEBUG monitor command, 2-14
Debugging programs
  DEBUG monitor command, 2-14
  in Basic, 3-34
DECODE statement, 5-59, 5-77
DECtape usage, 5-128
DEC. UUO, 5-86
Dedicated system, 1-4
DEF function, 3-40, 3-74
Defensive software, 1-12
DEFINE FILE statement, 5-55
Defined function, 3-41, 3-74
Defined locations, 5-111
DELETE, AID command, 4-39, 4-80
DELETE, BASIC command, 3-65
DELETE, monitor command, 2-12
Deletion commands, TECO, 8-79
DEMAND command, 4-18, 4-41, 4-80
Demonstration programs
  curve-fitting, 6-8
  random-number generator, 6-36
  remaindering, 6-26
  sum of squares, 6-4
DET function, 3-55, 3-74
DETACH, monitor command, 7-8, 7-15
Device assignments, 5-128
Device control statements, 5-58, 5-77
  BACK SPACE, 5-59
  END FILE, 5-59
  REWIND, 5-59
  SKIP RECORD, 5-59
  UNLOAD, 5-59
Device table, FORTRAN, 5-129
DEVTB. , 5-129
Diagnostic messages
  AID, 4-87
  BASIC, 3-75
  BATCH, 8-28
  FORTRAN, 5-116
  monitor, 7-16
  LINED, 8-63
  TECO, 8-87
Diagnostic software, 1-12
DIM statement, BASIC, 3-26, 3-28, 3-52, 3-74
DIMENSION statement, FORTRAN, 5-62, 5-78
  adjustable dimension, 5-63
Dimensioning, BASIC, 3-26, 3-28, 3-52, 3-54
Direct command, definition, 4-10
Direct steps, 4-15

Directory format
long format, 2-11
short format, 7-10
DIRECTORY, monitor command,2-11,7-10
DIRT. routine, 5-84
DISCARD command, 4-45, 4-80, 4-84
Disk usage, 5-128
Division, 4-10, 4-23
DO command, 4-18, 4-34, 4-46, 4-80
DO statement, 5-34, 5-77
DONE command, 4-51, 4-80
DOUBLE PRECISION, type declaration statement, 5-68, 5-78
Double precision constants, FORTRAN, 5-20
Double word, FORTRAN, 5-25, 5-27
DOUBT. routine, 5-84
Driver
equipment requirements, 8-30
general description, 8-29
implementation, 8-30
use of, 8-30
Dual memory protection and relocation, 1-8
Dummy arguments, AID, 4-20
Dummy arguments, FORTRAN, 5-72, 5-73
Dummy identifiers, 5-71, 5-72
DUMP, 8-27, 8-29, 8-31
DUMP subroutine, 5-91
$DUMP card, Batch control card, 8-27

E format, 5-40, 5-44
E, LINED command, 8-61
E, monitor command, 7-15
Edit commands, BASIC, 3-65
EDIT, monitor command, 4-9, 8-59
Editing files
EDIT monitor command, 2-9
TECO monitor command, 2-10
EF, TECO command, 8-82
EG, TECO command, 8-83
ENC. UUO, 5-86
ENCODE statement, 5-59
END OF FILE statement, 5-59
END statement, BASIC, 3-11, 3-19, 3-72
END statement, FORTRAN, 5-37
Entering a BASIC program, 3-30, 3-32
$EOF card, Batch control card, 8-26
EOF1 subroutine, 5-91
EOFC subroutine, 5-91
$EOJ card, Batch control card, 8-26
Equipment requirements for BATCH, 8-11
EQUIVALENCE statement, 5-65, 5-66

Errors, BASIC
grammatical, 3-34
logical, 3-34
ERRSET subroutine, 5-91
EX, TECO command, 8-83
Examples, BATCH, 8-33, 8-39
Examples, LINED, 8-61
EXECUTE, monitor command, 2-14
Executing a BASIC program, 3-31, 3-33, 3-35
Executing programs
EXECUTE monitor command, 2-14
Executive mode, 1-9
Executive program, 1-5
Exit commands, TECO, 8-83
EXIT subroutine, 5-91
$EXLDR card, Batch control card, 8-23
map, 8-23
EXP function, 3-14, 3-73
Exponentiation, 4-10, 4-23
Expression, definition, 4-10, 4-78
Expressions, FORTRAN
logical, 5-26
numeric, 5-24
EXTERNAL statement, 5-76
External storage, definition, 4-78
External subprograms, 5-71

F format, 5-40—5-44
$F4 card, Batch control card, 8-21
Field delimiters, 5-43
Field specifications, 5-40
Field width, 5-40,—5-44
FILE command, AID, 4-53, 4-81, 4-84
File, definition, 4-78
File protection, 1-11
Filename extension, 2-6
Filenames, 2-6
Files, 2-6
FILES command, Basic, 3-67, 3-73
Filing system, 1-10
FIN. UUO, 5-85
FINISH, monitor command, 7-7
FLIRT. routine, 5-84
FLOUT. routine, 5-84
FNEND statement, 3-40
FOR statement, Basic, 3-22, 3-23, 3-71
FORM command, 4-54, 4-81
Form, definition, 4-78
FORMAT statement, 5-39
alphanumeric fields, 5-44
blank fields, 5-49
complex fields, 5-46
logical fields, 5-43

mixed fields, 5-45
multiple records, 5-46
numeric fields, 5-40
variable field width, 5-43
Formats stored as data, 5-47
Formula, 4-19, 4-78
FORSE. , 5-83
FORMAT processing, 5-83
I/O device control, 5-84
UUO dispatching, 5-84
FORTRAN compilation, 8-21
FORTRAN compiler diagnostics
command errors, 5-116
compilation errors, 5-117
FORTRAN compiler switches, 5-115
FORTRAN library functions, 5-86
FORTRAN library subprograms, 5-71, 5-86, 5-90
FORTRAN library subroutines, 5-90
FORTRAN operating system, 5-83
FORSE. , 5-83
I/O conversion routines, 5-84
FORTRAN UUO's, 5-85
FORTRAN program and MACRO subprogram linkage, example of, 5-97
FORTRAN UUO's, 5-85
Function, definition, 5-24
Function identifier, 5-24, 5-72
FUNCTION statement, 5-72
FUNCTION subprograms, 5-72
FUNCTION statement, 5-72
Function subprogram linkage, example of, 5-96
Function type, 5-24, 5-73
Function value, 5-24
Functions, AID
argument, 4-26
cosine, 4-26
digit part, 4-26
exponent part 4-28
exponential, 4-26
first, 4-26, 4-34
fraction part, 4-26
integer part, 4-26
logarithm, natural, 4-27
maximum, 4-27, 4-34
minimum, 4-27, 4-34
product, 4-27, 4-34
signum, 4-27
sine, 4-27
square root, 4-27
sum, 4-27, 4-34
truth value, 4-28, 4-32
Functions, Basic, 3-14, 3-27, 3-54, 3-55, 3-73

G format, 5-40—5-44
$ GET card, Batch control card, 8-25
GET monitor command, 7-12
Getting information from the monitor
DAYTIME monitor command, 2-16
PJOB monitor command, 2-16
RESOURCES monitor command, 7-13
SYSTAT monitor command, 7-13
TIME monitor command, 2-16
Getting on the system, 2-4
Glossary of AID terms, 4-78
GO command, 4-57, 4-81
GO TO statement, Basic, 3-11, 3-18, 3-71
GO TO statement, FORTRAN
assigned, 5-32, 5-77
computed, 5-31, 5-77
unconditional, 5-31, 5-77
GOSUB statement, 3-41, 3-72

H-conversion, 5-45
HALT monitor command, 1-10
Hierarchy
of logical operators, 5-28
of numeric operators, 5-26, 5-28
of relational operators, 5-28
Hilbert matrix, 3-56

I format, 5-40—5-44
I LINED command, 8-60
I TECO command , 8-80
IBUFF call, 5-90
Identifiers, 4-10, 4-17, 4-78
indexed, 4-21
Identity matrix, 3-52, 3-72
IF clause, 4-59, 4-81
IF END statement, Basic, 3-70, 3-73
IF statement, FORTRAN
logical, 5-33, 5-77
numerical, 5-33, 5-77
IF - THEN statement, Basic, 3-10, 3-18, 3-71
IFILE subroutine, 5-91
ILL subroutine, 5-02
Immediate storage, definition, 4-78
IMPLICIT statement, 5-68, 5-78
Impure code, 1-7
IN. UUO, 5-85
Indexed identifiers, 4-21
Indirect command, definition, 4-10
Indirect steps, 4-15

INF. UUO, 5-85
Input commands, TECO, 8-75
Input file, TECO, 8-69
INPUT statement, Basic, 3-48, 3-72
  data file, 3-69, 3-73
Input/output control, 1-9
Input/output processor, 1-6
Insertion command, TECO, 8-80
Instruction set, PDP-10, 5-132
INT function, 3-37, 3-73
Integer constants, 5-19
Integer function, 3-37, 3-73
INTEGER, type declaration statement, 5-68, 5-79
Intelligent terminal, 1-14
Interaction, 1-3
Internal subprograms, 5-71
Interrupting execution of a Basic program, 3-31, 3-35
INTI. routine, 5-84
INTO. routine, 5-84
I/O bound, 1-6
I/O conversion routines, 5-84
I/O list, 5-52
I/O records, 5-53
Item-list, definition, 4-78
Iterative clauses, 4-34

J TECO command, 8-76
$JOB card ,Batch control card, 8-16
  DUMP, 8-18
  NOGO, 8-18

K TECO command, 8-79
KEY card, Batch control card, 8-18
KJOB monitor command, 4-17, 4-10

L format, 5-43
L TECO command, 8-77
Latency optimization, 1-10
$LDR card , Batch control card, 8-23
  LIB, 8-23
  Loader, 8-23
Least squares, 6-9
Leaving the monitor
  KJOB monitor command, 2-17, 3-32, 3-33
LEGAL subroutine, 5-92
LENGTH command, 3-65
LET command, AID, 4-19, 4-60, 4-81
LET command, Basic, 3-10, 3-16, 3-71
LIB40, 5-83
LINE command, 4-62, 4-82
Line continuation field, 5-15
Line format, 5-15
Line numbers, Basic, 3-10, 3-12, 3-31
Line spacing, 5-48
LINED
  auxiliary commands, 8-62
  command level, 8-60
  commands, 8-59
  conventions and restrictions, 8-62
  error, 8-63
  examples, 8-61, 6-62
  implementation, 8-64
  monitor commands, 8-59
  text mode, 8-60
LINT. routine, 5-84
LIST, Basic command, 3-65
LIST, monitor command, 7-9
Lists, Basic, 3-25
Literal constants, FORTRAN, 5-21
LOAD, monitor command, 2-14
Loading CHAIN, 8-49
Loading programs
  DEBUG, monitor command, 2-14
  EXECUTE, monitor command, 2-14
  LOAD, monitor command, 2-14
Locations
  defined, 5-111
  required, 5-111
LOG, Basic function, 3-14, 3-73
LOGICAL, type declaration statement, 5-68, 5-79
Logical constants, 5-21
Logical device names, 7-3
Logical expressions, 5-26
Logical fields, 5-43
Logical IF statement, 5-33, 5-77
Logical operators, 5-27, 5-28
LOGIN, monitor command, 2-4
Login procedure, 2-4
Loops, 3-21
  nested, 3-23
LOUT. routine, 5-84

$MAC card, 8-21
MACRO main programs, 5-103
MACRO subprograms, 5-95
  example, 6-26
MAGDEN subroutine, 5-92
Magnetic tape usage, 5-130

Magnitude
of double precision constants, 5-20
of integer constants, 5-19
of real constants, 5-19
Main memory, 1-5
MAKE, monitor command, 2-7, 8-70
Manipulating core images
GET, monitor command, 7-12
R, monitor command, 7-12
RUN, monitor command, 7-11
SAVE, monitor command, 7-11
Manipulating files
DELETE, monitor command, 2-12
DIRECTORY, monitor command, 2-11
RENAME, monitor command, 2-12
TYPE, monitor command, 2-11
Manipulating Teletype assignment
ATTACH, monitor command, 7-8
DETACH, monitor command, 7-8
TALK, monitor command, 7-8
Master File Directory, 1-10
Mathematical operators,
absolute value, 4-10, 4-23
addition, 4-10, 4-23
brackets, 4-10, 4-23
division, 4-10, 4-23
exponentiation, 4-10, 4-23
multiplication, 4-10, 4-23
parentheses, 4-10, 4-23
subtraction, 4-10, 4-23
Matrices, 3-51
Memory blocks, 1-6
Memory protection and relocation,
Batch, 8-9
Memory protection register, 1-5
MFD, 1-10
Mixed fields, 5-45
Modem, 1-8
Modularity, 1-12
Monitor, 1-5
Monitor calls, 1-10
Monitor mode, 1-9
Monitor overhead, 1-7
MTOP. UUO, 5-85
Multiple record formats, 5-46
termination of, 5-47
Multiplication, 4-10, 4-23
Multiprocessor configurations, 1-14
Multiprogramming, 1-5

N, TECO command, 8-85
NAMELIST statement, 5-39, 5-49, 5-79
input data, 5-50
output data, 5-51
Natural logarithm functions, 3-14, 3-73
n-dimensional arrays, simulation of, 3-57
NEW Basic command, 3-29, 3-65
NEXT Basic command, 3-22, 3-23, 3-71
NLI. UUO, 5-85
NLO. UUO, 5-86
NMLST. routine, 5-85
Nonexecutable statements
FORMAT statement, 5-39
NAMELIST statement, 5-49
Normal exit of a DO statement, 5-34
NUM Basic function, 3-54, 3-74
Numbers, Basic, 3-15
Numeric expressions, 5-24
Numeric fields, 5-40
repetition of, 5-46
repetition of groups, 5-46
Numeric IF statement, 5-33, 5-77
Numeric operations, 5-26
Numeric operators, 5-24

O format, 5-40—5-44
OBUFF call, 5-90
Octal constants, 5-20
OCTI. routine, 5-85
OCTO. routine, 5-85
OFILE subroutine, 5-92
OLD Basic command, 3-29, 3-65
ON-GO TO, Basic, 3-19, 3-72
On-line, 1-9
Open subroutines, 5-71
Operating system, 1-5
Operating system diagnostics,
FORTRAN, 5-120
Operators
logical, 5-27
numeric, 5-24
priorities of, 5-28
relational, 5-27
Order of precedence, 4-10, 4-23
OUT. UUO, 5-85
OUTF. UUO, 5-85
Output commands, TECO, 8-82
Output file, TECO, 8-69
Overlapped I/O, 1-10
Overlay, 1-8

P LINED command, 8-61
P TECO command, 8-82
PAGE command, 4-63, 4-82
Page, TECO, 8-69
Parentheses, 4-10, 4-23

Parts, 4-15, 4-78
$ PAUSE card, Batch control card, 8-27
PAUSE statement, FORTRAN, 5-36, 5-77
PDUMP subroutine, 5-92
Peripheral devices, 7-3
Permanent device assignment
ASSIGN monitor command, 7-4
Permanent resident area (CHAIN), 8-47
Physical device names, 7-3
PJOB monitor command, 2-16
Precision
of double-precision constants, 5-20
of real constants, 5-19
PRINT statement, Basic, 3-11, 3-17, 3-45, 3-71
PRINT statement, FORTRAN, 5-53, 5-77
Printable characters, 3-62
Priorities of operators, 5-28
Priority interrupt system, 1-5
Private device, 1-10
Program names, 3-30
Programmed operators, 1-10
Programming phases, 2-3
Propositions, 4-20, 4-31
Pseudo-Teletype (PTY), 8-9
Public device, 1-10
PUNCH statement, 5-54, 5-78
Pure code, 1-7
PW, TECO command, 8-82

Queue, 1-6
QUIT command, 4-64, 4-82

R command, TECO, 8-77
R monitor command, 7-12
R AID, 4-9
R LINED, 8-62
Random access of records
READ, 5-57, 5-78
WRITE, 5-55, 5-78
Random number generator, example, 6-36
Randon numbers, 3-38, 3-73
RANDOMIZE statement, 3-39, 3-73
Range of a DO statement, 5-34
READ statement, Basic, 3-10, 3-11, 3-16, 3-71
READ statement, FORTRAN, 5-56, 5-78
Real constants, 5-19
REAL, type declaration statement, 5-68, 5-79
REASSIGN , monitor command, 7-7
RECALL command, 4-65, 4-82, 4-84
Record, 1-10
REENTER, monitor command, 4-10, 7-15
Re-entrant software, 1-7
Relational operators, 5-27
Relational symbols, 3-15
RELEAS subroutine, 5-92
Reliability, 1-12
Relocatable binary program, 2-3, 2-13
REM statement, 3-49, 3-72
Remembered arguments, 2-10
Removable resident area (CHAIN), 8-48
Removable storage device, 1-11
RENAME , Basic command, 3-65
RENAME, monitor command, 2-12
Repetition
of field specifications, 5-46
of groups, 5-46
REPLACE command, 3-65
Replacement operator, 5-29
Requesting line printer output
CREF, monitor command, 7-10
DIRECTORY, monitor command, 7-10
LIST, monitor command, 7-9
Required locations, 5-111
REREAD statement, 5-57, 5-78
RERED. UUO, 5-85
RESEQUENCE command, 3-66
RESET TIMER command, 4-66, 4-82
RESET. UUO, 5-85
RESOURCES, monitor command, 7-13
RESTORE command, 3-49, 3-61, 3-72
data file, 3-69, 3-73
RETURN key, 2-19
RETURN statement, Basic, 3-41, 3-72
RETURN statement, FORTRAN, 5-75,5-77
REWIND statement, 5-59, 5-78
RND function, 3-38, 3-72
Round robin operation, 1-4
RTB. UUO, 5-85
RUBOUT key, 2-19, 3-30, 3-35
Rules of form, AID, 4-11
RUN, Basic command, 3-13, 3-31, 3-65
$ RUN card, Batch control card, 8-25
RUN, monitor command, 7-11
RUNNH command, 3-31, 3-65

S, LINED command, 8-62
S, TECO command, 8-84
$SAVE card, Batch control card, 8-24
SAVE, monitor command, 3-36,3-66,7-11
SAVRAN subroutine, 5-92
Scalar variables, 5-22

Scale factor, 5-19, 5-20, 5-42
SCHEDULE, monitor command, 7-15
Scheduling algorithm, 1-6
SCRATCH command, 3-66
  data file, 3-68, 3-73
Search commands, TECO, 8-84
Secondary memory, 1-5
Segment read-in area (CHAIN), 8-48
SET command, 4-17, 4-67, 4-82
SETRAN subroutine, 5-92
SGN function, 3-40, 3-74
Shared system device, 8-11
Sign function, 3-40, 3-74
SIN function, 3-14, 3-73
Sine, 3-14, 3-73
SIZE argument, 4-78
SKIP RECORD statement, 5-59,5-78
SLIST. UUO, 5-85
SLITE subroutine, 5-93
Slow peripherals, 1-11
Source program definition, 2-3
Spacing, 5-48
Specification statements, 5-61, 5-78
  data specification, 5-66
  storage specification, 5-62
  type declaration, 5-68
Spooling, 1-12
SQR function, 3-14, 3-73
$ SSAVE card, Batch control card, 8-24
SSAVE, monitor command, 7-15
SSWTCH subroutine, 5-93
Standard processor, 2-13
$ START card, Batch control card, 8-26
START, monitor command, 7-12
Starting a core image program
  CONTINUE, monitor command, 7-13
  HALT, monitor command, 7-13
  START, monitor command, 7-12
Statement field, 5-16
Statement number field, 5-15
Statement numbers, 5-15
Statements, 3-10
STEP instruction, 3-22, 3-24
Steps, 4-15, 4-79
STOP command, AID, 4-68, 4-82
STOP command, Basic, 3-49, 3-72
STOP statement, FORTRAN, 5-37, 5-77
Storage specification statements, 5-62
  COMMON, 5-64
  DIMENSION, 5-62
  EQUIVALENCE, 5-65
Stored formats, 5-47
Strings, Basic, 3-59
Subprogram calling sequences, 5-95
Subprogram linkage, example, 5-96, 5-104
SUBROUTINE statement, 5-74
Subroutine subprograms, 5-74
  CALL statement, 5-75
  RETURN statement, 5-75
  SUBROUTINE statement, 5-74
Subroutines, 3-41
  nested, 3-42
SUBSCRIPT INTEGER, type
  declaration statement, 5-68, 5-79
Subscripts, 3-25, 3-28
Subtraction, 4-10, 4-23
Swapping, 1-5
Symbiont operation, 1-12
Symbolic logic, 5-26
SYS, 8-11, 8-12
SYSTAT, monitor command, 7-13
  example, 7-14
SYSTEM command, 3-66
System device, 8-11, 8-12

T format, 5-48
T, TECO command, 8-78
Tab, horizontal, 3-47, 5-15
Tables, Basic, 3-25
TALK, monitor command, 7-8
#TAPE, Batch control card, 8-21
TAN function, 3-14, 3-73
TECO command arguments, 8-74
TECO command string syntax, 8-72
TECO error messages, 8-87, 8-88
TECO general operating procedure, 8-69
TECO initialization, 8-70
TECO, monitor command, 2-10, 8-70
TECO special symbols, 8-71
TECO text type-out, 8-78
Teletype consoles, 4-12
Teletype control characters, 2-18
Teletype special keys
  ALTMODE key, 2-20
  RETURN key, 2-19
  RUBOUT key, 2-19
Temporary device assignment, 7-4
Termination of a program, 5-37
TIME, AID command, 4-79
TIME, monitor command, 2-16
Time quantum, 1-4
Time slice, 1-4
TIME subroutine, 5-93
TIMER argument, 4-79
Timesharing, 1-4
TO command, 4-69, 4-82

Translating programs
COMPILE, monitor command, 2-13
Trapping, 1-9
TYPE, AID command, 4-71, 4-83
Type declaration statements, 5-68
TYPE,monitor command, 2-11
TYPE statement, FORTRAN, 5-54, 5-78

UFD, 1-10
Unconditional GO TO statement, 5-31, 5-77
Unit records, 5-40
UNLOAD statement, 5-59, 5-78
USE command, 4-75, 4-83, 4-84
User control levels, Batch
console commands, 8-13
control cards, 8-13
User-defined functions, 4-20 , 4-29
User directories, 2-11
User mode, 1-9
User's File Directory, 1-10

Variable field width, 5-43
Variables, AID, 4-10, 4-17, 4-79
Variables, Basic
alphanumeric string, 3-59
numeric, 3-15
subscripted, 3-25
Variables, FORTRAN
array, 5-22
scalar, 5-22
Vectors, 3-51
Verbs, AID
cancel, 4-37, 4-80
delete, 4-39, 4-80
demand, 4-18, 4-41, 4-80
discard, 4-45, 4-80, 4-84
do, 4-18, 4-34, 4-46, 4-80
done, 4-51, 4-80
file, 4-53, 4-81, 4-84
form, 4-54, 4-81
go, 4-57, 4-81
if clause, 4-59, 4-81
let, 4-19, 4-60, 4-81
line, 4-62, 4-82
page, 4063, 4-82
quit, 4-64, 4-82
recall, 4-65, 4-82, 4-84
reset timer, 4-66, 4-82
set, 4-17, 4-67, 4-82
stop, 4-68, 4-82
to, 4-69, 4-82
type, 4-71, 4-83
use, 4-75, 4-83, 4-84

WEAVE, 3-66
Word format, PDP-10, 5-126
WRITE statement, Basic, 3-68, 3-73
WRITE statement, FORTRAN, 5-54, 5-78
WTB. UUO, 5-85

X format, 5 -49

Y, TECO command, 8-75

DIGITAL EQUIPMENT CORPORATION WORLD-WIDE SALES AND SERVICE

MAIN OFFICE AND PLANT

146 Main Street, Maynard, Massachusetts, U S A. 01754 • Telephone From Metropolitan Boston 646-8600 • Elsewhere· (617)-897-5111 • TWX. 710-347-0212 Cable· DIGITAL MAYN Telex 94-8457

UNITED STATES

NORTHEAST

REGIONAL OFFICE·
15 Lunda Street, Waltham, Massachusetts 02154
Telephone· (617)-891-1030 TWX.710-324-0919

WALTHAM
15 Lunda Street, Waltham, Massachusetts 02154
Telephone (617)-891-6310/6315 TWX. 710-324-0919

CAMBRIDGE/BOSTON
899 Main Street, Cambridge, Massachusetts 02139
Telephone: (617)-491-6130 TWX: 710-320-1167

ROCHESTER
130 Allens Creek Road, Rochester, New York 14618
Telephone (716)-461-1700 TWX 710-599-3211

CONNECTICUT
1 Prestige Drive, Meriden, Connecticut 06450
Telephone: (203)-237-8441 TWX: 710-461-0054

MID-ATLANTIC—SOUTHEAST

REGIONAL OFFICE:
U.S. Route 1, Princeton, New Jersey 08540
Telephone (609)-452-9150 TWX. 510-685-2338

NEW YORK
95 Cedar Lane, Englewood, New Jersey 07631
Telephone. (201)-871-4984, (212)-594-6955, (212)-736-0447
TWX· 710-991-9721

NEW JERSEY
1259 Route 46, Parsippany, New Jersey 07054
Telephone: (201)-335-3300 TWX: 710-987-8319

PRINCETON
Route One and Emmons Drive,
Princeton, New Jersey 08540
Telephone: (609)-452-2940 TWX: 510-685-2337

LONG ISLAND
1919 Middle Country Road
Centereach, L I., New York 11720
Telephone: (516)-585-5410 TWX: 510-228-6505

PHILADELPHIA
1100 West Valley Road, Wayne, Pennsylvania 19087
Telephone (215)-687-1405 TWX: 510-668-4461

WASHINGTON
Executive Building
7100 Baltimore Ave., College Park, Maryland 20740
Telephone: (301)-779-1100 TWX: 710-826-9662

MID-ATLANTIC—SOUTHEAST (cont.)

DURHAM/CHAPEL HILL
2704 Chapel Hill Boulevard
Durham, North Carolina 27707
Telephone (919)-489-3347 TWX 510-927-0912

HUNTSVILLE
Suite 41 — Holiday Office Center
3322 Memorial Parkway S W , Huntsville, Ala. 35801
Telephone (205)-881-7730 TWX: 810-726-2122

ORLANDO
Suite 232, 6990 Lake Ellenor Drive, Orlando, Fla 32809
Telephone (305)-851-4450 TWX 810-850-0180

ATLANTA
Suite 116, 1700 Commerce Drive, N.W.,
Atlanta, Georgia 30318
Telephone: (404)-351-2822 TWX: 810-751-3251

KNOXVILLE
5731 Lyons View Pike, S W , Knoxville, Tenn. 37919
Telephone. (615)-588-6571 TWX· 810-583-0123

CENTRAL

REGIONAL OFFICE.
1850 Frontage Road, Northbrook, Illinois 60062
Telephone (312)-498-2560 TWX 910-686-0655

PITTSBURGH
400 Penn Center Boulevard,
Pittsburgh, Pennsylvania 15235
Telephone· (412)-243-8500 TWX· 710-797-3657

CHICAGO
1850 Frontage Road, Northbrook, Illinois 60062
Telephone (312)-498-2500 TWX 910-686-0655

ANN ARBOR
230 Huron View Boulevard, Ann Arbor, Michigan 48103
Telephone (313)-761-1150 TWX 810-223-6053

INDIANAPOLIS
21 Beachway Drive — Suite G
Indianapolis, Indiana 46224
Telephone (317)-243-8341 TWX 810-341-3436

MINNEAPOLIS
15016 Minnetonka Industrial Road
Minnetonka, Minnesota 55343
Telephone: (612)-935-1744 TWX: 910-576-2818

CLEVELAND
Park Hill Bldg , 35104 Euclid Ave.
Willoughby, Ohio 44094
Telephone: (216)-946-8484 TWX: 810-427-2608

CENTRAL (cont.)

ST. LOUIS
Suite 110, 115 Progress Pky., Maryland Heights,
Missouri 63043
Telephone: (314)-872-7520 TWX: 910-764-0831

DAYTON
3101 Kettering Blvd., Dayton, Ohio 45439
Telephone: (513)-299-7377 TWX: 810-459-1676

DALLAS
8855 North Stemmons Freeway, Suite 204
Dallas, Texas 75247
Telephone (214)-638-4880 TWX· 910-861-4000

HOUSTON
3417 Milam Street, Suite A, Houston, Texas 77002
Telephone: (713)-524-2961 TWX: 910-881-1651

WEST

REGIONAL OFFICE·
560 San Antonio Road, Palo Alto, California 94306
Telephone: (415)-328-0400 TWX: 910-373-1266

ANAHEIM
801 E. Ball Road, Anaheim, California 92805
Telephone: (714)-776-6932 or (213)-625-7669
TWX: 910-591-1189

WEST LOS ANGELES
2002 Cotner Avenue, Los Angeles, California 90025
Telephone. (213)-479-3791 TWX 910-342-6999

SAN FRANCISCO
560 San Antonio Road, Palo Alto, California 94306
Telephone: (415)-326-5640 TWX: 910-373-1266

ALBUQUERQUE
6303 Indian School Road, N.E.
Albuquerque, N M. 87110
Telephone: (505)-296-5411 TWX· 910-989-0614

DENVER
2305 South Colorado Blvd., Suite #5
Denver, Colorado 80222
Telephone· 303-757-3332 TWX· 910-931-2650

SEATTLE
1521 130th N.E , Bellevue, Washington 98004
Telephone (206)-454-4058 TWX 910-443-2306

SALT LAKE CITY
431 South 3rd East, Salt Lake City, Utah 84111
Telephone. (801)-328-9838 TWX 910-925-5834

INTERNATIONAL

CANADA

Digital Equipment of Canada, Ltd.
CANADIAN HEADQUARTERS
150 Rosamond Street, Carleton Place, Ontario
Telephone: (613)-257-2615 TWX: 610-561-1651

OTTAWA
120 Holland Street, Ottawa 3, Ontario
Telephone: (613)-725-2193 TWX 610-562-8907

TORONTO
230 Lakeshore Road East, Port Credit, Ontario
Telephone: (416)-278-6111 TWX: 610-492-4306

MONTREAL
9675 Cote de Liesse Road
Dorval, Quebec, Canada 760
Telephone 514-636-9393 TWX· 610-422-4124

EDMONTON
5531-103 Street
Edmonton, Alberta, Canada
Telephone: (403)-434-9333 TWX. 610-831-2248

EUROPEAN HEADQUARTERS

Digital Equipment Corporation International-Europe
81 Route De L'Aire
1227 Carouge / Geneva, Switzerland
Telephone: 42 79 50 Telex. 22 683

GERMANY

Digital Equipment GmbH
COLOGNE
5 Koeln, Bismarckstrasse 7, West Germany
Telephone 52 21 81 Telex 888-2269
Telegram: Flip Chip Koeln

MUNICH
8000 Muenchen 19, Leonrodstrasse 58
Telephone 516 30 54 Telex 524226

ENGLAND

Digital Equipment Co. Ltd.
READING
Arkwright Road, Reading, Berkshire, England
Telephone: Reading 85131 Telex. 84327

MANCHESTER
6 Upper Precinct, Worsley
Manchester, England m28 5az
Telephone. 061-790-4591/2 Telex: 668666

LONDON
Bilton House, Uxbridge Road, Ealing, London W 5
Telephone. 01-579-2781 Telex 22371

FRANCE

Equipement Digital S A R.L.
PARIS
233 Rue de Charenton, Paris 12, France
Telephone 344-76-07 Telex 21339

BENELUX

Digital Equipment N.V.
(serving Belgium, Luxembourg, and The Netherlands)
THE HAGUE
Koninginnegracht 65, The Hague, Netherlands
Telephone: 635960 Telex: 32533

SWEDEN

Digital Equipment Aktiebolag
STOCKHOLM
Vretenvagen 2, S-171 54 Solna, Sweden
Telephone 08 98 13 90 Telex 170 50
Cable· Digital Stockholm

SWITZERLAND

Digital Equipment Corporation S A.
GENEVA
81 Route De L'Aire
1227 Carouge / Geneva, Switzerland
Telephone. 42 79 50 Telex: 22 683

ITALY

Digital Equipment S p A
MILAN
Corso Garibaldi, 49, 20121 Milano, Italy
Telephone 872 748, 872 694, 872 394 Telex 33615

AUSTRALIA

Digital Equipment Australia Pty. Ltd.
SYDNEY
75 Alexander Street, Crows Nest, N.S.W. 2065. Australia
Telephone 439-2566 Telex 20740
Cable: Digital, Sydney

MELBOURNE
60 Park Street, South Melbourne, Victoria, 3205
Telephone 69-6142 Telex 30700

WESTERN AUSTRALIA
643 Murray Street
West Perth, Western Australia 6005
Telephone 21-4993 Telex 92140

BRISBANE
139 Merivale Street, South Brisbane
Queensland, Australia 4101
Telephone 44047 Telex 40616

JAPAN

TOKYO
Rikei Trading Co., Ltd (sales only)
Kozato-Kaikan Bldg.
No. 18-14, Nishishimbashi 1-chome
Minato-Ku,Tokyo, Japan
Telephone: 5915246 Telex: 7814208

Digital Equipment Corporation International
(engineering and services)
Fukuyoshicho Building, No 2-6, Roppongi 2-Chome,
Minato-Ku, Tokyo
Telephone 585-3624 Telex No . 0242-2650

TO THE READER OF THE PDP-10 TIMESHARING HANDBOOK

We at Digital want to improve the quality and usefulness of our publications. However, we cannot achieve this goal by ourselves. We need your collaboration: your corrections, your observations, your critical evaluations. Will you please provide us with such constructive information by filling out this questionnaire and mailing it back to us?

1. (a) Is the handbook a useful document? .... ☐ YES ☐ NO
   (b) If you answer is YES, tell us what features make it useful.

____

____

____

____

   (c) If your answer is NO, tell us what features prevent it from being a useful document.

FOLD

____

____

____

____

2. (a) Is the text clear and readily understandable? .... ☐ YES ☐ NO
   (b) If your answer is NO, cite the paragraphs, chapters, or sections that are unclear or difficult to understand.

____

____

____

____

3. (a) Are you pleased with the organization of the handbook? .... ☐ YES ☐ NO
   (b) Should the organization be changed? .... ☐ YES ☐ NO
   (c) What changes do you suggest?

FOLD

____

____

____

____

4. (a) Should the organization of individual chapters or books be changed? ...... ☐ YES ☐ NO
   (b) If your answer is YES, give us your suggestions.

____

____

____

____

5. (a) Should more demonstration programs be added when the handbook is revised? ☐ YES ☐ NO

(b) What kind of programs would you like to see added?

6. (a) Should more utility programs be added when the handbook is revised? ..... ☐ YES ☐ NO
(b) Give us your suggestions.

– FOLD

7. (a) Should anything be deleted from the handbook when it is revised? .... ☐ YES ☐ NO
(b) If your answer is YES, give us your suggestions.

8. List any further suggestions you have for the improvement of this handbook.

- FOLD

FIRST CLASS
PERMIT NO. 33
MAYNARD, MASS.

BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY
IF MAILED IN THE UNITED STATES

Postage will be paid by:

PDP-10 Software Writing Group
Programming Department
Digital Equipment Corporation
Maynard, Massachusetts 01754

STAPLE

SAMPLE LOGIN PROCEDURE

Make contact with your computer facility by whatever means the facility has estabilshed (e.g., acoustic coupler, telephone, or data phone).

↓

Turn the little plastic knob on the right-hand side of the Teletype to ON.

↓

Type C on the Teletype (i.e., hold down the CTRL key while striking C). This establishes communication with the Time-Sharing Monitor. The Monitor signifies its readiness to accept commands by responding with a period (.).

↓

Type LOGIN, or LOG, followed by a carriage return. The system will respond with an informative message like the following:

```
        JOB n   NAME OF SYSTEM
        #
```

JOB n is the job number the system has just assigned to you. NAME OF SYSTEM is usually the Monitor name and version number.

↓

Type your project-programmer numbers after the number sign, followed by a carriage return.

↓

The time-sharing system will then type

```
            PASSWORD:
```

↓

Type your secret password followed by a carriage return. The system will keep the password secret by not printing it on the paper.

↓

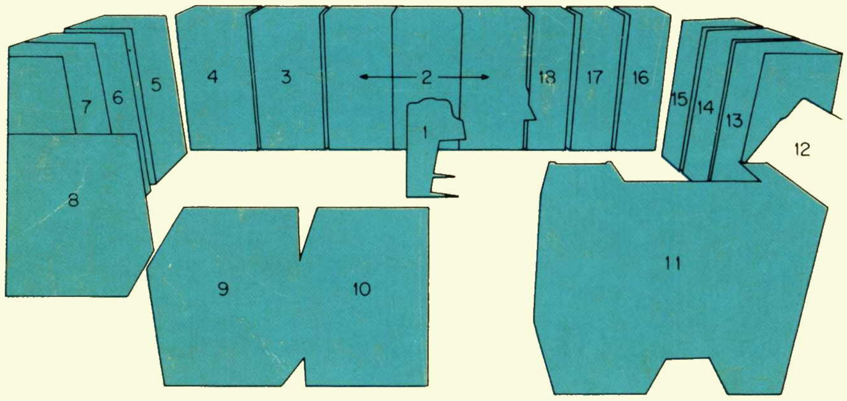
If the project-programmer numbers and the password match the project-programmer numbers and password stored in the system accounting file, the system responds with the time, date, TTY number, C, and a period.

Example:

```
        1301 8-May 70 TTY23
```

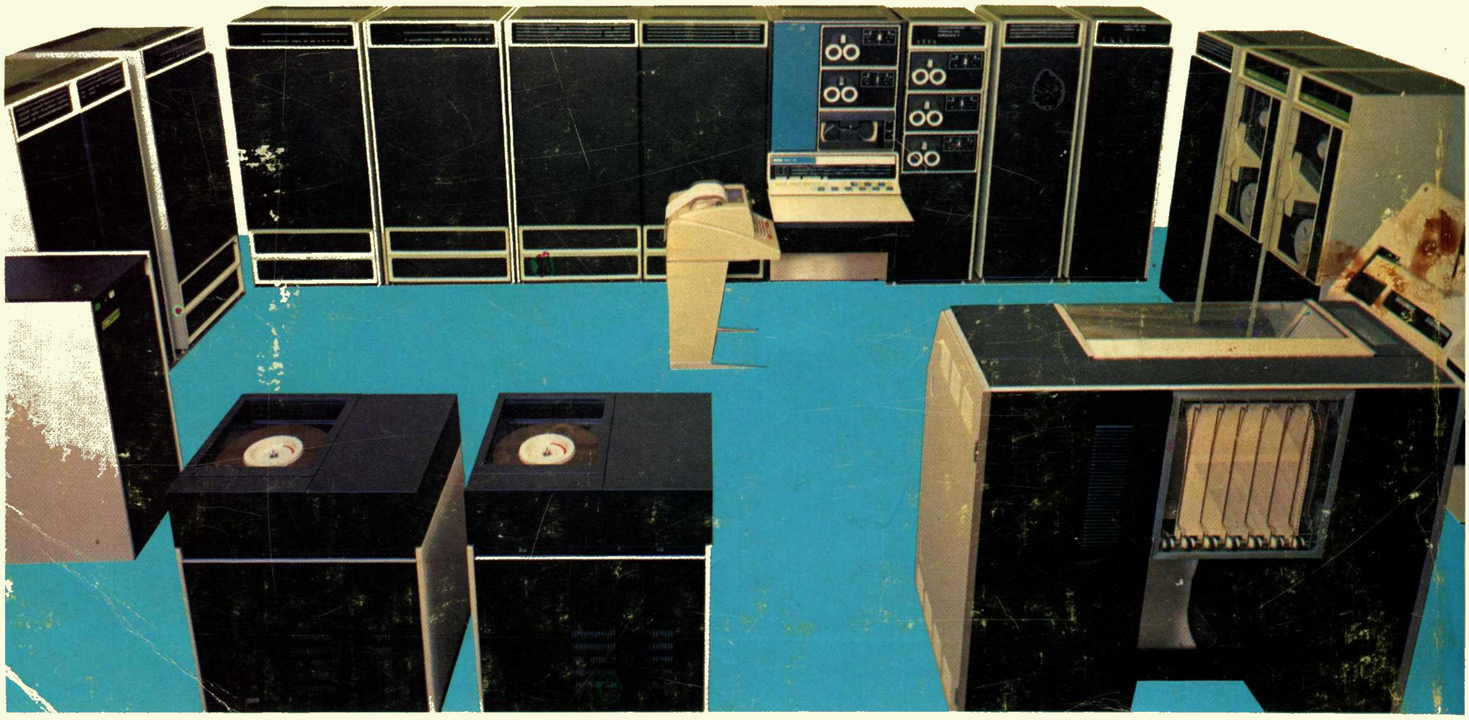
↓

Now the time-shraing system is ready to accept any commands you wish to type in. You may direct it to load and start a program from the System Library (.R prog), start a program already loaded in core (.START), or perform any of a variety of other operations. (See inside of back cover for a summary of Time-Sharing Monitor commands.)



1. Console Teletype	10. Disk Pack Unit*
2. Central Processor	11. Line Printer
3. 16K, 1.0 μsec Memory	12. Card Reader
4. 16K, 1.0 μsec Memory	13. Magnetic Tape Transport
5. 16K, 1.0 μsec Memory	14. Magnetic Tape Transport
6. Data Channel	15. Magnetic Tape Control
7. Swapping Disk Control	16. Communications System
8. Swapping Disk	17. Line Printer/Card Reader Control
9. Disk Pack Unit*	18. DECtape Control and 3 DECtape Units

*Disk Pack Control Not Shown



102X • 00470 • AKW
0-09-50

PRINTED IN U.S.A.