

TSS/8
USER'S GUIDE

8

THE UNIVERSITY OF WISCONSIN-MILWAUKEE
COMPUTER AND MANAGEMENT SERVICES DIVISION

THE UNIVERSITY OF WISCONSIN-MILWAUKEE COMPUTER AND MANAGEMENT SERVICES DIVISION

DOCUMENTATION REGISTRATION FORM

(Completion of this form will place the purchaser's name on a mailing list to receive updates of the purchased document)

| NAME (1 | olease print) | DATE | |
|---------|----------------------------------|--|----|
| I HAVE | PURCHASED A COPY | OF(Title of Documentation) | |
| PLEASE | SEND UPDATES AND | REVISIONS OF THIS DOCUMENT TO: | |
| | | Campus Address* | |
| | | Off-campus Address | |
| * | are asked to spenew material wil | ing expenses, purchasers with a campus address ecify this as their preferred address to which it be sent. Purchasers with no campus address hal sent to the off-campus address listed. | |
| CAMPUS | ADDRESS: | Dept. | |
| | | Room & Bldg. | 3 |
| | | Telephone | |
| | | | |
| OFF-CAL | MPUS ADDRESS: | Street | |
| | | City, State, Zip | 1 |
| | | Telephone | |
| -plea | se, fold, staple | , and mail this form to address shown. Thank yo | ou |

TO: The University of Wisconsin-Milwaukee Computer & Management Services Division, Retail Sales EMS EB68 Milwaukee, Wisconsin 53201

TSS/8 USERS GUIDE

Prepared by the TSS/8

Programming Group at

the University of Wisconsin-Milwaukee

Editor:

L. P. Levine

May, 1973

Please return, filled in, the card enclosed with this manual. Updates and revision to this manual will be sent automatically to any purchaser whose card is on record.

This handbook contains material taken with permission from the TSS/8 Timesharing System Users Guide, copyright 1972, by the Digital Equipment Corporation, Maynard, Massachusetts.

First printing May, 1973.

TABLE OF CONTENTS

| Cont | cents | | Page |
|------|--------------------------------|-----------|------|
| INTE | RODUCTION | CHAPTER 1 | |
| 1.1 | General Description | • | 1-1 |
| 1.2 | User Programs | | 1-1 |
| 1.3 | User Files | | 1-2 |
| 1.4 | TSS/8 User Console | | 1-2 |
| ELEM | ENTARY MONITOR COMMANDS | CHAPTER 2 | |
| 2.1 | Calling Monitor | | 2-1 |
| 2.2 | Logging in on TSS/8 | | 2-2 |
| 2.3 | Logging out of TSS/8 | | 2-4 |
| 2.4 | Rubout | | 2-4 |
| 2.5 | System Library Program Control | | 2-5 |
| 2.6 | Communication with Other Users | | 2-5 |
| 2.7 | System Status Report | | 2-6 |
| 2.8 | Resource Sharing | | 2-6 |
| 2.9 | Error Messages | | 2-9 |
| SYST | EM PROGRAM GROUPS | CHAPTER 3 | |
| 3.1 | FOCARL | | 3-1 |
| 3.2 | BASIC | | 3-2 |
| 3.3 | FORTRAN | | 3-3 |
| 3.4 | ALGOL | | 3-4 |
| 3.5 | PALD | | 3-4 |

| Cont | tents | | | Page |
|------|--|--|--|--|
| FUNI | OAMENTA | L UTILITY PROGRAMS | CHAPTER 4 | |
| 4.1 | EDIT 4.1.1 4.1.2 | Summary of Symbolic Editor O EDIT Command Summary | perations | 4-1 4-2 4-3 |
| 4.2 | CAT 4.2.1 4.2.2 | Example of CAT Usage S Option of CAT | | 4-4 4-4 4-5 |
| 4.3 | SYSTA | T Example of SYSTAT Usage | | 4-5 4-5 |
| 4.4 | 4.4.2 4.4.3 4.4.4 4.4.5 4.4.6 4.4.7 4.4.8 4.4.9 | PIP Conventions Using PIP to Load Paper Tape Using PIP to Punch Out a Dis Using PIP With High Speed Re Using PIP to Transfer BIN Fo Moving Disk Files Deleting Disk Files Transferring BASIC Files Transferring SAVE Format Fil Summary of PIP Options | k File ader and Punch rmat Files | 4-6 4-6 4-7 4-7 4-8 4-8 4-9 4-9 |
| 4.5 | 4.5.2 4.5.3 4.5.4 4.5.5 4.5.6 4.5.7 4.5.8 | Deleting Files Moving Files Renaming a File Control C Exit From COPY Summary of COPY Options UWM TSS/8 COPY Conventions | | 4-10 4-11 4-12 4-12 4-13 4-13 4-13 4-14 4-14 |
| 4.6 | DUMP | - | | 4-16 |
| 4.7 | LABLD | P | | 4-17 |
| 4.8 | LINK | | | 4-17 |
| 4.9 | LOGOU | r | | 4-19 |
| 4.10 |) LOGII | N | | 4-20 |
| 4.11 | CPAS | g | | 4-21 |

1.1 GENERAL DESCRIPTION

TSS/8 (Time-Sharing System for the PDP-8/I, -8/e and -8 Computers) is a general-purpose, time-sharing system offering up to 16 users (24 in certain applications) a comprehensive library of System Programs. These programs provide facilities for editing, assembling, compiling, debugging, loading, saving, calling, and executing user programs on-line. They include FORTRAN, ALGOL and LISP. Two conversational, interactive systems, FOCARL and BASIC versions are also included. Also available to the user is an Assembler in which all of the features of the hardware are accessible. All of these program packages are briefly described in Chapter 3.

By separating the central processing operations from time-consuming interactions with human users, the computer can, in effect, work on a number of programs simultaneously. Cycling between programs and giving only a fraction of a second at a time to each program or task, the computer can deal with many users seemingly at once. The appearance is created that each user has the computer to himself. The execution of various programs is done without their interfering with each other and without lengthy delays in the response to individual users.

The heart of TSS/8 is a complex of subprograms called the Monitor. Monitor coordinates the operations of the various programs and user consoles, ensuring that the user is in contact at all times with his program. Monitor allocates the time and services of the computer to the various users; it grants a slice of processing (computing) time to each job, and schedules jobs in sequential order to make most efficient use of the system device (disk). Monitor handles user requests for hardware operations (reader, punch, etc.), swaps (moves) programs between memory and disk, and manages the user's private files. The Elementary Monitor Commands are described in Chapter 2.

1.2 USER PROGRAMS

When the user is working on a program with TSS/8, his work exists in the computer as though he had his own 4K (4096 word) PDP-8. Several users can run programs at virtually the same time, because TSS/8 Monitor controls the scheduling of execution times. Monitor brings a program into core from the disk, allows it to execute for a short time, and takes note of the state at which execution is stopped. Monitor then brings the next user program into core, and repeats the process. The user is allotted a 4K block of storage that contains his particular

program; this 4K block will be swapped from core onto a 4K area of disk storage when it is necessary for Monitor to bring in another program to run.

After the user's program has been executed, for a period of time it is placed at the end of the queue (line) of user programs waiting to run. If only one program is ready to run, it is allowed to do so without interruption until another program is ready.

If a user wishes to maintain a permanent copy of his program, it is necessary to save a copy within the file area of the disk (an area separate from the swapping area). Later sections of this manual describe the procedures to create and update such files.

1.3 USER FILES

A TSS/8 user is any person logged in on TSS/8. Each user has an account number and password assigned to him by the installation manager or the person responsible for his particular TSS/8; the account number and password allows the user access to the computer. His account number is also used to identify whatever files the user may own within the TSS/8 file system.

The disk (a large external memory device used for storage of programs and data) is divided into logical areas called files. A user can create files and store them in the file storage area of disk. The user can also specify which groups of users may access his files and for what purpose (read, or read and write).

Parts of the disk are used to store System Files; those programs which are accessible to anyone using the computer. A major portion of this manual, Chapter 4, deals with how to use System Files, generally referred to as System Library Programs.

With the appropriate Monitor commands, the user can create new files and manipulate old files (extend, reduce or delete them). These commands are explained in the Advanced User Guide. Most individual System Library Programs are able to handle user files as input or output with commands issued at the user's console.

1.4 TSS/8 USER CONSOLE

The user console is a model 33 teletype or equivalent. Any device capable of generating asynchronous ASCII (American Standard for Computer Information Interchange) code at either 10 or 20 characters per second full duplex may be interfaced with TSS/8. Unless the console is directly wired to the Tss/8, it must be connected to the computer via a dial-up connection utilizing a MODEM (Modulator-Demodulator) similar to a Bell 103 type device operated in the originate mode. If there is question as to whether a given console can communicate with TSS/8, the TSS/8 System Manager can help you. A copy of the ASCII Code is shown in Appendix A.

CHAPTER 2

ELEMENTARY MONITOR COMMANDS

TSS/8 offers the user a variety of hardware and software resources. The TSS/8 Monitor controls the allocation and use of these resources. Many of the functions of the Monitor are invisible, and of no concern to the user, for example, the way it allows many users to run programs on a single computer. In other instances the user explicitly tells Monitor what he would like to do and the resources he wishes to utilize. He does so by typing one or more of the commands described in this chapter.

The Monitor commands described in the first half of this chapter are those the user needs to log into the system, to utilize the TSS/8 System Library Programs, and to log out of the system. All TSS/8 users must be familiar with these commands. The commands described in sections 2.5, 2.6, and 2.7 are not needed to run TSS/8 System Library Programs such as BASIC and FOCAL, but are frequently useful. The Monitor commands described in Chapter 8 are primarily useful for creating assembly language programs and files.

NOTE

All Monitor commands must be terminated by typing the RETURN key. All words within a Monitor command line are separated by one or more spaces.

2.1 CALLING MONITOR

The user enters commands to System Programs, such as BASIC and FOCAL, in exactly the same way that he enters commands to Monitor (i.e., by typing them in at the keyboard); therefore, the system must have some way of distinguishing between the two cases. It does so by defining two modes of console operation: Monitor mode and program mode. When a user's console is in Monitor mode, all input is interpreted as being commands to Monitor. Otherwise, all input is assumed to be to the user program.

A special character, CTRL/B, (obtained by striking B with the CTRL key held down; and echoed on the Teletype as †B) is used to unconditionally place the user's console in Monitor mode. Typing CTRL/B tells the system that the command to follow is to be interpreted as a command to Monitor, regardless of the mode that the Teletype is in. Generally, the command which follows the CTRL/B will be the S command.

†BS

†B†BS

Return to Monitor mode.

Return to Monitor mode from a program which is typing out. (The two CTRL/B's stop the typeout, allowing the S command to be typed.)

It is not necessary to precede each Monitor command with CTRL/B. Once in Monitor mode, a console stays in that mode until a command is typed which starts a user program. To signify that it is in Monitor mode, the system types a dot (.) on the left margin of the console printer paper. This dot indicates that the characters typed in next will be treated as a Monitor command. Thus, the CTRL/B capability is important when a user is running a program and wishes to type a Monitor command. He may, for example, be using one language (or System Program) and want to change to another, as shown below.

· R FOCARL

FOCAPL, VERSION 10 FUNCTIONS? NOVE

*

т

* 1PS

.R PASIC

NEW OR OLF --

Monitor always responds to †BS by typing a dot at the left-hand margin.

2.2 LOGGING IN ON TSS/8

To prevent unauthorized usage and to allow Monitor to maintain a record of system usage, TSS/8 requires that each user identify himself to the system before using it. Before attempting to log in, the user should ensure that the console LINE/OFF/LOCAL knob is turned to the LINE position (see section 1.4.1) before striking the RETURN key. If the console is connected to TSS/8 and is not already in use, Monitor rolls the console paper up two lines and prints a dot at the left margin of the paper.

The dot indicates that TSS/8 is in Monitor mode and that Monitor is waiting for the user to issue a command.

.LCGIN

Request access to TSS/8.

The LOGIN command allows the user to access the TSS/8 system.

Be sure your console is in full duplex mode. The user types LOGIN followed by a valid account number and password. Providing the console is free (not already logged in), the command, account number, and password will not be printed on the console paper as the keys are typed. If the command name letters are being printed, stop typing the command; instead, strike the RETURN key, log out using the LOGOUT command (see section 2.3), at this point a successful LOGIN can be accomplished. The LOGIN command is formatted as shown below:

·LOGIN 1234 ABCD

(only the dot is printed)

where . is printed by Monitor, LOGIN is the command name, 1234 represents the account number, and ABCD represents the password.

NOTE

A command name and each parameter (except the last) is always followed by a space, and the command line is always terminated with the RETURN key.

When a user types something other than a valid LOGIN command on a console, Monitor responds in one of the following ways.

| Response | Explanation |
|---------------------------------------|--|
| · HELLO? | (user typed HELLO) |
| LCCIN PLEASE? | (user typed ASSIGN D 3) |
| · ILLEGAL REQUEST | (user typed LOGIN ABCD ABCD) |
| •LOGIN 4771 DEMO ALREADY LOCGED IN | (user typed valid LOGIN on an already logged in console) |
| UNAUTHORIZED ACCOUNT | (user typed an incorrect account number or password) |

In the first example, HELLO is not a command, so it is repeated with a question mark by Monitor. In the second example, ASSIGN D 3 is a valid command but it is not appropriate until after the user logs in; therefore, Monitor asks the user to log in. In the third example, Monitor finds that the LOGIN command is improperly formatted (the first parameter must be from one to four numbers). The console printout tells the user that he has made an ILLEGAL REQUEST. When the console is already logged in and the user types the LOGIN command, the characters typed echo at the console, and Monitor informs the user that the console is occupied with the message ALREADY LOGGED IN? If the user attempts to use an incorrect account number or password, Monitor

replies UNAUTHORIZED ACCOUNT. Thus, Monitor can distinguish an invalid command from a valid command; it can also distinguish whether the valid command is appropriate when issued, whether the command is properly formatted, and whether the account number and password are acceptable. In all the examples above, Monitor ignores the command and prints another dot.

When Monitor finds the LOGIN command properly formatted and the account number and passwords acceptable, it responds by identifying the version of the system being used, the job number it has assigned to the user, the number of the console being used, and the time-of-day in hours, minutes, and seconds. For example:

TSS/8.19 JOB 03 K01 08:45:21

AND USUALLY THE SYSTEM MANAGER WILL ENTER HERE A COMMENT OR NOTE TO THE USER CONCERNING THE SYSTEM

Monitor then prints another dot and waits for the user to issue the next command. The job number assigned is an internal number by which the system identifies each on-line user.

2.3 LOGGING OUT OF TSS/8

The LOGIN command tells Monitor that the user is ready to begin an on-line session. The LOGOUT command indicates that the user is finished and ready to leave his Teletype.

.LCCCUT

Disconnect the user from the system and record the amount of time he has used.

Monitor responds to LOGOUT by typing the amount of computer time used in the session and the total real time of the session. For example:

.LOCCUT RUN TIME 00:00:34 ELAPSED TIME 00:35:41 PLEASE TURN OFF YOUR TIY

Computer time used in this example was thirty four seconds, while the elapsed time between LOGIN and LOGOUT was thirty five minutes and forty one seconds.

2.4 RUBOUT

The RUBOUT key (marked DELETE on some keyboards) may be used to erase single characters from a monitor command. Thus, if a user wishes to logout and inadvertently types LOGI before noticing his error, he may type the RUBOUT key to rubout the letter "I". Monitor will echo the rubbed out character. If a user types LOGI (RUBOUT) OUT, the keyboard will show: LOGIIOUT where the first I is the error and the second I is the echo of the rubbed out character.

2.5 SYSTEM LIBRARY PROGRAM CONTROL

Once logged in, the user can call any of the TSS/8 Library Programs described in Chapters 3 through 7. To call such a program, the user types the command R (meaning run) followed by one or more spaces and the program name.

For example:

.R BASIC NEW OR OLD --

Monitor fetches the BASIC language processor from the System Library and starts it. BASIC begins its dialogue by asking if the user wishes to work on a new program or retrieve an old one from disk storage. Notice that once BASIC begins, the console is no longer in Monitor mode. Dots are no longer printed at the margin. All input is considered to be commands to BASIC.

If the user types a program name which cannot be found in the System Library, Monitor responds with an error message and returns the console to Monitor mode.

•R BASICK FILE NOT FOUND?

The exact contents of a TSS/8 System Library may vary from installation to installation.

2.6 COMMUNICATION WITH OTHER USERS

Although TSS/8 gives each system user the impression that he is the only user of the system, it is actually supporting many users at a time. Often, it is useful to communicate with another user, or with the system operator; this is done through use of the TALK command.

-TALK

Type out a message on another TSS/8 Teletype.

For example, to tell the system operator (Teletype 0) to turn on the high-speed punch, a user types the following (where the initial dot was typed by Monitor):

•TALK Ø PLEASE TURN ON THE HIGH SPEED PUNCH

This command causes the following to be typed on console 0:

where K02 is the number of the physical console which sent the message and 2403 is the account number logged in at that console. If the destination teletype is printing at the time of receipt of message, the message will be interspersed with the useful output, perhaps spoiling it. The TALK command should be used with discretion except when addressing keyboard 0 (zero).

2.7 SYSTEM STATUS REPORTS

The command SYSTAT initiates a typeout of the full status of TSS/8; how many users are on-line, what they are doing, etc.

.SYSTAT

Report system status.

The command SYSTAT is equivalent to typing R SYSTAT. The format of the status report is described in the section on SYSTAT in Chapter 4.

To learn the amount of computer time used since logging in, the user issues the TIME command:

The elapsed processor time of the user since he logged in is printed.

TIME 9

The time of day is printed.

TIME C1

The amount of processor time used by job C1 since login

For example:

•TIME 00:00:09

.

.TIME 0

.

.TIME 02

2.8 RESOURCE SHARING

All TSS/8 users, when logged in, have free access to the System Library, the disk storage capability, and the TSS/8 computer. Monitor automatically handles resource requests on a rotating basis. Monitor also maintains a pool of available devices which must be assigned to be used. These are devices, such as the high-speed paper-tape reader, which by their very nature cannot be assigned to several programs simultaneously. Therefore, Monitor grants individual users exclusive access to these devices when needed. Thus, users still share the device, although not simultaneously. Once a user has ASSIGNED a device, his "device time" bill runs at a rate twice as fast as the real-time clock, and continues to do so until he RELEASEs the device or does a LOGOUT. If he has two devices assigned, the bill runs at triple rate, and so on. Users should use discretion when assigning devices.

All TSS/8 systems include a high-speed, paper-tape reader in the pool of available devices. Many systems also include a high-speed, paper-tape punch, and one or more DECtapes. These assignable devices are normally used with System Library Programs PIP and COPY to store programs or data on paper tape or DECtape.

When a device is assignable (present on the system) and available (not being used), the ASSIGN command may be used to assign the desired unit or units to the console issuing the command. The valid ASSIGN commands are formatted as shown below:

.ASSIGN R

Assign the high-speed paper-tape reader.

ASSIGN P

Assign the high-speed paper-tape punch.

. ASSIGN D

Assign a DECtape unit.

where R, P, and D are device designators for reader, punch, and DECtape, respectively. If other devices are assignable, the system manager will inform the user of the appropriate device designators. The following is an example of using an invalid device designator.

```
•ASSIGN X
ILLEGAL REQUEST
```

Monitor ignores the request, responds with the appropriate message, and prints another dot.

When a valid ASSIGN command is issued, Monitor checks for the availability of the device and responds accordingly. For example:

•ASSIGN R
R ASSIGNED
•ASSIGN P
JOB 02 HAS P

(reader is assignable, available, and assigned)

(punch is unavailable because job number is using it, and thus not assigned)

When the system contains multiple units of a device, the user simply specifies the device; Monitor assigns an available unit and responds with the unit number. For example:

•ASSIGN D
D 2 ASSIGNED

If all DECtape units are busy, Monitor prints the message shown below:

•ASSIGN D DEVICE NOT AVAILABLE A specific unit can be requested, leaving a space between the device designator and the device number.

•ASSIGN D 4 D 4 ASSIGNED

(assignment was accomplished)

NOTE

If the user assigns a device with a nonexistent device number, that device will not be assigned; an error message does not result because that device is not busy. An error message only results when the device is already assigned.

The ASSIGN command can assign only one device at a time. Therefore, when multiple devices are to be assigned, each must be assigned separately. The following will not accomplish the desired assignments, either with or without the illegal commas.

•ASSIGN R. D 2. D 1 R ASSIGNED

Monitor accepted the first device designator (and unit number if any) and ignored the rest of the command. If device R is unavailable, Monitor prints the appropriate message. The following completes the desired assignments (assuming available devices).

•ASSIGN D 2 D 2 ASSIGNED

•ASSIGN D 1 D 1 ASSIGNED

When the user has finished working with an assigned device, the RELEASE command must be used to terminate the assignment and allow other users access to the device. (When a user logs out of TSS/8 any devices he still has assigned to him are automatically released.)

. RELEASE

Terminate a previous device assignment and make the device available to other users.

An assigned device is released when the user types the RELEASE command, a space, the device designator (and unit number if required), and the RETURN key as shown below.

·RELEASE R

•RELEASE D 3

In the previous example, the reader and DECtape unit number 3 are released. Monitor prints a dot on the next line if the release is accomplished; otherwise, it prints a message. If, for example, a request is made to release a device which has not been assigned to the issuing console, the following happens:

```
•RELEASE P
ILLEGAL REQUEST
```

Monitor printed ILLEGAL REQUEST after it checked and found that the specified device was not assigned to the console issuing the command.

NOTE

All commands must be formatted properly; ILLEGAL REQUEST is printed if the user fails to separate the device designator and unit number with a space.

When multiple device units exist on the system, each must be individually released. For example:

```
•RELEASE D 1
•RELEASE D 2
•RELEASE R
```

Monitor does not check when releasing a device as it does when checking to assign an available device. The user could have two device units (for example, two DECtape units) assigned and Monitor would not know which to release; therefore, device numbers are necessary with a RELEASE command. However, where only one unit of a specific device (one reader, one punch, etc.) is on the system, the device designator alone is sufficient. Examples follow.

```
• RELEASE D

ILLEGAL REQUEST

• RELEASE R

• RELEASE D 1

(due to multiple DECtape units)

(accomplished)

• (D 1 is released)
```

2.9 ERROR MESSAGES

An appropriate error message is printed whenever: a Monitor command cannot be performed at the time it was requested, a typing error was made, or the command is illegal (nonexistent). Following each error message, Monitor ignores the request and prints another dot, after which the user can issue another command.

Table 2-1 Monitor Error Messages

| Command | Explanation |
|----------------------|--|
| \$1? | The System Interpreter does not understand the command. \$1 = command |
| LOGIN PLEASE? | The user attempted to use a console which is not logged in. |
| UNAUTHORIZED ACCOUNT | The user attempted to log into the system with an invalid account number or name. |
| ALREADY LOGGED IN? | The user tried to log in on a console which is already in use. |
| FULL | The TSS/8 system is full. Another user cannot log in until one of the present on-line users logs out. |
| TYPE †BS FIRST | The user attempted to use a system command which cannot presently be honored due to the status of the user's program. The message may appear even after the user has typed †BS, since his program may continue until the I/O in progress at the time of the halt is completed. The user should wait a few seconds and then type his command a second time. |
| ILLEGAL REQUEST | The user requested an illegal command. This error usually results when some parameter has been given an incorrect value or the request refers to a facility not owned by the user. |

3.1 FOCARL (Formula Calculator of Carleton College)

FOCARL is an easy to use, easy to learn interpretive language permitting not only "immediate" mode but also programs of as large as 100 lines in length and with data pools as large as 150 variables to be written and executed. Calculations are done with 6 significant figure accuracy over a range from 10^{600} to 10^{-600} . Programs may be stored and recalled from disk, chained together when required, and altered at will. Data may be written and read either sequentially or randomly. Because of its command structure, FOCARL is the easiest language to learn of those on TSS/8, even extending to grammar school children. Error messages are numeric. FOCARL is a rewrite and extension of the DEC supported language FOCAL and has many of its excellent features. It is generally available only on Digital Equipment Company's machines. Further information is available in the FOCARL manual available in the C&MSD main office.

All FOCARL commands may be of the form of a single letter followed by a space. FOCARL variable names are two characters with the first character being any letter except F and the second character being any letter or number. FOCARL permits FORTRAN-like arrays. The commands, their meaning, and examples are shown in figure 3-1. An example of a dialogue between FOCARL and a user is shown in figure 3-. All user input is underlined.

FIGURE 3-1

| Command | Abbrev. | Example | Explanation |
|---------|---------|----------------|--|
| ASK | A | ASK X, Y, Z | Type a : and wait for input. |
| COMMENT | C | CONTINUE WITH | Dummy line. No execution. |
| DO | D | DO 4.1 DO 5 | Do line 4.1 and return. Do all numbers 5.01 and up until RETURN. |
| FOR | F | FOR I=X,Y; | Execute what follows ";" as I goes from X to Y by step of 1. |
| GO | G | GO | Execute program from first num- bered line. |
| GOTO | G | GOTO 5.6 | Execute line 5.6 next. |

FIGURE 3-1 continued. . .

| Command | Abbrev. | Example | Explanation |
|---------|---------|---------------------------|--|
| IF | I | IF (B-4) 2.4, 2.5, 2.3 | Transfer to line 2.4 if B-4<0, to line 2.5 if B-4=0, and to line 2.6 if B-4>0. |
| QUIT | Q | QUIT | A stop execution command. Return to edit phase. |
| RETURN | R | RETURN | Return from DO. |
| SET | S | SET $A = 3$ | An arithmetic statement. |
| TYPE | T | TYPE "A=",A | The output statement. |
| WRITE | W | W 3 | List out all lines of code beginning with 3. |

3.2 BASIC (Dartmouth College's Basic Language)

BASIC is a universal language now available on nearly all major computing machines. Its language structure is between FORTRAN and COBOL in some senses. TSS/8 BASIC is an easy to use compiler language although its command structure is more cumbersome than FOCARL. No "immediate" mode is possible although editing may be done with BASIC. Programs as large as 300 lines in length and with data pools of hundreds of variables in core and thousands on disk or on tape are possible. Storing of programs on disk, recalling them, changing, and on-line editing are possible. In spite of its more complex command structure, its universally makes it the language of choice for many applications. Accuracy is 6 significant figures with numbers as large as 10200. All of the Dartmouth BASIC capabilities except the MAT command are available in BASIC version 5. Further information is available in the BASIC manual available in the C&MSD main office.

All BASIC commands are of the form of a word. BASIC variables are either a single letter or a letter followed by a number. Dimensioned arrays are permitted. The commands, their meaning, and examples are shown in figure 3-2. An example dialogue between BASIC and a user is shown in figure 3-4 with user input underlined.

FIGURE 3-2

| Command | Example | Explanation |
|---------|------------------------|---|
| DATA | DATA 3,4 | Numbers are used by READ statement. |
| DEF | DEF FNA $(X) = X+45$ | A user defined function with X as a dummy variable. |
| DIM | DIM A(34) | Dimension an array for the elements $A(0)$ through $A(34)$. |
| END | END | Each BASIC program must have one END statement. |
| FOR | FOR A=1 TO 9 STEP 2 | Used for looping. If Step is omitted STEP=1. Loop ends with NEXT A statement. |
| GOSUB | GOSUB 100 | Subroutine call. Must end with RETURN statement. |
| IF | IF A=3 GOTO 45 | If test. Comparitors are $\langle , \rangle, = , \langle = , \rangle = , \langle \rangle$. |
| INPUT | INPUT A | Input from keyboard. |
| LET | LET A=34 | An arithmetic statement. |
| EXT. | NEXT A | End of FOR loop. |
| PRINT | PRINT A,B | The output statement. |
| READ | READ X,Z | Read from DATA statement elsewhere in program. |
| REM | REM IS THIS OK | A comment. |
| RESTORE | RESTORE | Reset the pointer so that the next READ gets the first DATA Statement. |
| STOP | STOP | Stop execution. Back to edit phase. |

3.3 FORTRAN (Formula Translator)

TSS/8 FORTRAN is a version of this well known language FORTRAN. FORTRAN programs are written and edited using EDIT and are compiled and executed via the FORT processor. Error messages are numeric. One input file and one output file are permitted a TSS/8 FORTRAN program. No subroutines are permitted. File I/O is via ASCII files, making TSS/8 FORTRAN the language of choice for certain applications. Input and output FORMATS

are E, I or A with no variation of field width possible. Integer variables range from -2047 to +2047 while real variables range from 10^{600} to 10^{-600} . Further information is available in the 4K FORTRAN manual available in the Computing Services office.

3.4 ALGOL (Algorythmic Processor)

TSS/8 ALGOL is SUBSET ALGOL 60 (IFIP) with other restrictions. For example, there are no user procedures, Boolean arrays or two dimensional arrays. There are certain size and format limitations. Format control is equivalent to TSS/8 FORTRAN. ALGOL-8 manuals are available in the Computing Services office.

3.5 PALD (PDP-8 Program Assembler Language - Disk)

PALD is a language in which all of the features of TSS/8 are readily available. All of the machine language instructions are accessible via PALD as well as all of the Executive requests in the system (called IOPs in TSS/8). Complete access to tapes, disk files, real time clock, and bit manipulation are possible with this language package. The system consists of PALD itself, EDIT for creation and editing of source code, LOADER and LINK for creation of executible code from the Binary output of PALD and ODTHI for on-line debugging of the created code. Since TSS/8 itself is quite well protected from user errors, any code the user may wish to write is runnable in TSS/8. PALD may be used to write code for stand-alone PDP/8 series machines, using the disk features of TSS/8 to advantage during the compile and error checking procedures, and loading only the Binary code into the stand alone machinery when successful compilation is achieved and test execution in TSS/8 has been accomplished. A PALD manual is available in the Computing Services office.

```
• R FOCARL
FOCARL, VFRSION 10
FUNCTIONS? NO
* TYPF 2 + 2
4.000000*
* TYPF 22*34.5
 759 . 000*
*1.1 TYPF "FNTFR A VUMPFR"
*1.2 TYPF !
*1.3 ASK X
*1.4 TYPF X12,!
* 1 • 4 TYPF A 200 • 00
* 1 • 5 GO TO 1 • 3
                        ___CNTRL C" TYPED.
*1.5 GOTO 1.3
*WRITF ALL
C FOCARL- 10
@1.10 TYPF "FNTER A NUMPER"
01.20 TYPF !
01.30 ASK X
@1.4@ TYPF X+2,!
01.50 GOTO 1.3
*G0
FNTER A NUMBER
: 12
144.000
: -1
1.00000
                    __ "CNTRL C" TYPED.
: P200.00 @ 01.30
*LFAVE
† PS
```

. R FASIC

NEW OR OLD--Y NEW PROGRAM NAME--TEST

READY

```
10 REMARK THIS IS A TEST PROGRAM

20 INPUT X
30 PRUNT --- INT X12

40 GO TO 10

50 FND

LIST

10 REMARK THIS IS A TEST PROGRAM

20 INPUT X

30 PRINT X12

40 GO TO 10

50 FND

50 FND
```

READY

RUN

READY

30 PRINT X+3

LIST

10 REMARK THIS IS A TEST PROGRAM
20 INPUT X
30 PRINT X t 3
40 GO TO 10
50 FND

READY

RUN

RFADY

FYF †FS

4.1 EDIT

TSS/8 Editor provides the user with a powerful tool for creating and modifying source files on-line. Its precise capabilities and commands are detailed in the PDP-8 Symbolic Editor Programming Manual (Order No. DEC-08-ESAB-D). EDIT allows the user to delete, insert, change, and append lines of text; and then obtain a clean listing of the updated file. EDIT also contains commands for searching the file for a given character.

EDIT considers a file to be divided into logical units, called pages. A page of text is generally 50-60 lines long, and hence corresponds to a physical page of program listing. A FORTRAN-D program is generally 1-3 pages in length; a program prepared for PAL-D may be several pages in length. EDIT operates on one page of text at a time, allowing the user to relate his editing to the physical pages of his listing. EDIT reads a page of text from the input file into its internal buffer where the page becomes available for editing. When a page has been completely updated, it is written onto the output file and the next page of the input file is made available. EDIT provides several powerful commands for "paging" through the source file quickly and conveniently.

NOTE

The end of a page of text is marked by a form feed (CTRL/L) character. Form feed is ignored by all TSS/8 language processors.

To call the Editor, type:

•R EDIT

EDIT responds by requesting INPUT: Type and enter the name of the source file to be edited. If a new file is to be created using EDIT, there is no input file. In this case, strike the RETURN key. EDIT then requests OUTPUT: Type in the name of the new, edited, file to be created. The name of the output file must be different from the name of the input file. If EDIT is being called to list the input file, there is no need to create an output file; strike the RETURN key. When EDIT sets up its internal files and is ready for a command, it rings the bell on the Teletype.

For example:

•R EDIT
INPUT:WXZOLD
OUTPUT:XYZNEW

(Bell rings at this point.)

4.1.1 Summary of Symbolic Editor Operations

 $\begin{array}{cc} \text{Table} & \textbf{4.1} \\ \text{Summary of Symbolic Editor Operations} \end{array}$

| Special Characters | Function |
|--|--|
| Carriage Return (RETURN Key) | Text Mode – Enter the line in the text buffer. Command Mode – Execute the command. |
| Back Arrow (←) | Text Mode – Cancel the entire line of text, continue typing on same line. |
| | Command Mode - Cancel command. Editor issues a ? and carriage return/line feed. |
| Rubout (\) | Text Mode – Delete from right to left one character for each rubout typed. Does not delete past the beginning of the line. Is not in effect during a READ command. |
| | Command Mode – Same as back arrow. |
| Form Feed (CTRL/FORM Combination) | Text Mode – End of inputs return to command mode. |
| Period (.) | Command Mode – Current line counter used as argument alone or in combination with + or – and a number (.,.+5L). |
| Slash (/) | Command Mode – Value equal to number of last line in buffer. Used as argument (/-5,/L). |
| Line Feed (↓) | Text Mode – Used in SEARCH command to insert a CR/LF combination into the line being searched. |
| Right Angle Bracket (>) | Command Mode – List the next line (equivalent to .+1L). |
| Left Angle Bracket (<) | Command Mode – List the previous line (equivalent to .–1L). |
| Equal Sign (=) or Colon (:) | Command Mode – Used in conjunction with . and / to obtain their value (.=27). |
| Tabulation (CTRL/TAB Key Combination) | Text Mode – Produces a tabulation which, on output, is interpreted as spaces if bit 1 of the switch register is set to 0, or as a tab character/rubout combination if bit 1 is set to 1. |

4.1.2 EDIT Command Summary

Table 4-2
EDIT Command Summary

| Command | Format(s) | Meaning |
|-----------|-----------|---|
| READ | R | Read text from the input file and append to buffer until a form feed is encountered. |
| APPEND | A | Append incoming text from keyboard to any already in buffer until a form feed is encountered. |
| LIST | L | List the entire buffer. |
| | nL | List line n. |
| | m,nL | List lines m through n inclusive. |
| PROCEED | Р | Output the contents of the buffer to the output file, followed by a form feed. |
| | nP | Output line n, followed by a form feed. |
| | m,nP | Output lines m through n inclusive followed by a form feed. |
| TERMINATE | Т | Close out the output file and return to TSS/8 Monitor. |
| NEXT | N | Output the entire buffer and a form feed, kill the buffer and read the next page. |
| | nΝ | Repeat the above sequence n times. |
| KILL | К | Kill the buffer (i.e., delete all text lines). |
| DELETE | nD | Delete line n of the text. |
| | m,nD | Delete lines m through n inclusive. |
| INSERT | I | Insert before line 1 all the text from the keyboard until a form feed is entered. |
| | nĬ | Insert before line n until a form feed is entered. |
| CHANGE | nC . | Delete line n, replace it with any number of lines from the key- board until a form feed is entered. |
| | m,nC | Delete lines m through n, replace from keyboard as above until form feed is entered. |
| MOVE | m,n\$kM | Move lines m through n inclusive to before line k. |
| GET | G | Get and list the next line beginning with a tag. |
| SEARCH | S | Search the entire buffer for the character specified (but not echoed) after the carriage return. Allow modification when found. TSS/8 Editor outputs a slash (/) before beginning a SEARCH. |

Table 4 – 2(Cont) EDIT Command Summary

| Command | Format(s) | Meaning |
|------------------|------------|---|
| SEARCH (Cont) | nS m,nS | Search line n, as above, allow modification. Search lines m through n inclusive, allow modification. |
| END | E | Output the contents of the buffer. Read in any pages remaining in the input file, outputting them to the output file. When everything in the input file has been moved to the output file, close it out and return to the TSS/8 Monitor. E is equivalent to a sufficient number of N's followed by a T command. |
| †C | CTRL/C | Stop listing and return to Command Mode. |

4.2 CAT

TSS/8 Monitor maintains a library of disk files for each user. The System Library Program CAT is used to obtain a catalog of the contents of this library. For each file, CAT types the size of the file in units of disk segments. At UWM, the disk segment is 256 (decimal) words of disk storage. The protection code and extension for the file are also given. If the program was created by any of the System Library Programs, it has a protection code of 12, meaning that other users can read the file, but only the owner can change it. To call CAT, type:

·R CAT

4.2.1 Example of CAT Usage

DISC FILES FOR USER 0,57 ON 25-APR-73

| NAME | SIZE | PROT | DATE |
|--------------|------|------|-----------|
| INPUT .BAS | 1 | 12 | 25-APR-73 |
| TWOTWO . BAC | 1 | 12 | 25-APR-73 |
| DATA12.DAT | 1 | 12 | 25-APR-73 |
| LINKER. ASC | 1 | 12 | 25-APR-73 |
| RUNNER . SAV | 1 2 | 12 | 25-APR-73 |

In the example above, the first file has as its name INPUT and has an extension of .BAS. Its size is one disk segment (256 words) and it has a protection code of 12. Protection codes are explained in the Advanced User Guide. It was created on April 25, 1973. The extension of .BAS means that the file is a BASIC program file in source format. The second file TWOTWO.BAC is in BASIC compiled form and can be executed by a RUN command but not listed or changed. The third file DATA12.DAT is a BASIC data file. The fourth file LINKER.ASC is an ASCII file for use by FORTRAN or PALD. The last file RUNNER.SAV is in save format. Other extensions are discussed in the Advanced TSS/8 User's Guide.

4.2.2 S option of CAT

If the user calls the program CAT with an S option:

.R CAT:S

a short form of a system status report is printed showing which job slots are occupied, by which users, and where they are. The terms used in CAT:S are defined in SYSTAT (section 4.3).

4.3 SYSTAT (System Status)

It is frequently useful to know the status of the system as a whole; how many users are on-line, where they are, what they are doing, etc. The SYSTAT program provides this capability. To call SYSTAT, type:

·SYSTAT

·R SYSTAT

SYSTAT responds by printing on the first line: the version of the TSS/8 Monitor being run, the time, and the date. SYSTAT reports the uptime which is the length of time in hours, minutes, and seconds since the system was last put on-line.

SYSTAT lists all on-line users. Each user (WHO) is identified by his account number. The job number (JOB) assigned to him by the monitor and the number of the console he has dialed into (WHERE) are indicated, as is the particular program he is running (WHAT). The exact running state (STATE) of each user, whether he is actually executing a job (RUN), typing in (KEY), or our (TTY), doing input or output to another system device (DISC), in suspension (WAIT) or in control mode (CTRL) is indicated. The amount of computer time used by each user (RUNTIME) since he logged in is given.

If more users are on-line than the system has core fields to hold them, the fact that the system is swapping is reported. The number of free core blocks used internally by TSS/8 Monitor for Teletype buffering and various other purposes is typed out. Then SYSTAT reports any unavailable devices, i.e., devices which are assigned to individual users. The job to which they are attached and their status (AS if they are assigned but not active, AS+INIT if they are assigned and active) is also indicated. Finally, the number of available segments of disk storage is reported.

4.3.1 Example of SYSTAT Usage

•R SYSTAT

STATUS OF UWM'S TSS/8.22 AT 15:17:33 ON 25 APR 73

UPTIME 26:32:33

| JOB | OHW | WHERE | VHAT | STATE | RUNT IME |
|-----|-------|-------|--------|-------|----------|
| 1 | 20,60 | K02 | FOCARL | KEY | 00:00:29 |
| 2 | 10, 0 | K04 | SYSTAT | RUN | 00:00:01 |

AVAILABLE CORE OK FREE CORE=238

BUSY DEVICES: NONE 135 FREE DISC SEGMENTS

TBS

4.4 PIP (Peripheral Interchange Program)

All TSS/8 System Library programs

operate only on files which are on the disk.

Disk is a convenient storage medium for many files; however, it may be more useful to keep some programs on paper tape. PIP provides a convenient means of transferring files between disk and paper tape, for those users who wish to preserve copies of their files off-line.

4.4.1 PIP Conventions

PIP may be considered a link between disk file storage and paper-tape devices. To punch out a desired file, PIP obtains that file from the disk and punches it on paper tape. Similarly, to load a paper tape, PIP inputs the tape from the reader, then outputs it to a disk file.

The way files are named is important to PIP. Files on disk are always named. Paper tapes, on the other hand, have no names as far as the system is concerned (although the user can label the physical tape in any manner he chooses). Paper tapes never have file names; therefore, PIP uses the absence of a file name to indicate a paper tape (absence of a file name is indicated by striking the RETURN key).

The way in which INPUT: and OUTPUT: is indicated provides the means for determining the direction of file transfer. If PIP is to get its input from the disk, the input is a file name; if the input is from a paper tape no file name is given. Similarly, if PIP is to output to the disk, the file name is indicated; if output is to paper tape, no name is given. To call PIP, type:

·R PIP

4.4.2 Using PIP to Load a Paper Tape to a Disk File

To move a paper tape to disk, strike the RETURN key when PIP requests INPUT: Since PIP must output to the disk, respond to OUTPUT: by typing a file name. When PIP requests OPTION: type T to indicate that the

paper tape is being loaded from the Teletype reader. For example:

·R PIP

INPUT: OUTPUT: FILE1 OPTION:T

The paper tape, in the low-speed reader, is read in and stored in the system as FILE1. That tape must have no \uparrow C or \uparrow B character in it.

4.4.3 Using PIP to Punch Out a Disk File

To move a disk file onto paper tape, the use of file names is reversed since PIP must input a disk file and output it to paper tape. The option remains the same. For example:

·R PIP

INPUT:FILE1
OUTPUT:
OPTION:T

The contents of FILE1 are then punched out at the Teletype.

4.4.4 Using PIP with the High-Speed Reader and Punch

PIP can also be used with high-speed paper-tape devices. The format of the INPUT: and OUTPUT: responses is the same. However, for the high-speed reader, the option is R and for the punch it is P.

Since the reader and punch are assignable devices, they are not always available (other users may have one or both assigned). Therefore, whenever PIP is given a command which utilizes one of these devices, it checks to make sure that the device is available. If it is, PIP automatically assigns it (thus, it is not necessary to assign the device before running PIP). If the device is unavailable, PIP so informs the user. For example:

INPUT: OUTPUT: ABCD OPTION:R

PIP reads the paper tape in the high-speed reader and stores it in the system as ABCD.

INPUT:ABCD OUTPUT: OPTION:P PIP punches out file ABCD on the high-speed punch.

INPUT:ABCD OUTPUT: OPTION:P DEVICE NOT AVAILABLE

The punch is assigned to another user, or there is no punch on the TSS/8 system, or there is one but it is turned off.

4.4.5 Using PIP to Transfer BIN Format Files

The examples above work for all ASCII file transfers (except BASIC programs, explained below.) They are also valid for punching out BIN format files with either high- or low-speed devices. Loading BIN format tapes, however, is a special case.

BIN format tapes must end with trailer codes. The easiest way to ensure that they do is to cut off the tape near the end of the trailer code. Failure to do this (or cutting it off very unevenly) does not prevent PIP from loading tape into the disk file. However, later attempts to load the file with LOADER will result in load errors.

NOTE

UWM's ISS/8 will not allow any BIN format tapes to be loaded from the low-speed reader.

4.4.6 Moving Disk Files

PIP can be used to move the contents of one file into another. This is often useful in copying a file from another user's library (providing the file is not protected) into your own library. To copy from disk file to disk file, specify a file name for both input and output. Reply to OPTION: by striking the RETURN key. For example:

INPUT: FOCAL 2 OUTPUT: FOCALX OPTION:

PIP gets FOCAL from account number 2's library and moves it into the file FOCALX.

4.4.7 Deleting Disk Files

One of the principal reasons for punching out files on paper tape is to free disk space. Once punched out, the disk file is no longer needed. PIP offers a convenient means of deleting files, the Delete option:

INPUT:ABCD OUTPUT: OPTION:D

PIP deletes file ABCD, provided that the file is not protected against being changed.

4.4.8 Transferring BASIC-8 Files

BASIC-8 stores its programs in a unique file format. Therefore, it is not possible to load or punch BASIC-8 files in the usual way. To provide a convenient means of handling BASIC-8 programs, the B option is available in PIP.

The B option is used for both reading and punching BASIC-8 programs. The responses to INPUT: and OUTPUT: indicate the direction of the transfer; the high-speed reader or punch is always assumed for the B option. (To read or punch tapes at low-speed, use BASIC-8 itself.)

PIP assumes that any BASIC-8 tapes it loads are clean and error-free. Only tapes actually created by BASIC should be loaded with PIP. Tapes created off-line, and thus liable to contain errors, should be loaded low-speed by BASIC-8 itself with the TAPE command.

4.4.9 fransferring SAVE Format Files

Another special TSS/8 file format is that of the SAVE files, those programs directly executed by TSS/8. (The System Library Programs are examples of SAVE format files.) PIP provides the S option, to allow these files to be punched on paper tape. SAVE format tapes make sense only to TSS/8 PIP. They cannot be input to any other System Program.

The responses to INPUT: and OUTPUT: indicate the direction of the transfer; the high-speed reader or punch is always assumed for the S option.

NOTE

SAVE format tapes include a checksum. If PIP detects an incorrect read, it prints LOAD ERROR, and terminates the load, repeating the request for input.

4.4.10 Summary of PIP Options

| Option | Explanation | | |
|------------|--|--|--|
| , T | Transfer a file between the disk and the Teletype reader or punch. The response to INPUT: and OUTPUT: indicates the direction of the transfer. | | |
| R | Read a tape from the high-speed reader and store it as a disk file. | | |
| Р | Punch out the contents of a disk file on the high-speed punch. | | |
| D | Delete the file specified for input. | | |
| В | Transfer a BASIC-8 program file between the disk and the high-speed reader or punch. The response to input and output indicates the direction of the transfer. | | |
| S | Transfer a SAVE format file between the disk and the high-speed reader or punch. The response to INPUT: and OUTPUT: indicates the direction of the transfer. | | |

4.5 COPY

Many TSS/8 installations include one or more DECtapes. For these installations, DECtape provides a convenient and inexpensive means of file storage. The COPY program is used to transfer files between disk and DECtape.

Any tape which is to be used by COPY must be initialized by an operator or by the system manager by his running program KLEER. The program is reserved to operators and the system manager to prevent accidental erasure of user tape information. COPY format tapes should not be used for other programs (such as direct output of a BASIC program). If they are, they must be reKLEERed by an operator. Tapes purchased from C&MSD are delivered KLEERed and ready for COPY use.

4.5.1 Using and Calling COPY

COPY is the intermediary between disk and DECtape. To write a disk file out to DECtape, COPY inputs the file from the disk, then outputs it to the DECtape. To bring a DECtape file onto the disk, COPY inputs from the DECtape, then outputs to the disk.

Files kept on DECtape have file names just as they do on the disk. To avoid confusion, the user must tell COPY where the file is to be found. If it is on DECtape, the DECtape designation and the number of the DECtape unit must preface the file name. The DECtape number is always separated from the file name by a colon. Thus D1:FILE1 means the file name FILE1 on the DECtape which is currently mounted on DECtape unit number one. The number of available tape units varies among installations. The maximum is eight, (numbered 0 - 7). If a file name is not prefaced by a DECtape number, then this file is assumed to be on the system disk.

Files stored on DECtape do not have protection codes in the sense that disk files do. They are, however, protected against unauthorized access. When a DECtape is not mounted, it is not available to any user. When it

is mounted, it is available only to the user who has assigned the DECtape unit on which it is mounted. Even then it can not be altered unless the DECtape unit is set to WRITE ENABLE.

Users should be sure to assign a DECtape unit before mounting their tape, and dismount the tape before releasing the device. Normally, the DECtape unit to be used should be assigned before calling COPY.

To call COPY, type:

• R COPY

COPY responds by asking which option the user wishes to employ. The COPY options are discussed below.

4.5.2 Loading Files from DECtape

To load a file onto the disk from DECtape, use the COPY option. When COPY requests OPTION- respond with COPY, or C, or strike the RETURN key (the COPY option is assumed). When COPY requests INPUT- type the number of the DECtape unit on which the file can be found (D0, D1, D2, D3, D4, D5, D6, or D7) followed by a colon and the name of the file on the DECtape. When COPY requests OUTPUT- type and enter the name to be given to the output file on the disk. COPY then moves the DECtape file onto the disk. (When using COPY, it is not mandatory to insert a space between the device designator and the device number.) For example:

OPTION- COPY INPUT - D4:PQR OUTPUT - PQR

If COPY cannot find the DECtape specified, (the selected drive does not have a tape mounted on it) it will respond with DECTAPE SELECT ERROR and return to the OPTION request. If the requested file name does not exist on the dectape on the requested drive, COPY prints a new request for INPUT preceded by a ?. If the disk file specified for output already exists, COPY prints a new request for OUTPUT preceded by ?. For example:

OPTION- COPY
INPUT- D4: FORK
DECTAPE SELFCT FRROR

OPTION- COPY
INPUT- D1: FORK
?INPUT- D1: FOCAL
OUTPUT- FILE

4.5.3 Saving Disk Files on DECtape

Saving a disk file on DECtape is very similar to loading one. The option is still COPY. For input, respond with the name of the file on the disk. For output, type the DECtape unit number, colon, and the name to be given to this file. For example:

OPTION - C INPUT - ABCD OUTPUT - D4:ABCD

If COPY cannot find the file on the disk, or if it is protected, COPY prints a ? and repeats the request for input. If COPY cannot set up the desired DECtape file (a file by that name already exists on the tape) COPY types a ? and repeats the request for output. If there is no tape on the specified drive, or if the tape is not WRITE ENABLED, the error message DECTAPE SELECT ERROR is printed and COPY returns to the request for the OPTION.

4.5.4 Listing Directories

COPY can be used to list the directory of a device. To list a directory, respond to OPTION-by typing LIST, or just L. COPY then asks which device it is to list. To list a DECtape's directory, respond with the device name (D0,...,D7). Do not follow it by a colon. For example:

OPTION- L DEVICE- DO

1036. FBFF PLOCKS

NAME SIZF DATE 0112 •ASC 32 13-FFP-73

The unit of DECtape storage is the block, which is 128 (decimal) words. Because the unit of disk storage, the segment, is generally 256 words, a file occupies twice as many blocks of DECtape storage as it did segments on the disk.

COPY can also be used to list the user's disk directory. Use the LIST option, but respond to DEVICE- by simply striking the RETURN key. The directory listing is similar to the listing obtained by running the CAT program.

4.5.5 Deleting Files

COPY can be used to delete files, either on the disk or on a selected DECtape. To delete a file, respond to OPTION- by typing DELETE, or just D. Respond to INPUT- by typing the name of the file to be deleted.

If the file is on a DECtape, preface the file name with the DECtape unit number and a colon. For example:

OPTION - DELETE INPUT - D4:ABCD

If COPY cannot find the file to be deleted, or having found it, cannot delete it (it is a protected disk file or a DECtape file on a unit which is not WRITE ENABLED), COPY prints a ? and repeats the request for INPUT-.

4.5.6 Moving Files From One Place to Another

COPY can be used just like PIP to copy a file from another user's space to yours. To do so, respond to OPTION- by typing C or COPY to the INPUT-request by typing the file name and user number, separated by a space, and to the OUTPUT-request by typing the name desired:

OPTION-C INPUT -CAT 2 OUTPUT-CATALG

Will copy the file CAT from user 2's file space to the user's space in the location CATALG.

4.5.7 Renaming a File

Copy can be used to rename a file or change its extension. To do so, respond to the OPTION- request by typing R or RENAME and the name and location (if on tape) of the file after INPUT-. The new name may be typed after OUTPUT-. If no extension is typed after OUTPUT-, the extension is unchanged. If one is, it becomes the new extension.

OPTION-R INPUT-LOOP OUTPUT-LOOP.BAS

(LOOP used to have an extension of .ASC)

4.5.8 Control C

At any time in COPY the user may type a letter C while holding down the CNTRL key. Copy will abort its current operation, respond with the symbol \P C, and return to the OPTION- mode.

4.5.9 EXIT from COPY

To leave COPY respond to OPTION- by typing E. This will release all drives used in COPY.

4.5.10 Summary of COPY Options

| <u>Option</u> | Explanation |
|---------------|--|
| COPY | Transfer a file between disk and DECtape |
| LIST | List a directory |
| DELETE | Delete a file |
| RENAME | Change name or extension or protection |
| EXIT | Leave copy and release drives |

4.5.11 UWM TSS/8 COPY Conventions

If an operator is on duty (as indicated in the LOGIN message or by someone logged in on KOO) he may be asked to mount a particular tape on a drive by a TALK O command. The user should assign a drive before issuing that command:

```
•AS D
D 2 ASSIGNED
•TA @ PLFASE MOUNT SRMMI ON DR
•
** K@@/@@#3: GA
B COPY
```

```
OPTION- PTA @ DONE WITH D2. THANKS.
** K@0/0043: GA
OPTION- F
PS
```

In the above, the user assigned drive 2, asked the operator to mount a tape (#2001) on drive 2 with write enabled (\$) and then used the tape with COPY. Before ending with the E option, he talked to the operator again (using a CTRL/B to return to MONITOR) asking to have the tape removed. After the operator indicated that he had complied, the user exited with the E option. Note that if the user had exited COPY before insuring that the tape had been removed he would have left an assignable tape free in a mode in which it could have been written on. This is bad practice.

When no operator is on duty, the system is left with an auxiliary libary tape on drive 1 with the WRITE ENABLE switch off. Any user may copy any of the files from this tape for his own use. On drive O is left a scratch tape with the WRITE ENABLE switch on. Any user may write files on this tape. Users are requested to not delete files not belonging to them, and no to add useless files to this tape.

4.5.12 Example of COPY Usage

• R COPY

OPTION- C INPUT- D0:0112 OUTPUT- 0112

OPTION- L.
DEVICE-

DISC FILES FOR USFR 24.3 ON 15-FEF-73.

| NAME | S | IZF | PROT | DATE |
|--------|-------|-----|------|-----------|
| QUFUFZ | • PAS | 3 | 12 | 7-FFP-73 |
| ALLOC | · PAS | 3 | 12 | 8-FFF-73 |
| CREATE | • PAS | 1 | 12 | 13-FEF-73 |
| INPUT | PAS | 3 | 12 | 13-FFF-73 |
| AVG | FAS | 3 | 12 | 13-FFF-73 |
| CAVD | PAS | 3 | 12 | 13-FEB-73 |
| 0112 | •ASC | 16 | 12 | 15-FFF-73 |

TOTAL DISC SECMENTS: 32

OPTION- D INPUT- 0112

OPTION- R INPUT- QUEUFZ OUTPUT- QUEUE <10>

OPTION- L DEVICE-

DISC FILES FOR USER 24.3 ON 15-FFP-73.

| NAMF | 9 | IZF | PROT | DATE |
|--------|-------|-----|------|-----------|
| QUEUF | • PAS | 3 | 10 | 7-FEP- 73 |
| ALLOC | • PAS | 3 | 12 | 8-FEP-73 |
| CREATE | PAS | 1 | 12 | 13-FEB-73 |
| INPUT | · FAS | 3 | 12 | 13-FER-73 |
| AVG | •FAS | 3 | 12 | 13-FEP-73 |
| CAND | • PAS | 3 | 12 | 13-FEP-73 |

TOTAL DISC SEGMENTS: 16

OPTION- F

4.6 DUMP

DUMP is a program which will dump a disc file in octal. DUMP may be executed by typing:

.R DUMP

The program will begin by typing:

FILE:

You should then enter the name of the file you want to dump. The file name may have several forms. The following list illustrates some of the valid forms:

FILE: PROG FILE: "PROG"

FILE: PROG N ('N' is the account number of the owner)

FILE: "PROG" N

FILE: PROG* ('*' indicates the system library)

FILE: "PROG"*

If an imbedded space, slash, or asterisk, or a leading quote (") is desired in the file name, the name must be enclosed in quotes ("). If the name is enclosed in quotes, a quote character in the file name must be given as 2 consecutive quotes. Thus, the file name AB"BA can be specified in either way:

FILE: AB"BA FILE: "AB""BA"

The program will then proceed to dump the file in octal; the dump may be terminated at any time by typing fC (ctrl/c).

A second method of specifying the file name is as follows:

.R DUMP: PROG

Here again, the file name may have any one of the above forms.

If the dump is desired to start at an address other than zero, you may specify the starting address by typing a slash after the filename-account specification followed by the desired starting address. The following examples illustrate this for a dump of file "prog" beginning at address $\emptyset 4 \emptyset \emptyset$:

.R DUMP:PROG / ØØØ4ØØ

.R DUMP: PROG $426\emptyset/4\emptyset\emptyset$ (Here "PROG" belongs to user $426\emptyset$)

.R DUMP

FILE: "PROG"/400

FILE: $PROG^*/4\emptyset\emptyset$ (Here "PROG" is from the system library)

4.7 LABLDP

LABLDP is a program which allows the user to label a tape punched on the high-speed punch; after the tape is labeled, the program loads and starts PID, which may then be used to punch out a disc file immediately following the label.

To execute LABLDP, make sure the punch is available, then type:

.R LABLDP

When LABLDP is running, each character typed will be punched as a large readable label on the high-speed punch. When you have punched the label, type †C (ctrl/c); this will cause PIP to be loaded and started. Your disc file may then be punched out normally using PIP.

4.8 LINK

LINK is a program which will input up to 20 binary-format files and output a save-format file. LINK may be executed by typing

.R LINK

The program will begin by typing

INPUT:

You should then enter the name(s) of the input files. The filenames may be in any of the following forms:

INPUT: FILE

INPUT: FILE1, FILE2

INPUT: PROG 123 ('PROG' belongs to user 123)
INPUT: PROG* ('PROG' is in the system library)

INPUT: "PROG", PROG, FILE 123, "TEST 1"

If the filename contains an imbedded space, comma, or asterisk, or a leading quote ("), the name must be enclosed in quotes ("). If the name is enclosed in quotes, a quote character in the file name must be given as two consecutive quotes. Thus the name AB"BA may be given in 2 ways:

INPUT: AB"BA
INPUT: "AB""BA"

In addition, the following control characters are recognized:

Rubout - Deletes one character at a time

Ctrl/U - Deletes the entire line

Ctrl/C - Restarts the program; INPUT: is again requested

After all, input files are checked for availability, link types OUTPUT:

You may enter one file name in any of the above forms or you may just type <return>. If you enter a file name, the file is opened or, if it does not exist, it is created. If no output file is specified, link scans the input file(s) as if they were being converted but the output is merely thrown away. If you specify an output file, link will type

OPTION:

Here you may type an "O" (oh) or <RETURN>. If you type <RETURN>, link will reduce the output file to one segment and zero that segment. If you type "O", link will overlay the output file with the new data; thus providing a simple way to apply large patches to programs. Link will always zero each new segment as it is added to the output file.

If link finds no errors during the conversion, it will finish by typing the message "DONE".

4. 9 LOGOUT (A program to log the user out of TSS/8)

LOGOUT may be called by typing:

.R LOGOUT

or

.LOGOUT

or

.K

It logs the user out of the system, releases the teletype port for another user, releases all assigned devices (DECtapes, paper tape reader, etc.) and deletes all files with a .TEM extension. LOGOUT permits various options which delete ALL unprotected user files (K), list all files before logging out (L), or permit the user to individually save or delete files (I).

To use one of these features, the user types

.LOGOUT:L

which will log the user out and delete all of his unprotected files.

.LOGOUT: I

will produce a line by line listing of all files. The user may respond at the end of each line with the letter P and a carriage return to protect the file from erasure (use P only if you know how to change the protection mask, as later deletion will not be possible without this knowledge); with the letter S and a carriage return to save the file without protection; or only a carriage return to delete the file. Files with .TEM extension are deleted without any response by the user. A typical episode with LOGOUT:I is shown below.

4.10 LOGIN (a program to print the message of the day)

The program LOGIN is automatically run when a user logs into the TSS/8 system. It permits a message to be typed indicating whether or not an operator is on duty, and printing his operator number if one is present. LOGIN then prints the "message of the day" for each user as he logs into the system. LOGIN's output can be aborted at any time by the user typing [CTRL] C (the letter C with the shift button marked [CTRL] depressed) if seeing a repetition of the LOGIN message is not desired. Users are reminded that crucial information is sometimes stored in the message and it should not be bypassed until enough of it has been seen to insure that there is nothing new in it for you. A repetition of the LOGIN message will result if the user types:

.R LOGIN when in monitor mode. Typical LOGIN output is of the form:

·R LOGIN

OPERATOR 57 ON DUTY. TSS/8 WILL BE HALTED AT 10:30 THIS EVENING FOR INSTALLATION AND TESTING OF NEW MONITOR COMMANDS. USERS WHO WILL BE ADVERSELY AFFECTED SHOULD CALL THE NUMBERS BELOW.

PROBLEMS? CALL 963-5440 OR -4010.

tBS

LOGIN is a UWM written program and will run only on TSS/8 systems which have been modified for UWM changes.

4.11 CPASS (a program to change passwords)

CPASS is a program which permits certain account holders to change their passwords at will. In UWM TSS/8 only those accounts whose third digit is 0, 1, 4 or 5 can change their password using CPASS. Accounts whose third digit is 2, 3, 6 or 7 cannot. Thus account 2403 can use CPASS while account 2423 cannot. Normal practice is to give non-CPASS accounts to classes where no individual student should be permitted to "lock out" the class by changing the password for the class account. (Instructors of such classes are able to change the password by requesting this service from the system manager.)

To use CPASS, type:

.R CPASS

The program will respond:

OLD PASSWORD: The user must type in his present password followed by a [return]. (Any error at this time will result in the user being logged out without warning with the password left unchanged.) The old password will not echo to insure its security.

When the old password has been accepted by the program, it will respond:

NEW PASSWORD

: The user may enter a new password (4 characters) followed by a [return]. The new password will echo so that the user may read it and copy it down but it will be overstruck after the [return] key is pressed to maintain its security. The system will then respond as to whether the password has been changed or not depending on the account number (see above).

[CTRL] C may be struck at any time to halt CPASS without changing the password. CPASS is a UWM written program and will work only on TSS/8 systems which have been modified for UWM changes.

COMPUTING SERVICES PUBLICATION

THE UNIVERSITY OF WISCONSIN-MILWAUKEE

OCTOBER, 1973

PUBLICATION NUMBER 102

TSS/8 LABLDP

A PROGRAM WHICH PUNCHES BLOCK CHARACTERS INTO PAPERTAPE

Description

LABLDP is a program which runs in a TSS/8 environment which punches large block characters into a papertape in easily readable format. The user enters characters into a "punch buffer". When either control C or "Bell" is struck, LABLDP will punch the contents of that buffer and then automatically cause PIP to be swapped into user space and started. LABLDP thus can be easily used to label a papertape which may then have a file dumped onto it by PIP.

Use

To use LABLDP type:

R LABLDP

while in Monitor mode. If the papertape punch is already in use, LABLDP will respond, "PUNCH BUSY," and halt. Otherwise, LABLDP will assign the punch and permit the user to enter text. An examination of the text already entered can be obtained at any point by striking the LINE FEED key. If the text to be entered extends for more than one line, the CARRIAGE RETURN key may be used. (CARRIAGE RETURN inserts one space into the punch buffer.) If a typing error is made, the RUBOUT key will echo and delete one character from the output string each time it is struck.

Preceding the text entered by the user, LABLDP puts in the punch buffer the account number of the user enclosed by brackets, and the current date. If this information is not desired as part of the label, the RUBOUT key may be struck several times in order to erase any desired portion.

The characters are punched into the papertape in a format 6 bits high by 5 bits wide. Bits 1 through 6 are used with bit 1 at the top of the letter. Bit 8 is always punched as an underscore. This underscore may be removed by depositing 0000 in location 0010. Any of the ASR 33 printable characters (ASCII 240-337) may be used. All other characters are ignored.

Special Characters

LINE FEED: Typing a line feed causes the current punch buffer to be output on the user teletype. If the punch buffer is more than 80 characters long, a CR-LF sequence is inserted into it for the sake of appearance. The line feed may be used at any time.

RUBOUT: The rubout causes the last character input to be echoed and then deleted from the punch buffer. As many characters may be erased as desired. If too many rubouts are used, the BELL will be rung.

CARRIAGE RETURN: This character causes a CR-LF sequence to be sent to the user console and a space to be sent to the punch buffer.

Bell (CTRL G) or

CTRL C

Send punch buffer to the punch, load PIP, jump to location O. The punch is not released. (Because of the nature of the timing loops in TSS/8, Bell is a safer method of exiting LABLDP than CTRL C.)

System Information

LABLDP occupies two segments (512 words) of disc space and requires that PIP be available on the system library (user 2). LABLDP will run properly in its present form through January 4, 1975.



TSS/8
FOCARL







THE UNIVERSITY OF WISCONSIN-MILWAUKEE
COMPUTER AND MANAGEMENT SERVICES DIVISION

INTRODUCTION

This manual describes FOCARL (Decus No. 8-329), a superset of FOCAL, 1969 developed at Carlton College for the TSS/8 time-sharing system. This manual is not intended as a FOCARL primer; rather, it is a brief description of the FOCARL language with emphasis placed on the differences between FOCARL and FOCAL, 1969. The only obvious difference to the casual FOCAL user is the absence of the equal signs when typing the results of calculations.

To use FOCARL, type:

.R FOCARL

FOCARL will identify itself and ask

FUNCTIONS?

The possible responses to this are described under FOCARL Functions (page 3).

TABLE OF CONTENTS

| | Page |
|---|--|
| General Notes | 1 |
| Special Characters | 1 |
| FOCARL Abbreviations | 2 |
| FOCARL Functions | 3 |
| FOCARL Variables | 4 |
| Basic Command Summary | 6 |
| ASK TYPE SET DO RETURN FOR QUIT IF GOTO WRITE ERASE MODIFY COMMENT LOGOUT | 6 8 8 9 9 9 10 10 10 11 12 |
| High-Speed Papertape Use | 12 |
| Disc Storage of Programs | 12 |
| FOCARL LIBRARY Commands | 13 |
| LIBRARY SAVE LIBRARY CALL LIBRARY DELETE LIBRARY EXTEND | 13 13 14 14 |
| Line Printer Output | 15 |
| Low-Speed Papertage Use | 15 |

| | <u>Page</u> |
|--|----------------------|
| Disc Data Files (%L) | 16 |
| Output Files Input Files Using the Same File for Input and Output Notes on Using Data Files | 16 17 17 18 |
| ASCII Format I/O (%A) | 19 |
| Evaluation of Alphabetic Responses to INPUT | 20 |
| Execution Notes | 21 |
| Appendix A - FOCARL Command Summary | 24 |
| Appendix B - Table of Character Codes | 26 |
| Appendix C - FOCARL Error Codes | 29 |
| Appendix D - Sample Program with Tracing | 31 |

.

GENERAL NOTES ON THIS MANUAL

- (CR) denotes a carriage return.
- 2. The only spaces necessary in a line are the spaces separating the command (and sub-command in LIBRARY commands) from its argument. The other spaces shown in the command lines in this manual are included for ease of reading.
- 3. The arrow (♠) is used in this manual to denote control characters. For example, ♠C indicates the C pressed while holding the CTRL key.

KEYS WITH A SPECIAL FUNCTION IN FOCARL

CARRIAGE RETURN (CR)

Typing a carriage return directs FOCARL to analyze and execute the line just typed in. Until a carriage return is typed, FOCARL merely reads in the characters one-by-one and stores them. FOCARL also generates a line feed in response to every user-typed carriage return. In a MODIFY command, a carriage return terminates the line at the last character typed, deleting the untyped portion of the line.

LINE FEED (LF)

Recognized by FOCARL as equivalent to a CR, except in a MODIFY command; in a MODIFY command, it retains the remainder of the searched line.

ALTMODE

When used in response to an ASK statement, it directs FOCARL to retain the former value of the variable.

RUBOUT

Except when inputting data, RUBOUT will delete one previous character for each time it is struck, up to the * at the left margin, or up to the line number when used in a MODIFY command.

←(SHIFT O)

In response to an ASK command, deletes the value just typed in, allowing the user to type in another value.

In a MODIFY command, deletes that portion of the line already typed (doesn't delete the line number).

When typing in a program line or a command string, ←deletes that part of the line to the left of it.

HERE IS

Used for punching leader/trailer tape when punching a low-speed paper tape on the TTY punch.

CONTROL CHARACTERS (denotes control key depressed)

- †C--used to stop execution of a program running in FOCARL
- ♦G--used in a MODIFY command to change the search character
- †G--used to sound the TTY bell when in an I/O statement
- L--used in MODIFY command to continue listing line up to the next occurrence of the search character
- R--unduplexes TTY to read in paper tapes on TTY tape reader
- ↑T--duplexes TTY; used following ↑R to restore printing

ABBREVIATIONS ALLOWED IN FOCARL

All commands in FOCARL may be abbreviated to their initial letters. Thus, any word with the same initial letter as a FOCARL command will be interpreted as that command. Examples:

COMMAND EFFECT

TRY A Types the value of A

GOSH Executes program starting at beginning DARNIT ALL Executes program starting at beginning

There are two pairs of commands with the same initial letters:

GO/GOTO LOGOUT/LIBRARY

G without an argument is GO--begin execution at lowest line number .

G with an argument (e.g., G 1.07) is GOTO--begin execution at line number specified.

L without an argument: LOGOUT--return to monitor.

L with arguments: LIBRARY command.

First argument should be sub-command (C,D,S,X,I,O) Second argument is file name.

Third argument is line number (in LIBRARY XTEND only).

Multiple commands may be specified on one line, so long as they are separated by semi-colons. There are certain restrictions:

- 1. Any command string following a FOR command on the same line will be executed once for each iteration of the FOR command.
- 2. Any command following a COMMENT statement on the same line is assumed by FOCARL to be part of the comment and won't be executed.
- 3. Any command following an ERASE or LIBRARY DELETE command on the same line will not be executed.
- 4. In input/output statements, commas separating direct quotes, !'s, and #'s from the arguments following them may be omitted.
- 5. Execution of a MODIFY, WRITE, QUIT, ERASE (with an argument), LIBRARY CALL, or LOGOUT command in a program will terminate program execution.

FOCARL FUNCTIONS

In the list of function forms and comments which follows, (A) denotes a number, a variable, or an arithmetic expression e.g., FABS(BC), FSQT(7), FITR (SR*4\)(L*2.1)), FABS (FCOS (1.57)).

PERMANENT FUNCTIONS

| FSQT(A) | A=>Ø; otherwise error |
|----------|------------------------------------|
| FABS (A) | absolute value of A |
| FSGN(A) | yields +1 if $(A) = > \emptyset$; |
| | -1 if (A) < Ø |
| FITR(A) | yields integer portion |
| | of number, not greatest |
| | integer <= |
| | FITR(7.4) = 7, $FITR(-3.5) = -3$ |
| FRAN(A) | here the value of A does |
| | not influence the value of |
| | the function call; generates |
| | a number between 0 and 1 |
| | FABS (A) FSGN (A) |

Close input file FCLS(A) uninitializes data file as input file

Line printer output FLPT(A) if A=Ø, output on line printer; if A>Ø or A<Ø,

TTY output

OPTIONAL FUNCTIONS

| Exponential | FEXP(A) | raises E (2.71828) to the |
|-------------------|---------|------------------------------------|
| | | (A) power $-616 \le (A) = \le 616$ |
| Natural logarithm | FLOG(A) | FLOG(2) = .69315, $FLOG(2.71828)$ |
| | | =1.00 |
| Arctangent | FATN(A) | (A) is an angle in radians |
| Cosine | FCOS(A) | (A) is an angle in radians |
| Sine | FSIN(A) | (A) is an angle in radians |

RETAINING OPTIONAL FUNCTIONS

Legitimate responses to the question FUNCTIONS? are:

| AL(CR) or ALL(CR) | Retains all optional functions |
|------------------------------|--------------------------------|
| N(CR) NONE(CR) or (CR) | Retains no optional functions |
| FEXP(CR) EXP(CR) or E(CR) | Retains exponential function |
| FLOG(CR) LOG(CR) or L(CR) | Retains natural logarithim |
| FATN (CR) ATN (CR) or A (CR) | Retains arctangent |
| FCOS(CR) COS(CR) or C(CR) | Retains both sine and cosine |
| | functions |
| FSIN(CR) SIN(CR) or S(CR) | Retains both sine and cosine |
| | functions |

To retain several of the functions, separate the names with commas. (E.g., FUNCTIONS? E,FLOG,ATN(CR))

All responses must be followed by a carriage return. If the response was not understood, FOCARL will ask FUNCTIONS? again; when a correct response is given, FOCARL types an *.

VARIABLES IN FOCARL

VARIABLE NAMES

A variable name in FOCARL must begin with a letter other than F. F is used to denote a function call and cannot be used as the first letter of a variable name. After the initial letter, any alphanumeric character (a letter [A-Z] or a digit $[\emptyset-9]$) may follow. FOCARL only remembers the first two characters of a variable name, so any two variables whose names begin with the same two first characters are considered to be the same.

*SET VIAL=12(CR)
*SET VILE=35(CR)
*TYPE VIAL, VILE(CR)
35 35*

*SET S123=7.Ø2(CR)
*SET S124=14/3(CR)
*TYPE S123,S124(CR)
4.667 4.667*

SUBSCRIPTED VARIABLES

You may differentiate between two variables with the same first two (or more) characters by using subscripts. The subscript must be a non-negative integer, enclosed in parentheses. Any non-subscripted variable is assumed to have a subscript of zero.

*SET C=3.Ø2(CR)
*SET C(Ø)=5.8(CR)
*TYPE C,C(Ø)(CR)
5.8 5.8*

LISTING THE DEFINED VARIABLES

To get a list of the variables that have been defined and their current values (stored in the symbol table), type TYPE \$ (CR).

ARITHMETIC OPERATIONS AND PRIORITIES

- + Addition
- Subtraction
- * Multiplication
- / Division
- **↑** Exponentiation

The form is A † B, where A is an arithmetic expression and B is a non-negative integer.

The priority of arithmetic operations is:

- 1. Exponentiation
- 2. Multiplication and division (equal priority)
- 3. Addition and subtraction (equal priority)

Evaluation proceeds from left to right according to the above priorities, respecting the associations of parentheses. Thus the expression, 5.2*(3.7/2.2*4-8.99/(FSQT(4.8)*7)) is evaluated as follows:

```
5.2 * (3.7 / 2.2 † 4 - 8.09 / (FSQT(4.8) * 7))

5.2 * (3.7 / 23.4256 - 8.09 / (FSQT(4.8) * 7))

5.2 * (0.15795 - 8.09 / 2.19089 * 7)

5.2 * (0.15795 - 8.09 / (15.3362))

5.2 * (0.15795 - 0.52751)

5.2 * (-0.36956)

-1.92.72
```

COMMAND SUMMARY

INPUT/OUTPUT COMMANDS: ASK, TYPE

ASK

The ASK command is the input statement of the FOCARL language. It allows the user to enter numerical values which the computer stores and references as named variables. FOCARL types a ":" to indicate that it is waiting for a numerical value to be entered. The user must type a delimiter (see next paragraph for allowable delimiters) following the number, before FOCARL will recognize the value. This is necessary because FOCARL does not use formatted input. If a mistake is made in typing in an input value and a delimiter has not been typed, the error can be corrected by typing \leftarrow (shift 0) and then typing the correct value followed by a delimiter. Hitting the ALTMODE key in response to the ASK statement's ":" instructs FOCARL to retain the former value of the variable, instead of accepting an input value.

The following characters may be used as delimiters following a value input in response to an ASK command: comma, colon, semicolon, carriage return, or a space.

Any of the delimiters, except a space, may be used without typing in a value to specify a value of zero for the variable. A space must be preceded by a value, a minus sign or a decimal point before it will serve as delimiter.

TYPE

The TYPE command is FOCARL's output command. It is used to print out the results of the computer's calculation so they can be seen and interpreted by the user.

OPTIONS IN BOTH ASK AND TYPE COMMANDS

QUOTE

A character string enclosed in quotation marks will be reproduced verbatim when that statement is executed. $\dagger G$ (bell) is the only special character which will be reproduced correctly.

EXCLAMATION POINT (!)

An exclamation point as an argument of a TYPE or ASK statement will cause FOCARL to generate a carriage return/line feed combination.

NUMBER SIGN (#)

as an argument of an I/O command will generate a carriage return without a line feed.

DOLLAR SIGN (\$)

The \$ option will generate a print-out of the symbol table, a list of all of the defined variables and their current values.

OUESTION MARK (?)

A variable name enclosed in ?'s in an I/O statement will be typed out. The ASK statement will type a colon after the variable name, and then wait for the user to type in the value; the TYPE statement will follow the variable name with its current value. Multiple variables can be included in the same set of ?'s, as long as they are separated by spaces or commas. In this instance, FOCARL will type both the variable name and the delimiter before typing the value (TYPE) or :(ASK).

NUMERIC FORMATTING IN THE TYPE STATEMENT

There are three types of numeric formatting available in FOCARL: integer, floating-point or decimal, and exponential or E format. The format is specified by using the % option.

%b Yields b-place integer output;

%a.bc Yields a-place output with up to bc
digits to the right of the decimal point;

Yields exponential format: a six place decimal followed by the letter E, followed by the power of ten to which the decimal is to be multiplied by to yield the correct value.

If a number has more significant digits to the left of the decimal point than is allowed for by the integer or floating-point format that number will be output under a modified E-format: only the number of significant digits (up to 6) given in the specified format are retained in the decimal part of the E-formatted number.

If a number contains more significant digits to the right of the decimal than is allowed under the output format, the number will be rounded to fit the specified format.

Once a format is specified, output will continue to be typed under this format until another format is requested or a default condition causes output to be typed in E-notation as explained above.

ASSIGNMENT STATEMENT

SET

The SET statement associates a numeric value with a variable name. The syntax of the set statement is: SET va=ae, where va is a variable and ae is an arithmetic expression, i.e., a constant, variable function, or formula.

CONTROL COMMANDS: DO, RETURN, FOR, QUIT

DO

The DO command is used to suspend sequential execution of commands, execute intervening commands, and then return to sequential execution. The intervening commands can be a single line or an entire group as specified by the argument of the DO command. The DO command without an argument, used as a direct command, will be interpreted as a DO ALL command by FOCARL. This begins execution at the smallest line number.

RETURN

The RETURN command is used in conjunction with a DO statement. A RETURN command signals the end of execution of a DO command. Execution continues with the next sequential command, or, if the DO was nested internal to another DO command or FOR iteration, execution continues as they direct.

FOR

The FOR command is used to perform iterations, the syntax of a FOR statement is:

FOR VA=VI, VF, IN; command string

When the FOR command is executed, the variable, denoted by VA, is set equal to the initial value, VI. With VA=VI, the command string following the ";" is executed, then the increment, IN, is added to VA and this value is checked against the final value, VF. If VA<=VF, the command string is executed again, VA is incremented by IN, and its value again compared with VF. This cycle continues until VA>VF. Then execution is directed to the next sequential line, unless the FOR loop was nested within another FOR loop. With nested FOR loops, the completion of the iterations of the inner FOR loop is the completion of only one execution of the command string of the outer FOR loop. Thus, the inner FOR statement will be directed through all of its iterations for each iteration of the outer loop. Thus, the statement:

FOR I=1,4,1; FOR J=1,3,1; TYPE "*"

Will cause the TYPE command to be executed 12 times. If no increment is specified in the FOR statement, FOCARL assumes an increment of 1. Thus, the statement could have been written:

FOR I=1,4; FOR J=1,3; TYPE "*"

QUIT

The QUIT command terminates the program execution.

BRANCH COMMANDS: IF, GOTO

IF

The IF statement is called a conditional branch statement, that is, certain conditions (values of variables, usually) must exist before it will transfer control (branch) to another part of the program. The format of the statement is:

IF (AR) AB.CD, EF.GH, IJ.KL

Where AR is an arithmetic expression, and AB.CD, EF.GH, and IJ.KL are line numbers, the IF statement checks for three conditions: $\langle \emptyset, =\emptyset, \rangle \emptyset$. If the value of the arithmetic expression is negative, it branches to line AB.CD; if zero, to line EF.GH; if positive, to line IJ.KL.

GOTO

GOTO is an unconditional branch statement. Whenever a GOTO AB.CD command is encountered, the program continues execution beginning at line AB.CD until the end of the program or until a QUIT statement terminates the program run.

EDITING COMMANDS: WRITE, ERASE, MODIFY

WRITE

The WRITE command is used for listing out the user's program. There are three options to the WRITE command:

WRITE AB.CD Prints out line AB.CD (if it

exists)

WRITE B Prints out all of group B (if

it exists)

WRITE or WRITE ALL Lists out the entire program

The WRITE command can be used either as a direct or an indirect command, but there is little reason to use it as an indirect command.

ERASE

The ERASE command is used to delete information stored in core. There are four options to the ERASE command:

ERASE AB.CD Deletes line AB.CD from your pro-

gram (if it existed)

ERASE B Deletes all of group B (if it existed)
ERASE Erases the symbol table (i.e., sets

all variables = Ø)

ERASE ALL Erases entire program and symbol table

The option ERASE is the only one which should be programmed indirectly. It is often included at the beginning of the program before any variables have been defined, to set all variables equal to zero, eliminating any "carry-over" from the previous program.

MODIFY

The MODIFY command is used for correction of a program line without making the user retype the entire line. The MODIFY command cannot be programmed indirectly, since it necessitates supervisory interaction by the user. The format of the MODIFY statement is:

MODIFY AB.CD

Where AB.CD is the number of the line to be modified, FOCARL waits for the user to type in the character he wants to search the line for. It will type out the line until it finds the search character, it will then wait for the user to perform one of the following options:

- 1. To type in the new text following the search character.
- 2. Hit the RUBOUT key to delete the last character typed.
- 3. Type left arrow (←; shift 0) to delete that part of the line that has been typed out.
- 4. Hit the return key to terminate the line at the search character, deleting the remainder of the line.
- 5. Type form-feed (\L; CTRL/L) to continue typing out the line until another occurrence of the search character.
- 6. Type bell (G) to change search character, and then typing the new search character.
- 7. Type line feed to save the rest of the line.

If it doesn't find the search character, it will type out the entire line and not alter it in any way.

EXECUTION COMMANDS: GO, GOTO, DO

GO(CR), DO(CR), or DO ALL(CR) causes execution of the program starting with the smallest line number.

GOTO AB.CD(CR), used as a direct command, starts program execution at line AB.CD.

DO B(CR) as a direct command, executes only group B.

MISCELLANEOUS COMMANDS

COMMENT The COMMENT command is used to include comments in a program listing. Anything following the COMMENT is stored as text and not executed.

LOGOUT The LOGOUT command is used to exit from the FOCARL language and to return to TSS/8 monitor. The user must then logout of monitor by typing LOGOUT or K in response to monitor's "." before he is off the system, i.e., no longer logged in.

USAGE OF PERIPHERAL DEVICE IN FOCARL

HIGH-SPEED PAPER TAPE READER

Mount the tape in the reader so that the following conditions are satisfied:

- 1. The holes in the tape match up with the gears in the reader.
- 2. The leader tape, not your program, is over the read head. Type the command (assuming that you are in FOCARL) H R. The tape should begin reading. When it has been read in completely, FOCARL will respond with an *. Remove the tape from the reader.

NOTE

You must not read oiled paper tape in the high-speed reader!

HIGH-SPEED PAPERTAPE PUNCH

To punch your program on the high-speed punch, make sure the punch is available, then type: H P

FOCARL will punch your program and then release the punch and type *.

DISK STORAGE OF USER PROGRAMS

For most purposes, disk storage is to be treated as only temporary storage. This is a result of two factors: the large number of systems which are stored permanently on disk; and the unpredictability of the computer. The first factor means that there are fewer available disk segments for user programs, which necessitates a priority ranking of user programs stored on the system, and periodic purging of old and low priority files.

The second factor must be considered as one of the hazards of using a computer. The disk, the fastest medium of external storage, is also the most vulnerable. Information on it can be lost if the system crashes. Therefore, if you want to save a program for later use, make a paper-tape copy of it.

FOCARL LIBRARY COMMANDS

The LIBRARY commands in FOCARL are used to reference disk files. Each disk file has a name and an extension. The file name is from one to six alphanumeric characters, the first of which must be alphabetic. FOCARL appends an extension of .FRL to files it creates. These are also the only files that it can reference.

The syntax of a LIBRARY command is:

L sub-command file-name

The six available sub-commands are:

CALL To load a program stored on disk into

the users core

DELETE To delete a program stored on disk
SAVE To store a copy of a user's program

on the disk

XTEND To chain from a program in core to a

program stored on disk

OUTPUT To create a data file on disk

INPUT To reference a data file to read the

information back into core

LIBRARY OUTPUT and INPUT will be explained more fully in the section data file in FOCARL.

LIBRARY SAVE: L S NAME

This command stores a copy of the program presently in the user's core in a disk file with the given name. This file is stored in the library of the account that the user is logged in under.

LIBRARY CALL: L C NAME

The CALL command is used to copy a program stored on disk into the user's core. FOCARL looks for a file of the given name, and checks that it is a FOCARL program file and not a data file. It then erases the program that is in core and reads in the program from disk.

LIBRARY CALL can also be used to call a program from the systems library. An asterisk following the name is used to indicate a call for a file from the system library, instead of the user's library.

LIBRARY DELETE: L D NAME

C FOCARL- 10

This command deletes the disk file with the given name from the library of the account the user is logged in under.

LIBRARY EXTEND: L X NAME LINE-NUMBER

The LIBRARY XTEND command is used in "chaining" from one FOCARL program to another. Because of the limited core available for program and variable storage, the user may be unable to execute his entire program as one if it uses a large number of variables. The LIBRARY XTEND command, included as a line in the program, preforms three operations when it is executed:

- 1. It retains the symbol table, but erases the program core.
- It loads the program specified from disk into the user's core.
- 3. It begins execution of the program at the line number specified. If no number is specified, execution starts at the lowest line number of the program.

The XTEND command can also be used to reference system library programs by following the program name with an *. A space must separate the program name (or * if system library program) and the line number if specified.

The following examples illustrate the use of the LIBRARY commands:

```
* L C PART1

* WRITE
C FOCARL- 1Ø

Ø1.1Ø TYPE "WE WILL BEGIN HERE!",!
Ø1.2Ø ASK ?A B C?,!
Ø1.3Ø TYPE "NOW WE WILL TRY TO XTEND!",!,!,!
Ø1.4Ø L X PART2
Ø1.5Ø TYPE "HERE WE COME BACK TO THE FIRST PART TO QUIT!",!
Ø1.6Ø QUIT

* L C PART2

* WRITE
```

Ø3.2Ø TYPE "THIS IS THE PROGRAM WE XTEND TO.",!
Ø3.3Ø TYPE "SECOND LINE OF XTEND PROGRAM!",!
Ø3.4Ø TYPE "A IS ",A," B IS ",B," C IS ",C,!
Ø3.5Ø L X PART 1 1.5
* L C PART1
* GO
WE WILL BEGIN HERE!
A :-2.5 B :13.74 C :.Ø89
NOW WE WILL TRY TO XTEND!

THIS IS THE PROGRAM WE XTEND TO.
SECOND LINE OF XTEND PROGRAM!
A IS -2.50000 B IS 13.7400 C IS 0.08900
HERE WE COME BACK TO THE FIRST PART TO OUIT!

LINE PRINTER OUTPUT

The function FLPT is used to direct output to the line printer and then to resume TTY printing. Any call for the function FLPT(A) with A=Ø will direct subsequent output to the line printer. Any call for the function FLPT(A), A different from zero results in TTY output. Any character typed on the TTY keyboard, even if line printer output is specified, will be echoed on the TTY.

An innocuous way of changing output device is by using one of the following commands: (X is a dummy variable)

SET X=FLPT(X) Sets X=X

IF (FLPT(X)) If no arguments follow, execution continues with the next command

TTY output is assumed when the user enters FOCARL.

USING THE LOW-SPEED TAPE READER AND PUNCH ON THE TTY

To punch a tape on the TTY punch:

- 1. Type W or WRITE.
- 2. Turn the tape punch on.
- 3. Hit the here is key once or twice to punch out some leader.
- 4. Hit the return key.
- 5. When the program is done punching, hit here is once or twice to punch trailer tape.
- 6. Turn the punch off and label the tape.

To read a tape on the TTY reader:

- 1. Type ♦R to unduplex the TTY.
- 2. Put the tape in the tape reader.
- 3. Move the reader switch to start.
- 4. When the tape is done reading, move the reader switch to free and remove the tape.
- 5. Type †T to resume normal printing (duplex).

DISK DATA FILES IN FOCARL

In addition to storing FOCARL programs on the disk, it is also possible, using FOCARL, to create and access data files on disk. The files are set up as one-dimensional arrays of numbers, with the numbers referenced via their location in the file.

INITIALIZING A DATA FILE--OUTPUT

The LIBRARY OUTPUT command is used to create and initialize a data file. The format of the command is simply: L O file-name. This command will be executed correctly if two conditions exist:

- There are disk segments available for a new file.
- 2. There is not a file by that name already stored in the library of the account the user is logged in under.

If either of these conditions fails, an error occurs and an error message will be typed.

WRITING ON AN OUTPUT FILE

Once an output file has been initialized, it can be used for storing data. Data is output to the disk by using the %L format option of the TYPE command. The format of the commands are:

TYPE %L alpha , beta , theta

Where alpha specifies the starting location (on disk) for storage of the output values beta, theta, and so on

TYPE %L, beta , gamma

Where beta is written in the location following the last number output to disk, gamma is stored in the location following beta, and so on.

INITIALIZING A DATA FILE--INPUT

Before you may recall information from a data file, it must be initialized as an input file. The LIBRARY INPUT command (L I name) is used to initialize an existing data file as an input file. If the file does not exist, FOCARL will return an error message.

READING FROM AN INPUT FILE

Once data has been initialized, information may be read from it by specifying the %L format in an ASK statement. Analogous to the TYPE commands:

ASK %L, alpha, beta, theta, ...

Will read values starting from location alpha of the disk file and store them as the variables beta , theta , and so on.

ASK %L, gamma, delta

Will read in the next two sequential locations and store their values as the variables gamma and delta, respectively.

USING THE SAME DATA FILE FOR INPUT AND OUTPUT SIMULTANEOUSLY

Certain problems may occur when the user has the same file initialized as both an input and an output file. Problems result if the output file must be increased in length while it is initialized as an input file. Disk segments have a fixed length. Each segment used as a data file in FOCARL will store 84 numbers. When the 85th number is output to disk, FOCARL must append another segment to the data file. This cannot be done if the file is also initialized (open) as an input file. There are two ways of circumventing this problem:

- Never output any data to disk after the file has been initialized as an input file.
- 2. Always "close" the input file before you do a disk output command. Any call for the function FCLS will "uninitialize" the input file. A reasonably innocuous method of calling FCLS is by using one of the following commands: (X is a dummy variable)

COMMAND EFFECT

T %A, FCLS(X) Types a non-printing character

S X=FCLS(X) Sets X=X

NOTES ON USING DATA FILES AND %L FORMAT

Specifying %L in either a TYPE or an ASK statement sets up both disk input and output.

Once %L is specified, all I/O is assumed to be disk until another format is used.

When specifying a location in a disk I/O statement, no comma separates %L and its first argument.

ERROR 31.: 6

This error is generated if FOCARL cannot extend the output file another segment. This results if the file is also initialized (open) as an input file, or if there are no free disk segments.

When inputting from and outputting to the same data file, each command keeps its own pointer as to the location it will next reference on disk.

The next part should help clarify some of the ambiguities that have arisen in the preceding explanation.

*
*WRITE
C FOCARL-10

Ø1.Ø5 TYPE "THIS IS A DEMONSTRATION PROGRAM, ILLUSTRATING THE",!

Ø1.07 TYPE "USE OF DISK DATA FILES.",!,!

Ø1.Ø9 TYPE "THE PROGRAM STORES THE SQUARES OF THE FIRST 1ØØ",!

Ø1.11 TYPE "POSITIVE INTEGERS ON DISK AND RETRIEVES THEM AS"

Ø1.12 TYPE "REQUESTED BY THE USER,",!,!

Ø1.15 L O T1; C THIS CREATES DATA FILE AND INITIALIZES IT FOR OUTPUT

Ø1.20 FOR I=1,100; TYPE %L, L 12; C WRITES FIRST 100 SQUARES ON DATA FILE

Ø1.3Ø ASK %, "NUMBER YOU WANT THE SQUARE OF", QU,!

Ø1.32 IF (QU) 1.97,1.97,1.33

Ø1.33 IF (QU-FITR(QU)) 1.95,1.35,1.95

Ø1.35 IF (QU-1Ø1) 1.37; TYPE "VALUE TOO LARGE",!; GOTO 1.3

Ø1.37 L I T1; C THIS INITIALIZES FILE FOR INPUT

Ø1.38 ASK %L QU, AN; C THIS GETS THE VALUE FROM DISK

Ø1.40 TYPE %5, "THE SQUARE OF", QU, "IS ", AN,!

Ø1.43 ASK "AGAIN? NO=.5 ", AG, !; IF (AG-.5) 1.3,1.9,1.3

Ø1.9Ø L D T1;C DELETES DATA FILE, SINCE IT IS TRIVIAL
Ø1.92 QUIT;C END OF PROGRAM RUN
Ø1.95 TYPE "VALUE IS NOT AN INTEGER. TRY AGAIN",!;GOTO 1.3
Ø1.97 TYPE "PROGRAM WORKS FOR POSITIVE INTEGERS <=1ØØ",1;GOTO 1.3
*
*GO
THIS IS A DEMONSTRATION PROGRAM, ILLUSTRATING THE
USE OF DISK DATA FILES.</pre>

THE PROGRAM STORES THE SQUARES OF THE FIRST 100
POSITIVE INTEGERS ON DISK AND RETRIEVES THEM AS REQUESTED BY THE USER.

NUMBER YOU WANT THE SQUARE OF:33 THE SQUARE OF 33 IS 10/89 AGAIN? NO=.5:0 NUMBER YOU WANT THE SQUARE OF:99 THE SQUARE OF 99 IS 9801 AGAIN? NO=.5:

NUMBER YOU WANT THE SQUARE OF: Ø
PROGRAM WORKS FOR POSITIVE INTEGERS <=1ØØ
NUMBER YOU WANT THE SQUARE OF: 33.4
VALUE IS NOT AN INTEGER. TRY AGAIN!
NUMBER YOU WANT THE SQUARE OF: 2Ø1
VALUE TOO LARGE
NUMBER YOU WANT THE SQUARE OF: 97
THE SQUARE OF 97 IS 94Ø9
AGAIN? NO=.5:.5

ASCII FORMAT INPUT/OUTPUT

Each character on the TTY keyboard is interpreted and stored by the computer as a number. The code which the computer uses is called ASCII, the United States of America Standard Code for Information Interchange. This code, or a more compact form of it, is used in all inter-machine communications. The %A formatting option in FOCARL allows the user to work with both the numeric and character representations of each character. For a list of the character set and the numeric codes corresponding to each character, see appendix A.

%A INPUT

Specifying ASCII format in an ASK statement results in two immediately noticeable changes: the ":" is not typed, as it is under a numeric format, and only one character is accepted for each variable asked. When a character, for example, A, is typed in response to an ASK statement, the numeric value of A, 193, is stored as the variable.

%A OUTPUT

When %A is specified in a TYPE statement, the computer will type out all values as their character counterparts, if they are defined.

Note

Just as with any numeric format, all input and output will be handled under this format until another format is specified.

EVALUATION OF ALPHABETIC RESPONSES TO INPUT

FOCARL will accept and evaluate alphabetic character strings typed in response to a numerically formatted ASK statement. The values ascribed to the letters (except E) correspond to their position in the alphabet, i.e., A=1, B=2, Z=26, M=13, etc. E denotes exponent, or power of 10. When E is the first character of a string typed in response to an input request, the value of the string is zero. If E is the last character of the string, a delimiter other than a space must be used and the power of 10 is assumed to be zero. If the number corresponding to a letter is greater than 9, the ten's digit will be carried if that character is part of a string.

The following example should help to clarify the above explanations. A complete list of the numeric values of the characters is given in appendix A.

```
Ø1.1Ø ASK A; TYPE A,!; GOTO 1.1
*GO
:A 1.000
:B 2.ØØØ
:E , Ø.ØØØ
:H 8.ØØØ
:J 10.00
:L 12.00
:M 13.00
:HJ 9Ø.ØØ
:ABC 123.Ø
:X 24.ØØ
:Y 25.00
:Z 26.ØØ
:XY 265.Ø
:XYZ 2676
:ABEA 120.0
:ABEO 12.00
:CEC 3ØØØ
:ZEA 26Ø.Ø
?ØØ.ØØ @ Ø1.1Ø
                 (†C typed to stop program execution)
```

EXECUTION NOTES

FOR J=A,B,C; command string

The FOR command will perform one execution of the command string before checking the range of values for J. The FOR command can only be used with a positive increment. If no increment is specified, it is assumed to be 1. FOR commands may be nested within one another.

IF (arithmetic expression) N,Z,P (N,Z, and P are line numbers)

IF command branches to N if the value of the arithmetic expression is negative; to Z if the value is zero; and to P if the value is positive.

Less than three arguments can be given following any IF command.

2.31 IF (A-6) 2.39; TYPE "A=>6",!

If A<6 [(A-6)<0], branches to line 2.39; otherwise, the TYPE command will be executed.

4.3Ø IF (A-62); TYPE "HOWDY"

TYPE command will always be executed, since no branch is indicated. The Ø alphabetic string option of evaluating alphabetic responses to an input request may be used in an IF statement. See line 4.1 of the program in appendix C.

DO

DO N (N is a group number)

This command will begin execution of group N at the lowest line number and will proceed through the group sequentially, with the following exceptions:

- A branch statement to a non-sequential statement within the group will be executed normally, altering sequential execution.
- A DO N.AB to a line within a group N will be executed correctly.
- 3. A branch command to a line outside the group will result in FOCARL executing only that line, before returning to the next line in group N and continuing execution there. However, if the line branched to contains a DO or a GOTO command, that command will be executed before FOCARL return to group N. This "chaining"

effect can continue to multiple levels. The DO will execute only one line when a branch command directs it outside the group specified. But that line will be executed completely before returning to the DO group, group N.

DO L.AB (L.AB is a line number)

This command causes the line L.AB to be executed. If line L.AB contains a DO statement, this will be executed completely before returning to line L.AB.

If line L.AB contains a branch statement, only the line specified by the branch command will be executed. If this line contains a DO or branch command, it also will be executed before control begins to line L.AB.

The DO command can also be used to do recursion by including a DO N command internal to group N. For example, the program line 2.21 IF (BC*4-AL) 3.1,2.27;DO 2

MODIFY AB.CD

A MODIFY command may be programmed indirectly, but program execution halts after this command is executed. Executing a MODIFY command erases the symbol table.

RETURN

A RETURN command, if encountered while not executing a DO command, will function as a QUIT command, terminating program execution.

COMMENT

Once a COMMENT command is encountered, the remainder of the line is assumed part of the comment. Thus, COMMENT should be the last command to appear in a line, since commands following it on the same line won't be executed.

LIBRARY EXTEND

When a LIBRARY XTEND command is executed, it loads the program extended to over the program previously stored in core. If this program is longer than the first, it will overlay variables before it finishes loading the entire program and the symbol table will be overwritten and destroyed.

LOGOUT

If the user accidentally executes a LOGOUT command, he can reenter FOCARL by simply typing ST to monitor's ".". His program and symbol table will still be intact.

ERROR MESSAGES

When FOCARL detects a syntax error in programming or is unable to execute a command, it will generate an error message. The error message has the form:

? error number @ line number

If the error was detected in a direct command or while typingin a program line, no line number is given, just "? error number ". A list of error diagnostics is given in appendix B.

TRACING

FOCARL provides an option to allow the user to trace the execution of his program. When a ? is encountered, except in a comment line or within a quote in an input/output statement, FOCARL enables the trace feature. Trace remains enabled until another ? is encountered. Each "odd-numbered" ? enables trace, while each "even-numbered" occurrence disables trace.

When enabled, trace starts typing every character that FOCARL encounters as it executes the program.

The ? will be recognized wherever it appears in a command statement. It can be imbedded in a line number or a variable name, or before, following, or even within a command, etc.

An example of a trace program is given in appendix C.

COMMAND SUMMARY

| COMMAND | OPTION | SYNTAX | EFFECT |
|---------|-----------------------------|--|---|
| TYPE | % B % A . BC % A % L ! # \$ | TYPE %,A TYPE %4,A T %6.Ø4,A T %A,L T %L,A T ! T # T \$ T "HI" T ?A? A %A,B A %L,C | Numeric output in exponential format Numeric output in integer format Output in decimal format ASCII character output Output to a disk data file Generates carriage return/line feed Generates a carriage return Generates a printout of the symbol table under the existing format Verbatim printing of the character string Types out variable name then value Accepts only one character for B stores ASCII value of character Reads value from disk data file |
| | # | A ! A # | Generates CR/LF combination Generates a carriage return (CR) |
| | ? | A "A IS" ,A A ?S? | Verbatim printing of character strings |
| | \$ | A \$ | Types out variable name, asks for value Generates listing of symbol table |
| SET | | S A=3.4/7.1 | |
| GO | | GO | Starts execution at lowest line number |
| GOTO | | GOTO 1.03 | Begins (continues) execution at line 1.03 |
| DO | | DO DO 3 DO 1.23 | Begins execution at lowest line number Executes group three, beginning at lowest line number of group three Executes line 1.23 |
| WRITE | | WRITE ALL WRITE 3 WRITE 2.72 | Types out list of the entire program Lists group three Lists line 2.72 |
| ERASE | | ERASE 3 ERASE 2.51 ERASE ALL | |
| QUIT | | QUIT | Stops program execution |
| RETURN | | RETURN | Denotes end of a DO subroutine |
| COMMENT | | C THIS IS | Denotes a non-executed line of text |

| COMMAND | SYNTAX | ACTION |
|----------------|---|---|
| FOR | FOR I=1,5,1;T "*"I FOR J=1,6;T %2,J | Performs iteration of command String following ";" |
| IF | IF (A-6) 1.3,1.5,1.3 .N.,.Z.,.P. IF (B-13)4.2,5.1;S B=A | Conditional branch statement: (A-6)<0 branches to 1.3 (A-6)=0 branches to 1.5 (A-6)>0 branches to 1.3 |
| MODIFY | MODIFY 1.37 | Used to modify a program line |
| LIBRARY CALL | | Loads a program from disk into core |
| LIBRARY SAVE | L C DEMO* L S TEST | Stores the program in core on disk |
| LIBRARY DELETE | L D TEST | Deletes the program TEST from disk |
| LIBRARY EXTEND | L X TEMP | Chains from program in core to the program called TEMP on disk |
| | L X TEMP 1.7 L X TRIAL2* 2.25 | one program carred there on arsk |
| LIBRARY OUTPUT | L O TRY1 | Creates a data file on disk and initializes it for output |
| LIBRARY INPUT | L I TRY1 | Initializes data file for input |
| LOGOUT | L OR LOGOUT | Exits from FOCARL, returns user to TSS/8 monitor |

```
Ø1.10 T "THIS PROGRAM GENERATES A TABLE OF THE NUMERIC CODES OF" ,!
Ø1.20 T "ALPHABETIC CHARACTERS UNDER THE POSSIBLE FORMATS" ,!
                                        NUMERIC" ,!
Ø1.30 T "CHARACTER
                            ASCII CODE
                      8A
Ø1.4Ø T "
                       "DECIMAL" (OCTAL)
                                             (DECIMAL)",!
\emptyset1.5\emptyset F I=161,175;D 2;D 3.\emptyset5;T!
Ø1.52 F I=176,185;S CO=-176;D 2;D 3.Ø5;D 3.Ø6
\emptyset1.54 F I=186,192;D 2;D 3.\emptyset5;T!
Ø1.56 F I=193,218;S CO=-192;D 2;D 3.Ø5;D 3.Ø6
Ø1.58 F I=219,223;D 2;D 3.Ø5;T!
Ø1.6Ø G 2.25
Ø2.1Ø C THIS DECODES DECIMAL TO OCTAL BASE 1Ø TO BASE 8
Ø2.11 I (I-197) 2.15,2.13,2.15
Ø2.13 S CO=−197;C THIS SETS A VALUE OF Ø FOR E UNDER NUMERIC
\emptyset2.15 S HI=FITR (I/64);S H=HI+176
\emptyset 2.17 \text{ S MD-FITR}((I-(HI*64))/8); \text{ S M=MD+176}
\emptyset 2.19 \text{ S LO}=I-(HI*64+MD*8); S L=LD+176
Ø2.21 RETURN
Ø2.25 T "SPACE
                          ";S I=16\emptyset;D 2;D 3.09
Ø2.27 T "LINE FEED
                          ";S I=138;D 2;D 3.Ø9
Ø2.29 T "RETURN
                          ";S I=141;D 2;D 3.Ø9
                          ";S I=135;D 2;D 3.Ø9
Ø2.31 T "BELL
Ø2.33 T "RUBOUT
                          ";S I=255;D 2;D 3.Ø9
Ø2.34 T "LEADER",!,"TRAILER";S I=128;D 2;D 3.Ø9
Ø2.99Ø QUIT
               ",%A,I,"
Ø3.Ø5 T "
                             ",%3,I,"
                                           ",%A,H,M,L
                        ",%2,I+CO,!
Ø3.Ø6 T "
                        ",%3,I,
Ø3.Ø9
                                           A, %A, H, M, L,
*GO
THIS PROGRAM GENERATES A PROGRAM OF THE NUMERIC CODES OF
ALPHABETIC CHARACTERS UNDER THE POSSIBLE FORMATS
```

| ! 161 241 " 162 242 # 163 243 \$ 164 244 \$ 165 245 & 166 246 ' 167 247 (168 250) 169 251 * 170 252 + 171 253 | CHARACTER | %A | ASCII CODE | NUMERIC |
|--|-----------|-----------|------------|-----------|
| " 162 242 # 163 243 \$ 164 244 \$ 165 245 & 166 246 ' 167 247 (168 250) 169 251 * 170 252 + 171 253 | ı | (DECIMAL) | (OCTAL) | (DECIMAL) |
| # 163 243 \$ 164 244 \$ 165 245 \$ 166 246 ' 167 247 (168 250) 169 251 * 170 252 + 171 253 | | | | |
| % 165 245 & 166 246 ' 167 247 (168 250) 169 251 * 170 252 + 171 253 | # | | | |
| % 165 245 & 166 246 ' 167 247 (168 250) 169 251 * 170 252 + 171 253 | π S | | | |
| \$ 166 246 167 247 (168 250) 169 251 * 170 252 + 171 253 | | | | |
| ' 167 247 (168 250) 169 251 * 170 252 + 171 253 | | | | |
| (168 250) 169 251 * 170 252 + 171 253 | | | | |
|) 169 251 * 170 252 + 171 253 | (| | | |
| * 170 252 + 171 253 | | | | |
| + 171 253 | | | | |
| | + | 171 | 253 | |
| | , | 172 | 254 | |
| - 173 255 | - | | | |
| • 174 256 | | | 256 | |
| / 175 257 | | | 257 | |
| 0 176 260 0 | | | 260 | 0 |
| $\frac{1}{2}$ 177 271 1 | 1 | | | 1 |
| 2 178 262 2 3 179 263 3 | 2 | | | 2 |
| 1 177 271 1 2 178 262 2 3 179 263 3 4 180 264 | 3 | | | 3 |
| | | | | |
| 5 181 265 5 6 182 266 6 | 5 | | | |
| 7 100 | 7 | | | |
| 201 | | | | |
| 2 10- | | | | |
| 9 185 271 9 : 186 272 | | | | 9 |
| ; 187 273 | | | | |
| 188 274 | | | | |
| = 189 275 | <u> </u> | | | |
| > 190 276 | > | | | |
| ? 191 277 @ 192 300 | ? | | | |
| | | 192 | | |
| A 193 301 1 | | | | 1 |
| B 194 302 2 | | 194 | | 2 |
| C 195 303 3 | | | 303 | |
| D 196 304 4 | | | | 4 |
| E 197 305 0 | | | | 0 |
| F 198 306 6 | | | | 6 |
| G 199 307 7 H 200 310 8 | | | | 7 |
| H 200 310 8 I 201 311 | | | | 8 |
| 7 000 | | | | |
| T 202 | | | | |
| 7 204 | | | | |
| W 20.5 | | | | |
| N 206 | | | | |
| 0 207 | | | | |
| P 208 320 16 | | | | |
| Q 209 321 16 | | | 321 | |

| R S T U V W X Y Z [\ | 210 211 212 213 214 215 216 217 218 219 220 221 222 223 | 322 323 324 325 326 327 330 331 332 333 334 335 336 337 | 18 19 20 21 22 23 24 25 26 |
|---|--|--|--|
| SPACE | 160 | 240 | |
| LINE FEED RETURN BELL RUBOUT | 138 141 135 255 | 212 215 207 377 | |
| LEADER TRAILER | 128 | 200 | |

ERROR DIAGONISTICS FOR FOCARL - VERSION 10

| CODE | MEANING |
|--|---|
| ?00.00 | Interrupt via control-C |
| ?01.40 ?01.78 ?01.96 ?01.:5 ?01.;4 | Illegal step or line number used Group number too large Double periods found in a line number Line number is too large Group zero is an illegal line number |
| ?02.32 ?02.52 ?02.79 | Nonexistent group referenced by DO Nonexistent line referenced by DO Storage was filled by push-down list |
| ?03.05 ?03.28 | Nonexistent line used after GOTO or IF Illegal command used |
| ?04.39 ?04.52 ?04.60 | Left of "=" in error in FOR or SET Excess right terminators encountered in SET Illegal terminator in FOR or SET |
| ?05.48 ?06.54 ?06.06 | Bad argument to MODIFY Storage is filled by variables Illegal use of function or number |
| ?07.22 ?07.38 ?07.:9 ?07.;6 | Operator is missing in expression, or double 'E' No operator used before parenthesis No argument given after function call Illegal function name, double operations, or excess parenthesis |
| ?08.47 ?09.11 ?10.:5 ?11.39 ?11.73 ?17.35 | Parentheses do not match Bad argument in ERASE Storage is filled by text, or line too long Illegal LIBRARY command Disc error Initial dialogue error |
| ?20.42 ?20.44 ?23.36 ?24.04 ?24.30 ?24.:1 ?24. 4 ?25.15 ?25.64 ?25.97 | Logarithm of zero requested Logarithm of negative number requested Literal number is too large Library program is too large LIBRARY EXTEND error-too many variables Cannot save program, disc full Cannot delete disc file Device busy Illegal library name File not found for CALL or DELETE, file already in existence for SAVE, or disc full |

| ?25. 3 ?26.99 ?28.73 ?29.17 | Not FOCARL file (.FRL) Exponent is too large or negative Division by zero requested Cannot assign the line printer |
|--|--|
| ?30.72 ?30.05 ?31.10 ?31.12 ?31.:6 ?31. 7 ?31.55 ?31.76 | Illegal high speed command Imaginary square root requested High speed reader or punch is hung Disc errorcannot recover Attempt to extend data file failed Unavailable function used Input file not initialized Output file not initialized |

```
Ø1.10 C THIS PROGRAM WILL FIND THE ROOTS OF A QUADRATIC EQUATION
Ø1.15 T "TYPE IN THE VALUES OF A,B,AND C, WHERE A IS THE",!
Ø1.16 T "COEFFICIENT OF X € 2, B THE COEFFICIENT OF X, AND C",!
Ø1.17 T "IS A CONSTANT.",!,!
Ø1.20 A ?A B C ?,!,!
Ø1.23 S RD=B+2-4*A*C
Ø1.25 IF (RD) 3.05,1.3,2.05
Ø1.30 T "EQUATION HAS ONLY ONE ROOT: ",%6.04,-(B/2),!,!
Ø1.32 G 4.1
02.05 T "EQUATION HAS TWO REAL ROOTS:
\emptyset 2.1\emptyset S SR=FSQT(RD)
\emptyset 2.20 \text{ T} - (B-SR)/2," ",-(B+SR)/2,!,!
02.30 G 4.1
03.05 T "EQUATION HAS TWO IMAGINARY ROOTS:",!
\emptyset3.\emptyset7 S RA=FSQT(-RD)
\emptyset 3.10 \text{ T} -B/2," + ",RA," * I "
Ø3.2Ø S RA=-RA; T " ",!;D 3.1;T !,!
Ø3.30 G 4.1
Ø4.10 A "AGAIN Y OR N "L,!,!; IF (L-ØN) 1.2,4.2,1.2
04.20 QUIT
*
TYPE IN THE VALUES OF A,B, AND C, WHERE A IS
COEFFICIENT OF X 2, B THE COEFFICIENT OF X, AND C
IS A CONSTANT.
A: B:2 C:3
EQUATION HAS TWO IMAGINARY ROOTS:
-1.00000 + 2.82843 * I
-1.000000 + -2.82843 * I
AGAIN Y OR N :Y
A :1 B :2 C :1
EQUATION HAS ONLY ONE ROOT: -1.0000
AGAIN Y OR N :Y
    :37 B :75.9 C :-236
EQUATION HAS TWO REAL ROOTS: 62.9073 -138.807
AGAIN Y OR N :
*GO?
```

```
C C T "TYPE IN THE VALUES OF A,B, AND C, WHERE A IS THE !
T "COEFFICIENT OF X + 2, B THE COEFFICIENT OF X, AND C!
T "IS A CONSTANT.!
, <u>l</u>
A :1 :2 :1 ,!
,!
S RD=B^{\dagger}2-4*A*C
IF (RD) 3.05,1.3,T "EQUATION HAS ONLY ONE ROOT: \$6.04,-(B/2),-1.0000!
, !
G 4.1
A "AGAIN Y OR N ,:Y !
, !
; IF (L-\emptyset N) 1.2,4.2,1.2
A :1 :2 :3 ,!
, !
S RD=B\uparrow2-4*A*C
IF (RD) 3.05,T "EQUATION HAS TWO IMAGINARY ROOTS:!
S RA= FSQT (-RD)
T -B/2,- 1.0000" + RA, 2.8284" * I S RA=-RA; T " !
;D 3.1;T -B/2,- 1.0000" + RA,-2.8284" * I T !
, !
G 4.1
A "AGAIN Y OR N ,:N!
,!
; IF (L-\emptyset N) 1.2,4.2,QUIT
```

TIME SHARE 8 DOCUMENTATION.

Please keep in the same room as the Time - Share System.

Product Code: DEC-T8-AJZA-D

Product Name: TSS/8 Extended

BASIC (EDUSYSTEM 50)

Date created: October 1, 1971

Maintainer: Development

Copyright



1971 by Digital Equipment Corporation

The material in this manual is for information purposes only and is subject to change without notice.

The following are trademarks of Digital Equipment Corporation, Maynard, Massachusetts

DEC

DECtape

Digital

PDP

PS/8

For additional copies, order DEC-T8-AJZA-D from Program Library, Digital Equipment Corporation, Maynard, Massachusetts 01754 Price \$5.00

CONTENTS

| Tatm | PODUGETOR | |
|------|------------------------------------|----------|
| TNT | RODUCTION | 1 |
| 1. | Editing BASIC Statements | 2 |
| 2. | Saving Compiled Programs | 2 |
| 3. | File Protection | 3 |
| 4. | Project-Programmer Numbers | 3 |
| 5. | Restricted Accounts | |
| 6. | Catalog Format | 3 |
| 7. | Strings in BASIC | 4 |
| 7.1 | Reading String Data | 4 |
| 7.2 | Printing Strings | 5 |
| 7.3 | Inputting Strings | 6 |
| 7.4 | Line Input | 6 |
| 7.5 | | 7 |
| 7.6 | Working with Strings | 8 |
| 7.7 | The CHANGE Statement | 9 |
| 7.8 | A Note About CHANGE | 10 |
| , | The CHR\$ Function | 11 |
| 8. | Modification to Legal Array Names | 12 |
| 9. | Modification to DATA Statement | 12 |
| 10. | Program Chaining | 12 |
| 11. | Disk Data Files | 13 |
| 11.1 | File Records | 14 |
| 11.2 | Opening a Disk File | 15 |
| 11.3 | Reading/Writing Disk Files | 15 |
| 11.4 | Closing and Deleting Disk Files | 17 |
| 12. | DECtape Files | 18 |
| 12.1 | DECtape File Records | 18 |
| 12.2 | Opening a DECtape File | 19 |
| 12.3 | Reading and Writing DECtape Files | 20 |
| 12.4 | Closing DECtape Files | |
| 12.5 | Using Data Tapes with PS/8 FORTRAN | 21 21 |
| 13. | Line Printer Output | 22 |
| 14. | Papertape Output | 22 |
| 15. | Truncation Function, FIX(X) | 23 |

| 16. | ON GOTO | 23 |
|----------|------------------------------------|------------|
| 17. | Implied LET | 24 |
| 18. | SLEEP | 24 |
| 19. | Comments | 2 5 |
| 20. | Blank Lines | 25 |
| 21. | More than One Statement on a Line | 25 |
| | | |
| Appendix | A Error Messages | 27 |
| Appendix | B Internal Dala Codes | 30 |
| | | |
| Notes to | System Users About BASIC Version 4 | 33 |

INTRODUCTION

This document is an update to the TSS/8 BASIC-8 manual (Chapter 12 of Programming Languages). It includes all changes to BASIC since its original release. It therefore supersedes all previous update notices.

The following commands are described in the original documentation and hence are not included here.

| LET |
|------------|
| DATA |
| READ |
| PRINT |
| GOTO |
| IFTHEN |
| FOR |
| NEXT |
| DIM |
| END |
| RANDOMI ZE |
| GOSUB |
| RETURN |
| INPUT |
| STOP |
| REM |
| RESTORE |
| |

DEF EDIT COMPILE BYE CATALOG DELETE KEY LIST NEW OLDREP**LACE** RUN SAVE SCRATCH TAPE UNSAVE

1. Editing BASIC Statements

If an entered line is grossly erroneous, it may be corrected by retyping it. Minor errors in already-entered statements may be corrected by using the EDIT command. Type EDIT followed by the line number of the statement to be edited. BASIC responds by typing a left bracket ([). Type a "search character". BASIC types a close bracket, then types out the statement up to the first occurrence of the specified search character. The user may then:

- Type new characters which are inserted at that point in the statement.
- 2. Type one or more back arrows (+) to delete characters to the left.
- 3. Type the ALT MODE key to delete the entire line up to that point (but not the line number).
- Type LINE FEED to finish the edit, keeping the remainder of the line unchanged.
- Type RETURN to finish the edit, deleting the remainder of the line.
- 6. Type CTRL/L to continue to the next occurrence of the search character.
- 7. Type CTRL/G to specify a new search character.

2. Saving Compiled Programs

BASIC compiles the current program each time it is run. If, however, it is a program which will be used frequently, but not changed, it may be stored in its compiled form. To save a compiled program, type:

COMPILE name

The program will be saved on the disk under the specified name. If a file by that name exists, BASIC will type DUPLICATE FILE NAME and will not compile that program.

Once a program has been compiled, it may be called out and run just like an ordinary BASIC source program. It may not, however, be listed, saved, or changed. If an attempt is made to do any of these things, the message EXECUTE ONLY is typed. The compile capability may therefore be used to protect programs from unauthorized access.

Compiled files are distinguished from regular BASIC programs by their file extensions. BASIC source programs have an extension of .BAS. Compiled files have an extension of .BAC. These extensions are typed out along with the file name when a catalog is requested.

3. File Protection

TSS/8 permits a user to specify a protection code for each file. (See TSS/8 TIME SHARING SYSTEM in INTRODUCTION TO PROGRAMMING for a full description of protection codes). The commands which write disk files (SAVE, REPLACE, COMPILE) also permit the user to specify what protection is to be given to a file. This is done by following the file name with the protection code in angle brackets. For example,

SAVE FOO < 1Ø>

will create and save a file named "FOO.BAS" having a protection code of 10. When no protection is specified, a protection of 12 is automatically assumed.

4. Project-Programmer Numbers

In specifying the Account Number for an OLD file, the user may optionally type a Project-Programmer number (giving the Account Number as two 2-digit numbers instead of a single 4-digit number). In this way, the user may RUN files from another user's disk area.

For example, both of the following are acceptable:

OLD PROGRAM NAME -- FOO 13,3

(where 13 is the Project Number and 3 is the Programmer Number), or

OLD PROGRAM NAME -- FOO 1303

(where 1303 is the equivalent account number).

5. Restricted Accounts

As an added system protection, BASIC checks to see if an attempt is being made to run BASIC under Accounts 1 or 2. If so, BASIC prints the error message:

IMPROPER ACCOUNT # ABORT †BS

thus preventing BASIC from interfering with the System Directories or the System Library.

6. Catalog Format

The CATALOG command types file extensions, file size, and file protection, in addition to the file name. For example:

CATALOG

| NAME | | SIZE | PROT |
|--------|-------|------|------|
| DUMP | .SAV | 8 | 16 |
| XYZMP' | r.ASC | 32 | 1ø |
| BACKU | P.BIN | 38 | 12 |
| FACTR | L.BAS | 4 | 12 |
| FACTO | R.BAC | 2 | 12 |
| DATAl | .DAT | 7 | 12 |
| BASØ1 | 7.TMP | 1 | 17 |
| BAS11 | 7.TMP | 1 | 17 |

7. Strings in BASIC

TSS/8 BASIC has the ability to manipulate alphabetic information (or "strings"). A string is a sequence of characters, each of which is one of the printing ASCII characters (given in the table in Appendix B). In TSS/8 BASIC, strings consist of six or fewer characters; strings of more than six characters are truncated on input to exactly six characters.

Variables can be introduced for simple strings, string vectors, and string matrices. A string variable is denoted by following the variable name with the dollar sign character (\$). For example:

| Al\$ | A simple string of up to 6 characters |
|-----------|---|
| V\$(7) | The seventh string in the vector V\$(n) |
| M\$ (1,1) | An element of a string matrix M\$(n,m) |

As usual, when string arrays or matrices are used a DIM statement is required. For example:

10 DIM V(10), M\$(5,5)

reserves eleven strings for the vector V\$ and 36 strings for the matrix M\$.

7.1 Reading String Data

Strings of characters may be read into string variables from DATA statements. Each string data element is a string of one to six characters enclosed in quotation marks. The quotation marks are, of course, not part of the actual string. For example:

1Ø READ A\$, B\$, C\$
2ØØ DATA "JONES", "SMITH", "HOWE"

The string "JONES" is read into A\$, "SMITH" into B\$, and "HOWE" into C\$. If the string contains more than six characters, the excess characters are ignored.

1Ø READ A\$

2Ø PRINT A\$

3Ø DATA "TIME-SHARING"

4Ø END

causes only

TIME-S

to be printed.

String and numeric elements may be intermixed in DATA statements. A READ operation always fetches the next element of the appropriate type. In the following example:

1Ø READ A, A\$, B 2Ø DATA "YES", 2.5, "NO", 1

2.5 is read into A, "YES" into A\$, and 1 into B.

The standard RESTORE statement resets the data pointers for both string and numeric elements. Two special forms of the RESTORE command, RESTORE* and RESTORE, may be used to reset just the numeric and string data list pointers, respectively.

RESTORE* Resets the numeric DATA list RESTORE\$ Resets the string DATA list

RESTORE Resets both numeric and string data lists

1Ø READ A, A\$, B

2Ø DATA "YES", 2.5, "NO", 1

3Ø PRINT A, A\$, B

4Ø RESTORE*

50 READ A, A\$, B

60 PRINT A, A\$, B

7Ø END

would print:

2.5 YES 1 2.5 NO 1

If line 40 were changed to RESTORE, this program would print:

2.5 YES 1 2.5 YES 1

since the numeric as well as string data lists were reset.

7.2 Printing Strings

The regular BASIC PRINT statement may be used to print out string information. If the semi-colon character is used to separate string variables in a PRINT command, the strings are printed with no intervening spaces. For example, the program:

1Ø READ A\$, B\$, C\$
2Ø PRINT C\$; B\$; A\$
3Ø DATA "ING", "SHAR", "TIME-"
4Ø END

causes the following to be typed:

TIME-SHARING

7.3 Inputting Strings

String information may be entered into a BASIC program by means of the INPUT command. Strings typed at the keyboard may contain any of the standard Teletype characters except back-arrow (+) and quotation mark. Back-arrow, as always, is used to delete the last character typed.

¹Teletype is a trademark of the Teletype Corporation.

Commas are used as terminators just as with numeric input. If a string contains a comma the whole string must be enclosed in quotation marks. The following program demonstrates string input:

1Ø INPUT A\$, B\$, C\$
2Ø PRINT C\$, B\$, A\$
3Ø END
RUN

? JONES, SMITH, HOWE HOWE SMITH

JONES

READY

Strings and numeric information may be combined in the same INPUT statement as in the following example. Note that if an input string contains more than six characters, only the first six are retained.

1Ø INPUT A, A\$, B\$
2Ø PRINT A\$, B\$, A
3Ø END
RUN
? Ø1754, MAYNARD, MASS.
MAYNAR MASS.

1754

The numeric variable A is set to 1754, the string "MAYNAR" is put in the string variable A\$, and the string "MASS." is put into the string variable B\$.

7.4 Line Input

Strings of more than six characters may be entered by means of the LINPUT (line input) command. A LINPUT statement is followed by one or more string variables. For example:

100 LINPUT A\$(1), A\$(2), A\$(3), A\$(4), A\$(5)

The first six characters to be typed are stored in the first string variable, the next six in the second, and so on until the line of input is terminated by a carriage return. Commas and quotes are treated as ordinary characters and hence are stored in the string variables. For example, if the following line were typed in response to the above LINPUT command:

? MAYNARD, MASS. Ø1754

then the values of the string variables would be:

A\$(1) = "MAYNAR" A\$(2) = "D, MAS" A\$(3) = "S. Ø17" A\$(4) = "54" A\$(5) = ""1

In this example, the maximum number of characters which could be typed would be 30. Any additional characters would be ignored. In and cases, the maximum number of characters which may be typed to TSS/8 BASIC is 50. If a longer line is typed, the message LINE TOO LONG is typed. The line must be re-entered.

It is possible to mix numeric and string variables in a LINPUT statement, but it is not recommended. As an illustration of how this might be done, consider the example given earlier:

1Ø LINPUT A, A\$, B\$

where the user might type:

? Ø1754, MAYNARD, MA

This still sets the numeric variable A to 1754 (when used in LINPUT statements, numeric input remains unchanged). However, the string variable A\$ would now be "MAYNAR", and the string variable B\$ would be "D, MA".

NOTE

When inputting strings with LINPUT, the error messages: "MORE?" and "TOO MUCH INPUT, EXCESS IGNORED" cannot occur.

7.5 Working with Strings

Strings may be used in both LET and IF statements. For example:

1Ø LET Y\$= "YES"

 $1\emptyset$ IF Z\$= "NO" THEN $1\emptyset\emptyset$

¹Strings may also consist of zero characters. Such a string is empty, or "null". If printed, it causes nothing to be output. The null string is usually represented by a pair of quotes with nothing in between (""). The null string should not be confused with a string of one or more spaces.

The first statement stores the string "YES" in the string variable Y\$. The second branches to statement 100 if Z\$ contains the string "NO". For two strings to be equal, they must contain the same characters in the same order and be the same length. In particular, trailing blanks are significant since they change the length of the string. "YES" is not equal to "YES".

The relation operators < and > may also be used with string variables. When used with strings, these relations mean "earlier in alphabetic order" or "later in alphabetic order", and they may be used to alphabetize a list of strings. The relationals >=, <=, <> may also be used in a similar manner. The arithmetic operations (+, -, *, /, +) are not defined for strings. Thus, statements such as LET A\$ = 3*5 and LET C\$ = A\$+B\$ have no meaning, and should never be used in a BASIC program. They will not, however, cause a diagnostic to be printed, and the results of such operations are undefined.

7.6 The CHANGE Statement

The BASIC command CHANGE may be used to access and alter individual characters within a string. Every string character has a numeric code (see Appendix B), a number which is used to stand for that particular character. The CHANGE statement converts a string into an array of numbers, or vice versa. The CHANGE statement has the form:

100 CHANGE A TO A\$

or

100 CHANGE A\$ TO A

where A\$ is any string variable (or an element of a subscripted string variable) and A is an array variable with at least six elements. Any array variables used in CHANGE statements must have appeared in a DIM statement with a dimension of at least six.

The following program illustrates the use of the CHANGE statement. In this example, CHANGE is used to change a string variable into an array of numbers.

```
1Ø DIM A(6)
2Ø READ A$
3Ø CHANGE A$ TO A
4Ø PRINT A(Ø); A(1); A(2); A(3); A(4); A(5); A(6)
5Ø DATA "ABCD"
6Ø END
RUN
```

The CHANGE statement takes each character of the string and stores its corresponding numeric code in elements one to six of the array. Remaining array elements are set to zero. The length of the string (\emptyset -6 characters) is then stored in the zero element of the array. In the example above, the character codes for A, B, C, and D are stored in A(1) to A(4). A(5) and A(6) are set to zero. The number 4 is stored in A(\emptyset) since the string A\$ is of length 4.

CHANGE may also be used to change an array of numeric codes into a character string. The following program illustrates this use of the CHANGE statement.

```
10 DIM A(6)
20 FOR I=0 TO 5
30 READ A(I)
40 NEXT I
50 CHANGE A TO A$
60 PRINT A$
70 DATA 5, 65, 66, 67, 68, 69
RUN
```

ABCDE

The length of the resulting string is determined by the zero element of the array. In the above example, the string is of length five. The elements of the array, starting at subscript are assumed to be numeric character codes (32 to 94). These are converted to characters and are stored in the string. If any codes are encountered which are not valid character codes, or if an invalid string length is given, the message BAD VALUE IN CHANGE STATEMENT AT LINE n is typed out, and execution is stopped.

7.7 A Note About CHANGE

A BASIC string of less than six characters always has the remaining character positions filled with zeros. For this reason, when such a string is changed to an array, the first six array elements are set to zero. The CHANGE statement always fills six array elements, even though the strings may not be six characters long. The user should be very careful to always dimension the array used in a CHANGE statement to at least six. If a string of characters is transformed into an array of less than six elements, an undetected error will occur.

The CHANGE statement is usable with strings not created by BASIC. It can, for example, be used to access files other than BASIC data files.

Each string variable corresponds to three computer words. The CHANGE statement treats these three words as six bytes, converts each byte to its numeric character code equivalent and stores it in the corresponding array element. The zero element of the array, the string length, is set equal to the number of bytes (character) before the first zero byte. When reading unspecified data, there may be non-zero bytes following this zero byte. If so, they will be transferred to the array well.

7.8 The CHR\$ Function

Occasionally it is desirable to type a character other than the printing ASCII set, or to compute the value of a character to print. For this purpose, the CHR\$ function is used in a PRINT statement. The argument of the CHR\$ function (Modulo 256) is sent as a character to the Teletype. For example:

- 1Ø FOR I=Ø TO 9
- 20 PRINT CHR\$ (I+48);
- 3Ø NEXT I
- 4Ø END

prints "Ø123456789", since 48 to 57 are the ASCII values for the characters "Ø" to "9". The following special characters can also be printed using the CHR\$ function:

| Bell Line Feed Carriage Return Quote (") Back-Arrow (+) Form Feed | CHR\$ (7) CHR\$ (1Ø) CHR\$ (13) CHR\$ (34) CHR\$ (95) |
|---|---|
| Form Feed | CHR\$ (12) |

NOTE

The Teletype will accept characters from Ø to 255 (decimal), many of which do nothing on most kinds of terminals. Some of the special (non-printing) characters should not be used. For example, CHR\$(4) causes a Dataphone to disconnect.

For each ASCII code there is a second acceptable form permitted in CHANGE and CHR\$. The second code is obtained by adding 128 to the code given in the table in Appendix B. For example, CHR\$ would type "A" in response to either 65 or 193 as an argument.

8. Modification to Legal Array Names

Arrays and matrices may have two-character names. Thus, A7(1) and X1(5,5) are legal names and may be used.

9. Modification to DATA Statement

A DATA statement may now be legally terminated by a comma. For example:

10 DATA 1,2,3, 20 DATA 4,5,6

is now treated the same as

1Ø DATA 1,2,3,4,5,6

10. Program Chaining

Most BASIC programs are easily accommodated by TSS/8 BASIC. If a program becomes very long, however, it may be necessary to break it down into several segments. Typically, programs of more than two to three nundred statements must be split up. A program that has been broken down into more than one piece is commonly referred to as a "chained" program.

Each part of a chained program is saved on the disk as a separate file. The last statement of each part to be executed is a CHAIN statement specifying the name of the next program section. This file is then loaded and executed. It may in turn chain to still another section of the program. The general form of the CHAIN command is:

414 CHAIN "NAME"

or

414 CHAIN A\$

where "NAME" is the name of the next segment to be executed (one to six characters) enclosed in quotation marks. The name of the next segment may also be contained in a string variable. In either case the file of that name is loaded and run. Thus, the statement:

999 CHAIN "SEG2"

is equivalent to:

OLD PROGRAM NAME-SEG2

RUN

except that it happens automatically. Each separate part of the program links to the next part of the program chain.

The individual sections of a chained program may be either regular source files (.BAS) or compiled files (.BAC). If the sections are source files, however, they must be compiled before they are run. A chained program runs more efficiently if all its sections have been compiled.

If an error occurs while compiling or running a chained program, the name of the section being run, the one having the error, is typed out as part of the error message. In all cases, whether a program terminates by an error or a STOP or END, BASIC returns to the first program in the chain. This is the one which is available for editing and rerunning when BASIC types READY.

Most chained programs require information from one section to be passed on to the next. The first section may, for example, accept input values and perform some preliminary calculations. The intermediate results must then be passed to the next section of the programs. This passing of values is done by means of BASIC's file capability, which is explained in the next section. Whenever a CHAIN operation is performed, program data which has not been saved in a file is lost. Variable and array values are not automatically passed to the next program.

ll. Disk Data Files

The standard BASIC language provides two ways of handling program data items. They may be stored within the program (in DATA statements) or they may be typed in from the terminal. DATA statements, however, allow for only a limited amount of data. Also, the data is accessible only to the program in which it is embedded. Typing data from the terminal allows it to be entered into any program, but it is a time-consuming process. In either case the data, or the results of calculations, cannot be conveniently stored for future use. All these limitations may be overcome by the use of external data files.

A data file is separate from the program or programs which use it. It is a file on the disk just like a saved program, but it contains numbers or strings rather than program statements. This information may be read or written by a BASIC program. (Information is stored in a data file in a coded format. Therefore, it cannot be listed by the BASIC Editor or TSS/8 EDIT.) A file may be as long as necessary, subject only to the file limitations of TSS/8. (Maximum file size is about 350,000 characters). String and numeric information may be combined in a single file. The number of data files a user may have is again limited by TSS/8 (about 100, space allowing). When first created, the contents of a file are unspecified, until it is entered.

11.1 File Records

A file is made up of logical units called "records". A record may be as small as a single numeric or string variable. More typically, it is a group of variables or arrays. The design of the program itself usually dictates the most efficient size of the record. If, for example, the program manipulates a series of 5 x 5 matrices, each record could contain one such matrix. If the program operates on 80 character alphanumeric records, 14 string variables might make up a record.

The size and composition of a record is defined with a RECORD statement. Like the DIM statement, RECORD is followed by a series of variables. They may, however, be unsubscripted as well as subscripted. For example:

```
1Ø RECORD A(5,5)
```

The set of variables mentioned in a RECORD statement, taken altogether, constitute a record. Each element within the record is in essence a field. Numeric and string information may be mixed in order to make up the most convenient record.

Variables mentioned in a RECORD statement should not appear in a DIM statement. The RECORD statement reserves variable space exactly as a DIM statement does. The difference is that the variables are also identified as being used for file input and output. Non-subscripted variables appearing in RECORD statements must not have been used previously in a program. RECORD statements should always be the first in a program.

¹Ø RECORD B\$(14)

¹⁰ RECORD A, B, C\$(8), D, E(5)

Records may be any length. A long record is typically more efficient since more information is transferred in a single operation.

Records should, however, be only as long as necessary since excess variables will make the file longer. In particular, it is important to remember that all arrays and matrices have zero elements. A(5,5) has 36 elements, not 25. If A appears as part of a record, all 36 elements should be used. It is also useful to try to make record sizes 43 variables long, or a multiple of 43. Each RECORD statement reserves program variable space in units of 43 whether or not the record is that big. Unless the record fills out this area, some program variable space is wasted. It is not worth it, however, to make an inherently small record 43 variables long just to conform to this convention. To do so would be to make the file unnecessarily large.

11.2 Opening a Disk File

Disk data files are completely separate from the programs which use them. Therefore, the program must specify which file or files it will use. The OPEN command is used for this purpose. OPENing a file associates it with an internal file number, either 8 or 9. (A program may have two disk files open at a time.) For example:

1ØØ OPEN 9, "DATA1Ø"
1ØØ OPEN 8,A\$

The name of the file to be opened may be explicitly stated in the OPEN command. If it is, it must be contained in quotation marks. The file name may also be contained in a string variable, allowing the program to decide which file to open, perhaps on the basis of input from the program's user. In either case, the name of the file is preceded by the internal file number, either 8 or 9. This argument may also be an expression whose value is either 8 or 9. If a file is opened on an internal file number where a file is already open, the previously opened file is closed first.

If no file of that name exists, the file is created. In either case, once the file is open, it is available for both reading and writing. BASIC disk data files have an extension of .DAT.

11.3 Reading/Writing Disk Files

Once open, files may be read and written one record at a time, using the GET and PUT statements. GET statements read one record's

worth of information directly into the variables in the specified RECORD. PUT statements write out the present values of the variables in the specified RECORD. Both GET and PUT statements are followed by the internal file number (8 or 9 or an expression), the line number of the RECORD statement containing the variables to be transferred, then the name of a "control" variable. For example:

```
100 RECORD A, B, C$(30), D(8)
110 OPEN 8, "FILE1"
120 LET I=0
130 GET 8, 100, I
```

The control variable specifies the file record to be transferred. In the example above, FILE1 is open as internal file 8. The value of I is zero. Therefore, the GET statement in line 130 reads the first record (record 0) of FILE1 into A, B, and the arrays C\$ and D. Single numeric values are read into A and B, 31 strings are read in C\$, and 9 numeric values are read into D. After each transfer, whether it is a GET or a PUT, the value of the control variable is automatically incremented. Successive GET's or PUT's automatically proceed to the next record of the file.

The PUT command has a similar format. For example, if line 130 of the above program had been:

the present values of A, B, C\$, and D would have been written out to the first record of FILE1.

File records may be accessed randomly by simply setting the control variable to the desired record number before doing the GET or PUT. Single records may be read, changed, and then written back without the need to process the entire file. When reading a file, the record referenced in the GET statement must, of course, be the same as the record referenced in the PUT statement which wrote the data into the file. The total length of the record and the relationship of string and numeric fields within the records used for the GET's and PUT's must be the same. If they are not, improper information will be read and written.

New files may be created by opening a file which does not already exist. As successive records are written out to the file its length

is extended as necessary. When a new file is created, it is useful to immediately write an "end-of-file" code in the last record. Writing the last record first forces the entire file to be allocated, making sure that enough disk space is available. It also provides an end-of-file marker. Programs which read this file may then check for this end-of-file to avoid reading past the end, which is an error. Existing files may be enlarged by writing a new record farther out. If the program does not know how big the file will be, it may simply write records out serially. The file will be automatically extended as needed. When all the records have been written, one final end-of-file mark may be added.

In general all records read or written in a specific file should be the same length, that is, contain the same number of variables. However, if the user is careful he may intermix records of different lengths in a file. Suppose the following statement is executed:

4Ø PUT 8,1ØØ,N

and the value of N is n and the record specified by statement 100 is of length m. The PUT statement will write m variables in the file starting at the m*n variable.

The simple rule for computing the first variable in the file to be accessed is the record length times the record number. (Remember, the first record is record number zero.)

11.4 Closing and Deleting Disk Files

Ohce all work has been completed on a file, it should be "closed" by a CLOSE statement. Once it is closed, it may not be read or written unless it is reopened. The file does, however, remain on disk and is available for future use. The CLOSE command is followed by the internal file number to be closed (8 or 9). For example:

95Ø CLOSE 8

If the disk file was just created for temporary scratch use (to pass parameters during a CHAIN, for example) it should be deleted at the end of the program instead of closed. The UNSAVE command is used to delete files. For example:

1000 UNSAVE 9

The file open on internal file number 9 is deleted from the disk. Both CLOSE and UNSAVE may be followed by an expression instead of a constant.

Open disk files are automatically closed at the end of the program, unless the program CHAINs to another program. In this case, all open files remain open and the new program may access them without executing an OPEN statement.

12. DECtape Files

Large permanent files are best stored on DECtape instead of disk. Each DECtape holds up to 380,000 characters of information. DECtape files may be dismounted for safekeeping, thereby insuring their privacy. Files on DECtape are very similar to files on disk except that they do not have file names. Each reel of DECtape is a discrete file. When mounted on a DECtape drive, records may be read and written directly on the tape.

A DECtape unit, and hence the file mounted on it, may be used by only one user at a time. If no one is using the unit, a user may assign it. Once assigned, that user has exclusive access to it until he releases it. Each DECtape drive has a "write-lock" switch which physically locks out any writes to that unit. If the write-lock switch is set, programs may not write on the tape even though the unit is assigned.

DECtape files may be used in a variety of ways. Programs which need very big files should use DECtape to avoid swamping the disk. Administrative files, such as student or employee records, are best stored on DECtape. Since they are removable, and can be write-locked when mounted, their usage can be tightly controlled. DECtapes are also useful for information retrieval. A data tape may be kept permanently mounted, but write-locked. Individual users may run programs which assign and query that file, then melease it for others to use.

12.1 DECtape File Records

Records for DECtape files are specified the same way as for disk files, with a RECORD statement. All rules for disk records also apply to DECtape records. In fact, the same RECORD statement may be used for both a DECtape and disk file. (This is useful when reading a tape file to a disk file for processing. Disk files are considerably faster than tape files.)

It is possible to specify any record length for a DECtape file, but a size of 43 variables is suggested, even more strongly than for disk files. DECtapes are physically structured into blocks, each of which will hold exactly 43 variables. If the record specified by the program is, for example, 44 variables, it will require two full blocks of the tape.

Records which are multiples of 43 variables are efficient in utilizing DECtape space, but are not efficient in speed. Such records are written in consecutive DECtape blocks. The tape unit cannot read or write consecutive blocks without stopping the tape and rewinding it slightly.

This tape "rocking" also occurs when single block records (43 variables or less) are read or written as consecutive DECtape records. (In this case, each DECtape file record corresponds to a physical tape block.) The most efficient way to utilize DECtape is to make each record 43 variables in length, and write them onto every tenth record in the file (records \emptyset , $1\emptyset$, $2\emptyset$, etc.). When the entire length of the tape has been traversed (the last block of the tape is number 1473) write next into records 1, 11, 21, etc. In this way, every record will eventually be filled. Programs which will be used repeatedly should utilize the tape in this manner.

12.2 Opening a DECtape File

DECtape files, like disk files, are completely separate from the programs which use them. Therefore, the program may specify which tape, or tapes it will use. The OPEN command is used for this purpose. Since DECtape files do not have names¹, the OPEN command specifies the DECtape unit number to be used. It is assumed that the proper tape reel has been mounted. If the file is to be updated, the unit should be write-enabled. If not, it should be write-locked. The OPEN command is thus followed by the unit number to be used $(\beta-7)$:

100 OPEN 2 100 OPEN 7

It is important to note that BASIC DECtape files are not the same as the file-oriented DECtapes used by TSS/8 COPY. There is no directory on a BASIC DECtape file. Each tape is considered to be one file of pure data.

The unit number could be an expression. Making the unit number a variable is very useful since it is hard to predict which units will be available at the time the program is run. When it is a variable, the user may mount the file on any free unit, then type the number into the program via an INPUT statement.

When the OPEN command is executed, the indicated DECtape unit is assigned. It cannot subsequently be opened or assigned by any other user. Thus, it is possible to try to open a unit which is already assigned. If, in the above examples, units 2 or 7 were assigned, the program would be terminated and an error message typed out. An alternative form of the OPEN command allows the program itself to handle this situation. OPEN commands may include an ELSE clause which specifies a line number. If the OPEN command fails, BASIC automatically performs a GOTO to this line number. For example:

100 OPEN 2 ELSE 900

If unit 2 is available, it is assigned and BASIC goes on to execute the next statement. If unit 2 is not available, statement 900 is executed next. It could print a message and perhaps ask for an alternate unit number.

12.3 Reading and Writing DECtape Files

DECtape files are read and written using the same GET and PUT commands as are used for disk data files. The internal file number is a number between \emptyset and 7, or an expression. Unlike disk files, DECtape files are of a constant length equal to the capacity of the tape. The exact number of records per reel depends on the record size as follows:

| Record Size | Tape | Capacity |
|---|------|--------------------|
| 1-43 variables 44-86 variables 87-129 variables | 737 | records records |

As indicated in the section on DECtape records, a record size of 43 variables or less is recommended since it conforms to the physical blocking of the tapes themselves. It is also desirable to space the records out along the tape so that the tape does not rock. The following subroutine could be used to write 1474 records on the tape in this fashion. It assumes that R is set to zero before it is called the first time and that the unit number is in U.

- REM SUBROUTINE TO WRITE RECORDS ALONG TAPE 5 a a 51Ø
- REM WRITES ONE RECORD EACH TIME CALLED
- 515 PUT U, 10, R 'REMEMBER THIS INCREMENTS R 517
- LET R=R+9 SPACE OUT 10 BLOCKS 524
- IF R≤1474 THEN 55Ø 'OK TO RETURN 53Ø
- IF R=1479 THEN 560 'TAPE IS FULL 54Ø
- LET R=R-1479
- 545 IF R>Ø THEN 55Ø
- LET R=R+1Ø 547
- 55Ø RETURN
- 56Ø STOP 'TAPE IS FULL

The following function may also be used to convert a logical record number (Ø to 1469) to a physical record block spaced along the tape. This function will not use blocks Ø-3. They are therefore available for headers or labels.

FNC (X) = $(X-INT(X/147)*147)*1\emptyset+INT(X/147)+4$

Both the subroutine and the function assume a record length of 43 vari-

Once opened, any record on the tape may be read. The tape unit must, however, be write-enabled if it is to be written. Trying to PUT to a write-locked tape is an error.

12.4 Closing DECtape Files

Once all work on a DECtape file has been completed it may be closed. Closing a file releases the tape unit and makes it available to other Thus, if the tape contains important information (and especially if it is write-enabled) the CLOSE should not be done until the tape reel has been removed. If no CLOSE statement is encountered in the program, the unit remains assigned even after the program finished. It will remain assigned until a TSS/8 RELEASE command is executed or the user logs out. An example of a CLOSE command is:

1100 CLOSE 6

12.5 Using Data Tapes with PS/8 FORTRAN

Numeric DECtape data files written by TSS/8 BASIC may be read by PS/8 FORTRAN by use of FORTRAN's RTAPE and WTAPE subroutines, and vice versa. (String and Hollerith variables use different character codes.) Thus it is possible to use BASIC to prepare an input or update tape for a stand-alone FORTRAN program. This provides a convenient way to do

large jobs in off-hours, without having to leave the time-sharing mode for very long.

13. Line Printer Output

If a line printer is available, it may be used both to list BASIC programs and as an output device for the programs themselves. The line printer may only be used by one user at a time.

The commands associated with line printer output are LLIST and LPRINT.

LLIST is similar to the LIST command except that the program listing is output to the line printer rather than to the Teletype. The LLIST command assumes that no other user has the line printer assigned and responds by typing WHAT? if the line printer is not available. After the listing is complete, the line printer is released and is available to any user.

BASIC programs may use the line printer as an output device during execution by means of the LPRINT command. LPRINT is exactly like PRINT except that, again, the information goes to the line printer rather than to the Teletype. All formatting conventions of the PRINT command are available with LPRINT. In particular, CHR\$(12) may be used to skip to the top of the next form.

The command LPRINT also assumes that no other user has the line printer assigned. However, using this command when the line printer is not available causes the program to terminate. Once LPRINT successfully assigns the line printer, it remains assigned until the program terminates.

The OPEN and CLOSE commands may be used to assign and release the line printer. An OPEN command with a device number of 11 will assign the line printer, or if it is not available and an ELSE clause is specified, transfer control to the line number specified in the ELSE clause. CLOSE 11 will release the line printer.

14. Papertape Output

The high speed paper tape punch may also be used as an output device. Like the line printer, the paper tape punch can only be used by one user at a time. The OPEN and CLOSE commands with an internal file

number of 10 will respectively assign and release the paper tape punch as shown in the following example:

10 OPEN 10 ELSE 100 GOTO 100 IF PUNCH UNAVAILABLE 20 CLOSE 10

Here too, a GOTO statement in combination with an ELSE clause can be used to transfer program control should the paper tape punch not be available.

The command LPRINT causes output to go to the paper tape punch when this device has been assigned. For example:

10 OPEN 10 20 LPRINT "THIS GOES TO PTP."

causes the statement "THIS GOES TO PTP." to be punched onto papertape.

If the device is not released via a CLOSE command, it remains assigned even after the program terminates.

15. Truncation Function, FIX(X)

The truncation function returns the integer part of X. For example:

FIX (10.2)=10 FIX (-11.001)=-11

FIX is like INT for positive arguments, and can be defined as:

FIX(X) = SGN(X) * INT(ABS(X))

16. ON GOTO

The ON...GOTO statement may be used to provide a many-way branch. The general form of the ON...GOTO is:

ON expression GOTO line number, line number....

If the value of the integer part of the expression is 1, a GOTO is performed to the first statement. If the value of the integer part of the expression is 2, a GOTO to the second statement number is performed, etc.

If the value is less than one, or greater than the number of statement numbers, the program terminates and an error message is typed out.

Examples of ON...GOTO are:

999 ON N GOTO 100,400,200,600,499 872 ON A+SQR(B*C) GOTO 100,200

17. Implied LET

The word "LET" may be left out of LET statements. Thus these two statements are completely equivalent:

100 LET X=2 100 X=2

18. SLEEP

The SLEEP statement causes a BASIC program to pause for a specified interval, then continue running. SLEEP is followed by the number of seconds the program is to pause. For example:

222 SLEEP 3Ø

or

22Ø LET N=15 222 SLEEP 2*N

causes a 30-second delay in the program.

The SLEEP statement is a useful way for a program to wait for a device (DECtape or line printer) which is busy. The ELSE clause in the OPEN statement can go to a routine which pauses for awhile, then retries the OPEN. When the current user finishes with the device and releases it, the program may then proceed to OPEN and use it. This capability is especially useful when many users may be looking up information on a single DECtape file. It may also be used to allow two programs to communicate with each other. Each writes information on a tape file for the other, or others, to read.

SLEEP should always be used when waiting for a device. While the program is sleeping it is not using any processor time. A SLEEP time of 30 to 60 seconds is recommended. It is particularly important that the program not wait by repetitively retrying the OPEN. To do so wastes computer time and slows down the other users. The integer part of the

argument is used to determine the number of seconds to delay. This value must be between Ø and 4095.

19. Comments

An entire statement of comments may be included in the BASIC program by means of the REM statement. Often comments are easier to read If they are placed on the same line with an executable statement rather than in a separate REMARK statement. This can be accomplished by ending an executable statement with an apostrophe. Everything to the right of the apostrophe up to the statement terminator (carriage return or backslash as described in section 21) is ignored (unless the apos--rophe occurs within a print literal or string constant). For example:

- 10 LET X=Y 'THIS IS A COMMENT'
- 2Ø PRINT "BUT 'THIS IS NOT A COMMENT"
 3Ø LET X\$="A'B"

Thus, a comment is added to line 100 with an apostrophe, but in lines 20 and 30 the apostrophe is treated as a valid character.

20. Blank Lines

To make BASIC programs easier to read, blank lines can be inserted anywhere in a BASIC program. These can be used to break a program into logical sections, or (as is often done) to insert remarks with the apostrophe feature. For example:

> 10 PROGRAM WRITTEN BY SAM JONES 1**øø --- --**

Note that to insert a blank line, you must type one or more spaces after the line number; typing the line number alone will just delete that line from the program.

More than One Statement on a Line

As many statements as will fit may be typed on a single program line. Each statement must be separated by the backslash character "\". (SHIFT/L). The only statement requiring a line number is the initial one. For example:

1Ø FOR I=1 TO 1Ø \ PRINT I \ NEXT I

Note that the backslash character acts as a statement terminator and thus cannot be included in a comment statement.

APPENDIX A

ERROR MESSAGES

The following error messages have been added to those already contained in PROGRAMMING LANGUAGES.

IMPROPER ACCOUNT # ABORT †BS

BAD FILE FORMAT

MISUSE OF CHR\$ IN LINE n

BAD VALUE IN CHANGE STATEMENT AT LINE n

TIME LIMIT EXCEEDED AT LINE n

The error message:

ILLEGAL FORMULA IN LINE n

has been changed to:

ILLEGAL SYNTAX IN LINE n

PROGRAM IS "progname"

A user logged in under account numbers 1 (system account) or 2 (system library) and tried to run BASIC. This is prohibited.

The program specified in response to OLD PROGRAM NAME was not acceptable to BASIC. This is generally caused by: (1) trying to load an obsolete compiled (.BAC) file¹, or (2) trying to load a non-BASIC (FORTRAN or PAL-D) file.

The CHR\$ function was used in an invalid manner. CHR\$, like TAB, can appear only in PRINT statements.

While performing CHANGE A TO A\$, one of the elements of the array A was found to contain an illegal value.

The number of statements executed by a job has exceeded the maximum permitted². Generally, some error was made and the program is caught in a loop.

This message may immediately follow an error message, to identify the current program in a series of CHAINed programs. If there is no CHAINing, this message will not occur.

¹Version 4 will not accept compiled files created by earlier versions of BASIC.

²Thîs limît is established by the System Manager, and may be set to "infinity" if desired.

PROGRAM NOT FOUND AT LINE n

The file which the user tried to access with a CHAIN state-ment does not exist in his disk area. The PROGRAM IS message will also occur.

BAD SLEEP ARGUMENT IN LINE n

The argument of the SLEEP command must have a number greater than or equal to \emptyset , and less than or equal to $4\emptyset95$.

ARRAY OR RECORD USED BEFORE DEFINITION IN LINE n

The RECORD statement more occur before any reference to it is made. A DIM statement must occur before an array is used. (RECORD and DIM are placed at the beginning of a program.)

IMPROPER DIM OR RECORD STATEMENT IN LINE n

Syntax error in DIM or RECORD statement, or an array name that was previously dimensioned is used again. (Replaces IMPROPER DIM STATEMENT IN LINE n.)

CAN'T CREATE FILE IN LINE n

An OPEN statement tried to create a file, but there is: (a) no disk space available, (b) no file name specified, or (c) a null string has been given as the file name.

CAN'T DELETE FILE IN LINE n

UNSAVE cannot delete a file. This is usually due to the fact that another user has the file open, or the file is protected with a code >20.

UNOPEN DISK UNIT IN LINE n

The user tried to do a GET, PUT or UNSAVE to device 8 or 9, without a file being previously opened on the device.

DEVICE BUSY IN LINE n

The user tried to OPEN DECtape, \emptyset -7, line printer, or paper tape punch, but the device was unavailable, and there was no ELSE clause in the OPEN statement.

INVALID RECORD NUMBER IN LINE n

The record number must be a number which is greater than or equal to \emptyset and less than or equal to $4\emptyset95$.

For DECtape I/O the maximum record number is further limited by the DECtape size.

ON INDEX OUT OF RANGE IN LINE n

The value of the index is less than one, or greater than the number of statement numbers. INVALID DEVICE NO. IN LINE n

The device number in the file I/O statement is not between Ø and ll inclusive (or X and ll inclusive, where X is a number set by the system manager).

GET BEYOND END OF FILE IN LINE n

Disk file is too small to have a record with the number specified in the GET statement at Line n.

GET/PUT ERROR IN LINE n

A hardware error occurred in GET or PUT. (This is usually due to a DECtape unit being write-locked.)

CHAIN TO BAD FILE AT LINE n

The file specified by the CHAIN has an invalid format; it is not a BASIC format file. The "PROGRAM IS..." message will follow this error message. The program name will be the name of the bad file.

APPENDIX B

INTERNAL DATA CODES

Using the file I/O capabilities and the CHANGE statement, it is possible to examine data which was written on a DECtape or disk file by a program other than BASIC. There are two data formats, Numeric Data and String Data.

1. Numeric Data

Each numeric value in TSS/8 BASIC is three PDP-8 words long. The format is as follows:

Word 1

Sign Binary Exponent High Order Mantissa

Ø 11

Word 2

Mantissa

11

Word 3

Low Order Mantissa

A one in the sign bit means that the number is negative. The exponent is kept in "excess 200" form where

 200_{8} is 2^{0} 201_{8} is 2^{1} 177_{8} is 2^{-1}

The assumed decimal point is preceding bit 9. Also, the number is always normalized, meaning that bit 9 is always 1 unless the number is zero. (Zero is represented by three zero words.)

Note that this format is the same as the format used by FORTRAN and described in Programming Languages.

2. String Data

Each string variable is three PDP-8 words long. Each word contains two 6-bit bytes or characters. If a string variable is filled by a GET from a source which was not written by a BASIC program, a BASIC program may examine the data in the variable by performing a CHANGE on that variable. The six bytes will be translated as if they were internal character codes for BASIC string characters. The following table shows how this translation interprets the 64 possible bytes. Note that after such a CHANGE, the β^{th} element of the array contains a count of the number of characters occurring before the first null.

| a | | (Numeric Code) | | | | (Numeric Code) | |
|--------------|------------|-------------------|-------|------------|----------------|-------------------|----------------|
| Stri | ng Byte | ASCII Char. | ASCII | Strin | g Byte | ASCII Char. | ACGTT |
| <u>Octal</u> | . Decimal | Code | Char. | Octal | <u>Decimal</u> | _Code | ASCII Char. |
| øø | ø. | ø | NULL | 4ø | 32 | 63 | ? |
| Ø1 | 1 | 32 | SPACE | 41 | 33 | 64 | . @ |
| Ø2 | 2 | 33 | 1 | 42 | 34 | 65 | A |
| ØЗ | 3 | 34 | . " | 43 | 35 | 66 | В |
| Ø4 | 4 | 35 | # | 44 | 36 | 67 | C |
| Ø5 | 5 . | 36 | \$ | 45 | 37 | 68 | D |
| Ø6 | 6 | 37 | · 8 | 46 | 38 | 69 | E |
| Ø7 | 7. | 38 | . & | 47 | 39 | 7 ø | F. |
| 1ø | 8. | 3 <u>9</u> | * | 5 <i>ø</i> | 40 | , 71 | G |
| 11 | 9 | 4ø | C | 51 | 41 | 72 | н |
| 12 | 1ø | 41 |) | 52 | 42 | 73 | Ī |
| 13 | 11 | 42 | * | 53 | 43 | 74 | J |
| 14 | 12 | 43 | + | 54 | 44 | 75 | K |
| 15 | 13 | 44 | , | 55 | 45 | 76 | L |
| 16 | 14 | 45 | - | 56 | 46 | 77 | M |
| 17 | 15 | 46 | • | 57 | 47 | 78 | N |
| 2ø | 16 | 47 | / | 6ø | 48 | 79 | 0 |
| 21 | 17 | 48 | ø | 61 | 49 | 8 <i>ø</i> | P |
| 22 | 18 | 49 | 1 | 62 | 5ø | 81 | Q |
| 23 | 19 | 5ø | : 2 | 63 | 51 | 82 | R R |
| 24 | 2ø | 51 | 3 | 64 | 52 | 83 | S |
| 25 | 21 | 52 | 4 | 65 | 53 | 84 | T |
| 26 | 22 | 53 | 5 | 66 | 54 | 85 | U |
| 27 | 23 | 54 | 6 | 67 | 55 | 86 | v |
| 3ø | 24 | 55 | 7 | 7ø ' | 56 | 87 | W |
| 31 | 25 | 56 | 8 | 71 | 57 | 88 | X |
| 32 | 26 | 57 | 9 | 72 | 58 | 89 | Y |
| 33 | 27 | 58 | : | 73 | 59 | 9ø | Z |
| 34 | 28 | 59 | ; | 74 | 6 / 2 | 91 | I . |
| 35 | 29 | 6ø | < | 75 | 61 | 92 | ` |
| 36 | 3 <i>ø</i> | 61 | = | 76 | 62 | 93 |] |
| 37 | 31 | 62 | > | 77 | 63 | 94 | † |

NOTES TO SYSTEM USERS ABOUT

BASIC VERSION 4

1. Program Size

The features of Version 4 were implemented at almost no cost in terms of program performance. In fact, most BASIC programs will probably run faster under version 4 than under version 3. It is still possible that some programs which ran under version 3 will no longer run. This is due to the decrease in symbol table space in the compiler. However, additional space is now available at runtime so there should be many programs which are able to run under version 4 which would be too large for version 3. The following are the current restrictions imposed by the amount of available space:

- a. During compilation, version 4 has (roughly) 1200 words for symbol tables (version 3 had 1350 words). This space is used as follows:
 - 2 words for each line in the program.
 - 3 words for each simple variable and every one-variable array.
 - 4 words for each two-dimensional matrix, each DEF statement and each RECORD statement.
 - 4. 5 words for every constant used in the program.
- b. At runtime there are at least 943 words available in version 4 (version 3 had 909 words). If the file I/O capabilities are not utilized by the program being run (i.e., no RECORD, GET, PUT, OPEN, CLOSE, or UNSAVE statements) then 1215 words are available. Three words are used by every variable, array element, constant, and every DATA item (both string and numeric).

Each RECORD statement may cause additional space to be used. The space used for variables in RECORD statements is allocated in blocks of 129 words. Thus, the statement:

RECORD A,B\$(9),C(4,4)

causes 105 words to be allocated for the variable A, the string array B\$, and the array C, and also causes 24 more words to be allocated although unused by the program. NOTE: This space is used as buffer space if the record is read or written on DEC-tape.

The space left over after deducting for the above is used for the program itself. The program is "paged" off the disk; so with less core space available, more disk activity will take place. Since more space is available, programs will probably run faster under BASIC Version 4.

The considerable gain in space available at runtime is due to the fact that version 4 uses the same core area for the INPUT statement processing routines and seven of the arithmetic functions. When the interpreter is called, the input routines are initially loaded. The first occurrence of a call to one of the overlayed functions (SIN, COS, TAN, ATN, LOG, EXP, and SQR) causes the code for these functions to be loaded over INPUT. These functions then remain resident until an INPUT statement is executed. At that time, the input routines are again loaded. Thanks to the new TSS/8 Monitor (Version 8.22A) which keeps BASIC's file segment window permanently resident, the overlay time is normally insignificant. Since the question mark is typed by the input processor before it loads the main input routines, the user generally will not notice that any delay has occurred.

2. Using the "TIME LIMIT" Feature

When the following PATCH is made, BASIC increments a counter every time the program executes a statement. When the counter overflows, the "TIME LIMIT EXCEEDED AT LINE n" error message occurs, so this feature actually counts executed statements, not real time. None-theless, it is an excellent way to prevent inexperienced programmers from tying up the system in a compute-bound loop.

If this PATCH is not made, the counter is never incremented and so the "TIME LIMIT EXCEEDED" error can never occur.

The PATCH is:

LOAD BASIC 14157 400 6226

| At Location | Deposit | Comments |
|-------------|---------|-----------------------|
| 22Ø1 | val (1) | high 12-bits of COUNT |
| 22Ø2 | val (2) | low 12-bits of COUNT |
| 22Ø3 | 5225 | JMP to ENABLE PATCH |

then

.SAVE BASIC 14157 400 6226

The contents of 2201 and 2202 contain a negative 24-bit number (COUNT), which is the number of statements to execute before giving the "TIME LIMIT EXCEEDED" error.

We have found that the following setting is good:

22Ø1=777Ø 22Ø2=Ø

which sets COUNT to -32,768.

This will time out a program like:

1Ø GOTO 1Ø

in about 15 seconds of CPU time, while permitting "real" jobs a minute or more of CPU time, which is enough in nearly all cases.

In some cases, more complex decisions must be made before setting this time-out feature. For example, suppose we wanted to permit users with Project Numbers of \emptyset unlimited time, while timing out everyone else.

.LOAD BASIC 14157 400 6226

Deposit the following PATCH in core:

| Location | Value | Comments |
|--|--|---|
| 22Ø1 22Ø2 22Ø3 | 777ø ø 5235 | high 12-bits of COUNT low 12-bits of COUNT JMP to special PATCH |
| 2234 2235 2236 2237 224Ø 2241 | 77ØØ 6617 Ø234 765Ø 52Ø4 5225 | (7700 ACT AND (7700 SNA CLA JMP to ignore PATCH JMP to enable PATCH |

Then:

.SAVE BASIC 14157 400 6226

Locations 2234 through 23 \emptyset 2 and 2334 through 2377 are available for such patches.

3. Prohibiting Use of DECtape Units

It is possible to deny BASIC programs the use of certain device numbers in file I/O statements. In particular, this means certain DEC-tape units can be protected from use by BASIC programs. This is done by the following PATCH:

.LOAD BASIC 14157 400 6226 .DEP 5653 -n .SAVE BASIC 14157 400 6226

where:

n = octal equivalent of lowest valid device number.

For example, depositing a -4 in the location would make \emptyset , 1, 2, and 3 invalid when used as the device numbers in a file I/O statement. This means DECtape units \emptyset through 3 are not available to BASIC programs while units 4 through 7 are available.

All DECtape usage may be prevented by depositing a -10 (octal value 7770). All DECtape and disk usage may be denied by depositing a -12 (octal value 7766).

4. Random Comments

- a. The file BASIC occupies 38 segments of disk, rather than the 35 segments needed by Version 3.
- b. Some users have wanted to be able to run BASIC under account #2 (System Library) which is currently a restricted account. To "unrestrict" account #2, make the following PATCH:

.LOAD BASIC .DEP 4212 7410 .SAVE BASIC

5. Patches to other CUSPs

Since BASIC Version 4 introduces a new file extension, .DAT (bits \emptyset -4 of protection word = \emptyset 1 \emptyset 01), it is necessary to update COPY, CAT and LOGOUT to recognize this extension, using the following patches:

COMPUTING SERVICES DIVISION

Publication Number 101

TSS/8 BASIC

Second Printing July, 1974

THE UNIVERSITY OF WISCONSIN-MILWAUKEE

Please complete and return the Documentation Registration Form enclosed with this manual. Updates and revisions will be sent automatically to any purchaser whose form is on record.

This handbook contains extensive amounts of information taken from the 1970 editions of "Programming Languages" and "TSS/8 Timesharing Systems Users Guide", both published by the Digital Equipment Corporation, Maynard, Massachusetts. These excerpts were taken with permission of the publisher.

Permission is hereby granted to reproduce this material in any form desired. Please inform the Editor when this has been done.

First Printing June, 1973. Second Printing July, 1974.

THE UNIVERSITY OF WISCONSIN-MILWAUKEE COMPUTING SERVICES DIVISION

DOCUMENTATION REGISTRATION FORM

(Completion of this form will place the purchaser's name on a mailing list to receive updates of the purchased document)

| NAME (please print) | | DATE |
|--|---|-----------------------------|
| I HAVE PURCHASED A COPY | OF (Title of Document | ation) |
| PLEASE SEND UPDATES AND | REVISIONS OF THIS DOCUMENT TO: | |
| | Campus Address* | |
| are asked to spec new material will | Off-campus Address ag expenses, purchasers with a calcify this as their preferred address be sent. Purchasers with no cam al sent to the off-campus address | ess to which pus address |
| CAMPUS ADDRESS: | Dept. | |
| | Room & Bldg. | |
| | Telephone | |
| OFF-CAMPUS ADDRESS: | StreetCity, State, Zip | |
| | Telephone | |
| Please fold, staple, | and mail this form to address s | hown. Thank you |

TO: The University of Wisconsin-Milwaukee Computing Services Division, Retail Sales EMS EB75 Milwaukee, Wisconsin 53201 This manual is intended as a complete textbook for the teaching of the BASIC language as well as a description of the implementation of BASIC ASCII version V as it occurs in TSS/8.

Publication Number 101

TSS/8 BASIC

Second Printing July, 1974

Prepared by the TSS/8 Programming Group of the Computing Services Division at The University of Wisconsin-Milwaukee.

Editor

L. P. Levine

Second Printing July, 1974

Please complete and return the Documentation Registration Form enclosed with this manual. Updates and revisions will be sent automatically to any purchaser whose form is on record.

This handbook contains extensive amounts of information taken from the 1970 editions of "Programming Languages" and "TSS/8 Timesharing Systems Users Guide", both published by the Digital Equipment Corporation, Maynard, Massachusetts. These excerpts were taken with permission of the publisher.

Permission is hereby granted to reproduce this material in any form desired. Please inform the Editor when this has been done.

First Printing June, 1973.

Second Printing July, 1974.

TABLE OF CONTENTS

| Cont | ents | | | | Page |
|------|--|---|-----------------------------|---|--|
| HOW | TO USE | THIS MANUAL | CHAPTER | 1 | |
| МАСН | INE IND | EPENDENT BASIC | CHAPTER | 2 | |
| 2.1 | Introd | luction to BASIC progra | mming | | 2.1 |
| | 2.1.2 | About computing LOGIN procedure How to use this manua | 1 | | 2.2 2.4 2.4 |
| 2.2 | Fundam | entals of Programming | in BASIC | | 2.6 |
| | 2.2.2 2.2.3 2.2.4 2.2.5 2.2.6 2.2.7 2.2.8 2.2.9 2.2.10 2.2.11 2.2.12 2.2.13 2.2.14 2.2.15 2.2.16 2.2.17 2.2.18 2.2.19 | Sign function SGN(X) Integer function INT Random number functi RANDOMIZE statement Time function TIM(X) User defined functio Input/output stateme READ statement DATA statement RESTORE statement INPUT statement PRINT statement | s (X) on RND(X) ns | | 2.6 2.8 2.10 2.12 2.14 2.15 2.15 2.15 2.16 2.17 2.18 2.20 2.21 2.26 2.26 2.27 2.29 2.31 2.34 |
| 2.3 | Subscr | ipts and Loops | | | 2.37 |
| | 2.3.1 2.3.2 2.3.3 2.3.4 2.3.5 2.3.6 | NEXT statement | | | 2.37 2.39 2.41 2.41 2.42 2.43 |

| Cont | ents | | | Page |
|------|---|--|-----------|--|
| 2.4 | Transfe | er of Control | | 2.45 |
| | 2.4.2 2.4.3 2.4.4 2.4.5 2.4.6 2.4.7 2.4.8 | Unconditional transfer Conditional transfer IF-THEN and IF-GOTO sta Subroutines GOSUB statement RETURN statement Location of subroutine STOP and END statements Nesting subroutines | tement | 2.45 2.47 2.48 2.48 2.50 2.51 2.51 |
| 2.5 | Errors | and How to Make Correct | ions | 2.53 |
| 2.6 | 2.5.2 2.5.3 2.5.4 Running 2.6.1 2.6.2 2.6.3 2.6.4 2.6.5 2.6.6 2.6.7 2.6.8 2.6.9 2.6.10 2.6.11 2.6.12 | Single letter correction Erase a line Erasing a program in constopping a RUN in TSS/8 a BASIC Program LOGIN procedure Initial dialog RUN command Editing phase SAVE command REPLACE command UNSAVE command LIST command LIST command DELETE command NEW and OLD commands CATALOG command BYE command ALTMODE key | re | 2.53 2.54 2.55 2.56 2.57 2.57 2.57 2.59 2.60 2.61 2.62 2.63 2.63 2.63 2.64 2.64 |
| TSS/ | 8 ADVANC | ED BASIC | CHAPTER 3 | |
| 3.1 | Impleme | ntation Notes | | 3.1 |
| 3.2 | Punchin | g a Paper Tap e | | 3.1 |
| 3.3 | Reading | and Listing a Paper Tap | pe | 3.3 |
| | 3.3.1 | Transferring a file to I | DECtape | 3.4 |
| 3.4 | EDIT co | mmand | | 3.4 |
| 2 5 | COMPTER | command | | 2 7 |

| Cont | ents | | Page | |
|------|---|--|--|--|
| | | File extensions File protection | 3.8 3.9 | |
| 3.6 | String | gs in BASIC | 3.11 | |
| | 3.6.2 3.6.3 3.6.4 3.6.5 3.6.6 3.6.7 3.6.8 | | 3.11 3.13 3.13 3.14 3.16 3.16 3.18 3.19 3.20 | |
| 3.7 | Progra | m CHAINing | 3.20 | |
| 3.8 | Disk D | ata Files | 3.22 | |
| | 3.8.2 3.8.3 | File records OPENing a disk file Reading/writing disk files CLOSing and deleting disk files | 3.22 3.24 3.25 3.27 | |
| 3.9 | DECtap | e Files | 3.28 | |
| | 3.9.2 3.9.3 3.9.4 | DECtape file records OPENing a DECtape file Reading and writing DECtape files CLOSing DECtape files Using DECtapes with PS/8 FORTRAN | 3.29 3.30 3.31 3.32 3.33 | |
| 3.10 | Line | Printer Output | 3.33 | |
| 3.11 | l Papertape Output 3.34 | | | |
| 3.12 | 2 ON-GOTO Command 3.35 | | | |
| 3.13 | SLEEP | Command | 3.35 | |
| 3.14 | Comme | nts | 3.36 | |
| 3.15 | Blank Lines 3.37 | | | |
| 3.16 | More Than One Statement on a Line 3.37 | | | |

| Cont | ents | | Page |
|-------|---|-----------------------------|-----------------------------------|
| 3.17 | Internal Data Codes | | 3.38 |
| | 3.17.1 Numeric data 3.17.2 String data | | 3.38 3.39 |
| SUMM | ARY | CHAPTER 4 | |
| 4.1 | Summary of BASIC Statements | | 4.1 |
| | 4.1.1 Functions | | 4.4 |
| 4.2 | Summary of BASIC Edit and Contr | ol Commands | 4.4 |
| 4.3 | Summary of BASIC Error Messages | | 4.6 |
| | 4.3.1 Edit phase messages 4.3.2 Input phase messages 4.3.3 RUN phase messages 4.3.4 Non-fatal execution phas 4.3.5 Other messages | e mes sa ge s | 4.6 4.7 4.7 4.11 4.12 |
| ASCII | I CONSTANTS | APPENDIX A | |
| INDEX | X | APPENDIX B | |

This manual describes the BASIC language as implemented on the Digital Equipment Corporation TSS/8 Time-Sharing Computing System. As such, it is a compromise, attempting to describe BASIC in its original Dartmouth College form, and to include those changes, both of a limitation and enhancement nature, which have been made in BASIC as it was implemented on TSS/8.

We have therefore chosen to discuss the BASIC language in two sections, reserving for chapter 2 those portions of BASIC which are machine independent, and placing in Chapter 3 those portions which are specific to the TSS/8. Even this cannot be done successfully, as some of the features and restrictions of

- 8 TSS/8 BASIC are so ingrained in the language implementation that
- 8 a sensible discussion is not possible without reference to them
- 8 in chapter 2. When we have done so we will indicate that the
- 8 text is referring to TSS/8 BASIC by a marginal "8" just as we
- 8 have done here.

Chapter 3 is devoted almost exclusively to TSS/8 implementation notes.

Chapter 4 contains a list of all TSS/8 BASIC Statements, commands and error messages.

2.1 INTRODUCTION TO BASIC PROGRAMMING

BASIC is an easy to learn, conversational, computer language for scientific, business, and educational applications.

It is used to solve both simple and complex mathematical problems and is directed from the user's Teletype.

In writing a computer program, it is necessary to use a language or vocabulary that the computer will recognize. There are many computer languages and BASIC is one of the simplest because of the small number of readily learned commands needed, its easy application in solving problems, and its practicality in an educational environment.

BASIC is similar to other programming languages in many respects and is aimed at facilitating communication between the user and the computer. The novice computer user will benefit from reading the entire manual from the beginning. The user who is already familiar with a language such as FOCAL or FORTRAN should first turn to the language summary in chapter 4.

As a BASIC user, you type in your computational procedure as a series of numbered statements, making use of common English words 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 more advanced techniques needed to perform more intri-

cate manipulations and to express your problem more efficiently and concisely. Once you have entered your statements, you give a RUN command. This command initiates the execution of your program and causes the return of your results almost instantly.

2.1.1 About Computing

As we approach a computer terminal, there is a certain way we attack a problem. It is not enough to understand the technical commands of a computer language, we must also be able to correctly and adequately express the problem to be solved. For this reason it will be helpful to outline the process of setting up a problem for computer solution.

The first step is to define the problem to be solved in detail. Understand each fact and possibility within the problem before attempting to go any further. Problems to be solved with BASIC are generally of a level which admit to fairly straightforward analysis.

In computing there is always more than one correct way of approaching a given problem. Generally a standard mathematical method for solution can be found, or a method developed. Programs using the same method can still be written in more than one correct way.

For some complicated programs a flowchart is useful. A flowchart is a diagram which outlines the procedures for solving the problem, step by step.

Having a diagram of the logical flow of a problem is a tremendous advantage to you when determining the mathematical techniques to be used in solving the problem, as well as when you write the BASIC program. In addition, the flowchart is often a valuable aid when checking the written program for errors.

A flowchart is a collection of boxes and directed lines. The boxes indicate, in a general fashion, what is to be done; the directed lines indicate the sequence of the boxes. The boxes have various shapes representing the type of operation to be performed in the program (input, computation, etc.). Texts on flowcharting are available at academic bookstores. A student interested in this subject is advised to obtain one.

Following satisfactory completion of a flowchart, you proceed to write the program. To do this you need to understand the various instructions and capabilities of the BASIC language. The rest of this chapter is designed to teach you how to write programs in the BASIC language in a minimal amount of time.

Once the correct procedure has been coded, it is time to try it on the computer. At this point it is possible the program will not work perfectly as originally written. BASIC will locate any mistakes the programmer has made in typing his program and print appropriate error messages to help him correct them. It is important to understand that even if the program does run, the results will only be correct if the problem has been correctly analyzed and proper code written to achieve the correct

solution. A computer can only do what you tell it to do. If you have unknowingly told a computer to do something other than what you wanted it to do, the results will be accurate according to the information the computer processed. The computer cannot know what you really want, only what you have told it.

- 8 2.1.2 LOGIN Procedures *
- 8 It is assumed in this manual that the user has access to a
- 8 TSS/8 system and has an account number available to him on that
- 8 system. Although it is not required for an understanding of
- 8 BASIC, a copy of the TSS/8 USER'S GUIDE will aid him in commu-
- 8 nicating with the computer.

2.1.3 How to Use This Manual

The most straightforward treatment of the BASIC programming language will be obtained by reading this manual from the beginning. Examples are taken directly from Teletype output so that the reader will become familiar with the computer output and formats. Once you have mastered the principles of BASIC language, you will most likely only need to refer to the summaries found in chapter 4.

Detailed examples appear and may be run on the computer as a first exercise before attempting an original program.

The early sections of this manual contain directions on how to write a BASIC program. The section on Implementation Notes is recommended for every reader. Once you have written *The marginal "8" is used to call attention to TSS/8 BASIC special features in chapter 2.

several BASIC programs you will find chapter 3 helpful; reading that chapter too early in your programming experience may be confusing. As soon as you are ready to try running a BASIC program on the computer turn to the section on Running a BASIC program.

2.2 FUNDAMENTALS OF PROGRAMMING IN BASIC

2.2.1 An Example Program and Output

At this point the program in Figure 2-1 may mean little to you, although the output (following the word RUN) should be fairly clear. One of the first things you notice about the

- 8 program is that each line begins with a number. TSS/8 BASIC
- 8 requires that each line be numbered with an integer from 1 to
- 2046. When the program is ready to be run, BASIC executes the statements in the order of their line numbers, regardless of the order in which you typed the statements. This allows the later insertion of a forgotten or new line. The programmer is, therefore, advised to leave gaps in his numbering on the first typing of a program. Numbering by fives or tens is a common practice.

The next thing we notice about the program is that each line begins with a word, a command to the computer to tell it what to do with the information on that line. BASIC does not understand the statement V=0 unless we write LET V=0. Once we understand the usage of these commands we are able to describe our problem to the computer.

2.2.2 REM Statement

The REM or REMARK statement allows the programmer to insert notes to himself or anyone who will read the program later. The form is:

(line number) REM (message)

FIGURE 2-1

```
LIST
   10 REMARK - PROGRAM TO TAKE AVERAGE OF
   15 REMARK - STUDENT GRADES AND CLASS GRADES
   20 PRINT "HOW MANY STUDENTS, HOW MANY GRADES PER STUDENT";
   30 INPUT A,B
  40 LET I = 0
   50 \text{ FOR J} = 1 \text{ TO A-1}
   55 LET J=0
   30 PRINT "STUDENT NUMBER =";J
   75 PRINT "ENTER GRADES"
   75 LET D=J
   80 FOR K = D TO D+(B-1)
  81 INPUT G
  82 LET y = y + G
  85 NEXT K
  90 LET 7 = 7/B
  95 PRINT "AJERAGE GRADE =";V
  96 PRINT
  99 LET Q = Q + V
 100 NEXT J
 101 PRINT
102 PRINT
 103 PRINT "CLASS AVERAGE =";Q/A
 104 STOP
 140 END
READY
RUN
HOW MANY STUDENTS, HOW MANY GRADES PER STUDENT? 3,4
STUDENT NUMBER = Ø
ENTER GRADES
? 80
? 86
? 90
? 88
AVERAGE GRADE = 86
STUDENT NUMBER = 1
ENTER GRADES
? 90
? 91
? 95
? 100
AVERAGE GRADE = 94.25
STUDENT NUMBER = 2
EITER GRADES
? 78
? 77
? 73
? 55
ATERAGE GRADE = 70.75
```

CLASS AVERAGE = 83.56667

Everything following REM is ignored by the computer. In Figure 2-1, line 10 is a remark describing what the program does. It is often useful to put the name of the program and information on what the program does at the beginning for future reference. Remarks throughout the body of a long program will help later debugging by explaining the function of each section of code within the whole program.

2.2.3 Numbers

In BASIC, as in all languages, there are conventions to be learned. The most important initial concepts are (1) how do we express a number to the computer and (2) how do we create algebraic symbols.

BASIC treats all numbers as decimal numbers, which is to say that it assumes a decimal point after an integer, or accepts any number containing a decimal point. The advantage of treating all numbers as decimal numbers is that the programmer can use any number or symbol in any mathematical expression, knowing that the computer can combine the numbers given. (In some languages integers must be used separately from decimal numbers.)

A third form (other than integers and real numbers) we use in expressing numbers to the computer is called exponential form. In this form a number is expressed as a decimal number times some power of 10. For example:

23.4E2 = 2340

The E can be read as "times 10 to the--power" depending upon the positive or negative integer following E. A number can be expressed in exponential form by the programmer anywhere in his program. You may input data in any form. Results of computations are printed out as decimal numbers if they are in the range $.01 \le 1.000,000$. Outside this range numbers are automatically printed out in E format. The computer handles seven significant digits in normal operation and input/output, as seen below:

| Value Typed In | Value Typed Out by BASIC |
|----------------|--------------------------|
| .01 | .01 |
| .0099 | 9.90000E-3 |
| 999999 | 999999 |
| 1000000 | 1.00000E+6 |

The computer automatically omits printing leading and trailing zeros in integer and decimal numbers and formats all exponential numbers in the following form:

(sign) digit . six digits $\mathbf{E} \stackrel{\pm}{=} \mathbf{exponent}$ value For example:

- -3.470218E+8 is equal to -347,021,800
 - 7.260000E-4 is equal to .000726

All letters are printed as capitals at the Teletype console. Therefore, a convention used by programmers, and which occurs on most Teletype output, is that to distinguish zeros from the letter "oh" we slash zeros (\emptyset). This enables accurate input to the computer (when you are typing a program previously written down) and ease of understanding in reading computer output (in which zeros

are all slashed). All examples in this manual show the zero as an "\(\beta " \) and the letter "oh" as an "O". Not all teletypewriters follow this convention; some slash the letter "oh" and not the zero, and some print both with no slash. In all cases however, the computer will recognize the key next to the letter P as an "oh" and the key next to the number 9 as a zero. Notice that unlike a typewriter, the letter "el" does not produce the number one (1) on the console keyboard. All numbers are on the top row of the keyboard. Notice also that BASIC will not insert commas into large numbers, as we are accustomed to doing (i.e., 1,742,300 is printed as 1742300).

2.2.4 Variables

A variable in BASIC is an algebraic symbol for a number, and is formed by a single letter or a letter followed by a digit. For example:

| Acceptable Variables | Unacceptable Variables |
|----------------------|---|
| I | 2Ca digit cannot begin |
| в3 | a variable |
| x | ABtwo or more letters cannot form a variable |

We assign values to variables by either inputting these values or indicating them in a LET statement.

2.2.5LET Statement

tells the computer to add the values of B and C and store the result in a variable called A (The number 10 is the line number mentioned earlier).

 $2\emptyset$ LET D = 7.2

means to store the value 7.2 in the variable D.

 $3\emptyset$ LET D = $4\emptyset6$

causes the value of D which was 7.2 (above) to be changed to 406.

The equal sign means replacement rather than equality. In algebra the formula:

$$X = X + 1$$

is meaningless, but when we say:

 $1\emptyset$ LET X = X + 1

we mean "add one to the current value of X and store the result back in the variable X."

Values of variables can be reassigned throughout the program as the programmer wishes. The equal sign, then shows a replacement relationship where the expression after the equal sign is evaluated and replaces the old value (if any) of the variable indicated.

The LET statement is of the form:

(line number) LET (variable) = (formula)

where a formula is either a number, another variable, or an arithmetic expression. The LET statement is the most elementary BASIC statement, used when computation is to be performed or, to put it more generally, whenever a new value is assigned to a variable.

- All of the above is true for standard BASIC. TSS/8 BASIC
- 8 permits the expression:
- $10 \ X = X + 1$
- 8 to be interpreted as the same expression as:
- 8 100 LET X = X + 1
- 8 This simpler form is allowed to permit students familiar with
- 8 the FORTRAN language to make the conversion to BASIC more easily.

2.2.6 Arithmetic Operations

Looking at the console keyboard we can find some of the usual arithmetic symbols (+, -, and =). BASIC can perform addition, subtraction, multiplication, division and exponentiation as well as other more complicated operations explained later. Each mathematical formula fed to the computer must be on a single line, with a line number and an appropriate command. The five operators used in writing most formulas are:

| Symbol | | |
|----------|---|--------------|
| Operator | Meaning | Example |
| + | Addition | A + B |
| - | Subtraction | A - B |
| * | Multiplication | A * B |
| / | Division | А / В |
| ↑ | Exponentiation (Raise A to the Bth power) | A ↑ B |

In BASIC, the mathematical formula:

$$A = 7 \left[\frac{B^2 + 4}{X} \right]$$

would be written:

$$10 \text{ LET A} = 7 * ((B^2 + 4)/X)$$

How does the computer know what operation to perform first?

There are conventions built into computer languages; BASIC performs arithmetic operations with the order of evaluation indicated below:

- Parentheses receive the top priority. Any expression within parentheses is evaluated before an unparenthesized expression.
- 2. In absence of parentheses the order of priority is:
 - a. Exponentiation
 - b. Multiplication and Division
 - c. Addition and Subtraction
- 3. If 1 or 2 does not clear ambiguity, the order of evaluation is from left to right as we would read the formula.

So in the example above, B^{1} 2 is evaluated first, then $(B^{1}+4)$ and then $(B^{1}+4)/X$, finally $7*(B^{1}+4)/X$. Keeping the conventions above in mind, $A^{1}B^{1}$ C will be evaluated as $(A^{1}B)^{1}$ C, likewise A/B^{2} C is evaluated as $(A/B)^{2}$ C.

2.2.7 Parentheses and Spaces

Use of parentheses allows us to change the order of priority of evaluation in rule 2 above. They also prevent any
confusion or doubt on our part as to how the expression is
evaluated. To make a formula easier to write as well as read,
it is frequently a good idea to provide more parentheses than
strictly required. For example, which is easier to read:

```
A*B^{\dagger}2/7 + B/C*D^{\dagger}2
(A*B^{\dagger}2)/7 + (B/C)*D^{\dagger}2
((A*B^{\dagger}2)/7) + ((B/C)*D^{\dagger}2)
(((A*(B^{\dagger}2))/7) + ((B/C)*(D^{\dagger}2)))
```

Each of the above formulas will be executed the same way, but which makes the most sense to the programmer reading it, or perhaps trying to make corrections later? On the other hand, which has superflous parentheses not required for clarity?

Spaces may also be used freely to make formulas easier to read.

 $1\emptyset$ LET B = D \uparrow 2 +1

instead of:

1ØLETB=D↑2+1

2.2.8 Functions

BASIC performs several mathematical calculations for the programmer, eliminating the need for tables of trig functions, square roots and logarithms. These functions have a three letter call name, (the argument X can be a number, variable, formula, or another function) and are written as follows:

| <u>Functions</u> | Meaning |
|--------------------|---|
| SIN(X) | Sine of X (where X is expressed in radians) is returned. |
| COS(X) | Cosine of X (where X is expressed in radians) is returned. |
| TAN(X) | Tangent of X (where X is expressed in radians) is returned. |
| ATN(X) | Arctangent of X is returned as an angle in radians. |
| EXP(X) | ex (where e=2.712818) is returned. |
| LOG(X) | Natural logarithm of X, log X, is returned. |
| ABS (X) SQR (X) | Absolute value of X, $ X $, is returned. Square root of X, \sqrt{X} , is returned. |

These functions are built into BASIC and can be used in any statement as part of a formula. For example:

 $1\emptyset$ LET A = SIN(ABS(X))/2

will cause A to be set equal to one half the value of the sine of the absolute value of X.

2.2.9 More complex functions

Other functions are available, and although they are not as readily useful to the beginning programmer, they will become so as skill in designing program logic increases.

2.2.10 Sign Function, SGN(X)

The sign function returns the value +1 if X is a positive value, 0 if X is 0, and -1 if X is negative. For example: SGN(3.42)=1, SGN(-42)=-1, and SGN(23-23)=0.

```
LIST

12 REM- SGN FUNCTION EXAMPLE

22 READ A,B

25 PRINT "A="A,"B="B

32 PRINT "SGN(A)="SGN(A),"SGN(B)="SGN(B)

42 PRINT "SGN(INT(A))="SGN(INT(A))

53 DATA -7.32,.44

60 END

READY

PUN

A=-7.32 B= .44

SGN(A)=-1 SGN(B)= 1

SGN(INT(A))=-1

READY
```

2.2.11 Integer Function, INT(X)

The integer function returns the value of the greatest integer not greater than X. For example INT(34.67) = 34. INT can be used to round numbers to the nearest integer by asking for INT(X+.5). For example: INT(34.67+.5) = 35. INT can also be used to round to any given decimal place, by asking for INT(X*10 \uparrow D+.5)/10 \uparrow D

where D is the number of decimal places desired, as in the fol-

lowing program:

```
10 REM- INT FUNCTION EXAMPLE
20 PRINT "NUMBER TO BE ROUNDED";
30 INPUT A
40 PRINT "NO. OF DECIMAL PLACES";
50 INPUT D
60 LET B=INT(A*10+D+.5)/10+D
70 PRINT A"ROUNDED IS"B
80 END

RUN

NUMBER TO BE ROUNDED? 45.55
NO. OF DECIMAL PLACES? 1
45.55 ROUNDED IS 45.5
```

For negative numbers the largest integer contained in the number is a negative number with the same or a larger absolute value. For example: INT(-23) = -23, but INT(-14.39) = -15.

2.2.12 Random Number Function, RND(X)

The random number function produces a random number between 0 and 1. The numbers are reproducible in the same order for later checking of a program. The argument X in the RND(X) function call can be any number, as that value is ignored and serves no function.

```
10 REM-RANDOM NUMBER EXAMPLE
```

RUN

RANDOM NUMBERS.

| .2988412 | • 7 295008 | ·3125257 | •3ø95365 |
|-----------|--|---|---|
| • 4334217 | .4961024 | .5010025 | .04103271 |
| -3045837 | •1923863 | •9121199 | .241212 |
| •2537987 | ·Ø3323139 | .871119 | •9248194 |
| ·3Ø56748 | •4531268 | •4677783 | ·2285 317 |
| •6601784 | •7607476 | .1230533 | •3916202 |
| | •4334217 •3043837 •2537987 •3056748 | .4834217 .4961024 .3045837 .1923853 .2587987 .03323139 .3056748 .4531268 | .4334217 .4961024 .5010025 .3045837 .1923853 .9121199 .2537987 .03323139 .871119 .3056748 .4531268 .4677783 |

READY

In order to obtain random digits from 0 to 9, change line 40 to read:

$4\emptyset$ PRINT INT($1\emptyset*RND(\emptyset)$);

and tell BASIC to run the program again. This time the results will look as follows:

²⁰ PRINT "RANDOM NUMBERS."

 $^{3\}emptyset$ FOR I = 1 TO $3\emptyset$

^{40&#}x27; PRINT RND(Ø),

⁵⁰ NEXT I

⁵⁰ END

RUN

| RANDOM | NUMBERS. | | | |
|--------|----------|---|-----|---|
| 2 | 2 | 7 | 3 | 3 |
| Ø | 4 | 4 | 5 | Ø |
| 2 | 3 | I | 9 | 2 |
| 9 | 2 | 2 | . 3 | 9 |
| 8 | 3 | 4 | 4 | 2 |
| 7 | 6 | 7 | 1 | 3 |

READY

It is possible to generate random numbers over any range. For example, if the range (A,B) is desired, use:

 $(B-A)*RND(\emptyset)+A$

to produce a random number "n" in the range A(n(B.

2.2.13 RANDOMIZE statement

- 8 If you want the random number generator to calculate dif-
- 8 ferent random numbers every time the program is run, BASIC pro-
- 8 vides the RANDOMIZE statement. RANDOMIZE is normally placed at
- 8 the beginning of a program which uses random numbers (the RND
- 8 function). When executed, RANDOMIZE causes the RND function to
- 8 choose a random starting value, so that the same program run
- 8 twice will give different results.
- 8 For example:
- 8 10 RANDOMIZE
- 8 $2\emptyset$ PRINT RND(\emptyset)
- 8 3Ø END
- 8 will print a different number each time it is run. For this
- 8 reason, it is a good practice to debug a program completely be-
- 8 for inserting the RANDOMIZE statement. (RANDOMIZE uses the low

- 8 order 12 bits of the time of day as a starting value, thus there
- 8 are 4096 distinct starting points.)

The form of the statement is as follows:

(line number) RANDOMIZE

8 or (line number) RANDOM (abbreviated form)

To demonstrate the effect of the RANDOMIZE statement on two runs of the same program, we insert the RANDOMIZE statement as statement 15 below:

```
15 RANDOM
20 FOR I = 1 TO 5
30 PRINT "VALUE" I "IS" RND(0)
40 NEXT I
50 END
RUN
```

VALUE 1 IS .9648481 VALUE 2 IS .4638802 VALUE 3 IS .224618 VALUE 4 IS .7978773 VALUE 5 IS .7656411

READY

RUN

VALUE 1 IS .9902387 VALUE 2 IS .5400521 VALUE 3 IS .2031336 VALUE 4 IS .7334242 VALUE 5 IS .5722818

READY

Clearly, the output from each run is different.

- 8 2.2.14 The Time Function TIM(X) (TSS/8 BASIC only)
- A function called the TIM function is available to provide
- 8 several housekeeping functions dealing with the time of execu-
- 8 tion of a TSS/8 BASIC program as well as the account number,
- 8 the keyboard number and the JOB number of a user.
- 8 The function is called like this:
- 8 $l \emptyset LET R = TIM (N)$
- 8 The following list gives the result (R) for each valid parameter
- 8 to TIM (N):
- 8 <u>N</u> <u>R (information returned)</u>
- 8 \emptyset The CPU time used by this job since LOGIN (seconds)
 - $\frac{1}{2}$ The current month (1-12)
- 8 2 The day of the month (1-31)
- 3 The last two digits of the year (e.g. 72)
- 8 4 The number of the keyboard currently in use
- 5 The current user's account number
 6 The current time (in minutes page
 - 6 The current time (in minutes past midnight)
 - 7 The current job number
- 8 Although the keyboard number, account number, and job number are
- 8 kept and printed in octal by the system, the TIM function returns
- 8 them as decimal numbers such that, when printed, they will print
- 8 as if they were octal. For example, account number 7260 (octal)
- 8 will be returned by TIM (5) as a decimal 726%, so that it may
- 8 be printed with no further conversion. If the parameter in TIM is
- 8 negative or greater than 7, the error message
- 8 RB IN 1Ø
- 8 appears, indicating the function was referenced badly; the func-
- 8 tion returns a zero and execution continues normally.
- 8 What follows is a demonstration of this function:

```
10 REM - DEMONSTRATION OF TIM(0)
20 LET U = TIM(0)
30 LET T1 = INT(U/60)
40 LET T2 = INT((V-T1*60)*1000+.05)/1000
50 PRINT "CPU TIME USED SO FAR IS "T1"MINUTES AND"T2"SECONDS"
60 END
RUN

CPU TIME USED SO FAR IS 0 MINUTES AND 1.35 SECONDS
```

READY

- Note that to implement the "TIM" function, the function graph "FIX" had to be removed. This function may be used (with a
- 8 different name) by including this "DEF" statement in your pro-
- 8 gram:
- 8 10 DEF FNT(X)=SGN(X)*INT(ABS(X))

2.2.15 User defined functions

In some programs it may be necessary to execute the same mathematical formula in several different places. BASIC allows the programmer to define his own functions and call these functions in the same way he would call the square root or trig functions.

These user defined functions consist of a three-letter function name, the first two letters of which should be FN.

We define the function once at the beginning of the program before its first use. The defining or DEF statement is formed as follows:

(line number) DEF FNA(X) = formula(X)

where A may be any letter. The argument (X) must be the same on each side of the equal sign and may consist of one or more variables. For example:

 $1\emptyset$ DEF FNA(S) = S \uparrow 2

will cause a later statement:

 $2\emptyset$ LET R = FNA(4)+1

to be evaluated as R=17.

The two following programs

Program #1:

- 10 DEF FNS(A)=A+A
- 20 FOR I = 1 TO 5
- 30 PRINT LIFNS(I)
- 40 NEXT I
- 50 END

READY

Program #2:

- 10 DEF FNS(X)=X+X
- 20 FOR I = 1 TO 5
- 30 PRINT I, FNS(I)
- 40 NEXT I
- 50 END

READY

both cause the same output:

RUN

| 1 | 1 |
|---|------------|
| 2 | 4 |
| 3 | 2 7 |
| 4 | 253 |
| 5 | 3125 |

READY

The argument in the DEF statement can be seen to have no significance; it is strictly a dummy variable. The function itself can be defined in the DEF statement in terms of numbers, variables, other functions or mathematical expressions. For example:

 $1\emptyset$ DEF FNA(X) = $X\uparrow 2+3*X+4$

15 DEF FNB(X) = FNA(X)/2 + FNA(X)

 $2\emptyset$ DEF FNC(X) = SQR(X+4) + 1

The statement in which the user defined function appears may have that function combined with numbers, variables, other functions or mathematical expressions. For example:

 $4\emptyset$ LET R = FNA(X+Y+Z)

The user defined function can be a function of more than one variable, as shown below:

25 DEF FNL(X,Y,Z) = SQR($X^2 + Y^2 + Z^2$)

A later statement in a program containing the above user defined function might look like the following:

55 LET B = FNL(D,L,R)

where D, L, and R have some values in the program.

The program in Figure 2-2 contains examples of a multivariable DEF statement in lines 11, 21, and 31.

```
LIST
  1 REM MODULUS ARITHMETIC
10 REM FIND X MOD M
  11 DEF FNM(X,M) = X - M*INT(X/M)
  20 REM FIND A+B MOD M
  21 DEF FNA(A,B,M) = FNM(A+B,M)
  30 REM FIND A*B MOD M
  31 DEF FNB(A,B,M) = FNM(A*B,M)
 100 PRINT "ADDITION AND MULTIPLICATION TABLES MOD M
 110 PRINT "ENTER VALUE OF M";
 120 INPUT M
 130 PRINT
 140 PRINT "ADDITION TABLE MOD"M
 150 GOSUB 800
 200 FOR I = 0 TO M-1
 205 PRINT I;" ";
 210 \text{ FOR } J = 0 \text{ TO M-1}
 220 PRINT FNA (I,J,M);
 230 NEXT J
 240 PRINT
 250 NEXT I
 260 PRINT
 27Ø PRINT
 280 PRINT "MULTIPLICATION TABLES MOD"M
 290 GOSUB 800
 300 \text{ FOR I} = 0 \text{ TO M-I}
 3Ø5 PRINT I;" ";
 310 FOR J = 0 TO M-1
 320 PRINT FNB(I,J,M);
 330 NEXT J
 340 PRINT
 350 NEXT I
 35Ø STOP
 800 PRINT
 810 PRINT TAB(5);0;
 820 \text{ FOR I} = 1 \text{ TO M-1}
 830 PRINT I;
 840 NEXT I
 85Ø PRINT
 850 \text{ FOR I} = 1 \text{ TO } 3*M+5
 870 PRINT "-";
 880 NEXT I
 890 PRINT
 900 RETURN
1000 END
```

READY

Figure 2-2A

RUN

ADDITION AND MULTIPLICATION TABLES MOD M ENTER VALUE OF M? 5

ADDITION TABLE MOD 5

| | Ø | | 2 | 3 | 4 |
|---|---|---|---|---|---|
| Ø | Ø | 1 | 2 | 3 | 4 |
| 1 | 1 | 2 | 3 | 4 | Ø |
| 2 | 2 | 3 | 4 | Ø | 1 |
| 3 | 3 | 4 | Ø | 1 | 2 |
| 4 | 4 | Ø | 1 | 2 | 3 |

MULTIPLICATION TABLES MOD 5

| | Ø | 1 | 2 | 3 | 4 | |
|---|---|---|---|---|---|--|
| ø | Ø | ø | Ø | Ø | z | |
| 1 | | | 2 | | 4 | |
| 2 | Ø | 2 | 4 | 1 | 3 | |
| 3 | Ø | 3 | 1 | 4 | 2 | |
| 4 | Ø | 4 | 3 | 2 | i | |

READY

Figure 2-2B

2.2.16 INPUT/OUTPUT statements

One of the most important groups of statements is the group of I/O (Input/Output) statements. These I/O statements allow us to bring data into our programs during execution when and from where we choose. Similarly, we can choose the output format which best suits our needs. In the case of the example programs in Figure 2-1 (at the beginning of the chapter), data was typed in at the console keyboard as the computer requested it.

2.2.17 READ Statement

A simple way to put data into a program is with READ and DATA statements. One statement is never used without the other. The READ statement is of the form:

(line number) READ (variables separated by commas) For example:

10 READ A,B,C

where A, B, and C are the variables we wish to assign values. In order to assure that all variables are assigned values before computation begins, READ statements are usually placed at the beginning of a program, or at least before the point where the value is required for some computation.

2.2.18 DATA Statement

Now that we have told the computer to read the values for three variables, we must supply those values in a DATA statement of the form:

(line number) DATA (numeric values separated by commas)
For example:

7Ø DATA 1,2,3

The DATA statement provides the values for the variables in the READ statement(s). The values must be separated by commas, in the same order as the variables are listed in the READ statement. Thus at execution time A=1, B=2, and C=3 according to the two lines above.

The DATA statement is usually placed at the end of a program before the END statement, so as to be easily accessible to the programmer should he wish to change his values.

A given READ statement may have more or fewer variables than there are values in any one DATA statement. READ causes BASIC to search all available DATA statements, in the order of their line numbers until values are found for each variable. A second READ statement will begin reading values where the first stopped. If at some point in your program you attempt to read data which is not present or if your data is not separated by commas, BASIC will stop and print an OUT OF DATA IN LINE XXXX message at the console, indicating the line which caused the error.

2.2.19 RESTORE Statement

If it should become necessary to use the same data more than once in a program, the RESTORE statement will make it possible to recycle through the DATA statements beginning with the

lowest numbered DATA statement. The RESTORE statement is of the form:

(line number) RESTORE

For example:

85 RESTORE

READY

will cause the next READ statement following line 85 to begin reading data from the first DATA statement in the program, regardless of where the last data value was found.

You may use the same variable names the second time through the data or not as you choose, since the values are being read as though for the first time. In order to skip unwanted values dummy variables must be read. In the following example, BASIC prints:

4 1 2 3

on the last line because it did not skip the value for the original N when it executed the loop beginning at line 210.

```
10 REM-PROGRAM TO ILLUSTRATE USE OF RESTORE
  20 READ N
  25 PRINT "YALUES OF X ARE:"
  30 \text{ FOR I} = 1 \text{ TO N}
  40 READ X
  50 PRINT X,
  30 NEXT I
  70 RESTORE
 185 PRINT
 190 PRINT "LIST OF X VALUES FOLLOWING REGTORE STATEMENT:"
 210 \text{ FOR I} = 1 \text{ TO N}
 220 READ X
 230 PRINT X
 240 NEXT I
 250 DATA 4,1,2
 250 DATA 3,4
 270 END
RUN
VALUES OF X ARE:
LIST OF X VALUES FOLLOWING RESTORE STATEMENT:
                1
                                    -2.28-
```

2.2.20 INPUT Statement

The second way to input data to a program is with an INPUT statement. This statement is used when writing a program to process data to be supplied while the program is running. The programmer types in the values as the computer asks for them. Depending upon how many values are to be brought in by the INPUT command, the programmer may wish to write himself a note reminding himself what data is to be typed in at what time. In the example program in Figure 2-3, the question is asked at execution time "INTEREST IN PERCENT?", "AMOUNT OF LOAN?", and "NUMBER OF YEARS?" The programmer knows which value is requested and proceeds to type and enter the appropriate number.

The INPUT statement is of the form:

(line number) INPUT (variables separated by commas)
For example:

10 INPUT A,B,C

will cause the computer to pause during execution, print a question mark and wait for the user to type in three numerical values separated by commas and entered to the computer by hitting the RETURN key at the end of the list.

As you will notice in Figure 2-3, the question mark is grammatically useful if you care to formulate a verbal question which the input value will answer. This will be further explained in the section on the PRINT Statement.

```
10 REM PROGRAM TO COMPUTE INTEREST PAYMENTS
 20 PRINT "INTEREST IN PERCENT";
 25 INPUT J
 26 LET J = J/100
 30 PRINT "AMOUNT OF LOAN";
 35 INPUT A
 40 PRINT "NUMBER OF YEARS";
 45 INPUT N
 50 PRINT "NUMBER OF PAYMENTS PER YEAR";
 55 INPUT M
 6Ø LET N = M * N
 55 LET I = J/M
 70 \text{ LET B} = 1 + 1
 75 LET R = A*I/(1-I/B*N)
 78 PRINT
 80 PRINT "AMOUNT PER PAYMENT =";R
 85 PRINT "TOTAL INTEREST =";R*N-A
88 PRINT
 90 LET B = A
 95 PRINT "INTEREST APP TO PRIN BALANCE"
100 \text{ LET L} = B * I
110 LET P = R - L
120 LET B = B - P
130 PRINT L.P.B
140 IF B>=R GO TO 100
150 PRINT B*I, R-B*I
160 PRINT "LAST PAYMENT =" B*I+B
SAM END
```

RUN

INTEREST IN PERCENT? 9
AMOUNT OF LOAN? 2500
NUMBER OF YEARS? 2
NUMBER OF PAYMENTS PER YEAR? 4

AMOUNT PER PAYMENT = 344.9617 TOTAL INTEREST = 259.6932

| APP TO PRIN | BALANCE |
|-------------|---|
| 288.7117 | 2211.288 |
| 295.2077 | 1916.081 |
| 301.8498 | 1614.231 |
| 308.6415 | 1305.589 |
| 315.5859 | 990.0035 |
| 322.6866 | 667.317 |
| 329.947 | 337.3699 |
| 337.3708 | |
| 344.9508 | |
| | 295.2077 301.8498 308.6415 315.5859 322.6866 329.947 337.3708 |

READY

The output for the program begins after the word RUN and includes a verbal description of the numbers. This verbal description on the output is optional with the programmer, although it has a definite advantage in ease of use and understanding.

Only one question mark is printed per INPUT statement, so the programmer must be careful to insert the correct number of variables at that point, separating them by commas if more than one are to be typed. When the correct number of variables have been typed, hit the RETURN key to enter them to the computer.

- 8 If too few values are listed, the message:
- 8 MORE?
- 8 will appear. If too many values are typed, the message:
- 8 TOO MUCH INPUT, EXCESS IGNORED
- 8 will be given.

2.2.21 PRINT Statement

The PRINT statement is the output statement for BASIC. Depending upon what follows the PRINT command, we can create numerous different output formats and even plot points on a graph.

In order to skip a line on the output sheet, type only a line number and the command PRINT:

1Ø PRINT

When the computer comes to line 10 during the run, the paper on the console will be advanced by one line. In the example program in Figure 2-3, line 78 causes a blank line on the output sheet between the section where the user enters data to the computer

and the section where the computer supplies the results of the program.

In order to have the computer print out the results of a computation, or the value of any variable at any point in the program, the user types the line number, the command PRINT, and the variable names separated by commas:

10 PRINT A,C+B,SQR (A)

This will cause the values of A, C + B, and the square root of A to be printed in the first three of the five fixed format columns (of 14 spaces each) which BASIC uses for most output. For example the statement:

10 PRINT A,B,C,D,E

will cause the values of the variables to be printed like this:
12.3 12.3 12.3 12.3

where A, B, C, D, E equal 12.3. When more than five variables are listed in the PRINT statement and separated by commas, the sixth value begins a new line of output.

The third possibility for the PRINT statement is to print out a message, or some text. The user may ask that any message be printed by placing the message in quotation marks. For example:

10 PRINT "THIS IS A TEST"

when line 10 above is encountered during execution the following will be printed:

THIS IS A TEST

(Going back to the example program in Figure 2-3, notice the function of lines 80, 85, and 160.)

Looking at Figure 2-3 shows that the PRINT statement can combine the second and third options. One PRINT command tells the computer to print:

AMOUNT PER PAYMENT = 344.9617

The command which did this was line 80:

8Ø PRINT "AMOUNT PER PAYMENT ="; R

It is not necessary to use the standard five column format for output. A semi-colon (;) will cause the following text or data to be printed following the last character of text or data printed. A comma (,) will cause a jump to the next of the five output format columns. BASIC allows the user to omit format control characters (,) or (;) between text and data, and assumes a semi-colon. For example:

8Ø PRINT "AMOUNT PER PAYMENT =" R

will result in the same output as line 80 above.

In addition to the capabilities already mentioned, the PRINT statement can also cause a constant to be printed at the console. For example:

10 PRINT 1.234, SQR(100/4)

will cause the following to be printed at execution time:

1.234 5

Any number present in a PRINT statement will be printed exactly as shown. Any algebraic expression in a PRINT statement will be evaluated with the current value of the variables and the result printed.

In Figure 12-3, line 160 reads:

16Ø PRINT "LAST PAYMENT =" B*I+B

and caused the following to be printed upon execution:

LAST PAYMENT = 344.9608

This demonstrates the omission of the format control character as well as the ability of the PRINT statement to print text and do calculations.

The following example program illustrates the use of the control characters in PRINT statements:

- 10 READ A.B.C
- 20 PRINT A,B,C,A+2,B+2,C+2
- 30 PRINT
- 4Ø PRINT A;B;C;A+2;B+2;C+2
- 50 DATA 4,5,6
- SØ END

RIJN

4 5 6 16 25 35

4 5 6 16 25 36

READY

If a number should happen to be too long to be printed on the end of a single line, BASIC automatically moves the number entirely to the beginning of the next line.

2.2.22 TAB Function

- 8 When using the PRINT statement thus far, we have had to
- 8 print a blank character wherever we wanted blank space; there
- 8 was no real control over printing. The TAB function is a more

- 8 sophisticated technique allowing the user to position the print-
- 8 ing of characters anywhere on the Teletype paper line. This line
- 8 is 72 characters long, and the print positions can be thought of
- 8 as being numbered from 0 to 71, going from left to right. The
- 8 TAB function argument can be positive or negative: TAB(-1)
- 8 causes a tab over to position 71, TAB(3) causes a tab to posi-
- 8 tion 3. (The TAB function can be thought of as operating mod-
- 8 ulo 72.)
- 8 After performing TAB(n), the next character to be printed
- 8 will be placed in position n. If n is a position to the left
- 8 of the current position, a carriage return (without a line feed)
- 8 is used to correctly position the printing head.
- 8 For example:
- 8 1Ø PRINT "X =";TAB(2);"/";3.14159
- 8 will print the slash on top of the equal sign, as shown below:
- $8 \quad x \neq 3.14159$
- 8 Figure 2-4 is an example of the sort of graph that can be
- 8 drawn with BASIC using the TAB function:

```
30 FOR X = 0 TO 15 STEP .5

40 PRINT TAB(30+15*SIN(X)*EXP(-.1*X));"*"

50 NEXT X

60 END
```

RUN

READY

Figure 2-4

2.3 SUBSCRIPTS AND LOOPS

2.3.1 Subscripted Variables

In addition to simple variable names, there is a second class of variables which BASIC accepts called subscripted variables. Subscripted variables provide the programmer with additional computing capabilities for dealing with lists, tables, matrices, or any set of related variables. In BASIC, variables are allowed one or two subscripts. A single letter (or letter followed by a number) forms the name of the variable followed by one or two integers in parentheses, separated by commas, indicating the place of that variable in the list. You can have several arrays in a program subject only to the amount of core space available for data storage. For example, a list might be described as A(I) where I goes from 1 to 5 as shown below:

This allows the programmer to reference each of the five elements in the list A. A two dimensional matrix J (I,J) can be defined in a similar manner, but the subscripted variable J can be used only once. A(I) and A(I,J) cannot be used in the same program.

It is possible, however, to use the same variable name as both a subscripted and as an unsubscripted variable. Both B and B (J) are valid variable names and can be used in the same program.

Input can be done easily using subscripted variables, as shown in Figure 2-5.

*TSS/8 BASIC

```
10 REM - PROGRAM DEMONSTRATING READING OF
  11 REM - SUBSCRIPTED YARIABLES
  15 DIM A(5),B(2,3)
  18 PRINT "A(I) WHERE A = 1 TO 5:"
  20 \text{ FOR I} = 1 \text{ TO } 5
  25 READ A(I)
  3Ø PRINT A(I);
  35 NEXT I
  38 PRINT
  39 PRINT
  4\emptyset PRINT "B(I,J) WHERE I = 1 TO 2"
  41 PRINT " AND J = 1 TO 3:"
  42 \text{ FOR I} = 1 \text{ TO } 2
  43 PRINT
  44 \text{ FOR } J = 1 \text{ TO } 3
  48 READ B(IJ)
  50 PRINT B(I,J);
  55 NEXT J
  56 NEXT I
  60 DATA 1,2,3,4,5,6,7,8
  31 DATA 8,7,5,5,4,3,2,1
  55 END
RUN
A(I) WHERE A = 1 TO 5:
1 2 3 4 5
B(I,J) WHERE I = 1 TO 2
        AND J = 1 TO 3:
 3 7 5
READY
```

Figure 2-5

2.3.2 DIM Statement

As in the preceding examples, we see that the use of subscripts requires a dimension (DIM) statement to define the maximum number of elements in the array. The DIM statement is of the form:

(line number) DIM
$$v_1(n_1), v_2(n_2, m_2)$$

where v_{11} indicates an array variable name and n and m are integer numbers indicating the largest subscript value required during the program. For example:

$1\emptyset$ DIM A(6,1 \emptyset)

The first element of every array is automatically assumed to have a subscript of zero. Dimensioning A(6,10) sets up room for an array with 7 rows and 11 columns. This matrix can be thought of as existing in the following form:

as shown in the Figure 2-6.

```
10 REM - MATRIX CHECKING PROGRAM.

15 DIM A(5,10)

20 FOR I = 0 TO 6

22 LET A (I,0)=I

25 FOR J = 0 TO 10

28 LET A(0,J) = J

30 PRINT A(I,J);

35 NEXT J

40 PRINT

45 NEXT I

50 END
```

RUN

```
10
Ø
   1
      2
          3
            4
               5
                  5
   Ø
      Ø
               Ø
                   Ø
1
          Ø
               Ø
2
   Ø
      Ø
          Ø
             Ø
                   Ø
                       Ø
                          D
                             Ø
                                 Ø
3
      Ø
         Ø
                          Ø
4
   Ø
      Ø
         Ø
             Ø
                Ø
                   Ø
                       Ø
                                 Ø
5
      Ø
         Ø
             Ø
                       Ø
6E
         Ø
```

DEADY

Figure 2-6

(Notice that a BASIC variable has a value of zero until it is assigned a value.)

If the user wishes to conserve core space, and not make use of the extra variables set up within the array, he should, for example, say DIM A(5, 9) which would result in a 6 by 10 array which would be referenced beginning with the A(0, 0) element.

You can define more than one array in a single DIM statement:

100 DIM A(200), B(4,7)

will dimension both the list A and the matrix B.

A number must be used to define the maximum size of the array. A variable inside the parentheses is not acceptable and would result in an error message by BASIC at RUN time. The amount of user core not filled by the program will determine the amount of data the computer can accept as input to the program at any one time. In some programs a PROGRAM TOO LARGE message may occur, indicating that core will not hold an array of the size requested. In that event, the user should change his program to process part of the data in one run and the rest later.

2.3.3 Loops

So far in this manual we have seen FOR and NEXT statements used several times in examples. These two statements define the beginning and end of a loop, where a loop is a set of instructions which modifies itself and repeats until some terminal condition is reached.

2.3.4 FOR STATEMENT

The FOR statement is of the form:

(line number) FOR (variable) = (formula) TO (formula)

STEP (formula)

For example:

 $1\emptyset$ FOR K=2 TO $2\emptyset$ STEP 2

which will iterate (cycle) through the designated loop using K as 2, 4, 6, 8, . . . ,20 in calculations involving K. When the value 20 is reached, the loop is left behind and the program goes to the line following the NEXT statement (described below).

The variable mentioned in the definition must be unsubscripted, although a common use of such loops is to deal with subscripted variables using the FOR variable as the subscript of a previously defined variable. The formulas mentioned in the definition can be real or integer numbers, variables, or expressions.

2.3.5 NEXT STATEMENT

The NEXT statement signals the end of the loop and at that point the computer adds the STEP value to the variable and checks to see if the variable is still less than the terminal value. When the variable exceeds the terminal value control falls through the loop to the following statement.

When control falls through the loop the variable value is one step greater than it was when the loop was last executed. For some programs this information may be useful.

If the STEP value is omitted, +1 is assumed. Since +1 is the usual STEP value, that portion of the statement is frequently omitted.

In the following example we see a demonstration of the last two paragraphs. The loop is executed 10 times, the value of I is 11 when control leaves the loop and +1 is the assumed STEP value.

```
10 FOR I = 1 TO 10
20 NEXT I
30 PRINT I
40 END
RUN
```

READY

11

If line 10 had been:

10 FOR I=10 TO 1 STEP -1

the value printed by the computer would be \$\mathcal{B}\$.

The numbers used in the FOR statement can be "formulas" as indicated earlier. A formula in this case can be a variable, a mathematical expression, or a numerical value.

The value of each formula is evaluated upon first encountering the loop. While the values of the variables, if any, used in evaluating these formulas can be changed within the loop, the values assigned in the FOR statement remain as they were initially defined.

In the last example program, the value of I (in line 10) can be successfully changed in the program. The loop:

 $1\emptyset$ FOR I=1 TO $1\emptyset$

15 LET I=1Ø

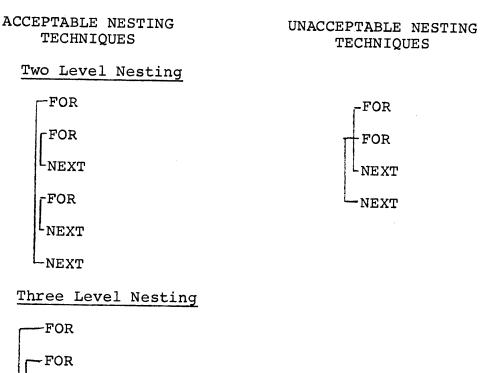
20 NEXT I

will only be executed once since the value 10 has been reached by the variable I and the termination condition is satisfied.

2.3.6 NESTING LOOPS

It is often useful to have one or more loops within a loop. This technique is called nesting. Nesting is allowed as long as

the field of one loop (the numbered lines from the FOR statement to the corresponding NEXT statement, inclusive) does not cross the field of another loop. A diagram is the best way to illustrate acceptable nesting procedures:



- A maximum of eight (8) levels of nesting is permitted.

 Exceeding that limit will result in a STACK OVERFLOW error
- 8 message.

-FOR

NEXT

-FOR

-NEXT

NEXT

NEXT

If the value of the counter variable is originally set equal to the terminal value, the loop will execute once, regardless of the STEP value. If the starting value is beyond the terminal value, the loop will not execute.

It is also possible to exit from a FOR-NEXT loop without the counter variable reaching the termination value. A conditional transfer may be used to leave a loop. Control may only transfer into a loop which had been left earlier without being completed, ensuring that the termination and STEP values are assigned.

2.4 TRANSFER OF CONTROL

Certain statements can cause the execution of a program to jump to a different line either unconditionally or depending upon some condition within the program. Looping is one method of jumping to a designated point until a condition is met. The following commands give the programmer additional capabilities in this area.

2.4.1 Unconditional Transfer

The GOTO statement is an unconditional command telling the computer to either jump ahead or back in the program. For example:

100 GOTO 50

or

24 GOTO 78

The GOTO statement is of the form:

(line number) GOTO (line number)

When the logic of the program reaches the GOTO statement, the statement(s) immediately following it will not be executed, but the statements beginning with the line number indicated are performed.

The program in Figure 2-7 never ends; it does a READ, prints something and attempts to do this over and over until it runs out of data, which is sometimes an acceptable, though not advisable, way to end a program:

```
10 REM - PROGRAM ENDS WITH ERROR MESSAGE
```

READY

RUN

| X | = | 1 | X†2 | = | 1 |
|---|---|-----|-----|---|-----|
| X | = | 5 | X12 | = | 25 |
| Х | = | 1 Ø | X†2 | = | 100 |
| Х | = | 15 | X†2 | = | 225 |
| X | = | 20 | X†2 | = | 400 |
| X | = | 25 | X12 | = | 325 |

OUT OF DATA IN LINE 20

READY

Figure 2-7

¹¹ REM - WHEN OUT OF DATA.

²⁰ READ X

²⁵ PRINT "X ="X,"X+2 ="X+2

³⁰ G (TO 20

³⁵ DATA 1,5,10,15,20,25

⁴⁰ END

2.4.2 Conditional Transfer

If a program requires that two values be compared at some point, logic may direct us to different procedures depending on the comparison. In computing, we logically test values to see whether they are equal, greater, or less than another value, or a possible combination of the three.

In order to compare values we use a group of mathematical symbols not discussed earlier. These symbols are as follows:

| BASIC SYMBOL | Math Symbol | BASIC 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 |

2.4.3 IF-THEN AND IF-GOTO

The IF-THEN and IF-GOTO statements both allow the programmer to test the relationship between two formulas (variables, numbers, or expressions). Providing the relationship we have described in the IF statement is true at that point, control will transfer to the line number indicated. The statements are of the form:

The use of the word THEN or GOTO is the programmer's choice. For example:

1Ø IF A=5 GOTO 7Ø

causes transfer from line 10 to line 70 if A is equal to 5. If A is not equal to 5, control passes to the next line of the program following line 10.

2.4.4 SUBROUTINES

When particular mathematical expressions are evaluated, several times throughout a program, the DEF statement enables the user to write that expression only once. The technique of looping allows the program to do a sequence of instructions a specified number of times. If the program should require that a sequence of instructions be executed several times in the course of the program, this too is possible. A subroutine is a section of code performing some operation that is required at more than one point in the program. Sometimes a complicated I/O operation for a volume of data, a mathematical evaluation which is too complex for a user defined function, or any number of other processes may be best performed in a subroutine.

2.4.5 GOSUB Statement

Subroutines may be placed physically anywhere in a program, and always before the END statement. The program begins execution and continues until it encounters a GOSUB statement of the form:

(line number) GOSUB (line number)
where the number after GOSUB is the first line number of the
subroutine. Control then transfers to that line in the subroutine. For example:

5Ø GOSUB 2ØØ

```
1 REM - THIS PROGRAM ILLUSTRATES GOSUB AND RETURN
 10 DEF FNA(X) = ABS(INT(X))
 20 INPUT A,B,C
 30 GOSUB 100
 40 LET A := FNA(A)
 50 LET B = FNA (B)
 50 LET C = FNA(C)
 70 PRINT
 80 GOSUB 100
 9Ø STOP
100 REM - THIS SUBROUTINE PRINTS THE SOLUTION
110 REM - OF THE EQUATION A*X+2+B*X+C=0
120 PRINT "THE EQUATION IS " A "* X+2 +" B "* X +" C
130 LET D=B+2 - 4*A*C
140 IF D <> 0 THEN 170
150 PRINT "THERE IS ONLY ONE SOLUTION ... X ="-B/2*A
160 RETURN
170 IF D<0 THEN 200
180 PRINT "THERE ARE TWO SOLUTIONS ..."
185 PRINT "X =" (-B+SQR(D))/(2*A) "AND"(-B-SQR(D))/(2*A)
190 RETURN
200 PRINT "THERE ARE 2 IMAGINARY SOLUTIONS"
205 \text{ PRINT "X} = ("-B/(2*A)","SQR(-D)/(2*A)")";
207 PRINT " AND ("-B/(2*A)","-SuR(-D)/(2*A)")."
210 RETURN
 900 END
DIIN
? 1,4,4.6
THE EQUATION IS 1 * x + 2 + 4 * x + 4.6
THERE ARE 2 IMAGINARY SOLUTIONS
X = (-2, .7745936) AND (-2, -.7745966).
```

THE EQUATION IS 1 * X+2 + 4 * X + 4
THERE IS ONLY ONE SOLUTION ... X =-2

READ

Figure 2-8

2.4.6 RETURN Statement

Having reached line 50, as shown above, control transfers to line 200; the subroutine is processed until the computer encounters a RETURN statement of the form:

(line number) RETURN

which causes control to return to the line following the GOSUB statement. (Before transferring to the subroutine, BASIC internally records the next line number to be processed after the GOSUB statement; the RETURN statement is a signal to transfer control to this line.) In this way, no matter how many subroutines or how many times they are called, BASIC always knows where to go next. The program in Figure 2-8 demonstrates a simple subroutine:

Lines 100 through 210 constitute the subroutine. The subroutine is executed from line 30 and again from line 80. When control returns to line 90, the program encounters the STOP statement and terminates execution. Note that even though the program logically ends with a STOP, the END command must still be present.

For another detailed example of a subroutine, see Figure 2-2.

8 2.4.7 LOCATION OF THE SUBROUTINE

- Normal practice in BASIC is to place the subroutines at
- 8 the end of the code before the DATA and END cards. Although
- 8 this is logically correct and good programming practice in BASIC,
- 8 the overlaying of TSS/8 BASIC is such as to make for slow execu-
- 8 tion of large programs when the subroutines called are located
- 8 far from the calling locations in the program. The problem is
- 8 not severe unless (1) the program is large, (2) there are sizeable
- 8 arrays, and (3) the subroutine in question is called repeatedly
- 8 from a location far from where it appears in the program.

2.4.8 STOP and END Statements

Either the STOP statement or the END statement may be used to terminate execution, but the END statement must be the last statement of the entire program. STOP may occur several times throughout the program. No BASIC program will run without an END statement of the form:

(line number) END

The format of the STOP statement is simply:

(line number) STOP

STOP is logically equivalent to a GOTO nn, where nn is the line number of the END statement.

2.4.9 Nesting Subroutines

More than one subroutine can be used in a single program in which case they can be placed one after another at the end of the program (in line number sequence). A useful practice is to assign distinctive line numbers to subroutines, for example if you have numbered the main program with line numbers up to 199, you could use 200 and 300 as the first numbers of two subroutines.

Subroutines can also be nested, in terms of one subroutine calling another subroutine. If the execution of a subroutine encounters a RETURN statement, it will return control to the line following the GOSUB which called that subroutine; therefore, a subroutine can call another subroutine, even itself. Subroutines can be entered at any point and have more than one RETURN statement where certain conditions will cause control to reach any one RETURN statement. It is possible to transfer to the beginning or any part of a subroutine; multiple entry points and RETURNs make a subroutine more versatile.

- 8 The maximum level of GOSUB nesting is about forty (40)
- 8 levels, which should prove more than adequate for all normal
- 8 uses. Exceeding this limit will result in the message:

GOSUB--RETURN ERROR

2.5 ERRORS AND HOW TO MAKE CORRECTIONS

2.5.1 Single Letter Corrections

Nobody being perfect, we all make typing errors if not logical errors. The first is by far the easier to correct. If you notice an error immediately as you type it, for example:

10 LEB

instead of LET as you meant to begin the line, hit the RUBOUT key or SHIFT/O (left pointing or back arrow) once for every character you wish to remove, including spaces. This will result in the printing (by BASIC) of a back arrow to show that the rubout has been accomplished. Make the correction and continue typing as shown below.

1Ø LEB←T A=1Ø*B

if that was the intended line. BASIC does not even see the mistake; it is erased, except on the console as you typed it. The typed line enters BASIC only when you hit the RETURN key. Before that time you can correct errors with the RUBOUT key or SHIFT/O. If you desire a neat, corrected listing at the end of your work, that is possible too. More on that later.

 $2\emptyset$ DEN F+++F FNA(X,Y)=X 2+3*Y

is the same as:

 $2\emptyset$ DEF FNA(X,Y)=X 2+3*Y

to BASIC. Notice you must erase spaces, as well as printing characters.

- 8 2.5.2 Erasing a Line
- 8 If at any time you have typed a line and not yet hit the
- 8 RETURN key, the line can be erased by striking the ALTMODE
- 8 (ESCAPE on some machines) key. TSS/8 BASIC will echo back:
- 8 \$ DELETED
- 8 at the end of the line to indicate that the entire line has
- 8 been removed.

Once you have hit the RETURN key and have entered a line into the computer it can still be corrected by simply typing the line number and proceeding to retype the line correctly. The old line is automatically deleted as you type the line again, even if it was longer than the new line.

You can delete an entire line by typing the line number and hitting the RETURN key. This removes the entire line and line number from your program.

NOTE

Typing a line number followed by back arrows does not erase the line identified with that number.

If you accidentally type the line number of a previous line you do not want erased, the RUBOUT key will remove the unwanted line number, leaving the original line intact. For example:

10 LET A=4

 $1\emptyset \leftarrow 2\emptyset$ LET B=A+7

will leave line 10 as it is and allow you to type line 20.

Following an attempt to run a program you may receive an error message. Most errors can be corrected by typing the line number, typing the line over again with the correction, and hitting RETURN. The program is then ready to be run again. You can make as many changes or corrections between runs as you wish. (For a more advanced technique in program editing, see Chapter 3.)

2.5.3 Erasing a Program in Core

Assuming you have written a program on-line in BASIC, have completed it and now wish to run another program in BASIC, but do not wish to save the old program, when BASIC types READY, answer:

SCRATCH

8 or SCR

8

- 8 (The 3 letter abbreviation SCR above is valid only for TSS/8
- 8 BASIC. All keyboard commands such as SCRATCH, LIST, REPLACE,
 - RENAME, may be abbreviated to 3 letters in TSS/8 BASIC.) The SCRATCH command will erase the old program and leave a fresh, blank area in which you can work. When erasure of the space is complete, BASIC will respond with the word READY. You go on from this point. The previous program name is maintained for the cleared area. You can, alternatively, reply to READY with NEW, if you wish to create a new program, or OLD, if you wish to recall a saved program for further work. SCRATCH is much faster than NEW or OLD in clearing core.

If, after BASIC types READY, you merely begin typing a new program without clearing core, BASIC will retain the name of the previous program and in effect you will write over that program as though you were changing each single line. However, if you do not remove or type over all of the previous line numbers you will discover the unchanged lines appearing in the new program as well. To avoid this, telling BASIC to SCRATCH the old program and create a new program gives you a blank area on which to write.

8 2.5.4 Stopping a Run in TSS/8 BASIC

If your program begins to print what you know will be a long list of unwanted output for one reason or another, you can stop a running program by depressing the CTRL (control) key and hitting the C key. CTRL/C will cause †C to be printed on the console paper, and will stop execution, returning you to edit mode (BASIC prints READY). You can make changes, save the program, or whatever you wish.

8 NOTE

8

8

8

8

8

8

The up arrow (↑) in the command ↑BS or ↑C is not to be confused with the up arrow used to express exponentiation. The ↑ indicated on the console keyboard is for raising a number to a power. The ↑C, for example, is a short way of writing CTRL/C where the CTRL key is depressed while the C is struck.

8 2.6 RUNNING A BASIC PROGRAM

8 2.6.1 LOGIN Procedure

- Before you attempt to use TSS/8, someone in charge of the
- 8 computer will issue you an account number and a password. When
- 8 you sit down at a teletype console, insure that your console is
- 8 connected to the TSS/8 system and type a RETURN. The computer
- 8 should then respond with a printed dot or period. Connection
- 8 to the computer may be done by merely turning the teletype
- 8 switch to LINE, or it may require dialing up the computer over
- 8 the phone system. If difficulty is experienced when you strike
- 8 the RETURN key, speak to your TSS/8 consultant.
- In answer to the dot type:
- 8 LOGIN account number password
- 8 Enter the three terms with a single space between them and strike
- 8 the RETURN key. For example:
- 8 .LOGIN 175 DEMO
- 8 None of the characters in the line you typed will be printed at
- 8 the console, in order to preserve the secrecy of these codes.
- 8 When you successfully log onto the system, some opening message
- 8 will be printed ending with another dot. In reply type:
- 8 R BASIC

2.6.2 Initial Dialogue

This puts you in communication with BASIC which will then type out:

NEW OR OLD --

If you are entering a new program you reply NEW, if calling in an old program you have saved in a file, reply OLD. To enter the command to BASIC, you must strike the RETURN key. BASIC will then reply:

NEW PROGRAM NAME --

or OLD PROGRAM NAME --

as the case may be. You will type in any six-or-less character identifier as your program name. An old program's name must be typed correctly, so it is a good idea to choose an appropriate, easily remembered name. An example of how to call a program which you had previously saved would look as follows:

OLD PROGRAM NAME--PRIME

where PRIME is the name of the program.

- Programs (called files when they have been saved) may be
- 8 loaded from another user's account, file protect permitting. #
- 8 When BASIC asks for the old program name, you may reply:
- 8 OLD PROGRAM NAME--PRIME 12Ø
- 8 where PRIME is the name of a program and 120 is the file account
- 8 number under which PRIME is stored.
- 8 If a file exists for use of a large number of programmers
- 8 it will likely be placed in the System Program Library and may
- 8 be called by typing the name of the program immediately followed
- 8 by an asterisk:
- 8 OLD PROGRAM NAME -- PRIME*
- 8 will call PRIME from the System Program Library.

Twhen you SAVE a file on disk the protection code of that program allows anyone knowing the account number to access the program. For additional information on file protection codes, see chapter 3.

Following the program name supplied by the user, BASIC then types READY if the program can be found or else it types some error message if it cannot. For example: (user input underscored)

R BASIC

NEW OR OLD--OLD
OLD PROGRAM NAME--PRINTE
CAN'T FIND "PRINTE"
NEW OR OLD--

indicates that the old program PRINTE cannot be found. Perhaps it has not been stored, has been erased, or is stored under some other name.

When BASIC has printed READY, you may begin to type in a new program hitting RETURN after each line, or change or run an old program in accordance with the conventions already established.

2.6.3 RUN Command

When your program is ready to be run (be sure there is an END statement), type RUN, press the RETURN key, and the program will attempt to execute. If there is some error in the way you wrote your BASIC code, an error message will be printed, following which you may correct the errors one line at a time. Then type the RUN command again. If the program executes correctly you will obtain whatever printed output you requested. When the END

statement is reached, BASIC stops execution and again types READY.

2.6.4 Editing Phase

To simplify matters, we can think of BASIC as having two phases, a run phase and an editing phase. The run phase is the time between when you type RUN and when BASIC types READY; this is the time during which BASIC is compiling and executing your program. Once BASIC has printed READY, it is able to accept commands directly from your Teletype; during this editing phase you can prepare your program and can direct BASIC to perform a variety of services such as the SCRATCH command. (You can force an entry to the editing phase with a CTRL/C.) The commands used in the editing phase can all be abbreviated to three letters, some have arguments, others do not, as explained below.

2.6.5 SAVE Command

8

8

8

When you have completed working on a program, you may save it on disk to call again in the future. To do this type:

SAVE

or SAV

This would save your program in a file with the same name you typed in response to the question NEW PROGRAM NAME--(If you think you might forget it, write the name in a REMARK statement at the beginning of the program.)

It is also possible to say:

SAVE name

8

or SAV name

where name is not the original name you gave as a reply to NEW PROGRAM NAME--, however, the name you tell BASIC to save the program under is the name you must give to retrieve the program in response to a later query of OLD PROGRAM NAME--

NOTE

Spaces do have significance in program names (i.e., SAVE TIP TOP will be saved as TIP). In general, then, spaces are delimiters for all editing phase commands.

2.6.6 REPLACE Command

If you have called an old program and made some changes in it, you can then return the corrected program to the disk under its old name using the REPLACE command. This command deletes the old program of that name as it enters the new one.

In response to READY, type:

REPLACE

8

or REP

or, alternatively:

REPLACE name

8

or REP name

which causes the program presently being worked on to replace the old copy of the same program on the disk. If a program name is indicated, that name is used as the file name.

2.6.7 UNSAVE Command

If you wish to delete a program from your disk storage area, type:

UNSAVE name

8

or UNS name

The program with the name specified will be deleted from your permanent file. This is done when you no longer plan to use that program. In general, programs which are not going to be run frequently are best stored on paper tape, reserving disk storage for more active programs. It is possible to delete several files with a single UNSAVE command separating the program names with commas.

2.6.8 LIST Command

Once your program works you may discover you have several feet of Teletype paper filled with corrections and other gibberish. To obtain a clean listing of your program, type LIST followed by the RETURN key. The whole program will be printed. You can then tell the computer to RUN and your output will follow.

For debugging purposes it is sometimes useful to list part of your program. LIST followed by one line number or two line numbers separated by a comma or space will result in BASIC printing either that single line or the lines between and including the two numbers given.

2.6.9 DELETE Command

DELETE followed by two line numbers separated by a comma or space will cause all lines between and including the two given to be deleted from the program. If only one line number is given, that line will be deleted. For example:

DEL 10 20

causes all lines between 10 and 20 inclusive to be deleted.

2.6.10 NEW and OLD Commands

If you have completed working with one program and have saved that program for future use, you may wish to work on another BASIC program or leave the terminal. If you wish to call an old program, type OLD. To indicate that you wish to begin a new program, type NEW. In either case BASIC will request a program name and, following your reply, type READY. These commands may be used at any time, not only in direct response to the question BASIC asks of NEW or OLD PROGRAM NAME--.

2.6.11 CATALOG Command

If you type CATALOG followed by the RETURN key, a listing of all program names in your disk file will be printed by BASIC. For example:

CATALOG

NAME SIZE PROT SCORE .BAS 1 12 DAVE .BAS 1 12 BASØØ2.TMP 1 17 BASIØ2.TMP 1 17

READY

The program names have appended to them the terms .BAC and .BAS which are explained in Chapter 3.

- 8 All of the above commands, SAVE, REPLACE, UNSAVE, LIST, DELETE,
- 8 and CATALOG may be abbreviated to their first three letters in
- 8 TSS/8 BASIC.

2.6.12 BYE Command

- 8 When you are ready to leave the Teletype, type BYE and hit
- 8 RETURN, this will return control to TSS/8 Monitor which prints
- $8\,$ a dot at the left margin. Then type LOGOUT and hit RETURN.
- 8 Wait until the computer has finished its concluding message be-
- 8 fore turning the LINE-OFF-LOCAL knob to OFF or hanging up the
- 8 phone.

2.6.13 ALTMODE key (ESC on some terminals)

- 8 Striking the ALTMODE key (which is non-printing and non-
- 8 spacing) will cause any of the preceding commands (DELETE, LIST,
- δ SAVE, etc.) to be erased. ALTMODE must be struck before the RE-
- 8 TURN key which enters the command into the computer. If you do
- 8 change your mind about a command, you can alter it as shown
- 8 below:

-ALTMODE struck here

- 8 SAVE FOOS DELETED
- 8 TSS/8 BASIC replies \$ DELETED to show that the command has been
- 8 erased, you may then retype the line.

This chapter deals with additional features of BASIC which, once you have learned the BASIC language, will make programming somewhat easier. They are specifically for TSS/8 BASIC.

3.1 Implementation Notes

The TSS/8 BASIC language is compatible with Dartmouth BASIC except as noted below:

- There are no matrix operations.
- 2. All array (subscripted) variables must appear in a DIM statement.
- 3. User defined functions are restricted to one line.
- 4. Maximum size of a BASIC program can be said to be roughly 350 lines. The exact size of a program that a user can run depends upon several factors: the number and size of arrays, number of nested loops and subroutines, number of variables, and user defined functions. A program using an unusually large number of any of these factors will, of course, have less room in which to run.

3.2 Punching a Paper Tape

It may be useful in many cases to have a copy of a program you have written in BASIC stored on paper tape. You can create

such a copy easily if the teletype you are using is tape equipped. Once you have completed your program to the point that you wish to copy it, punch a listing of it through BASIC. The steps involved are:

- 1. Type TAPE followed by hitting RETURN. Any characters you type now will not echo on the console or on your tape.
- 2. Punch the ON button on the tape punch.
- 3. Type LIST followed by the RETURN key. This causes the program to be listed on paper tape and on the console.
- 4. Punch the OFF button on the tape punch.
 Using LIST when in TAPE mode will result in the following:
 - 1. The word LIST will not echo. No leading spaces are printed before line numbers as in a normal LIST.
 - 2. Blank tape is "printed" before and after the program.

You will notice that when you tear off the tape from the punch there will be an arrow head on the tape. This shows the direction in which the tape is later to be inserted into the tape reader. If your teletype does not make such an arrowhead, mark the beginning of the tape "head".

Paper Tape Diagram

Once you have finished punching your program you will wish to return to regular operating mode on the computer. During TAPE mode no characters you type will be echoed. RUBOUTS are ignored, as is blank tape. Typing KEY followed by the RETURN key will bring you back to normal operating mode. You may then continue working on that program, call another program, or log out.

A paper tape can be duplicated or copies made by positioning the tape in the reader depress the ON button, turn the LINE-OFF-LOCAL knob to LOCAL, and turn the reader control switch to START. Tape will be reproduced as it is.

3.3 Reading and Listing a Paper Tape

To read in a paper tape from the low speed reader on the Teletype, first create a new program name in BASIC and proceed as follows:

- 1. Position paper tape in the reader head:
 - a. Raise retainer cover,
 - b. Set reader control to FREE,
 - c. Position paper tape with feed holes over the sprocket wheel and the arrow (cut) pointing outward from the console.
- 2. Type TAPE, hit the RETURN key.
- 3. Set reader control switch to START until reading has been completed. Reader will not stop at blank tape. You must turn the reader control switch to FREE.

- 4. In order to get back into regular operating mode where the characters you type will be echoed at the console, type KEY and hit the RETURN key.
- 5. BASIC will type READY, you can then ask BASIC to LIST or RUN your program.
- 3.3.1 Transferring a File to Paper Tape or DECtape from Disk

 It is not in the scope of this manual to describe the trans
 fer of BASIC programs from disk to paper tape or DECtape. If you

 wish to use these facilities, refer to writeups on PIP (Peripheral

 Interchange Processor) and COPY (both found in the TSS/8 User's

 Guide).

3.4 EDIT Command

Frequently it is only necessary to correct several characters in a line. Rather than retype the entire line, which may be a complex formula or output format, there is a command which allows you to access a single line and search for the character you wish to change. The form of the EDIT command is as follows:

EDI line number

[character]

(Notice that the EDIT command just as all other commands in TSS/8 BASIC may be abbreviated to three letters.) It is then followed by the line number of the statement to be changed. Enter the command by striking the RETURN key. At this point

BASIC types [and waits for you to type a search character after which BASIC types]. The character you give will be some character which already exists on the line (one of the legal BASIC characters, ASCII 240 through 335 inclusive on the ASCII table in Appendix A). After the search character is typed, BASIC prints out the contents of that line until the search character is printed. At this point printing stops, and the user has the following options:

- 1. Type in new characters which are inserted following the ones already printed.
- 2. Type a Form Feed (CTRL/L); this will cause the search to proceed to the next occurrence, if any, of the search character.
- 3. Type a BELL (CTRL/G); this allows the user to change the search character. BASIC types back another [and the user can specify a new search character.
- 4. Use the RUBOUT (or SHIFT/O) key to delete one character to the left each time RUBOUT is depressed. RUBOUT echoes as ←.
- 5. Type the RETURN key to terminate editing of the line at that point, removing any text to the right.
- 6. Type the ALTMODE key to delete all the characters to the left except the line number.
- 7. Type the LINE FEED key to terminate editing of the line, saving the remaining characters.

On completion of the EDIT operation, BASIC types READY. Note that line numbers cannot be changed using EDIT, i.e., you cannot search for a line number digit. Any illegal characters will be ignored.

The following example demonstrates the EDIT command where the incorrect line reads as follows:

60 PRINT "PI=3.14146 AFOU*!"

READY

To edit the line would result in the following output on the Teletype:

FDI 60 60[6] PRINT "PI=3.14146++59[*] APOU*+T

RFADY.

LIST 60
60 PRINT "PI=3.14159 APOUT!"
REARY

The operations involved in editing the line were as follows:

First the number 6 was indicated as the search character. BASIC ignores the line number, but will print it. When the 6 was printed, RUBOUT was struck twice to remove the two incorrect digits and 59 inserted in their place. CTRL/BELL is struck resulting in BASIC accepting another search character. BASIC then prints to the search character * which is removed with a RUBOUT and replaced with T. A LINE FEED is struck to terminate the edit and save the remaining characters.

3.5 COMPILE Command

When a program is debugged and working to your satisfaction, it is faster to be able to directly RUN a program without waiting for BASIC to recompile it each time. To enable you to store a compiled program the COMPILE command has been added to BASIC. The form of the command is as follows:

COMPILE name

or COM name

The program in core will be compiled and saved in the specified file. COMPILE will not overwrite an existing file (it is like SAVE in this respect); if the name is in use the error message: DUPLICATE FILE NAME

will be printed, and the program will not be compiled.

The compiled program may then be loaded and run in the usual manner. For example:

NFW OR OLD--OLD OLD PROGRAM VAME--FTFALL*

PFADY

COMPILE FOOTPL

READY

OLE

OLD PROGRAM NAME -- FOOTPL

READY

PUV

In the example above, the programmer told BASIC to load a System Library Program file named FTBALL into core (the * after FTBALL indicates the System Library files). The programmer told

BASIC to compile the program now in core and store the compiled program in his personal file with the name FOOTBL. Once BASIC has done this it replies READY. The programmer indicates that he wishes to call an old program into core, this old program is the already compiled version of the original program which can be made to execute by giving BASIC the RUN command.

Compiled BASIC files may not be listed or changed in any way; therefore a program should not be saved as a compiled file until it has been completely debugged. If you attempt to list or change a compiled file the error message:

EXECUTE ONLY

will be printed.

3.5.1 File Extensions

In order for the user to easily tell the difference between compiled, uncompiled, and temporary files within your storage area on disk, the following conventions are followed and will help you tell the difference when you run the CATALOG command.

- 1. SAVE and REPLACE commands will always write out a file with the extension .BAS appended to the file name given by the user.
- 2. COMPILE will always write out a file with the extension .BAC to the file name.
- 3. BASIC data files will have the extension .DAT
 4. Certain files will have the extension .TMP indicating a temporary file.

3.5.2 File Protection

TSS/8 permits a user to specify a protection code for each file in his library. The code is made up of five "switches" each of which permits or forbids a class of user from writing or reading a file. The protection code is made up of the sum of the "switch" settings. The switches are:

- forbids any user from reading the file
- 2 forbids any user from writing on the file
- 4 forbids a user in my programming group from reading the file
- 10 forbids a user in my programming group from writing on the file
- 20 forbids me from writing on the file.

In the material above, "my programming group" is that class of users with account numbers whose first two digits are the same as mine; thus accounts 2355 and 2365 are in the same programming group. A protection code of Ø would permit any user to read on the file or to write on it. A protection code of 12 (the sum of 10 and 2) would permit any user to read the file but only permit the file owner to write on it. A protection code of 37 (20+10+4+2+1) permits the file owner to read it but forbids other users from reading or writing. (A protection code of 37 will freeze the file from deletion except through other programs such as COPY. See instructions on using COPY in the TSS/8 USER'S GUIDE.)

Using TSS/8 BASIC, the commands which write disk files (SAVE, REPLACE, COMPILE) also permit the user to specify what protection is to be given to a file. This is done by following the file name with the protection code in angle brackets. For example,

SAVE FOO <10>

will create and save a file named "FOO.BAS" having a protection code of 10. When no protection is specified, a protection of 12 is automatically assumed.

USING BASIC TO SAVE A FILE WITH A PROTECTION CODE >17 WILL RESULT IN A FILE WHICH CANNOT BE DELETED BY BASIC. ONLY THE RENAME FEATURE OF THE PROGRAM "COPY" WILL PERMIT DELETION OF THE FILE. Take care with this option.

3.6 Strings in BASIC

TSS/8 BASIC has the ability to manipulate alphabetic information (or "strings"). A string is a sequence of characters, each of which is one of the printing ASCII characters (given in the table in Appendix A). In TSS/8 BASIC, strings consist of six or fewer characters; strings of more than six characters are broken into 6 character groups.

Variables can be introduced for simple strings, string vectors, and string matrices. A string variable is denoted by following the variable name with the dollar sign character (\$). For example:

Al\$ A simple string of up to 6 characters

V\$(7) The seventh string in the vector V\$(n).

M\$(1,1) An element of a string matrix M\$(n,m).

As usual, when string arrays or matrices are used a DIM statement is required. For example:

100 DIM V (100), M\$ (5,5)

reserves eleven strings for the vector V\$ and 36 strings for the matrix M\$.

3.6.1 Reading String Data

Strings of characters may be read into string variables from DATA statements. Each string data element is a string of one to six characters enclosed in quotation marks. The quotation marks are, of course, not part of the actual string. For example:

1Ø READ A\$, B\$, C\$

200 DATA "JONES", "SMITH", "HOWE"

The string "JONES" is read into A\$, "SMITH" into B\$, and "HOWE" into C\$. If the string contains more than six characters, the excess characters are ignored.

1Ø READ A\$

2Ø PRINT A\$

3Ø DATA "TIME-SHARING"

4Ø END

causes only

TIME-S

to be printed.

String and numeric elements may be intermixed in DATA statements. A READ operation always fetches the next element of the appropriate type. In the following example:

> 1Ø READ A, A\$, B 2Ø DATA "YES", 2.5, "NO", 1

2.5 is read into A, "YES" into A\$, and 1 into B.

The standard RESTORE statement resets the data pointers for both string and numeric elements. Two special forms of the RESTORE command, RESTORE*and RESTORE\$, may be used to reset just the numeric and string data list pointers respectively.

RESTORE* Resets the numeric DATA list
RESTORE\$ Resets the string DATA list
RESTORE Resets both numeric and string
DATA lists.

1Ø READ A,A\$,B

2Ø DATA "YES", 2.5, "NO", 1

3Ø PRINT A,A\$,B

4Ø RESTORE*

5Ø READ A,A\$,B

6Ø PRINT A,A\$,B

7Ø END

would print:

2.5 YES 1 2.5 NO 1

If line 40 were changed to RESTORE, this program would print:

2.5 YES 1 2.5 YES 1

since the string as well as the numeric data lists were reset.

3.6.2 Printing Strings

The regular BASIC PRINT statement may be used to print out string information. If the semi-colon character is used to separate string variables in a PRINT command, the strings are printed with no intervening spaces. For example, the program:

1Ø READ A\$, B\$, C\$
2Ø PRINT C\$; B\$; A\$
3Ø DATA "ING", "SHAR", "TIME-"
4Ø END

causes the following to be typed:

TIME-SHARING

3.6.3 Inputting Strings

String information may be entered into a BASIC program by means of the INPUT command. Strings typed at the keyboard may contain any of the standard teletype characters except back arrow (+) and quotation mark. Back arrow, as always, is used to delete the last character typed. Commas are used as terminators just as with numeric input. If a string contains a comma the whole string must be enclosed in quotation marks. The following program demonstrates string input.

1Ø INPUT A\$, B\$, C\$
2Ø PRINT C\$, B\$, A\$
3Ø END
RUN

? JONES, SMITH, HOWE HOWE SMITH JONES

READY

Strings and numeric information may be combined in the same INPUT statement as in the following example. Note that if an input string contains more than six characters, only the first six are retained.

1Ø INPUT A, A\$, B\$
2Ø PRINT A\$, B\$, A
3Ø END
RUN
? Ø1754, MAYNARD, MASS.
MAYNAR MASS.

1754

The numeric variable A is set to 1754, the string "MAYNAR" is put in the string variable A\$, and the string "MASS." is put into the string variable B\$.

3.6.4 Line Input LINPUT

Strings of more than six characters may be entered by means of the LINPUT (line input) command. A LINPUT statement is followed by one or more string variables. For example:

 $1\emptyset\emptyset$ LINPUT A\$(1), A\$(2), A\$(3), A\$(4), A\$(5)

The first six characters to be typed are stored in the first string variable, the next six in the second, and so on until the line of input is terminated by a carriage return. Commas and quotes are treated as ordinary characters and hence are stored in the string variables. For example, if the following line were typed in response to the above LINPUT command:

? MAYNARD, MASS. Ø1754

then the values of the string variables would be as follows:

A\$(1) = "MAYNAR" A\$(2) = "D, MAS" A\$(3) = "S. Ø17" A\$(4) = "54" A\$(5) = ""1

In this example, the maximum number of characters which could be typed would be 30. Any additional characters would be ignored. In all cases, the maximum number of characters which may be typed to TSS/8 BASIC is 50. If a longer line is typed, the message LINE TOO LONG is typed. The line must be re-entered.

It is possible to mix numeric and string variables in a LINPUT statement, but it is not recommended. As an illustration of how this might be done, consider the example given earlier:

1Ø LINPUT A,A\$,B\$

where the user might type:

? Ø1754, MAYNARD, MASS.

This still sets the numeric variable A to 1754 (when used in LINPUT statements, numeric input remains unchanged). However, the string variable A\$ would now be "MAYNAR", and the string variable B\$ would be "D, MAS".

NOTE

When inputting strings with LINPUT, the error messages: "MORE?" and "TOO MUCH INPUT, EXCESS IGNORED" cannot occur. LINE TOO LONG will occur if more than 50 characters are input.

¹Strings may also consist of zero characters. Such a string is empty, or "null". If printed, it causes nothing to be output. The null string is usually represented by a pair of quotes with nothing in between (""). The null string should not be confused with a string of one or more spaces.

3.6.5 Working with Strings

Strings may be used in both LET and IF statements. For example:

1Ø LET Y\$= "YES"
1Ø IF Z\$= "NO" THEN 100

The first statement stores the string "YES" in the string variable Y\$. The second branches to statement 100 if Z\$ contains the string "NO". For two strings to be equal, they must contain the same characters in the same order and be the same length. In particular, trailing blanks are significant since they change the length of the string. "YES" is not equal to "YES".

The relation operators < and > may also be used with string variables. When used with strings, these relations mean "earlier in alphabetic order" or "later in alphabetic order", and they may be used to alphabetize a list of strings. The relationals >=, <=, <> may also be used in a similar manner. The arithmetic operations (+, -, *, /, †) are not defined for strings. Thus, statements such as LET A\$ = 3*5 and LET C\$ = A\$+B\$ have no meaning, and should never be used in a BASIC program. They will not, however, cause a diagnostic to be printed, and the results of such operations are undefined.

3.6.6 The CHANGE Statement

The BASIC command CHANGE may be used to access and alter individual characters within a string. Every string character has a numeric code (see Appendix A), a number which is used to stand for that particular character. The CHANGE statement converts a string into an array of numbers, or vice versa. The CHANGE statement has the form:

100 CHANGE A TO A\$

or

100 CHANGE A\$ TO A

where A\$ is any string variable (or an element of a subscripted string variable) and A is an array variable with at least seven elements. Any array variables used in CHANGE statements must have appeared in a DIM statement with a dimension of at least six.

The following program illustrates the use of the CHANGE statement. In this example, CHANGE is used to change a string variable into an array of numbers.

```
1Ø DIM A(6)
2Ø READ A$
3Ø CHANGE A$ TO A
4Ø PRINT A(Ø); A(1); A(2); A(3); A(4); A(5); A(6)
5Ø DATA "ABCD"
6Ø END
RUN
4 65 66 67 68 Ø Ø
```

The CHANGE statement takes each character of the string and stores its corresponding numeric code in elements one to six of the array. Remaining array elements are set to zero. The length of the string (\emptyset -6 characters) is then stored in the zero element of the array. In the example above, the character codes for A, B, C, and D are stored in A(1) to A(4). A(5) and A(6) are set to zero. The number 4 is stored in A(\emptyset) since the string A\$ is of length 4.

CHANGE may also be used to change an array of numeric codes into a character string. The following program illustrates this use of the CHANGE statement.

```
1Ø DIM A(6)
2Ø FOR I=Ø TO 5
3Ø READ A(I)
4Ø NEXT I
5Ø CHANGE A TO A$
6Ø PRINT A$
7Ø DATA 5, 65, 66, 67, 68, 69
8Ø END
RUN
```

ABCDE

The length of the resulting string is determined by the zero element of the array. In the above example, the string is of length five. The elements of the array, starting at subscript 1, are assumed to be numeric character codes (32 to 94). These are converted to characters and are stored in the string. If any codes are encountered which are not valid character codes, or if an invalid string length is given, the message BAD VALUE IN CHANGE STATEMENT AT LINE n is typed out, and execution is stopped.

3.6.7 A Note About CHANGE

A BASIC string of less than six characters always has the remaining character positions filled with zeros. For this reason, when such a string is changed to an array, the first six array elements are set to zero. The CHANGE statement always fills six array elements, even though the strings may not be six characters long. The user should be very careful to always dimension the array used in a CHANGE statement to at least 6. If a string of characters is transformed into an array of less than 6 elements, an undetected error will occur.

The CHANGE statement is usable with strings not created by BASIC. It may, for example, be used to access files other than BASIC data files. Each string variable corresponds to three computer words. The CHANGE statement treats these three words as six bytes, converts each byte to its numeric character code equivalent and stores it in the corresponding array element. The zero element of the array, the string length, is set equal to the number of bytes (character) before the first zero byte. When reading unspecified data, there may be non-zero bytes following this zero byte. If so, they will be transferred to the array as well.

3.6.8 The CHR\$ Function

Occasionally it is desirable to type a character other than the normal ASCII set, or to compute the value of a character to print. For example one might wish to transmit to the printer certain control characters or even the lower case letters. For this sort of purpose the CHR\$ function is used in a PRINT statement. The argument of the CHR\$ function (modulus 256) is sent as a character to the teletype. For example:

```
16 304 I = 9 TO 9
```

20 PRINT CHR\$(In.8):

30 WEXT I

LØ END

prints "Ø123456789" since 48 to 57 are the ASCII values for the characters "Ø" to "9". The following special characters may be printed using the CHR\$ function:

| Bell | CHR\$(7) |
|-----------------|-----------|
| Line Feed | CHR\$(10) |
| Carriage Return | CHR\$(13) |
| Quote (") | CHR\$(34) |
| Back Arrow () | CHR\$(95) |
| Form Feed | CHR\$(12) |
| | NOTE |

The teletype will accept characters from \emptyset to 255 (decimal), many of which do nothing on most kinds of teletypes. Some of the special (non-printing) characters should not be used. For example, CHR\$(\mathbb{I}_{+}) causes a Dataphone to disconnect.

For each ASCII code there is a second form accepted by CHR\$ and CHANGE. The second code is obtained by adding 128 to the code given in the whird column of the table in Appendix A. For example, CHRS would mype "A" in response to either 65 or 193 as an argument.

lower case characters correspond to the arguments 97 through 122 when used with GHRS. Thus CHRS(97) would print an "a" on a teletype capable of bandling that character (an ASR 38, for example) thus permitting TSS/8 BASIC to handle all printable ASCII characters in output mode.

3.6.9 Modification to DATA Statement

A DATA statement may now be legally terminated by a comma. For example:

1Ø DATA 1,2,3, 2Ø DATA 4,5,6

is now treated the same as

1Ø DATA 1,2,3,4,5,6

3.7 Program Chaining

Most BASIC programs are easily accommodated by TSS/8 BASIC. If a program becomes very long, however, it may be necessary to break it down into several segments. Typically, programs of most than two to three hundred statements must be split up. A program that has been broken down into more than one piece is commonly referred to as a "chained" program.

Each part of a chained program is saved on the disk as a separate file. The last statement of each part to be executed is a CHAIN statement specifying the name of the next section of the program. This file is then loaded and executed. It may in turn chain to still another section of the program. The general form of the chain command is:

414 CHAIN "NAME"

or

414 CHAIN AS

where "NAME" is the name of the next segment to be executed (one to six characters) enclosed in quotation marks. The name of the next segment may also be contained in a string variable. In either case the file of that name is loaded and run. Thus, the statement:

999 CHAIN "SEG2"

is equivalent to:

OLD PROGRAM NAME-SEG2

RUN

except that it happens automatically. Each separate part of the program links to the next part of the program chain.

The individual sections of a chained program may be either regular source files (.BAS) or compiled files (.BAC). If the sections are source files, however, they must be compiled before they are run. A chained program runs more efficiently if all its sections have been compiled.

If an error occurs while compiling or running a chained program, the name of the section being run, the one having the error, is typed out as part of the error message. In all cases, whether a program terminates by an error or a STOP or END, BASIC returns to the first program in the chain. This is the one which is available for editing and rerunning when BASIC types READY.

Most chained programs require information from one section to be passed on to the next. The first section may, for example, accept input values and perform some preliminary calculations. The intermediate results must then be passed to the next section of the programs. This passing of values is done by means of BASIC's file capability, which is explained in the next section.

Whenever a CHAIN operation is performed, program data which has not been saved in a file is lost. Variable and array values are not automatically passed to the next program.

3.8 Disk Data Files

The standard BASIC language provides two ways of handling program data items. They may be stored within the program (in DATA statements) or they may be typed in from the terminal. DATA statements, however, allow for only a limited amount of data. Also, the data is accessible only to the progam in which it is embedded. Typing data in from the terminal allows it to be entered into any program, but it is a time-consuming process. In either case the data, or the results of calculations, cannot be conveniently stored for future use. All these limitations may be overcome by the use of external data files.

A data file is separate from the program or programs which use it. It is a file on the disk just like a saved program, but it contains numbers or strings rather than program statements. This information may be read or written by a BASIC program. (Information is stored in a data file in a coded format. Therefore, it cannot be listed by the BASIC Editor or TSS/8 EDIT.) A file may be as long as necessary, subject only to the file limitations of TSS/8 (maximum file size is about 350,000 characters). String and numeric information may be combined in a single file. The number of data files a user may have is again limited only by TSS/8 (about 100, space allowing). When first created, the contents of a file are unspecified until it is written in.

3.8.1 File Records

A file is made up of logical units called "records". A record may be as small as a single numeric or string variable.

More typically, it is a group of variables or arrays. The design of the program itself usually dictates the most efficient size of the record. If, for example, the program manipulates a series of 5 x 5 matrixes, each record could contain one such matrix. If the program operates on 80 character alphanumeric records, 14 string variables might make up a record.

The size and composition of a record is defined with a RECORD statement. Like the DIM statement, RECORD is followed by a series of variables. They may, however, be unsubscripted as well as subscripted. For example:

- 1Ø RECORD A(5,5)
- 10 RECORD B\$(14)
- 10 RECORD A, B, C\$(8), D, E(5)

The set of variables mentioned in a RECORD statement, taken altogether, constitute a record. Each element within the record is in essence a field. Numeric and string information may be mixed in order to make up the most convenient record.

Variables mentioned in a RECORD statement
should not appear in a DIM statement. The RECORD statement reserves variable space exactly as a DIM statement does.
The difference is that the variables are also identified as
being used for file input and output. Non-subscripted variables
appearing in RECORD statements must not have been used previously
in a program. RECORD statements should always be
the first statements in a program.

Records may be any length. A long record is typically more efficient since more information is transferred in a single operation. Records should, however, be only as long as necessary since excess variables will make the file longer. In particular,

it is important to remember that all arrays and matrices have zero elements. A(5,5) has 36 elements, not 25. If A appears as part of a record, all 36 elements should be used. It is also useful to try to make record sizes 43 variables long, or a multiple of 43. Each RECORD statement reserves program variable space in units of 43 whether or not the record is that big. Unless the record fills out this area, some program variable space is wasted. It is not worth it, however, to make an inherently small record 43 variables long just to conform to this convention. To do so would be to make the file unnecessarily large.

3.8.2 Opening a Disk File

Disk data files are completely separate from the programs which use them. Therefore, the program must specify which file or files it will use. The OPEN command is used for this purpose. OPENing a file associates it with an internal file number, either 8 or 9. (A program may have two disk files open at a time.) For example:

100 OPEN 9, "DATA10" 100 OPEN 8,A\$

The name of the file to be opened may be explicitly stated in the OPEN command. If it is, it must be contained in quotation marks. The file name may also be contained in a string variable, allowing the program to decide which file to open, perhaps on the basis of input from the program's user. In either case, the name of the file is preceded by the internal file number, either 8 or 9. This argument may also be an expression whose value is either 8 or 9. If, when a file is opened on an internal file number, a file was already open there, it is closed first.

If no file of that name exists, the file is created. In either case, once the file is open, it is available for both reading and writing. BASIC disk data files have an extension of .DAT.

3.8.3 Reading/Writing Disk Files

Once open, files may be read and written one record at a time, using the GET and PUT statements. GET statements read one record's worth of information directly into the variables in the specified RECORD. PUT statements write out the present values of the variables in the specified RECORD. Both GET and PUT statements are followed by the internal file number (8 or 9 or an expression), the line number of the RECORD statement containing the variables to be transferred, then the name of a "control" variable. For example:

```
100 RECORD A, B, C$(30), D(8)
110 OPEN 8, "FILE1"
120 LET I=0
130 GET 8, 100, I
```

The control variable specifies the file record to be transferred. In the example above, FILE1 is open as internal file 8. The value of I is zero. Therefore, the GET statement in line 130 reads the first record (record 0) of FILE1 into A, B, and the arrays C\$ and D. Single numeric values are read into A and B, 31 strings are read in C\$, and 9 numeric values are read into D. After each transfer, whether it is a GET or a PUT, the value of the control variable is automatically incremented. Successive GET's or PUT's automatically proceed to the next record of the file.

The PUT command has a similar format. For example, if line 130 of the above program had been:

13Ø PUT 8, 1ØØ, I

the present values of A, B, C\$, and D would have been written out to the first record of FILE1.

File records may be accessed randomly by simply setting the control variable to the desired record number before doing the GET or PUT. Single records may be read, changed, and then written back without the need to process the entire file. When reading a file, the record referenced in the GET statement must, of course, be the same as the record referenced in the PUT statement which wrote the data into the file. The total length of the record and the relationship of string and numeric fields within the records used for the GET's and PUT's must be the same. If they are not, improper information will be read and written.

New files may be created by opening a file which does not already exist. As successive records are written out to the file its length is extended as necessary. When a new file is created, it is useful to immediately write an "end-of-file" code in the last record. Writing the last record first forces the entire file to be allocated, making sure that enough disk space is available. It also provides an end-of-file marker. Programs which read this file may then check for this end-of-file to avoid reading past the end, which is an error. Existing files may be enlarged by writing a new record farther out. If the program does not know how big the file will be, it may simply write records out serially. The file will be automatically extended as needed. When all the records have been written, one final end-of-file mark may be added.

In general all records read or written in a specific file should be the same length, that is contain the same number of variables. However, if the user is careful he may intermix records of different lengths in a file. Suppose the following statement is executed:

40 PUT 8,100,N

and the value of N is n and the record specified by statement 100 is of length m. The PUT statement will write m variables in the file starting at the m*n variable.

The simple rule for computing the first variable in the file to be accessed is the record length times the record number. (Remember the first record is record number zero.)

3.8.4 Closing and Deleting Disk Files

Once all work has been completed on a file, it should be "closed" by a CLOSE statement. Once it is closed, it may not be read or written unless it is reopened. The file does, however, remain on the disk and is available for future use. The CLOSE command is followed by the internal file number to be closed (8 or 9). For example:

95Ø CLOSE 8

If the disk file was just created for temporary scratch use (to pass parameters during a CHAIN, for example) it should be deleted at the end of the program instead of closed. The UNSAVE command is used to delete files. For example:

1000 UNSAVE 9

The file open on internal file number 9 is deleted from the disk. Both CLOSE and UNSAVE may be followed by an expression instead of a constant.

Open disk files are automatically closed at the end of the program, unless the program CHAINs to another program. In this case, all open files remain open and the new program may access them without executing an OPEN statement.

3.9 DECtape Files

Large permanent files are best stored on DECtape instead of disk. Each DECtape holds up to 380,000 characters of information. DECtape files may be dismounted for safekeeping, thereby insuring their privacy. Files on DECtape are very similar to files on disk except that they do not have file names. Each reel of DECtape is a discrete file. When mounted on a DECtape drive, records may be read and written directly on the tape.

A DECtape unit, and hence the file mounted on it, may be used by only one user at a time. If no one is using the unit, a user may assign it. Once assigned, that user has exclusive access to it until he releases it. Each DECtape drive has a "write-lock" switch which physically locks out any writes to that unit. If the write-lock switch is set, programs may not write on the tape even though the unit is assigned.

DECtape files may be used in a variety of ways. Programs which need very big files should use DECtape to avoid swamping the disk. Administrative files, such as student or employee

records, are best stored on DECtape. Since they are removable, and can be write-locked when mounted, their usage can be tightly controlled. DECtapes are also useful for information retrieval. A data tape may be kept permanently mounted, but write-locked. Individual users may run programs which assign and query that file, then release it for others to use.

3.9.1 DECtape File Records

Records for DECtape files are specified the same way as for disk files, with a RECORD statement. All rules for disk records also apply to DECtape records. In fact, the same RECORD statement may be used for both a DECtape and disk file. (This is useful when reading a tape file to a disk file for processing. Disk files are considerably faster than tape files.)

It is possible to specify any record length for a DECtape file, but a size of 43 variables is suggested, even more strongly than for disk files. DECtapes are physically structured into blocks, each of which will hold exactly 43 variables. If the record specified by the program is, for example, 44 variables, it will require two full blocks of the tape.

Records which are multiples of 43 variables are efficient in utilizing DECtape space, but are not efficient in speed. Such records are written in consecutive DECtape blocks. The tape unit cannot read or write consecutive blocks without stopping the tape and rewinding it slightly.

This tape "rocking" also occurs when single block records (43 variables or less) are read or written as consecutive DEC-tape records. (In this case, each DECtape file record corresponds to a physical tape block.) The most efficient way to utilize DECtape is to make each record 43 variables in length, and write them onto every tenth record in the file (records \emptyset , $1\emptyset$, $2\emptyset$, etc.). When the entire length of the tape has been traversed (the last block of the tape is number 1473) write next

into records 1, 11, 21, etc. In this way, every record will eventually be filled. Programs which will be used repeatedly should utilize the tape in this manner.

3.9.2 Opening a DECtape File

DECtape files, like disk files, are completely separate from the programs which use them. Therefore, the program may specify which tape, or tapes it will use. The OPEN command is used for this purpose. Since DECtape files do not have names¹, the OPEN command specifies the DECtape unit number to be used. It is assumed that the proper tape reel has been mounted. If the file is to be updated, the unit should be write-enabled. If not, it should be write-locked. The OPEN command is thus followed by the unit number to be used $(\emptyset-7)$. Only units 2-7 may be used at UWM.

100 OPEN 2

155 OPEN 7

The unit number could be an expression, Making the unit number a variable is very useful since it is hard to predict which units will be available at the time the program is run. When it is a variable, the user may mount the file on any free unit, then type the number into the program via an INPUT statement.

When the OPEN command is executed, the indicated DECtape unit is assigned. It cannot subsequently be opened or assigned by any other user. Thus, it is possible to try to open a unit which is already assigned. If, in the above examples, units 2 or 7 were assigned, the program would be terminated and an error message typed out. An alternative form of the OPEN command allows the program itself to handle this situation. OPEN commands may include an ELSE clause which specifies a line number. If the OPEN command fails, BASIC automatically performs a GOTO to this line number. For example:

1ØØ OPEN 2 ELSE 9ØØ

¹It is important to note that BASIC DECtape files are not the same as the file-oriented DECtapes used by TSS/8 COPY. There is no directory on a BASIC DECtape file. Each tape is considered to be one file of pure data.

If unit 2 is available, it is assigned and BASIC goes on to execute the next statement. If unit 2 is not available, statement 900 is executed next. It could print a message and perhaps ask for an alternate unit number.

3.9.3 Reading and Writing DECtape Files

DECtape files are read and written using the same GET and PUT commands as are used for disk data files. The internal file number is a number between 2 and 7, or an expression. Unlike disk files, DECtape files are of a constant length equal to the capacity of the tape. The exact number of records per reel depends on the record size as follows:

| Record Size | Tape C | apacity |
|--|--------|-------------------------------|
| 1-43 variables 44-86 variables 87-129 variables etc. | 737 | records records records |

As indicated in the section on DECtape records, a record size of 43 variables or less is recommended since it conforms to the physical blocking of the tapes themselves. It is also desirable to space the records out along the tape so that the tape does not rock. The following subroutine could be used to write 1474 records on the tape in this fashion. It assumes that R is set to zero before it is called the first time and that the unit number is in U.

```
500 REM SUBROUTINE TO WRITE RECORDS ALONG TAPE
```

- 51Ø REM WRITES ONE RECORD EACH TIME CALLED
- 515 PUT U,10, R 'REMEMBER THIS INCREMENTS R
- 517 LET R=R+9 'SPACE OUT 10 BLOCKS
- 524 IF R<1474 THEN 55Ø 'OK TO RETURN
- 53Ø IF R=1479 THEN 56Ø 'TAPE IS FULL
- 54Ø LET R=R-1479
- 545 IF $R > \emptyset$ THEN 55 \emptyset
- 547 LET R=R+1Ø
- 550 RETURN
- 560 STOP 'TAPE IS FULL

The following function may also be used to convert a logical record number (\emptyset to 1469) to a physical record block spaced along the tape. This function will not use blocks \emptyset -3. They are therefore available for headers or labels.

$$FNC(X) = (X-INT(X/147)*147)*10 + INT(X/147)+4$$

Both the subroutine and the function assume a record length of 43 variables or less.

Once opened, any record on the tape may be read. The tape unit must, however, be write-enabled if it is to be written. Try-ing to PUT to a write-locked tape is an error.

3.9.4 Closing DECtape Files

Once all work on a DECtape file has been completed it may be closed. Closing a file releases the tape unit and makes it available to other users. Thus, if the tape contains important information (and especially if it is write-enabled) the CLOSE should not be done until the tape reel has been removed. If no CLOSE statement is encountered in the program, the unit remains assigned even after the program finished. It will remain assigned until a TSS/8 RELEASE command is executed or the user logs out. An example of a CLOSE command:

11ØØ CLOSE 6

3.9.5 Using Data Tapes with PS/8 FORTRAN

Numeric DECtape data files written by TSS/8 BASIC may be read by PS/8 FORTRAN by use of FORTRAN's RTAPE and WTAPE sub-routines, and vice versa. (String and Hollerith variables use different character codes). Thus it is possible to use BASIC to prepare an input or update tape for a stand-alone FORTRAN program. This provides a convenient way to do big jobs in off-hours, without having to leave the time-sharing mode for very long.

3.10 Line Printer Output

If a line printer is available, it may be used both to list BASIC programs and as an output device for the programs themselves. The line printer may only be used by one user at a time.

The commands associated with line printer output are LLIST and LPRINT.

LLIST is similar to the LIST command except that the program listing is output to the line printer rather than to the Teletype The LLIST command assumes that no other user has the line printer assigned and responds by typing WHAT? if the line printer is not available. After the listing is complete, the line printer is released and is available to any user.

BASIC programs may use the line printer as an output device during execution by means of the LPRINT command. LPRINT is exactly like PRINT except that, again, the information goes to the line printer rather than to the Teletype. All formatting conventions of the PRINT command are available with LPRINT. In particular, CHR\$(12) may be used to skip to the top of the next form.

The command LPRINT also assumes that no other user has the line printer assigned. However, using this command when the line printer is not available causes the program to terminate. Once LPRINT successfully assigns the line printer, it remains assigned until the program terminates.

The OPEN and CLOSE commands may be used to assign and release the line printer. An OPEN command with a device number of 11 will assign the line printer, or if it is not available and an ELSE clause is specified, transfer control to the line number specified in the ELSE clause. CLOSE 11 will release the line printer.

3.11 Papertape Output

The high speed paper tape punch may also be used as an output device. Like the line printer, the paper tape punch may only be used by one user at a time. The OPEN and CLOSE commands with an internal file number of 10 will respectively assign and release the paper tape punch as shown in the following example:

 $1\emptyset$ OPEN $1\emptyset$ ELSE $1\emptyset\emptyset$ 'GOTO $1\emptyset\emptyset$ IF PUNCH UNAVAILABLE 20 CLOSE $1\emptyset$

Here too, a GOTO statement in combination with an ELSE clause can be used to transfer program control should the paper tape punch not be available.

The command LPRINT causes output to go to the paper tape punch when this device has been assigned. For example:

1Ø OPEN 1Ø 2Ø LPRINT "THIS GOES TO PTP."

causes the statement "THIS GOES TO PTP." to be punched onto papertape.

If the device is not released via a CLOSE command, it remains assigned even after the program terminates.

3.12 ON GOTO

The ON...GOTO statement may be used to provide a many-way branch. The general form of the ON...GOTO is:

ON expression GOTO line number, line number

If the value of the integer part of the expression is 1, a

GOTO is performed to the first statement. If the value of

the integer part of the expression is 2, a GOTO to the second

statement number is performed, etc. If the value is less than

one, or greater than the number of statement numbers, control will

"fall through" to the next line. Examples of ON...GOTO:

20 ON V1 CO TO 100,200,300 30 RFM - IF N1 <1 OR V1 >3 FXFCUTE THIS LINE 800 ON (Pt2 - 4*4) CO TO 200,400,12

READY

3.13 SLEEP

The SLEEP statement causes a BASIC program to pause for a specified interval, then continue running. SLEEP is followed by the number of seconds the program is to pause. For example:

222 SLEEP 30 or 220 LET N=15 222 SLEEP 2*N

causes a 30 second delay in the program.

The SLEEP statement is a useful way for a program to wait for a device (DECtape or line printer) which is busy. The ELSE clause in the OPEN statement can go to a routine which pauses for a while, then retries the OPEN. When the current user finishes with the device and releases it, the program may then proceed to OPEN and use it. This capability is especially useful when many users may be looking up information on a single DECtape file. It may also be used to allow two programs to communicate with each other. Each writes information on a tape file for the other, or others, to read.

SLEEP should always be used when waiting for a device. While the program is sleeping it is not using any processor time. A SLEEP time of 30 to 60 seconds is recommended. It is particularly important that the program not wait by repetitively retrying the OPEN. To do so wastes computer time and slows down other users. The integer part of the argument is used to determine the number of seconds to delay. This value must be between Ø and 4Ø95.

3.14 Comments

An entire statement of comments may be included in the BASIC program by means of the REM statement. Often comments are easier to read if they are placed on the same line with an executable statement rather than in a separate REMARK statement. This can be accomplished by ending an executable statement with an apostrophe. Everything to the right of the apostrophe up to the statement terminator (carriage return or backslash as described in section 21) is ignored (unless the apostrophe occurs within a print literal or string constant.) For example:

- 10 LET X=Y 'THIS IS A COMMENT'
- 2Ø PRINT "BUT 'THIS IS NOT A COMMENT"
- 3Ø LET X\$="A'B"

Thus, a comment is added to line 10 with an apostrophe, but in lines 20 and 30 the apostrophe is treated as a valid character.

3.15 Blank Lines

To make BASIC programs easier to read, blank lines can be inserted anywhere in a BASIC program. These can be used to break a program into logical sections, or (as is often done) to insert remarks with the apostrophe feature. For example:

10 'PROGRAM WRITTEN BY SAM JONES 100 - _

Note that to insert a blank line, you must type one or more spaces after the line number; typing the line number alone will just delete that line from the program.

3.16 More than One Statement on a Line

As many statements as will fit may be typed on a single program line. Each statement must be separated by the backslash character "\" (SHIFT/L). The only statement requiring a line number is the initial one. For example:

10 FOR I=1 TO 10 PRINT I NEXT I

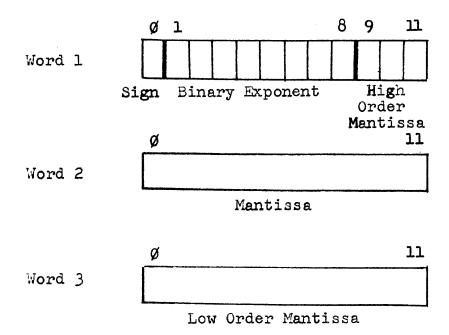
Note that the backslash character acts as a statement terminator and thus cannot be included in a comment statement.

3.17 Internal Data Codes

Using the file I/O capabilities and the CHANGE statement it is possible to examine data which was written on a DECtape or disk file by a program other than BASIC. There are two data formats, Numeric Data and String Data.

3.17.1 Numeric Data

Each numeric value in TSS/8 BASIC is three PDP-8 words long. The format is as follows:



A one in the sign bit means that the number is negative. The exponent is kept in "excess 200" form where

$$200_{8}$$
 is 2^{9}
 201_{8} is 2^{1}
 177_{8} is 2^{-1}

The assumed decimal point is preceding bit 9. Also, the number

is always normalized, meaning that bit 9 is always 1 unless the number is zero. (Zero is represented by three zero words.)

Note that this format is the same as the format used by FORTRAN and described in Programming Languages.

3.17.2 String Data

Each string variable is three PDP-8 words long. Each word contains two 6-bit bytes or characters. If a string variable is filled by a GET from a source which was not written by a BASIC program, a BASIC program may examine the data in the variable by performing a CHANGE on that variable. The six bytes will be translated as if they were internal character codes for BASIC string characters.

Appendix A shows how this translation interprets the 64 possible bytes. Note that after such a CHANGE, the \emptyset^{th} element of the array contains a count of the number of characters occurring before the first null.

CHAPTER 4

SUMMARY

| TS | |
|----|----|
| | TS |

| Command | Example of Form | Explanation |
|---------|---------------------------|--|
| LET | LET v=f | Assign the value of the for- mula f to the variable v. |
| READ | READ v1, v2, , vn | Variables vl through vn are assigned the value of the corresponding numbers in the DATA string. |
| DATA | DATA nl, n2,, nn | Numbers nl through nn are to be associated with corresponding variables in a READ statement. |
| PRINT | PRINT al, a2, | Print out the values of the specified arguments, which may be variables, text, or format control characters (, or ;). |
| GOTO | GOTO n | Transfer control to line n; continue execution from there. |
| IF-THEN | IF fl r f2 THEN n | If the relationship r between the formulas fl and f2 is true, then transfer control to line n; if not, continue in regular sequence. |
| IF-GOTO | IF fl r f2 GOTO n | Same as IF-THEN |
| FOR-TO | FOR v=f1 TO f2 STEP f3 | Used to implement loops: The variable v is set equal to the formula fl. From this point the loop cycle is completed following which v is incremented after each cycle by f3 until its value is greater than or equal to f2. If STEP f3 is omitted, f3 is assumed to be +1. |

| Command | Example of Form | Explanation |
|-----------|--|--|
| NEXT | NEXT V | Used to tell the computer to return to the FOR statement and execute the loop again until v is greater than or equal to f2. |
| DIM | DIM v(s) DIM v(s1, s2) | Enables the user to create a table or array with the specified number of elements where v is the variable name and s is the maximum subscript value. Any number of arrays can be dimensioned in a single DIM statement. |
| GOSUB | GOSUB n | Allows the user to enter a sub- routine at several points in the program. Control transfers to line n. |
| RETURN | RETURN | Must be at the end of each sub- routine to enable control to be transferred to the statement fol- lowing the last GOSUB. |
| RANDOMIZE | RANDOMIZE RANDOM | Enables the user to obtain an unreproducible random number sequence in a program using the RND function. |
| INPUT | INPUT v1; v2,, vn | Causes typeout of a ? to the user waits for the user to supply the values of the variables v1 through vn. |
| REM | REM | When typed as the first three letters of a line allows typing of remarks within the program. |
| RESTORE | RESTORE | Sets pointer back to the begin- ning of the string of DATA values. |
| DEF | DEF FNB (x) = f(x) DEF FNB(x, y) = f(x, y) | The user may define his own functions to be called within his program by putting a DEF statement at the beginning of a program. The function name begins with FN and must have three letters. The function is then equated to a formula f(x) which must be only one line long. Multiple variable function definitions are allowed. |

| Command | Example of Form | Explanation |
|---------|-----------------|--|
| RECORD | RECORD A(5),B | The format in which data is written on a file. A combination DIM statement for arrays and a "Target" for GET and PUT statements. |
| GET | GET n,m,J | Transfer data from the file open as file "n" according to the RECORD in line "m". Acquire the "J"th record. (After execution increment J by one.) |
| PUT | PUT n,m,J | Transfer data to the file open as file "n" according to the RECORD at line "m". Write in the "J"th location. (After execution increment J by one.) |
| OPEN | OPEN 8, "name" | Open a file named "name" as file #8. Data may be subsequently PUT or GET to that file where 8 is used in place of "n". Only the number 8 or 9 may be used. |
| OPEN | OPEN 3 | Prepare DECtape unit 3 for data transfer in either direction. Data may be PUT or GET to or from that tape where "n" =3. Only the numbers 2 through 7 may be used (UWM convention). |
| CLOSE | CLOSE n | Close a file from use. "n" may take the value 0-11. The file or device is freed for others. |
| UNSAVE | UNSAVE 9 | Erase the file from the library. Only 8 or 9 may be unsaved. |
| OPEN | OPEN 10 | Prepare the Lineprinter (11) or the Paper tape punch (10) for output. When through with de- vice it should be closed. |
| LPRINT | LPRINT "HI",X | Same as PRINT, except output goes to Printer or Paper tape punch, whichever has been opened. |
| ON-GOTO | ON n GOTO m,k,j | Branch on different integer values of "n" to different line numbers "m", "k", |

| Command | Example of Form | Explanation |
|---------|-----------------|---|
| CHAIN | CHAIN "JOE" | Stop execution of the present program and begin execution of program JOE. All data in the present program that were not saved in a file are lost. |
| STOP | STOP | Equivalent to transferring control to the END statement. |
| END | END | Last statement in every program, signals completion of the program. |

4.1.1 Functions

In addition to the usual arithmetic operations of addition (+), subtraction (-), multiplication (*), division (/), and exponentiation (†); BASIC provides the following function capabilities:

| SIN(X) | Sine of X |
|---------|---|
| COS(X) | Cosine of X |
| TAN (X) | Tangent of X |
| ATN(X) | Arctangent of X |
| EXP(X) | e^{X} (e=2.712818) |
| LOG(X) | Log of X (natural logarithm) |
| ABS (X) | Absolute value of X (X) |
| SQR(X) | Square root of X (\sqrt{X}) |
| INT(X) | Greatest integer in X |
| RND(X) | Random number between 0 and 1 is a repeatable |
| | sequence, value of X ignored. |
| SGN(X) | Assign value of +1 if X is positive, 0 if 0, |
| | or -1 if negative. |
| TAB(X) | Controls the position of the printing head on |
| | the Teletype. |
| TIM(X) | TIME + housekeeping function (TSS/8) |
| | - 3 mile 22011 (1887 0) |

NOTE: Trig functions use radians.

4.2 SUMMARY OF BASIC EDIT AND CONTROL COMMANDS

Several commands for editing BASIC programs and for controlling their execution enable you to: delete lines, list your program, save programs on disk, delete or replace old programs on

disk with new programs, call in programs from disk, etc. The commands may be given at any time during the editing phase, and are not preceded by a line number.

| Command | TSS/8 Abbreviation | Action |
|---------|--------------------------|--|
| BYE | BYE | Causes an exit to TSS/8 Monitor, user has left BASIC. |
| CATALOG | CAT | Returns a list of programs which are on file under your account number. |
| COMPILE | COM name | BASIC compiles the program in core and stores it on disk with the given name. |
| DELETE | DEL n n | Delete the line with line number n, an alternate form is to type the line number and the RETURN key. |
| | DEL n, m | Delete the lines with line num- bers n through m inclusive. |
| EDIT | EDI n | Allows the user to search line n for the character c. |
| KEY | KEY | Return to KEY (normal) mode. (See TAPE) |
| LIST | LIS LIS n LIS n, m | List the entire program in core. List line n. List lines n through m inclusive. |
| LLIST | LLIST LLIST n,m | Same as LIST command except output to the line printer, if one is available. |
| NEW | NEW | BASIC will clear core and ask for the new program name. |
| OLD | OLD | BASIC will clear core, ask for the old program name, and re- trieve the program from disk leaving it in core. |

| Command | TSS/8 Abbreviation | Action |
|---------|-----------------------|---|
| REPLACE | REP REP name | Replace the old file on disk with the updated version of the same name currently in core. If a name is not indicated under which BASIC is to store the new version, the old name is retained. |
| RUN | RUN | Compile and run the program currently in core. |
| SAVE | SAV SAV name | Save the contents of user core as the file whose name is indicated. |
| SCRATCH | SCR | Erase the current program from core. |
| TAPE | TAP | Enter TAPE mode, characters typed will not echo on the console paper. |
| UNSAVE | UNS name UNS name, | Delete the named program(s) from the disk. |
| CTRL/C | CTRL/C | Stops a running program, types C and returns to the editing phase. BASIC replies READY. |
| ALTMODE | ALTMODE | Pressing the ALTMODE or ESCAPE key erases all input on the line. |
| | | |

4.3 SUMMARY OF BASIC ERROR MESSAGES

Message

4.3.1 The following error messages may be printed by BASIC during the editing phase:

Explanation

| WHAT? | The editor cannot understand the command just given. |
|--------------------|--|
| BAD FILE NAME | An illegal character was put in the file name |
| CAN'T UNSAVE: name | UNSAVE cannot delete the file with the name given. |

Explanation Message DUPLICATE FILE NAME BASIC cannot SAVE over an existing file, use a different name, or use the REPLACE command. ILLEGAL LINE NUMBER Line number was outside range of 1 to 2046. CAN'T FIND "name" The file name given following OLD CAN'T FIND "name" PROGRAM NAME -- cannot be opened. FOR USER n Either it does not exist or it is CAN'T FIND "name" read protected against this user, IN SYSTEM LIBRARY as indicated. Attempt to LIST or change a BASIC EXECUTE ONLY FILE compiled file. The program specified in response BAD FILE FORMAT to OLD PROGRAM NAME was not acceptable to BASIC. This is generally caused by trying to load a non-BASIC

4.3.2 During input to the editor or when executing an INPUT command, the following messages may be printed in response to input:

(FORTRAN or PALD) program.

| Message | Explanation |
|---------|-------------|

LINE TOO LONG

The line just typed exceeded the available core buffer, retype the line.

\$ DELETED In response to an ALTMODE character the line has been deleted. Retype the line.

(bell-bell) Two bells mean that the previous character was illegal, it is automatically deleted.

Back arrow is printed any time a RUBOUT or SHIFT/O is used, the previous character is deleted.

4.3.3 The following error messages may be typed out by BASIC following a RUN command:

Message Explanation DEF STATEMENT MISSING A function was called which was not defined in a DEF statement. DIMENSION TOO LARGE IN Self explanatory. LINE n FOR WITHOUT NEXT Unmatched FOR statement in program. GOSUB--RETURN ERROR IN Either subroutines are nested too LINE n deeply, or a RETURN was encountered without a previous GOSUB ILLEGAL CHARACTER IN Self explanatory. LINE n ILLEGAL CONSTANT IN Format of a constant in line n is LINE n not valid. ILLEGAL FOR NESTING IN FOR-NEXT loops have been nested too LINE n deeply, or NEXT statements were encountered before the FOR was executed. ILLEGAL FORMAT IN LINE n Illegal syntax for BASIC statement. ILLEGAL SYNTAX IN LINE n Error in expression syntax. ILLEGAL INSTRUCTION IN Statement in line n was not a legal LINE n BASIC command. ILLEGAL LINE NUMBER IN Line number n is outside the range LINE n 1 to 2046. ILLEGAL VARIABLE IN LINE n An array variable was used in line n, where it was not permissible. MISUSED TAB IN LINE n The TAB function may appear only in PRINT statements.

NEXT WITHOUGH FOR IN LINE n

NO END STATEMENT

OUT OF DATA IN LINE n

Self explanatory.

Self explanatory.

Attempt to do a READ past the available data.

Explanation

PROGRAM TOO LARGE

Self explanatory. Try reducing arrays or use fewer variables.

STACK OVERFLOW IN LINE n

Expression too complicated. Try typing it as two separate statements.

SUBSCRIPT ERROR IN LINE n

Negative subscript was calculated for an array.

UNDEFINED LINE NUMBER, LINE n Tried to reference a line which does not exist.

MISUSE OF CHR\$ IN LINE n

The CHR\$ function was used in an invalid manner. CHR\$, like TAB, can appear only in PRINT statements.

BAD VALUE IN CHANGE STATE-MENT AT LINE n

While performing, CHANGE A TO A\$, one of the elements of the array A was found to contain an illegal value.

PROGRAM IS "progname"

This message may immediately follow an error message, to identify the current program in a series of CHAINed programs. If there is no CHAINing, this message will not occur.

PROGRAM NOT FOUND AT LINE n

The file which the user tried to access with a CHAIN statement does not exist in his disk area. The PROGRAM IS message will also occur naming the missing program.

BAD SLEEP ARGUMENT IN LINE n

The argument of the SLEEP command must have a number greater than or equal to \emptyset , and less than or equal to 4095.

ARRAY OR RECORD USED
BEFORE DEFINITION IN
LINE n

The RECORD statement must occur before any reference to it is made. A DIM statement must occur before an array is used. (RECORD and DIM are placed at the beginning of a program.)

Explanation

IMPROPER DIM OR RECORD STATEMENT IN LINE n

Syntax error in DIM or RECORD statement, or an array name that was previously dimensioned is reused.

CAN'T CREATE FILE IN LINE n

An OPEN statement tried to create a file, but there is: (a) no disk space available, (b) no file name specified, or (c) a null string has been given as the file name.

CAN'T DELETE FILE IN LINE n

UNSAVE cannot delete a file. This is usually due to the fact that another user has the file open, or the file is protected with a code $> 2\emptyset$.

UNOPEN DISK UNIT IN LINE n

The user tried to do a GET, PUT, or UNSAVE to device 8 or 9, without a file being previously opened on the device.

DEVICE BUSY IN LINE n

The user tried to OPEN DECtapes \emptyset -7, line printer, or paper tape punch, but the device was unavailable, and there was not ELSE clause in the OPEN statement.

INVALID RECORD NUMBER IN LINE n

The record number must be a number which is greater than or equal to \emptyset and less than or equal to 4095.

For DECtape I/O the maximum record number is limited further by the DECtape size.

INVALID DEVICE NO. IN LINE n

The device number in the file I/O statement is not between Ø and ll inclusive, (or X and ll inclusive where X is a number set by the system manager).

GET BEYOND END OF FILE IN LINE n

Disk file is too small to have a record with the number specified in the GET statement at Line n.

Explanation

GET/PUT ERROR IN LINE n

A hardware error occurred in GET or PUT. (This is usually due to a DECtape unit being write-locked.)

CHAIN TO BAD FILE AT LINE n

The file specified by the CHAIN has an invalid format; it is not a BASIC format file. The "PROGRAM IS . . . " message will follow this error message. The program name will be the name of the bad file.

4.3.4 The following messages are typed out at execution and are non-fatal (i.e., the program continues to execute):

Message

Explanation

IC IN n

Illegal constant in INPUT, retype

the value.

LN IN n

An attempt to compute the logarithm of a number less than or equal to zero. The maximum negative number will be used as the result.

MORE?

Response to INPUT did not contain the number of values requested. Respond by supplying the additional values.

OV IN n

Overflow--value is too large for BASIC to use, the largest possible number will be used instead.

PW IN n

Attempt to raise a negative number to a non-integer power. The absolute value raised to the indicated power will be used instead.

RB IN n

Error in use of TIM function.

SQ IN n

Attempt to compute the square root of a negative number. The square root of the absolute value will be used instead.

Explanation

TOO MUCH INPUT, EXCESS

IGNORED

Response to INPUT contained more values than requested. This message

has no effect on the program.

UN IN n

Underflow--value is too small for BASIC to use, zero will be used

instead.

/Ø IN n

Attempt to divide by zero. The largest possible number will be

given as a result.

4.3.5 The following errors may occur during any BASIC operation:

Message

Explanation

ABORT TBS

A non-recoverable disk error has occurred. BASIC halts.

DISK FULL

ILLEGAL OPERATION IN LINE n

SYSTEM ERROR

There is no room left on the disk; delete some files and try again. These are failures of BASIC. When they occur they should be reported via a SOFTWARE TROUBLE REPORT.

SYSTEM I-O ERROR

TSS/8 disk I/O failure, try again.

APPENDIX A

Various Numerical Equivalents of ASCII Constants

| 00 | 0 | 0 | 000 | null | | 40 | 32 | 63 | 277 | ? |
|-----------|----------|----------|--------------|-------|------------------------|--------|---------|----------|----------|--------|
| 01 | 1 | 32 | 240 | space | | 41 | 33 | 64 | 300 | @ |
| 02 | 2 | 33 | 241 | 1 | | 42 | 34 | 65 | 301 | A |
| 03 | 3 | 34 | 242 | 11 | | 43 | 35 | 66 | 302 | В |
| 04 | 4 | 35 | 243 | # | | 44 | 36 | 67 | 303 | C |
| 05 | 5 | 36 | 244 | \$ | | 45 | 37 | 68 | 304 | D |
| 06 | 6 | 37 | 245 | % | | 46 | 38 | 69 | 305 | E |
| 07 | 7 | 38 | 246 | & | | 47 | 39 | 70 | 306 | F |
| 10 | 8 | 39 | 247 | 1 | | 50 | 40 | 71 | 307 | G |
| 11 | 9 | 40 | 250 | (| | 51 | 41 | 72 | 310 | H |
| 12 | 10 | 41 | 251 |) | | 52 | 42 | 73 | 311 | I |
| 13 | 11 | 42 | 252 | * | | 53 | 43 | 74 | 312 | J |
| 14 | 12 | 43 | 253 | + |) de resedente de sale | 54 | प्रिपे | 75 | 313 | K |
| 15 | 13 | 44 | 254 | , | | 55 | 45 | 76 | 314 | L |
| 16 | 14 | 45 | 255 | - | | 56 | 46 | 77 | 315 | M |
| 17 | 15 | 46 | 256 | • | | 57 | 47 | 78 | 316 | N |
| 20 | 16 | 47 | 257 | / | | 60 | 48 | 79 | 317 | 0 |
| 21 | 17 | 48 | 260 | Ø | | 61 | 49 | 80 | 320 | P |
| 22 | 18 | 49 | 261 | 1 | | 62 | 50 | 81 | 321 | Q |
| 23 | 19 | 50 | 262 | 2 | | 63 | 51 | 82 | 322 | R |
| 24 | 20 | 51 | 263 | 3 | | 64 | 52 | 83 | 323 | S |
| 25 | 21 | 52 | 264 | 4 | | 65 | 53 | 84 | 324 | T |
| 26 | 22 | 53 | 265 | 5 | | 66 | 54 | 85 | 325 | U |
| 27 | 23 | 54 | 266 | 6 | | 67 | 55 | 86 | 326 | V |
| 30 | 24 | 55 | 267 | 7 | | 70 | 56 | 87 | 327 | W |
| 31 | 25 | 56 | 270 | 8 | | 71 | 57 | 88 | 330 | X |
| 32 | 26 | 57 | 271 | 9 | | 72 | 58 | 89 | 331 | Y |
| 33 | 27 | 58 | 272 | • | | 73 | 59 | 90 | 332 | Z |
| 34 | 28 | 59 | 273 | ; | } | 74 | 60 | 91 | 333 | [|
| 35 | 29 | 60 | 274 | < | | 75 | 61 | 92 | 334 | 1 |
| 36 | 30 | 61 | 275 | | | 76 | 62 | 93 | 335 |] |
| 37 | 31 | 62 | 276 | > | Į | 77 | 63 | 94 | 336 ► | † > |
| Str | Str | BASIC | 276 ASCII | ASCII | | Str | Str | BASIC | ASCII | ASCII |
| 37 String | 3 String | IC | II | II | | String | String | | | |
| | | in | င္ပ | char | | | | int | const | char |
| Octal | eci | ter | const | H | | Octal | 9C1: | ret. | 18t | F |
| Ĺ | Decimal | internal | . • | -A | -1- | j., | Decimal | internal | | |

APPENDIX B

| PAGE | TERM | PAGE | TERW |
|-------|---|--------|-------------------------|
| | 8 IN THE MARGIN | 3.27 | CLOSE |
| 1 - 1 | | | CLOSING DECTAPE |
| | ABS(X) | | CLOSING FILES |
| 2.54 | | | COMMAND SUMMARY |
| 2.64 | ALTMO DE | | |
| 2.12 | ARITHMETIC OPERATIONS | 2.64 | COMMAND 'BYE' |
| 2.37 | ARRAYS | 2.53 | CUMMAND 'CATALUG' |
| | ASCII CHARACTERS | 3.16 | COMMAND 'CHANGE' |
| 3.19 | | 3.20 | COMMAND 'CHAIN' |
| | ASCII CONSTANTS | | COMMAND 'CLOSE' |
| 2.15 | ATN(X) | | COMMAND 'COMPILE' |
| AP A | BASIC INTERNAL CODES | 3.7 | COMMAND COMPILE |
| 3.37 | BLANK LINES | 2.20 | COMMAND 'DATA' |
| 2.64 | BYE | 2.21 | COMMAND DEF |
| | CATALOG | 2.63 | COMMAND DELETE |
| 2.63 | | 2.39 | COMMAND 'DIM' |
| 3.20 | CHAIN | 3.4 | COMMAND 'EDIT' |
| 3.16 | CHANGE | J • 43 | COMPAND EDIT |
| 0.00 | CHAR. " | 2.51 | COMMAND 'END' |
| 2.53 | | 2.41 | COMMAND 'FOR' |
| 3.36 | CHAR. | 3.25 | |
| 2.12 | CHAR. "*" | 2.45 | |
| 2.12 | CHAR. "+" | | COMMAND 'GOSUB' |
| 2.12 | CHAR. "-" | 2.48 | COMMAND GOSGE |
| 2.12 | CHAR. "/" | 2.47 | COMMAND 'IF-THEN' |
| 2.9 | CHAR. "0" | 2.47 | |
| | CHAR. "1" | 2.29 | COMMAND 'INPUT' |
| 2.10 | | 3.3 | COMMAND 'KEY' |
| 2.47 | | 2.10 | COMMAND 'LET' |
| 2.47 | CHAR. "<=" | | |
| 2.47 | CHAR. "<>" | 3.14 | |
| 2.47 | CHAR. "=" | 2.62 | COMMAND 'LIST' |
| | | 3.33 | COMMAND 'LLIST' |
| 2.47 | • | 3.33 | COMMAND 'LPRINT' |
| 2.47 | O 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 | 2.63 | |
| 2.54 | CHAR. "ALTMODE" | 2.00 | |
| 2.56 | CHAR. "CTRL/C" | 2.42 | COMMAND 'NEXT' |
| 2.50 | CHAR. "CTRL/L" | 2.63 | COMMAND TOLD |
| 3.5 | UNAR. CIRLY | 3.35 | COMMAND 'ON-GOTO' |
| 3.5 | CHAR. "CTRL/G" | 3.94 | COMMAND OPEN |
| | CHAR. "ESC(APE)" | 3.30 | |
| 2.10 | CHAR. "L" | 3.30 | COMMAND OF EACH LOSS |
| 2.19 | 0" G" . RAHO | 2.31 | COMMAND [PRINT] |
| 2.32 | CHAR. "QUOTE" | 3.25 | COMMAND 'FUT' |
| 2.53 | | 2.18 | COMMAND 'RANDUNIZE' |
| | | 2.26 | COMMAND 'READ' |
| 2.53 | | 3.22 | |
| 3.37 | CHAR. "\" | 0.00 | |
| 2.12 | CHAR. "T" | 2.6 | COMMAND TREMARK |
| | CHAR. "TBS" | 2.61 | CUMMAND 'REPLACE' |
| 2.56 | | 2.27 | |
| 2.56 | CHAR. "TC" | | |
| 3.19 | CHARACTER GENERATOR FUNC | | |
| 3.19 | CHRS (X) | 3.12 | COMP. HAVE ALLO LO GEOM |

| PAGE | TERM | | PAGE | TERM |
|-------------|--|---|------|---------------------------------------|
| 2.49 | COMMAND 'RETURN' | _ | 2.30 | FIGURE 2-3 |
| 2.59 | COMMAND RUN' | 7 | 2.36 | FIGURE 2-4 |
| 2.60 | COMMAND 'SAVE' | | | FIGURE 2-5 |
| | COMMAND 'SCRATCH' | | | FIGURE 2-6 |
| 3.35 | COMMAND 'SLEEP' | | | FIGURE 2-7 |
| | COMMAND 'STOP' | | 2.50 | FIGURE 2-8 |
| 3.1 | CUMMAND 'TAPE' | | 3.3 | FILE EXTENSIONS FILE PROTECTION FILES |
| 2.62 | COMMAND 'UNSAVE' | | 3.9 | FILE PROTECTION |
| | COMMENTS | | 2.58 | FILES |
| J.7 | COMPILE | | 3.22 | FILES OF DATA |
| 2.47 | CONDITIONAL TRANSFER | | 2.2 | FLOWCHART |
| | COS(X) | | 2.15 | FN (BASIC FUNCTIONS) |
| | DARTMOUTH | | 2.15 | FN ABS |
| | DATA | | | FN ATN |
| 3.20 | DATA | | 3.19 | FN CHRS |
| 3.38 | DATA CODES | | 2.15 | FN COS |
| 3.22 | DATA FILES | | | FN DEF |
| 2.8 | DECIMAL NUMBERS | | | FN EXP |
| | DECTAPE FILES | | 2.21 | FN FNX |
| 3.29 | DECTAPE RECORD | | 2.21 | FN FNX |
| 3.4 | DECTAPE USE | | 2.16 | FN INT |
| 2.21 | | | 2.15 | · · · - · · - |
| | DELETE | | 2.17 | |
| | DELETING FILES | | 2.15 | FN SGN |
| 2.43 | DEPTH OF LOOPS | | 2.15 | FN SIN |
| | DEPTH OF NESTING | | 2.15 | FN SCR |
| | DIM | | 2.34 | FN TAB |
| | DIMENSIONING | | | FN TAN |
| | DISK DATA FILES | | 2.20 | FN TIM |
| | DOLLAR SIGN VARIABLES | | 2.21 | FNX(X) |
| 2.9 | E FORMAT EDIT | | | FNX(X,Y,Z) |
| 4.4 | EDIT COMMANDS SHOWS | | 2.41 | |
| 2.60 | EDIT COMMANDS SUMMARY EDITING PHASE | | | FUNC SUMMARY |
| 2.51 | EDITING PRASE | | | FUNCTIONS |
| | | | 2.6 | FUNDAMENTALS |
| 2.54 | ERASING A LINE | | 3.24 | GET |
| 2.55 | ERASING A PROGRAM | | 2.45 | |
| 2.53 | ERROR CORRECTION | | 2.48 | GOSUB |
| 4.4 | ERROR MESSAGES | | | HIEPARCHY |
| 2.54 | ESC(APE) | | 2.20 | HOUSEKEEPING FUNCTION - |
| | ESC(APE) | | 2.47 | IF GO TO |
| | EXP(X) | | 2.47 | IF THEN |
| | FIGURE 2-1 | | 3.1 | IMPLEMENTATION |
| 2.24 | FIGURE 2-2A | | 2.57 | INITIAL DIALOG |
| 2.25 | FIGURE 2-2B | | 2.29 | INPUT |
| * * * * * * | | | | |

| PAGE | TERM | PAGI | E TERM |
|-------------------------------------|--|-------------------|--|
| 2.26 3.13 2.16 2.16 | INPUT/OUTPUT INPUTTING STRINGS INT(X) INTEGER FUNCTION INTEGER SWITCH | 2.3 3.3 3.1 | OTHER USER'S FILES OUTPUT COMMAND PAPERTAPE OUTPUT PAPERTAPE PROGRAMS PAPERTAPE USE |
| 2.8 3.38 2.1 3.3 | INTEGERS INTERNAL CODES INTRODUCTION KEY KEYBOARD FORMAT | 2.3 3.3 3.1 | PARENTHESES PRINT PRINTER USE PRINTING STRINGS PROGRAM COPY |
| 2.64 2.10 3.1 | LAST LINE IN PROGRAM LEAVING BASIC LET LIMITATIONS OF TSS/8 LINE ERASURE | 2.5 2.5 3.9 | PROGRAM CHAINING PROGRAM ERASURE PROGRAM FILES PROTECTION "SWITCHES" PROTECTION OF FILES |
| 3.20 3.14 2.62 | LINE PRINTER LINKING PROGRAMS LINPUT LIST LISTS | 2.1 | 25 PUT 17 RANDOM NUMBER FUNCTION 18 RANDOMIZE 26 READ 31 READING DECTAPE FILES |
| 2.15 2.4 2.57 | LLIST LOG(X) LOGIN PROCEDURE LOGIN PROCEDURE LOOPS | 3. 2. 3. | READING DISK FILES READING STRINGS REAL NUMBERS RECORD RECORD |
| | LPRINT MARGINAL "8" | 2. 2. 3. | 6 REMARK 61 REPLACE 27 RESTORE 12 RESTORE 12 RESTORE |
| 2.43 2.52 2.63 2.42 2.8 | NESTING LOOPS NESTING SUBROUTINES NEW NEXT NUMBERS | 2. 2. 2. | .12 RESTORE* .49 RETURN .17 RND(X) .59 RUN .60 SAVE |
| 3.35 3.24 | NUMERIC DATA CODES OLD ON GOTO OPEN OPEN | 2. 2. 2. | .55 SCRATCH .15 SGN(X) .15 SIGN FUNCTION .15 SIN(X) .35 SLEEP |
| 3.30 3.24 3.30 | OPEN OPEN ELSE OPENING DISK FILES OPENING DECTAPE FILES ORDER OF OPERATION | 2. 2. 2. | .14 SPACES .61 SPACES .15 SQR(X) .41 STEP .51 STOP |

| PAGE | TERM | PAGE | TERM |
|----------------------|--|------|------|
| 3.39 AP A 3.13 | STRING DECIMAL CONSTANTS STRING INPUT STRING OUTPUT | : | |
| 3.11 2.51 2.48 | STRING OUTPUT STRINGS SUBROUTINE LOCATION SUBROUTINES SUBSCRIPTS | | |
| 4.4 2.58 2.34 | SUMMARY OF BASIC SUMMARY OF EDIT COMMANDS SYSTEM FILES TAB(X) TABULATION | | |
| 3•1 2•9 2•32 | TAN(X) TAPE TELETYPE CONSULE TEXT PPINTING TIM(X) | | |
| 2.41 2.45 2.62 | TIME FUNCTION TO TRANSFER OF CUNTRUL UNSAVE UNSAVE | | |
| 2.4 2.21 2.58 | UNSAVING FILES USE OF THE MANUAL USER DEFINED FUNCTIONS USER FILES VARIABLES | | |
| | WORKING WITH STRINGS WRITING DECTAPE FILES WRITING DISK FILES | | |