os/8 handbook

digital

digital os/8 handbook

digital

OS/8 handbook

prepared
by
software documentation
software engineering department
digital equipment corporation

pdp8 handbook series

FIRST PRINTING, APRIL 1974

ERROR REPORTING

If you find any errors in this handbook, or if you have any questions or comments concerning the clarity or completeness of this handbook, please direct your remarks to:

Digital Equipment Corporation
Software Communications, Parker Street
Maynard, Massachusetts 01754

ADDITIONAL COPIES

Additional copies of this handbook may be obtained by ordering DEC-S8-OSHBA-A-D. Please send your order to the address below.

Digital Equipment Corporation
Communications Services, Parker Street
Maynard, Massachusetts 01754

introduction

The OS/8 Operating System is a sophisticated operating system designed for the PDP-8/E computer. This system permits use of a wide range of peripherals and all available core up to 32K. OS/8 offers a versatile Keyboard Monitor that supervises a comprehensive library of system programs. These features make OS/8 a significant improvement in small computer operating systems.

OS/8 SYSTEM PROGRAMS

Besides the Monitor facilities, OS/8 includes a library of powerful system programs which allow the user to do program development using FORTRAN II or assembly language. A brief summary of the system programs follows:

1. Concise Command Language (CCL)
   CCL provides the user with an extensive set of terminal commands. Typical commands available in CCL include: COPY, DIRECTORY, HELP, RENAME, LIST, DELETE, etc.
2. Symbolic Editor (EDIT)
   EDIT is used to create or modify source files for use as input to language processing programs such as PAL8, SABR, or FORTRAN. EDIT contains powerful text manipulation commands for quick and easy editing.
3. PAL8 Assembler
   PAL8 is the assembler for the OS/8 system. PAL8 accepts source files in the PAL language and generates absolute binary files as output. PAL8 also generates listing files which can be used as input to CREF.
4. Peripheral Interchange Program (PIP)
   PIP allows the user to transfer files between devices which are in the OS/8 system. Complete file and directory maintenance functions are available in PIP.

5. Absolute Binary Loader (ABSLDR)
   ABSLDR accepts the binary output files produced by PAL8 and loads them into core.

6. Octal Debugging Technique (ODT)
   ODT is a powerful octal debugging tool. All of the features of older versions of ODT are implemented, but the OS/8 version is designed so that no user core is needed.

7. File-Oriented Transfer Program (FOTP)
   FOTP allows the user to transfer groups of files between two OS/8 file-structured devices with minimal terminal interaction and device overhead, e.g., all ASCII files can be transferred between a DECtape and disk with one terminal command.

8. Cross Reference (CREF)
   CREF operates on the listings produced by PAL8 and SABR. It produces a sequence numbered listing and a table indicating where each user-defined tag and literal is referenced.

9. DIRECT
   DIRECT allows the user to print extended directory listings.

10. BOOT
    The BOOT program loads the standard hardware bootstraps into core.

11. Cassette/Magtape Positioner (CAMP)
    CAMP allows the user to manipulate cassettes and magnetic tapes.

12. Resources (RESORC)
    RESORC integrates system monitor tables and prints a listing of active device handlers.

13. Magtape/Cassette PIP (MCPIP)
    MCPIP is a file transfer program to be used with cassettes and magnetic tapes.

14. PIP10
    PIP10 is a file transfer program which reads and writes PDP-10 ASCII DECtape files using a TC08 or TD8E DECtape controller.

15. FORTRAN II

    The OS/8 system contains an extensive and powerful FORTRAN package, consisting of the FORTRAN compiler, SABR assembler, Linking Loader, and Library function routines. Some of the many features of FORTRAN II are:

    a. FORTRAN II is very easy to use. If desired, a FORTRAN source program can be compiled, loaded, and executed with a single terminal command.
    b. Implied DO loops are permitted in FORTRAN II.
    c. FORTRAN II contains facilities to do program chaining; this technique can be used to increase the effective program size.
    d. Device independent I/O is available, as well as the standard devices (console terminal, high-speed reader/punch, card reader, and line printer).

16. Library Setup (LIBSET)

    OS/8 LIBSET allows the user to create his own FORTRAN II run-time libraries. The standard library supplied with the system is LIB8. By using LIBSET, the user can write his own routines in SABR and create a library.

17. System Build (BUILD)

    BUILD allows rapid and easy alteration of the device configuration in the system. New devices can be inserted by simple keyboard commands. BUILD also makes interfacing user-coded device handlers a quick and easy job.

OTHER PROGRAMS AVAILABLE WITH OS/8

In addition to the standard OS/8 system programs listed previously, the following programs are available with OS/8:

BASIC
BATCH
TECO
FORTRAN IV

BASIC, BATCH, and TECO are provided in a single extension kit. OS/8 BASIC is an interactive language with a variety of applications. It contains such features as chaining, string manipulation, and file-oriented input/output. Also included with BASIC are certain functions for use on the LAB-8/E.

OS/8 BATCH provides the user with a batch processing monitor that is integrated into the OS/8 Monitor structure. The system is organized in such a way that it may be used in either a keyboard input configuration or as a batch stream processor. BATCH permits the user to prepare his job on punched cards, high-speed paper tape, or the OS/8 system device and leave it for the computer operator to start and run.

OS/8 TECO is a powerful text editing and correcting program that runs under the OS/8 operating system. TECO may be used to edit any ASCII text such as program listings, manuscripts, correspondence, etc.

OS/8 FORTRAN IV provides full ANSI FORTRAN IV under the OS/8 operating system. The system is highly optimized with respect to memory requirements, and an overlay feature is included that can permit programs requiring up to 300K of virtual storage to run on a PDP-8 or PDP-12. The library functions permit the user to access a number of laboratory peripherals, to evaluate a number of transcendental functions, to manipulate alphanumeric strings, and to output to a standard incremental plotter.

OS/8 I/O DEVICES

OS/8 provides true device-independence. For the first time on a PDP-8 computer, programs can be written without concern for specific I/O devices. In running a program, the user can select the most effective I/O devices available. Further, if the system configuration is altered, programs need not be rewritten to take advantage of the new configuration.

The OS/8 system controls the copying of data from any medium to any other medium by means of subroutine calls to execute I/O routines. Logical names can be assigned to devices within the system to enable symbolic referencing of devices.

Variable length I/O buffers can be specified by the user program. Large buffers ensure efficient use of storage devices and a minimum of time spent in data transfer operations by minimizing disk and tape motion. OS/8 takes full advantage of the RK8E disk pack for fast bulk storage, yet full system services are possible with a single DECtape.

HARDWARE CONFIGURATIONS

The OS/8 system can operate with a wide variety of devices as the system device.[1] The devices which can be used are:

TC01/TC08 DECtape
LINCtape (PDP-12)
TD8E DECtape
DF32/RF08 disk
RK8E disk
RK8 disk

TD8E DECtape can be used either with 12K words of core memory or with 8K words of core memory and 256 words of Read-Only-Memory (ROM).

If DF32 is the system device, at least 64K (2 platters) must be available. In addition, if disk is the system device, cassettes or the high-speed reader/punch provides a very useful tool.

The minimum OS/8 configuration is a PDP-8 series computer with 8K words of core memory, one DECtape used as the system device, and a console terminal. A multiple DECtape system performs appreciably faster than a single DECtape system. The multiple DECtape system reduces DECtape motion since it is possible to copy directly (without intermediate searching) from the system DECtape to another DECtape (or vice versa) when editing or assembling.

A typical medium-sized system might contain a PDP-8/E with at least 8K words of core memory, TD8E DECtape and control, and an RK8E disk pack and control. A disk system offers the additional convenience of easy and fast access to files and large amounts of storage.

Up to 15 devices can be interfaced to a single OS/8 system. These optional devices include:

As many as 8 DECtape units (TC01/TU55, TC08/TU56, or TD8E/TU56.

TA8E/TU60 cassette units
TM8E/TU10 magnetic tape units

[1] The term system device refers to the device on which the OS/8 system resides and which it utilizes for system functions. Thus, DECtape unit 0 is the system device for a DECtape-based system. A nonsystem device is any peripheral not specifically used for system functions, such as LPT:, PTR:, DTA2:, etc.

High-speed paper tape reader/punch.
Up to four RK8E disks.
Up to four RK8 disks.
Up to four RS08 disks.
Up to four DF32 disks.
Card reader (optical mark or punched cards).
Line printer.
PDP-12 LINCtape.
PDP-12 scope.

Any other device for which it is impossible to write a device handler in one or two pages of core.

SYSTEM SOFTWARE COMPONENTS

The main software components of the OS/8 system are five:

Keyboard Monitor
Command Decoder
Library of system programs
Device handlers
User-Service Routine (USR)

The Keyboard Monitor provides communication between the user and the OS/8 executive routines by accepting commands from the console terminal. The commands enable the user to create logical names for devices, run system and user programs, save programs, and call ODT.

The Command Decoder allows the user to communicate with a system library program by accepting a command string from the keyboard indicating input/output files. Following the keyboard command to run a system library program, the Command Decoder prints an asterisk (*) and then accepts the command line containing the files to be used as input, file name, and destination of output, etc.

The library of system programs contains the programs mentioned previously and any of the extension programs chosen by the user.

Device handlers are subroutines designed to transfer data to and from peripheral devices. OS/8 is able to interface with as many as 15 different peripherals at a time. During system generation, device handlers become an integral part of the system; both system and user programs have access to any available device. (The BUILD program allows quick and easy alteration of any available device.)

The User Service Routine (USR) control the directory operations for the OS/8 system. A program can use the USR by means of standard subroutine calls such as those used to activate device handler subroutines. Some of the functions performed by the USR are loading device handlers, searching file directories, creating and closing output files, calling the Command Decoder, and chaining of programs. The details on the operation and use of the USR are contained in the OS/8 Software Support Manual (DEC-S8-OSSMB-A-D). For normal OS/8 usage, the USR function is unseen by the user and need be of no concern.

When OS/8 is operating, the Command Decoder, Keyboard Monitor, and USR are swapped into core from the system device as required, and when their operation has been completed, the previous contents of core are restored.

The core-resident portion of OS/8 is extremely small (256 words), allowing for a maximum use of core by user programs.

USING THE OS/8 HANDBOOK

The OS/Handbook provides a complete user's guide for the OS/8 operating system and system programs. The handbook is divided into three parts. Part one contains detailed instructions for getting a new OS/8 system running. Also included in part one are the fundamentals of OS/8, including the Keyboard Monitor, Concise Command Language (CCL), Command Decoder, the Absolute Loader (ABSLDR), Octal Debugging Technique (ODT), and Peripheral Interchange Program (PIP). The user must have a complete understanding of the material contained in Chapter 1 to use the OS/8 operating system.

Part two contains complete descriptions of the OS/8 utility programs. These programs allow the user to perform a variety of editing, I/O transfers, system generation, and file-oriented operations.

Part three describes the assemblers available with OS/8: PAL8, SABR, FLAP, and RALF.

Part four describes the higher-level languages which can be run under OS/8: BASIC, FORTRAN II, and FORTRAN IV.

contents

CHAPTER 1 OS/8 FUNDAMENTALS

CHAPTER 2 UTILITY PROGRAMS

CHAPTER 3 PAL8

CHAPTER 4 SABR

CHAPTER 5 FLAP/RALF

CHAPTER 6 BASIC

CHAPTER 7 FORTRAN II

CHAPTER 8 FORTRAN IV

APPENDICES

LIST OF TABLES

LIST OF ILLUSTRATIONS

OS/8 Fundamentals

getting on line
keyboard monitor
command decoder
ccl
editor
pip
absldr
odt

chapter 1

os/8 fundamentals

GETTING ON LINE WITH OS/8

OS/8 software is distributed to the user in a form appropriate for his particular hardware configuration. The general system categories are DECtape (LINCtape), cassette, and paper tape. This section provides the information that the user of any of these types of systems needs to start using OS/8. The procedures for bootstrapping a disk system and for restarting OS/8 are also contained in this section. To get on line with OS/8 when the system is first installed, refer to the section on the specific distribution media.

DECtape Systems

This category includes TC01/TC08, TD8E, and LINCtape (PDP-12) hardware configurations. Since the software is supplied on a system DECtape (or LINCtape), it is not necessary to build an initial system, as it is when using cassettes or paper tapes.

Two tapes are distributed with each DECtape (LINCtape) system. System Tape #1 contains the system programs and all OS/8 Monitor functions. System Tape #2 contains TDINIT.SV (used in TD8E system initialization) and two TD8E DECtape monitor images (8K ROM and 12K). Other files on this second tape are the device handlers in a format suitable for the OS/8 BUILD program. Each file contains a handler for a specific device type. These files are to be used as input for the LOAD command in BUILD and are described in the BUILD section of Chapter 2. In addition to these files, the tape also contains relocatable binary files of the FORTRAN II library subroutines. LIBSET, the FORTRAN II librarian, is used to create a FORTRAN II library as described in Chapter 7.

Finally, the tape contains several OS/8 help files (.HL extension). These help files are intended to provide the user with a

quick command summary for most OS/8 programs. They can be printed with either OS/8 PIP or the CCL command HELP.

TC01/TC08 DECTAPE USERS

The following short procedure is used to start OS/8 on a TC01/TC08 system:

1. Mount the system DECtape (DEC-S8-OSYSB-A-UC1) on unit 0 (this appears as unit 8 on some DECtape units), making certain to wind at least 10 feet of tape onto the empty reel. Set the tape unit switches to REMOTE and WRITE LOCK.
2. Bootstrap the OS/8 DECtape by following one of two methods. If the system includes an MI8-E hardware bootstrap option:
   a. Place the terminal on line. Raise the SING STEP switch on the PDP-8/E console. Press the CONT switch. Then lower and raise the HALT switch. At least one console indicator lamp should light.
   b. Having mounted the OS/8 System Tape #1 on unit 0 as described above, lower and raise the SW switch on the left side of the console.

   If the system does not include a hardware bootstrap, this procedure will have no effect. In this case, execute step 1 above, place the terminal on line, and then perform the switch manipulations shown in Table 1-1. For each step in the table, place each of the PDP-8/E console SWITCH REGISTER switches numbered 0 to 11 either in the up position if the corresponding table entry is a 1, or in the down position if the corresponding table entry is a 0. When all 12 switches have been set to correspond to a line in the table, follow the instructions in the right hand column and proceed to the next line. In step 4, for example, place switches 2, 4, 7, and 10 in the up position; place switches 1, 3, 5, 6, 8, 9, and 11 in the down position; lift the DEP switch; and proceed to step 5. The table also includes octal values of the binary switch settings for the benefit of users familiar with octal numbers.

Table 1-1 TC01/TC08 DECtape Bootstrap

STEP #	OCTAL VALUES	SWITCH REGISTER SETTING				AND THEN
		012	345	678	91011	
1	0000	000	000	000	000	press EXTD ADDR LOAD
2	7613	111	110	001	011	press ADDR LOAD
3	6774	110	111	111	100	lift DEP key
4	1222	001	010	010	010	lift DEP key
5	6766	110	111	110	110	lift DEP key
6	6771	110	111	111	001	lift DEP key
7	5216	101	010	001	110	lift DEP key
8	1223	001	010	010	011	lift DEP key
9	5215	101	010	001	101	lift DEP key
10	0600	000	110	000	000	lift DEP key
11	0220	000	010	010	000	lift DEPKey
12	7754	111	111	101	100	press ADDR LOAD
13	7577	111	101	111	111	lift DEP key
14	7577	111	101	111	111	lift DEP key
15	7613	111	110	001	011	press ADDR LOAD and press CLEAR and press CONT

Either bootstrapping procedure first rewinds the DECtape on unit 0 to the end zone, then starts it moving forward, reading block 0 into locations beginning at 7600 in field 0. In block 0 of the DECtape is a larger bootstrap which continues reading the tape, installing the resident Monitor code, and finally turning control over to the OS/8 Keyboard Monitor.

3. DECtape unit 0 will rock and the console terminal will respond by printing a dot (.) at the left margin. At this point, OS/8 is active; DECtape unit 0 must be set to WRITE ENABLE.

NOTE

If the terminal does not respond properly, check that the bootstrap was loaded correctly, that unit 0 is selected and set to REMOTE, that the correct tape is mounted, and that the terminal is set to REMOTE or LINE. If trouble persists, contact the local Digital sales office.

TD8E DECTAPE USERS

OS/8 supports TD8E DECtape hardware in two configurations: TD8E DECtape and 12K or more core, and TD8E DECtape and 8K core and 256-word Read-Only-Memory (ROM).

TD8E DECtape users must run a special initialization program before OS/8 can be used. This program need only be run once to create the proper configuration; thereafter, the appropriate TD8E bootstrap (discussed shortly) can be used to start OS/8.

TD8E Initialization Program

Use the following procedures to initialize the TD8E DECtape system.

1. Mount the binary DECtape (DEC-S8-OSYSB-A-UC2) on DECtape unit 0. Set the tape unit switches to REMOTE and WRITE LOCK.
2. Turn the console terminal to LINE or REMOTE.
3. Execute one of the TD8E bootstraps (see *TD8E Bootstraps* in this section).
4. When the bootstrap is executed correctly, the message:

```
TD8E INITIALIZER PROGRAM VERSION 4
```

is printed on the terminal. Then depending upon which type of TD8E configuration is present, one of the following messages is printed to indicate the system on which OS/8 will be built.

a. `8K ROM SYSTEM`
   is printed if the user has the 256-word ROM.

b. `12K SYSTEM`

   is printed if the user has no ROM but does have 12K or more of core memory.

NOTE

If neither the ROM nor 12K of memory exists, the message:

```
NEED ROM OR 12K
```

appears, and the machine halts. This indicates that the configuration is not suitable for running the TD8E version of OS/8.

5. After the message specifying the hardware configuration (a or b above), the following instructions to the user appear:

```
MOUNT A CERTIFIED DECTAPE ON UNIT 1 WRITE-ENABLED
ALWAYS KEEP ORIGINAL SYSTEM DECTAPES WRITE-LOCKED
STRIKE A CHARACTER TO CONTINUE
```

   Perform the specified operations. At this point, the current OS/8 Monitor is written onto a blank DECtape on unit 1. Note that the original tape (on unit 0) is not written upon.

6. When the copy operation is complete, the following instructions are printed:

```
DISMOUNT TAPE #2 FORM UNIT 0 AND SAVE IT
MOUNT ORIGINAL SYSTEM TAPE #1 ON UNIT 0
PREPARE TO COPY FILES OVER
STRIKE A CHARACTER TO CONTINUE
```

   The system programs will now be copied from System Tape #1 (DEC-S8-OSYSB-A-UC1) to the tape being created. Perform the specified operations and type any character except CTRL/Z to continue. PREPARE TO COPY FILES OVER means to expect copying to take place; no additional preparation is implied. The following message is printed:

```
COPYING FILES FROM UNIT 0 TO UNIT 1
```

   and the system copies the files and updates the DECtape directory.

NOTE

If the user wishes to perform nonstandard special processing, he can respond with a CTRL/Z to the preceding dialogue. If CTRL/Z is typed, the following messages appear:

```
TYPE 1 TO COPY FILES FROM UNIT 0 TO UNIT 1
TYPE 2 TO ZERO THE DIRECTORY OF UNIT 1
TYPE 3 TO LEAVE THE DIRECTORY OF UNIT 1 ALONE
STRIKE A CHARACTER TO CONTINUE
```

Reply with either a 1, 2, or 3 (which will not echo) to indicate the desired option. Typing any character other than those indicated will repeat the request message. One of the following confirmatory messages will appear, to indicate the options 1, 2, or 3, respectively:

```
COPYING FILES FROM UNIT 0 TO 1
ZEROING THE DIRECTORY ON TAPE UNIT 1
DIRECTORY ON UNIT 1 PRESERVED
```

7. When the files have been copied, the following instructions appear:

```
REMOVE AND SAVE TAPE ON UNIT 0
TAKE NEW TAPE (ON UNIT 1) WHICH WAS JUST CREATED
AND PLACE IT ON UNIT 0
IT IS YOUR NEW OS/8 SYSTEM TAPE
STRIKE A CHARACTER TO CONTINUE
```

Remove the original OS/8 tape and save it for later use. Set DECtape unit 0 to WRITE-ENABLE, and type any character to continue. The tape on unit 0 will be initialized to a TD8E configuration.

When the initialization is completed, a dot (.) is printed at the left margin of the terminal. OS/8 is active on a TD8E based system.

TD8E Initialization Error Messages

The messages listed in Table 1-2 may appear during the TD8E initialization process.

Table 1-2 TD8E Initialization Error Messages

Message	Meaning
FATAL IO ERROR	Unable to read from newly copied system tape.
MOUNT CORRECT TAPE ON UNIT 0	Cannot copy tape currently mounted.
NEED ROM OR 12K	Improper hardware configuration.

Table 1-2. TD8E Initialization Error Messages (Cont.)

Message	Meaning
NOT ORIGINAL OS/8 SYSTEM TAPE #2	The tape copied from was not an original OS/8 tape supplied by Digital.
STRIKE A CHARACTER TO CONTINUE	An I/O error occurred on the DECtape. Type any character to retry the operation.
TYPE ANY OTHER CHARACTER TO RETRY THIS I/O OPERATION	First retry failed. Type any other character to retry another time.
TYPE A TO ABORT AND START OVER AGAIN	Return to Step 1.

TD8E Bootstraps

8K ROM Bootstrap (PDP-8/E)

1. Set the switch register on the PDP-8/E console to 7470 (octal), i.e., set switches 0, 1, 2, 3, 6, 7, and 8 in the up position, and set switches 4, 5, 9, 10 and 11 in the down position.
2. Raise the SING STEP switch. Lower and raise the HALT switch.
3. Press the EXTD ADDR LOAD, ADDR LOAD, CLEAR, and CONT switches. The tape bootstrap will be executed and a message will be printed (if initializing) or the OS/8 Keyboard Monitor will print a dot (.) to indicate that it is active. If initializing, set DECtape unit 0 to WRITE-LOCK. If OS/8 is already active, set DECtape unit 0 to WRITE ENABLE.

12K TD8E Bootstrap

The contents of the 12K TD8E bootstrap are included in Table 1-3.

The tape bootstrap will be executed and a message will be printed (if initializing) or the OS/8 Keyboard Monitor will print a dot (.) to indicate that it is active. If initializing, set DECtape unit 0 to WRITE-LOCK. If OS/8 is already active, set DECtape unit 0 to WRITE ENABLE.

Table 1-3 12K TD8E DECtape Bootstrap

STEP #	OCTAL VALUES	SWITCH REGISTER SETTING				AND THEN
		012	345	678	91011	
1	7300	111	011	000	000	press ADDR LOAD and press EXTD ADDR LOAD
2	1312	001	011	001	010	lift DEP key
3	4312	100	011	001	010	lift DEP key
4	4312	100	011	001	010	lift DEP key
5	6773	110	111	111	011	lift DEP key
6	5303	101	011	000	011	lift DEP key
7	6777	110	111	111	111	lift DEP key
8	3726	011	111	010	110	lift DEP key
9	2326	010	011	010	110	lift DEP key
10	5303	101	011	000	011	lift DEP key
11	5732	101	111	011	010	lift DEP key
12	2000	010	000	000	000	lift DEP key
13	1300	001	011	000	000	lift DEP key
14	6774	110	111	111	100	lift DEP key
15	6771	110	111	111	001	lift DEP key
16	5315	101	011	001	101	lift DEP key
17	6776	110	111	111	110	lift DEP key
18	0331	000	011	011	001	lift DEP key
19	1327	001	011	010	111	lift DEP key
20	7640	111	110	100	000	lift DEP key
21	5315	101	011	001	101	lift DEP key
22	2321	010	011	010	001	lift DEP key
23	5712	101	111	001	010	lift DEP key
24	7354	111	011	101	100	lift DEP key
25	7756	111	111	101	110	lift DEP key

Table 1-3 12K TD8E DECtape Bootstrap (Cont.)

STEP #	OCTAL VALUES	SWITCH REGISTER SETTING				AND THEN
26	7747	111	111	100	111	lift DEP key
27	0077	000	000	111	111	lift DEP key
28	7400	111	100	000	000	lift DEP key
29	7300	111	011	000	000	press ADDR LOAD and press CLEAR and press CONT

Both the 8K ROM and 12K TD8E bootstraps perform the same function, reading record 0 of the system tape into memory and then starting it at location 7400 in field 0. The code that is read into 7400 is a larger bootstrap which installs all resident tables and then turns control over to the OS/8 Keyboard Monitor or the TD8E initialization program. The 12K system must move down to tape block 154 to accomplish the full bootstrap, which explains the extra tape motion.

When the TD8E system (either 8K ROM or 12K) is initialized, only TD8E DECtapes 0 and 1 (DTA0, DTA1) are available on the system. The others (DTA2-DTA7) are not in the system. To make other drives available, the user must run the BUILD program. Reference the BUILD section of Chapter 2 for details concerning reconfiguring a system.

LINCTAPE (PDP-12 USERS)

The following is the bootstrap procedure for PDP-12 systems:

1. Mount the system LINCtape (DEC-12-OSYSB-A-AC1) on LINCtape unit 0. Set the LINCtape switches to WRITE LOCK and REMOTE. Set the terminal to LINE or to REMOTE.
2. Set the left switches to 0700. Set the right switches to 0000. Set the MODE key to LINC.
3. Press I/O PRESET.
4. Press DO.

   The LINCtape bootstrap will be executed, causing unit 0 to move. When tape movement stops, ensure that the AC contains -1 (has all lights on). If the AC does not contain -1, return to step 1 above.

5. Press the START 20 key.
   The LINCtape on unit 0 will move again, and a dot (.) will be printed at the left margin of the terminal. OS/8 is now active.
6. Set LINCtape unit 0 to WRITE ENABLE.

Building OS/8 From Cassette

When OS/8 software is supplied on cassettes, the BUILD system library program is used to create the initial OS/8 system. The following procedures are used to build OS/8 onto a mass storage device.

1. The OS/8 cassette containing BUILD (DEC-S8-OSYSB-A-TC1) supplied by Digital is WRITE protected (lugged red tabs expose write protect holes). Open the locking bar on the right side of cassette transport unit 0 by pushing it to the right. Hold the cassette so that the DIGITAL trademark in large letters is upright and to the front. Insert the cassette into transport unit 0, rotating it over the drive sprockets without forcing it, so that the locking bar closes over the back edge.
   Press the rewind button on the cassette transport unit once to rewind the tape to the beginning of its leader/trailer segment. When the unit stops moving, the tape is positioned for data transfer operations.
2. Bootstrap the OS/8 cassette by following one of two methods.
   If the system includes an MI8-E hardware bootstrap option:
   a. Place the terminal on line. Raise the SING STEP switch on the PDP-8/E console. Press the CONT switch. Then lower and raise the HALT switch. At least one console indicator lamp should light.
   b. Having mounted the OS/8 system cassette on unit 0 as described above, lower and raise the SW switch on the left side of the console.

   If the system does not include a hardware bootstrap, this procedure will have no effect. In this case, execute step 1 above and then perform the switch manipulations in Table 1-4. For each step in the table, place each of the PDP-8/E console SWITCH REGISTER switches numbered 0 to 11 either in the up position if the corresponding table entry is a 1, or in the down position if the corresponding table entry is a 0. When all twelve switches have been set to correspond to a line

in the table, follow instructions in the right hand column and proceed to the next line. In step 3, for example, place switches 2, 4, 9, and 10 in the up position; place switches 0, 1, 3, 5, 6, 7, 8, and 11 in the down position; lift the DEP switch; and proceed to step 4. The table also includes octal values of the binary switch settings for the benefit of users familiar with octal numbers.

Table 1-4 Cassette Bootstrap

STEP #	OCTAL VALUES	SWITCH REGISTER SETTING				AND THEN
		012	345	678	91011	
1	4000	100	000	000	000	press ADDR LOAD and press EXTD ADDR LOAD
2	1237	001	010	011	111	lift DEP key
3	1206	001	010	000	110	lift DEP key
4	6704	110	111	000	100	lift DEP key
5	6706	110	111	000	110	lift DEP key
6	6703	110	111	000	011	lift DEP key
7	5204	101	010	000	100	lift DEP key
8	7264	111	010	110	100	lift DEP key
9	6702	110	111	000	010	lift DEP key
10	7610	111	110	001	000	lift DEP key
11	3211	011	010	001	001	lift DEP key
12	3636	011	110	011	110	lift DEP key
13	1205	001	010	000	101	lift DEP key
14	6704	110	111	000	100	lift DEP key
15	6706	110	111	000	110	lift DEP key
16	6701	110	111	000	001	lift DEP key
17	5216	101	010	001	110	lift DEP key
18	7002	111	000	000	010	lift DEP key
19	7430	111	100	011	000	lift DEP key
20	1636	001	110	011	110	lift DEP key
21	7022	111	000	010	010	lift DEP key
22	3636	011	110	011	110	lift DEP key
23	7420	111	100	010	000	lift DEP key
24	2236	010	010	011	110	lift DEP key
25	2235	010	010	011	101	lift DEP key
26	5215	101	010	001	101	lift DEP key
27	7346	111	011	100	110	lift DEP key
28	7002	111	000	000	010	lift DEP key
29	3235	011	010	011	101	lift DEP key
30	5201	101	010	000	001	lift DEP key
31	7737	111	111	011	111	lift DEP key

Table 1-4 Cassette Bootstrap (Cont.)

STEP #	OCTAL VALUES	SWITCH REGISTER SETTING				AND THEN
32	3557	011	101	101	111	lift DEP key
33	7730	111	111	011	000	lift DEP key
34	4000	100	000	000	000	press ADDR LOAD key and press CLEAR and press CONT

Either bootstrapping procedure should cause the system cassette to move and BUILD to print a $ at the left margin of the console terminal. If there is no response, check that the system cassette is properly mounted on transport unit 0 and repeat the bootstrapping procedure, paying particular attention to the switch manipulations. Be careful not to bounce the DEP switch.

3. When BUILD prints:

   $

   respond with the system device on which OS/8 is to be built. (At this point, the usual command editing features of BUILD are available; see Table 2-10 in the BUILD section of Chapter 2.) This response must be in the following form:

   $SYS dev=n

   where "dev" represents one of the legal replies taken from Table 1-5. The "n" is optional and need only be used to indicate the number of physical disk platters which are present if the system device is RF08 or DF32. The possible replies and the maximum value of n which can be used for each one are indicated below.

Table 1-5 System Devices

Device	Maximum n
DF32 (DF32 disk)	4
RF08 (RF08 disk)	4
RK8 (RK8 disk)	1
RK8E (RK8E disk)	1

n must be a digit in the range 1 to 4. If no value for n is specified, a value of 1 is assumed. If a response other than a digit is entered, the message:

```
?SYNTAX
```

is printed and the SYS command must be typed again. If n is specified as a digit but is too large for the device specified, the SYS command must be retyped. For example:[1]

```
$SYS RF08=5
?PLAT
$SYS RF08=4
```

4. When a correct SYS command has been entered, e.g.,

```
$SYS RK8E
```

BUILD prints another $. At this time, insert the desired devices for the initial system. The minimum system for cassettes must have inserted the terminal handler, the mass storage device, and the cassette handlers. (See the BUILD section of Chapter 2 for detailed information.)
In response to the $ printed by BUILD (indicated here by an underline), type the following; each command line should be followed by typing the carriage return key.

```
$IN TA8A:CSA0-1          (cassette unit 0, drives 0 and 1)
$IN KL8E:TTY             (terminal keyboard)
```

5. The user should also specify the device that is to be the default mass storage device by entering the DSK command. For example:

```
$DSK=SYS
```

Any device other than SYS (or carriage return) specified in the DSK command must be the permanent name of a device which appeared in one of the INSERT commands.

[1] Characters printed by the system are underlined to eliminate confusion with characters typed by the user.

6. When all desired devices have been entered with INSERT commands, type the following in response to the $:

```
$BUILD
```

BUILD responds by printing:

```
LOAD OS/8:
```

type CSA0, followed by carriage return, in response to this message, i.e.,

```
LOAD OS/8: CSA0
```

BUILD loads and writes the various parts of OS/8 onto the system device. If a SYS ERR message occurs at any time during the load, ensure that the system device is write-enabled and press the CONT switch to retry. If the retry is unsuccessful, return to step 2.

7. After writing OS/8, BUILD prints:

```
LOAD CD:
```

Respond with a carriage return. BUILD loads the Command Decoder from cassette unit 0 and writes it onto the system device.

8. When BUILD responds with another $, type the following:

```
$BOOT
```

to initiate the final system creation process. BUILD creates OS/8 on the system device, writes ABSLDR on the system device, and prints:

```
SYS BUILT
.
```

The dot indicates that the OS/8 Keyboard Monitor is activated. BUILD is still in memory at this time and must be written onto the system device. To save the copy of BUILD just used with the current date, type:

```
.DATE mm/dd/yy   (mm=month, dd=day, yy=year)

.SAVE SYS BUILD
```

This copy of BUILD reflects the current configuration of the system. It can be loaded and rerun with the command:

```
.RUN SYS BUILD
```

9. The OS/8 system programs must now be loaded from their respective cassettes. To load these programs, it is first necessary to load MCPIP (Magnetic Tape/Cassette Peripheral Interchange Program). Type the following commands to load MCPIP.

```
.GET SYS BUILD
.START 17400
.SAVE SYS MCPIP;12000=6400
```

Loading System Programs From Cassette

After creating an OS/8 system from cassettes, the user must transfer the system programs from cassette to the system device. This transfer operation is performed with MCPIP which the user has saved on the system device.

NOTE

Users with OS/8 software supplied on DECtape (LINCtape) already have core images of the system programs on the system device. This section concerns only users with software supplied on cassettes.

Each cassette supplied with OS/8 contains several OS/8 system programs. To transfer the programs to the system device, the user mounts the appropriate cassette on a cassette drive and types MCPIP commands as shown below. Use the following procedures to load the system programs.

1. Mount the system cassette DEC-S8-OSYSB-A-TC2 on cassette drive 0.
2. Mount the system cassette DEC-S8-OSYSB-A-TC3 on cassette drive 1.

3. Type the following to call MCPIP from the system device:

```
.R MCPIP
```

MCPIP responds with an asterisk, indicating that it is ready to receive a command line of I/O specifications.

4. Respond as follows to the asterisks printed by MCPIP:

```
*SYS:CCL.SV<CSA0:CCL.SV
*SYS:DIRECT.SV<CSA0:DIRECT.SV
*SYS:FOTP.SV<CSA0:FOTP.SV
*SYS:PIP.SV<CSA0:PIP.SV
*SYS:LIBB.SV<CSA0:LIBB.SV
*SYS:EDIT.SV<CSA0:EDIT.SV
*SYS:PAL8.SV<CSA0:PAL8.SV
*SYS:CREF.SV<CSA0:CREF.SV
*SYS:BITMAP.SV<CSA0:BITMAP.SV
*SYS:BOOT.SV<CSA0:BOOT.SV
*SYS:CAMP.SV<CSA0:CAMP.SV
*SYS:RK8FMT.SV<CSA0:RK8FMT.SV
*SYS:RKEFMT.SV<CSA0:RKEFMT.SV
*SYS:FORT.SV<CSA1:FORT.SV
*SYS:SABR.SV<CSA1:SABR.SV
*SYS:LOADER.SV<CSA1:LOADER.SV
*SYS:SRCCOM.SV<CSA1:SRCCOM.SV
*SYS:EPIC.SV<CSA1:EPIC.SV
*SYS:PIP10.SV<CSA1:PIP10.SV
*SYS:RESORC.SV<CSA1:RESORC.SV
*SYS:DTCOPY.SV<CSA1:DTCOPY.SV
*SYS:TDCOPY.SV<CSA1:TDCOPY.SV
*SYS:TDFRMT.SV<CSA1:TDFRMT.SV
*SYS:DTFRMT.SV<CSA1:DTFRMT.SV
```

5. The source file of CCL should be written onto the system device if the user desires to add his own CCL commands. To write this file on the system device, mount the system cassette DEC-S8-OSYSB-A-TC6 on cassette drive 0. Respond as follows to the asterisk printed by MCPIP,

```
*SYS:CCL.PA<CSA0:CCL.PA
```

This completes the building of the OS/8 system. If the OS/8 extension cassette is available, see the appropriate chapters for loading instructions. Additional device handlers may be loaded and made active using BUILD. See the BUILD section of Chapter 2 for this procedure.

Building OS/8 From Paper Tapes

An OS/8 system can be initially constructed on a mass storage device from the paper tapes supplied with each OS/8 kit. The paper tapes can be loaded from a low-speed reader on a Teletype or from a high-speed reader. This initial construction is only necessary when the software is not supplied on DECtape or cassettes.

The system library program BUILD is used to construct an OS/8 system from paper tapes in the following manner.

1. Load the RIM and Binary loaders into field 0 (refer to Appendix B for instructions on loading programs manually and on paper tape).
2. Using the Binary Loader, load the BUILD binary tape (DEC-S8-OSYSB-A-PB1) into memory.
3. After the entire BUILD binary tape has been loaded with no checksum errors (i.e., AC=0), set the switch register to 200 (octal), i.e., set switch 4 in the up position, set all other switches in the down position. Press the ADDR LOAD and CONT switches. BUILD prints:

   $

   (At this point, all the usual editing features of BUILD are available; see Table 2-10 in the BUILD section of Chapter 2.) Respond with the system (mass storage) device on which OS/8 is to be built. This response must be in the following form:[2]

   $SYS dev=n

   where "dev" represents one of the legal replies taken from Table 1-6. The "=n" is optional and need only be used to indicate the number of physical disk platters which are present if the system device is an RF08 or DF32 disk.
   The "n" must be a digit in the range 1 to 4. If no value for n is specified, a value of 1 is assumed. If a response other than a digit is entered, the message:

   ?SYNTAX

[2] Characters printed by the system are underlined to eliminate confusion with characters typed by the user.

is printed and the SYS command line must be typed again. If n is specified as a digit but is too large for the device specified, the SYS command must be retyped. For example:

```
$SYS RF08=5
?PLAT
$SYS RF08=4
```

Table 1-6 System Devices

Device	Maximum
DF32 (DF32 disk)	4
RF08 (RF08 disk)	4
RK8 (RK8 disk)	1
RK8E (RK8E disk)	1

4. When a correct SYS command line has been entered, e.g.,

```
$SYS RK8E
```

BUILD prints another $. At this time, insert the desired devices for the initial system. The devices listed below must be inserted for a minimum system with paper tape. Type the following commands, followed by carriage returns, to insert a low-speed paper tape configuration.

```
$IN KS33:PTP,PTR    (low-speed paper tape punch/reader)
$IN KL8E:TTY        (terminal keyboard)
```

Type the following commands, followed by carriage returns, to insert a high-speed paper tape configuration.

```
$IN PT8E:PTP,PTR    (high-speed paper tape punch/reader)
$IN KL8E:TTY        (terminal keyboard)
```

5. At this time, the user must specify the device that is to be the default mass storage device by entering the DSK command. For example:

```
$DSK=SYS
```

Any device other than SYS (or carriage return) specified in the DSK command must be the permanent name of a mass storage device which appeared in one of the INSERT commands.

6. When all desired devices have been entered with IN commands, type the following in response to BUILD's $.

```
$BUILD
```

BUILD responds by printing:

```
LOAD OS/8:
```

At this point, load the OS/8 Keyboard Monitor tape (DEC-S8-OSYSB-A-PB4) in the proper reader and respond PTR followed by a carriage return, i.e.,

```
LOAD OS/8: PTR
```

BUILD loads and writes the various parts of the OS/8 Keyboard Monitor onto the system device. If a SYS ERR message occurs at any time during the load, ensure that the system device is write-enabled and press the CONT switch on the PDP-8/E console to retry. If the retry is unsuccessful, return to step 2.

NOTE

When building from the low-speed reader (KS33), remember to turn off the reader when it reaches the leader/trailer at the end of the paper tape.

7. When the Keyboard Monitor has been successfully written onto the system device, BUILD prints:

```
LOAD CD:
```

Place the Command Decoder binary tape (DEC-S8-OSYSB-A-PB5) in the proper paper tape reader and respond PTR followed by a carriage return, i.e.,

```
LOAD CD: PTR
```

BUILD loads and writes the Command Decoder.

8. When BUILD responds with another $, type the following:

```
$BOOT
```

to initiate the final system creation process. BUILD creates OS/8 on the system device, writes ABSLDR on the system device, and prints:

```
SYS BUILT
.
```

The dot indicates that the OS/8 Keyboard Monitor is activated.

9. At this time, BUILD is still in memory and it is necessary to copy it onto the system device. To save the copy of BUILD with the current date, type:

```
.DATE mm/dd/yy      (mm=month, dd=day, yy=year)
.SAVE SYS BUILD
```

This copy of BUILD reflects the current configuration of the system. It can be loaded and rerun with the command:

```
.RUN SYS BUILD
```

See the BUILD section of Chapter 2 for details of using BUILD effectively.

ABSLDR (which resides on the system device) must now be used to load the various system programs. Refer to the following section for instructions.

Loading System Programs From Paper Tape

After an OS/8 system has been created from paper tapes using BUILD, the system programs must be loaded using ABSLDR. When loaded, the system programs are written onto the system device with the SAVE command.

NOTE

Users with OS/8 software supplied on DECtape (LINCtape) or cassettes need not be concerned with this section. The information in this section is only for users with software supplied on paper tape.

Use the following procedures to load the various system programs. The binary tape identification number is indicated in parentheses after the program name. When the Command Decoder prints an uparrow (↑), high-speed reader only, type any character on the keyboard to cause the tape to be read into memory.

In response to the dot (.) printed by the Keyboard Monitor, type:

```
R ABSLDR        (followed by the RETURN key)
```

ABSLDR prints an asterisk when it is ready to receive a command line. Enter the command as specified for each program, ending the command with an ALTMODE. ALTMODE echoes a $. When the Keyboard Monitor responds with a dot, enter the SAVE command. When the Keyboard Monitor responds with another dot, the system program has been written onto the system device and ABSLDR may be called again.

FORTRAN II (DEC-S8-OSYSB-A-PB6)

Place the FORTRAN II Compiler binary tape in the reader, and type the following responses to the . and * printed by the Keyboard Monitor and ABSLDR, respectively.

```
.R ABSLDR
*PTR:(SP)$
.SAVE SYS FORT
```

SABR (DEC-S8-OSYSB-A-PB7)

Place the SABR Assembler binary tape in the reader, and type the following responses to load and save SABR.

```
.R ABSLDR
*PTR:(SP)$
.SAVE SYS SABR
```

LOADER (DEC-S8-OSYSB-A-PB8)

Place the Linking Loader binary tape in the reader, and type the following responses to load and save LOADER.

```
.R ABSLDR
*PTR:/9$
.SAVE SYS LOADER
```

LIBSET (DEC-S8-OSYSB-A-PB9)

Place the Library Setup binary tape in the reader. Type the following:

```
.R ABSLDR
*PTR:12600$
.SAVE SYS LIBSET
```

LIB8 (DEC-S8-OSYSB-A-PR)

Place the LIB8 relocatable binary tape in the reader and type the following:

```
.R LIBSET
*/S$
```

The tape is read and a LIB8.RL file is created on the system device.

CREF (DEC-S8-OSYSB-A-PB10)

Place the CREF binary tape in the reader, and type the following responses to load and save CREF.

```
.R ABSLDR
*PTR:/9$
.SAVE SYS CREF
```

EDIT (DEC-S8-OSYSB-A-PB11)

Place the Editor binary tape in the reader, and type the following responses to load the tape and save EDIT on the system device.

```
.R ABSLDR
*PTR:/9$
.SAVE SYS EDIT
```

PAL8 (DEC-S8-OSYSB-A-PB12)

Place the PAL8 Assembler binary tape in the reader, and type the following responses to load and save PAL8.

```
.R ABSLDR
*PTR:/9$
.SAVE SYS PAL8
```

PIP (DEC-S8-OSYSB-A-PB13)

Place the PIP binary tape in the reader, and type the following responses to load and save PIP.

```
.R ABSLDR
*PTR:13000(89P)$
.SAVE SYS PIP
```

MCPIP (DEC-S8-OSYSB-A-PB14)

Place the MCPIP binary tape in the reader, and type the following responses to load and save MCPIP.

```
.R ABSLDR
*PTR:12000(89P)$
.SAVE SYS MCPIP
```

BITMAP (DEC-S8-OSYSB-A-PB15)

Place the BITMAP binary tape in the reader, and type the following responses to load and save BITMAP.

```
.R ABSLDR
*PTR:12000/9$
.SAVE SYS BITMAP
```

EPIC (DEC-S8-OSYSB-A-PB16)

Place the EPIC binary tape in the reader, and type the following responses to load and save EPIC.

```
.R ABSLDR
*PTR:$
.SAVE SYS EPIC
```

SRCCOM (DEC-S8-OSYSB-A-PB17)

Place the Source Compare binary tape in the reader, and type the following responses to load and save SRCCOM.

```
.R ABSLDR
*PTR:$
.SAVE SYS SRCCOM
```

CCL (DEC-S8-OSYSB-A-PB18)

Place the Concise Command Language binary tape in the reader, and type the following responses to load and save CCL.

```
.R ABSLDR
*PTR:12001(89)$
.SAVE SYS CCL
```

FOTP (DEC-S8-OSYSB-A-PB19)

Place the File Oriented Transfer Program binary tape in the reader, and type the following responses to load and save FOTP.

```
.R ABSLDR
*PTR:14600(89P)4
.SAVE SYS FOTP
```

RESORC (DEC-S8-OSYSB-A-PB20)

Place the Resources binary tape in the reader, and type the following responses to load and save RESORC.

```
.R ABSLDR
*PTR:12000(89)$
.SAVE SYS RESORC
```

DIRECT (DEC-S8-OSYSB-A-PB21)

Place the DIRECT binary tape in the reader, and type the following responses to load and save DIRECT.

```
.R ABSLDR
*PTR:14600(89P)$
.SAVE SYS DIRECT
```

PIP10 (DEC-S8-OSYSB-A-PB22)

Place the PIP10 binary tape in the reader, and type the following responses to load and save PIP10.

```
.R ABSLDR
*PTR:$
.SAVE SYS PIP10
```

CAMP (DEC-S8-OSYSB-A-PB23)

Place the Cassette and Magnetic Tape Positioner program binary tape in the reader, and type the following responses to load and save CAMP.

```
.R ABSLDR
*PTR:$
.SAVE SYS CAMP
```

BOOT (DEC-S8-OSYSB-A-PB24)

Place the BOOT binary tape in the reader, and type the following responses to load and save BOOT.

```
.R ABSLDR
*PTR:$
.SAVE SYS BOOT
```

This completes the building of the OS/8 system. If the OS/8 extension kit paper tapes are available, see the appropriate chapters for loading instructions. Additional device handlers may be loaded and made active using BUILD. See the BUILD section in Chapter 2 for this procedure.

Disk as the System Device

If disk is to be the OS/8 system device, an OS/8 system must be built onto the disk from the distribution media, i.e., cassettes, paper tape, or DECtape (LINCtape). The disks available as system devices are RF08, DF32, RK8, and RK8E. Refer to the appropriate part of this section for the cassette or paper tape building procedure. For DECtape or LINCtape distribution, refer to the BUILD section of Chapter 2.

Once an OS/8 system has been built on a disk, it may occasionally be necessary to start (bootstrap) the system into operation when nothing is in memory. For example, whenever an RK8E disk cartridge is placed into its slot and is to be used, the system should be bootstrapped. Also, if a program error is encountered such that the contents of locations 7600-7777 in either field 0 or field 1 are in doubt, the system should be bootstrapped. The following sections detail the specific bootstrap used for each type of disk.

RF08 AND DF32 DISKS

If the OS/8 system device is an RF08 or DF32 disk, use the bootstrap shown in Table 1-7.

Table 1-7 RF08/DF32 Disk Bootstrap

STEP #	OCTAL VALUES	SWITCH REGISTER SETTING				AND THEN
		012	345	678	91011	
1	0000	000	000	000	000	press EXTD ADDR LOAD
2	7750	111	111	101	000	press ADDR LOAD
3	7600	111	110	000	000	lift DEP key
4	6603	110	110	000	011	lift DEP key
5	6622	110	110	010	010	lift DEP key
6	5352	101	011	101	010	lift DEP key
7	5752	101	111	101	010	lift DEP key
8	7750	111	111	101	000	press ADDR LOAD and press CLEAR and press CONT

When the bootstrap has been loaded, the OS/8 Keyboard Monitor should respond with a dot (.). If it does not, repeat the bootstrap procedure. If an error persists, consult the local Digital sales office.

RK8E DISK

If only one RK8E disk unit is present on the OS/8 system, use the bootstrap shown in Table 1-8.

NOTE

If a PDP-12 computer is being used, execute an I/O PRESET in 8 mode before performing step 5 of the bootstrap in Table 1-8.

Table 1-8 Single RK8E Disk Bootstrap

STEP #	OCTAL VALUES	SWITCH REGISTER SETTING				AND THEN
		012	345	678	91011	
1	0000	000	000	000	000	press EXTD ADDR LOAD
2	0030	000	000	011	000	press ADDR LOAD
3	6743	110	111	100	011	lift DEP key
4	5031	101	000	011	001	lift DEP key
5	0030	000	000	011	000	press ADDR LOAD and press CLEAR and press CONT

If more than one RK8E disk unit is present on the system, the user may choose which unit (0-3) he wishes to be the system device. To specify the correct RK8E unit as the system device, load the OS/8 disk cartridge in the desired unit and enter the bootstrap shown in Table 1-9.

Table 1-9 Multiple RK8E Disk Bootstrap

STEP #	OCTAL VALUES	SWITCH REGISTER SETTING				AND THEN
		012	345	678	91011	
1	0000	000	000	000	000	press EXTD ADDR LOAD
2	0025	000	000	010	101	press ADDR LOAD
3	7604	111	110	000	100	lift DEP key
4	6746	110	111	100	110	lift DEP key
5	6743	110	111	100	011	lift DEP key
6	7604	111	110	000	100	lift DEP key
7	5031	101	000	011	001	lift DEP key
8	0025	000	000	010	101	press ADDR LOAD

Enter the desired unit number (0-3) in switch register settings 9 and 10 as follows:

unit 0 all switches down
unit 1 switch 10 up; all others down
unit 2 switch 9 up; all others down
unit 3 switches 9 and 10 up; all others down

Press CLEAR and CONT.

When either of the bootstraps has been loaded, the OS/8 Keyboard Monitor should respond with a dot (.). If it does not, repeat the bootstrap procedure. If an error persists, consult the local Digital sales office.

RK8 DISK

If the user has only one RK8 disk unit on his OS/8 system, the bootstrap in Table 1-10 is used to start OS/8.

Table 1-10 Single RK8 Disk Bootstrap

STEP #	OCTAL VALUES	SWITCH REGISTER SETTING				AND THEN
		012	345	678	91011	
1	0000	000	000	000	000	press EXTD ADDR LOAD
2	0030	000	000	011	000	press ADDR LOAD
3	6733	110	111	011	011	lift DEP key
4	5031	101	000	011	001	lift DEP key
5	0030	000	000	011	000	press ADDR LOAD and press CLEAR and press CONT

NOTE

If a PDP-12 computer is being used, execute an I/O PRESET in 8 mode before performing step 5 of the above bootstrap.

If more than one RK8 disk unit is present on the system, the user may choose which unit (0-3) he wishes to be the system device. To specify the correct RK8 unit as the system device, load the OS/8 disk cartridge in the desired unit and enter the bootstrap shown in Table 1-11.

Table 1-11 Multiple RK8 Disk Bootstrap

STEP #	OCTAL VALUES	SWITCH REGISTER SETTING				AND THEN
		012	345	678	91011	
1	0000	000	000	000	000	press EXTD ADDR LOAD
2	0026	000	000	010	110	press ADDR LOAD
3	7604	111	110	000	100	lift DEP key
4	6732	110	111	011	010	lift DEP key
5	6733	110	111	011	011	lift DEP key
6	5031	101	000	011	001	lift DEP key
7	0026	000	000	010	110	press ADDR LOAD

Enter the desired unit number (0-3) in the switch register settings 9 and 10 as follows:

unit 0 all switches down
unit 1 switch 10 up; all others down
unit 2 switch 9 up; all others down
unit 3 switches 9 and 10 up; all others down

Press CLEAR and CONT.

When either of the above bootstraps has been loaded, the OS/8 Keyboard Monitor should respond with a dot (.). If it does not, repeat the bootstrap procedure. If an error persists, consult the local Digital sales office.

Restarting OS/8

If the OS/8 system ever ceases apparent response to the user, the computer can be restarted by loading a restart address of either 7600 or 7605. To load a restart address, set the console switches to 7600 or 7605, press the HALT, ADDR LOAD, EXTD ADDR, CLEAR, and CONT switches. A period should be printed on the terminal. If there is no response, OS/8 is no longer in memory and must be bootstrapped in.

Starting at location 7600 causes the contents of locations 0-1777 to be saved on the system device. These locations are then available when the Keyboard Monitor resumes operation. Starting at 7605 does not save the core locations, but does save time on a DECtape configuration.

KEYBOARD MONITOR

The Keyboard Monitor provides communication between the user and the OS/8 executive routines by accepting commands from the terminal Keyboard. The Keyboard Monitor allows the user to create logical names for devices, run system and user programs, save programs and to call ODT.

System Conventions

The OS/8 system has various conventions which are quickly mastered by even the novice programmer. Naming procedures for devices and file extensions have been designed as simple mnemonics. OS/8 makes use of the terms: "word", "page", "record", and "block" as units of storage. In directory listings and elsewhere file lengths are referenced in terms of blocks (or records). The terms are defined as follows:

$$1 \text{ block} = 1 \text{ record} = 2 \text{ pages} = 256_{10} \text{ words}$$

Each word is composed of 12 bits. The internal structure of the PDP-8 words and pages is described in detail in Chapter 2 of ***Introduction to Programming***

PERMANENT DEVICE NAMES

Each device in the OS/8 system is referenced by means of a standard permanent device name. These names are used in all I/O designations and are listed in Table 1-12.

These names are the device names assigned when the OS/8 system is configured. They may be changed by reconfiguring the system; however, caution should be observed when doing so. Certain system programs operate on the premise that a specific device name will be present in the system; for instance, PIP makes use of the device name TTY: as the default device when doing directory listings, CREF assumes LPT: as the default output device, and the Command Decoder uses device DSK: as the general default output device. Therefore, it is suggested that the following device names remain present on the system:

SYS:
DSK:
TTY:
LPT:

Table 1-12 Permanent Device Names

Permanent Name	I/O Device
SYS	System device (disk if the system has a large disk —RK8 or RF08; otherwise DTA0).
DTAn	DECtape n, where n is an integer in the range 0 to 7, inclusive.
LTAn	When using BUILD, LINCtapes may be called LTA rather than DTA. n is an integer in the range 0 to 7 inclusive.
DSK	The default storage device for all files. The assignment of DSK is specified at system generation time. Usually DSK is the disk on a single disk system or DTA0 on a DECtape system.
TTY	Terminal keyboard and printer.
PTP	Paper tape punch.
PTR	Paper tape reader (before accepting input, the system prints an up-arrow (↑), to which the user replies by typing any key).
CDR	Card Reader
LPT	Line printer (performs a form feed before it begins printing output from a new program).
CSAn	Cassette drive n, where n is an integer in the range 0 to 7, inclusive.
MTAn	Magnetic tape drive n, where n is an integer in the range 0 to 7 inclusive.
DF	DF32 disk.
RF	RF08 disk.
RKAn	RK01 or RK05 disk unit n, where n is an integer in the range 0 to 3.
TV	VR12 scope (PDP-12 only).
BAT	Pseudo device which reads from BATCH input stream (see BATCH section in Chapter 2).

FILE NAMES AND EXTENSIONS

Files are referenced symbolically by a name of up to six alphanumeric characters followed, optionally, by a period and an extension of two alphanumeric characters. The extension to a file name is generally used as an aid for remembering the format of a file. Some commonly used extensions are given in Appendix G. Some programs (e.g., FOTP) also accept the characters * and ? in file names. These characters have special meanings to the programs involved.

In most cases the user will want to conform to the standard file name extensions established for OS/8. If an extension is not specified for an output file, some system programs append assumed extensions. Where an extension for an input file is not specified by the user, the system does a search for that file name with the default extension. Failing to find such a file, a search is then done for the original file without an extension. For example, if PROG were specified as an input file to PAL8, the Command Decoder would first look for the file PROG.PA (since .PA is the standard extension for PAL8 input files). If PROG.PA were not found, the Command Decoder would try to find the file PROG (with no extension). As not all system programs utilize default extensions, reference the following table and the individual system programs for details:

Table 1-13 Assumed Extensions

Extension	Meaning
.SV	Core image file or SAVE file; appended to a file name by the R, RUN, SAVE, and GET Keyboard Monitor commands.
.FT	8K FORTRAN source file.
.SB	8K SABR source file.
.PA	PAL8 source file.
.BN	Absolute binary file (default extension for ABSI.DR, BUILD, and BITMAP input files. Also used as the default extension for PAL8 binary output file).

Table 1-13 Assumed Extensions (Cont.)

Extension	Meaning
.RL	Relocatable binary file (default extension for a Linking Loader input file. Also used as the default extension for an 8K SABR output file).
.MP	File containing a loading map (used by the Linking Loader). Also used as default extension for BITMAP output files.
.LS	Assembly listing output file (default extension for PAL8 and SABR).
.TM	Temporary file generated by FORTRAN or SABR for system use (default extension for CREF input files and PAL8 output files).

For example, if the user types:

```
.RUN DSK PROG
```

the file PROG.SV (on device DSK) is run, if found. If the user types:

```
.RUN DSK PROG.A
```

then PROG.A (on device DSK) is run, if found.

Using the Keyboard Monitor

Each command to the Keyboard Monitor is typed at the terminal keyboard. If corrections are necessary, they must be made before entering the command line to the system. A command line is entered to the system by typing either the RETURN key, which causes a carriage return/line feed operation but no printed character, or an ALTMODE (ESCAPE on some Teletype Keyboards), which prints a $, but causes no carriage return/line feed. Correcting mistakes is accomplished by typing the RUBOUT key, which deletes the last character typed and causes a backslash (\) character to be printed followed by the character

which was deleted. Successive RUBOUTS each cause one more character to be printed and deleted. The first non-RUBOUT character typed (after the last RUBOUT in a sequence) causes a closing backslash (\) to be printed, thus enclosing the deleted characters with backslashes. For example:

User types: .RUN DSK (RUBOUT) (RUBOUT) (RUBOUT) DTA1:FILE
Teleprinter
Shows: .RUN DSK\KSD\DTA1:FILE

Keyboard
Monitor sees: .RUN DTA1:FILE

If at any time an input line becomes so corrected that it is no longer intelligible to the user, he can verify the contents of the line by typing the LINE FEED key. This causes the entire input line to be echoed as the Keyboard Monitor would see it at that point. The line is not considered to be entered to the system, and the user can proceed to edit, delete, or enter the line at his discretion.
For example:

User types: .RUN DTA3\3\2:PRG \G\OG (LINE FEED key typed)
System echoes: RUN DTA2:PROG

A command line may be deleted completely before it is entered by typing a CTRL/U (produced by pressing the CTRL key and U key simultaneously). This echoes as a ↑U, and returns control to the Keyboard Monitor without accepting the current input line. Typing a CTRL/U causes a dot (.) to be printed at the left margin and the Keyboard Monitor is ready to accept commands.

Control can be returned to the Keyboard Monitor while under any of the system library programs by typing a CTRL/C (produced by pressing the CTRL and C keys simultaneously). This echoes as a ↑C and the Keyboard Monitor signals that it is ready to accept input by printing a dot (.) at the left margin of the terminal screen or paper.

KEYBOARD MONITOR COMMANDS

The user has a choice of nine commands which he may type in response to the dot (.) printed by the Keyboard Monitor. These are: ASSIGN, DEASSIGN, GET, SAVE, ODT, RUN, R, START, and DATE. Commands may be abbreviated by typing only the first two characters. Execution occurs after typing the RETURN or ALT MODE key.

Any errors the user may make while utilizing these commands result in an error message being printed by the Keyboard Monitor. After occurrence of an error, control returns to the Keyboard Monitor and the command must be retyped. The error messages and their explanations are listed in Table 1-14, following the descriptions of the commands.

In addition to the Keyboard Monitor commands discussed in this section, certain extended commands and features are available to the user through the Concise Command Language (CCL). CCL simplifies the entry of certain commands and performs operations which could not be performed otherwise. See the CCL section in this chapter for further information.

ASSIGN Command

The ASSIGN command is of the form:

```
.ASSIGN dev udev
  or
.AS dev udev
```

This command causes a new, user-defined device name (udev) to be considered equivalent to the permanent device name (dev). Only one user name can be associated with a single device at a time. For example:

```
.AS DTA1 IN
```

causes all future references to IN to refer to DECtape unit 1, (references can still be made to the device DTA1 also).

If a user-defined device name is not indicated, any existing

user-defined name is removed and only the permanent device name is valid. For example:

```
.AS DTA1 IN
.AS DTA1
```

The above sequence changes the name of DECtape 1 to IN and then back to simply DTA1 again.

The user-defined name is composed of up to four alphanumeric characters; the user-defined name takes precedence over the permanent name. Device-independent programs are easily possible since a change in the user name of a device by means of the ASSIGN command can change the operation of a routine without changing the code.

Although user-defined names may be four characters long, the name may not be unique in the OS/8 system. (This is due to the fact that the device name is internally coded in only one word.) A three or four character name may be tested for uniqueness by typing an ASSIGN command as follows:

.AS name

If a 'name NOT AVAILABLE' message results, the name is unique within the current system, is not in the system tables, and therefore may be used.

All user-defined device names of one or two characters in length are unique.

DEASSIGN Command

The DEASSIGN command is of the form:

.DEASSIGN

or

.DE

and causes all permanent device names to be restored, discarding all previous user-defined device names. For example:

```
.AS DTA1 IN
.DE
```

causes DECtape 1 to be assigned the name IN. The DEASSIGN command removes the name IN from the system tables; DTA1 can no longer be referenced as IN.

GET Command

The GET command is of the form:

.GET dev file.ex

or

.GE dev file.ex

The GET command loads *core image* files (.SV format, not ASCII or binary) into core from a device. This device (dev) is specified along with the file name (file) and an optional file name extension (.ex). The file is loaded into core with its core control block; the core control block is then moved to a special area on the system device, where it is maintained on the system device and contains information about the file such as its starting address and areas of core occupied by the file. Also contained is a Job Status Word, which is saved (with the SAVE command) and loaded in location 7746 of field 0 with the file to indicate what parts of core the file uses and how, as follows:

Job Status Word

Bit Condition	Meaning
Bit 0 = 1	File does not load into locations 0-1777 in field 0, (0000-1777).
Bit 1 = 1	File does not load into locations 0-1777 in field 1, (10000-11777).
Bit 2 = 1	Program must be reloaded before it can be restarted because it modifies itself during execution.
Bit 3 = 1	Program being run will not destroy the BATCH monitor.
Bits 4 – 9	Unused, and reserved for future expansion.
Bit 10 = 1	Locations 0-1777 in field 0 need not be saved when calling the Command Decoder overlays.
Bit 11 = 1	Locations 0-1777 in field 1 need not be saved when calling the USR.

A core control block is created for each core-image file when the file is created by the Linking Loader, ABSLDR, or the SAVE command.

If a file name extension is not specified to the GET command, the extension .SV (for core-image file) is added automatically to the file name. For example:

```
.GE DTA3 OH
```

attempts to fetch the file OH.SV from device DTA3.

The GET command is typically used before a debugging session with ODT. GET is used to load the object program into core, then ODT is called, and the program can be altered and/or debugged (see the section on ODT for more details).

SAVE Command

The SAVE command is of the form:

```
.SAVE dev file.ex a-b,c,...;s=n
or
.SA dev file.ex a-b,c,...;s=n
```

where:

a-b,c,...	are the addresses of the areas and locations in core to be saved. (In this case, locations a through b, location c, and any other specified locations.) a, b, and c are five digit locations. (The first digit represents the field.) When a single location is indicated (c) the entire page on which c is located is saved.
;s	is the starting address of the file.
=n	n is a four digit octal number representing the contents of the Job Status Word (see the GET command).

The program currently in core is saved on the device (dev) specified, with the file name indicated (file.ex). If an extension is not specified, the extension .SV is automatically added by the system. If the remaining arguments are not given, the required information is taken from the current core control block (refer to the GET command).

There are some restrictions on the SAVE arguments which should be noted:

1. Each set of limits (a-b) must be in the same field and not cross field boundaries. For example:

```
.SAVE SYS FOO 0200-20200
```

   is illegal since the limits transcend a field boundary.

2. No two sets of limits can overlap; (i.e. a-b, c-d must not overlap). In fact, once a location on a specific page is included in the limits, any other location on that core page, whether overlapping or not, will produce an error message. For example:

   `.SAVE SYS FOO 0-177,200-377` is legal, but

   `.SAVE SYS FOO 0-200,201-377` is illegal.

3. In SAVEs involving memory fields other than field 0, the field must be specified before each of the two core limits. If the field is unspecified, field 0 is assumed. Thus:

   `.SAVE SYS FOO 20200-0377` is illegal, while

   `.SAVE SYS FOO 20200-20377` is legal.

4. SAVE files can include 7600 in any field. However, *extreme* care must be taken when manipulating these areas, particularly in fields 0 and 1, as the system resident code could be destroyed by GETting area 07600-07777. It is suggested that SAVEs involving 7600 be limited to fields above field 2.

5. If the first location of a page is not a multiple of 400, that page cannot be saved without the previous page. Thus the following commands are equivalent:

```
.SAVE DSK PROG 2634
.SAVE DSK PROG 2400-2777
```

If an error message is printed in response to a SAVE command, the program currently in core has not yet been saved. The core image, however, is still intact.

Examples of SAVE commands are:

```
.SAVE DSK CPROG 55,10500-10577;10502
```

This statement saves the program in core on the disk as a file named CPROG.SV. The areas of core saved are locations 0 to 177 in field 0 and locations 400 to 577 of field 1 (when a single core location or part of a page is indicated, the entire page on which the locations occur is saved). The starting address of the program is 502 in field 1. The core control block is updated to contain this information and the old Job Status Word is taken intact from the original core control block.

```
.SAVE DSK CPROG
```

The above statement causes the program in core to be saved on device DSK under the name CPROG.SV where the areas of core to be saved are taken from the core control block currently available.

ODT Command

The ODT command is of the form:

.ODT

or

.OD

This command causes the system ODT to be loaded into core and started. ODT is a system overlay, and as such takes up none of the user's program area unless the breakpoint feature is used, in which case ODT uses locations 4, 5, and 6 of every field in which a breakpoint had been placed. When using ODT to debug programs, the user-defined device names cannot be used; each I/O device must be called by its permanent device name.

ODT is described in greater detail later in this chapter.

RUN Command

The RUN command is of the form:

.RUN dev file.ex

or

.RU dev file.ex

The RUN command, like the SAVE command, handles *only* core-image files. The file indicated (file.ex) on the device specified (dev) is loaded into core and its core control block is moved to the system scratch area. The program is started at its starting address. The RUN command is equivalent to a GET and a START command.

If an extension to the file name is not specified, the extension .SV is automatically added to the file name. For example:

```
.RU DTA1 PROG
```

causes the file PROG.SV on DECtape 1 to be loaded and started.

R Command

The R Command is of the form:

.R file.ex

and is similar to

.RUN SYS file.ex

This command handles only core image files from the system device. The file is loaded and started. If the file name extension is not specified, the extension .SV is automatically added.

The R command differs from the RUN command in that a core control block is *not* written to the system device. In order to save a program which does not have its core control block in the usual location on the system device, all the optional arguments of the SAVE command must be explicitly stated. System programs are most often called using the R command, since they need not be resaved.

To call a program which is to be later updated and saved, use of the RUN or GET commands is suggested.

START Command

The START command is of the form:

.START nnnnn

or

.ST nnnnn

The program currently in core is started at location nnnnn. If the argument nnnnn is omitted, the program is started at the starting address specified in the core control block.
For example:

```
.ST 10555
```

starts the program in core at location 555 in field 1.

```
.ST
```

starts the program at the starting address given in the core control block.

The START command clears certain areas of core—the device handler in core table and the Command Decoder output area.

DATE Command

The DATE command is of the form:

.DATE mm/dd/yy

or

.DA mm/dd/yy

The DATE command sets up the date in the system for purposes of dating directory entries and listings, printing on program output, etc. For example:

```
.DA 3/13/74
```

indicates that the date is March 13, 1974.

Keyboard Monitor Error Messages

Table 1-14 lists the generalized and command Keyboard Monitor errors. All errors return control to the Keyboard Monitor and the command must be retyped. xxxx indicates the core location where the error was detected.

Table 1-14 Keyboard Monitor Error Messages

Message	Meaning
BAD ARGS	The arguments to the SAVE command are not consistent and violate restrictions listed in 1, 2, 3 under SAVE command.
BAD CORE IMAGE	The file requested was not a core-image file (it could have been an ASCII or binary file).
BAD DATE	The date has not been entered correctly (using slashes), or incorrect arguments were used, or the date was out of range.
ILLEGAL ARG.	The SAVE command was not expressed correctly; illegal syntax used.
MONITOR ERROR 2 AT xxxx (DIRECTORY I/O ERROR)	Attempt made to output to a WRITE-LOCKed device, usually DECtape; or an error has occurred reading/writing a directory.
MONITOR ERROR 5 AT xxxx (I/O ERROR ON SYS)	An error occurred while doing I/O to the system device. This error is normally the result of not WRITE-ENABLing the system device.
MONITOR ERROR 6 AT xxxx (DIRECTORY OVERFLOW)	This message results if a directory overflow has occurred (no room for tentative file entry in directory).
name NOT AVAILABLE	The device with the name given is not listed in any system table, or it is not available for use at the moment (check the device in question), or the user tried to obtain input from an output-only device (such as the high-speed paper tape punch).
name NOT FOUND	The file with the name given was not found on the device indicated, or the user tried to input from an output-only device.
NO!!	The user attempted to start (with .ST) a program which cannot be started. The user must not restart any user program or system library

Table 1-14 Keyboard Monitor Error Message (Cont.)

Message	Meaning
	program which modified itself while in core (bit 2 of the Job Status Word is set; see the GET command for details).
NO CCL!	The command was not a legal keyboard monitor command. It was, however, a valid CCL command, but the file CCL.SV was not found or an I/O error occurred while trying to read the file.
SAVE ERROR	An I/O error has occurred while saving the program. The program remains intact in core.
SYSTEM ERR	An error occurred while doing I/O to the system device. The system should be restarted at 7600 or 7605. Do *not* press CONTinue, as this is sure to cause further errors.
TOO FEW ARGS	An important argument has been omitted from a command. For example, `.RUN DSK` would generate this message, as the program to be run has not been entered in the command.
USER ERROR 0 AT xxxx	An input error was detected while loading the program. xxxx refers to the Monitor location where the error was generated.

abcd?	Where abcd is not a legal command; for example, if the user typed:
	`.HELLO`
	the system would echo:
	`HELLO?`

COMMAND DECODER

Once a system program has been called via the Keyboard Monitor, that system program may make use of the Command Decoder by permitting the user to enter a list of I/O files and devices. The Command Decoder prints an asterisk (*) at the left margin to indicate it is ready to accept a command string.

The Command Decoder uses the same keyboard characters as the Keyboard Monitor for the purpose of correcting typing mistakes. The RUBOUT key deletes one character per rubout. The CTRL/U (↑U) combination deletes an entire line. CTRL/C returns the user to the Keyboard Monitor, and the LINE FEED key causes the entire line (preceded by an asterisk) to be printed on the terminal as it appears in the TTY input buffer.

The description of files, file names, extensions, devices, and device names is contained in the section concerning the Keyboard Monitor; this description pertains to the Command Decoder as well.

Command Decoder Input String

The expected string for I/O specification takes the form:

DEV:OUTPUT FILES<DEV:INPUT FILES

(While the left angle bracket (<) is the accepted divider character between output and input files, the back arrow (←) may also be used.) There may be 0-3 output files and 0-9 input files, depending on the requirements of the individual system program. The particular I/O string used with each system library program is described in its respective section.
For example:

```
*DTA1:XY1,LPT:<DSK:PROG
```

The PAL8 assembler would use the first output file (DTA1:XY1) for the binary output of the assembly and the second output device (LPT:) for the listing. DSK:PROG or PROG.PA is the input source file.

Multiple file specifications are separated by commas. If no output files are indicated, the left angle bracket can be omitted. For example:

```
*DSK:PROG
```

would cause the file PROG on device DSK to be accepted as an input file.

The forms in which I/O files may be specified in a command string are illustrated below:

File Specifications

Form	Example	Meaning
DEVICE:FILE NAME	`*DTA3:FILE1`	The I/O file is to be found under the specified name (FILE1) on the device indicated (DTA3:)
DEVICE:	`*LPT:`	When a device is indicated without an associated file name, the device is usually a non-directory device. (If a directory device is used, the device can be read, but not written; for example, referencing DTA0 causes the entire DECtape Unit 0 to be used as the input file. DSK: is always the default output device.)
FILENAME	`*NAME<DTA2:PROG`	A file name used without an associated device indicates that the file will be found on an assumed device. For all output files and the first input file, the device is assumed to be DSK:. The example indicates DSK:NAME as an output file. For input files after

Form	Example	Meaning
		the first, the device is assumed to be the device of the previous entry. For example:
	*DSK:PROG1<DTA1:FILE1,FILE2,FILE3	
		causes the three input files to be taken from DTA1.
NULL FILE		The absence of an explicit file specification has different meanings in context, and is indicated by a comma which is not preceded by a file designation. For output files, a null file indicates that there is no output file for this position. If the example given were an input line to PAL8, the first output file (binary) would not be generated, but the listing would be output to the line printer. For input files, a null file indicates that the device of the most recent entry is to be used as a non-directory device:
	*,LPT<DTA1:QUEST,DTA2:STAR	
		*DSK:A<PTR:,,
		This input string allows three paper tapes to be read from the high-speed reader.

EXAMPLES OF COMMAND STRINGS

Some examples of command strings specifying I/O are shown below with appropriate explanations.

Example 1:

```
*DSK:BINARY,LPT:<SOURCE
```

The file named SOURCE is the input file on device DSK: The two output files are BINARY on DSK, and a second file on the line printer (LPT). The PAL8 assembler uses this format; however, the assembler also adds the extension .BN onto the file labeled BINARY. Thus, the output file on device DSK: will be named BINARY.BN.

Example 2:

```
*INPUT1,INPUT2,INPUT3,PTR:,
```

This is a string of input files with no output file. Notice that the left angle bracket is not necessary if there are only input files specified. This type of input might be given to one of the loaders (which do not require output files). Three files are taken from device DSK and then two are taken from the paper tape reader (PTR:,).

Example 3:

```
*DTA2:A,B<XYZ:C,D
```

The input files C and D are taken from device XYZ (which could be any device with the user-defined name XYZ). The output files are a file named A on DTA2 and a file named B on DSK.

Example 4:

```
*,LPT:<SRC
```

The input file is named SRC and is on DSK. The two output files specified are one null file (no output file in that position) and a file to be sent to the line printer (LPT).

Example 5:

```
*PTR:,,DTA1:X
```

As in Example 2, this is another input only file string. The first input file comes from the paper tape reader, as does the second (PTR:,,). The third input file is named X and is on DTA1.
Example 6:

```
*A<TTY:,
```

Both input files in this example come from the Teletype (generally the low-speed reader). The single output file is named A and is stored on DSK.

INPUT/OUTPUT SPECIFICATION OPTIONS

In addition to output and input files which are indicated on the file specification line to the Command Decoder, there are various options which can also be indicated on this line. These options are interpreted by the individual system programs and are covered in detail in the sections describing the various programs. Options are either numbers or alphanumeric option characters.

Numbers used as options are generally contained in the command line with the equal sign (=) or square brackets ([ ]) construction. The alphanumeric option characters are set off from the I/O specifications by the slash (/) character for single character options, and parentheses for a string of single characters. The usage of the slash, parentheses, equal sign, and square brackets is explained below. These explanations will serve as references and format specifications once the user has learned from reading about each individual program which options he will be needing.

The format for input to the Command Decoder looks generally like the following:

DEV:OUPUT FILES<DEV:INPUT FILES/OPTIONS

The Slash Construction

A single alphanumeric option character is preceded by a slash and can occur anywhere in the input line, even in the middle of a name, although the usual position is after the file specification. For example:

```
*TTY:/L<DSK:AB
```

is equivalent to:

```
*TTY:<DSK:AB/L
```

The option specified is L, which PIP interprets as a command to list the DSK directory beginning at file AB.

The Parentheses Construction

Any number of option characters can be grouped together inside parentheses. This construction is also valid anywhere in the input line. For example:

```
*OUT:X<IN:Y(AQZ)
```

is equivalent to:

```
*OUT:X<IN:Y/A/Q/Z
```

The Equal Sign Construction

An octal number up to seven digits long and preceded by an equal sign (=) may optionally be used as an indicator. This construction is often used to set a starting address, but may be assigned other functions as well. It may only occur once in a line and must be followed by a separator character (comma, left angle bracket, back arrow, ALT MODE key or RETURN key) or by other options and a separator character. The following example uses the equal sign construction and indicates three separate options:

```
*FILE1=1002(AQX),FILE2
```

Interpretation of options and = sign numbers varies depending upon the program which called the Command Decoder. See the individual system programs for details.

The Square Bracket Construction

The square bracket construction can only occur immediately after an output file name and consists of an open bracket, a *decimal* number between 1 and 255, and a close bracket. The square

bracket construction is generally only used by the more sophisticated user to optimize file storage.

The open bracket ([) is produced by holding down the SHIFT key while typing a K (i.e., SHIFT/K); the close bracket (]) is produced by typing a SHIFT/M. This construction is used to provide an upper limit on the number of blocks (256 words per block) to be contained in the output file in order to allow the system to optimize file storage. For example:

```
*BINARY[19],LISTING[200]<SOURCE/8
```

The output files are a file named BINARY on device DSK: having a maximum length of 19 blocks, and a file name LISTIN (only six characters are significant) on the device DSK with a maximum length of 200 blocks. The input file is SOURCE on device DSK; the option specified is 8, which is interpreted by the program being run.

Command Decoder Error Messages

The following is a complete list of the error messages which the Command Decoder generates if a command string is improperly input.

Table 1-15 Command Decoder Error Messages

Message	Meaning
ILLEGAL SYNTAX	The command line was formatted incorrectly or contains illegal characters.
name DOES NOT EXIST	The device with the name specified could not be found in the system tables.
name NOT FOUND	The file with the name specified does not exist on the device indicated.
TOO MANY FILES	More than three output files or nine input files were specified. Some programs may restrict the user to fewer files.

CCL (CONCISE COMMAND LANGUAGE)

CCL (Concise Command Language) provides the OS/8 user with an extended set of Keyboard Monitor commands. Some CCL commands allow the user to call a system program indirectly, perform an operation, and return to the Keyboard Monitor. These commands are more concise than the usual calling sequence of a program. For example, instead of typing the following to call PAL8:

```
.R PAL8
*FILE,LPT:<FILE
```

CCL can be used by typing the following command:

```
.PAL FILE-L
```

Other CCL commands perform functions not performed by other OS/8 programs.

The user may write his own CCL commands and add them to CCL. See the *OS/8 Software Support Manual* for instructions on adding CCL commands.

CCL Commands

CCL commands are entered at the terminal in the same manner as Keyboard Monitor commands, in response to the dot printed. Normally, CCL commands are terminated with a carriage return. Depending upon the command being used, control may return to the Monitor when the operation is completed or may remain within another OS/8 program. If the user wishes to remain within control of another program when control would normally return to the Monitor, he can terminate the CCL command with an ALTMODE. This termination procedure is the reverse of the way in which most OS/8 programs operate.

CCL COMMAND FORMAT

The full CCL command keyword need not be typed; each command has letters that are required. The CCL commands are listed below in alphabetic order. Letters that are not required are printed in italics, e.g., the CREATE command is shown as follows:

CREA*TE*
to indicate that only the letters CREA are required.
BAC*KSPACE*
BO*OT*
CCL
COMP*ARE*
COM*PILE*
COP*Y*
COR*E*
CREA*TE*
CREF
DA*TE*
DEA*SSIGN*
DEL*ETE*
DIR*ECT*
ED*IT*
EOF
EXE*CUTE*
H*ELP*
LI*ST*
LO*AD*
MAK*E*
MAP
MUNG
PAL
PRI*NT*
PU*NCH*
REN*AME*
RES
REW*IND*
SKIP
SQ*UISH*
SU*BMIT*
TE*CO*
TY*PE*
UA
UB
UC
UNL*OAD*
VER*SION*
ZERO

In some cases, there are two commands to run a program. For example, the MAKE and TECO commands both run the TECO program, as does the R TECO command.

Most CCL commands are entered by typing the command followed by an argument of the form:

dev:output files<dev:input files/options

This is the I/O specification format used by the Command Decoder. The slash construction and the parentheses construction can be used to include options in the program being run by CCL. If no input device is specified, DSK is assumed. If a file name is specified with no extension but is followed by a dot, no default extensions are tried. The actual format for each command is shown in the command discussion.

CCL remembers arguments used by some CCL commands, e.g., COMPILE, and can use these arguments in other commands. (This procedure is explained for each CCL command to which it applies.) If, however, the DATE command is used to change the current date, the remembered arguments are erased. Commands which require remembered arguments produce a BAD RECOLLECTION message if no previous argument existed.

CCL COMMAND OPTIONS

Some CCL commands assume the inclusion of options that would have to be specified if the OS/8 program were called directly. The DELETE command, for example, runs the FOTP program, including the /L and /D options. Other options may be included in the command line with the slash or parentheses construction.

CCL also has options that may be included in a CCL command line. These options are of the form:

−ex

where "ex" is one of the options specified in Table 1-16.

Table 1-16 CCL Options

Option	Meaning
-L	Send output to LPT.
-LS	Generate a listing file (used with the COMPILE, EXECUTE, and PAL commands). The listing file is written onto SYS if no output device is specified and is given a .LS extension.
-MP	Generate a core map (used with the COMPILE, EXECUTE, and PAL commands).
-NB	Do not create a binary file (used with the COMPILE, EXECUTE, and PAL commands).
-P	Send output to PTP.
-S	Send output to TV.
-T	Send output to TTY.

WILD CARD CONSTRUCTION

Certain CCL commands that run the FOTP or DIRECT programs may use a wild card construction. These commands are COPY, DELETE, DIRECT, LIST, RENAME, and TYPE.

The wild card construction means that the file name or the extension in a CCL command may be replaced totally with an asterisk or partially with a question mark to designate certain file names or extensions. The asterisk is used as a wild field to designate the entire file name or extension. For example:

TEST1.*	All files with the name TEST1 and any extension.
*.BN	All files with a BN extension and any file name.
.	All files.

The question mark is used as a wild character to designate part of the file name or extension. A question mark is used for each character that is to be matched; e.g., PR?? matches all files beginning with PR that are two to four characters long. For example:

TEST2.B?	All files with the name TEST2 and any extension beginning with B.
TES??.PA	All files with a PA extension and any file name from three to five characters long beginning with TES.
??.??	All files with file names of two characters or less.

The asterisk and the question mark can be specified together in the same command line:

???.* All files with file names of three characters or less.

A specification may not contain embedded *'s, e.g., A*B.* is an illegal specification and will produce the following message:

```
ILLEGAL SYNTAX
```

If an * or ? is included in a command other than COPY, DELETE, DIRECT, LIST, RENAME, or TYPE, the following message appears:

```
ILLEGAL * OR ?
```

See the FOTP section of Chapter 2 for more detail about using the wild card construction.

INDIRECT COMMANDS (@ CONSTRUCTION)

When many file names and options are to be included in a single CCL command, they can be put into a file and thus need not be typed each time they are required. This is accomplished by the use of the @ file construction, which may appear anywhere within the argument portion of a CCL command. The @ construction is of the form:

@dev:file.ex

If dev: is omitted, DSK: is assumed. The word file must be a file name. If the extension (ex) is omitted, .CM is assumed.

The information in the specified file is then put into the command string to replace the characters @ file.ex. For example, if the file FLIST.CM contains the string:

FILEB,FILEC/L,FILED

then the CCL command

```
.COMPILE FILEA,FILEB,FILEC/L,FILED,FILEZ
```

could be replaced by the CCL command

```
.COMPILE FILA,@FLIST,FILEZ
```

Carriage returns and line feeds within the file are ignored, but nulls are not; the nulls signify end-of-line. Command files may not exceed one block in length. If a command line is more than 512 characters in length, the following message is printed:

```
COMMAND LINE OVERFLOW
```

NONSTANDARD FILE NAMES (# CONSTRUCTION)

In rare instances, a user may create a file that has a file name unacceptable to OS/8. For example, a file name could contain embedded spaces: ABC D.EF. CCL provides an alternate means of specifying an 8-character file name (6 characters of name and 2 of extension, separated by a dot). The alternate specification is a 16-digit sequence of octal numbers which represent the internal packed 6-bit representation for the file name. These octal digits are preceded by a # to identify them as an alternate file name.

For example, the file name:

ABC D.EF

could be replaced by:

#0102036404000506

The 64 is the octal representation for a space. Note that all 16 digits must be given to use the alternate specification even though the file name does not contain 16 digits. This is done by specifying 00 for any nonexistent characters. If all 16 digits are not present, CCL prints the message:

```
BAD NUMBER
```

See Appendix A for a table listing the 6-bit octal code to be used in the alternate file name construction.

BAC*KSPACE Command*

The BACKSPACE command runs the OS/8 CAMP program and spaces a magnetic tape or cassette backward a specified number of files or records. This CCL command works in exactly the same way as the CAMP BACKSPACE command. When CAMP has completed a backspace operation, control returns to the Keyboard Monitor. See the CAMP section of Chapter 2 for a detailed explanation of the BACKSPACE command.

BO*OT Command*

The BOOT command chains to the OS/8 BOOT program, allowing the user to bootstrap onto another device or onto another PDP-8 system. The BOOT command is of the form:

.BOOT/dv

where dv is a mnemonic listed in Table 2-5 in the BOOT section of Chapter 2. For example, the command:

```
.BOOT/RF
```

bootstraps onto the RF08 disk.

If the user wishes to halt before performing the actual bootstrap, he can type the BOOT command followed by a period, e.g.

```
.BOOT/CA.
```

The computer will halt, allowing the user to mount a new device. Pressing the CONT switch completes the bootstrapping operation. This form of the BOOT command is particularly useful when only a single disk or DECtape drive exists on the system.

CCL *Command*

The CCL command disables the CCL program on the OS/8 Keyboard Monitor residing on the system device. This command has no arguments and is of the form:

```
.CCL
```

When the CCL command is used, the CCL feature of OS/8 is deactivated; OS/8 will not accept CCL commands. If CCL is desired at a later time, it must be reactivated with the R command, i.e.,

```
.R CCL
```

COMP*ARE Command*

The COMPARE command runs SRCCOM and compares two source files line by line and prints all their differences. This command has the form:

.COMP dev:file.ex<dev:file.ex,dev:file.ex

See the SRCCOM section of Chapter 2 for a complete description of SRCCOM.

COMPILE Command

The COMPILE command produces binary files and/or compilation listings for the specified program files. The assembler or compiler used is determined by the source file extension.

The COMPILE command is of the form:

.COM file.ex<file.ex

Table 1-17 lists the extensions and the compiler or assembler to which the COMPILE command will chain.

Table 1-17 Compiler/Assembler Extensions

Extension	Program
.BA	BASIC
.FT	FORT if present on SYS when CCL was enabled; otherwise F4.
.PA	PAL8
.RA	RALF
.SB	SABR

If no extension is explicitly given, each possible extension is tried in order until one is found and the appropriate compiler is invoked. If no output file name is specified, the name of the first input file is assumed.

If a nonstandard OS/8 file extension is used, i.e., other than those listed in Table 1-17, a processor switch of the form:

-ex

can be included in the command line to indicate the compiler/assembler to which COMPILE must chain. This processor switch must be a legal 2-character OS/8 extension. For example, to assemble a file named HANDLR.03 with the PAL8 assembler, the user would type:

```
.COMPILE HANDLR.03-PA
```

Each time the COMPILE, LOAD, PAL, or EXECUTE command is executed, the command with its arguments is remembered in a temporary file. Therefore, the file name used last can be recalled for the next command without specifying the arguments again. If, for example, the EXECUTE command:

```
.EXECUTE TEST1.PA
```

was entered previously, then the COMPILE command to specify TEST1.PA could be:

```
.COMPILE
```

COPY *Command*

The COPY command transfers files from one OS/8 I/O device to another. The command string can contain one output specification and from one to five input specifications. This command runs FOTP and includes the /L option among any other options specified by the user.

The COPY command is of the form:

.COPY dev:file.ex<dev:file.ex

When the user enters a COPY command, the message:

```
FILES COPIED:
```

is printed and each file copied is listed on the terminal.

Examples:

The following command copies all files with .FT extensions from DTA0 to DSK.

```
.COPY DTA0:*.FT
FILES COPIED:
PROG1.FT
DTA3.FT
TEST.FT
```

The following command transfers all files from four to six characters long beginning with FILE and having any extension.

```
.COPY DTA2:<DTA0:FILE??.*
FILES COPIED:
FILE1.PA
FILE2.PA
FILEX.DA
FILEZ.BN
```

To understand the COPY operation, see the FOTP section of Chapter 2.

CORE *Command*

The CORE command can be used in two different ways. One way is the same as the CORE command in BUILD to specify the highest core field available to the OS/8 system. This form of the CORE command is:

.CORE n

where n is an octal number in the range 0 to 7, specifying the number of 4K core banks available to OS/8. The following table indicates the value of n for the available core sizes.

n	core
0	all available core
1	8K
2	12K
3	16K
4	20K
5	24K
6	28K
7	32K

For example, a system which is to use only 20K of a 32K system would have the following CORE command:

```
.CORE 4
```

The other form of the CORE command is the command typed without an argument. When this form is used, the amount of core actually in use by OS/8 is printed on the terminal. For example, if a 32K system has been restricted to 20K by the CCL or BUILD CORE command, the following would appear:

```
.CORE
20K/32K CORE!
```

If all available core is in use on a 32K system, the following would be printed.

```
.CORE
32K CORE!
```

CREA*TE Command*

The CREATE command runs EDIT (the OS/8 Symbolic Editor) and opens a new file for creation. The file specification must consist of a single output file only. For example:

```
.CREA TEST1.FT
```

is the same as

```
.R EDIT
*TEST1.FT<
```

If no argument is given, the argument used in the last CREATE or EDIT command is assumed. See the Symbolic Editor section in this chapter for a detailed explanation of EDIT.

CREF *Command*

The CREF command runs the PAL8 assembler, including the /C option which causes PAL8 to chain to the CREF program. CREF produces a cross-reference listing file. If no listing file is specified, the listing is sent to the line printer. For example, the following command produces a CREF listing of the file DSK: PROG.PA on the line printer.

```
.CREF PROG
```

DA*TE Command*

If an argument is given with the DATE command, it is treated as the standard Keyboard Monitor DATE command. If no argument is given, this command prints the current day and date on the terminal or prints NONE if no date was specified. For example:

```
.DA
THURSDAY JANUARY 31, 1974
```

If the user has created a file named DATE.SV, the DATE command runs that program, allowing the user to implement messages of the day.

DEA*SSIGN Command*

The DEASSIGN command is exactly the same as the Keyboard Monitor DEASSIGN command. If CCL is enabled, CCL performs this function instead of the Monitor.

DEL*ETE Command*

The DELETE command deletes one or more files from disk or DECtape. The command string can contain one output specification and from one to five input specifications. This command runs FOTP and includes the /D and /L options among any other options specified by the user.

The DELETE command has the form:

.DEL dev:file.ex<dev: file.ex/options

When the user enters a DELETE command, the message:

```
FILES DELETED:
```

is printed and each file deleted is listed on the terminal.
Examples:

The following example deletes any DSK file with a .BN extension if a file with the same name and a .PA extension exists on DSK.

```
.DEL *.BN<*.PA
FILES DELETED:
TEST1.BN
TEST2.BN
```

The following example deletes from DTA0 any file of five or less characters that begins with DATA and has any extension.

```
.DEL DTA0:DATA?.*
FILES DELETED:
DATA1.PA
DATA1.LS
DATA3.BN
DATA6.PA
```

To understand the DELETE operation, refer to the FOTP section of Chapter 2.

DIR*ECT Command*

The DIRECT command produces listings of OS/8 device directories. The directories produced can be of several varieties, depending upon the options specified in the DIRECT command line. The standard directory listing consists of the following columns: file name, file name extension, length in blocks written, and creation date. The DIRECT command runs the DIRECT program. See the DIRECT section of Chapter 2 for a complete description of DIRECT and the available options.

In addition to the DIRECT options specified, the CCL options -L, -P, -T, and -S can be used in the DIRECT command line. For example, the command:

```
.DIR DTA1:/C-L
```

is the same as

```
.R DIRECT
*LPT:<DTA1:/C
```

Both these commands will list on the line printer all files with the current date that exist on the DECtape on DTA1.

ED*IT Commands*

The EDIT command runs EDIT (the OS/8 Symbolic Editor) and opens an already existing file for editing. For example, the CCL command:

```
.EDIT DATA1
```

is the same as:

```
.R EDIT
*DATA1<DATA1
```

For a detailed explanation of EDIT, refer to the Symbolic Editor section in this chapter.

If no argument is given, CCL assumes the argument used in the last CREATE or EDIT command. If the EDIT command is used with the < option, e.g.,

```
.EDIT TEST3<TEST2
```

CCL remembers the argument up to but not including the <. Thus the next EDIT command with no argument will edit the file TEST3.

EOF *Command*

The EOF command runs the CAMP program and writes a single mark (file gap) on the specified magnetic tape or cassette. The EOF command has the form:

.EOF dev:

where "dev" may be either MTAn or CSAn, signifying the device on which the file mark is to be written. For example:

```
.EOF MTA3:
```

writes an end-of-file mark on the magnetic tape mounted on MTA3.

The CCL EOF command operates in the same way as the CAMP EOF command. See the CAMP section of Chapter 2 for a detailed explanation of the CAMP commands.

EXE*CUTE Command*

The EXECUTE command produces binary files and/or compilation listings for the specified program files, loads the binary file, and executes the program. The EXECUTE command has the form:

.EXE file.ex,file.ex

The assembler or compiler used is determined by the source file extension. In addition to the extensions listed in Table 1-17, the EXECUTE command includes the following:

Extension	Program
.BN	ABSLDR
.RL	LOADER or LOAD

If no file is specified, a search is made for a file with one of the above extensions. The first such file found is executed.

The EXECUTE command, like the COMPILE command, will accept processor switches in the -ex form to control the compiler or assembler used.

Each time the EXECUTE, LOAD, PAL, or COMPILE command is executed, the command with its arguments is remembered

in a temporary file. If no argument is specified in a EXECUTE command, CCL remembers the argument of the last COMPILE, PAL, or LOAD command. For example, if the COMPILE command:

```
.COMPILE FILE1.PA
```

was previously executed, then the EXECUTE command to specify FILE1.PA could be:

```
.EXECUTE
```

HELP Command

The HELP command prints useful information on specified OS/8 programs. Each OS/8 program has a HELP file (.HL extension). It is these HELP files that are printed when a HELP command is issued.

If OS/8 software was supplied on DECtape (or LINCtape), the HELP files are present on System Tape #2 and can be run by mounting that tape and specifying the unit in the HELP command. For example, to print the HELP file for FOTP from the DECtape mounted on DTA1, type:

```
.HELP DTA1:FOTP
```

If the OS/8 software is supplied on paper tape, the HELP files are on DEC-S8-OSYSB-A-PA. The HELP file tape is composed of separate segments with a short length of leader/trailer code between them. The files are listed below in the order that they appear on the tape. These file segments must be separated and labeled before they can be used.

DIRECT.HL
BATCH.HL
SABR.HL
PIP.HL
FOTP.HL
ABSLDR.HL
PIP10.HL
BOOT.HL
LOADER.HL

BITMAP.HL
EDIT.HL
CREF.HL
BUILD.HL
PAL8.HL
ODT.HL
SRCCOM.HL
CCL.HL
TECO.HL
FORT.HL
LOAD.HL
LIBRA.HL
EPIC.HL

The HELP files must be transferred onto the system device. This is done by using PIP to load the paper tape and save the file on the system device. For example, to load and save CCL.HL, the user puts the appropriate tape in the reader and types the following:

```
.R PIP
*CCL.HL<PTR:$      $=ALTMODE
```

If the OS/8 software is supplied on cassettes, the HELP files must be transferred onto DSK with MCPIP. OS/8 HELP files are supplied on the system cassette DEC-S8-OSYSB-A-TC4. To load and save a HELP file, mount this cassette on cassette unit 0, drive 0. For example to load CAMP.HL from the cassette mounted on CSA0, type:

```
.R MCPIP
*DSK:CAMP.HL<CSA0:CAMP.HL
```

Once the desired HELP file is present on DSK, the HELP command can be issued by specifying the desired file name. If no extension is specified, the .HL extension is assumed. If no file name is specified, CCL.HL is assumed. The HELP file is printed on the terminal if no output device is specified.

LIST Command

The LIST command lists the contents of the specified file on the specified device. This command runs FOTP and includes the /U option. The LIST command has the form:

.LI dev:file.ex<dev:file.ex

If no output device is specified, LPT is assumed.

LO*AD Command*

The LOAD command runs one of the OS/8 loaders, depending on the extension of the first specified input file. The LOAD command is of the form:

.LO file.ex

A .BN extension runs ABSLDR. A .RL extension runs LOADER (or LOAD). If no extension is given, a search is made for a file with one of these extensions. The /G option may be specified to start execution of the program after it is loaded. If no argument is given, CCL remembers the argument of the last COMPILE, PAL, or EXECUTE command.

MAK*E Command*

The MAKE command runs TECO and opens the specified file for output. The MAKE command has the form:

.MA dev:file.ex

If no device is specified, DSK is assumed. If no file extension is specified, .PA is assumed. If the file specified already exists, CCL prints the message:

```
%SUPERCEDING
```

Example:

The CCL command:

```
.MA DTA1:TEXT.TX
```

is the same as the following:

```
.R TECO
*EWDTA1:TEXT.TX$$
```

To use the MAKE command, the user must be familiar with TECO as explained in Chapter 2.

MAP *Command*

The MAP command runs BITMAP and produces a core map of the specified file. The MAP command is of the form:

.MAP dev:file.ex

If no output device is specified, TTY is assumed. If no extension is specified, .BN is assumed.

Example:

The CCL command:

```
.MAP TEXT,DATA
```

is the same as

```
.R BITMAP
*TTY:<TEXT.BN,DATA.BN
```

See the BITMAP section of Chapter 2 for a complete explanation of the BITMAP program.

MUNG *Command*

The MUNG command allows the user to operate on source files and text using a predefined TECO macro. This command has the form:

.MUNG dev:file.ex,text

The MUNG command runs TECO which reads the first page of the specified file into Q-register Y. The contents of this file are assumed to be a TECO macro. If no extension is specified, .TE is assumed. If a dot is typed after the file name, no extension is assigned.

After the page is read in, all text between the comma and the end of the line is entered into the TECO text buffer. This text is presumed to be an argument to the macro. If no text is desired, no comma is necessary. With the text pointer at the end of the buffer, the macro in Q-register Y is executed. In the following example, the text will specify source files to be edited by the TECO macro.

If the text argument is too long, CCL prints the error message:

```
COMMAND TOO LONG
```

Example:

This example assumes that the user wishes to remove the line feeds from several files that contain carriage return and line feed characters at the end of each line. This operation is desirable for certain data files.

To perform this operation, the user has created a file called MACRO.TE. This file contains the following.

```
  HX!                    Argument to Q-register 1
! HKGY!                  Move macro into text buff
! J2SFILE$-4DG1!         Enter argument into macro
! J2SSTART!$Ø,.K!        Remove preamble
! JI  !!KXY$!            Insert command to kill Y
! HX1M1!                 Move text buff into Q-register 1
                         and execute as macro

!!START!
  EBFILE$!               Open file

 * * * Any user TECO code may be substituted here * * *

! <N
   $-D>!                 Search for and remove line feeds
! EX!                    Exit back to Monitor
```

To remove the line feeds from a series of files, the user specifies the name of the above file (MACRO.TE) and the name of the file from which the line feeds are to be removed. For example:

```
.MUNG MACRO,FILE1.DA
.MUNG MACRO,FILE2.DA
.MUNG MACRO,FILE10.DA
```

PAL *Command*

This command runs PAL8 and assembles the source file specified as the argument of the PAL command. The PAL command has the form:

.PAL file.ex

If no extension is given, a search is made for a file with a .PA extension. If no argument is given, CCL remembers the argument of the last COMPILE, EXECUTE, or LOAD command.
See Chapter 3 for a detailed explanation of PAL8.

PRI*NT Command*

The PRINT command runs a program named LPTSPL, if the user has such a program on his OS/8 system. This can be a user-written program or a program obtained from DECUS.

PU*NCH Command*

The PUNCH command runs PIP and punches the file specified on paper tape. This command has the form:

.PU dev:file.ex<dev:file.ex

If no output is specified, PTP is assumed.

REN*AME Command*

The RENAME command renames one or more files on disk or DECtape. The command can contain one output specification and one input specification. RENAME changes the name of the file from the input file name to the output file name. This command runs FOTP and includes the /R option. The form of the RENAME command is:

.REN dev:file.ex<dev:file.ex

When the user enters a RENAME command, the message:

```
FILES RENAMED:
```

is printed and each file renamed is listed on the terminal.
Examples:

```
.REN DTA0:FILE0.TX<DTA0:FILE1.TX
FILES RENAMED:
FILE1.TX

.REN NEWONE.BN<OLDONE.BN
FILES RENAMED:
OLDONE.BN
```

RES *Command*

The RES command runs the RESORC program and lists the device handlers present on an OS/8 system. This command has the form:

.RES dev:file.ex<dev:file.ex/options

Any option allowed on a RESORC command line is allowed with the RES command. See the RESORC section of Chapter 2 for a detailed explanation.

REW*IND Command*

The REWIND command runs the CAMP program and issues a rewind command to a specified OS/8 device controller. This command operates in the same way as the CAMP REWIND command. See the CAMP section of Chapter 2 for a complete description of the REWIND command.

SKIP *Command*

The SKIP command runs the CAMP program and advances over the number of files or records specified on a magnetic tape. See the CAMP section of Chapter 2 for a complete description of the SKIP command.

SQ*UISH Command*

The SQUISH command runs PIP, including the PIP /S option. This command has the form:

.SQ dev:<dev:

If no output device is specified, the output device is assumed to be the same as the input device. The following example:

```
.SQUISH SYS:
```

is the same as the PIP command:

```
*SYS:<SYS:/SS
```

SU*BMIT Command*

The SUBMIT command runs the BATCH program. This command is of the form:

.SUBMIT dev:file.ex<dev:file.ex

where the output dev and file.ex are the optional spooling output file and the input dev and file.ex are the BATCH input file. If no device is specified, DSK is assumed. If no input extension is specified, .BI is assumed. See the BATCH section of Chapter 2 for a complete description of the BATCH program.

TECO Command

The TECO command runs the TECO program which then opens the specified input file for reading and creates an output file. The TECO command may have one output file and at least one input file as arguments. If no argument is specified, the argument used in the last TECO or MAKE command is assumed. If no output file is specified, TECO does an edit backup on the specified file. If no file extension is specified, .PA is assumed.

Examples:

The CCL command:

```
.TECO FILE.BA
```

is equivalent to

```
.R TECO
*EBFILE.BA$Y$$
```

and the CCL command:

```
.TECO WIN2.PA<LTA2:WIN.PA
```

is equivalent to

```
.R TECO
WIN2.PA$ERLTA2:WIN.PA$Y$$
```

The first page of the input file is read into the text buffer before control is returned to the user.

If the TECO command is used with the < option, e.g.,

```
.TECO FILE1<FILE2
```

CCL remembers the argument up to but not including the <. Thus the next TECO command with no argument will edit the file FILE1.

TYPE Command

The TYPE command runs the FOTP program, including the /U option, and prints the specified file. The form of this command is:

.TY dev:file.ex<dev:file.ex

If no output device is specified, TTY is assumed. Thus the CCL command:

```
.TY DTA0:TEST1.DA
```

is the same as

```
.R FOTP
*TTY:<DTA0:TEST1.DA/L/U
```

UA, UB, UC *Commands*

The UA, UB, and UC commands are used to remember and recall arguments. When one of these commands is typed with an argument, CCL remembers the argument in a temporary file. This argument must be a legal CCL command. For example:

```
.UA COPY DSK:<DTA0:*.FT
```

If the UA command is then typed without an argument, the last UA argument is recalled and executed as a CCL command.

UNL*OAD Command*

The UNLOAD command runs the CAMP program and issues a rewind and turn off line command to the specified magnetic tape controller. This command may also be used to rewind a DECtape or to write-lock an RK8E disk. See the CAMP section of Chapter 2 for a complete description of the UNLOAD command.

VER*SION Command*

The VERSION command prints the version numbers of both the OS/8 Keyboard Monitor and CCL. This command has no arguments and is of the form:

```
.VER
```

ZERO *Command*

The ZERO command runs PIP, including the /Z option, and zeroes the device specified. Only file structured devices can be

specified in a ZERO command. The CCL command:

```
.ZERO DTA7:
```

is equivalent to

```
.R PIP
*DTA7:/Z<$
```

CCL Error Messages

The following error messages may appear in response to a CCL command.

Table 1-18 CCL Error Messages

Message	Meaning
BAD DEVICE	The device specified in a CCL command is not of the correct form, (e.g., DTA0.PA:).
BAD EXTENSION	Either an extension was specified without a file name (e.g., DTA1:.PA) or two extensions were specified (e.g., DTA1: FILE.PA.BN).
BAD MONITOR	The version of the Keyboard Monitor being used is not compatible with CCL. A newer version of the monitor must be obtained from Digital before CCL can be used.
BAD NUMBER	A CCL command which uses the # construction does not have the full 16-digit specification that is required.
BAD RECOLLECTION	An attempt was made to use a previously remembered argument when no argument was saved. This error occurs when no argument was previously saved or when the DATE command has been used since the argument was saved.
BAD SWITCH OPTION	The character used with a slash (/) to indicate an option is not a legal option.

Table 1-18 CCL Error Messages (Cont.)

Message	Meaning
CANNOT CHANGE CORE CAPACITY WHILE RUNNING BATCH	A CORE command was issued while the BATCH program was running.
% CAN'T REMEMBER	The argument specified in a CCL command line is too long to be remembered or an I/O error occurred.
CCL 3X OVERLAY & MONITOR INCOMPATIBLE	The version of CCL being used is not compatible with the Keyboard Monitor present on the system. Type R CCL to retry.
COMMAND LINE OVERFLOW	The command line specified with the @ construction is more than 512 characters in length.
COMMAND TOO LONG	The length of a text argument in a MUNG command is too long.
CONTRADICTORY SWITCHES	Either two CCL processor switches were specified in the same command line (e.g., FILE-PA-FT) or the file extension and the processor switch do not agree (e.g., FILE.FT-BA).
name DOES NOT EXIST	The device with the name given is not present on the OS/8 system.
ERROR IN COMMAND	A command not entered directly from the console terminal is not a legal CCL command. This error occurs when the argument of a UA, UB, or UC command was not a legal command.
ILLEGAL * OR ?	An * or ? was used in a CCL command that does not accept the wild card construction. Only CCL commands that run FOTP or DIRECT allow the wild card construction.
ILLEGAL SYNTAX	The CCL command line was formatted incorrectly.
INPUT ERROR READING INDIRECT FILE	CCL cannot read the file specified with the @ construction.

Table 1-18 CCL Error Messages (Cont.)

Message	Meaning
I/O ERROR ON SYS:	An error occurred while doing I/O to the system device. The system must be restarted at 7600 or 7605 (see Restarting OS/8 in the Getting On Line With OS/8 section of this chapter). Do not press CONT, as that will surely cause further errors.
I/O ERROR TRYING TO RECALL	An I/O error occurred while CCL was trying to remember an argument.
NO CCL!	CCL.SV is not present on the system device. Refer to the Getting On Line section of this chapter for instructions on loading programs onto the system device.
NOT ENOUGH CORE	The number specified in a CORE command is larger than the number of 4K core banks on the system.
name NOT FOUND	The file with the name given is not present on the specified device, or the user tried to input from an output-only device.
%SUPERCEDED	The file specified in a MAKE command already exists. This is a warning message indicating that the file is being replaced.
SWITCH NOT ALLOWED HERE	Either a CCL option was specified on the left side of the < or was used when not allowed. For example: COMPARE FILE-NB.
TOO MANY FILES	Too many files were included in a CCL command.

SYMBOLIC EDITOR

The Symbolic Editor is used to create and modify ASCII source files so that these files may be used as input to other system programs (such as FORTRAN, SABR, and PAL8).

The Editor considers a file to be divided into logical units called *pages*. A page of text is generally 50-60 lines long, and corresponds approximately to a physical page of a program listing. (Note that this is *not* the same as a core memory page.) The Editor reads one page of text at a time from the input file into its internal buffer where the page becomes available for editing. The Editor contains commands for creating, modifying, or deleting characters, lines, or complete logical pages of text.

Calling and Using the Editor

To call the Editor from the system device, type:

```
R EDIT
```

in response to the dot (.) printed by the Keyboard Monitor. The system prints an asterisk (*) at the left margin, and in answer to the asterisk, the user types the device designation and the output file name, a left angle bracket, and the input device and file designation(s). For example:

```
*DSK:ABC<PTR:,DSK:AA1
```

causes input from the paper tape reader and from a file named AA1 on DSK. The output file is named ABC and is stored on DSK.

Once I/O file designations are entered, the Symbolic Editor is ready to accept commands from the keyboard and signifies its readiness by printing a number sign (#) at the left margin. This symbol occurs whenever the Editor is waiting for a command.

Any device which operates in ASCII mode and has a device handler in the system is available for use by the Editor. For example, the high and low-speed reader/punch, DECtape, disk, card reader and line printer are each legal devices. The Editor only operates properly on ASCII files, however. No error message is given if non-ASCII files are input to the Editor, but the results of operations are garbled.

As many as nine and as few as zero input files are permitted. If the number of input files is zero, (that is, a new file is to be created using the terminal keyboard) the Editor allows input from the keyboard via the Append command.[3] The Editor uses a keyboard input routine which is independent of the OS/8 terminal handler, thus it is not necessary to specify TTY: as an input device if text is to be created. (It is, in fact, recommended that TTY: *not* be used as an input device, as input buffering may cause a loss of characters on input.) Commands which attempt to read from any other device (when no file name is specified) are disabled, and a question mark (?) appears when a Read command is attempted.

The Editor allows only one output file. If no output file is specified, the only output operations which may be performed are L (list buffer on TTY:) or V (list buffer on LP08 line printer).

EDITOR OPTIONS

The following three options are the valid I/O specification options for the Editor. (The format for I/O specification options has been previously described in the section detailing the Command Decoder. After reading these options, the reader is advised to turn to that section to review the various formats.)

Table 1-19 Editor Options

Option	Meaning
/A	Return control to the Editor after the file is closed (calls Command Decoder for new files). If /A is not used, control returns to the Keyboard Monitor.
/B	Convert two or more spaces to a TAB when reading from input device.
/D	Delete the old copy of the output file (if one exists) before opening the new output file on the device. If /D is not used, the old copy of the output file is not deleted until all data has been transferred to the new file by an E or Q command.

[3] See Example Using the Editor for an illustration of using the Editor to create a program.

For example, the I/O specification line:

```
*DTA2:FILE<DTA1:ARG/D
```

deletes FILE on DTA2 (if such a file exists) before creating a new FILE on DTA2.

Special Key Commands to the Editor

The Editor can be considered as operating in two different modes. During *command mode*, the Editor prints a # at the left margin indicating that it is waiting for a command from the keyboard. *Text mode* is the condition of the Editor when it is processing various editing and I/O commands (such as Insert and Append).

The following commands allow the user to transfer between modes. (These commands are produced by pressing the CTRL key and the appropriate character key simultaneously.)

Table 1-20 Editor Key Commands

Command	Mode in Which Used	Meaning
CTRL/C	Text and command mode	Returns control to the Keyboard Monitor. The text buffer is retained and the Editor remains accessible to the user with the START command. In text mode, text between the last carriage return and the ↑C is lost. The START command can be used to restart the Editor as follows: ↑C .START * START recalls the Command Decoder to accept new I/O

Table 1-20 Editor Key Commands (Cont.)

Command	Mode in Which Used	Meaning
		file designations. When the START command is given, and the previous output file is not closed, that output file and the contents of the output buffer are deleted.
CTRL/O	Text Mode	Stops the listing of text. Returns control to Command Mode.
CTRL/FORM	Text Mode	Returns the Editor to Command Mode.
CTRL/U	Text Mode	Typing CTRL/U while entering text from the keyboard causes text in the current line to be ignored. A carriage return/line feed is generated and the line may be retyped. (The command is equivalent to typing rubouts back to the beginning of the line.)

Other special Editor characters used to represent numbers or perform erasures are listed in Table 1-21.

Table 1-21 Special Characters

Character	Example	Meaning
	.+1C .-7L .L	The dot (.) character is used as the current line counter character. The dot can be used alone, with + or — an integer, or any place where a number can be used.
/	/-7L /-5L	The slash character is similar in use to the dot and represents the highest numbered line in the text buffer.
#	#	Typing the # in response to Editor's prompting # prints the current version number of the Editor.

Table 1-21 Special Characters (Cont.)

Character	Example	Meaning
RUBOUT Key		Typing the RUBOUT key in text mode deletes one character from the text buffer and causes a back-slash to be printed. The erasure is done right to left up to the last CR/LF. Typing the RUBOUT-key in command mode causes the entire command line to be deleted.

Editor Text Buffer

In text mode, the Editor performs I/O operations on text stored within the text buffer. Text is input to the Editor buffer until a form feed is encountered on input. A line of text is terminated by a carriage return. If no carriage return is present, the text entered on that current line is ignored. The buffer has room for approximately 5600 (decimal) characters. When text has been input to the extent that there are only 256 decimal locations available in the buffer, the TTY rings a warning bell. From this point on, whenever a carriage return is detected during text input, control returns to the Editor command mode and the TTY bell is rung. This line-at-a-time input may continue until the absolute end of buffer is encountered. At this point, no more text will be accommodated in the buffer; a '?' is printed and control returns to command mode.

TEXT COLLECTION

The OS/8 version of the Editor contains an automatic *text collector* which reclaims buffer space following the use of a D (delete), S (Search), or C (Change) command. Formerly, deleted text was not physically removed from the buffer; now this text is removed by the text collector, and the necessary pointers updated. If a full buffer condition is reached, the user may output lines of text (using the Punch command, for example), and then delete these lines from the buffer—text collection is automatic and always occurs on the three commands mentioned above.

NOTE

If extremely large amounts of text are deleted, the text collection process could take several seconds. For small amounts of text, no appreciable time is lost.

Search Mode

There are two types of searches available in the Editor. The first is the standard character search, and the second is the character string search which allows the user to search for a combination of characters. Each is explained in turn.

SINGLE CHARACTER SEARCH

The single character search is of the form:

#S
or #nS
or #m,nS

where m and n represent line numbers ($m \leq n$), and S initiates the search command.[4] This search command searches the entire text buffer or the line(s) indicated for the search character. The search character is typed by the user after he types the RETURN key which enters the command, and does not echo on the terminal The Editor prints the contents of the entire buffer or the indicated line(s) until the search character is found. When the search character is found, printing stops and the user may type one of the following:

Option	Result
text	Enter text at that point at which the search character was found and printing stopped.
CTRL/G (TTY bell rings)	Change the search character to the next character typed; search continues. If the character is not contained in the line, the remainder of the line will be printed and control will be returned to command mode.

[4]A command summary is included in Table 1-23 at the end of this section.

Option	Result
CTRL/FORM	Continue searching for the next occurrence of the character.
RETURN key	End line here, deleting all subsequent text on that line.
LINE FEED key	Make two lines out of the current line by inserting a carriage return at this point.
RUBOUT key	Delete characters from this line. Each rubout echoes a backslash (\) for each character deleted. When all characters have been deleted, echoing of '\' stops.

CHARACTER STRING SEARCH

The character string search can identify a given line in the buffer by the contents of that line or any unique combination of characters. This search returns the line number as a parameter that can be used to further edit the text. There are two types of string search available: intra-buffer search and inter-buffer search.

Intra-Buffer Character String Search

The intra-buffer search scans all text in the current buffer for a specified character string. If the string is not found, a ? is printed and control returns to command mode. If the string is found, the number of the line which contains the string is put into the current line counter and control waits for the user to issue a command.

Thus, searching for a character string in this manner furnishes a line number which can then be used in conjunction with other Editor commands. This provides a useful framework for editing, as it eliminates the need to count lines or search for line numbers by listing lines.

An intra-buffer search is signalled by typing the ALT MODE key (which echoes as $) in response to the Editor's #. The user then types the string to be found (up to 20 characters long—any additional characters typed are echoed but not included in the search). The search string cannot be broken across line boundaries.

Typing a single quote (') terminates the character string and causes the search to be performed beginning at line 1 of the text buffer. Use of the double quote (") causes the search to begin at the current line +1. (Use of ' and " as command elements prohibits their use in the search string.)

For example, assume the text buffer contains the following text:

```
ABC DEF GJO
1A2B3C4D5E6
.STRINGABCD
.
.
```

The user wants to list the line that contains ABC; he types:

```
#$ABC'L
```

The search begins with line 1 and continues until the string is found. The current line counter is set equal to the line in which the string ABC occurred, and the L command causes the line to be printed as follows:

```
ABC DEF GJO
```

Control returns to command mode, awaiting further commands. If the user wanted to find the next reference to ABC, he could type:

```
.
#"L
```

In this case, " is a command which causes the last string searched for to be used again, with the search beginning at the current line +1. It is not necessary to enter the search string again. The command may be used several times in succession. For example, if the user wanted to find the fourth occurrence of a string containing the characters FEWMET he could type:

```
#$FEWMET'""""L
```

This command will list the line which contains the fourth occurrence of that string. The L (List) command (or any other command code) can be given following either ' or ". The L command causes the line to be listed when and if it is found.

To clear the text string buffer, the user can type:

```
#$'
```

The system responds with a question mark and the text string buffer is cleared.

The properties of the commands ' and " allow for easy and useful editing, as the following example illustrates. In order to change the CIF 20 to CIF 10, the user can give the following commands:

```
#$DUM,'$CIF 20"C
CIF 10      /NEW FIELD          (CTRL/FORM)
```

The above set of instructions first causes the Editor to start at line 1 and search for the line beginning with DUM,. A search is then made for CIF 20, starting from the line after the line containing DUM,. When this string is found, the line number of the line containing the string CIF 20 becomes the current line number. The C (Change) command is given, and the user then changes the line to the correct instruction.

Since this search feature produces a line number as a result, any operations which can be done by explicitly specifying a line number can be done by specifying a string instead. For example:

```
#$STRING'+4L
```

will list the fourth line after the first occurrence of the text STRING in the text buffer.

```
#$LABEL1,',$LABEL2,"L
```

will list all lines between the two labels, inclusive.

```
#$PFLUG'S
```

will do a character search on the line which contains PFLUG. (The user types the search character after typing the RETURN key that enters the line.)

In cases where both strings and explicit numbers are used, strings should be used first. For example, the following commands:

```
#1+$BAD!'L
```

will not list the next line after the string BAD! occurs. The correct syntax is:

```
#$BAD!'+1L
```

Inter-Buffer Character String Search

The inter-buffer search scans the current text buffer for a character string. If the string is not found, the current buffer is written to the output file, the buffer is cleared, and the next buffer is read from the input device. The search then resumes at line 1 of the new buffer. This process continues until either the string is found or no more input is left. If input is exhausted, control returns to command mode with all the text having been written to the output file. If the string is found, control returns to command mode with the current line equal to the number of the line containing the first occurrence of the string. For example, a command to find the character string GONZO may appear as follows:

```
#J
$GONZO'
#.=0024
```

The J command initiates an inter-buffer search; the $ is printed automatically by the Editor, and the user types in the character string to be sought. The search proceeds, and when the string is found, control returns to command mode. The user types the .= construction to discover the number of the line in the current buffer on which the string is contained. To find further occurrences of the string GONZO, the user can use the F command. The F command uses the last character string entered to search the buffer starting from the current line count + 1.

```
#F
#.=0106
```

The above example causes a search for the string GONZO starting at the current line + 1. If no output file is specified to the J or F commands, the Editor reads the next input buffer without attempting to produce any output. This provides an easy way of paging through text for a particular string.

After the J or F commands have processed the entire input file, it is necessary to execute either an E or Q command to close the output file. If this is not done, the file will be deleted by the Monitor.

The following two commands may be used to abort the string search command, once given:

Command	Explanation
CTRL/U	A CTRL/U will return control to the Editor command mode if executed while entering text in a string search command; the string search command is ignored, as in the following example:

```
#J
$WORD↑U
#
```

The inter-buffer search for the characters WORD was aborted by the user typing ↑U before terminating the string with ' or ".

RUBOUT	Executing the RUBOUT key while entering text for a string search causes the text so far entered to be ignored and allows a new string to be inserted. Editor answers the command by printing $, as in the following example:

```
#$CHAR
$
```

An example of the use of the character string search is contained in the OS/8 Demonstration Run in Appendix D.

Editor Error Messages

Errors made by the user while running the Editor may be of two types. Minor errors (such as an Editor command string error, an attempt to execute a read or write command without assigning a device, or a search for a nonexistent string) will cause a question mark to be printed at the left margin. The command may be retyped. Major errors cause control to return to the Keyboard Monitor and may be due to one of the causes listed in Table 1-22. These errors cause a message to be printed in the form:

?n↑C

where n is an error code and ↑C indicates that control has passed to the Keyboard Monitor.

Table 1-22 Editor Error Codes

Error Code	Meaning
0	Editor failed in reading a device. Error occurred in device handler; most likely a hardware malfunction.
1	Editor failed in writing onto a device; generally a hardware malfunction.
2	File close error occurred. For some reason the output file could not be closed; the file does not exist on that device.
3	File open error occurred. This error occurs if the output device is a read-only device or if no output file name is specified on a file-oriented output device.
4	Device handler error occurred. The Editor could not load the device handler for the specified device. This error should never occur.

During the editing of a file, the output device specified in the command string may become full before the editing process is complete. If this is the case and a write is attempted on that device, an error occurs. The output file is closed, the message:

```
FULL
*
```

is printed; control returns to the Command Decoder for a new set of I/O specifications. The user must indicate a new output file which will contain the text that would not fit on the output device, and any further editing the user wishes to do. Since the contents of the text buffer are retained through this procedure, no text will be lost if this error occurs.

NOTE

If no output file is specified when control returns to the Command Decoder, the Editor returns to the Command Decoder again; this continues until an output device is specified. However, specifying an improper output device (such as PTR:) will cause a fatal error and the output buffer will be destroyed.

Assuming the output device is valid, the Editor will continue the operation which filled the old file, putting all output into the new output file. After editing is completed, the output files should be combined with PIP. The entire process may appear as follows:

```
.R EDIT
*OUT<IN
#Y
#J
$STRING'
FULL
*DTA3:OUT2<
#.L
          TAD STRING
#.D
#E
FULL
*DTA4:OUT3<
```

Device DSK: is full; DTA3: is specified as the new output device, and editing continues.

Device DTA3: has become full; DTA4: is now specified as the output device, and editing continues.

At this point the output "file" is the series of files—DSK:OUT, DTA3:OUT2, and DTA4:OUT3. When output is split like this, the split may have occurred in the middle of a line. Therefore, the output files should never be edited separately as the split lines will then be lost. In a case such as this, the files should be combined with PIP as follows:

```
.R PIP
*DTA2:OUT<DSK:OUT,DTA3:OUT2,DTA4:OUT3
```

The new file, OUT, may then be edited.

Example Using the Editor

The following example illustrates both the use of the Editor to create a new file, and a few of the commands available for editing. Sections of the printout are coded by letter—corresponding explanations follow:

A The user calls EDIT; the output file will be called FILE and will be stored on the default device. There is no input file since one will be created from the terminal keyboard. The Append command is used to insert text into the empty buffer.

B Text is inserted.

C The user makes a mistake and uses the RUBOUT key to correct it.

D More text is added.

E The user notices a typing mistake he has made several lines back in the text. He types a CTRL/FORM to finish the Append command, searches for the illegal character, corrects it, and then lists the line.

F The P command writes the current buffer into the output file placing a form feed after the last line. The K command deletes all text in the current buffer, in preparation for a new page of text.

G The user inserts new text using the Append command. When he is finished, he types a CTRL/FORM to end the command.

H The E command closes out the file. Control is returned to the Keyboard Monitor.

```
A   .R EDIT
    *FILE<

    #A

B   /PTP, PTR HANDLER FOR THOSE
    /WITHOUT HIGH SPEED I/O

            IFZERO NOHSPT+LIST <XLIST>
            IFNZRO NOHSPT  <
    PTP,    Ø
C           CLA CLL CMX\L    /SET LINK
            JMS PSETUP
    PTPLP,  KSF
D           JMP PTPCNT       /KEYBOARD FLAG OFF
            KRT
            AND PTP177
            TAD PTPM3

    #.-2S
E           KRT\S

    #.L
            KRS

F   #P

    #K

    #A
G           SZA CLA
            JMP PTPCNT
            .
            .
            .

H   #E
    .
```

Summary of Editor Commands

The commands discussed in Table 1-23 can each be given whenever the Editor prints a # at the left margin. These commands are of the general form:

#X
#n X

or

#m,nX

where m and n represent the line number designation, ($m \leq n$) and X represents the command letter. The command is entered to the Editor with the RETURN key. Numbers used in Editor commands are decimal numbers.

Table 1-23 Symbolic Editor Commands

Command	Format	Meaning
A	#A	Append the following text being typed at the keyboard until a form feed (ASCII 214 or CTRL/FORM) is found. The form feed returns control to command mode. Text input following the A command is appended to whatever is present in the text buffer.
B	#B	List the number of available core locations in the text buffer. The Editor returns the number of locations on the next line. To estimate the number of characters that can be accommodated in this area, multiply the number of free locations by 1.7.
C	#nC	Change the text of line n to the line(s) typed after the command is entered (typing a form feed terminates the command).
	#m, nC	Delete lines m through n and replace with the text line(s) typed after the command is entered. (Typing CTRL/FORM indicates the end of the inserted lines.) The C command utilizes the text collector in altering text.
D	#nD	Delete line n from the buffer.
	#m, nD	Delete lines m through n from the buffer. The space used by the line to be deleted is reclaimed as part of the DELETE function. (Refer to Text Collection in the section entitled Editor Text Buffer.)

Table 1-23 Symbolic Editor Commands (Cont.)

Command	Format	Meaning
E	#E	Output the current buffer and transfer all input to the output file, closing the output file.
F	#F	Follows a string search. Look for next occurrence of the string currently being sought. (See section under Search Mode concerning Inter-Buffer Character String Search.)
G	#G	Get and list the next line which has a label associated with it. A label in this context is any line of text which does not begin with one of the following: space (ASCII 240) / (ASCII 257) TAB (ASCII 211) RETURN (ASCII 215) At the termination of a G command, control goes to command mode with the current line counter equal to the line just listed.
	#nG	Get and list the first line which begins with a label, starting the search at line n.
I	#I	Insert whatever text is typed before line 1 of the text buffer. The form feed (CTRL/FORM) terminates the entering process and sends control to the command mode where Editor prints a #.
	#nI	Insert whatever text is typed (until a form feed is typed) before line n of the text buffer.
J	#J	Inter-buffer search command for character strings (see section under Search Mode concerning Inter-Buffer Character String Search).
K	#K	Kill the buffer. Reset the text buffer pointers so that there is no text in the buffer.

Table 1-23 Symbolic Editor Commands (Cont.)

Command	Format	Meaning
		NOTE The Editor ignores the commands nK or m,nK. This is to prevent the buffer from accidentally being destroyed if the user means to type a List command (m,nL).
L	#L	List entire contents of the text buffer on the terminal.
	#nL	List line n of the text buffer on the terminal.
	#m, nL	List lines m through n of the text buffer on the terminal. Control then returns to command mode.
M	#m, n$xM	Move lines m through n directly before line x in the text buffer. The $ character represents typing the dollar sign key (SHIFT/4). The old occurrence of the moved text is removed; no buffer space is lost.
N	#N	Write the current buffer to the indicated output file and read the next logical page.
	#nN	Write the current buffer to the output file, zero the buffer, and read the next logical page. This is done n times until the nth logical page is in the text buffer. Control then returns to command mode. The N command cannot be used with an empty text buffer. ? is printed if this is attempted.
P	#P	Write the entire text buffer to the output buffer.
	#nP	Write line n of the text buffer to the output buffer.
	#m, nP	Writes lines m through n, inclusive, to the output buffer. When this buffer is

Table 1-23 Symbolic Editor Commands (Cont.)

Command	Format	Meaning
		full, the text is output to the indicated output file. The P command automatically outputs a FORM character (214) after the last line of output.
Q	#Q	Immediate end-of-file. Q causes the text buffer to be output. All text written into the output buffer is then written into the output file and the file closed, with control returning to the Keyboard Monitor.
R	#R	Read from the specified input device and append the new text to the current contents of the buffer. If no input file was indicated or if no input remains, a ? is printed and control returns to command mode.
S	#S	Character search command (see the section entitled Search Mode).
T	#T	Punch trailer tape. Causes 32 frames of blank tape to be written into the output buffer (only to non-directory devices).
V	#V	If an LP08 line printer is available, the V command causes the entire text buffer to be listed on the line printer.
	#nV	List line n of buffer on the line printer.
	#m, nV	List lines m through n inclusive on the line printer.
Y	#nY	Skip to a logical page in the input file, without writing any output. For example: #5Y reads through four logical pages of input, deleting them without producing output. The fifth page is read into the text buffer and control automatically returns to command mode.

Table 1-23 Symbolic Editor Commands (Cont.)

Command	Format	Meaning
$	#$TEXT" #$TEXT' #"	Perform a character string search for the string TEXT. (See the section under Search Mode concerning Intra-Buffer Character String Search). Following a string search, #" causes a search for the next occurrence of the string.
.= or .: /= or /:		By typing these characters the user can obtain the current line number (.=) and the last line number in the text buffer (/=). The number is printed by the Editor immediately after the user types the equal sign. (The colon character is equivalent to the equal sign.)
>	$>	Equivalent to .+1L; list the next line in the text buffer on the teleprinter.
<	$<	Equivalent to .−1L; list the next line in the text buffer on the teleprinter.
LINE FEED Key		Equivalent to .+1L; list the next line in the text buffer on the teleprinter.
#		Print the current Editor version number.

PERIPHERAL INTERCHANGE PROGRAM (PIP)

PIP is the OS/8 system program which is used to transfer files between devices, merge and delete files, and list, zero, and compress directories.

Calling and Using PIP

To call PIP from the system device the user types:

```
R PIP
```

in response to the dot printed by the Keyboard Monitor. The Command Decoder then prints an asterisk at the left margin of the teleprinter paper and waits to receive a line of I/O files and options. PIP accepts up to nine input files and performs output to a single output file; options generally are placed at the end of the command string.

Since PIP performs file transfers for all file types (ASCII, Image or SAVE format, or Binary), there are no assumed extensions assigned by PIP to file names for either input or output files. All extensions, where present, must be explicitly specified.

Following completion of a PIP operation, the Command Decoder again prints an asterisk at the left margin and waits for another PIP I/O specification line. The user can return to the Keyboard Monitor by typing CTRL/C or by terminating the specification line with ALTMODE.

PIP OPTIONS

The various options allowed on a PIP I/O specification line are detailed in the following table. Either /A, /B, or /I is generally indicated for each transfer; if none of these are specified, the system proceeds as though /A had been typed.

Table 1-24 PIP Options

Option	Meaning
/A	Transfer files in ASCII mode. The file is modified as it is copied: embedded blank tape and rubouts are deleted and leader/trailer code is reducd to a standard length. PIP may also do some editing of the input file under control of the /C and /T options (see below).
/B	Transfer files in Binary mode (used for absolute and relocatable binary files). Leader/trailer code is reduced to a standard length, but the checksum is not recalculated. **NOTE** If several absolute binary files are combined into one, the /S option must be indicated to the Absolute Loader in order for the files to load properly. (The Linking Loader will not load combined files.)
/C	Eliminate trailing blanks. Valid in ASCII mode only.
/D	Delete the old copy of the output file before doing any data transfer. If /D is not used, the old copy is not deleted until all input has been processed. For example: `*DTA1:OFILE<DTA2:NFILE/D` will first delete file OFILE on DTA1, and then transfer the data from NFILE to a new OFILE. /D is useful when insufficient room exists on the output device for both the old file and the new file.

Table 1-24 PIP Options (Cont.)

Option	Meaning
	/D may be used to delete up to three files at a time by specifying the files to be deleted as output files and not specifying any input files. For example: `*OLDABC,DTA3:FILES/D<` This command string deletes OLDABC from DSK, and FILES from DTA3.
/E	List directories in extended form (the lengths of the empty files are also listed).
/F	List directories in short form (file names only).
/G	Ignore any errors which occur during a file transfer and continue copying.
/I	Transfer files in image mode. Used to transfer core image (SAVE format) files, and any other files which do not fall into either ASCII or Binary categories. This option always opens the output file even if no input files were specified. Thus, the /I combined with the −n option allows the user to substitute a named file for an empty one. For example, a 23-block file named IMPORT. PA was accidentally deleted. It can be recovered with the following command: Note that $23_{10} = 27_8$.
/L	List the directories of the input devices onto the output file starting at the file specified. Notice that in this case the input file itself is not transferred, only the directory. The directory listing is in extended form, but empty files are excluded. If no output file is specified, TTY: is assumed if it exists.
=n	Save n extra words per file entry in the directory to contain descriptive information about the file (only the 2 low order octal digits on nnnn are significant). For use with the /Z and /S options only. Typing =1 allows the date of

Table 1-24 PIP Options (Cont.)

Option	Meaning

the file creation to be automatically stored in the directory. (=1 is assumed after /Z or /S options unless otherwise specified. Specifying =0 will still reserve one extra word per entry.) Specifying =100 will reserve no extra words per entry.

If an = option is included with an image mode (/I) transfer, the low order 12-bits of the = option specify the desired length with which to close the output file. The output file is given this length except in the following two cases:

1. If the data written is greater than the specified length, the output file is given its correct size.
2. If the length specified is greater than the empty space available, the data is transferred but the file is not closed. The error message:

   MONITOR ERROR 6 AT xxxx
   (DIRECTORY OVERFLOW)

   is printed and control returns to the Keyboard Monitor. Data in the file following the EMPTY is *not* destroyed.

/O Okay to compress files or to zero the directory. When used with the /S or /Z option, /O prevents the messages ARE YOU SURE? and ZERO SYS? from printing. The system assumes that the user really wants the /S or /Z option.

/S Move all files from the input device to the output device, eliminating any embedded empty files. All device names should be explicitly stated, as no default devices are assumed. The directory of the output device will contain only those files that appeared on the input device. Whenever a /S is initiated, PIP asks:

```
ARE YOU SURE?
```

The user responds with a "Y" if he wishes the compression; typing any other character aborts the command.

Table 1-24 PIP Options (Cont.)

Option	Meaning

NOTE

When the /S option is used, the output device directory is read to determine whether it is a system directory. If a system exists on the output device, that system will be preserved on the /S transfer. To eliminate the system directory, a /Z must be performed before the /S.

In addition to compressing directories, /S provides a means of copying one device to another. DECtapes, for example, can be copied by compressing one DECtape onto another tape.

/T Perform the following conversions of special characters:

Character	Is Converted To:
TAB	enough spaces to reach the next TAB stop (every eighth position)
Vertical TAB	5 LINE FEEDs
FORM FEED	9 LINE FEEDs

/T option is valid in ASCII mode only.

/V Print the current version number of PIP. This option should be included in the first command line entered after PIP is called. The version number is printed on the console terminal.

/Y Copy the OS/8 System Area (records 0, 7-67) between the output and first input file. Both devices must be file structured devices. If no file name is specified after a device name, the System Area of that device is assumed. If the /Z option is used with /Y, a zeroed system directory is placed on the output device before the system transfer takes place. A system directory indicates that file storage starts at record 70 rather than record 7.

/Z Zero directory of output device before file transfer. Before using a DECtape for the first time, the /Z option should always be used to create an empty file directory. No input files are specified. For example:

Table 1-24 PIP Options (Cont.)

Option	Meaning
	*DTA2:/Z<=1 One extra word per entry is used if no "=" is specified. Thus, the DATE word is always left available in a new directory. If an attempt is made to zero the directory of the system device, the message: ZERO SYS? is printed. A response of 'Y' will zero the directory; any other response will abort the command and return control to the Command Decoder.

No data transfer occurs if no input files are specified. Thus, as mentioned previously, /Z can be used to zero a directory, and /D can be used to delete a permanent file without creating a file. For the three directory listing options (/E, /F, /L), if no output device is specified, the device TTY: is assumed. If no input device is specified, device DSK: is assumed.

EXAMPLES OF PIP SPECIFICATION COMMANDS

The following are legal command strings to PIP. When PIP has completed an operation, control returns to the Command Decoder for additional input.

Example 1 (ASCII Transfer):

```
.R PIP
*SYS:BLACK<PTR:
```

This command string transfers a tape from the paper tape reader to a file on the system device under the name BLACK. PIP assumes that the input tape is in ASCII format. (Control returns to the Command Decoder, therefore, the .R PIP command need only be given once.)

Example 2 (ASCII File Merge):

```
*DTA3:MERGE<DTA1:FILE1,FILE2
```

This command string instructs PIP to merge the ASCII files FILE1 and FILE2 on DTA1 into one ASCII file, MERGE, on DTA3.

Example 3 (Binary Transfer):

```
*BIN.BN<PTR:/B
```

The above command reads a binary paper tape from the paper tape reader and creates a binary file BIN.BN on the device DSK.

Example 4 (Image Transfer):

```
*SYS:GAG.SV<PAL8.SV/I
```

PIP transfers the core image file PAL8.SV from the device DSK to GAG.SV on the system device.

NOTE

A problem occurs when files longer than 255 blocks are transferred in Image Mode from a directory device. If this is attemped, the transfer will not end with the real end-of-file, but will continue until the output limit is reached; an error message will occur. For example, trying to transfer FORT.PA or SABR.PA from the directory device using Image Mode will cause this error. ASCII mode must be used for all PIP transfers of this type, or the FOTP program may be used.

Example 5 (Directory Listing):

```
*TTY:</E
```

This command string produces an extended listing of the device DSK on the Teletype. An extended listing contains all files with

their associated lengths, and all empty spaces in the directory. For example, an extended listing might appear as follows. (The current date is printed before the file listing provided the DATE command has been given; see the section concerning the Keyboard Monitor for a description of the DATE command.):

```
 2/17/72
EDIT  .SV   12 1/10/72
TEST2        4 1/10/72
ABCD  .DA    1 2/17/72
<EMPTY>      7
TEST2 .RL    4 1/10/72
<EMPTY>    702
 709 FREE BLOCKS
```

The file lengths and number of free blocks are designated as decimal values. The date of file creation is printed if at least one additional information word is present in the directory (refer to the section Additional Information Words in File Directories).

Example 6 (Directory Listing):

```
*/F
```

This command produces a directory listing of file names only. Thus, the preceding directory would appear on the teleprinter as follows:

```
 2/17/72
EDIT  .SV
TEST2
ABCD  .DA
TEST2 .RL
 709 FREE BLOCKS
```

Example 7 (Directory Listing):

```
*LPT:<DTA2:FETCH/L
```

A command such as the above produces a listing of the DTA2 directory on the line printer; however, the files that occur before FETCH are not listed. The /L option gives the regular listing which includes the file name and extension length, and date (if a date is contained in the directory). Empty files are not indicated in the listing.

Example 8 (System Area Transfer):

```
*DTA1:HEAD<SYS:/Y
```

Records 0 and 7-67 are transferred from SYS: to a file named HEAD on DTA1.

Example 9 (System Area Transfer):

```
*SYS:<DTA1:HEAD/Y
```

The contents of the file HEAD on DTA1 are transferred to the System Area (records 0 and 7-67) of the system device. The input file is checked for validity before the transfer occurs.

Example 10 (System Transfer with Directory Zero):

```
*DTA1:<DTA0:(YZ)
```

This first creates a zero system directory on DTA1, and then transfers the system area from DTA0 to to the System Area on DTA1. A system directory indicates that file storage begins at record 70 rather than record 7.

Example 11 (System Area Transfer):

```
*DTA1:TRAN<DTA2:TRAN/Y
```

This command string instructs PIP to transfer TRAN from DTA2 to DTA1. Since the /Y option is used, TRAN must be a copy of the OS/8 System Area. However, since transfers of this type involve files on both the I/O devices, and not the System Area, the transfer is treated as an image transfer and either the /Y or /I options can be used.

Additional Information Words in File Directories

If a device has any additional information words specified in its directory, OS/8 automatically enters the last date specified in a DATE command into the first of the additional information words when a file is created on that device. Dates put into these additional words appear in directory listings. Words after the first are not currently used by the OS/8 system.

Whenever a /Z or /S is given, additional words can be specified by a /Z=n or /S=n construction. The number of additional words can be changed by compressing a device onto itself. The

first additional information word is used by the system for the file's creation date.

NOTE

The system is initially created with one additional word in the file directory.

PIP Error Messages

The following messages are printed by PIP in response to user errors or improper command strings:

Table 1-25 PIP Error Messages

Message	Meaning
ARE YOU SURE?	Occurs when using the /S option. A response of 'Y' will compress the files.
BAD DIRECTORY ON DEVICE # n	Error message occurs when: 1. PIP is trying to read the directory, but it is not a OS/8 directory. 2. The output device does not have a system directory, i.e., file storage begins at record 7 (occurs during a /Y transfer). n is the number of the file in the input file list.
BAD SYSTEM HEAD	If the /Y option is used and the area being transferred does not contain OS/8, this error message results.
CAN'T OPEN OUTPUT FILE	Message has occurred due to one of the following: 1. Output file is on a read-only device. 2. No name has been specified for the output file. 3. A /Y transfer has been attempted to a non-directory device. 4. Output file has zero free blocks.
DEVICE # n NOT A DIRECTORY DEVICE	Message occurs when: 1. Trying to list the directory of a non-directory device. 2. The input designated in a /Y transfer is not on a directory device. n gives the number of the device in the input list.

Table 1-25 PIP Error Messages (Cont.)

Message	Meaning
DIRECTORY ERROR	An error has occurred while reading or writing the directory during a /S option. The option is aborted; output is likely to be garbled.
ERROR DELETING FILE	An attempt was made to delete a file that does not exist. Check that the device name was explicitly given for all files.
ILLEGAL BINARY INPUT, FILE # n	Self explanatory; n is the number of the file in the input file list.
INPUT ERROR, FILE # n	An input error occurred while reading file number n in the input file list.
IO ERROR IN (file name) —CONTINUING	An error has occurred during a /S transfer. The name of the file being transferred is indicated.
LINE TOO LONG IN FILE # n	In ASCII mode a line has been found greater than 140 characters. Make certain the file is an ASCII file. n is the number of this file in the input list.
NO ROOM FOR OUTPUT FILE	Self-explanatory; either room on device or room in directory is lacking.
NO ROOM IN (file name) —CONTINUING	Occurs during use of the /S option. The output device cannot contain all of the files on the input device. The message is printed for each file which will not fit into the output device. The file name is indicated.
OUTPUT ERROR	Output error—possibly a WRITE LOCKed device, parity error, or attempt to output to a read-only device.
PREMATURE END OF FILE, FILE # n	Message occurs in Binary Mode (/B) only. A physical end-of-file has been found before the final leader/trailer.
SORRY—NO INTERRUPTIONS	Error message occurs if: 1. ↑C (CTRL/C) is typed while compressing a file onto itself; the transfer continues.

Table 1-25 PIP Error Messages (Cont.)

Message	Meaning
	2. A /Y transfer is done with system device as the output device, or if the transfer has both input and output on the same device.
ZERO SYS?	If any attempt is made to zero the system device directory, this message occurs. Responding with 'Y' causes the directory to be zeroed. Any other character aborts the operation.

ABSOLUTE BINARY LOADER

The Absolute Binary Loader is used to load the binary output created by the PAL8 assembler. Input files are loaded according to the options discussed in this section, and a core control block is constructed (see the section concerning the GET command). The standard input devices are the paper tape reader, DECtape, LINCtape, the default storage device (DSK:), and SYS:, which represents the system device. Any other device which can contain absolute binary files can be used as an input device if a device handler exists. The terminal (TTY:) should not be used, as the binary code may appear as control characters to the TTY handler.

Calling and Using ABSLDR

ABSLDR normally accepts absolute binary files (relocatable files must be loaded with the Linking Loader); however, save (.SV) format files can be loaded with ABSLDR providing the /I option is used. If no extension to the input file name is typed, ABSLDR assumes the .BN extension. Up to nine input files are allowed, but if more than one program is present in a file, only the first program is loaded unless the /S option is used. (This feature allows ABSLDR to ignore any 'noise characters' which might be caused by reading over the end of a paper tape.)

The user calls the Absolute Binary Loader from the system device by typing:

```
R ABSLDR
```

in response to the dot printed by the Keyboard Monitor. The system responds by printing an asterisk at the left margin. The user then types an input line to ABSLDR, indicating input files and any options desired. ABSLDR does not recognize any output files, since the purpose of the loader is to load and optionally start binary output files. The format of the input line is:

*DEV:INPUT.EX/(Options)

By typing the RETURN key at the end of an input specification line, the loader is signalled that more input is to be given on the next line. If the ALT MODE key is used as a line terminator, no more input is expected, the Command Decoder is not recalled, and control returns to the Keyboard Monitor. For example:

```
.R ABSLDR
*DTA1:FILE1,FILE2,FILE3,FILE4    (Carriage RETURN)
*PTR:$
                                 (ALT MODE)
```

The preceding lines cause FILE1, FILE2, FILE3, and FILE4 to be loaded at their absolute locations in core from DECtape 1. A file is then to be read from the paper tape reader. The $ character is printed by the ALT MODE key which indicates a return to the Keyboard Monitor.

NOTE

If the /G option (load and begin execution) is specified, control always passes to the program just loaded, regardless of which line terminator was typed.

When ABSLDR has completed loading and control has returned to the Keyboard Monitor, the program loaded may *not* be physically in core at that moment. ABSLDR utilizes system scratch blocks to store those locations which would overlay various parts of the Monitor. To examine core locations after using ABSLDR, use ODT (see the section concerning ODT for instructions detailing its use).

ABSLDR OPTIONS

The various options accepted by ABSLDR are described in Table 1-26.

Table 1-26 ABSLDR Options

Option	Meaning
/8	Used when locations 0-1777 of field 0 are not being used by the program. Eliminates extra DECtape motions to save these locations, hence saves time. See the OS/8 Software Support Manual for details of Job Status Word.
/9	Similar to the /8 option; used when locations 0-1777 of field 1 are not to be saved.
/I	Treat the input file(s) as a core-image-file to be overlaid with the input of succeeding lines. (If this option is not used in the first command line, it cannot be used unless ABSLDR is recalled from the Keyboard Monitor level.) The /I option can be used to make patches to an already saved program without reassembling the entire program.
/R	Reset internal core map of ABSLDR to appear as though nothing has been loaded into core.
/S	Load all binary programs in the specified input file(s) (instead of loading only the first program in each file, which is normally done). /S and /I operate on a line-at-a-time basis. Each successive command line must have the option respecified if it is required. For example: *PTR:,,/S *DTA1:A,B,C These command strings instruct ABSLDR to take three files from PTR (loading all binary programs in each file) and three files from DTA1 (loading only the first binary program in each file). /S is not implemented on the second line.
/P	Sets bit 3 of the Job Status Word (location 07746) and prevents the Keyboard Monitor from reading a fresh version of the BATCH monitor into core every time the monitor level is reentered from the program level. This option can be used with system programs that never use more than 8K of core (PIP, FORTRAN II, SABR). The /P option should not be used with any program that occupies or modifies core above field 1. (See the BATCH section for further information.)

Table 1-26 ABSLDR Options (Cont.)

Option	Meaning
/G	Start program execution upon finishing the loading procedure. Normally, control returns either to the Monitor or Command Decoder (depending on the terminator key). If /G is specified, control is given to the program just loaded. The starting address is assumed to be 200 unless specified in the input string. Control stays with the user's program until it is released to the Monitor from within the program. No automatic return to Monitor or the Command Decoder occurs.
/n	Force loading of all files specified on this input line into field n (where n is an octal integer).
=n	Set the starting address of the program in core to n, where n is a 5 digit octal integer. ABSLDR inserts a starting address of 0200 in field 0 if no other address is indicated. Specifying 0 as a starting address is equivalent to not specifying a starting address, thus ABSLDR would insert a starting address of 0200.

EXAMPLES OF INPUT LINES

Example 1:

```
.R ABSLDR
*SYS:PROG.SV/I
*DTA1:PATCH$
.SAVE SYS:PROG
```

The above commands load the core-image file PROG.SV and then overlay part of that program file with a binary patch from DTA1. Control then returns to Monitor, at which time the user saves the patched program on the system device.

When using the /I option, the starting address and Job Status Word of the core image being loaded are ignored by the Loader. The user must specify the starting address and contents of the Job Status Word (unless the starting address is 200 in field 0, in which case it need not be specified).

Example 2:

```
.R ABSLDR
*PIP.SV/I
*PTR:=13002(89)$
.SAVE SYS PIP
```

In this example, the user overlays PIP with a binary patch which will not change its starting parameters. This could also be accomplished using an explicit SAVE:

```
.R ABSLDR
*PIP.SV/I
*PTR:$
.SAVE SYS PIP;13000=6003
```

Example 3:

```
.R ABSLDR
*PTR:(89G)$
```

One binary tape is loaded from the paper tape reader. Areas 00000-01777 and 10000-11777 of core are not used by the program. The starting address of the program is considered to be 00200; control is transferred to the user program.

Notes On Using ABSLDR Correctly

ABSLDR is a complex program which, when used incorrectly, can give unrecoverable errors. Points to remember when using ABSLDR are:

1. If an erroneous starting address is specified, control *will* be passed to that address, however random it may be. Thus, specifying a starting address in non-existent memory, for example, will very likely produce erroneous results, and should not be attempted.
2. Trying to load a program into non-existent memory should not be attempted.
3. Programs which load into 07600 or 17600 are ignored by ABSLDR. No error is generated, but these locations are never loaded. (It is a good idea not to use 7600 in any field.)
4. Old versions of ABSLDR should not be used with a new monitor.
5. New versions of ABSLDR should not be used with old monitors.

ABSLDR Error Messages

Table 1-27 lists the error messages output by ABSLDR. In each case, control returns to the Command Decoder; the entire procedure may be attemped again by resetting the loader (with the /R option) and using different inputs.

Table 1-27 ABSLDR Error Messages

Message	Meaning
BAD CHECKSUM, FILE # n	File number n of the input file list has a checksum error.
BAD INPUT, FILE # n	Attempt was made to load a non-binary file as file number n of the input file list, or a non-core image with /I option.
IO ERROR FILE # n	An I/O error has occurred in input file number n.
NO INPUT	No input file was found on the designated device.
NO /I!	Use of /I is prohibited at this point.

OCTAL DEBUGGING TECHNIQUE (ODT)

ODT allows the programmer to run his program on the computer, control its execution, and make alterations to the program by typing instructions at the keyboard.

Features

ODT features include location examination and modification; and instruction breakpoints to return control to ODT (breakpoints). ODT makes no use of the program interrupt facility and is invisible to the user program.

The breakpoint is one of ODT's most useful features. When debugging a program, it is often desirable to allow the program to run normally up to a predetermined point, at which the programmer may examine and possibly modify the contents of the accumulator (AC), the link (L), or various instructions or storage locations within his program, depending on the results he finds. To accomplish this, ODT acts as a monitor to the user program.

The user decides how far he wishes the program to run and ODT inserts an instruction in the user's program which, when encountered, causes control to transfer back to ODT. ODT immediately preserves in designated storage locations the contents of the AC and L at the breakpoint. It then prints out the location at which the breakpoint occurred, as well as the contents of the AC at that point. ODT will then allow examination and modification of any location of the user's program (or those locations containing the AC and L). The user may also move the breakpoint, and request that ODT continue running his program. This will cause ODT to restore the AC and L, execute the trapped instruction and continue in the user's program until the breakpoint is again encountered or the program is terminated normally.

Calling and Using ODT

As explained in the section concerning the Keyboard Monitor, ODT is called into use by typing:

```
ODT
```

in response to the dot printed by the Keyboard Monitor. Before ODT is called, the user should have a running version of his program in memory. None of the user's memory is disturbed by the running of ODT, because the sections of the program which ODT may occupy when in memory are preserved on the system device and swapped back into memory as necessary. ODT uses the Job Status Word of the particular program to determine whether or not swapping occurs. If the program does not use locations 0-1777 in field 0, less swapping occurs during use of the breakpoint feature.

If the user is typing any amount of a program directly into memory (in octal), the memory control block of the program may not reflect the true extent of the program. If octal additions are made below location 2000 in field 0, ODT may give erroneous results. The user can correct this condition by correcting the Job Status Word, which is location 7746 of field 0, and which can be examined and changed using ODT. Location 7745 of field 0 is the 12-bit starting address of the program in memory and location 7744 contains the field designation in the form 62n3, where n is the field designation of the starting address.

When using the breakpoint feature of ODT, the user should keep certain operating characteristics in mind:

1. If a breakpoint is inserted at a location which contains an auto-indexed instruction, the auto-indexed register is bumped immediately after the breakpoint is hit. Thus, when control returns to the user in ODT, the register will have been increased by one. The breakpoint instruction is executed properly, but the index register, if examined, may appear one greater than it should.
2. ODT keeps track of the TTY flag and restores the TTY flag when it continues from a breakpoint.
3. The breakpoint feature uses locations 4, 5, and 6 in the memory field in which the breakpoint is set.
4. The breakpoint feature of ODT uses the table of user-defined device names as scratch storage, destroying any device names the user may have created. After a session with ODT in which breakpoints are used, the user should give a DEASSIGN command to clear out the user-device name table.
5. Breakpoints must not be set in the Monitor, in the device handlers, or between a CIF and the following JMP instruction.

The user is advised not to use user-defined device names in programs being developed with ODT breakpoints.

If any operations are attempted in non-existent memory, ODT ignores the command and types "?". Thus, assuming the machine in use has 8K (fields 0 and 1) and the user attempts to examine locations in field 2 and above, ODT responds with ?.

ODT should not be used to debug programs which use interrupts. Typing CRTL/C returns control to the Keyboard Monitor; the program can be saved on any device.

Commands

SPECIAL CHARACTERS

Slash(/)—Open Preceding Location

The location examination character (/) causes the location addressed by the octal number preceding the slash to be opened and its contents printed in octal. The open location can then be modified by typing the desired octal number and closing the location. Any octal number from 1 to 4 digits in length is legal input. If more than 4 digits are entered, only the last 4 entered are accepted by ODT. Typing / with no preceding argument causes the latest named location to be opened.

For example:

```
400/1540
400/1540 2468?
400/1540 12345
/2345
```

Return—Close Location

If the user has typed a valid octal number after the content of a location is printed by ODT, typing the RETURN key causes the binary value of that number to replace the original contents of the opened location and the location to be closed. If nothing has been typed by the user, the location is closed but the content of the location is not changed. For example:

```
400/6046          location 400 is unchanged.
400/6046 2345     location 400 is changed to contain 2345.
/2345 6046        replace 6046 in location 400.
```

Typing another command will also close an opened register. For example:

```
400/6046 401/6031 2346   location 400 is closed and unchanged
400/6046 401/2346        and 401 is opened and changed to 2346.
```

Line Feed—Close Location, Open Next Location

The LINE FEED key has the same effect as the RETURN key, but, in addition, the next sequential location is opened and its contents printed. For example:

```
400/1540          location 400 is closed unchanged and 401 is
00401 /2345 1234  opened. User types change, 401 is closed con-
00402 /7650       taining 1234 and 402 is opened.
```

↑(Shift/N)—Close Location, Take Contents as Memory Reference and Open Same

The up arrow will close an open location just as will the RETURN key. Further, it will interpret the contents of the location as a memory reference instruction, open the location referenced and print its contents. For example:

```
404/3270 ↑
00470 /4512 0000
```

3270 symbolically is "DCA, this page, relative location 70," so ODT opens location 470.

←(Shift/0) Close Location, Open Indirectly

The back arrow will close the currently open location and then interpret its contents as the address of the location whose contents it is to print and open for modification. For example:

```
365/3203 ↑
00203 /3572 ←
03572 /0216
```

ILLEGAL CHARACTERS

Any character that is neither a valid control character nor an octal digit causes the current line to be ignored and a question mark printed. For example:

```
4: ?
40?
```

ODT opens no location.

```
406/1136 67K?
/1136
```

ODT ignores modification and closes location 406.

CONTROL COMMANDS

nnnnG—Transfer Control to User at Location nnnn

Clear the AC then go to the location specified before the G. All indicators and registers will be intialized and the breakpoint, if any, will be inserted. Typing G alone will cause a jump to location 0.

nnnnB—Set Breakpoint at User Location nnnn

Instructs ODT to establish a breakpoint at the location specified before the B. If B is typed alone, ODT removes any previously established breakpoint and restores the original contents of the break location. A breakpoint may be changed to another location whenever ODT is in control, by simply typing nnnnB where nnnn is the new location. Only one breakpoint may be in effect at one time; therefore, requesting a new breakpoint removes any previously existing one.

A restriction in this regard is that a breakpoint may not be set on any of the floating-point instructions which appear as arguments of a JMS.

The breakpoint (B) command does not make the actual exchange of ODT instruction for user instruction, it only sets up the mechanism for doing so. The actual exchange does not occur until a "go to" or a "proceed from breakpoint" command is executed.

When, during execution, the user's program encounters the location containing the breakpoint, control passes immediately to ODT (via location 0004). The C(AC) and C(L) at the point of the interruption are saved in special locations accessible to ODT. The user instruction that the breakpoint was replacing is restored, before the address of the trap and the content of the AC are printed. The restored instruction has not been executed at this time. It will not be executed until the "proceed from breakpoint" command is given. Any user location, including those containing the stored AC and Link, can now be modified in the usual manner. The breakpoint can also be moved or removed at this time.

An example of breakpoint usage follows the section "Continue and Iterate Loop . . ."

A—Open C(AC)

When the breakpoint is encountered the C(AC) and C(L) are saved for later restoration. Typing A after having encountered a breakpoint, opens for modification the location in which the AC was saved and prints its contents. This location may now be modified in the normal manner (see Slash) and the modification will be restored to the AC when the "proceed from breakpoint" command is given.

Open C(L)

Typing L opens the Link storage location for modification and prints its contents. The Link location may now be modified as usual (see Slash) and that modification will be restored to the Link when the "proceed from the breakpoint" command is given.

C—Proceed (Continue) From a Breakpoint

Typing C, after having encountered a breakpoint, causes ODT to insert the latest specified breakpoint (if any), restore the contents of the AC and Link, execute the instruction trapped by the previous breakpoint, and transfer control back to the user program at the appropriate location. The user program then runs until the breakpoint is again encountered.

NOTE

If a breakpoint set by ODT is not encountered while ODT is running the object (user's) program, the instruction which causes the break to occur will not be removed from the user's program.

nnnnC—Continue and Iterate Loop nnnn Times Before Break

The programmer may wish to establish the breakpoint at some location within a loop of his program. Since loops often run to many iterations, some means must be available to prevent a break from occurring each time the break location is encountered. This is the function of nnnnC (where nnnn is an octal number). After having encountered the breakpoint for the first time, this command specifies how many additional times the loop is to be iterated before another break is to occur. The break operations have been described previously in the section on the B command.

Given the following program, which increases the value of the AC by increments of 1, the use of the Breakpoint command may be illustrated.

```
*200

        0200   *200
00200   7300           CLA CLL
00201   1206   A,      TAD ONE
00202   2207   B,      ISZ CNT
00203   5202           JMP B
00204   5201           JMP A
00205   7402           HLT
00206   0001   ONE,    1
00207   0000   CNT,    0
               $
*200

A       0201
B       0202
CNT     0207
ONE     0206
0201B
200G
00201 (0;0000
C
00201 (0;0001
C
00201 (0;0002
4C
00201 (0;0006
```

ODT has been loaded and started. A breakpoint is inserted at location 0201 and execution stops here showing the AC initially set to 0000. The use of the Proceed command (C) executes the program until the breakpoint is again encountered (after one complete loop) and shows the AC to contain a value of 0001. Again execution continues, incrementing the AC to 0002. At this point, the command 4C is used, allowing execution of the loop to continue 4 more times (following the initial encounter) before stopping at the breakpoint. The contents of the AC have now been incremented to 0007.

M—Open Search Mask

Typing M causes ODT to open for modification the location containing the current value of the search mask and print its contents. Initially the mask is set to 7777. It may be changed by opening the mask location and typing the desired value after the value printed by ODT, then closing the location.

M Line Feed—Open lower search limit

The word immediately following the mask storage location contains the location at which the search is to begin. Typing the LINE FEED key to close the mask location causes the lower search limit to be opened for modification and its contents printed. Initially the lower search limit is set to 0001. It may be changed by typing the desired lower limit after that printed by ODT, then closing the location.

M Line Feed—Open upper search limit

The next sequential word contains the location with which the search is to terminate. Typing the LINE FEED key to close the lower search limit causes the upper search limit to be opened for modification and its contents printed. Initially, the upper search limit is the beginning of ODT itself, 7000 (1000 for low version). It may also be changed by typing the desired upper search limit after the one printed by ODT, then closing the location with the RETURN key.

nnnnW—Word Search

The command nnnnW (where nnnn is an octal number) will cause ODT to conduct a search of a defined section of memory, using the mask and the lower and upper limits which the user has

specified, as indicated above. The word searching operations are used to determine if a given quantity is present in any of the locations of a particular section of memory.

The search is conducted as follows: ODT masks the expression nnnn which the user types preceding the W and saves the result as the quantity for which it is searching. (All masking is done by performing a Boolean AND between the contents of the mask word, C(M), and the word containing the instruction to be masked.) ODT then masks each location within the user's specified limits and compares the result to the quantity for which it is searching. If the two quantities are identical, the address and the actual unmasked contents of the matching location are printed and the search continues until the upper limit is reached.

A search never alters the contents of any location. For example: search locations 3000 to 4000, for all ISZ instructions, regardless of what location they refer to (i.e. search for all locations beginning with an octal 2).

```
M/7777 7000
7453/5273 3000
```

Change the mask to 7000, open lower search limit.
Change the lower limit to 3000, open upper limit.

```
7454/1335 4000
2000W
00005 /2331
00006 /2324
00011 /2222
00033 /2575
```

Change the upper limit to 4000, close location.
Initiate the search for ISZ instructions
These are 4 ISZ instructions in this section of core.

Additional Techniques

CURRENT LOCATION

The address of the current location or last location examined is remembered by ODT and remains the same, even after the commands G, C, and B are typed. This location may be opened for inspection merely by typing the slash (/) character.

INDIRECT REFERENCES

When an indirect memory reference instruction or an address constant is encountered, the actual address may be opened by typing ↑ and ← (SHIFT/N and SHIFT/O, respectively).

Errors

The only legal inputs are control characters and octal digits. Any other character will cause the character or line to be ignored and a question mark to be printed by ODT. Typing G alone is an error. It must be preceded by an address to which control will be transferred. This will elicit no question mark also if not preceded by an address, but will cause control to be transferred to location 0.

Programming Notes Summary

ODT will not turn on the program interrupt, since it does not know if the user's program is using the interrupt. It does, however, turn off the interrupt when a breakpoint is encountered, to prevent spurious interrupts.

Breakpoints are fully invisible to "open location" commands; however, breakpoints may not be placed in locations which the user program will modify in the course of execution or the breakpoint will be destroyed. Caution should be used in placing a breakpoint between a call to USR function code 10 and the following call to USR function code 11.

If a trap set by ODT is not encountered by the user's program, the breakpoint instruction will not be removed.

ODT can be used to debug programs using floating-point instructions, since the intercom location is 0004, and since breakpoints may be set on a JMS with arguments following.

Summary of ODT Commands

The following table presents a brief summary of the ODT commands. All addresses can be input as 5 digits, and are printed as 5 digits.

Table 1-28 ODT Command Summary

Command	Meaning
nnnnn/	Open location designated by the octal number nnnnn, where the first digit represents the memory field. ODT prints the contents of the location, a space, and waits for the user to enter a new value for that location or close the location.
	Reopen latest opened location.

Table 1-28 ODT Command Summary (Cont.)

Command	Meaning
nnnn;	Deposit nnnn in the currently opened location, close that location and open the next sequential location for modification. A series of octal values can be deposited in sequential locations through use of the ; character. Multiple ;'s skip a memory location for each ; typed and prepare to insert subsequent values beyond the one(s) skipped.
RETURN key	Close the previously opened location.
LINE FEED key	Close location; open the next sequential location for modification, and print the contents of that location.
n+	Open the current location plus n for modification and print the contents of that location.
n−	Open the current location minus n for modification and print its contents.
↑ or ^ (up-arrow or circumflex)	Close location, take contents of that location as a memory reference and open the location referenced, printing its contents. **NOTE** No distinction is made between instruction op-codes when using ↑. Thus, *all* op-codes (0-7) are treated as memory reference instructions. Also, great care should be exercised when using ↑ with indirectly referenced auto-index registers. If ↑ is used in this case, the contents of the auto-index register is incremented by one. The user must check to see that the register contains the proper value before proceeding.
← or — (back-arrow or underline)	Close location, take contents of that location as a 12-bit address and open that address for modification, printing its contents.
nnnnnG	Transfer control of program to location nnnnn, where the first digit represents the memory field.
nnnnnB	Establish a breakpoint at location nnnnn, where the first digit represents the memory field. Only one breakpoint is allowed at any given time.

Table 1-28 ODT Command Summary (Cont.)

Command	Meaning
B	Remove the breakpoint.
A	Open for modification the location in which the contents of the accumulator were stored when the breakpoint was encountered.
L	Open for modification the location in which the contents of the link were stored when the breakpoint was encountered.
C	Proceed from a breakpoint.
nnnnC	Continue from a breakpoint and iterate past the breakpoint nnnn times before interrupting the user's program at the breakpoint location.
M	Open the search mask, initially set to 7777, which can be changed by typing a new value.
LINE FEED	Open the lower search limit. Type in the location (4 octal digits) where the search will begin.
LINE FEED	Open the upper search limit. Type in the location (4 octal digits) where the search will terminate.
nnnnW	Search the portion of core as defined by the upper and lower limits for the octal value nnnn. Search can only be done on a single memory field at a time. See the F command.
D	Open for modification the word containing the data field which was in effect at the last breakpoint. Contents of D always appear as multiples of 10_8—i.e., 10 means field 1, 20 field 2, etc.
CTRL/O	Stop any printing currently in progress.
F	Open for modification the word containing the field used by ODT in the W (search) command, in the ← and ↑ (indirect addressing) commands, or in the last breakpoint (depending upon which was used most recently. The contents of F are always expressed as multiples of 10_8 (as in the D command).
RUBOUT key	Cancels previous number typed, up to the last non-numeric character typed.

OS/8 utility programs

batch
bitmap
boot
build
camp
cref
direct
epic
fotp
mcpip
pip10
resorc
srccom
teco

chapter 2
utility programs

BATCH

Introduction

OS/8 BATCH provides PDP-8 users with a batch processing monitor that is integrated into the OS/8 monitor structure. The system is organized in such a way that it may be used in either a keyboard input configuration or as a batch stream processor.

BATCH may be run on any OS/8 system equipped with at least 12K of memory. A line printer, although optional, is highly desirable. BATCH will support up to 32K of memory and any I/O devices that are present in the system.

OS/8 BATCH processing is ideally suited to frequently run production jobs, large and long-running programs, and programs that require little or no interaction with the user. BATCH permits the user to prepare his job on punched cards, high-speed paper tape or the OS/8 system device and leave it for the computer operator to start and run. Output is returned to the user in the form of line printer and/or teleprinter listings that include program output as well as a comprehensive summary of all action taken by the user program, the monitor system and the computer operator.

BATCH provides optional spooling of output files. This feature serves to increase throughput on any system, but it is particularly valuable when a line printer is not available. BATCH also performs extensive command analysis and error diagnosis, as well as detailed interaction with the user/operator to facilitate initializing the system and establishing system parameters.

Almost any program that runs under interactive OS/8 may also be run under BATCH. Since BATCH is called from the keyboard in the same manner as any other system program, interactive users may use BATCH to execute multiprogram utility routines, even when continuous batch processing is not desired.

With a few exceptions, BATCH uses the standard OS/8 command set. BATCH assumes that the reader is familiar with the operation and use of OS/8.

BATCH Processing Under OS/8

OS/8 BATCH maintains an input file and an output file. The BATCH input file may be a punched card, high-speed paper tape, disk or DECtape file consisting of a series of BATCH commands. If the input file is a disk or DECtape file, it must reside on the OS/8 system device or on a device whose handler is co-resident with the OS/8 system device (e.g., RKB0 on RK05 systems).

Each command in the BATCH input file occupies one file record. If the file is a punched card file, each punched card constitutes one record, which must contain one complete BATCH command. If the file resides on paper tape, disk or DECtape, each record consists of one logical line, or all the characters between two line terminators, including the second terminator.

The BATCH output file is a line printer listing on which BATCH prints job headers, certain messages that result from conditions within the input file, an image of each record in the input file and certain types of user output. If a line printer is not present in the system, the output file is printed on the terminal.

BATCH accepts user input files (i.e., program and data files) from any device in the OS/8 system, with the exception that high-speed paper tape input files are not allowed when the BATCH input file also resides on high-speed paper tape. User output files may be directed to any output device in the system.

BATCH also permits optional spooling of output files. When spooling is requested, every non-file-structured output file is assigned a file name from a list of names maintained by BATCH and directed to a file-structured spool device instead of the user specified device. Spooling of output files increases BATCH throughput when system resources are scarce and permits slow output operations to be postponed until a more favorable time. For example, a batch processing run that generates many output listings may be initialized to re-route all listings from the terminal or line printer to a specified DECtape unit. This DECtape may be dumped onto the appropriate hard copy device after the run, when more time is available. The spool device may be any file-structured device selected by the user.

OS/8 BATCH is called from the keyboard by typing:

R BATCH

(terminated by a carriage return) in response to the dot generated by the OS/8 monitor. BATCH then calls the OS/8 Command Decoder to obtain its parameters, input device and file name (if file-structured). If CCL is enabled, BATCH may also be invoked via the SUBMIT command, in which case the BATCH parameters, input device and file name (if file-structured) are specified on the same line as the SUBMIT command.

The format for a BATCH command string is:

*SPDV:←DEV: INPUT/option/option

where SPDV: is the device on which to spool non-file-structured output. If SPDV: is not specified, no spooling is performed. Note that spooling applies only to non-file-structured output devices specified to the Command Decoder. The output of programs such as FOTP, which use a special mode of the Command Decoder, is not spooled by BATCH. DEV : INPUT is the input device and file if the input is from SYS: or a device whose handler is co-resident with SYS:. The default extension for BATCH input files is .BI. The Run-Time Options are used to specify input from the paper tape reader or the card reader. The Run-Time Options and their meanings are listed in Table 2-1.

Table 2-1 Run-Time Options

Option	Meaning
/C	The input file is to be read from the card reader (CR8/I or CR8/E)
/E	Treat OS/8 Keyboard Monitor and OS/8 Command Decoder errors as non-fatal errors. If /E is not specified, OS/8 Keyboard Monitor and OS/8 Command Decoder errors cause the current job to be aborted.
/P	The input file is to be read from the paper tape reader.
/Q	Do not output a BATCH log. $JOB and $MSG are the only line output to the terminal.
/T	Output the BATCH log to the terminal. This option need be specified only if a line printer is available. If a line printer is not available, the BATCH log is automatically output to the terminal.

Table 2-1 Run-Time Options (Cont.)

Option	Meaning
/U	BATCH will not pause for operator response to $MSG lines. Any attempt to use TTY:, PTR:, or CDR: as input devices to the Command Decoder in an unattended BATCH stream will cause the current job to be aborted.
/V	Print the version number of OS/8 BATCH on the terminal.
/6	Accept card input in DEC 026 format. This option is used only when the /C option is specified. The default card input format is DEC 029.

BATCH Monitor Commands

A BATCH command is a character or string of characters that begins with the first character of a record in the BATCH input file. If the input file is a disk, DECtape or paper tape file, each BATCH command must be followed by a carriage return/line feed combination. If the input file is a punched card file, each command must begin in the first column of a punched card. Disk and paper tape files may contain form feed characters. Form feed characters are ignored by BATCH on input.

OS/8 BATCH recognizes four monitor level commands. These commands allow routine housekeeping operations in a multi-job, batch processing environment and provide communication between the BATCH programmer and the computer operator. Table 2-2 lists the BATCH monitor commands, which may be considered as an extension of the OS/8 Keyboard Monitor command set. Note that the first character of the $JOB, $MSG and $END commands is a dollar sign (shift/4). The BATCH monitor does not recognize the ALT MODE character.

In the current version, any record that begins with a dollar sign character but is not one of the BATCH monitor commands listed above is copied onto the output file and ignored by BATCH.

A BATCH processing job consists of a $JOB command record and all of the commands that follow it up to the next $JOB or $END record. Normally, all the commands submitted by one user are processed as a single job, and all output from these commands appears under one job header.

Table 2-2 BATCH Monitor Commands

Command	Meaning
$JOB	Initialize for a new job and print a job header on the output file. The remainder of the $JOB record is included in the job header but ignored by BATCH. It should be used for job identification, to provide correlation between Teletype output, line printer output and spool device output.
$MSG	Ring the terminal bell and print an image of the record at the teleprinter. If the /U option was not specified, implying that an operator is present, BATCH will pause until any key is struck at the keyboard. If the /U option was specified, processing continues uninterrupted.
$END	Terminate batch processing and exit to the OS/8 Keyboard Monitor. A $END command record should be the last record of every BATCH input file.
/	Copy the record onto the output file, then ignore it. BATCH assumes that every record beginning with a slash is a comment.

After BATCH encounters a $JOB command, it scans the input file until the next Keyboard Monitor command is read. Any records that follow the $JOB command and precede the first Keyboard Monitor command are written onto the output file and ignored by BATCH.

The first character of every Keyboard Monitor command record is a dot (.). The rest of the record contains an OS/8 Keyboard Monitor command, which should appear in standard OS/8 format; however, commands that would be terminated with an ALT MODE under interactive OS/8 should be terminated with a dollar sign under BATCH. Every standard OS/8 Keyboard Monitor command is legal input to BATCH; however, the ODT command will go to the terminal for input instead of the BATCH file. Typing CTRL/C to ODT will terminate BATCH. Type: 7600G to ODT to resume the BATCH run.

BATCH executes a Keyboard Monitor command by stripping off the initial dot character and loading the remainder of the record into the Keyboard Monitor buffer. BATCH then passes control to the Keyboard Monitor, which executes the command as though it

had been typed at the keyboard. Keyboard Monitor commands that return control to the monitor level should be followed by a BATCH monitor command or another Keyboard Monitor command. Keyboard Monitor commands that transfer control to the program level should be followed by a Command Decoder file specification whenever the running program calls the Command Decoder. All OS/8 V3 CCL commands are legal under BATCH, including the SUBMIT command (which can be used to chain from one BATCH stream to another).

When a running program calls the Command Decoder, the Command Decoder determines whether batch processing is in progress and, if so, instructs BATCH to read the next record of the BATCH input file. BATCH expects this record to contain a Command Decoder file specification.

The first character of every Command Decoder file specification record is an asterisk (*). The rest of the record contains an OS/8 Command Decoder file (and/or option) specification, which should appear in standard OS/8 format. As with BATCH monitor commands and Keyboard Monitor commands, any Command Decoder specification that would be terminated with an ALT MODE under interactive OS/8 should be terminated with a dollar sign under BATCH.

BATCH executes a Command Decoder file specification by stripping off the initial asterisk character and loading the remainder of the record into the Command Decoder buffer. BATCH then passes control to the Command Decoder, which decodes the file specification as though it had been typed at the keyboard and returns control to the running program.

If BATCH reads a record from the input file, expecting to find a Command Decoder file specification, and finds a Keyboard Monitor command instead, BATCH returns control to the monitor level by recalling the Keyboard Monitor to execute the command. The running program is terminated and control remains at the monitor level. If BATCH encounters a BATCH monitor command when it expects to find a Command Decoder specification, it executes the BATCH monitor command and continues processing the input file. As long as a Command Decoder file specification is finally read before the next Keyboard Monitor command, control will eventually return to the running program, and the file specification will be executed.

A BATCH monitor command is legal at any level of command execution, and the BATCH monitor returns control to the level from which it was entered. Keyboard Monitor commands are also legal at any level (under BATCH, but not under interactive OS/8); however, the Keyboard Monitor terminates any program that may be running when it is called and returns control to the monitor level.

The computer operator may type CTRL/C at any time during a batch processing run. Typing CTRL/C at the program level causes an effective jump to location 07600, which recalls the BATCH monitor. The BATCH monitor then recognizes the CTRL/C and terminates the BATCH run.

The BATCH Input File

Figure 2-1 shows a listing of a BATCH input file. This listing was produced by using PIP to transfer the BATCH input file from disk to the console terminal, and the output has been reproduced intact. Assume that OS/8 BATCH is loaded on a 12K system containing one TU56 dual DECtape transport, a line printer, a Teletype terminal, and a disk as the system device. If the disk file shown in Figure 1 is specified as an input file, BATCH will begin processing by printing a job header and executing the DATE command.

Control remains at the monitor level, so BATCH executes the next command by calling and starting the Peripheral Interchange Program. PIP, in turn, calls the Command Decoder, which accepts and decodes the file/option specification that occupies the next executable record (following the comment) of the input file. The Command Decoder passes control to the program level, and PIP lists the short form of the system disk directory at the terminal.

If spooling is active, BATCH will intercept this output and store it in a temporary file on the spool device. Assuming that DTA0 is the spool device and this listing is the first non-file-structured output file intercepted by BATCH, the output will be stored in a file named BTCHA1. BATCH then prints the message:

```
#SPOOL TO FILE BTCHA1
```

on both the console terminal and the line printer. The next file that is rerouted to the spool device will be assigned the file name BTCHA2, and successive files will be named:

BTCHA3
BTCHA4
.
.
.
BTCHA9
BTCHB0
BTCHB1
.
.
.
BTCHZ9

```
$JOB OS/8 BATCH PROCESSING EXAMPLE #1
.DATE 3/5/74
.R PIP
/LIST SYSTEM DEVICE DIRECTORY ON TELETYPE
*TTY:<SYS:/F
/NOW LIST THE DIRECTORY OF DECTAPE #3 ON THE LPT
$MSG MOUNT TAPE #3 ON UNIT 1
*LPT:<DTA1:/L
/NOW TRANSFER FORTRAN SOURCE PROGRAM
/FROM DISK TO DECTAPE #3 (UNIT 1)
$MSG WRITE ENABLE UNIT 1
*DTA1:FORTS1.FT<DSK:FORTS1.FT
/COMPILE FORTRAN SOURCE
.R FORT
*DTA1:FORTS1.RL,FORTS1.LS<FORTS1.FT
/THAT CONCLUDES JOB #1
$JOB OS/8 BATCH PROCESSING EXAMPLE #2
$MSG MOUNT TAPE #2 ON UNIT 1, WRITE ENABLED
.R PAL8
*PTP:,DTA1:PROG.LS<DTA1:PROG.PA
.RUN DSK CREF
*DTA1:PROG.LS
/END OF EXAMPLE #2 AND END OF INPUT FILE
$END
```

Figure 2-1: Sample BATCH Input File

allowing a total of 260 spool device files, which is more than adequate in view of the limited maximum size of the OS/8 file directory (about 240 entries). If output to a spool device file is generated by a program that appends a default extension to output file names, the spool device file will be assigned a standard default extension. All of the spool device files may then be transferred to the terminal or line printer by using the program FOTP, with the input file specification dev:BTCH??.*.

Returning to the example of Figure 2-1, PIP executes the file specification that appears in the fifth record, of the input file and recalls the Command Decoder.

The Command Decoder then instructs BATCH to scan the input file for the next file specification record. BATCH processes the comment record by copying it onto the line printer, then processes the $MSG command by ringing the terminal bell, copying the $MSG record onto the terminal, and, assuming that an operator is present, pausing until any key is typed at the terminal.

Once the operator has resumed processing by typing any character, BATCH reads the eighth record in the file, recognizes it as a Command Decoder specification record, and transfers control back to the Command Decoder.

Processing continues in this manner until the third Command Decoder specification record is read. When BATCH searches for the next file specification record, it reads and executes the last $MSG command, then encounters a Keyboard Monitor command. BATCH passes this command to the Keyboard Monitor, which terminates PIP and calls the FORTRAN compiler to load and compile source program FORTS1. Upon completion of these operations, FORTRAN routes its output to the specified files and returns control to the monitor level. BATCH then encounters the second $JOB record, causing it to terminate the current job and print a new header.

The second job calls PAL8 to assemble a source program from disk. The output listing is directed to the user's DECtape #2, mounted on unit 1, while the binary output file is dumped onto high-speed paper tape. The job concludes by running CREF to produce a cross-referenced listing of the assembled program.

This job illustrates how OS/8 BATCH may be used to execute multiprogram utility routines. If user #2 is a programmer who usually follows a PAL8 assembly by running CREF, job #2 could be a utility routine that combines the call to PAL8, the call to CREF and both file specifications into a single software package which may be run under batch processing or in an interactive environment.

The $END record that appears as the last record in Figure 2-1 serves as a signal that batch processing has concluded and causes BATCH to recall the Keyboard Monitor and re-establish interactive

processing under OS/8. This command is always the last record of the BATCH input file.

BATCH Error Messages

BATCH generates two types of error messages. BATCH generates run-time error messages which appear in the form:

#BATCH ERR

the second type of error message is generated when the Keyboard Monitor or the Command Decoder recognizes a command error in the BATCH input file. When this occurs, either the Keyboard Monitor or the Command Decoder will transmit a standard OS/8 error message and BATCH will append a "#" character to the beginning of the message, so that it appears in the form:

#SYSTEM ERROR

Any occurrence of a Keyboard Monitor or Command Decoder error normally causes BATCH to abort the current job and scan the input file for the next $JOB command. If the /E option was specified, BATCH treats Keyboard Monitor and Command Decoder errors as non-fatal and continues the BATCH run.

Table 2-3 lists the BATCH error messages, their meanings, and the probable cause for the error.

Table 2-3 BATCH Error Messages

BATCH Error Message	Meaning
#MONITOR OVERLAYED	The Command Decoder attempted to call the BATCH monitor to accept and transmit a file specification, but found that a user program had overlayed part or all of the BATCH monitor. Control returns to the monitor level, and BATCH executes the next Keyboard Monitor command.
#BAD LINE. JOB ABORTED	The BATCH monitor detected a record in the input file that did not have one of the characters dot, slash, dollar sign or asterisk as the first character of the record. The record is ignored, and BATCH scans the input file for the next $JOB record.

Table 2-3 BATCH Error Messages (Cont.)

BATCH Error Message	Meaning
#SPOOL TO FILE BTCHA1	Where the "A" may be any character of the alphabet and the "1" may be any decimal digit. This message indicates that BATCH has intercepted a non-file-structured output file and rerouted it to the spool device. This is not, generally, an error condition. Spool device file names are assigned sequentially, beginning with file BTCHA1. Standard default extensions may be assigned by some system programs.
#MANUAL HELP NEEDED	BATCH is attempting to operate an I/O device, such as PTR or TTY, that will require operator intervention. If the initial dialogue indicated that an operator is not present, this message is suppressed, the current job is aborted, and BATCH scans the input file for the next $JOB command record. If an operator is present, he should have been notified what action to take by a $MSG command.
#ILLEGAL INPUT	A file specification designated TTY or PTR as an input device when the initial dialogue indicated that an operator is not available. The current job is aborted, and BATCH scans the input file for the next $JOB command record.
#INPUT FAILURE	Either a hardware problem prevented BATCH from reading the next record of the input file, or BATCH read the last record of the input file without encountering a $END command record. If a hardware problem exists, correct the problem and type any character at the Teletype to resume processing.

Table 2-3 BATCH Error Messages (Cont.)

BATCH Error Message	Meaning
#SYS ERROR	A hardware problem prevented BATCH from performing an I/O operation. Program execution halts, and the system must be restarted manually. This message often indicates that the system device is not write enabled.
INSUFFICIENT CORE FOR BATCH RUN	OS/8 BATCH requires 12K of core to run. Control returns to the OS/8 Monitor.
BATCH.SV NOT FOUND ON SYS:	A copy of BATCH.SV must exist on the system device. Control returns to the OS/8 Monitor.
WRONG OS/8 MONITOR	OS/8 BATCH requires an OS/8 Monitor no older than version 3.
DEV NOT IMPLEMENTED	BATCH cannot accept input from the specified input device because its handler is not permanently resident (SYS: or co-resident with SYS:). Control returns to the Command Decoder.
ILLEGAL SPOOL DEVICE	The device specified as a spooling output device must be file-structured. Control returns to the Command Decoder.

Running BATCH From Punched Cards

The carriage return and ALT MODE characters are not defined in the punched card character set. BATCH permits terminating carriage return characters to be omitted from punched card input files. Thus, when BATCH reads a punched card input file, it appends a carriage return to the content of each card, immediately following the last character on the card that is not a space character. As with disk, DECtape or paper tape input files, BATCH considers the dollar sign character to be equivalent to an ALT MODE when it appears on a punched card in any column except the first.

When BATCH is run with a punched card input file, it is possible for user input files to be embedded in the BATCH input file. User input files should be inserted into the BATCH input file in such a way that BATCH will never attempt to read a record of the user file. That is, user files should follow a command record that transfers control to the program level, and the running program must exhaust all records of the user file before returning to the monitor level.

Figure 2-2 illustrates how the second sample job of Figure 2-1 may be modified to run from a punched card input file with an embedded user file. In this example, PAL8 reads the punched card user file and assembles the source program, then returns control to the monitor level. BATCH reads the next card of the input file, which should contain the .R CREF command. If PAL8 has not read every record of the user input file, however, BATCH will encounter a record from this file rather than the Keyboard Monitor command record. This results in the message:

```
#BAD LINE. JOB ABORTED
```

and causes BATCH to scan the input file until the next $JOB record is read.

Restrictions Under OS/8 BATCH

OS/8 BATCH is a "friendly" system; that is, one which is largely unprotected from user errors. The BATCH monitor resides in locations 5000 to 7577 in the highest memory field available. BATCH also uses the following locations in field 0 and the memory field in which it resides:

LOCATION	USED AS:
07777	Batch processing flag.
N7774-N7777	Internal pointers.

Both the Keyboard Monitor and the Command Decoder check the batch processing flag whenever they are entered from the program level. Any user program that modifies location 07777 may cause batch processing to be terminated prematurely before the next record of the BATCH input file is read.

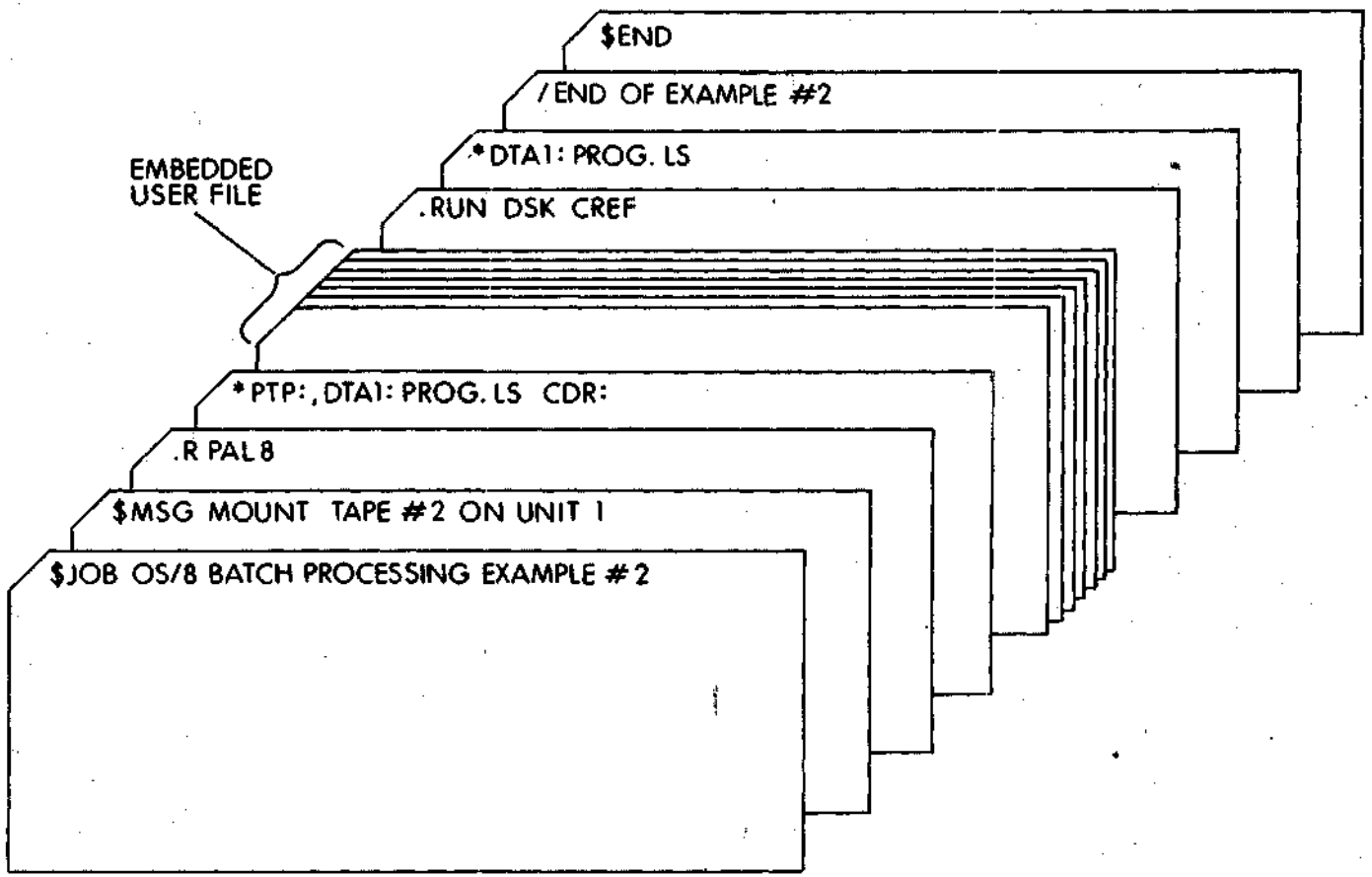


Figure 2-2 Punched Card Input File

When the Keyboard Monitor is entered from the program level (effective JMP to 07600 or 07605) it checks the batch processing flag and reads a new copy of the BATCH monitor into core if batch processing is in progress. The Command Decoder, however, does NOT perform this operation. Thus, the Command Decoder must not be called unless the BATCH monitor is already in core.

This means that large user programs may be loaded over the BATCH monitor as long as they do not modify the last four locations in the highest memory field; however, once a user core load has overwritten the BATCH monitor, execution must remain at the program level until the Keyboard Monitor has been re-entered and a new copy of the BATCH monitor is read into core. The Command Decoder must not be called after a user program has been loaded over the BATCH monitor.

In general, this restriction applies only to loader programs and only when the loader calls the Command Decoder more than once while building a large core load. Multiple calls to the Command Decoder may be avoided when loading large programs during batch processing if the core load is first built in a stand-alone environment and then saved for subsequent execution under BATCH.

In conjunction with this, note that it is impossible to save the core image of any program that overlays the BATCH monitor under BATCH. After the load operation but before the save is executed, the BATCH monitor will be read back into core, destroying part of the user program. Thus, the Keyboard Monitor SAVE operation will cause part of the BATCH monitor to be saved instead of that part of the user program which originally overlayed the BATCH monitor.

BATCH Demonstration Program

The following listing was produced by running BATCH on a 12K PDP-8/E system containing a disk, DECtape and a line printer. Only the Teletype output is reproduced here, and page breaks were inserted arbitrarily to divide the listing into convenient segments. The same BATCH input file has been processed twice, with two different system configurations.

Notice that the first BATCH processing run begins by listing the BATCH input file, and that the three demonstration programs are listed shortly thereafter.

```
.R BATCH
*SYS:DEMO/U
$JOB OS/8 BATCH DEMO
$MSG BEGIN BY LISTING BATCH INPUT FILE ON TERMINAL:
.R PIP
*TTY:<DEMO,BI
.DATE 3/5/74
$MSG SYSTEM DEVICE ASSIGNED LOGICAL NAME "IN"
.ASSIGN SYS IN
$MSG MOUNT SCRATCH DECTAPE ON UNIT 1
.R PIP
/ZERO DECTAPE DIRECTORY
$MSG WRITE ENABLE UNIT 1
*DTA1:</Z
/LIST SYSTEM DIRECTORY ON LINE PRINTER
*LPT:<IN:/E
/TRANSFER DEMO PROGRAMS TO DECTAPE
*DTA1:DEMO1.PA<DEMO1.PA
*DTA1:DEMO2.FT<DEMO2.FT
*DTA1:DEMO3.FT<DEMO3.FT
/LIST THE FIRST DEMO PROGRAM
*TTY:<IN:DEMO1.PA/T
/LIST THE SECOND DEMO PROGRAM
*TTY:<IN:DEMO2.FT/T
/LIST THE THIRD DEMO PROGRAM
*TTY:<IN:DEMO3.FT/T
/ASSEMBLE DEMO1.PA
.R PAL8
*IN:DEMO1.BN,DEMO1.LS<IN:DEMO1.PA
/PRINT CROSS REFERENCE LISTING
.R CREF
*LPT:<IN:DEMO1.LS
/LOAD ASSEMBLED BINARY INTO CORE
.R ABLSUR
```

```
*DEMO1,BN$
/RUN FIRST DEMO PROGRAM
.START 200
/NOW SAVE CORE IMAGE OF DEMO1.PA, BUT MUST
/RELOAD FIRST, SINCE DEMO1 IS SELF-MODIFYING
.R ABSLDR
*IN:DEMO1,BN$
.SAVE SYS DEMO1 0,200
/RUN DEMO1.SV TO BE SURE THAT IT WAS SAVED CORRECTLY
.RUN SYS DEMO1
/NOW COMPILE FORTRAN MAINLINE PROGRAM
.R FORT
*IN:DEMO2,BN,LPT:<IN:DEMO2,FT$
/COMPILE FORTRAN FUNCTION ROUTINE
.R FORT
*IN:DEMO3,BN,LPT:<IN:DEMO3,FT$
/TRANSFER BOTH BINARY FILES TO DECTAPE
.R PIP
*DTA1:DEMO2,BN<DEMO2,BN/B
*DTA1:DEMO3,BN<DEMO3,BN/B
/LOAD AND EXECUTE FORTRAN PACKAGE
.R LOADER
*DEMO2,BN,DEMO3,BN/G
/RENAME DEMO3,BN FOR FUTURE REFERENCE
.R PIP
*FACT<DEMO3,BN/I
*DEMO3,BN</D
/ADD FORTRAN FUNCTION TO FORTRAN LIBRARY
.R LIBSET
*LIB8,BN/S
*FACT$
/FINALLY, DELETE TEMPORARY FILE "FACT"
.R PIP
*FACT</D
/NOW CLEAN UP DISK AREA
*DEMO1,BN,DEMO1,SV,DEMO2,BN</D
$MSG DEVICE NAMES DEASSIGNED
.DEASSIGN
$END
```

```
$MSG SYSTEM DEVICE ASSIGNED LOGICAL NAME "IN"

$MSG MOUNT SCRATCH DECTAPE ON UNIT 1

$MSG WRITE ENABLE UNIT 1
```

```
*10
IR1,    300
*200
START,  CLA CLL
        TLS
        TAD I IR1
        JMS TYPE
        JMS TEST
        JMP .-4
TYPE,   0
        TSF
        JMP .-1
        TLS
        CLA
        JMP I TYPE
TEST,   0
        TAD IR1
        TAD M335
        SZA CLA
        JMP I TEST
        TSF
        JMP .-1
        JMP 7600
M335,   -335
*301
        215;212;212;241;241;241;305;330;305;303
        325;324;311;317;316;240;303;317;315;320
        314;305;324;305;241;241;241;215;212;212
$
C       FORTRAN DEMONSTRATION PROGRAM
        DIMENSION A(35)
        DO 10 N=2,34,2
        A(N)=FACT(N)
10      WRITE (1,60) N,A(N)
        STOP
60      FORMAT (I3,'! = ',E14.7)
        END


C       FORTRAN FUNCTION TO COMPUTE FACTORIALS
        FUNCTION FACT(N)
        IF (N-34) 1,5,5
1       IF (N) 2,4,2
2       M=N-2
        FACT=N
        DO 3 K=1,M
        C=N-K
3       FACT=FACT*C
        RETURN
4       FACT=1.
        RETURN
5       WRITE (1,6) N
```

```
        FACT=0.
        RETURN
6       FORMAT (I5,'! EXCEEDS CAPACITY OF PROGRAM.')
        END

!!!EXECUTION COMPLETE!!!


!!!EXECUTION COMPLETE!!!

  2! =  0.2000000E+01
  4! =  0.2400000E+02
  6! =  0.7200000E+03
  8! =  0.4032000E+05
 10! =  0.3628800E+07
 12! =  0.4790016E+09
 14! =  0.8717829E+11
 16! =  0.2092279E+14
 18! =  0.6402374E+16
 20! =  0.2432902E+19
 22! =  0.1124001E+22
 24! =  0.6204484E+24
 26! =  0.4032915E+27
 28! =  0.3048883E+30
 30! =  0.2652529E+33
 32! =  0.2631308E+36
   34! EXCEEDS CAPACITY OF PROGRAM.
 34! =  0.0000000E+00
$MSG DEVICE NAMES DEASSIGNED


#END BATCH
```

The next run is initiated via the SUBMIT command.

```
.SUBMIT SYS:<SYS:DEMO/U/T
$JOB OS/8 BATCH DEMO


$MSG BEGIN BY LISTING BATCH INPUT FILE ON TELETYPE:
.R PIP
*TTY:<DEMO.BI

#SPOOL TO FILE BTCHA1
.DATE 8/3/72
$MSG SYSTEM DEVICE ASSIGNED LOGICAL NAME "IN"
.ASSIGN SYS IN
$MSG MOUNT SCRATCH DECTAPE ON UNIT 1
.R PIP
/ZERO DECTAPE DIRECTORY
```

```
$MSG WRITE ENABLE UNIT 1
*DTA1:←/Z
/LIST SYSTEM DIRECTORY ON LINE PRINTER
*LPT:←IN:/E

#SPOOL TO FILE BTCHA2
/TRANSFER DEMO PROGRAMS TO DECTAPE
*DTA1:DEMO1.PA←DEMO1.PA
*DTA1:DEMO2.FT←DEMO2.FT
*DTA1:DEMO3.FT←DEMO3.FT
/LIST FIRST DEMO PROGRAM
*TTY:←IN:DEMO1.PA/T

#SPOOL TO FILE BTCHA3
/LIST SECOND DEMO PROGRAM
*TTY:←IN:DEMO2.FT/T

#SPOOL TO FILE BTCHA4
/LIST THIRD DEMO PROGRAM
*TTY:←IN:DEMO3.FT/T

#SPOOL TO FILE BTCHA5
/ASSEMBLE DEMO1.PA
.R PAL8
*IN:DEMO1.BN,DEMO1.LS←IN:DEMO1.PA
/PRINT CROSS REFERENCE LISTING
.R CREF
*LPT:←IN:DEMO1.LS

#SPOOL TO FILE BTCHA6
/LOAD ASSEMBLED BINARY INTO CORE
.R ABSLDR
*DEMO1.BN$
/RUN FIRST DEMO PROGRAM
.START 200


!!!EXECUTION COMPLETE!!!
/NOW SAVE CORE IMAGE OF DEMO1.PA, BUT MUST
/RELOAD FIRST, SINCE DEMO1 IS SELF-MODIFYING
.R ABSLDR
*IN:DEMO1.BN$
.SAVE SYS DEMO1 0,200
/RUN DEMO1.SV TO ENSURE THAT IT WAS SAVED CORRECTLY
.RUN SYS DEMO1


!!!EXECUTION COMPLETE!!!
/NOW COMPILE FORTRAN MAINLINE PROGRAM
.R FORT
*IN:DEMO2.BN,LPT:←IN:DEMO2.FT$
```

```
#SPOOL TO FILE BTCHA7
/COMPILE FORTRAN FUNCTION ROUTINE
.R FORT
*IN:DEMO3.BN,LPT:←IN:DEMO3.FTS

#SPOOL TO FILE BTCHA8
/TRANSFER BOTH BINARY FILES TO DECTAPE
.R PIP
*DTA1:DEMO2.BN←DEMO2.BN/B
*DTA1:DEMO3.BN←DEMO3.BN/B
/LOAD AND EXECUTE FORTRAN PACKAGE
.R LOADER
*DEMO2.BN,DEMO3.BN/G

  2! =  0.2000000E+01
  4! =  0.2400000E+02
  6! =  0.7200000E+03
  8! =  0.4032000E+05
 10! =  0.3628800E+07
 12! =  0.4790016E+09
 14! =  0.8717829E+11
 16! =  0.2092279E+14
 18! =  0.6402374E+16
 20! =  0.2432902E+19
 22! =  0.1124001E+22
 24! =  0.6204484E+24
 26! =  0.4032915E+27
 28! =  0.3048883E+30
 30! =  0.2652529E+33
 32! =  0.2631308E+36
   34! EXCEEDS CAPACITY OF PROGRAM.
 34! =  0.0000000E+00
/RENAME DEMO3.BN FOR FUTURE REFERENCE
.R PIP
*FACT←DEMO3.BN/I
*DEMO3.BN←/D
/ADD FORTRAN FUNCTION TO FORTRAN LIBRARY
.R LIBSET
*LIB8.BN/S
*FACT$
/FINALLY, DELETE TEMPORARY FILE "FACT"
.R PIP
*FACT←/D
/NOW CLEAN UP DISK AREA
*DEMO1.BN,DEMO1.SV,DEMO2.BN←/D
$MSG DEVICE NAMES DEASSIGNED
.DEASSIGN
$END

#END BATCH
```

Loading and Saving BATCH

The paper tape binary version of OS/8 BATCH may be loaded and saved on the OS/8 system device by typing the following commands in response to the dot generated by the OS/8 monitor:

```
.R ABSLDR
*PTR:(9P)$↑
.SAVE SYS BATCH
```

Once the ALT MODE ($) has been entered, the system will print an uparrow and pause. Load the binary paper tape into the high-speed reader, turn the reader on, and type any character at the keyboard to continue.

Loading and Saving Programs For Use Under BATCH

A program that never uses more than 8K of core can never destroy the BATCH monitor. When this sort of program is loaded from a DECtape system, considerable time is saved through use of the /P option.

The /P option is a new ABSLDR option designed for use under OS/8 BATCH. It causes the 400 bit of the job status word (location 07746) to be set and prevents the Keyboard Monitor from reading a fresh version of the BATCH monitor into core every time the monitor level is re-entered from the program level.

For example, OS/8 PIP never uses more than 8K of core. Thus, the best method of loading PIP would be:

```
.R ABSLDR
*PTR:(89P)=13000$
.
```

The /P option is not really necessary on a disk system, because very little time is required to refresh the BATCH monitor from disk. The /P option should not be used with any program that occupies or modifies core above field 1.

Transferring the System Software From Cassette to the System Device

The following BATCH file can be used to transfer the OS/8 System Software from cassette to the system device.

```
$JOB JOB TO LOAD SYSTEM CASSETTE # 2 TO SYSTEM DEVICE
.R MCPIP
*SYS:CCL.SV<CSA0:CCL.SV
*SYS:DIRECT.SV<CSA0:DIRECT.SV
*SYS:FOTP.SV<CSA0:FOTP.SV
*SYS:PIP.SV<CSA0:PIP.SV
*SYS:LIB8.RL<CSA0:LIB8.RL
*SYS:EDIT.SV<CSA0:EDIT.SV
*SYS:PAL8.SV<CSA0:PAL8.SV
*SYS:CREF.SV<CSA0:CREF.SV
*SYS:BITMAP.SV<CSA0:BITMAP.SV
*SYS:BOOT.SV<CSA0:BOOT.SV
*SYS:CAMP.SV<CSA0:CAMP.SV
*SYS:RK8FMT.SV<CSA0:RK8FMT.SV
*SYS:RKEFMT.SV<CSA0:RKEFMT.SV
$END


$JOB JOB TO LOAD SYSTEM CASSETTE #3 TO SYSTEM DEVICE
.R MCPIP
*SYS:FORT.SV<CSA1:FORT.SV
*SYS:SABR.SV<CSA1:SABR.SV
*SYS:LOADER.SV<CSA1:LOADER.SV
*SYS:SRCCOM.SV<CSA1:SRCCOM.SV
*SYS:EPIC.SV<CSA1:EPIC.SV
*SYS:PIP10.SV<CSA1:PIP10.SV
*SYS:RESORC.SV<CSA1:RESORC.SV
*SYS:DTCOPY.SV<CSA1:DTCOPY.SV
*SYS:TDCOPY.SV<CSA1:TDCOPY.SV
*SYS:TDFRMT.SV<CSA1:TDFRMT.SV
*SYS:DTFRMT.SV<CSA1:DTFRMT.SV
$END


$JOB JOB TO LOAD SYSTEM #4 TO SYSTEM DEVICE
.R MCPIP
*SYS:TC08SY.BN<CSA0:TC08SY.BN
*SYS:TD8ESY.BN<CSA0:TD8ESY.BN
*SYS:LINCSY.BN<CSA0:LINCSY.BN
*SYS:DF32SY.BN<CSA0:DF32SY.BN
*SYS:RF08SY.BN<CSA0:RF08SY.BN
*SYS:RK08SY.BN<CSA0:RK08SY.BN
*SYS:RK8ESY.BN<CSA0:RK8ESY.BN
*SYS:ROMMSY.BN<CSA0:ROMMSY.BN
*SYS:LINCNS.BN<CSA0:LINCNS.BN
*SYS:TC08NS.BN<CSA0:TC08NS.BN
*SYS:RK8ENS.BN<CSA0:RK8ENS.BN
*SYS:PT8E.BN<CSA0:PT8E.BN
*SYS:LSPT.BN<CSA0:LSPT.BN
```

```
*SYS:L645,BN<CSA0:L645,BN
*SYS:ASR33,BN<CSA0:ASR33,BN
*SYS:RK08NS,BN<CSA0:RK08NS,BN
*SYS:CR8E,BN<CSA0:CR8E,BN
*SYS:BAT,BN<CSA0:BAT,BN
*SYS:TD8EA,BN<CSA0:TD8EA,BN
*SYS:TD8EB,BN<CSA0:TD8EB,BN
*SYS:TD8EC,BN<CSA0:TD8EC,BN
*SYS:TD8ED,BN<CSA0:TD8ED,BN
*SYS:VR12,BN<CSA0:VR12,BN
*SYS:RF08NS,BN<CSA0:RF08NS,BN
*SYS:DF32NS,BN<CSA0:DF32NS,BN
*SYS:KL8E,BN<CSA0:KL8E,BN
*SYS:LPSV,BN<CSA0:LPSV,BN
*SYS:TM8E,BN<CSA0:TM8E,BN
*SYS:CSA,BN<CSA0:CSA,BN
*SYS:CSB,BN<CSA0:CSB,BN
*SYS:CSC,BN<CSA0:CSC,BN
*SYS:CSD,BN<CSA0:CSD,BN
*SYS:DIRECT,HL<CSA0:DIRECT,HL
*SYS:BATCH,HL<CSA0:BATCH,HL
*SYS:SABR,HL<CSA0:SABR,HL
*SYS:PIP,HL<CSA0:PIP,HL
*SYS:FOTP,HL<CSA0:FOTP,HL
*SYS:ABSLDR,HL<CSA0:ABSLDR,HL
*SYS:PIP10,HL<CSA0:PIP10,HL
*SYS:BOOT,HL<CSA0:BOOT,HL
*SYS:LOADER,HL<CSA0:LOADER,HL
*SYS:BITMAP,HL<CSA0:BITMAP,HL
*SYS:EDIT,HL<CSA0:EDIT,HL
*SYS:CREF,HL<CSA0:CREF,HL
*SYS:BUILD,HL<CSA0:BUILD,HL
*SYS:PAL8,HL<CSA0:PAL8,HL
*SYS:ODT,HL<CSA0:ODT,HL
*SYS:SRCCOM,HL<CSA0:SRCCOM,HL
*SYS:CCL,HL<CSA0:CCL,HL
*SYS:TECO,HL<CSA0:TECO,HL
*SYS:FORT,HL<CSA0:FORT,HL
*SYS:LOAD,HL<CSA0:LOAD,HL
*SYS:LIBRA,HL<CSA0:LIBRA,HL
*SYS:EPIC,HL<CSA0:EPIC,HL
$END



$JOB JOB TO LOAD SYSTEM CASSETTE # 5 TO SYSTEM DEVICE
.R MCPIP
*SYS:LIB8,RL<CSA1:LIB8,RL
*SYS:GENIOX,RL<CSA1:GENIOX,RL
*SYS:IOH,RL<CSA1:IOH,RL
*SYS:FLOAT,RL<CSA1:FLOAT,RL
*SYS:INTEGR,RL<CSA1:INTEGR,RL
```

```
*SYS:UTILTY.RL<CSA1:UTILTY.RL
*SYS:POWERS.RL<CSA1:POWERS.RL
*SYS:IPOWRS.RL<CSA1:IPOWRS.RL
*SYS:SQRT.RL<CSA1:SQRT.RL
*SYS:TRIG.RL<CSA1:TRIG.RL
*SYS:ATAN.RL<CSA1:ATAN.RL
*SYS:RWTAPE.RL<CSA1:RWTAPE.RL
*SYS:IOPEN.RL<CSA1:IOPEN.RL
*SYS:LIBSET.SV<CSA1:LIBSET.SV
*SYS:KL8E.PA<CSA1:KL8E.PA
$END
```

```
$JOB JOB TO LOAD SYSTEM CASSETTE #6 TO SYSTEM DEVICE
.R MCPIP
*SYS:CCL.PA<CSA0:CCL.PA
$END
```

```
$JOB JOB TO LOAD OS/8 EXTENSION CASSETTE TO SYSTEM DEVICE
.R MCPIP
*SYS:BATCH.SV<CSA1:BATCH.SV
*SYS:BASIC.SV<CSA1:BASIC.SV
*SYS:BCOMP.SV<CSA1:BCOMP.SV
*SYS:BLOAD.SV<CSA1:BLOAD.SV
*SYS:BRTS.SV<CSA1:BRTS.SV
*SYS:BASIC.AF<CSA1:BASIC.AF
*SYS:BASIC.SF<CSA1:BASIC.SF
*SYS:BASIC.FF<CSA1:BASIC.FF
*SYS:BASIC.UF<CSA1:BASIC.UF
*SYS:EAEOVR.BN<CSA1:EAEOVR.BN
*SYS:RESEQ.BA<CSA1:RESEQ.BA
*SYS:TECO.SV<CSA1:TECO.SV
*SYS:MSBAT.SV<CSA1:MSBAT.SV
*SYS:GENIOX.RL<CSA1:GENIOX.RL
$END
```

BITMAP

BITMAP is an OS/8 utility program which constructs a table (map) showing the memory locations used by given binary files.

Hardware and Software Requirements

BITMAP runs on the standard OS/8 configuration and requires the OS/8 software package. BITMAP uses 8K of core to map programs that use up to 16K of core, but requires 12K of core to map programs using more than 16K of core.

Loading Bitmap

Type

```
.R BITMAP
```

to call the BITMAP program from the system device[1]. The system responds by printing an asterisk (*) at the left margin. Type the input line to BITMAP, indicating input devices and file name (if input is from a mass storage device), any options desired, and an output device and file name (if output is to a mass storage device).

The standard input devices for BITMAP are: PTR, DTAn, DSK, and SYS. Any other device which can contain absolute binary files can be used as an input device if a device handler exists. TTY should not be used, as the binary code may appear to the TTY handler as control characters.

BITMAP accepts only absolute binary files. Relocatable and core image files may not be used. If no extension to the input file name is typed, BITMAP assumes the .BN extension. If more than one program is present in a file, only the first program is bitmapped. (This feature allows BITMAP to ignore any noise characters which might be caused by reading over the end of a paper tape.) This feature can be overridden by the /S switch.

Type the RETURN key at the end of an input specification line to signal that more input is to be given on the next line. Use the ALT MODE key as a line terminator when there is no more input; the Command Decoder is not recalled, and control returns to the Keyboard Monitor. The last line typed specifies the output device

[1] System output is underscored throughout this manual.
= carriage return.

on which the bit map is to be produced. Any legal OS/8 output device may be specified. If no output device is specified, output is to the console terminal. For example:

```
.R BITMAP
*DTA1:FILE1,FILE2,FILE3,FILE4
*LPT:<PTR:$↑
```

If an output file is specified without an extension, BITMAP inserts a .MP extension. The preceding lines cause FILE1, FILE2, FILE3, and FILE 4 from DECtape 1 to be considered. Then a file is read from the highspeed paper tape reader. The $ character is printed by the ALT MODE key which indicates a return to the Keyboard Monitor. The resulting bit map combining all the files read is produced on the line printer.

The various options accepted by BITMAP are given below:

Table 2-4 Bitmap Options

Option	Meaning
/R	Reset internal bit map of BITMAP to look as though nothing has been input.
/S	Consider all binary programs in the specified input file(s) (instead of only the first program in each file, which is normally done).
/n	Where n is an integer, forces mapping of all files specified on this input line as if it were initially n field n.
/T	This is used to change the style of output—i.e., put teletype style output on non-teletype or non-teletype style output on teletypes.

Examples of command lines to BITMAP:

```
.R BITMAP
*SYS:PROG.01
*DTA1:MAP<DTA5:PATCH.BN
```

The above commands create a bit map of the combined files PROG.01 on the system device and PATCH.BN on DECtape 5 and stores the output in the file MAP.MP on DECtape 1.

```
.R BITMAP
*LPT:<A,B,C$
```

This example shows a bit map being produced on the line printer for the combination of three binary files (A, B, and C) on the device DSK:.

```
.R BITMAP
*TTY:<PTR:/S$↑
```

One binary tape is read from the high-speed paper tape reader, and a bit map is produced on the terminal combining all binary files on that paper tape.

BITMAP Output

The output of BITMAP is a series of lines, each of which is comprised of a string of digits. Each digit represents a single core location, and can have the value 0, 1, 2 or 3. The value is assigned as follows:

- 0 means that the location was not loaded into.
- 1 means that the location was loaded into once.
- 2 means that the location was loaded into twice.
- 3 means that the location was loaded into three or more times.

Appearance of a 2 or 3 may imply a programming error (e.g., two separate routines are each trying to load values into the same location).

Each line of digits represents 100_8 core locations and lines are blocked in pairs to represent pages. On teletype output, the bit map is bordered by a set of octal coordinates which associates one core location to each digit. For any given entry in the map the corresponding core location can be determined by adding the horizontal and vertical coordinates that lie directly to the left and above the entry.

The following is an example of teletype style output:

```
----

 BITMAP V4   FIELD 0

      0000000011111111222222223333333344444444555555556666666677777777
      0123456701234567012345670123456701234567012345670123456701234567

00000 2222222271111111100000000000000000000000000000000000000000000000
00100

00200 2222222222222222222222222222222222222222222222222222222222222222
00300 2222222222222222221111111111111111111111111111111111111111122221

00400 2222222222222222222222222222222222222222222222222222222222222222
00500 2222222222222222222222222222222222222222222222222222222222222222

00600 1111111111111111111111111111111111111111111111111111111111111111
00700 1111111111111111111111111111111111111111111111111101111111111111

01000
01100

01200
01300

01400
01500

01600
01700

02000
02100
```

The following is an example of non-teletype style output:

```
 BITMAP V4   FIELD 0

00000 22222222 21111111 10000000 00000000 00000000 00000000 00000000 00000000
00100

00200 22222222 22222222 22222222 22222222 22222222 22222222 22222222 22222222
00300 22222222 22222222 22111111 11111111 11111111 11111111 11111111 11122221

00400 22222222 22222222 22222222 22222222 22222222 22222222 22222222 22222222
00500 22222222 22222222 22222222 22222222 22222222 22222222 22222222 22222222

00600 11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111111
00700 11111111 11111111 11111111 11111111 11111111 11111111 11101111 11111111

01000
01100

01200
01300

01400
01500

01600
01700

02000
02100
```

BITMAP Error Messages

After each error message control returns to the Command Decoder and the user can try the procedure again, or reset the program (using the /R option) and try again using different inputs.

Message	Meaning
I/O ERROR FILE #n	An I/O error occurred in input file number n.
BAD INPUT, FILE #n	A physical end of file has been reached before a logical end of file, or extraneous characters have been found in binary file n.
BAD CHECKSUM, FILE #n	File number n of the input file list had a checksum error.
NO INPUT	No binary file was found on the designated device.
ERROR ON OUTPUT DEVICE	Error occurred while writing on output device, i.e., output error on DECtape write.
NO /I	Cannot produce a bitmap of an image file.

Assembly Instructions

Use PAL8 to make BITMAP.BN from BITMAP.PA as follows

```
.R PAL8
*DEV:BITMAP<DEV:BITMAP
```

The listing file shown in parentheses is optional.

Use ABSLDR to make BITMAP.SV from BITMAP.BN on a DECtape file:

```
.R ABSLDR
*DEV:BITMAP=12000/9$
.SAVE DEV BITMAP
```

To load and save the binary paper tape (DEC-S8-OSYSA-C-PB17):

```
.R ABSLDR
*PTR:=12000/9 $↑
.SAVE DEV BITMAP
```

and store in the file MAP.MP on DECtape 1.

BOOT

BOOT is an OS/8 program used to bootstrap from one PDP-8 system to another and to bootstrap from one device to another by typing commands on the keyboard. BOOT can run conveniently from OS/8 and COS 300 and can also run from any other PDP-8 monitor system (e.g., CAPS-8).

To run BOOT from COS 300, see the *COS 300 System Reference Manual,* Chapter 9 (DEC-08-OCOSA-E-D).

To run BOOT from OS/8, type:

```
.R BOOT/dv
```

or

```
.RUN DEV:BOOT/dv
```

where dv is a 2-character mnemonic which must immediately follow a slash. This mnemonic represents the device type and the system to be bootstrapped. Do not attempt to bootstrap onto a device which is not ready or does not exist.

To run BOOT from an OS/8 device with CCL enabled, type:

```
.BOOT/dv
```

If the above form of call is used, BOOT.SV must be present on the system device.

If the following is typed:

```
.R BOOT
```

the system responds with a slash and the user can respond with the dv mnemonic.

If an illegal mnemonic is typed, the system prints:

```
NO
/
```

to allow a new mnemonic to be entered. Type RUBOUT to erase the line, then enter the command correctly.

If the device mnemonic is followed by a period, the program loads the correct bootstrap into core and then halts. Press CONT to branch to the bootstrap.

Table 2-5 lists the legal mnemonics for BOOT.

Table 2-5 Boot Mnemonics

Mnemonic	Device	System or Comments
CA	TA8E cassette	CAPS-8
DK	Any disk (RF08, DF32, RK8E, RK8)	OS/8, COS 300
DL	LINCtape	DIAL-V2, DIAL-MS
DM	RF08 or DF32	Disk Monitor
DT	Any tape (TC08, TD8E, LINCtape)	OS/8, COS 300
LT	LINCtape	OS/8, COS 300
PT	PT8E paper tape	Loads BIN/loader into field 0
RE	RK8E disk	OS/8, COS 300
RF	RF08, DF32 disks	OS/8, COS 300
RK	RK8 disk	OS/8, COS 300
TC	TC08 DECtape	OS/8, COS 300, Disk Monitor, DEC library system, and others
TD	TD8E DECtape	OS/8, COS 300
TY	TC08 DECtape unit 4	Typeset bootstrap
VE		Types BOOT's version number
ZE		Zeroes core (field 0)

More than one device of a particular type (e.g., disk, DECtape) may be present on the OS/8 system. When the DK or DT mnemonic is used, BOOT assumes the following priorities:

Disk	DECtape
1. RF08 or DF32	1. TC08
2. RK8E	2. TD8E
3. RK8	3. LINCtape

BUILD

BUILD is the system generation program for OS/8 which allows the user to:

1. Create an OS/8 monitor system from cassettes or paper tapes.
2. Maintain and update device handlers in an existing OS/8 system.
3. Add device handlers supplied by Digital to a new or existing system.
4. Add user-written device handlers to a new or existing system.

With BUILD, simple keyboard commands are used to manipulate the device handlers which make up the OS/8 peripheral configuration. BUILD allows the user to quickly and easily insert devices which are not standard on the system.

OS/8 Device Handlers

Each OS/8 configuration has certain device handlers that are available within BUILD when the system is supplied by Digital. The handlers supplied with BUILD depend on the distribution media of OS/8 software, i.e., DECtape (LINCtape), cassettes, or paper tape. The device handlers supplied with BUILD are detailed for specific distribution media in Tables 2-6, 2-7, and 2-8. (See Appendix H for more detailed information concerning OS/8 device handlers.)

Device handlers that are included with BUILD must be made active before they can be used by the OS/8 system. The BUILD commands INSERT, REPLACE, and SYSTEM are used to make device handlers active. A maximum of 15 handlers can be made active, including the system device (SYS) and the default mass storage device (DSK).

Inactive devices, even though they are included with the original BUILD, cannot be used on the system until they are made active. For example, several system handlers may be supplied with BUILD, but only one may be marked active.

All other device handlers supported on OS/8 are supplied with every configuration but those not included in the original BUILD must be loaded into BUILD before they can be used. This is accomplished with the BUILD command LOAD. See Table 2-9 for a complete list of the device handlers available with OS/8.

Handlers in BUILD are identified by two names, the first of which is the group name. This is the name assigned to an entire group of handlers all of the same type. For example, the nonsystem TC08 DECtape handler as supplied with a DECtape system, which has four separate handlers internally, has the group name TC.

In addition to the group name, a device also has a permanent device name. This is the name by which OS/8 identifies the physical device. For example, TC08 DECtape unit 3 has the group name TC and the permanent name DTA3.

DECTAPE (LINCTAPE) SYSTEMS

When OS/8 software is supplied on DECtape or LINCtape, the device handlers shown in Table 2-6 are included in BUILD. The handlers in Table 2-6 can be made active with the INSERT, SYSTEM, or REPLACE commands.

Table 2-6 Standard DECtape System Device Handlers

Handler	Group Name	Permanent Name(s)
TC08 DECtape system handler	TC08	SYS
TC08 nonsystem DECtape drives 0-3	TC	DTA0-DTA3
12K TD8E DECtape system handler and drives 0 and 1	TD8E	SYS, DTA0, DTA1
8K ROM TD8E DECtape system handler and drives 0 and 1	ROM	SYS, DTA0, DTA1
TD8E nonsystem DECtape drives 0 and 1	TD8A	DTA0, DTA1
TD8E nonsystem DECtape drives 2 and 3	TD8B	DTA2, DTA3
RK8E disk system handler	RK8E	SYS, RKB0
RK8E disk nonsystem handler	RK05	RKA0, RKA1, RKB0, RKB1
RK8 disk system handler	RK8	SYS, RKA1
RK8 disk nonsystem handler	RK01	RKA0, RKA1
LINCtape system handler	LINC	SYS
LINCtape nonsystem handler	LNC	LTA0-LTA3
RF08 disk system handler	RF08	SYS
Console terminal (2-page handler)	KL8E	TTY
High-speed I/O simulated on ASR-33 Teletype	KS33	PTR, PTP
High-speed reader/punch	PT8E	PTR, PTP
LP08, LS8E, LV8E line printers	LPSV	LPT
TA8E cassette drives 0 and 1	TA8A	CSA0, CSA1
PDP-12 scope	VR12	TV

Other device handlers available with OS/8 but not included in BUILD are listed in Table 2-9. The handlers supplied with a DECtape or LINCtape system are on the System Tape #2 (DEC-S8-OSYSB-A-UC2). To include extra handlers in BUILD, mount this tape and use the LOAD command.

CASSETTE SYSTEMS

When OS/8 software is supplied on cassettes, the device handlers shown in Table 2-7 are included in BUILD. These handlers can be made active with the INSERT, REPLACE, or SYSTEM commands.

Table 2-7 Standard Cassette System Device Handlers

Handler	Group Name	Permanent Name(s)
RK8E disk system handler	RK8E	SYS, RKB0
RK8E disk system handler	RK8	SYS, RKA1
RF08 disk system handler	RF08	SYS
DF32 disk system handler	DF32	SYS
Console terminal (2-page handler)	KL8E	TTY
High-speed I/O simulated on ASR-33 Teletype	KS33	PTR, PTP
High-speed reader/punch	PT8E	PTR, PTP
TA8E cassette drives 0 and 1	TA8A	CSA0, CSA1
LP08, LS8E, LV8E line printers	LPSV	LPT

Other handlers supplied with OS/8 but not included in BUILD are listed in Table 2-9. These handlers are present on the system cassette DEC-S8-OSYSB-A-TC4. To include extra handlers in BUILD, build an OS/8 system, use MCPIP to move specific device handlers onto the system device, then use the BUILD command LOAD.

PAPER TAPE SYSTEMS

When OS/8 software is supplied on paper tape, the device handlers shown in Table 2-8 are included in BUILD. These handlers can be made active with the INSERT, REPLACE, or SYSTEM commands.

Table 2-8 Standard Paper Tape System Device Handlers

Handler	Group Name	Permanent Name(s)
RK8E disk system handler	RK8E	SYS, RKB0
RK8 disk system handler	RK8	SYS, RKA1
RF08 disk system handler	RF08	SYS
DF32 disk system handler	DF32	SYS
Console terminal (2-page handler)	KL8E	TTY
High-speed I/O simulated on ASR-33 Teletype	KS33	PTR, PTP
High-speed reader/punch	PT8E	PTR, PTP
TA8E cassette drives 0 and 1	TA8A	CSA0, CSA1
LP08, LS8E, LV8E line printers	LPSV	LPT

Other handlers supplied with OS/8 but not included in BUILD are provided on two binary paper tapes. DEC-S8-OSYSB-A-PB2 contains the file-structured handlers. DEC-S8-OSYSB-A-PB3 contains character-oriented handlers. These tapes contain handlers which can be loaded into core using the BUILD command LOAD.

The BUILD device handler tapes are composed of separate segments, with a short length of leader/trailer code between them. (All of these handlers are in the special format described in BUILD Device Handler Format in this section.) Table 2-9 contains a list of the handlers that are included on the tapes. The handlers are listed in the order that they appear on the tapes. The TC08 handler is the first segment on handler tape #1 and the KL8E terminal handler is the first segment on handler tape #2. It is suggested that either the segments be labeled or separated for easier use.

To utilize a binary handler file, place the desired segment into the paper tape reader. Use the BUILD command LOAD to load that segment as follows:

```
$LOAD PTR[:]
↑
$
```

Type a colon (:) after the device name if BUILD was loaded from an OS/8 system device. The ↑ allows time to place the tape in the reader. Type any keyboard character to load the tape. When the $ reappears, the handler has been loaded into BUILD's table. Type the BUILD command PRINT to verify that the handler has been loaded.

Table 2-9 OS/8 Device Handlers

Handler	Group Name	Permanent Name(s)	File Name on DECtape LINCtape or Cassette
TC08 DECtape system handler	TC08	SYS, DTA0	TC08SY .BN
12K TD8E DECtape system handler	TD8E	SYS, DTA0, DTA1	TD8ESY .BN
8K ROM TD8E DECtape system handler	ROM	SYS, DTA0, DTA1	ROMMSY .BN
LINCtape system handler	LINC	SYS, LTA0	LINCSY .BN
RK8E disk system handler	RK8E	SYS, RKA0, RKB0	RK8ESY .BN
RK8 disk system handler	RK8	SYS, RKA0, RKA1	RK08SY .BN
RF08 disk system handler	RF08	SYS	RF08SY .BN
DF32 disk system handler	DF32	SYS	DF32SY .BN
TD8E DECtape drives 0 and 1	TD8A	DTA0, DTA1	TD8EA .BN
TD8E DECtape drives 2 and 3	TD8B	DTA2, DTA3	TD8EB .BN
TD8E DECtape drives 4 and 5	TD8C	DTA4, DTA5	TD8EC .BN
TD8E DECtape drives 6 and 7	TD8D	DTA6, DTA7	TD8ED .BN
TC08 DECtape drives 0-7	TC	DTA0-DTA7	TC08NS .BN
LINCtape drives 0-7	LNC	LTA0-LTA7	LINCNS .BN
RK8E disk nonsystem handler	RK05	RKA0-3, RKB0-3	RK8ENS .BN
RK8 disk nonsystem handler	RK01	RKA0-RKA3	RK08NS .BN
RF08 disk nonsystem handler	RF	RF, NULL	RF08NS .BN
DF32 disk nonsystem handler	DF	DF	DF32NS .BN
Console terminal (2-page handler)	KL8E	TTY	KL8E .BN
Console terminal (1-page handler)	AS33	TTY	ASR33 .BN
High-speed I/O simulated on ASR-33 Teletype	KS33	PTR, PTP	LSPT .BN
High-speed reader/punch	PT8E	PTR, PTP	PT8E .BN
LP08, LS8E, LV8E line printers	LPSV	LPT	LPSV .BN
Anelex 645 line printer	L645	LPT	L645 .BN
Card reader	CR8E	CDR	CR8E .BN
BATCH handler	BAT	BAT	BAT .BN
PDP-12 scope	VR12	TV	VR12 .BN
TU10 magnetic tape drives 0-7	TM8E	MTA0-MTA7	TM8E .BN
TA8E cassette drives 0 and 1	TA8A	CSA0, CSA1	CSA .BN
TA8E cassette drives 2 and 3	TA8B	CSA2, CSA3	CSB .BN
TA8E cassette drives 4 and 5	TA8C	CSA4, CSA5	CSC .BN
TA8E cassette drives 6 and 7	TA8D	CSA6, CSA7	CSD .BN

Calling and Using BUILD

BUILD is distributed as both a binary paper tape or cassette and as a core image file (BUILD.SV) on the system DECtape or LINCtape. The binary BUILD file should be loaded and saved on the

system device when the initial system is built (see Getting On Line with OS/8 in Chapter 1). To use the BUILD.SV file on the system device, type the following command in response to the dot printed by the OS/8 Keyboard Monitor:

```
RUN SYS BUILD
```

NOTE

It is important that the user specify the RUN or RU command, rather than the R command, when loading BUILD into core. This will allow the use of the SAVE command without specifying SAVE arguments.

BUILD responds by printing a $, signaling that it is ready to accept commands.

BUILD uses a keyboard monitor similar to that contained in the OS/8 system. Text is input from the terminal and interpreted by BUILD. Table 2-10 lists the special characters that are available for editing.

Table 2-10 BUILD Editing Characters

Character	Function
ALT MODE key	Terminate command; begin command execution. No carriage return/line feed is generated.
CARRIAGE RETURN	Terminate command; begin command execution. Also generate carriage return/line feed combination.
CTRL/C	Terminate command; return immediately to the OS/8 Keyboard Monitor.
CTRL/O	Terminate printing; return control to BUILD.
CTRL/U	Ignore line; the line may be typed again.
LINE FEED key	Examine contents of the command line.
RUBOUT key	Delete the last typed character from the command.

The standard characters permitted in a BUILD command line are:

A-Z, 0-9, SPACE, PERIOD, =, COMMA, COLON, HYPHEN

Typing any other character causes the error message:

```
SYNTAX ERROR
```

BUILD Commands

The commands available in BUILD are:

ALTER
BOOT
BUILD
CORE
CTL
DCB
DELETE
DSK
EXAMINE
INSERT
LOAD
NAME
PRINT
QLIST
REPLACE
SYSTEM
UNLOAD
VERSION

The general format of the command string is:

$command args

where command represents a legal command from the list and args represents a file name, device, group name, or other argument associated with the command. The command can be typed in full or abbreviated to the first two characters. For example:

$PRINT
$PR

are the same. If the user attempts to issue an illegal command. BUILD replies by printing the illegal command preceded by a ?. Thus the illegal command ERASE would appear:

```
$ERASE
?ERASE
$
```

THE HYPHEN CONSTRUCTION

Certain BUILD commands (DELETE, INSERT, REPLACE) allow the use of the hyphen construction to specify more than one permanent name. These permanent names must be four characters long and must differ only in the last character. Permanent names which meet this restriction can be inserted with the hyphen construction so long as the last characters form a sequence of consecutive ASCII characters.

For example, if the user wishes to delete DECtape handlers DTA0, DTA1, DTA2, and DTA3, he can type:

```
$DELETE DTA0,DTA1,DTA2,DTA3
```

or he can use the hyphen construction and type:

```
$DELETE DTA0-3
```

PRINT

Syntax: $PRINT or $PR

Function: Prints the detailed list of the BUILD devices tables. The following example shows five handlers.

```
RF08:     SYS
RK8E:    *SYS    *RKB0
KL8E:    *TTY
PT8E:     PTR    *PTP
LPSV:     LPT
```

Group names are printed first in each line followed by a colon. Following the group name are the list of permanent names available with each group. If one of the permanent names in a group is SYS, then this handler can be a system handler. An OS/8 system must have just one system handler. Some system handlers have other handlers coresident with them.

Any handler that is active is marked with an asterisk to the left of its permanent name (RKB0, TTY, PTP in the printout), and the devices will be included in the new OS/8 system. (i.e., these handlers were inserted with the INSERT, SYS, or REPLACE

commands. Other commands are available for removing, loading, and deactivating handlers.) The preceding printout indicates that RK8E is the system device. The handler RK8E:RKB0 is also marked as being active.

After printing the list of available handlers, the PRINT command might also print some additional information. If, for example, the user specified RK8E:RKB0 with the DSK command, the following is printed:

```
DSK=RK8E:RKB0
```

If the user specified that core is to be restricted to 12K with the CORE command, the message:

```
CORE=2
```

is printed, indicating that field 2 is to be the highest core field available to the OS/8 system.

QLIST

Syntax: $QLIST or $QL

Function: List the active permanent names on the system. No * is printed and the system device is the only group name printed. For example:

```
$QLIST
PTR DTA3 RK08:SYS LPT DTA4
```

LOAD

Syntax: $LOAD activename or $LOAD dev:filename

Function: LOAD is used to load a new device handler into BUILD. This handler can be one supplied by Digital or one written by the user. See the *OS/8 Software Support Manual* (DEC-S8-OSSMB-A-D) for instructions on writing device handlers. This handler is input into BUILD as a binary file or image.

If BUILD is being run stand-alone, e.g., to create an initial OS/8 system, the LOAD command has the form:

$LOAD activename

where activename is the permanent name of an input device handler that has been made active with the INSERT, REPLACE, or SYSTEM command. It must be a handler for a non-file structured device. For example, to load a new handler from a binary paper tape with the PTR handler already in BUILD, type:

```
$LOAD PTR
```

IF BUILD is being run under control of OS/8, the LOAD command has the form:

$LOAD dev:filename

where dev is an input device handler that exists in the current OS/8 system. (These are not the same as the handlers which are marked active by BUILD.) If no dev: is specified, DSK: is assumed.

If dev: is non-file structured (i.e., paper tape), the filename may be omitted. The filename has the form:

name. extension

Filename is the binary file of the new handler to be loaded. The default extension is .BN. If no extension is used, the dot (.) may be omitted.

Example:

$LOAD DTA3:HANDLR.03	A file named HANDLR, with an extension of 03 is loaded from DTA3.

Several files to be loaded may be specified on one line, separated by commas. A device must be specified for each file or DSK will be assumed. If multiple files are specified, each file must contain a separate handler to be loaded. For example:

```
$LOAD DTA3:FILE1,DTA5:FILE2
```

Once the LOAD command has been successfully issued, the new device handlers are available for further manipulation. The new handlers will appear in the PRINT output, but will not be marked as active.

INSERT

Syntax: $INSERT gname, pname

Function: After a LOAD command has made a handler or group of handlers available for insertion in the OS/8 system, the INSERT command is used to make particular entry points active. The INSERT command uses two arguments; gname and pname. Gname is the group name of the handler, for example, the gname for TC08 DECtape is TC. Pname is the permanent name by which the device is currently known to BUILD. See Table 2-9 for a complete list of permanent device names. TC08 DECtape thus has the group name TC and the permanent names DTA0-DTA7.
Examples:

```
$IN KL8E,TTY
$IN TC08,SYS
```

If no permanent name is specified (and no :), the first name in the device group is assumed. For example:

```
$INSERT TC
```

would assign DTA0 as the permanent name.

Several handlers in the same group can be inserted in the same command by separating the permanent names with commas. For example:

```
$IN TC,DTA0,DTA3,DTA7
```

If several permanent names (each four characters long) differ only in the last character, they can be simultaneously inserted with the hyphen construction so long as the last characters form a sequence of consecutive ASCII characters.
Example:

```
$INSERT TC,DTA2-5
```

is the same as

```
$INSERT TC,DTA2,DTA3,DTA4,DTA5
```

and

```
$INSERT RK01,RKA0-2
```

is the same as

```
$INSERT RK01,RKA0,RKA1,RKA2
```

If the permanent name specified is not part of the group name specified or if the group name does not exist, the following message is printed:

name NOT FOUND

If disk is the device being inserted, the group name can be followed by a construction of the form:

pname=n

Where n is a digit in the range 1 to 7 and represents the number of platters available. This option is used for the RF08 and DF32 disks. For example:

```
$IN RF,RF=2
```

If no such option is specified, =1 is assumed. If n is too large for the device specified, the following message is printed:

```
?PLAT
```

DELETE

Syntax: $DELETE aname

Function: DELETE takes a device which is currently marked as active and makes it inactive. (Devices which are active are marked with an * in the PRINT command output and are printed by the QLIST command.)

The argument for DELETE is the permanent name of the device. The current permanent name can be obtained from the PRINT or QLIST output. The major function of DELETE is make device slots available to BUILD.

For example, assume that the QLIST command output is:

```
DTA0 DTA1 RK8E:SYS RKB0 TTY LPT CSA0 CSA1 CSA2 CSA3
```

If the following command is issued to BUILD:

```
$DELETE CSA0,CSA1,CSA2,CSA3
```

CSA0, CSA1, CSA2, and CSA3 will no longer be permanent devices and the slots used by the TA8A and TA8B device groups will be made available to BUILD. The QLIST output after the above command will be:

```
DTA0 DTA1 RK8E:SYS RKB0 TTY LPT
```

Note that, as previously explained, the hyphen construction can be used in DELETE to remove a sequence of devices. Thus the command to make the cassette handlers inactive could also be typed as follows:

```
$DELETE CSA0-3
```

REPLACE

Syntax: $REPLACE pname=gname, pname2

Function: REPLACE combines the functions of DELETE and INSERT to provide a means of deleting one device and activating another in a single step. The arguments for REPLACE are:

pname	The permanent name of the device to be deleted. (Same as are argument of the DELETE command.)
gname, pname2	The group name and permanent name of the particular device to be inserted into the system (see INSERT for more details).

Example: Assume the PRINT output is:

```
PT8E: *PTP     *PTR
CR8E: *CDR
RK05:  RKA0     RKB0     RKA1     RKB1
```

REPLACE can be used to delete the card reader (CDR) and insert the RK05 group handler for RKA0:

```
$REPLACE CDR=RK05,RKA0
```

The output of PRINT after this REPLACE is:

```
PTR:  *PTP    *PTR
CR8E:   CDR
RK05: *RKA0    RKB0   RKA1   RKB1
```

The hyphen construction can be used with REPLACE to delete and insert more than one device handler. For example, assume that LINCtape handlers LTA0, LTA1, LTA2, and LTA5 are to be replaced with DECtape handlers DTA0, DTA1, DTA2, and DTA5. This replacement could be accomplished with the command:

```
$REPLACE LTA0-2,LTA5=TC,DTA0-2,DTA5
```

UNLOAD

Syntax: $UNLOAD gname, or $UNLOAD gname, pname

Function: UNLOAD is used to physically delete a handler group (gname) or a permanent name (pname) from the BUILD system. (This differs from DELETE, which does not physically eliminate a device.) UNLOAD is primarily used when the NO ROOM error occurs during a LOAD command.

For example, assume that the entire group of LINCtape handlers is to be removed. The command is typed as:

```
$UNLOAD LNC
```

This command unloads the LINCtape handler LNC and all permanent names (LTA0, LTA1, LTA2, LTA3, etc.) associated with it.

To remove a particular permanent name from BUILD, e.g., DTA3, type:

```
$UNLOAD TC,DTA3
```

This command does not unload the handler, just the entry point name.

To remove several permanent names, but not the entire group, the UNLOAD command is used with commas separating the permanent names. For example:

```
$UNLOAD TC,DTA0,DTA2
```

The hyphen construction *cannot* be used with the UNLOAD command.

NAME

Syntax: $NAME pname=pname2

Function: The NAME command allows the user to alter the device name which will be used by OS/8. The first argument, pname, must be the current name of a device marked active in the PRINT output. Pname2 is the name the user wishes to call this device. Only 4-character device names may be used in the NAME command. If longer names are entered, all characters beyond the first four are ignored. After the NAME command is used, pname2 is the current permanent name; pname is unknown to BUILD.

Example: Assume that the PRINT output is:

```
TC   : *DTA0   *DTA1    DTA2    DTA3
RK8E: *SYS    *RKB0
```

To change the paper tape reader so that it is recognized by the permanent name READ, the following command is used:

```
$NAME PTR=READ
```

The output from PRINT would then be:

```
TC   : *DTA0   *DTA1    DTA2    DTA3
RK8E: *SYS    *RKB0
KK8E: *TTY
PT8E: *PTP    *READ
```

If the permanent name specified as pname is not a currently active device, the message:

pname NOT FOUND

is printed. If this message appears, check the PRINT output to determine the correct permanent name.

ALTER

Syntax: $ALTER gname, loc=newvalue

Function: The ALTER command allows the user to change locations in device handlers. The arguments are:

gname	Group name of the handler.
loc	Relative octal location to be altered. If the handler is a 1-page handler, loc must be an octal number in the range 0-0177. If it is a 2-page handler, loc must be an octal number in the range 0-0377.
newvalue	An octal number specifying the new contents of the location specified by loc. If no =newvalue is entered, BUILD prints the old value of loc followed by a slash. Newvalue can then be entered or a carriage return can be typed to retain the old value.

EXAMINE

Syntax: $EXAMINE gname, loc

Function: EXAMINE allows the user to examine, but not modify, a location within a device handler. See the ALTER command.

DSK

Syntax: $DSK=gname, pname or $DSK=aname

Function: The DSK command is used to specify which device is to be designated as DSK, the default storage device for OS/8. If the first form of the command is used, i.e.,

$DSK=gname,pname

the gname is the group name of the device and pname is the permanent name. For example:

```
$DSK=TC08:DTA0
```

assigns DTA0 as the device called DSK.

When the DSK command is issued, the permanent name need not have been entered. However, the permanent name must be entered, via an INSERT, REPLACE, or SYSTEM command before the BOOT command is issued.

If the second form of the command is used, i.e.,

$DSK=aname

aname must be a permanent name marked as active by BUILD. For example, the following command specifies the already active device RKA0 as the default device DSK:

```
$DSK=RKA0
```

If no DSK command is entered, or if the command is issued without an argument, i.e.,

```
$DSK=
```

or

```
$DSK
```

BUILD specifies SYS as DSK when a BOOT command is issued:

CORE

Syntax: $CORE n

Function: The CORE command is used to specify the highest core field available to the OS/8 system being built. The n is an octal number in the range 0 to 7. If n is 0 or omitted, or if the CORE command is not used, the system built will use all of the available core. If n specifies more core than is physically available, the following message is printed:

```
?CORE
```

The following table indicates the value of n for the available core sizes:

n	core
0	all available core
1	8K
2	12K
3	16K
4	20K
5	24K
6	28K
7	32K

For example, a system which is to use only 24K of a 32K system would have the following CORE command:

```
$CORE 5
```

DCB

Syntax: $DCB aname or $DCBaname=newvalue

Function: The DCB command allows examination or modification of the DCB word associated with a permanent name. (See the section on BUILD Device Handler Formats for information on DCB words.

The DCB word is the first word after the permanent name in a description (from the handler header information words). Aname must be the permanent name of a device currently marked as active in the PRINT output.

Example:

```
$DCB DTA4=6160
```

changes the DCB of DTA4 so that this handler becomes a read-only device. This command could also be typed as:

```
$DCB DTA4
4160/6160
```

CTL

Syntax: $CTL aname=loc

Function: The CTL command allows modification of the control

word which is the word after the DCB word in the handler header block. For example:

```
$CTL LTA3=24
```

changes the entry point of the LTA3 handler to be relative location 24.

VERSION

Syntax: $VERSION or $VE

Function: The VERSION command prints the version number of BUILD on the terminal.

SYSTEM

Syntax: $SYSTEM sname=n

Function: The SYSTEM command specifies devices which are system handlers or coresident with system handlers. The number n reflects the number of platters included in the system device (valid only for multiple platter RF08 and DF32 disks). The available system handlers and their associated values for n are listed in Table 1-6). The argument sname must be one of the legal device system names. If it is not, BUILD prints:

```
?SYS
```

thus requesting a new system specification.

Action is not taken on the SYSTEM command until the BOOTSTRAP command is given, so the user may respecify a device with SYS. The system device used is the last one issued prior to the BOOT command. Specifying a new system device is not always necessary. For example, if the user wishes to insert new peripheral handlers, then this command is not needed. If it is not issued, the OS/8 system which is resident is not affected beyond altering the device tables.

The SYSTEM command is included only for compatibility with older versions of BUILD. The system device can be specified with the INSERT command. For example, the command:

```
$SYS RF08=2
```

is the same as the command:

```
$INSERT RF08,SYS=2
```

If the device specified in the SYS command is not the current system device, the user will have an opportunity to have a zero directory placed on his new system device. If the system device is the same as the current system device, no new directory will be written.

BUILD

Syntax: $BUILD or $BU

Function: The BUILD command is used only when building an initial OS/8 system from cassettes or paper tape. When the BUILD command is typed, BUILD prints:

```
LOAD OS/8:
```

to which the user must respond with the device that contains the new OS/8 monitor, e.g.,

```
LOAD OS/8: CSA0
```

BUILD then loads and writes the various parts of OS/8 onto the system device. After writing OS/8, BUILD prints:

```
LOAD CD:
```

to which the user responds with the appropriate device, or types carriage return to specify that the device is the same as that specified in the LOAD OS/8: message. BUILD loads the Command Decoder and writes it onto the system device.

The BUILD command must *not* be used at any time other than while building an initial OS/8 system. When this command is typed, OS/8 assumes that the user is building a new OS/8 system and automatically zeroes the system device directory. See Getting On Line With OS/8 in Chapter 1 for instructions on building an initial system.

BOOTSTRAP

Syntax: $BOOTSTRAP or $BO

Function: BOOTSTRAP is the command which finally implements all the changes that have been made using BUILD. BOOT rewrites all relevant Monitor tables and device handlers to reflect the updated system status. The devices which BUILD had marked active now become device handlers in the system.

When a BOOTSTRAP command is typed, the system device must have been explicitly specified with either the SYSTEM or INSERT command. If no SYS is specified, the message:

```
SYS NOT FOUND
```

is printed.

If the system device specified is different from the current system device, BUILD copies the system from the current system device to the new system device. After the copy is complete, BUILD asks:

```
WRITE ZERO DIRECT?
```

to determine whether a new (zero) directory is to be written on the new system device. If the reply is YES, a zero directory will be placed on the device. Any other reply causes the old directory to be retained.

NOTE

Care should be exercised if the old directory is to be retained. The directory must be that of an OS/8 system device.

After this question has been answered, BUILD updates the system and prints:

```
SYS BUILT
.
```

Control returns to the Keyboard Monitor. When the BOOTSTRAP command has performed its functions and the Keyboard Monitor is once again active, it is a good idea to save the copy of BUILD just used. In this way, an image of the current system status is preserved, and the saved copy of BUILD can be used

again. When it is used again, the devices which were initially marked active are still marked active. To save BUILD, type:

```
.SAVE SYS BUILD
```

in response to the dot printed by the Keyboard Monitor. This assumes that the user originally loaded BUILD into core with a RU or RUN command.

BUILD Error Messages

The following is a list of error messages which may appear when using BUILD. These messages are usually indicative of a syntax or user error.

Table 2-11 BUILD Error Messages

Message	Explanation
?BAD ARG	No device name was included in the LOAD command.
?BAD INPUT	An error was detected in the binary file; it is not a proper input for the LOAD command.
?BAD LOAD	An attempt was made to load a binary handler that is not in the correct format.
?BAD ORIGIN	The origin in a binary file is not in the range 200-577.
?CORE	A CORE command specified more memory than is physically available, or the BOOT command was issued on an 8K system with a 2-page system handler active. Two page system handlers require at least 12K of core to be present on the OS/8 system.
?DSK	The device specified in a DSK command is not a file-structured device.
?HANDLERS	More than 15 handlers, including SYS and DSK, were active when a BOOT command was issued.
I/O ERR	An error occurred while reading from an input device during a LOAD command.
?NAME	A device or file name was not designated in a command that requires one to be present.
NO ROOM	Too many device handlers were present on the system when a LOAD or BUILD command was typed. The UNLOAD command must be used to remove a handler before another can be loaded.

Table 2-11 BUILD Error Messages (Cont.)

Message	Explanation
name NOT FOUND	The device or file name designated in the command was not found.
?PLAT	The =n in a SYS command is too large for the device specified, e.g., RF08=5.
?SYNTAX	An illegal character was typed in a BUILD command line. The line must be retyped.
?SYS	This message appears when one of the following conditions exists. a. A permanent name in a SYS command was not a system handler or coresident with one. b. A BOOT command was issued when two or more system handlers were active. c. A BOOT command was issued when an active handler which must be coresident with a SYS handler did not have the system handler active.
SYS ERR	An I/O error occurred with a system handler. The computer halts. Press CONT to retry or restart the BUILD procedure from the beginning. Do not assume that a valid OS/8 system remains in core.
SYS NOT FOUND	No active handler with the name SYS was present when a BOOTSTRAP command was issued.

BUILD Device Handler Format

The BUILD command LOAD is used to load device handlers not provided by BUILD into core where they can be inserted into the OS/8 system. The format of the input to LOAD is a binary file containing the handler, as well as a header block which contains information pertaining to the devices included in that file. The user should code the handler in PAL8 machine language according to the following format.

The structure of the source for a BUILD device handler is:

```
*0
HEADER BLOCK
*200
BODY OF DEVICE
HANDLER
```

The origins at 0 and 200 are vital to BUILD. The *0 is an important part of the header block and, if it is omitted, no load is done. The *200 is also necessary for the load. If the handler contains an origin outside the range 200-577, an error message is generated and the load is aborted.

HEADER BLOCK

The header block contains the following information:

Word 1:	-X, where X is the number of separate handlers contained in this file. Thus a handler for TC08 has the first word equal to −10(octal).
Words 2-9:	Descriptor block for the first handler in the group.
Words 10-17:	Description block for second handler in the group.
. . .	. . .
	Descriptor block for second handler in the group. If the handler is a system handler, this is followed by the length of the bootstrap and the bootstrap itself.

Thus, each handler in the group must have an 8-word block describing its characteristics. If more than 12 handlers are in a group, an error is generated during the LOAD.

DESCRIPTOR BLOCK

Each 8-word descriptor block contains the following information.

Words 1,2:	Device type name. This name is the group name, or type, of all the handlers in this group and is usually designated by the DEVICE pseudo-op.
	Example: DEVICE RK8

Words 3,4: OS/8 device name. This is the name (permanent name) by which the particular device will be recognized in the OS/8 system to be configured. It can be altered by the NAME command.

Example: DEVICE RKA0

Word 5: Device Control Block. This word reflects the type of device, in accordance with Table 2-12. Bits 9-11 specify the maximum number of platters on the device (0=1).

Example: 4050

Word 6: Entry point word. This word must contain the entry point offset in bits 5-11 (see ENTRY POINT OFFSET). Bit 0 should be a 1 if the handler is a 2-page handler. Bit 1 should be a 1 if the entry point is SYS. Bit 2 should be a 1 if the entry point is coresident with SYS.

Example: 0020

Word 7: Must be **0.**

Word 8: Must be 0, except for a system handler which uses it to specify the block length of the device.

As an example, consider the handler for the nonsystem RK05 handlers. This file contains four separate handlers; the source code would appear as follows:

```
        *0
        -4                      /4 DEVICES

DEVICE RK05; DEVICE RKA0; 4050; 0020; ZBLOCK 2
DEVICE RK05; DEVICE RKB0; 4050; 0021; ZBLOCK 2
DEVICE RK05; DEVICE RKA1; 4050; 0022; ZBLOCK 2
DEVICE RK05; DEVICE RKB1; 4050; 0023; ZBLOCK 2

        *200

            (HANDLER BODY)
```

The device type of the group is RK05 (Words 1-2). The permanent device names are RKA0, RKB0, RKA1, RKB1. Since each device is RK05, the device control block (DCB) word for each is identical.

The entry point word indicates where the entry point for that particular device occurs relative to the top of the page. Thus, in the above example, RKA0 enters at the 20th location from the top of the page, RKB0 at the 21st, etc.

It is vital that this information be accurate. If errors are made in this data, unpredictable results occur when the system is generated.

BREAKDOWN OF DCB WORD

The DCB word for a device provides specific information which is used in the OS/8 Monitor. Its structure is detailed in Table 2-12.

Table 2-12 DCB Word

Bit	Meaning
0	1 if file-structured device
1	1 if read-only device (e.g., PTR)
2	1 if write-only device (e.g., LPT)
	Device Type
3-8	00 = console terminal 01 = high-speed paper tape reader 02 = high-speed paper tape punch 03 = card reader 04 = line printer 05 = RK8 Disk 06 = RF08 (1 platter) 07 = RF08 (2 platter) 10 = RF08 (3 platter) 11 = RF08 (4 platter) 12 = DF32 (1 platter) 13 = DF32 (2 platter) 14 = DF32 (3 platter) 15 = DF32 (4 platter) 16 = TC08 DECtape 17 = LINCtape 20 = TM8E magnetic tape 21 = TD8E DECtape 22 = BAT—BATCH handler 23 = RK8E disk

Table 2-12 DCB Word (Cont.)

Bit	Meaning
	24 = NULL—NULL handler 25-26 = Unused 27 = TA8E cassettes 30 = PDP-12 scope 31-37 = Unused by Digital 40-77 = Reserved for user-written handlers
9-11	Used only by OS/8 Monitor

Whenever a device is to be inserted into OS/8, this structure must be followed to obtain correct results.

ENTRY POINT OFFSET

Word 6 of each device descriptor block specifies the relative entry point of that particular handler. Devices supplied by Digital have a fixed set of entry points, described below.

Care should be used when coding new device handlers for insertion into the system. The entry point offset for the new handler must not be the same as that for any other file-structured device in the system. For example, OS/8 currently uses relative entry points 7-23 for file structured devices. No new handler should have entry points at 7 to 23 of the page. If this occurs, the system may perform incorrectly.

Current file device and entry point offsets are listed below:

Device	Entry Relative to Top of Page
TC08 DECtape	10-17
TD8E DECtape	10-17
LINCtape	10-17
System device	7
RK8 disk	20-23

Thus, the user-coded file devices should use entry points other than 7-23.

If a new file-structured user device is added to the system, it will be necessary to alter the device length table in PIP to permit zeroing of the device directory. To do this, ODT is used as follows:

```
.GET SYS PIP
.ODT
136nn/0000 xxxx
↑C                    user types CTRL/C
.SAVE SYS PIP
```

The nn represents the 2-digit device indicated in Table 2-12. The xxxx is the negative of the last block number on the device. Both nn and xxxx are octal numbers.

For example, if the new device is assigned a code of 40 (currently the first unused entry), and the last OS/8 block on the device was block 1000, PIP would be changed as follows:

```
.GET SYS PIP
.ODT
13640/0000 7000
↑C
.SAVE SYS PIP
```

CAMP (CASSETTE AND MAGNETIC TAPE POSITIONER)

The CAMP (Cassette and Magnetic Tape Positioner) program is used to position cassettes, magnetic tapes, and certain other devices. To call CAMP from the system device, type:

```
R CAMP
```

in response to the dot printed by the Keyboard Monitor. CAMP prints a # to indicate that it is ready to receive a command. The command line entered may be terminated with a carriage return (CAMP retains control) or an ALTMODE (control returns to the Keyboard Monitor).

CAMP Commands

Each CAMP command begins with a keyword, consisting of two or more letters. The full CAMP command need not be typed; each command has letters that are required. The CAMP commands are listed below in alphabetic order. Letters that are not required are printed in italics.

BA*CKSPACE*
EO*F*
HE*LP*
RE*WIND*
SK*IP*
UN*LOAD*
VE*RSION*

BACKSPACE COMMAND

The BACKSPACE command spaces a magnetic tape or cassette backward a specified number of files or records. This command may also be issued indirectly with the CCL BACKSPACE command. (See the CCL section of Chapter 1.)

The BACKSPACE command has the form:

BA dev: nnnn { Records / Files

Where "dev" is the permanent name of a cassette or magnetic tape drive. The "nnnn" is an unsigned decimal number representing the number of records or files to backspace. This number must be in the range 0-4095. If no number is entered, nnnn=1 is assumed. This number is followed by a keyword beginning with

either an R, indicating records, or an F indicating files. If neither F nor R is entered, F is assumed.
Examples:

```
#BA CSA0: 2 F
```

positions the cassette mounted on CSA0 backward two files.

```
#BA MTA1:
```

positions the magnetic tape mounted on MTA1 backward two files.

If a file mark is read before the proper number of records have been spaced over, the message:

```
% CAN'T - AT BOF
```

is printed and the device is moved forward one record to leave the device positioned at the beginning of the file (just before a data record).

The file at which the device is currently positioned is not counted when an attempt is made to backspace a number of files. For example, the command:

```
#BA MTA1: 3 F
```

moves backward over four file marks and then moves forward one record, leaving the tape positioned at the beginning of the file. If nnnn=0, this command backspaces to the beginning of the file at which the tape is currently positioned.

EOF COMMAND

The EOF command writes a single file mark (file gap) on the specified magnetic tape or cassette. This command may also be issued indirectly with the CCL EOF command.

The EOF command has the form:

EOF dev:

where "dev" is the permanent name of a cassette or magnetic tape drive.

Example:

```
#EOF CSA1:
```

HELP COMMAND

The HELP command prints a short message on the console terminal, reminding the user of the CAMP command syntax. This command is of the form:

```
#HELP
```

REWIND COMMAND

The REWIND command issues a rewind command to one of the following OS/8 device controllers: cassette, magnetic tape, or TC08 DECtape.

The REWIND command is of the form:

REWIND dev:

where "dev:" can be any OS/8 file-structured device. If "dev" is a cassette, control returns to CAMP while the cassette is rewinding: CAMP prints another #, indicating that it is ready to receive another command. If "dev" is magnetic tape or TC08 DECtape, the device rewinds immediately and control returns to the OS/8 Keyboard Monitor while the device is rewinding. If a REWIND command is issued to any other OS/8 device (e.g., LINCtape), control returns to CAMP after the device is rewound.

Example:

```
#RE DTA1:
```

SKIP COMMAND

The SKIP command advances over the number of files or records specified on a magnetic tape. This command may also be issued indirectly with the CCL SKIP command. The SKIP command is not implemented for cassettes.

The SKIP command has the form:

#SKIP MTAn: { nnnn Records / Files / EOD }

where MTAn may be any magnetic tape drive, depending upon the number of magnetic tape drives on the OS/8 system. The "nnnn" is an unsigned decimal number representing the number of files or records to be advanced over. This number must be in the range 0-4095. EOD indicates that the tape is to be advanced to the end of data. The end of data on a magnetic tape is a point between two file marks. If EOD is specified, the tape must be rewound before issuing the command if it is already past the end of data. If neither "nnnn" nor EOD is specified, nnnn=1 is assumed.

If a number is specified, it may be followed by a keyword beginning with either an R, indicating records, or an F, indicating files. If neither F nor R is entered, F is assumed.

Examples:

```
#SKIP MTA0: 2 RECORDS
```

advances the magnetic tape on MTA0 forward two records.

```
#SKIP MTA1: 6 F
```

advances the magnetic tape on MTA1 forward six files.

If a file mark is read before the proper number of records have been advanced over, the warning message:

```
% CAN'T - AT EOF
```

is printed and the tape is moved backward one record to position the tape at the end of the file (just after the last data record but before the file mark). If nnnn=0, nnnn=1 is assumed when skipping records.

The file at which the tape is currently positioned is counted when an attempt is made to advance over a number of files. Thus nnnn=1 means to advance to the beginning of the next file. If nnnn is greater than 0, the tape is positioned at the beginning of a file (just after a file mark but before any data records). If nnnn=0, the tape is advanced to the end of the file at which it is currently positioned (before a file mark, but after all data records).

If the end of data is encountered before the specified number of files have been skipped, the warning message:

```
% CAN'T - AT EOD
```

is printed and the tape is positioned at the end of data. If a tape is already positioned at the end of data, the SKIP command produces meaningless results.

UNLOAD COMMAND

The UNLOAD command rewinds and turns off line a magnetic tape controller and returns to CAMP for another command while the tape rewinds. Since the magnetic tape is turned off line, it must be manually turned on line to be used after a UNLOAD command.

The UNLOAD command may also be used to unload TC08 and TD8E DECtapes off of their reels. When used on DECtapes, the UNLOAD command rewinds the DECtape on the unit specified, selects a different unit, and returns control to CAMP for another command. This DECtape unit cannot be used until another legal command, e.g., the Keyboard Monitor ASSIGN command, is issued to the DECtape controller.

The UNLOAD command can also be used to write-lock an RK8E disk.

The UNLOAD command is of the form:

#UNLOAD dev:

where "dev" may be any one of the following:

magnetic tape
TC08 DECtape
TD8E DECtape
RK8E disk

VERSION COMMAND

The VERSION command prints the version number of CAMP on the terminal. This command is of the form:

```
#VERSION
```

CAMP Error Message Summary

The error messages listed in Table 2-13 may appear during a CAMP operation.

Table 2-13 CAMP Error Messages

Messages	Explanation
% CAN'T—AT BOF	A file mark was read before the specified number of records were read over in a BACKSPACE command. The device is moved forward so that it is positioned at the beginning of the file.
? CAN'T—AT BOT	A BACKSPACE command cannot move the device backward the specified number of files because the device is positioned at the beginning of the first file.
% CAN'T—AT EOD	The specified number of files cannot be advanced over because the end of data was encountered. The tape is positioned at the end of data.
% CAN'T—AT EOF	A file mark was read before the specified number of records were advanced over in a SKIP command. The tape is moved backward one record to leave it positioned at the end of the file.
? CAN'T—DEVICE DOESN'T EXIST	The device specified in a CAMP command is not present on the OS/8 system.
? CAN'T—DEVICE IS READ-ONLY	The device specified in a CAMP command is a read-only device, e.g., PTR.
? CAN'T—DEVICE IS WRITE-ONLY	The device specified in a CAMP command is a write-only device, e.g., TTY.
? CAN'T FOR THIS DEVICE	The operation specified does not make sense for the device specified, e.g., REWIND LPT:.

Table 2-13 CAMP Error Messages (Cont.)

Messages	Explanation
? CAN'T I/O ERROR	This message is followed by a brief explanation of the input/output error that occurred.
? NUMBER TOO BIG	The "nnnn" specified in a BACKSPACE or SKIP command is greater than 4095.
? SYNTAX ERROR	An illegal character was typed in a CAMP command or a command was formatted incorrectly. The command must be retyped.

CROSS-REFERENCE PROGRAM (CREF)

CREF aids the programmer in writing, debugging and maintaining assembly language programs by providing the ability to pinpoint all references to a particular symbol. CREF operates on output from either the PAL8, SABR, or RALF assembler.

Calling and Using CREF

To call CREF from the system device, type

```
R CREF
```

in response to the dot printed by the Keyboard Monitor. The Command Decoder is loaded and replies by printing an asterisk at the left margin. The user enters one output file specification and one input file specification.

NOTE

The input to CREF must be the listing pass output from either the PAL8, SABR, or RALF assembler. If this is not the case, CREF will not operate properly.

If no output file is specified, CREF assumes the output is to be sent to the line printer. If no input file extension or output file extension is specified, the extension .LS is assumed. If no input file is specified, control returns to the Command Decoder until an input file is specified. The CREF version number is printed at the end of the CREF table in the form Vn, where n is the current version number.

CREF OPTIONS

The following options are available to the user. The option is placed in the command string, along with the file specifications.

Table 2-14 CREF Options

Option Code	Meaning
/P	Disable pass one listing output. The output is re-enabled when $ (or END if SABR code) is encountered. Thus the $ (END) and symbol table are printed if the /P option is used. Inoperable for RALF output.
/U	Disable pass one listing output and the symbol table. Inoperable for RALF output.

Table 2-14 CREF Options (Cont.)

Option Code	Meaning
/R	Interpret input as RALF code.
/Q	Interpret input as SABR code. Signal CREF to accept special SABR characters. If the /Q option is used, the /X option is forced on.
/X	Do not process literals. For programs with too many symbols and literals for CREF, this option may create enough space for CREF to operate.
/E	Do not eliminate the file CREFLS.TM. If the /E option is not specified, and CREF was chained to from PAL8, the file CREFLS.TM is eliminated.
/M	Cross-reference mammoth files in two major passes. Pass one processes the symbols from A through LGnnnn; pass two processes the symbols from LHnnnn through Z and literals. This permits significantly large files to be cross-referenced. If the /M option is used, the file CREF.SV must be on the system device.

EXAMPLES OF CREF USAGE

Examples of calling and using CREF are given below.

Example 1:

```
.R CREF
*PTEMP
```

The Command Decoder prints an *, CREF assigns LPT: as the output device. The input file is PTEMP, assumed to be on device SYS, with the extension .LS. If the file SYS:PTEMP.LS is not found, a search for SYS:PTEMP is attempted.

Example 2:

```
.R CREF
*SBRLS/R
```

Given to the Command Decoder, this command string causes output to be sent to the line printer. The input is expected to be a SABR listing file named SBRLS.LS or SBRLS from device SYS:.
Example 3:

```
.R CREF
*DTA1:LIST<DTA3:PALIST/X
```

This command string causes output to be sent to DECtape unit 1, as a file named LIST.LS. Input is expected to be a PAL8 listing file called PALIST.LS or PALIST. No literals appear in the CREF output table.
Example 4:

```
.R CREF
*DTA2:LIST<SYS:BIGLST
```

The source listing, symbol table, and cross-reference of symbols in the file BIGLST or BIGLST.LS on SYS is in the file LIST.LS on DTA2. To list the CREF output the user may now run PIP.SV as follows:

```
.R PIP
*LPT:<DTA2:LIST.LS
```

Pseudo-Op Handling

CREF recognizes certain pseudo-ops of the PAL8 and SABR assemblers, these certain pseudo-ops cause CREF to perform actions similar to those taken by the assembler whose output is being processed. These pseudo-ops are described below:

PAL8 Pseudo-Op	Action Taken by CREF
EXPUNGE	CREF purges its current symbol table of all permanent and user-defined symbols. If any literals were in the symbol table, they are not deleted.
FIXTAB	Causes all symbols (except literals) to be marked as permanent symbols. After a FIXTAB, no references will be reported by CREF.
TEXT	Ignores characters between delimiters.
$	End-of-input signal.

SABR Pseudo-Op	Action Taken by CREF
END	End-of-input signal.
OPDEF	Creates a new permanent symbol table, a non-skip type instruction.
SKPDF	Creates a new permanent symbol table, a skip type instruction.

NOTE

Symbols entered by OPDEF and SKPDF are processed by CREF. All references to these defined symbols are listed. However, no reference is flagged as a definition (i.e., no reference is followed by a # in the CREF listing).

TEXT	Ignores characters between delimiters.

Interpreting CREF Output

The output of CREF consists of two parts. On the first pass through the input file CREF generates a sequence numbered listing file. The sequence numbers are decimal. The /P and /U options disable this part of the output.

The cross-reference table appears after the listing. This table contains every user-defined symbol and literal, sorted alphabetically. Each literal is indicated by an underline (or back-arrow on most DEC terminals) and followed by the field and address at which the literal occurs. For each symbol and literal there appears a list of numbers specifying the line in which each is referenced.

If CREF finds too many references to fit into core at one time,

multiple passes are required to process all symbols. The minimum number of passes is two. The maximum number of passes depends on the size of the input file, and the amount of core available. CREF calculates the number of core fields available and uses all available space for reference tables. If there is not enough core available, three or more passes are required. For example, the current OS/8 SABR assembler (5518 source lines, 849 symbols) requires four passes through CREF on an 8K machine.

The following example illustrates a program which has been assembled with PAL8 and listed with CREF. Form feeds on the terminal have been converted to a series of carriage return/line feed combinations and a dotted tear line. Notice that in the CREF table the line where the symbol is defined is followed by a # . Symbols defined by OPDEF or SKPDF in SABR, and all literals do not have a # following them.

/EXAMPLE PROGRAM

```
/  EXAMPLE PROGRAM                          PAL8-V9B  03/05/74  PAGE 1

   1                    /   EXAMPLE PROGRAM
   2                    /   ILLUSTRATING DETAILS OF LISTING FORMAT
   3                    /   USING PAL8 AND CREF
   4            0200    *200
   5   00200    7300    START,  CLA CLL
   6   00201    1207            TAD A           /CURRENT PAGE SYMBOL
   7   00202    1777'           TAD B           /OFF-PAGE SYMBOL, LINK GENERATED
   8   00203    1177            TAD I2          /PAGE ZERO LITERAL
   9   00204    1376            TAD I3          /CURRENT PAGE LITERAL
  10   00205    3777'           DCA LINK        /OFF-PAGE SYMBOL, LINK GENERATED
  11   00206    5610            JMP I ADDRP2    /USER CREATED LINK
  12   00207    0011    A,      0011
  13   00210    0400    ADDRP2, P2              /INDIRECT ADDRESS
  14   00376    0003
  15   00377    0407
  16            0400    *400
  17   00400    1207    P2,     TAD LINK        /PAGE 2 START
  18   00401    1377            TAD I3          /NOTE THAT THIS IS A NEW LITERAL
  19   00402    1177            TAD I2          /NOTE THAT THIS IS SAME OLD LITERAL
  20   00403    1377            TAD I3          /SAME AS CURRENT PAGE LITERAL
  21   00404    3207            DCA B           /CURRENT PAGE SYMBOL
  22   00405    6213            CDF CIF 10      /CHANGE FIELDS
  23   00406    5776'           JMP FLD1        /OFF PAGE SYMBOL, LINK GENERATED
  24   00407    0000    LINK,   0
  25            0407            B=LINK
  26   00576    0200
  27   00577    0003
  28   00177    0002
  29            0001    FIELD 1
  30   10200    1377    FLD1,   TAD I3          /FIELD 1, DEFAULT TO PAGE 1 *200
  31   10201    1177            TAD I2          /NEW LITERAL, BECAUSE IN PAGE 0 OF NEW FIELD
  32   10202    6203            CIF CDF 0       /CHANGE FIELDS AGAIN
  33   10203    5200            JMP START       /NO LINK GENERATED, SAME PAGE, OTHER FIELD
  34   10377    0003
  35                    $
  36   10177    0002
```

```
/   EXAMPLE PROGRAM                          PAL8-V9B  03/05/74  PAGE 2

A       0207
ADDRP2  0210
B       0407
FLD1    0200
LINK    0407
P2      0400
START   0200

ERRORS DETECTED: 0
LINKS GENERATED: 3


A                6     12#
ADDRP2          11     15#
B                7     21      25#
FLD1            23     30#
LINK            10     17      24#     25
P2              13     17#
START            5#    33
.00177           8     19
.00376           9
.00577          18     20
.10177          31
.10377          30

V3
```

Restrictions

CREF has the following restrictions:

1. CREF can handle a maximum of 896 (decimal) symbols in one major pass. (In 8K, PAL8 is limited to 897 symbols while SABR is limited to fewer than 800 symbols.) If more than 896 symbols are found, an error message is generated.
2. If any symbol in the input file has more than 2044 (decimal) references, an error message is generated.
3. If more than 8192 (decimal) source lines are input, sequence numbers return to 4096, not 0.
4. If the /D option is used in PAL8 (to generate a DDT compatible symbol table) and the output listing is put through CREF, no symbol table listing will appear.
5. Use of semicolons—This is a restriction which, when not observed, could cause errors in the CREF table. It is recommended that the user follow these suggestions when preparing source files in order to insure a proper CREF listing. Semicolons should not be used on lines with pseudo-ops. In particular, a combination such as the following must not be used:

```
*3000
TEST  %ERROR% ; TAD [42

EXPR=0
```

In this case, CREF does not process the page zero literal properly. A literal is generated which is derived from the expanded TEXT message. No error message is generated, but the literal table entry is meaningless. As a general rule, semicolons should not be used as line terminators inside conditional assembly brackets (<>). For example:

```
EXOR=0
IFNZRO EXOR<CLA;TAD B; HLT \ERROR>
\THIS IS THE NEXT LINE PAST IFNZRO
```

The conditional code is not assembled; however, CREF does not realize this and tries to process the bracketed instructions. As a result of these semicolons, extra symbols may be processed and some valid references missed. However if the code had been assembled CREF would operate properly. There are two ways around this:

a. Write straight line code:

```
EXOR=0
IFNZRO EXOR <
CLA
TAD B
HLT ERROR
>
```

b. Use XLIST around conditional code, in the above example:

```
IFZERO EXOR <XLIST>
IFNZRO EXOR <CLA;TAD B; HLT\ERROR>
IFZERO EXOR <XLIST>
```

XLIST turns off the listing if the code does not assemble and turns it back on after the conditional code.

6. Formats—There are several output formats that can be used in generating a PAL8 listing file:
   /T Form feeds converted to carriage return/line feeds.
   /H No heading or form feeds generated.
   /D DDT compatible symbol table is generated.
   For best results with CREF, none of these switches should be

used. This generates a heading and form feed in the output. CREF automatically converts form feeds to carriage return/ line feeds if output is to the terminal.

7. PAL8 generated links do not cause a reference to a link to be noted by CREF. Only literals specifically generated with (and [are processed by CREF.

CREF Error Messages

CREF errors are non-recoverable errors, and control returns to the Keyboard Monitor through location 07605 (no core saved). Table 2-15 lists the error messages printed by CREF.

Table 2-15 CREF Error Messages

Error Message	Meaning
SYM OVERFLOW	More than 896 (decimal) symbols and literals were encountered during a major pass.
ENTER FAILED	Entering an output file was unsuccessful—possibly output was specified to a read only device.
OUT DEV FULL	The output device is full (directory devices only).
CLOSE FAILED	CLOSE on output file failed.
INPUT ERROR	A read from the input device failed.
DEV LPT BAD	The default output device, LPT, cannot be used, as it is not available on this system.
2045 REFS	More than 2044 (decimal) references to one symbol were made.
HANDLER FAIL	This is a fatal error on output, and can occur if either the system device or the selected output device is WRITE-LOCKed.

DIRECT

DIRECT is an OS/8 program that produces listings of OS/8 device directories. The directories produced can be of several varieties, depending upon the options specified in the DIRECT command line. The standard directory listing consists of the following columns: file name, file name extension, length (decimal) in blocks written, and creation date.

DIRECT supports the wild card construction, using * in place of the file name or extension or ? in place of a character. See the FOTP section of this chapter for a description of wild card construction.

Calling and Using DIRECT

To call DIRECT from the system device, type:

```
.R DIRECT
```

in response to the dot printed by the Keyboard Monitor. DIRECT may also be called via the CCL command DIR (see the CCL section in Chapter 1). The Command Decoder prints an asterisk at the left margin, indicating that it is ready to accept a line of I/O files and options. One output specification and one to five input specifications can be entered in a DIRECT command line. The I/O command line may be terminated with a carriage return (DIRECT retains control) or with an an ALTMODE (control returns to the Keyboard Monitor).

The output specification consists of a device upon which the directory is to be produced, a file name, and a file name extension. All parts of the output specification are optional, as is the output specification itself. A file name and extension should be specified if it is desired to save the directory for listing at a later time. If no output device is specified, TTY is assumed. If a file name is given without an extension, the extension .DI is assumed. The wild card ? and * are not permitted in DIRECT output file names or extensions.

A DIRECT input specification consists of a device, an optional file name, and an optional extension. The wild card * and ? are permitted in input specifications. If an input device is specified with no file name or extension, *.* is assumed. DIRECT determines which files have the form specified and prints a directory listing of just those files.

DIRECT OPTIONS

The following table lists the options that may be used in a DIRECT I/O specification line. Examples of the use of these options are shown following Table 2-16.

Table 2-16 DIRECT Options

Option	Meaning
/B	Include the starting block numbers (octal) for each file in the directory.
/C	List only files with the current date, i.e., the date entered with the most recent DATE command.
/E	Include empty file spaces in the directory listing.
/F	List a short form of the directory, omitting file lengths and dates.
/I	List additional information words in octal, other than the first which is listed as the date.
/L	List the standard form of the directory, including file name, extension, length in blocks, and creation date. The /L option is assumed if none is specified.
/M	List only the empty spaces in the directory.
=n	Use n columns in the directory listing. This option allows the user to specify the number of directory entries per line of output. The "n" must be in the range 0 to 7. The =n option is useful when a wide column printer, e.g., 132 columns, is being used.
/O	List only files with other than the current date.
/R	List the remainder of the files after the first one found. This option causes DIRECT to find the first file that matches the specifications given and then list a directory that includes the first matching file and all files that follow it on the device. The /C and /O options are still considered when listing these remaining files. If /R and /V are used in the same command, only the first file of the form specified is listed.
/U	Treat each input specification separately. The /U option creates a separate directory listing for each input specification.
/V	List files not of the form specified.
/W	Print the version number of DIRECT.

DIRECT EXAMPLES

The following are legal command strings to DIRECT and the resultant DIRECT output. To facilitate understanding of the DIRECT options, the same device (DTA0) is used for each of the examples, and the current date is 21-JAN-74.

When DIRECT has completed an operation, control returns to the Command Decoder for additional input.

Example 1:

This example shows a directory of all the files on DTA0, listed in two columns on the terminal (TTY).

```
.R DIRECT
*DTA0:=2

 21-JAN-74

MTPALA.PA    1  18-JAN-74      WNTSTA.BA    1  18-JAN-74
MTPALB.PA    1  18-JAN-74      WNTSTB.BA    1  19-JAN-74
WNTSTC.BA    1  19-JAN-74      WNPALA.PA    1  19-JAN-74
WNPPPA.PA    1  19-JAN-74      WNTSTD.BA    1  21-JAN-74
WNPALB.PA    1  21-JAN-74      MTPALC.PA    1  21-JAN-74
WNXX  .BA    1  21-JAN-74      WNXY  .BA    1  21-JAN-74

 718 FREE BLOCKS
```

Example 2:

This example shows all files that have a file name beginning with WN, have any file extension, and do not have the current date. The directory is listed in two columns on TTY.

```
*DTA0:WN????.*/O=2

 21-JAN-74

WNTSTA.BA    1  18-JAN-74      WNTSTB.BA    1  19-JAN-74
WNTSTC.BA    1  19-JAN-74      WNPALA.PA    1  19-JAN-74
WNPPPA.PA    1  19-JAN-74

 718 FREE BLOCKS
```

Example 3:

This example shows files that have any file name, have a .BA extension, and have the current date. The directory is listed in a single column on TTY.

```
*DTA0:*.BA/C

 21-JAN-74

WNTSTD.BA     1  21-JAN-74
WNXX  .BA     1  21-JAN-74
WNXY  .BA     1  21-JAN-74


 718 FREE BLOCKS
```

Example 4:

This example demonstrates the use of the /U option to produce separate directories for each input spcification. The command specifies that all files beginning with WN and having .BA extensions be listed first, and that all files beginning with WN and having .PA extensions be listed next. The short form of the directory is to be listed on the line printer (LPT) in three columns.

```
*LPT:<DTA0:WN????.BA,WN????.PA/F/U=3

 21-JAN-74

WNTSTA.BA       WNTSTB.BA       WNTSTC.BA
WNTSTD.BA       WNXX  .BA       WNXY  .BA

718 FREE BLOCKS

 21-JAN-74

WNPALA.PA       WNPPPA.PA       WNPALB.PA


 718 FREE BLOCKS
```

Example 5:

This example demonstrates the use of the /V option to print files not of the form specified and the use of the /O option to exclude files with the current date. All files except those beginning with WN are to be printed in a single column on TTY.

```
*DTA0:WN????.*/O/V

 21-JAN-74

MTPALA.PA     1  18-JAN-74
MTPALB.PA     1  18-JAN-74


 718 FREE BLOCKS
```

Example 6:

This example demonstrates the use of the /R option to list part of the directory. DIRECT is to find the first file that begins with WN and has a .PA extension; that file and all files that follow are to be listed. The directory is listed in two columns on TTY.

```
*DTA0:WN????.PA/R=2

 21-JAN-74

WNPALA.PA    1  19-JAN-74      WNPPPA.PA    1  19-JAN-74
WNTSTD.BA    1  21-JAN-74      WNPALB.PA    1  21-JAN-74
MTPALC.PA    1  21-JAN-74      WNXX  .BA    1  21-JAN-74
WNXY  .BA    1  21-JAN-74

 718 FREE BLOCKS
```

Direct Error Messages

The following error messages may appear when running the DIRECT program.

Table 2-17 DIRECT Error Messages

Message	Meaning
BAD INPUT DIRECTORY	This message occurs when the input device has a bad directory, e.g., the device is not an OS/8 device, or a DECtape has not been zeroed.
DEVICE DOES NOT HAVE A DIRECTORY	The input device is a non-directory device, e.g., PTR. DIRECT can only read directories from file-structured devices.
EQUALS OPTION BAD	The =n option is not in the range 0-7.
ERROR CLOSING FILE	System error.
ERROR READING INPUT DIRECTORY	An error occurred while reading the directory.

Table 2-17 DIRECT Error Messages (Cont.)

Message	Meaning
ERROR WRITING FILE	An error occurred while writing the output file.
ILLEGAL *	An asterisk (*) was included in the output file specification or an illegal * was included in the input file name.
ILLEGAL ?	A question mark (?) was included in the output file specification.
NO ROOM FOR OUTPUT FILE	Self-explanatory; the output device does not have sufficient space for the directory to be written.
THERE IS NO HOPE—THERE IS NO TTY HANDLER IN YOUR SYSTEM!	A command was issued to print a directory on the terminal when no TTY handler is present on the OS/8 system. Use BUILD to insert a TTY handler in the system.

EPIC

Introduction

EPIC, the Edit, Punch and Compare utility program for OS/8, is designed primarily to assist users by performing the following functions:

1. Read and punch paper tape files and patches
2. Edit arbitrary files
3. Compare files in any format

When EPIC is loaded, the command line determines which function is desired. Each of these functions is discussed as a separate topic in these next few pages. This section assumes an elementary knowledge of OS/8.

Loading EPIC

To load the EPIC program type R EPIC in response to the OS/8 monitor's dot (.). Specify the EPIC function desired by including one of the following numeric options in the file command line:

0 paper tape
1 edit
2 compare

```
.R EPIC
*TRANS.AS</0$
```

punch the file TRANS stored on SYS.

```
.R EPIC
*DTA1:FILEA.SV</1$
```

fetch FILEA from DTA1 for editing

```
.R EPIC
*DSK:ABC.SV<DTA1:XYZ.SV/2$
```

compare file ABC on the disk with file XYZ on DTA1 and output block numbers and locations of each non-match on the Teletype.

After one of these numeric options has been included in a command, it need not be specified again in subsequent sequential commands requiring the same option. Specifying the number puts EPIC in a mode and it remains in that mode until another number is specified. Initially, EPIC is set to option 0. The character ALTMODE, which prints as $ on the terminal, is used to end a command that includes a numeric option.

Restart Procedure

EPIC can be restarted at location 0200. Default options remain active. The default options are discussed later in this section.

Paper Tape Facility

The paper tape option (/0) of EPIC punches OS/8 files and file patches onto paper tape and creates OS/8 files from paper tapes. Whole files or patches (blocks) of files can be read or punched. Parity checks are punched to assure accurate reads. Note that a unique paper tape format is used so that tapes must be both punched and read by EPIC. A file punched by PIP, for example, is not acceptable to EPIC.

Command Format

To request the paper tape facility, the option 0 must be specified. The form of the response to the command decoder's * determines whether a tape is to be punched or read. In both cases, no input files or devices are specified. To punch a tape, the file name is specified; to read a tape, no file name is required (that information is encoded on the paper tape). The command line specifying the mode of EPIC is terminated by ALTMODE.

To punch a tape, the response is:

*dev:name</0/other options$

To read a tape, the response is:

*dev:</0/other options$

If a file name is specified, EPIC looks up the name on the specified device and punches the file (including the file name) onto paper tape. If no file name is specified, EPIC reads in a paper tape and enters it onto the output device under the name it read in from the tape.

The other options for handling paper tape are:

L	Use low speed paper tape reader or punch
E	Do not punch end of tape upon completion
P	Punch or read a patch (instead of the whole file)
Z	Set relative block to 0
=n	Punch relative block n
Y	Clear default name

These options can be combined to achieve the desired results.

L Option: If the /L option is not specified, EPIC assumes a high-speed paper tape device. Thus, SYS:</0 means read a tape from the high-speed reader to device SYS but SYS:</0/L means read it from the low-speed device.

E Option: The /E option can be used to punch a series of patches to a file for all patches except the last one. With the /E option the end of tape mark is not punched. The end of tape must have the "end of tape" punch, a 377 punch and a length of leader/ trailer tape.

P Option: The /P option is required to indicate the tape to be read or punched is a patch, not an entire file. Generally, the command required to read in a patch is simply dev:</P. File name and block specifications are already punched on the tape.
Option /Z or =n must be used with the /P option to indicate punching block 0 or some other block (relative block n), respectively. The patch is read on top of an existing file on the specified output device, i.e., modifying an old file, not creating a new one.

Y Option: The /Y option is used to clear the default file name when switching from punching to reading paper tape and when reading more than one paper tape.

Default Options

Throughout EPIC, if options, files, or devices are not specified, the program defaults to the last such item specified. There is an initial default device: SYS is assumed if no output device is specified. No options are assumed initially, however, except for relative block 0. Note that device and file name options carry between EPIC modes 0, 1 and 2. Specifying an option (i.e., L, P, E, Z, etc.) in a command string disables default to any options from the previous command (except 0, 1, 2).
For example, to punch blocks 0, 1 and 30 of the file TRANS on the SYS device and read them back onto that file on DTA3, the commands are:

	Punch block 0 of TRANS on high-speed punch with no end of tape punch. Note that EPIC defaults to the paper-tape option initially so 0 is not required in this case.
.R EPIC	Punch block 1 of file TRANS with no end of tape character on high speed device.
*TRANS</P/E/Z$	Punch block 30 of the file TRANS on high-speed punch. Punch end of tape (P disables E).
*=1	
*=30/P	
*DTA3:</Y	Read the tape from the high speed device and put out to file whose name is encoded in the patch on device DTA3 until end of tape is reached. File name and relative block are punched on the tape so this information is not necessary. Y clears the default name. (TRANS)

Error Conditions

If an error occurs while reading a block of paper tape, EPIC outputs an appropriate error message (the error messages are listed at the end of this section), and halts; the user should reposition the paper tape to the leader/trailer just in front of the block just read before continuing (refer to the section on Paper Tape Format); three consecutive read errors terminate the command. When EPIC is reading in a non-patch file it checks the initial block read of every tape and every block that is reread because of error to determine if the read was accurate up to name and block number. If the wrong block number or file name is read, EPIC outputs an appropriate message indicating the type of error and halts with AC=7777 to allow the user to reposition the tape over the correct block or enter the correct tape before continuing.

Low Speed I/O

The execution of EPIC differs for low speed I/O. Before starting a low speed punch EPIC halts with 7777 in the AC to allow

the user to turn on the low speed punch and then press the CONT key on the computer console. Upon completion of a punch command EPIC halts with the AC=0 to allow the user to turn off the punch. When the CONT key is pressed, EPIC recalls the command decoder. For low speed input EPIC halts only upon completion of the read.

If a file or a series of files to be punched exceeds 32 blocks, EPIC segments it by punching end of tape after 32 blocks. This end of tape punch is done automatically and independently of the E option; its purpose is to keep tapes physically short enough to fit into a paper tape tray. Upon physical end of tape, EPIC halts with the AC=0 if the low speed punch is being used to allow the user to turn off the punch before continuing. As soon as the punch is turned off, EPIC outputs the message END OF TAPE ENTER NEXT and then halts with the AC=7777 to allow both high and low speed users to remove the paper tape. Note that low speed users get both halts, but high speed users only get the 7777 halt. In general, a halt with AC=0 means turn paper tape device off and a halt with AC=7777 means turn device on. All halts are terminated by depressing the console CONTinue key. If EPIC encounters end of tape while reading a non-patch file it outputs the message END OF TAPE ENTER NEXT and halts with AC=7777 indicating that the file is segmented across a number of tapes and that the user should enter the next tape.

Device Codes

Most of the execution time is spent waiting for paper tape devices. During I/O wait, EPIC holds the device code and version number in the AC. The device code is in bits 3-5 and the version number is in bits 6-11. The codes are as follows:

1 high speed reader
2 high speed punch
3 low speed reader (console TTY)
4 low speed punch (console TTY)

If the user forgets to turn on the high speed reader, EPIC hangs with lxx in the AC. EPIC can always be restarted at 0200. The OS/8 CTRL/C is normally in effect; the exceptions are when EPIC is waiting for a paper tape device or when input is from the low speed reader.

NOTE

When input is from the low speed reader EPIC forces the output device to be SYS because it is the only OS/8 I/O handler that does not check for CTRL/C.

Thus, if the user were to enter the command:

```
DTA2:<ALT
```

EPIC would force it to be

```
SYS:<ALT
```

Editing Capability

Option 1 of EPIC is the file editing and searching facility. With this feature, patches can be added directly to the file by specifying relative blocks and locations in the file.

INITIAL COMMAND FORMAT

The general format of a command for the editing option is:

```
.R EPIC
*DEV:NAME</OPTIONS/1$
```

The /1$ specifies edit mode for EPIC.
As with the paper tape option, default conditions apply. If no device and/or file name is specified, the last one mentioned is used. When editing, the only option available in the initial command is

/Y Clear default name (if one exists)

Editing is performed one block at a time. The relative block currently being processed is the current block; the location currently being processed is the current location (0-377). Relative block 0 is the first block of the file if a file name is specified or block 0 of the device if no file name is specified.

EDITING COMMANDS

After the initial (file specification) command, a series of keyboard commands are used to perform the editing. The general format of an editing command is

x

or

x,nl,n2

where x is a command letter and nl,n2 are octal numeric arguments. If a numeric argument is used, the letter is followed by a comma. Up to 32^{10} characters can be typed on a line. Default conditions apply to these commands as well. If carriage return is the only character typed as an editing command, the last command specified is executed. The commands available are as follows:

Table 2-18 EPIC Commands

Command	Meaning
E	Exit to command decoder; write out current block of file if it has been modified.
R, n	Read relative block n (octal) of file and set current location to 0. Do not write current block. If n is not specified, the current block is read. If the relative block is out of range, a ? is printed. There are 1341 blocks per OS/8 tape and 6260 per RK8 disk platter.
W	Write the current block of file if it has been modified and read in the next sequential block of the file. If the current block is the last block of the file, a ? is printed and the current location is unmodified.
S, n1, n2	Search the current block for the value n1 with the mask n2. If either n1 or n2 or both are omitted, the last value specified is used. The initial mask is 7777. Masking is performed in a logical AND fashion. If the S command is terminated by the RETURN key the search is for the current block only. If terminated by the LINE FEED key, the search continues to the end of the file. If the search fails (either in the block for a carriage return or at end of file for line feed) EPIC prints a ?. If the search is successful EPIC prints

Table 2-18 EPIC Commands (Cont.)

Command	Meaning
	m1 m2 m3 /
	where m1 is the relative block, m2 is the relative location within the block and m3 is the contents of the location. (m1 is omitted if a previous match was found in the same block.) To change the contents, type the new contents (octal) after the slash. To continue the search type the LINE FEED key; to terminate the search type the RETURN key. (If the contents are not to be changed, type one of the terminators.)
O, n	Open location n of the current block. If n is not specified, the last opened location is the default. If there is no default, location 0 is opened. EPIC responds with
	m1 /
	which is the contents of location n. This location may be modified as in search. Terminating with the LINE FEED key closes the current location and opens the next. If the current location is the last one in the block, location 0 of the next block is opened and the current block is written out as if it had been modified.
C	Print current status, as:
	m1 (F or B) m2 m3 m4
	where m1 is the current block, m2 is the current location, m3 is the search word and m4 is the mask word. If F is typed, the file has been modified since option 1 was requested; B indicates the current block has been modified. Once a modified block has been written to the file, the F is the only code output.

Thus a reasonable sequence is:

.R EDIT	Call EPIC.
*DSK:ISOMER</1$	Edit file ISOMER on DSK.
R,2	Read block 2.
S,3126,7770	Search for a 312x in that block.
?	Not there.
>	Search for it throughout the file.
0004 0110	Found at block 4, location 110.
3124 /3121	Change contents to 3121.
,,7777	Search for 31xx throughout the rest of the block (locations 110-377).
0004 0132	Found at location 132 of block 4.
3126 / 3127	Contains 3126. Change to 3127.
C	Check status.
0004 B 0132 3126 7777	At location 132 of block 4 which has been modified; the current search word is 3126 and mask is 7777.
W	Write block 4.
?	Block 4 written but file is only four blocks long, no block 5 to read.
R,2	Read block 2.
O,10	Open location 10.
1367 /1364>	Contains 1367. Change to 1364.
3324 .	Check next location. No modifications.
E	Exit editing option.
*	

Compare Capability

A third feature of EPIC is file compare (/2). Because EPIC uses an absolute compare technique, there are no limitations in the data format or the length of the file. The files to be compared must reside on the system device.

COMMAND FORMAT

Option 2 of EPIC requires only one command, specified as follows:

SYS:file1<SYS:file2/options/2$

The first file to be compared is specified to the left of the angle bracket, the second file to the right. The options are:

A Abort when the first non-match is found.

B List physical block number for each file where a non-match exists.

If no options are specified, the block numbers and locations of each non-match are listed on the terminal.

For example, to compare files PYTHG1 and PYTHG2 and find all unequal locations, the sequence is as follows:

```
*SYS:PYTHG1<SYS:PYTHG2/2$
SYS:0174   SYS:0631
0152 7450  3421
0153 5741  2021
0154 3421  3022
*
```

To compare them and list unequal blocks the command is:

```
*SYS:PYTHG1SYS:PYTHG2/B/2$
```

If this block match followed the preceding locations match command, a sufficient command and its results are:

```
*/B
SYS:0174   SYS:0631
```

To abort after the first non-match, the sequence is:

```
*/A
SYS:0174   SYS:0631
```

Error Messages

EPIC can print one of the following error messages when performing paper tape (option 0) operations.

Table 2-19 EPIC Error Messages

Message	Explanation
BAD =BLK	When EPIC is punching a patch it checks the block specified by "=n" to see if it is within range. If the block is out of range EPIC outputs this error message and returns to the command decoder. For example if a file JOE were two blocks long and the user requested: JOE:←/P=3 the error message would be printed.
END OF TAPE	EPIC was expecting a block of tape and found end of tape instead. EPIC halts with AC=7777 to allow the user to reposition the tape. When the user depresses CONTinue EPIC attempts to read the block.
END OF TAPE ENTER NEXT	When EPIC is reading a file that is segmented across a number of paper tapes and encounters the end of a segment, it outputs this message and halts with AC=7777 to allow the user to enter the next segment of paper tape. Press the Console CONT key to continue reading.
I/O ERROR	If EPIC encounters an error while reading or writing a mass storage device, or a paper tape read fails three consecutive times, it outputs this error message, deletes the output file if one exists, and returns to the command decoder.
L/T ERROR	EPIC was expecting leader trailer and found non-leader trailer while attempting to read a block. The program prints this error message and halts with AC=7777 to allow the user to reposition the tape then press the Computer Console CONT key.
NEED:name1 FOUND name2	EPIC read a block of tape for the file NAME2 when it was expecting a block of the file NAME1. This error would typically occur when a user comes to the end of a segment for NAME1 and enters some segment of NAME2 instead of the next segment for NAME1. EPIC halts with AC=7777 to allow the user to enter the correct paper tape.

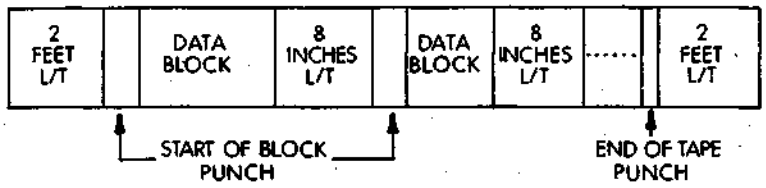
Table 2-19 EPIC Error Messages (Cont.)

Message	Explanation
NEED:n1FOUND:n2	EPIC read block n2 of the file when it was expecting block n1 of the file. EPIC halts with AC=7777 to allow the user to reposition the paper tape. This error typically occurs when the user repositions the tape to the wrong block after a read error.
PARITY ERROR	EPIC failed to read a block correctly, e.g. the reader dropped some bits. EPIC halts with AC=7777 to allow the user to reposition the tape so that it can try the read again.
PTR:NAME IS TOO BIG FOR dev:	The paper tape file NAME will not fit on the specified output device DEV:. EPIC aborts the command and returns to the command decoder. EPIC makes the check for size before writing on the output device..

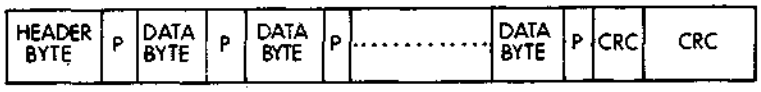
USR n dev:name	The USR encountered an error while attempting to perform a fetch, lookup, enter, or close on the file NAME on device DEV. n=1 is a fetch, n=2 is lookup, n=3 is enter, n=4 is close. EPIC aborts the command and returns to the command decoder. For example, if the user requests EPIC to punch a file on SYS that does not exist:
	
	`SYS:NILL<`
	
	EPIC outputs the message
	
	`USR 0002 SYS:NILL`
	
	indicating that it could not find the file NILL on the device SYS.

Paper Tape Format

Paper tapes punched by EPIC have the following format:



Leader trailer is any string of 0 or 200 punches; usually it's just 200 punches; leader trailer is terminated by a 201 punch which indicates the start of a data block. The first punch after the last data block is 377 which is end of tape. Each data block has the following format:



Each byte is 12 punches (96 bits) and corresponds to 8 12 bit words; each byte is followed by an even odd parity punch of the eight words in the byte. Each block is terminated by two CRC punches of longitudinal parity.

The header byte contains information about the file e.g., file name and relative block number. The data bytes constitute the actual data of the block; there are 32 data bytes per 256 word block.

Loading EPIC From Paper Tape

For users who receive EPIC on paper-tape, use the following procedure to load the tape and save it on a mass storage device.

```
.R ABSLDR
*PTR:$↑
```

Use ABSLDR
Read from reader; after ↑ is output, type any key to start reader

```
.SA SYS EPIC 0-7577;0200=0
```

Save on mass storage with starting address of 200

EPIC Assembly Instructions

The PAL8 (version 9) assembler is used to assemble EPIC as follows:

```
.R PAL8
*DEV:EPIC.BN,DEV:EPIC.LS<DEV:EPIC.PA
```

To create the save file, use ABSLDR:

```
.R ABSLDR
*DEV:EPIC.BN$
.SA DEV EPIC 0-7577/0200=0
```

Call ABSLDR.
Load EPIC.BN on device specified.
Save EPIC on device specified.
0-7577 = area in core used during execution. 0200 = restart address.

FILE ORIENTED TRANSFER PROGRAM (FOTP)

FOTP is an OS/8 program used to transfer files from one device to another, to delete files from a device, and to rename files. FOTP is significantly faster than PIP and performs certain functions not available with PIP. For example, FOTP can transfer files longer than 256 blocks and can perform multiple file transfers and deletions without requiring multiple accesses of the directory.

FOTP copies files in image mode, i.e., it copies the file word for word, character for character, without making any changes in the file. (This corresponds to the /I option in PIP.) Thus FOTP may be used to copy core image and binary files as well as ASCII files, without specifying options to identify the type of file.

Calling FOTP

To call FOTP from the system device, type:

```
R FOTP
```

In response to the dot printed by the Keyboard Monitor. (FOTP may also be called indirectly by several CCL commands. See the CCL section of Chapter 1.) The Command Decoder prints an asterisk at the left margin and waits to receive a line of I/O files and options. FOTP accepts one output specification and up to five input specifications. The I/O specification line may be terminated with a carriage return (FOTP retains control) or with an ALTMODE (control returns to the Keyboard Monitor).

INPUT SPECIFICATIONS

FOTP input specifications consist of a device, a file name, and a file name extension. Input specifications are optional but must be present if no output specification is included.

Within the input specification, FOTP allows a wild card construction to be used. This means that the file name or the extension may be replaced totally with an asterisk or partially with a question mark to designate certain file names or extensions. The asterisk is used as a wild field to designate the entire file name or extension. For example:

TEST1.*	All files with the name TEST1 and any extension.
*.BN	All files with a BN extension and any file name.
.	All files.

The question mark is used as a wild character to designate part of the file name or extension. A question mark is used for each character that is to be matched; e.g., PR?? matches on four characters or less. For example:

TEST2.B?	All files with the name TEST2 and any extension beginning with B.
TES??.PA	All files with a PA extension and any file name up to five characters beginning with TES.
??.??	All files with file names of two characters or less.

The asterisk and the question mark can be specified together in the same command line.

???.*	All files with file names of three characters or less.

The following are examples of legal FOTP input specifications:

```
DSK:
SYS:A
LTA3:TEST1A
DTA7:A.BN
FILE
FILE3.DA
4
NAME?.TX,NAM??.BN
N?ME.
??????.D?
*
*.BN
PRN:*.??
?W?B?Z.?A
```

A specification may not contain embedded *'s, e.g., A*B.* is an illegal specification. The following are illegal input specifications:

```
A,B,C
A:B:C
A?*.B
.AB
DAT:A.*B
A?B:C
*:BIN
```

For each input specification given, if no device is explicitly given, then the device associated with the previous specification is assumed. If no device is explicitly given for the first specification, then DSK: is assumed. Thus, the following input specifications are equivalent:

DSK:B	B
SYS:B.*,C.*,D.*	SYS:B.*,SYS:C.*,SYS:D.*
B.*,DTA0:,SYS:*.BN	DK:B.*,DTA0:,SYS:*.BN

As many as five input specifications can be included in a single command line. If all the files are on the same device, the input device need be specified only once. For example:

```
DTA0:*.BN,*.SV,*.RL
```

refers to files on DTA0 that have .BN, .SV, or .RL extensions with any file name.

OUTPUT SPECIFICATIONS

FOTP output specifications consist of a device, a file name, and a file extension. Output specifications are optional. The wild card asterisk may be used in output specifications, but the question mark is illegal.

If no output device is specified but a file name is given, then DSK: is assumed. If no file name is specified, then *.* is assumed. Thus the following output specifications are equivalent:

A	DSK:A
A.*	DSK:A.*
DTA3:	DTA3:*.*

Using FOTP

Since FOTP performs file transfers in a different manner than other OS/8 transfer programs, the following is a detailed description of the way in which FOTP works. One of the main uses of FOTP is to copy files from one device to another. The following examples are used to show how FOTP examines each aspect of a command to determine what operation will actually take place.

Example 1:

To copy the file SMILE.PA from DTA3 to DTA5, changing its name to FROWN.PA, type:

```
DTA5:FROWN.PA<DTA3:SMILE.PS
```

in response to the * printed by the Command Decoder.

1. If FOTP does not find the file SMILE.PA on DTA3, the message:

```
NO FILES OF THE FORM SMILE.PA
```

   is printed and no transfer is performed.
2. FOTP examines DTA5 to determine whether it already contains a file FROWN.PA. If FROWN.PA is already on DTA5, FOTP deletes it before beginning the transfer. This process is known as predeletion.
3. The /N option is used to specify that no predeletion is desired. Thus the command:

```
DTA5:FROWN.PA<DTA3:SMILE.PA/N
```

   begins to copy SMILE.PA to DTA5 without deleting the old FROWN.PA. FOTP does this by opening a tentative file named FROWN.PA on DTA5. When the transfer operation is successfully completed, the tentative file is closed. Closing this tentative file makes it a permanent file and, at the same time, deletes any old files of the same name. This process is known as postdeletion.
4. FOTP assigns the creation date of SMILE.PA to FROWN.PA. This is an advantage over PIP, which would assign the current date to the new file. If files are always transferred with FOTP, the original creation date of the file is preserved. Thus this feature of FOTP allows the user to differentiate between versions of a file since the more recent version should have a later date.
5. The /T option of FOTP can be used to assign the current date to a file. For example, if SMILE.PA is undated, FOTP assigns the current date to the newly created FROWN.PA.

```
DTA5:FROWN.PA<DTA3:SMILE.PA/T
```

6. Advanced users may be using the additional information words feature of OS/8. This feature allows the knowledgeable user to associate additional information (other than the creation date) with each file entry in a device directory. FOTP transfers such additional information words from SMILE.PA to FROWN.PA. (PIP does not perform this function.)
   If the file structure on DTA5 has space for more additional information words than appeared with SMILE.PA, then those extra words are set to 0.
   If the file structure on DTA5 does not have enough space for all the additional information words associated with SMILE.PA, then FROWN.PA is given as many as can fit (from the left). Excess information words (on the right) are not transferred.

Example 2:

Normally, one copies files from one device to another without changing the file name. For example, to copy the file TEST.PA from DTA1 to DTA2, type:

```
DTA2:TEST.PA<DTA1:TEST.PA
```

in response to the * printed by the Command Decoder. Since this transfer operation is so common, FOTP allows the output file name to be abbreviated to *.*. The *.* means that the input file name is to be used as the output file name. Thus the preceding command could be typed as:

```
DTA2:*.*<DTA1:TEST.PA
```

Since the *.* specification is so frequently used, it is the default, i.e., if no output file name is specified, *.* is assumed. Thus the preceding command may be further simplified to:

```
DTA2:<DTA1:TEST.PA
```

Example 3:

One of the more attractive features of FOTP is that it allows multiple files being transferred from one device to another to be

included in the same command line. For example, to transfer five FORTRAN source files from SYS to RKA2, the user could type:

```
RKA2:*.*<SYS:DATA1.FT,DATA2.FT,DATA3.FT,DATA4.FT,DATA5.FT
```

The wild card characters * and ?, explained previously, are particularly useful when doing multiple file transfers. For example, to transfer all FORTRAN II source files from SYS to RKA2, type:

```
RKA2:*.*<SYS:*.FT
```

The specification *.FT means files with any name that have the .FT extension.

To copy all files from DTA1 to DSK, type:

```
DSK:*.*<DTA1:*.*
```

Note that the *.* specification has different meaning when placed on the left side of the < than it does when placed on the right. When used on the output (left) side, *.* means that the output file name is the same as the input file name. When used on the input (right) side, *.* means transfer or consider all files on this device. For example:

```
RKA2:<SYS:TEST1.PA,TEST2.PA,TEST3.PA
```

copies three files from SYS to RKA2. PIP would require three commands, each transferring one file, to perform the same operation.

Note that in the preceding example, no output file name is specified, so *.* is assumed. No device is specified for the files TEST2.PA and TEST3.PA, so the device specified as the previous input device (SYS) is assumed.

Frequently, several files with similar names (as above) are to be copied from one device to another. In many cases, these files can be referenced by a single file specification by using the ? wild character. For example the command:

```
DTA2:*.*<DTA1:TEST?.PA
```

transfers all files on DTA1 that have the extension .PA and that have names beginning with TEST followed by one other character.

ADVANTAGES OF PREDELETION

The default mode (and the recommended one) of FOTP is to use predeletion when copying files. Predeletion creates space on the output device for the new file. Suppose that, in Example 1 above, DTA5 were almost full. There might not be enough space on DTA5 for SMILE.PA. If, however, FROWN.PA is first deleted, this could create enough space for SMILE.PA

Predeletion normally places the new file in the space occupied by the file being replaced. In Example 1 above, if FROWN.PA is first deleted, the space where it resided is empty. This empty space could then be used for the new copy of FROWN.PA (the former SMILE.PA). If predeletion were not used, the new tentative file for FROWN.PA would probably be placed at the end of the tape. This procedure would create a gap (EMPTY) when the old copy of FROWN.PA was deleted; thus the files on DTA5 would be ordered differently.

ADVANTAGES OF POSTDELETION

Postdeletion is a slightly safer method of transferring files since the original file is not deleted until a transfer is successfully completed. Suppose that, in Example 1 above, SMILE.PA is an updated version of the FROWN.PA that exists on DTA5 and that these are the only two copies of a certain source file. If predeletion is performed and SMILE.PA is discovered to have a permanent input error, that source file will have ceased to exist because SMILE.PA will be unreadable and FROWN.PA will have been deleted. The use of postdeletion in this case would save the original copy (FROWN.PA) even though the updated version (SMILE.PA) could not be read.

CONTROL CHARACTERS

The special characters CTRL/C and CTRL/P are used to terminate FOTP operations. When CTRL/C is typed, FOTP continues operation until the files on the output device are the same as those in the output device directory. Control then returns to the OS/8 Keyboard Monitor.

CTRL/P causes FOTP to terminate the current operation but FOTP retains control. The output device directory is updated to

reflect the operations completed before the termination occurred. FOTP prints an asterisk and can receive another I/O specification line.

If CTRL/C or CTRL/P is typed when deleting (/D) or renaming (/R), no FOTP operations are performed and the message:

```
ORIGINAL DIRECTORY PRESERVED
```

is printed.

FOTP Options

The options listed in Table 2-20 may be used in a FOTP specification line.

Table 2-20 FOTP Options

Option	Meaning
/C	Current date. Consider only those input files with the current date when performing a FOTP operation. For example, if the command: `*DSK:<DTA0:*.*/C` Is typed, FOTP transfers from DTA0 to DSK only those input files that have the current date.
/D	Do not perform any I/O transfers, i.e., perform only deletions. /D is not an abbreviation for delete although it usually performs that operation. This option compares the input specification with the output specification, if any, for matching files. If a match is made, FOTP performs as though transferring the file, and then deletes the transferred file. If no transfer occurs, no postdeletion occurs. Predeletion might still occur unless the /N option is included. If no output device is specified, FOTP assumes the first input device specified as the output device. If no output files or extensions are specified, i.e., *.* is specified or assumed, the input file names become the output file names. If no input files are specified, no deletion takes place.
/F	Failsafe. The /F option protects files during a transfer operation. It is particularly useful when transferring a great number of files from disk to DECtape. The /F option allows a new volume to be mounted if a large file will not

Table 2-20 FOTP Options (Cont.)

Option	Meaning

	fit on the output device or if all files will not fit on the output device. If, for example, a user wishes to transfer all .BN files from DSK to DTA0, he types:
	
	`DTA0:<DSK:*.BN/F`
	
	If the output device becomes full before transfer is complete (or if a large file will not fit), FOTP prints:
	
	`MOUNT NEXT OUTPUT VOLUME:`
	
	Dismount the current tape and mount a new tape on the same unit. Type any character to continue. The device mounted must have a good OS/8 directory. FOTP then continues the transfer on the new volume and updates the directories of both volumes.

/L	List on the terminal the names of files affected during the FOTP operation. Note that neither the device nor the output file is listed.
/N	No predeletion. Delete output file names after a successful I/O transfer occurs. If an I/O transfer proceeds, any other files of the same name will automatically be deleted when the file is closed.
/O	Other than the current date. Consider only those input files with a date other than the current date when performing a FOTP operation.
/Q	Query the user about each relevant file name to determine whether he wants the specified operation to occur for that file. This relevant file name could be either an input or output file name depending upon the type of FOTP operation being performed. For example, if input files are being renamed, FOTP prints the affected input file names. If output files are being deleted, FOTP prints the output files that will be affected. FOTP prints each relevant file name on the terminal and waits for the user to respond. A response of Y causes the specified operation to be performed. Any other response causes that file to be ignored and FOTP prints the next relevant file name.

Table 2-20 FOTP Options (Cont.)

Option	Meaning
/R	Rename the output file without performing any transfer. This operation is performed by specifying the same device as both the input and output device. For example: `DSK:TEST3.PA<DSK:TEST2.PA/R` would change the name of the DSK file TEST2.PA to TEST3.PA without performing any transfer.
/T	Assign the current date to the corresponding input file.

/U	Treat each input specification separately. This option causes FOTP to find files in the same order as they are entered in the input specifications. For example, the command:
	
	`DTA0:<DSK:TEST.PA,DATA1.FT,TEST2.PA/U/L`
	
	`TEST.PA`
	`DATA1.FT`
	`TEST2.PA`
	
	finds the files in the order that they were specified in the command, not in the order in which they may appear on DSK.

/V	Consider only input files which do not have the form specified by the input specifications. For example, the command: `DTA0:<SYS:*.SV,*.HL/V` transfers to DTA0 all files on SYS other than those with .SV or .HL extensions.
/W	Print the version number of FOTP on the terminal.

EXAMPLES OF FOTP SPECIFICATION COMMANDS

The following are legal command strings to FOTP. When FOTP has completed an operation, control returns to the Command Decoder for additional input, unless the ALTMODE is used to terminate the FOTP command line.

Example 4:

```
DTA0:<A.B
```

This command string transfers the file A.B from the device DSK to DTA0.
Example 5:

```
DTA3:<SYS:A,B,C,D,E
```

This command string transfers the files A, B, C, D, and E from the system device to DTA3.
Example 6:

```
DTA2:<DTA5:*.FT/L
```

This command string transfers all FORTRAN source files from DTA5 to DTA2, producing a log of those copied.
Example 7:

```
LPT:<*.FT,*.BA/U
```

This command string lists all FORTRAN files, then all BASIC files on the line printer.
Example 8:

```
DSK:<DTA3:*.SV,*.BN,DTA2:K????.*/V/L
```

This command string copies from DTA3 to DSK all files other than core image (.SV) and binary (.BN); it then copies from DTA2 to DSK all files other than those with names beginning with K. A listing is printed of all files copied.
Example 9:

```
DTA1:C.D<A.B/T
```

The above command copies the file A.B from DSK to DTA1, changing its name to C.D, and assigns the current date to the file.

Example 10:

```
SYS:*.PL<LTA2:*.PA/N
```

The above command copies from LTA2 to the system device all files with .PA extension, changing the extension to .PL.
Example 11:

```
*.LS,*.TM,*.BK,TMP???.*/D/O
```

This command string deletes any disk file which has an extension of .LS, .TM, or .BK or has a name beginning with TMP if the file does not have the current date.

Error Messages

The error messages listed in Table 2-21 may appear during a FOTP operation.

Table 2-21 FOTP Error Messages

Message	Meaning
ALREADY EXISTS (file name)	An attempt was made to rename an output file with the name of an existing output file.
BAD INPUT DIRECTORY	The directory on the specified input device is not a valid OS/8 device directory.
BAD OUTPUT DEVICE	Self-explanatory. This message usually appears when a non-file structured device is specified as the output device.
BAD OUTPUT DIRECTORY	The directory on the specified output device is not a valid OS/8 device directory.
DELETES PERFORMED ONLY ON INPUT DEVICE GROUP 1 CAN'T HANDLE MULTIPLE DEVICE DELETES	More than one input device was specified with the /D option when no output specification (device or file name) was included.

Table 2-21 FOTP Error Messages (Cont.)

Message	Meaning
ERROR ON INPUT DEVICE, SKIPPING (file name)	The file specified is not transferred, but any previous or subsequent files are transferred and indicated in the new directory.
ERROR ON OUTPUT DEVICE, SKIPPING (file name)	The file specified is not transferred, but any previous or subsequent files are transferred and indicated in the new directory.
ERROR READING INPUT DIRECTORY	Self-explanatory.
ERROR READING OUTPUT DIRECTORY	Self-explanatory.
ERROR WRITING OUTPUT DIRECTORY	Self-explanatory.
ILLEGAL *	An * was entered as an embeded character in a file name, e.g., TMP*.BN.
ILLEGAL ?	A ? was entered in an output specification.
NO FILES OF THE FORM xxxx	No files of the form (xxxx) specified were found on the current input device group.
NO ROOM, SKIPPING (file name)	No space is available on the output device to perform the transfer. Predeletion may already have occurred.
SYSTEM ERROR-CLOSING FILE	Self-explanatory.
USE PIP FOR NON-FILE STRUCTURED DEVICE	An input device specified is not a file-structured device, e.g., PTR.

MAGTAPE/CASSETTE PERIPHERAL INTERCHANGE PROGRAM (MCPIP)

MCPIP is an OS/8 program that is used to transfer files between standard cassettes or magnetic tapes and other OS/8 system devices, delete such files, and transfer directories. MCPIP allows the OS/8 user to read or write any standard cassette file on a cassette or magnetic tape. In particular, MCPIP can read or write any file created by or to be used by the CAPS-8 system or by the OS/8 system (using any OS/8 device handler). MCPIP can also read or write any magnetic tape file that is in standard cassette file format, i.e., a file created by MCPIP or by CAPS-8.

MCPIP may be run on any OS/8 system equipped with at least 8K of memory and TA8E cassette or TM8E magnetic tape drives. MCPIP supports any OS/8 system device. Before running MCPIP, the user must load the OS/8 cassette or magnetic tape handlers as described in Getting On Line with OS/8 in Chapter 1.

Calling and Using MCPIP

To call MCPIP from the OS/8 system device, the user types:

```
R MCPIP
```

in response to the dot printed by the Keyboard Monitor. The Command Decoder then prints an asterisk at the left margin of the terminal and waits to receive a line of I/O files and options. MCPIP accepts one input file and performs output to a single output file. The contents of the input file are transferred to the output file in image mode. In response to the asterisk, the user types an I/O specification of the following form:

*outfile<infile/(options) = size

Each file specification consists of a device and an optional file name (for file-structured devices). To perform I/O on a given cassette drive, the user's OS/8 system should be configured with an OS/8 cassette handler for that drive.

The permanent device names for cassettes are CSA0-CSA7. Magnetic tapes have the permanent device names MTA0-MTA7. Permanent device names for other OS/8 devices are listed in the Keyboard Monitor section of Chapter 1. These device names are used in the I/O specification, along with any file name that is

necessary. For example, to transfer a CAPS-8 file named DATA01 to the disk, the user types:

```
*DSK:DATA01<CSA1:DATA01
```

if the standard cassette is mounted on drive 1 and if the user's OS/8 system has a handler for drives 0 and 1 (unit 0) with entry point names of CSA0 and CSA1. If a cassette handler is specified without any file name, MCPIP uses the handler without modification, i.e., it uses the cassette as a non-file structured device similar to a paper tape reader or punch. Thus, the command:

```
*CSA2:<DSK:SI SCO.BN
```

would perform the same operation with MCPIP as the command:

```
*CSA2:<SI SCO.BN/I
```

would perform with OS/8 PIP.

If the user specifies a magnetic tape handler with a file name, MCPIP considers the magnetic tape as a file-structured device and assumes that it has the same format as a standard cassette.

Since MCPIP performs file transfers for all file types, there are no assumed extensions assigned by MCPIP to file names for either input or output files. All extensions, where present, must be explicitly specified, except when the /B option is used.

Following completion of a MCPIP operation, the Command Decoder again prints an asterisk at the left margin and waits for another MCPIP I/O specification line. The user can return to the Keyboard Monitor by typing CTRL/C or by ending a MCPIP specification line with an ALTMODE.

MCPIP OPTIONS

The various options allowed on a MCPIP I/O specification line are detailed in Table 2-22.

Table 2-22 MCPIP Options

Option	Meaning
/B	Transfer files in special CAPS-8 binary format. If the /B option is used and no extensions are specified, MCPIP assumes .BN for OS/8 files and .BIN for cassette files. If input is from PTR: (high-speed paper tape reader), the paper tape must be positioned on the leader.
[ ]	The square bracket ([]) option allows the user to specify a decimal file type on a cassette output file. The notation in brackets does not refer to the file sizes in this case. Hence, to create a file with the name CAS50.BI on cassette drive 1 and give it a file type of 3, the user types: `*CSA1:CAS50.BI[3]<` For output files other than cassette, square brackets have the same meaning as in OS/8 PIP. For information on file types, see the *Cassette Programming System User's Manual* (DEC-8E-OCASA-B-D), Appendix E.
/D	Delete the file specified from the output cassette or magnetic tape. The /D option is only valid if the output device is a cassette or magnetic tape. For example: `*MTA1:OFILE</D` will delete OFILE from the magnetic tape on drive 1.
=n	Specify in the low order 12 bits of n the number of words (characters) per record which occur in the cassette or magnetic tape output file. The low order 12 bits of the n specification may be between 0 and 1000 (octal), inclusive. If not specified, 200 is assumed. The = option need not be specified for cassette or magnetic tape input files because MCPIP will determine the record size from the file's header record. If the output record size specified is greater than 1000 or if an input record size is 0, MCPIP prints an error message since it cannot handle variable-length records. The high order 11 bits of the = option are used to specify the version number for the file. The = option is ignored if the output file is not a cassette or magnetic tape file.

Table 2-22 MCPIP Options (Cont.)

Option	Meaning
/L	Read the input cassette or magnetic tape directory and write it onto the output file. Notice that in this case the input file itself is not transferred, only the directory. The /L option applies only if the input device is a cassette or magnetic tape.
/Z	If no filename is specified, zero the cassette or magnetic tape on the drive specified as output, by writing a sentinel file on it. Every magnetic tape or cassette should be zeroed before it is used for the first time. If a filename is specified (for a cassette or magnetic tape drive), write a sentinel file after the file specified.

Although cassette or magnetic tape file names may have 3-character extensions, OS/8 allows only 2-character extensions. Thus, when looking up a cassette file, although all three characters may be specified, only the first two are significant. For example, CSA0:FILE.PAL might match a file called FILE.PAT. All files on a standard cassette must be unique with respect to the file name and the first two characters in the extension. On output, the third character of the extension is always a space (unless the /B option is specified).

NOTE

If CTRL/C is typed while a write operation is in progress on a cassette or magnetic tape, MCPIP writes an end-of-file before returning to the Keyboard Monitor.

MCPIP Error Messages

Error messages which appear while MCPIP is running are shown in Table 2-23. If an output file is specified on a cassette or magnetic tape and a file by that name already exists, the file on the output drive is deleted before any transfer is performed. If MCPIP detects an error while a cassette or magnetic tape output file is open, it tries to close the output file by writing a sentinel file on the output cassette or magnetic tape.

Table 2-23 MCPIP Error Messages

Message	Meaning
CANNOT HANDLE VARIABLE LENGTH RECORDS	The records on the input and output files specified are not the same size. MCPIP cannot handle variable length records.
CLOSE ERROR	MCPIP is not able to close the file. A bad file just created on magnetic tape or cassette must be removed by placing a sentinel file after the preceding file. (See the /Z option.)
device DOES NOT EXIST	The device specified does not exist on the OS/8 system. "Device" is a set of four characters given when MCPIP expected an OS/8 device name such as DTA0.
ENTER ERROR	Error occurred while trying to enter an output file. This message usually means that the cassette or magnetic tape has no sentinel file.
FETCH ERROR	Error occurred while trying to fetch an OS/8 device handler.
file NOT FOUND	The file specified cannot be found. "File" is the actual name of the file that was not found.
ILLEGAL * OR ?	Wild card * or ? was specified in a MCPIP command line. MCPIP does not accept the wild card construction.
ILLEGAL SYNTAX	The command line to the Command Decoder contains an illegal character or was incorrectly formatted.
INPUT ERROR	An input error occurred while reading the file.

Table 2-23 MCPIP Error Messages (Cont.)

Message	Meaning
NO INPUT FILE	No input file was specified when one was required.
NO OUTPUT FILE	No output file was specified when one was required.
OUT-IN	Both the input and the output devices were specified as the same cassette or magnetic tape drive.
OUTPUT DEVICE FULL	Either room on device or room in the directory is lacking.
OUTPUT ERROR	Output error—possibly a WRITE LOCKed device, parity error, or attempt to output to a read-only device.
RECORD SIZE TOO BIG	The output record size specified is greater than 1000 or an input record size is 0.
TOO MANY FILES	More than 1 output device was specified or more than 1 input device was specified.

PIP10

PIP10 is an OS/8 utility program used to provide file compatability with the DECsystem-10 computer. PIP10 is capable of transferring files to and from DECsystem-10 formatted DECtapes. PIP10 provides the facilities for transferring ASCII, Image (PAL10 binary output), and sequenced ASCII (LINED output) files.

PIP10 uses an internal DECsystem-10 DECtape routine. This routine optimizes file storage in the same way that the DECsystem-10 Monitor does, thus resulting in the most efficient algorithm for block storage.

PIP10 has the following features:

- Automatically determines which of the specified DECtapes is a DECsystem-10 tape (384(10) words/blocks).
- Works interchangeably on TC08 and TD8E DECtape controllers.
- Reads and writes to DECsystem-10 tapes in both forward and reverse directions on TC08 tapes, forward only on TD8E.
- Keeps the DECsystem-10 DECtape directory in core during the file-copying operations of PIP10, thus eliminating the necessity for rereading the directory. The directories are purged from core when PIP10 reads another command line.
- Permits transfers between two OS/8 devices as well as transfers between two DECsystem-10 tapes.
- Zeroes DECsystem-10 DECtape directories, deletes DECsystem-10 files, and lists DECsystem-10 directories.

Note that PIP10 cannot be used while running the OS/8 BATCH program.

Calling and Using PIP10

To use PIP10, type:

```
.R PIP10
```

PIP10 responds with an asterisk (*) and waits to receive a command line of I/O files and options. The command line must have one output specification and may have from zero to nine input specifications. Multiple input files are merged onto the output file.

A DECsystem-10 file name may have a 0- to 3-character file extension; an OS/8 file name may have a 0- to 2-character extension.

Remember that PIP10 automatically determines which DECtape mounted is a DECsystem-10 tape. Thus no indication of that nature is necessary.

Following completion of a PIP10 operation, the PIP10 command decoder again prints an asterisk at the left margin and waits for another PIP10 I/O command line. To return to the Keyboard Monitor, type CTRL/C.

NOTE

PIP10 uses its own command decoder, not the standard OS/8; however, the command decoders are functionally the same.

PIP10 Options

The various options allowed on a PIP10 I/O command line are detailed in the following table. The general format for PIP10 command lines is the same as that for the standard OS/8 Command Decoder.

Option	Meaning
/B	Transfer files in DECsystem-10 binary mode. The output device must be a DECsystem-10 DECtape.
/D	Delete the old copy of the output file before continuing the transfer. If /D is not used, the file is copied before the old copy is deleted.
/F	List the short form of DECsystem-10 DECtape directory.
/I	Copy in Image mode (compatible with PAL10 binary files) rather than ASCII mode.
/L	List the directory of the input device. This input device must be a DECsystem-10 DECtape. If no output device is specified, TTY is assumed to be the output device.
/P	Preserve LINED sequence numbers in DECsystem-10 format. Sequence numbers are normally deleted.
/Z	Zero the output device directory. The output must be a DECsystem-10 DECtape.

PIP10 Examples

The following examples assume that a DECsystem-10 DECtape is mounted on DTA7. In an actual operation, any unit may be used since PIP10 can access any of the tape drives.

Example 1:

```
*DTA7:FILE.EXT<FILE.EX/Z
```

The command line in Example 1 zeroes the DECsystem-10 directory on DTA7 and transfers FILE.EX from DSK to the DECsystem-10 DECtape on DTA7. If /Z is not specified, the DECsystem-10 tape should always have a valid directory on it before transfers are attempted.

Example 2:

```
*DTA7:FILE.EXT<DTA1:P1,PTR: ,, DTA7:PARZ,TTY:
```

In Example 2, five input files are merged onto one DECsystem-10 output file (FILE.EXT). The first input file is an OS/8 file (P1) on DTA1; the second and third files are read from the paper tape reader; the fourth is a DECsystem-10 file named PARZ on DTA7; and the fifth is from the terminal. This example shows that input files need not be all OS/8 or all DECsystem-10.

Example 3:

```
*DTA1:FILE.BN[10]<DTA7:FILE.BIN/I
```

The command line in Example 3 copies the DECsystem-10 file (FILE.BIN) in Image mode since the DECsystem-10 file is a binary file. /I must be used to copy DECsystem-10 binaries. Note the use of square brackets [ ] in the command; they have the same meaning as in the OS/8 command decoder.

Example 4:

```
*DTA7:FILE.EXT</D
```

Example 4 indicates the deletion of a DECsystem-10 file (FILE.EXT) from a device.

Example 5:

```
*DTA7:/L
```

If DTA7 has a DECsystem-10 DECtape mounted, the command line in Example 5 will produce a directory listing of the device.

Error Messages

All errors cause PIP10 to abort the current command and print another asterisk. The command can then be entered correctly.

Message	Meaning
DEVICE FULL	DECsystem-10 ran out of space on the output file during a transfer.
ERROR DELETING FILE	The output file of a /D command was not found, or an error occurred which deleted the file.
FILE NOT FOUND	The requested file was not found on the specified device.
I/O ERROR	I/O device error, e.g., parity, write lock, out of paper.
NO SUCH DEVICE	Device name used is not legal in this OS/8 system.
NOT OS8 FILE	The output device specified with a /L or /F option was not an OS/8 device or file.
NOT PDP-10 FILE	The output device specified with a /Z option was not a DECsystem-10 tape, or the input device specified with a /L or /F option was not a DECsystem-10 tape.

Message	Meaning
OUTPUT FILE OPEN ERROR	The output file could not be opened. Check output directory to ensure that enough space exists on the device.
PIP10 CANNOT BE CHAINED TO	Self-explanatory.
SYNTAX ERROR	Invalid PIP10 command line.

RESOURCES (RESORC)

RESORC is an OS/8 program that is used to determine the device handlers present on a given OS/8 system. Other information about the handlers is available through the use of RESORC options.

Calling and Using RESORC

To call RESORC from the system device, type:

```
R RESORC
```

in response to the dot printed by the Keyboard Monitor. RESORC may also be called via the CCL command RES (see the CCL section in Chapter 1). The Command Decoder prints an asterisk at the left margin and waits to receive a line of I/O files and options. RESORC accepts up to nine input files and performs output to a single output file; options generally are placed at the end of a command string.

The output specification is the device, and optionally the file name and extension, to which the RESORC listing is sent. TTY is assumed if no output device is specified. If no file name is specified, RE is assumed. If no file name extension is specified, .LS is assumed.

The input specification may be one of three types:

A. No input specification
   If no input specification is entered, the OS/8 system device is assumed.

B. A device name only (dev:)
   If the input specification is a device name only, the device must be file-structured and is presumed to contain a valid OS/8 directory and Keyboard Monitor. The device handlers built into the system on that device are the ones listed by RESORC. These handlers are not available to the user unless he bootstraps onto the specified device (see the BOOT program in this chapter).

C. A device and a file name (dev:file.ex)
   If this type of input specification is used, the file must be what is known as a system-head file. (Such files are created by the /Y option in PIP and are copies of the system portions of devices.) If no file name extension is specified, the

extension .SY is assumed. RESORC prints the handlers in the system that were saved on the specified file. System-head files are 50 (decimal) blocks long.

RESORC Options

RESORC has three operating modes which are specified by options in the command line. These modes are:

Option	Mode
/E	Extended mode—detailed handler information
/F	Fast mode—1-line printout (default)
/L	Limited mode—3-column printout

FAST MODE (/F OPTION)

If the /F option is specified in a RESORC command line, or if no options are specified, RESORC prints the permanent device names for handlers which exist on the system. If RESORC cannot determine the ASCII device name for one of the devices, it prints the internal octal representation of the device name and encloses it in parentheses. (This octal representation is included in the *OS/8 Software Support Manual.)* For example:

```
.R RESORC
*/F
SYS, DSK, DTA2, DTA0, DTA1, (4667), TTY, LPT
```

The first two devices are always SYS and DSK. When the fast mode is used, the devices are separated by commas and listed in order of their internal device numbers.

LIMITED MODE (/L OPTION)

If the /L option is used in a RESORC command line, the handler information is printed in three columns. For example:

```
.R RESORC
*/L

128 FREE BLOCKS

 NAME TYPE USER
 SYS  RK8E
 DSK  RK8E IN
 DTA0 TC08 0
 TTY  TTY
 LPT  LPTR LPT

OS/8 V3F
```

Preceding the table of device names, RESORC prints the number of free blocks on the device. This information is not given for system-head files since it is not available.

The first column (NAME) lists the permanent names of devices on the system. The second column (TYPE) lists the physical type of the handler. Each type of device is assigned a unique number by OS/8. RESORC associates this number with a name as listed in Table 2-24. Note that physically different devices which are similar in function have the same internal type code. For example, line printers LP08, LS8E, and L645 have an internal code of 04.

The third column (USER) lists the name given to the device with the Monitor ASSIGN command. If RESORC cannot determine the name from the internal octal, it prints the octal code enclosed in parentheses.

Table 2-24 RESORC Device Types

Internal Type Code	RESORC Name	Explanation
00	TTY	Console terminal
01	PTR	Paper tape reader
02	PTP	Paper tape punch
03	CR8E	Card reader
04	LPTR	Line printer
05	RK	RK8 disk
06	RF08	RF08 disk (1 platter)
07	RF08	RF08 disk (2 platter)
10	RF08	RF08 disk (3 platter)
11	RF08	RF08 disk (4 platter)
12	DF32	DF32 disk (1 platter)
13	DF32	DF32 disk (2 platter)
14	DF32	DF32 disk (3 platter)
15	DF32	DF32 disk (4 platter)
16	TC08	TC08 DECtape
17	LINC	LINCtape
20	TM8E	Magnetic tape
21	TD8E	TD8E DECtape
22	BAT	Batch input handler
23	RK8E	RK8E disk
24	NULL	NULL handler
27	TA8E	Cassette
30	VR12	PDP-12 scope

Codes 25-26 and 31-37 are reserved for future use by Digital. Codes 40-57 are reserved for user handlers.

EXTENDED MODE (/E OPTION)

When the /E option is used in a command line, RESORC provides more detailed information about the handlers configured into the system. The /E option produces a table with the following headings.

Heading	Meaning
#	Internal device number for the handler. If a number is missing, there is no internal number for this handler.
NAME	Permanent device name for the handler. If RESORC cannot determine the name, it prints the internal coding.
TYPE	Type of device as listed in Table 2-24.
MODE	One or more of the following three letters: R The handler may be used for reading. W The handler may be used for writing. F The handler controls file-structured devices.
SIZ	The size of the device in decimal OS/8 blocks. This is only applicable for file-structured devices.
BLK	The block on the system device in which this handler resides. If this number is followed by a +, this indicates that the handler is two pages long. If this entry is SYS, the handler is permanently resident in core location 07600.
KIND	This entry tries to differentiate the handler more specifically than the TYPE column. Since several devices of the same type have the same device code, there may be several handlers for the same device. If the device type has only one handler, this entry may be blank. The KIND specification has no meaning for user-written handlers. Table 2-25 details the kinds of handlers that may be on the system.

Table 2-25 Kinds of Handlers

Kind	Type	Description	How Identified
AS33	TTY	1-page handler	by number of pages
KL8E	TTY	2-page handler	by number of pages
KS33	PTR	low-speed reader	by IOT codes
PT8E	PTR	high-speed reader	by IOT codes
KS33	PTP	low-speed punch	by IOT codes
PT8E	PTP	high-speed punch	by IOT codes
026	CR8E	DEC-026 card codes	by table codes
029	CR8E	DEC-029 card codes	by table codes
LP08	LPTR	old LP08 handler	location dependent
LS8E	LPTR	old LS8E handler	location dependent
LV8E	LPTR	LP08/LS8E/LV8E handler	location dependent
LV8E	LPTR	LPSV altered for LV8E	location dependent
L645	LPTR	Anelex line printer	location dependent

U Unit—the particular unit number of a multiple unit device handler. For example, the RK8E disk can have as many as four physical drives (0, 1, 2, 3) on an OS/8 system. OS/8 considers the disk cartridge in each drive as two logical units. The lower half is the A unit and the upper half is the B unit. Thus drive 2 consists of two logical units called A2 and B2.

Since the U column in the printout has space for only one character, RESORC numbers the logical units from 0 to 7. The following table shows the correspondence between the U printout, the logical unit, and the physical device.

U	Logical Unit	Physical Device
0	A0	0
1	B0	0
2	A1	1
3	B1	1
4	A2	2
5	B2	2
6	A3	3
7	B3	3

V	Version number (letter) of handler. No entry means the handler predates OS/8 Version 3. Version numbers are of the form A-Z. The 6-bit of the ASCII representation of the handler version letter resides in the handler's entry point location. For example, a handler with a version of A has a representation of 01. (See Appendix A for a list of the 6-bit octal codes.)
ENT	The relative entry point of the handler.
USER	Same as for /L option. Current user name for the handler as assigned by the Monitor ASSIGN command.

In addition to the preceding, the /E option also provides the following information. If a device was specified, as opposed to a system-head file, RESORC prints:

- number of files in directory
- number of blocks used
- number of segments used
- number of free blocks
- number of empties
- number of additional information words

RESORC also lists the following:

- number of free device slots
- number of free block slots
- version number of Monitor if device is a system device

```
.R RESORC
*/E

164 FILES IN 1025 BLOCKS USING 6 SEGMENTS
2167 FREE BLOCKS (14 EMPTIES)

#  NAME TYPE MODE SIZ  BLK KIND U V ENT USER
01 SYS  RK8E RWF 3248  SYS      0 B  07
02 DSK  RK8E RWF 3248  SYS      0 B  07
03 DTA0 TD8E RWF  737  16+ TD8A 0 A  10
04 DTA1 TD8E RWF  737  16+ TD8A 1 A  14
05 RKB0 RK8E RWF 3248  SYS      1 B  21
06 TTY  TTY  RW        17+ KL8E   C 176
```

```
07 PTP  PTP  W        20  PT8E   A  00
10 PTR  PTR  R        20  PT8E   A 112
11 LPT  LPTR W        21  LPSV   B  03

FREE DEVICE SLOTS: 06,  FREE BLOCK SLOTS: 04
OS/8 V3F
```

RESORC Error Messages

The following messages may appear during a RESORC operation.

Table 2-26 RESORC Error Messages

Message	Meaning
?BAD DIRECTORY	Input device directory cannot be read.
%BAD MONITOR	The input device may be a system device but the Monitor cannot be accessed.
%DEV IS NOT FILE STRUCTURED	The input device specified is not a file-structured device, e.g., PTR.
?INPUT ERROR	An input error occurred during a RESORC operation.
%NON SYSTEM DEVICE	The input device specified in a RESORC command line is not an OS/8 system device.
%NOT A SYSTEM HEAD	The file name specified is not a system-head file.
?OUTPUT DEVICE FULL	The output device specified does not have enough room to copy the RESORC file.
?OUTPUT DEVICE IS READ ONLY	The output device specified is a read-only device, e.g., PTR.
?OUTPUT ERROR	An error occurred while attempting to output during a RESORC operation.
?TTY DOES NOT EXIST	An output device was not specified in the RESORC command line and the TTY handler does not exist on the OS/8 system. See the BUILD section of this chapter for instructions on inserting TTY handlers.

SRCCOM

SRCCOM is an OS/8 utility program which compares two source files line by line and prints all their differences. Usually, the two files are different versions of a single program, in which case SRCCOM prints all the editing changes which transpired between the two versions, making it a useful debugging tool.

SRCCOM Assembly Instructions

To make SRCCOM.BN from SRCCOM.PA, type

```
.R   PAL8
*dev:SRCCOM (,dev:SRCCOM.LS) ←dev:SRCCOM
```

The listing file shown in parentheses is optional.
To make SRCCOM.SV from SRCCOM.BN, type

```
.R   ABSLDR
*dev:SRCCOM$
.SA dev SRCCOM
```

To load and save the binary papertape (DEC-S8-OSYSA-<-PB19)

```
.R ABSLDR
*PTR:$↑       (Type and character in response to ↑)
.SAVE dev SRCCOM
```

Loading SRCCOM

To use SRCCOM, type

```
.R SRCCOM
*OUTPUT<INPUT1,INPUT2
```

INPUT1 and INPUT2 are the source files to be compared and the input devices. Both files must be specified and be non-empty. If an input device is omitted, it is assumed to be DSK.

OUTPUT specifies the output file and device where the differences will be listed. If an output file name is specified, the default output device is DSK. If the output device is non-file structured, a file name is unnecessary. If output is to a file-structured device, an output file name must be specified. If no output specification exists, TTY is assumed.

The following run-time options are accepted by SRCCOM:

Table 2-27 Run-Time Options

Option	Meaning
/C	Do not count differing comment fields as a difference.
/S	Do not compare tabs and spaces when considering lines different.
/T	Convert tabs to spaces on output.
/B	Count blank lines in the comparison. A blank line is conconsidered as a carriage return only. In particular space carriage return combination under /S/B is not treated as a blank line.
/X	Like /C, but does not print comment fields on the output file.

Examples:

```
.R SRCCOM
*DSK:DIFFIL<DTA1:ORIG,DTA2:COPY
```

Compare the source files ORIG on DTA1 and COPY on DTA2, and store the differences on DSK as DIFFIL.

```
.R SRCCOM
*DIFFIL<FIRST,SECOND
```

Compare the source files FIRST and SECOND on DSK, and output the differences to DIFFIL on DSK.

```
.R SRCCOM
*LPT:<DTA1:FILE1,PTR:
↑
```

Compare source files FILE 1 on DTA1 and one from the high-speed paper tape reader, and output the differences to the line printer.

SRCCOM Output

The first line of output printed by SRCCOM is "SRCCOM Vx, where x is the current version number, then two header lines followed by as many difference groups as necessary. The header lines are printed as follows:

file 1) header line of file 1
file 2) header line of file 2

A difference group has the form:

```
1) /nnn  line 1, file 1
1)       line 2, file 1
1)       line 3, file 1
  .
  .
1)       line n, file 1
  .
****
2) /nnn  line 1, file 2
2)       line 2, file 2
  .
  .
  .
2)       line m, file 2
```

where nnn is the number of the difference group and lines 1 through n-1 of file 1 and 1 through m-1 of file 2 did not agree. SRCCOM compares areas of the two programs, and prints differences until it finds 3 lines which agree. The last lines printed (line n of file 1 and line m of file 2) are the first lines that agreed. The number of consecutive lines to check for agreement may be changed to any number (k) with the option =k in the command line.

Example:

File 1	File 2	SRCCOM OUTPUT
		file 1) A
A	A	file 2) A
B	X	1) B
C	C	1) C
D	D	****
E	E	2) X
F	G	2) C
G	H	********
H	J	1) F
I		1) G
J		1) H
		1) I
		1) J
		****
		2) G
		2) H
		2) J

Occasionally a decimal number appears following the close parenthesis after the file number. This decimal number indicates the source page in this file from which this line and all following lines (until the next such number) come.

If the two files are identical, SRCCOM prints the message:

NO DIFFERENCES

in the output file.

Error Messages

SRCCOM error messages are of the form:

?n

where n is a single digit. The meaning of the various digits are

?0 Insufficient core; this means that the differences between the files are too large to allow for effective comparison. Use of the /X option may alleviate this problem.

?1 Input error on file #1 or less than 2 input files specified.

?2 Input error on file #2.

?3 Output file too large for output device.

?4 Output error.

?5 Could not create output file.

TECO

Introduction

OS/8 TECO is a powerful text editing and correcting program that runs under the OS/8 operating system. OS/8 TECO may be used to edit any form of ASCII text such as program listings, manuscripts, correspondence and the like. Since OS/8 TECO is a character-oriented editor rather than a line editor, text edited with OS/8 TECO does not have line numbers associated with it, nor is it necessary to replace an entire line of text in order to change one character.

Because OS/8 TECO is very versatile, it is necessarily complex. This chapter is, therefore, divided into two parts. The first part contains basic information and introduces enough OS/8 TECO commands to allow the novice OS/8 TECO user to begin creating and editing text files after only a few hours of instruction. The introductory commands are sufficient for any editing application; however, they are less convenient, in most cases, than the advanced commands presented later.

The second part introduces the full OS/8 TECO command set, including a review of the introductory commands presented earlier. This part also introduces the concept of OS/8 TECO as a programming language and explains how basic editing commands may be combined into editing "programs" which are sophisticated enough to handle the most complicated editing tasks.

Specific examples of the use of OS/8 TECO commands have been de-emphasized throughout this manual. This was done because all of the OS/8 TECO commands have a consistent, logical format which will quickly become apparent to the novice user. However, each section of the chapter is concluded with one or more elaborate examples which employ most of the commands introduced up to that point. Users who are learning the TECO commands should experiment with each command as it is introduced, then duplicate the examples on their computer. Hereafter, OS/8 TECO will be referred to as simply TECO.

Introductory Commands

TECO considers text to be any string of ASCII codes. Text is broken down into units of pages, lines and characters. A page of text consists of all the ASCII codes between two form feed

characters, including the second form feed. A line of text consists of all the ASCII codes between two line feeds, including the second line feed. A character is one ASCII code. Thus, every page of text contains one form feed character, which is the last character on the page. Every line of text contains one line feed, which is the last character on the line.

TECO maintains a text buffer in which text is stored. The buffer usually contains one page of text consisting of up to 4000 characters, but the terminating form feed character never appears in the buffer. TECO also maintains a buffer pointer. The pointer is simply a movable position indicator which is always located between two characters in the buffer, before the first character in the buffer, or after the last character. The pointer is never located on a character.

Line feed and form feed characters are inserted automatically by TECO. A line feed is automatically appended to every carriage return entered into the buffer, and a form feed is appended to the content of the buffer by certain output commands. Additional line feed and form feed characters may be entered into the buffer as text. If a form feed character is entered into the buffer, it will cause a page break upon output. That is, all text preceding the form feed will appear on one page, and the text following the form feed will appear on the next page.

Finally, TECO also maintains an input file and an output file, both of which are selected by the user through use of file specification commands. The input file is any device except the keyboard from which text may be accepted. For example, if a block of text is stored on paper tape, the paper tape reader would be specified as an input device when the tape is edited.

The output file is any device except the user terminal on which edited text may be written. If the paper tape file mentioned above were to be edited and written onto DECtape, for example, the output file would be a user-named DECtape file (with optional file extion) on a specified DECtape transport unit.

If TECO resides on the system device it may be called from the keyboard by typing:

R TECO

(terminated with a carriage return) in response to the dot generated by the OS/8 monitor. TECO will respond by printing an asterisk at the left margin to indicate that it is ready to accept user

commands. At this point, one or more commands may be typed at the keyboard, and TECO will execute the commands upon receipt of two consecutive ALT MODE characters. The ALT MODE is a non-printing character which may be labelled ESCAPE on some keyboards. TECO echoes a dollar sign ($) whenever an ALT MODE is received.

A TECO command consists of one or two characters which cause a specific operation to be performed. Some TECO commands may be preceded or followed by arguments. Arguments may be either numeric or textual. A numeric argument is simply an integer value which might be used to indicate such things as the number of times a command should be executed. A text argument is a string of ASCII characters which might be words of text, for example, or the OS/8 designation of a storage file.

If a command requires a numeric argument, the numeric argument always precedes the command. If a command requires a text argument, the text argument always follows the command. All text arguments are terminated by a special character (usually an ALT MODE) which indicates to TECO that the next character typed will be the first character of a new command.

If more than one command is typed in response to the asterisk generated by TECO, the command string will be executed from left to right until either all commands have been executed or a command error is recognized. When an error is encountered, a message is printed and the rest of the command string is ignored. In any case, TECO prints another asterisk at the left margin as soon as it finishes execution of a command string, so that additional commands may be entered.

The extensive text editing capability of TECO implies a large and versatile command set. However, the novice TECO user will find that little more than a dozen basic commands suffice for most editing requirements. The following section introduces the basic TECO commands. The full command set will be described later in this chapter.

TECO will accept input text from any input device in the OS/8 system. If input is supplied from any device except the keyboard, the input device must be specified by means of an ER command terminated by an ALT MODE. If the input device is a file-structured device such as disk or DECtape, the file name and extension

(if any) should also be supplied. If a file name is specified but no device is explicity defined, the OS/8 default device is assumed. The ER command causes TECO to search for the specified file and print an error message if the file is not found. This command does not cause any portion of the file to be read into the text buffer, however. The following examples illustrate use of the ER command.

Command	Function
ERdev:filnam.ex$	General form of the ER command where "dev: filnam.ex" is the OS/8 designation of the input file. The command is terminated by an ALT MODE, which echoes as a dollar sign.
ERPTR:$	Prepare to read an input file from the reader.
ERPROG.PA$	Prepare to read input file PROG.PA from the OS/8 default device DSK.
ERDTA1:PROG$	Prepare to read input file PROG from DTA1.

TECO will write output text onto any device in the OS/8 system. If output is written onto any device except the user terminal, the output device must be specified by means of an EW command terminated by an ALT MODE. If the output device is a file-structured device, a file name and extension (if any) must also be supplied. If a file name is specified but no device is explicitly defined, the OS/8 default device is assumed. The following examples illustrate use of the EW command, which has the same format as the ER command.

Command	Function
EWdev:filnam.ex$	General form of the EW command where "dev: filnam.ex" is the OS/8 designation of the output file. The command is terminated by an ALT MODE, which echoes as a dollar sign.
EWSYS:<TEXT.LS$	Prepare to write output file TEXT.LS on the system device.
EWDSK:PROG$	Prepare to write output file PROG on the OS/8 default device DSK.
EWTEXT.AS$	Prepare to write output file TEXT.AS on the OS/8 default device DSK.

It is not always necessary to specify an input file. If the user desires to create a file without using any previously edited text as

input, he may insert the necessary text directly into the text buffer from the keyboard and, at the end of each page, write the content of the buffer onto an output file. Since all input is supplied from the keyboard, no input file is necessary.

An output file is unnecessary if the user desires only to examine an input file, without making permanent changes or corrections. In this case, the content of the input file may be read into the text buffer page by page and examined at the terminal. Since all output is printed on the user terminal, no output file is needed.

TECO will only keep one input file and one output file open at a time. The current input file may be changed by simply using the ER command to specify a new input file. Before an output file may be changed, it is essential that the current output file be closed by means of an EF command (or one of the other file closing commands presented later). If this is not done, the content of the file may be lost. The EF command is presented below, along with several examples of file specification command strings.

EF
Close the current output file. If the last EW command specified a directory device file which already existed on the device (so that TECO created a second file with the same file name and extension), the old version of the file is deleted at this time.

ERDTA1:INPUT.TX$EWDTA2:OUTPUT.TX$$
Open an input file "INPUT.TX" to be found on DECtape unit 1 and an output file "OUTPUT.TX" on DECtape unit 2. If the file OUTPUT.TX does not already exist, it will be created. The double ALT MODE ($$) terminates the command string and causes the string to be executed.

EFEWTEXT.AS$$
Close the current output file and open an output file "TEXT.AS" on the OS/8 default device. Note that the ALT MODE which terminates the EW command may be one of the two ALT MODEs which terminate the command string.

ERPTR:$EFEWSYS:FILE$$
Read the input file from the high-speed paper tape reader, then close the current output file and open "FILE" on the system device as an output file.

The following commands permit pages of text to be read into the TECO text buffer from an input device or written from the buffer

onto an output device. Once a page of text has been written onto the output file, it cannot be recalled into the text buffer unless the output file is closed and then opened as an input file.

Command	Function
Y	Clear the text buffer, then read the next page of the input file into the buffer.
P	Write the content of the text buffer onto the next page of the output file, then clear the buffer and read the next page of the input file into the buffer.
nP	Execute the P command n times, where n must be an integer in the range $0 \leqslant n \leqslant 4095$. If n is not specified, a value of 1 is assumed.

The buffer pointer provides the only means of specifying the location within a block of text at which insertions, deletions or corrections are to be made. The following commands permit the buffer pointer to be moved to a position between any two adjacent characters in the buffer. TECO positions the pointer before the first character in the buffer after every Y or P command.

Command	Function
L	Move the pointer forward to a position between the next line feed and the first character on the next line. That is, advance the pointer to the beginning of the next line.
nL	Execute the L command n times, where n may be any integer. A positive value of n moves the pointer to the beginning of the nth line following the current pointer position. A negative value moves the pointer backward n lines and positions it at the beginning of the nth line preceding the current position. If n is zero, the pointer is moved back to the beginning of the line on which it is currently positioned.
C	Advance the pointer forward across one character.
nC	Execute the C command n times, where n must be an integer in the range $-2048 \leqslant n \leqslant 2047$. A positive value of n moves the pointer forward across n characters (carriage return/line feed counts two characters). A negative value of n moves the pointer backward across n characters. If n is zero, the pointer position is not changed.

These commands may be used to move the buffer pointer across any number of lines or characters in either directions, however they

will not move the pointer across a page boundary. If a C command attempts to move the pointer backward beyond the beginning of the buffer or forward past the end of the buffer, an error message is printed and the command is ignored.

If an L command attempts to exceed the page boundaries in this manner, the pointer is positioned at the boundary which would have been exceeded. Thus, the command "-2000L" would position the pointer before the first character in the buffer. The command "2000L" would position the pointer after the last character in the buffer. No error message is printed in either case.

The following commands permit portions of the text in the buffer to be printed out for examination. These commands do not move the buffer pointer.

Command	Function
T	Type the content of the text buffer from the current position of the pointer through and including the next line feed character.
nT	Execute the T command n times, where n must be an integer in the range $-2048 \leqslant n \leqslant 2047$. A positive value of n causes the n lines following the pointer to be typed. A negative value of n causes the n lines preceding the pointer to be typed. If n is zero, the content of the buffer from the beginning of the line on which the pointer is located up to the pointer is typed. This facilitates locating the buffer pointer.
HT	Type the entire content of the text buffer.

The 0T command is particularly useful for determining the position of the buffer pointer. This command should be used frequently to determine that the pointer is actually located where the user expects it to be.

The following commands permit the user to insert or delete text from the buffer.

Command	Function
Itext$	Where "text" is a string of ASCII characters terminated by an ALT MODE, which echoes as a dollar sign. The specified text is inserted into the buffer at the current position of the pointer, with the pointer positioned immediately after the last character of the insertion. Insertion commands should be limited to a maximum length of 10 to 15 lines.

Command	Function
K	Delete the content of the text buffer from the current position of the pointer through and including the next line feed character.
nK	Execute the K command n times, where n may be any integer. A positive value of n causes the n lines following the pointer to be deleted. A negative value of n causes the n lines preceding the pointer to be deleted. If n is zero, the content of the buffer from the beginning of the line on which the pointer is located up to the pointer is deleted.
HK	Delete the entire content of the text buffer.
D	Delete the character following the buffer pointer.
nD	Execute the D command n times, where n may be any integer. A positive value of n causes the n characters following the pointer to be deleted. A negative value of n causes the n characters preceding the pointer to be deleted. If n is zero, the command is ignored.

Like the L command, D and K commands may not execute across page boundaries. If any D or K command attempts to delete text up to and across the beginning or end of the buffer, text will be deleted only up to the buffer boundary and the pointer will be positioned at the boundary. No error message is printed.

The following commands may be used to search for a specified string of characters which may occur somewhere in the input file. They cause the buffer pointer to be positioned immediately after the last character in the specified string, if it is found.

Command	Function
Stext$	Where "text" is a string of from 1 to 31 ASCII characters terminated with an ALT MODE, which echoes as a dollar sign. This command searches the text buffer for the next occurrence of the specified character string following the current pointer position. If the string is found, the pointer is positioned after the last character in the string. If it is not found, the pointer is positioned immediately before the first character in the buffer and an error message is printed.
Ntext$	Performs the same function as the S command except that the search is continued across page boundaries, if necessary, until the character string is found or the end of the input file is reached. If the end of the input file is reached, an error message is printed and it is necessary to close the output file and reopen it as an input file before any further editing commands may be executed.

Both the S command and the N command begin searching for the specified character string at the current position of the pointer. Therefore, neither command will locate any occurrence of the character string which precedes the current pointer position, nor will it locate any character string which continues across a page boundary.

Both commands execute the search by attempting to match the command argument character for character with some portion of the buffer contents. If an N command reaches the end of the buffer without finding a match for its text argument, it writes the content of the buffer onto the output file, clears the buffer, reads the next page of the input file into the buffer, and continues the search.

At this point, all of the basic TECO commands have been introduced. Recall that TECO indicates it is ready to accept user commands by printing an asterisk (*). Once TECO has printed an asterisk, one or more commands may be typed at the terminal. Errors may be corrected by typing the RUBOUT key to delete characters. Each depression of the RUBOUT key deletes one character, beginning with the last character typed, and then prints the deleted character at the terminal. An entire command string may be deleted in this manner, if necessary. Once the correct command(s) have been entered, typing a double ALT MODE ($$) causes TECO to execute the command(s) in the order they were entered, and print another asterisk so that additional commands may be typed.

If TECO encounters an erroneous command, it prints an error message and ignores the erroneous command as well as all commands which follow it. All error messages are of the form:

?n

where n is a number which references the list of error codes that appears at the end of this chapter. Every error message is followed by an asterisk at the left margin, indicating that TECO is ready to accept additional commands. If the first command entered after a TECO-generated error message is a single question mark character (?), TECO will print the erroneous command string up to and including the character which caused the error message. This facilitates locating errors in long command strings and determining how much of a command string was executed before the error was encountered.

At the conclusion of an editing job, users may type control-C to exit TECO and return to the keyboard. Control-C may be typed at any time during an editing run; it will cause an immediate exit to the monitor as soon as it is recognized by TECO. Control-C should not be typed while any output file is open.

The following example illustrates how TECO may be used to create an OS/8 FORTRAN program for immediate execution. The same procedure may be employed to create and execute programs under PAL8, SABR, and on so.

```
T-4:

.R TECO
*EWATEST.FT$$
*I      WRITE(1,1)
1       FORMAT('  COMPILER TEST')
        RJ=3
        RK=7
        X=.5
        RII=RK/RJ*(2-RK*RK/(3.*RJ))
        R=10.6
        S=3.5
        RI=5
        RJ=2
        RN=7
        Z=R+S*RI/RJ*RN/3
        WRITE(1,2)RII,Z
1       FORMAT(F10.2,F12.5)
        END
$PEF$$
*↑C
.R F4
*ATEST/A
US  2
ML  0017
.R TECO
*ERATEST.FT$EWBTEST.FT$Y$$
*SF10.$0LDI2$0LT$$
2       FORMAT(F10.2,F12.5)
*PEF$$
*↑C
.R F4
*BTEST.FT
 COMPILER TEST
     -8.04     31.01667
```

User calls TECO, specifies output file, and creates FORTRAN source program.

He then closes the file and exits to the monitor.

User calls FORTRAN and executes test program.

FORTRAN lists two errors, so user calls TECO and edits corrections into new output file.

User finally calls FORTRAN and successfully executes the text program.

The remainder of this chapter is devoted to a detailed description of the full TECO command set. It is assumed that the reader is familiar with the elementary TECO commands presented earlier.

TECO Character Set

TECO accepts the full ASCII character set, which is presented in Appendix A. Most terminals will not transmit and receive all of the ASCII codes; however, characters that are not available on the user's terminal may be inserted into the TECO text buffer by means of special commands which will be presented later in this section.

TECO command strings may be entered by using upper case characters, as indicated throughout this chapter, or by using the corresponding lower case characters. A file which contains upper and lower case text may be edited in the same manner as a file which contains only upper case text. If such file is edited from a terminal that does not accept lower case characters, all characters will be printed at the terminal as their upper case equivalents.

TECO considers certain ASCII characters to be special characters. Most of the special characters are immediate action commands. Typing these characters in a command string causes TECO to perform a specified function immediately, without waiting for the double ALT MODE which terminates the command string. Immediate action commands may be entered at any point in a command string—even in the middle of a command or text argument. For this reason, the special characters should not be used in text arguments, except where specifically indicated throughout this chapter.

Table 2-28 lists the special characters, their functions and the restrictions associated with each character.

Table 2-28 Restrictions on Special Characters

Character	Restriction
ALT MODE	The ALT MODE character is a command terminator. It may not be used in the argument of any command except where noted specifically throughout this chapter. TECO echoes a dollar sign when an ALT MODE is received. ALT MODE may be labelled ESCAPE on some terminals.
RUBOUT	Typing a RUBOUT character causes the last character typed to be deleted. Typing consecutive RUBOUTs deletes one character for each RUBOUT typed, beginning with the last character typed. TECO echoes the deleted character whenever a RUBOUT is typed.

Table 2-28 Restrictions on Special Characters (Cont.)

Character	Restriction
CTRL/U	CTRL/U causes the current line to be deleted, and eches a ↑U and a carriage return/line feed.
CTRL/C	CTRL/C causes an immediate exit from TECO to the OS/8 Keyboard Monitor. If an output file is open when the CTRL/C command is executed, the contents of the file will be lost.
CTRL/P	CTRL/P causes an immediate branch to the starting address of TECO.
CTRL/G	Typing two consecutive CTRL/G characters causes all commands which have been entered but not executed to be erased. (If the terminal has a bell, it will ring.) This command is used to erase an entire command string.
CTRL/S	If CONTROL/S is typed as the first character of a new command string, the entire previous command string, even if it was in error, is saved as a text string in Q-Register Z. The previous contents, if any, of Q-Register Z are destroyed.

The CTRL/Z character is used as an end-of-file terminator. Inserting this character into a file may cause the file to be terminated prematurely the next time it is read as an input file.

TECO also attaches special significance to the carriage return, line feed, space and null characters. A line feed is appended to every carriage return entered into the text buffer. Thus, it is necessary to type a carriage return and then a RUBOUT in order to enter a carriage return character which is not followed by a line feed.

Carriage return, line feed and space characters are ignored between commands in a command string; they may be inserted for clarity or convenience whenever necessary. The null character (CTRL-shift-P) is ignored by all TECO input commands.

Control characters which are not special characters (i.e. immediate action commands) may be included in the text argument of any TECO command. When used in this manner, the control character must be produced by striking the CONTROL key and a character key simultaneously. TECO will echo an uparrow followed by the

character which was typed whenever most control characters are entered; however, some control characters do not echo, while others, such as CTRL/L (form feed) or CTRL/G (bell) echo as the function they perform.

Many control characters are also TECO commands. When a control character is entered as a command, it may be produced by striking the CONTROL key and the character key simultaneously or else by typing an uparrow followed by the desired character. This is advantageous because all control characters echo normally when typed in the uparrow/character format.

File Specification Commands

An input file must be specified whenever TECO is requested to accept text from any source except the keyboard. An output file must be specified whenever a permanent change is made to the input file. Input and output files are selected by means of file specification commands, which always include the OS/8 designation of the input or output device. If the device is a directory device, the file specification command also includes a file name and extension. If a file extension is not explicitly defined, the null extension is assumed. If a file name is specified but no device is explicitly defined, the OS/8 default device is assumed.

Almost every editing job begins with at least one file specification command. Additional file specification commands may be executed during an editing job whenever required; however, TECO will only keep one input file and one output file active at a time, and the same file may not be used for both input and output. When an output file is opened on a directory device, it is essential that the file be closed by a TECO file closing command before any other output file is opened. If this is not done, the content of the file will be lost.

Note that a bulk storage (directory device) input file must be a file that presently exists on the system. A bulk storage output file may be a file which presently exists, in which case TECO will create a second file with same name and extension, then delete the original file when the file is closed. It may also be a nonexistent file, in which case TECO will create the specified file. Table 2-29 lists the full file specification command set.

Table 2-29 File Specification Commands

Command	Function
ERdev:filnam.ex$	Opens a file for input where "dev:filnam.ex" is the OS/8 file designation and "$" signifies an ALT MODE.
EWdev:filnam.ex$	Opens a file for output where "dev:filnam.ex" is the OS/8 file designation and "$" signifies an ALT MODE.
EBdev:filnam.ex$	The EB command may be used for directory device files only. It opens file "dev:filnam.ex" for input and file dev:filnam.BK for output then, upon receiving any file closing command, switches the file names before closing the files. Thus, dev:filnam.ex is always the current, updated file and "dev:filnam.BK" is the previous version of the file, which may be retained as a backup file.
EF	Closes the current output file.
EC	Moves the remainder of the current input file to the current output file, then closes the output file.
EG	Performs the same function as the EC command, but then transfers control to the OS/8 CCL Processor to re-execute the most recently typed CCL command of the group: PAL, COMPILE, EXECUTE, and LOAD. This allows the user to go from TECO to a compiler and then to execution of a program without returning to the OS/8 Keyboard Monitor.
EX	Performs the same function as the EC command, but then returns control to the OS/8 monitor.
CTRL/G	Typing CTRL/G causes an exit to the OS/8 monitor as soon as all previous commands have been executed. It is equivalent to the uparrow form of CTRL/C.

Many editing jobs are most conveniently accomplished by using the EB command to open the designated input file and backup file, then terminating the job with either an EC command, which returns control to TECO, or an EX command, which returns control to the OS/8 monitor. Note that once a directory device output file has been opened with an EW or EB command, it must be closed

with an EF, EC, EG or EX command or else the content of the file will be lost.

Page Manipulation Commands

The following commands permit whole pages of text to be read into the text buffer from an input file or written from the buffer onto an output file.

Table 2-30 Page Manipulation Commands

Command	Function
A	Appends the next page of the input file to the current content of the text buffer, thus combining the two pages of text on a single page with no intervening form feed character.
Y	Clears the text buffer and then reads the next page of the input file into the buffer.
PW	Writes the content of the buffer onto the output file and appends a form feed character. The buffer is not cleared and the pointer position remains unchanged.
nPW	Executes the PW command n times, where n must be an integer in the range $0 \leqslant n \leqslant 4095$.
m,nPW	Writes the content of the buffer from the m+1th character through and including the nth character onto the output file. M and n must be integers in the range $0 \leqslant n \leqslant 4095$ and m should be less than n. A form feed is not appended to this output, nor is the buffer cleared. The pointer position remains unchanged.
HPW	Equivalent to the PW command except that a form feed is not appended to the output.
P	Writes the content of the buffer onto the output file, appending a form feed, then clears the buffer and reads the next page of the input file into the buffer.
nP	Executes the P command n times, where n must be an integer in the range $0 \leqslant n \leqslant 4095$.
m,nP	Equivalent to m,nPW.
HP	Equivalent to HPW.

All of the input commands listed in Table 2-30 assume that the input file is organized into pages of less than 3800 characters each. If any page of the input file contains more than 3800 characters, the input commands will continue reading characters into the buffer

until either the first line feed following the 3800th character is read or the 4000th character is read, whichever comes first. Special techniques for handling files with pages in excess of 4000 characters in length will be developed later in this section.

Buffer Pointer Manipulation Commands

Table 2-31 summarizes the complete buffer pointer manipulation command set. These commands may be used to move the pointer to a position between any two characters in the buffer, but they will not move the pointer across either buffer boundary. If any R or C command attempts to move the pointed backward past the beginning of the buffer or forward past the end of the buffer, the command is ignored and an error message is printed. If any L command attempts to exceed the buffer boundaries in this manner, the pointer is positioned at the boundary which would have been exceeded. No error message is printed.

Table 2-31 Buffer Pointer Manipulation Commands

Command	Function
J	Moves the pointer to a position immediately preceding the first character in the buffer.
nJ	Moves the pointer to a position immediately following the nth character in the buffer. N must be an integer in the range $0 \leqslant n \leqslant 4095$.
ZJ	Moves the pointer to a position immediately following the last character in the buffer.
C	Advances the pointer forward across one character.
nC	Executes the C command n times, where n must be an integer in the range $-2048 \leqslant n \leqslant 2047$. If n is positive, the pointer is moved forward across n characters. If n is negative, the pointer is moved backward across n characters. If n is zero, the pointer position is not changed.
−C	Equivalent to −1C.
R	Moves the pointer backward across one character.
nR	Executes the R command n times, where n is an integer in the range $-2048 \leqslant n \leqslant 2047$. If n is positive, the pointer is moved backward across n characters. If n is negative, the pointer is moved forward across n characters. If n is zero, the pointer position is not changed.

Table 2-31 Buffer Pointer Manipulation Commands (Cont.)

Command	Function
−R	Equivalent to −1R.
L	Advances the pointer forward across the next line feed and positions it at the beginning of the next line.
nL	Executes the L command n times, where n is an integer in the range $-2048 \leqslant n \leqslant 2047$. A positive value of n advances the pointer to the beginning of the nth line following its current position. A negative value of n moves the pointer backwards to the beginning of the nth line preceding its present position. If n is zero, the pointer is moved to the beginning of the line on which it is currently positioned.
−L	Equivalent to −1L.

Text Type-Out Commands

Table 2-32 summarizes the commands which may be used to type out part or all of the content of the buffer for examination. These commands do not move the buffer pointer.

Table 2-32 Text Type-Out Commands

Command	Function
T	Types out the content of the buffer from the current position of the buffer pointer through and including the next line feed character.
nT	Executes the T command n times, where n is an integer in the range $-2048 \leqslant n \leqslant 2047$. If n is positive, the n lines following the current position of the pointer are typed. If n is negative, the n lines preceding the pointer are typed. If n is zero, the content of the buffer from the beginning of the line on which the pointer is located up to the pointer is typed.
−T	Equivalent to −1T,
m,nT	Types out the content of the buffer from the m+1th character through and including the nth character in the buffer. M and n must be integers in the range $0 \leqslant n \leqslant 4095$, and m should be less than n.
.,.+nT	Types out the n characters immediately following the buffer pointer. N should be greater than zero.
.−n,.T	Types the n characters immediately preceding the buffer pointer. N should be greater than zero (i.e. −n should be less than zero).
HT	Types out the entire content of the buffer.

Users may stop the execution of any T command by typing CTRL/O at the keyboard. Typing CTRL/O terminates execution of the current T command, causes all subsequent T commands to be ignored while the rest of the current command string is executed. When used in this manner, the CTRL/O must be entered while TECO is actually in the process of typing out text at the terminal.

Deletion Commands

Table 2-33 summarizes the text deletion commands, which permit deletion of single characters, groups of adjacent characters, single lines or groups of adjacent lines.

Table 2-33 Text Deletion Commands

Command	Function
D	Delete the first character following the current position of the buffer pointer.
nD	Execute the D command n times, where n is an integer in the range $-2048 \leqslant n \leqslant 2047$. If n is positive, the n characters following the current pointer position are deleted. If n is negative, the n characters preceding the current pointer position are deleted. If n is zero, the command is ignored.
−D	Equivalent to −1D.
K	Deletes the content of the buffer from the current position of the buffer pointer through and including the next line feed character.
nK	Executes the K command n times, where n is an integer in the range $-2048 \leqslant n \leqslant 2047$. If n is positive, the n lines following the current pointer position are deleted. If n is negative, the n lines preceding the current pointer position are deleted. If n is zero, the content of the buffer from the beginning of the line on which the pointer is located up to the pointer is deleted.
−K	Equivalent to −1K.
m,nK	Deletes the content of the buffer from the m+1th character through and including the nth character. M and n must be integers in the range $0 \leqslant n \leqslant 4095$, and m should be less than n.
HK	Deletes the entire contents of the buffer.

Insertion Commands

Table 2-34 lists the full text insertion command set. All text insertion commands cause the string of characters specified in the command to be inserted into the text buffer at the current position of the buffer pointer. Following execution of an insertion command, the pointer will be positioned immediately after the last character of the insertion.

The length of an insertion command is limited primarily by the amount of core available for command string storage. During normal editing jobs, it is most convenient to limit insertions to about 10 or 15 lines each. If a very long insertion command begins to exceed the TECO command storage capacity, TECO will ring the terminal bell once when ten characters of storage remain and once after each additional character that is entered. When this occurs, the command string should be terminated immediately. Attempting to enter more than 10 additional characters into the current command string causes a fatal error.

With the exception of the nI$ command, insertion command arguments may contain any ASCII characters that are not special characters. The nI$ command will insert any character into the buffer, including the special characters.

Table 2-34 Text Insertion Commands

Command	Function
Itext$	Where "text" is a string of ASCII characters terminated by an ALT MODE, which echoes as a dollar sign. The specified text string is entered into the buffer at the current position of the pointer, with the pointer positioned immediately after the last character of the insertion.
nI$	Where n is any ASCII code. This form of the I command inserts the single character whose ASCII code is n into the buffer at the current position of the buffer pointer. It may be used to insert characters that are not available on the user's terminal or special characters such as RUBOUT which may not be inserted with the standard I command.
<I>text$	Where <I> is a tabulation, produced by pressing the CONTROL key and the I key simultaneously. The TAB character echoes as from one to eight spaces on most terminals. This command is equivalent to the I command except that the tabulation is also inserted into the buffer immediately preceding the specified text string.

Table 2-34 Text Insertion Commands (Cont.)

Command	Function
@I/text/	Equivalent to the I command except that the text to be inserted may contain ALT MODE characters as long as it does not contain two consecutive ALT MODEs. A delimiting character (shown as a slash here) must precede and follow the text to be inserted. This delimiter may be any character which does not appear in the insertion except for the special characters.

Search Commands

In many cases, the easiest way to position the buffer pointer is by means of a character string search. The search commands cause TECO to scan through text until a specified string of characters, from 1 to 31 characters in length, is found and then position the buffer pointer, at the end of the string. A character string search always begins at the current position of the pointer and proceeds in the forward direction.

Table 2-35 Search Commands

Command	Function
Stext$	Where "text" is a string of 1 to 31 characters terminated by an ALT MODE. This command searches the text buffer for the next occurrence of the specified character string following the current position of the buffer pointer. If it is not found, the pointer is positioned immedately before the first character in the buffer and an error message is printed.
FStext1$text2$	The FS command is used to search for a character string within the current editing buffer (function of the S command) and replace it with another string. If the string to be replaced is not found after the current position of the buffer pointer and before the end of the buffer, the search fails and no replacement is made. Text1 is the string to be deleted and text2 is the string to be inserted in its place. If text2 is omitted, text1 is deleted without any string replacing it. However, even when text2 is omitted, its terminating ALT MODE must be present.

Table 2-35 Search Commands (Cont.)

Command	Function
FNtext1$text2$	The FN command is used to search for a character string in a page of the input file which may not yet have been read into the buffer (function of the N command) and replace it with another string. If the search fails no replacement occurs. Text1 is the string to be deleted and text2 is the string to be inserted in its place. If text2 is omitted, text1 is deleted without any string replacing it. However, even when text2 is omitted, its terminating ALT MODE must be present.
←text$	The backarrow command is identical to the N command except that the search is continued across page boundaries by executing effective Y commands instead of P or HPY commands, so that no output is generated.
↑Rtext1$text2$	The 1↑R command is identical to the FS command, and is included only for compatibility with older versions of OS/8 TECO.

If a search command is entered without a text argument, TECO will execute the search command as though it had been entered with the same character string argument as the last search command executed. For example, suppose the command "STHE END$" results in an error message, indicating that character string "THE END" was not found on the current page. Entering the command "N$" causes TECO to execute an N search for the same character string. Although the text argument may be omitted, the command terminator (ALT MODE, in this case) must always be entered.

Any of the TECO search commands may be preceded by the number n, in which case TECO will search for the nth occurrence of the specified text string.

Any of the search commands listed above may be preceded by a colon (:). The colon is a search command modifier which suppresses error message generation and causes the next sequential command to be executed with an argument of zero, if the search fails. If the search succeeds, the next sequential command is executed with an argument of -1. If the next sequential command belongs to the class of commands which require a positive argument ($0<n<4095$), the -1 is interpreted as a positive 4095. If the next

sequential command does not require an argument, it is excuted as it stands. The following examples illustrate use of the colon modifier.

COMMANDS:

```
:Stext$
:nStext$
:Ntext$
:nNtext$
:←text$
:n←text$
:FStext1$text2$
:nFStext1$text2$
:FNtext1$text2$
:nFNtext1$text2$
```

FUNCTION: In each case, execute the search command. If the search is successful, execute the next sequential command with an argument of −1 (or 4095, if it is a command which must have a positive argument). If the search fails, execute the next command with an argument of zero. If the next command does not require a numeric argument, execute it as it stands.

The @ character is another search command modifier. Inserting an @ character between the numeric argument of any search command and the command itself causes TECO to accept the first character following the command as a delimiting character which will will also be the command terminator. This character may be any character which does not appear in the search command argument, except for the special characters. When the @ command modifier is used, search command arguments may contain ALT MODE characters, as long as they do not contain two consecutive ALT MODEs. The following examples illustrate use of the @ command modifier.

COMMANDS:

```
@S/text/
n@S/text/
@N/text/
n@N/text/
@←/text/
n@←/text/
@FS/text1/text2/
n@FS/text1/text2/
@FN/text1/text2/
n@FN/text1/text2/
```

FUNCTION: In each case, execute the search command with text string "text" as an argument. This argument must be preceded and followed by a delimiting character which does not appear in the argument (a slash is shown here). The search command argument may contain ALT MODE characters, as long as it does not contain two consecutive ALT MODEs.

Match Control Characters

TECO executes a search command by attempting to match the search command argument character-for-character with some portion of the input file.

There are four special control characters that may be used in search command arguments. These characters alter the usual matching process that occurs when a search is executed. TECO considers match control characters to be single characters (they echo as two characters) and counts them as one of the maximum 31 characters in the search command argument. Table 2-36 lists the match control characters and their functions.

Table 2-36 Match Control Characters

Character	Function
CTRL/X	A CTRL/X character indicates that this position in the character string is unimportant. TECO accepts every other character as a match for control-X.
CTRL/S	A CTRL/S character indicates that any separator character is acceptable in this position. TECO accepts any character that is not a letter (upper or lower case A to Z) or a digit (0 to 9) as a match for CTRL/S.
CTRL/N	The CTRL/N character and the character following it are treated as a single character in the search command argument, but counted as 2 of the maximum 31 characters. TECO accepts any character as a match for the control-N/character combination EXCEPT the character which follows the CTRL/N. The combination CTRL/N/CTRL/S is legal; TECO accepts any character which is not a separator as a match for ↑N↑S.
CTRL/Q	A CTRL/Q character in a search command argument indicates that the character following the CTRL/Q is to be interpreted literally rather than as a command. This character may be used to search for ↑X, ↑N, and ↑S characters. It does not count as one of the maximum 31 characters in the search command argument.

As with all other control characters entered into text arguments, match control characters must be typed by holding the CONTROL key depressed while striking the character key. The uparrow construction may not be used.

Command Loops

The user may cause a command string to be executed any number of times by placing the command string within angle brackets and preceding the brackets with a numeric argument which designates the number of iterations. Iterated command strings are called command loops. Loops may be nested in such a way that one command loop contains another command loop which, in turn, contains other command loops, and so on. Command loops should not be nested to more than about 15 levels.

The general form of a command loop is:

n<command string>

where "command string" is the sequence of commands to be iterated and n is the number of interations. N must be a positive integar. If n is not supplied, a value of 4096 is assumed.

If a search command which is not preceded by a colon modifier is entered into a command loop and the search fails, the command loop is exited immediately and the command following the right angle brackets of the loop is the next command to be executed. No error message is printed.

Q-Registers

TECO provides 36 data storage registers, called Q-registers, which may be used to store single integers and/or ASCII character strings. Each Q-register is divided into two storage areas. In the number storage area, each Q-register can store one integer in the range $-2048 \leqslant n \leqslant 2047$. In the text storage area, each Q-register can store an ASCII character string of up to 2000 characters which may be either text or a TECO command string. Each Q-register has a single character name which is one of the upper case letters A to Z or one of the digits 0 to 9. In this manual, a Q-register name is indicated by a lower case "q", which stands for any one of the 36 Q-registers.

Table 2-37 lists the commands which permit characters to be loaded into the Q-registers.

Table 2-37 Q-Register Loading Commands

Command	Function
↑Uqstring$	Where ↑U is an uparrow-U character, "q" is the name of a user-specified Q-register, "string" is a string of ASCII characters, and "$" signifies an ALT MODE. This command inserts character string "string" into the text storage area of Q-register "q". (Do not confuse uparrow-U with CTRL/U; CTRL/U is an editing command.)
@↑U/string/	Equivalent to the ↑U command except that the character string to be inserted into Q-register q may contain ALT MODE characters as long as it does not contain two consecutive ALT MODE characters. The insertion must be delimited before and after by any character (a slash is shown here) which does not appear in the insertion.
nUq	Load n into the number storage area of Q-register q, where n must be an integer in the range $-2049 \leqslant n \leqslant 2048$.
n%q$	Add n to the contents of the number storage area of Q-register q, where n should be an integer that will not cause overflow. If n is not present, it is assumed to be equal to 2.
n%$	Equivalent to n%q$ except that the resulting value contained in Q-register q is used as a numeric argument for the next command. If the next command does not require a numeric argument, this value is discarded.
Xq	Copy the contents of the buffer from the current position of the pointer through and including the next line feed character into the text storage area of Q-register q.
nXq	Execute the Xq command n times, where n is an integer in the range $-2048 \leqslant n \leqslant 2047$. If n is positive, the n lines following the current pointer position are copied into the text storage area of Q-register q. If is negative, the n lines preceding the pointer are copied. If n is zero, the contents of the buffer from the beginning of the line on which the pointer is located up to the pointer is copied.
m,nXq	Copy the contents of the buffer from the m+1th character through and including the nth character into the text storage area of Q-register q. M and n must be positive, and m should be less than n.

Table 2-38 lists the commands which permit characters to be retrieved from the Q-registers.

Table 2-38. Q-Register Execution Commands

Command	Function
Gq	Copy the contents of the text storage area of Q-register q into the buffer at the current position of the buffer pointer, leaving the pointer positioned after the last character copied.
Qq	Use the integer stored in the number storage area of Q-register q as the argument of the next command. The characters "Qq" may be considered as equivalent to "the value contained in the number storage area of Q-register q", where "q" is any Q-register name.
Mq	Execute the contents of the text storage area of Q-register q as a command string.
nMq	Execute the contents of the text storage area of Q-register q as a command string and use n as a numeric argument for the first command in this string.

Branching Commands

TECO commands may be combined in sophisticated command strings which are capable of solving even the most complex editing problems. In fact, TECO might be considered a programming language which accepts an input file as data and processes this input to produce an output file. As with most programming languages, TECO provides an unconditional branch command and a set of conditional execution commands.

To provide for branching within a command string, there must be some means of naming locations inside the string. TECO permits location tags which have the form:

! tag !

to be placed between any two commands in a command string. The name "tag" will be associated with this location when the command string is executed. Tags may contain any number of ASCII characters and any character except for special characters and exclamation points. Since tags are ignored by TECO except when a branch command references the tagged location, they may also be used as comments within complicated command strings.

The unconditional branch command is the O command which has the form:

Otag$

where "tag" is a named location elsewhere within the command string and "$" signifies an ALT MODE. When an O command is executed, the next command to be executed will be the command following the tag referenced by the O command, and command execution continues normally from this point.

If an O command is stored in a Q-register as part of a command string which is to be executed by an M command, the tag referenced by the O command must also reside in the same Q-register.

An important restriction on the O command prevents any O command which is inside a command loop from branching to a tagged location preceding the command loop. However, it is always possible to branch out of a command loop to a location which follows the command loop.

Conditional Execution Commands

All conditional execution commands are of the form:

n"Gcommand string'

where "n" is a numeric argument on which the decision is based, "G" may be any of the conditional executional commands listed in Table 2-39, and "command string" is the command string which will be executed if the condition is satisfied. If the condition on n is not satisfied, the command string will not be executed. Note that the numeric argument is separated from the conditional execution command by a double quote (") and the command string is terminated with an apostrophe (').

Conditional execution commands may be nested in the same manner as iteration commands. That is, the command string which is to be executed if the condition on n is met may contain conditional execution commands, which may, in turn, contain further conditional execution commands.

Table 2-39 lists the conditional execution commands. Each conditional execution command must be followed by a command string (not shown in Table 2-39) which will be executed only if the condition is satisfied. This command string must be terminated by an apostrophe. If the condition is not satisfied, the first command following the apostrophe will be the next command executed.

Table 2-39 Conditional Execution Command

Command	Function
n-m"A	Execute the following command string (terminated by an apostrophe) if n is greater than or equal to m. Otherwise skip the following command string. N and m should be integers in the range $0 \leqslant n \leqslant 4095$.
n-m"B	Execute the following command string (terminated by an apostrophe) if n is less than m. Otherwise skip the following command string. N and m should be integers in the range $0 \leqslant n \leqslant 4095$.
n"G	Execute the following command string (terminated by an apostrophe) if n is greater than zero. Otherwise skip the following command string. N must be an integer in the range $-2048 \leqslant n \leqslant 2047$.
n"L	Execute the following command string (terminated by an apostrophe) if n is less than zero. Otherwise skip the following command string.
n"E	Execute the following command string (terminated by an apostrophe) if n is equal to zero. Otherwise skip the following command string.
n"C	Execute the following command string (terminated by an apostrophe) if n is the decimal ASCII code of any character which is one of the upper or lower case letters A to Z or one of the digits 0 to 9. Otherwise skip the following command string.

In general, integers "n" and "m" will be variables (e.g. the content of a Q-register) whose values are computed during execution of the command string.

There is one further conditional execution command which is not related to the commands listed in Table 2-39. The n; command, where n is any integer, may be inserted between any two commands in an iterated command loop. It has the general form:

m<string1n;string2>string3

where "m" is the iteration count, "stringl", "string2", and "string3" are command strings and "n;" is the conditional exit command. When the n; command is executed, it will cause TECO to exit the command loop so that "string3" will be executed next if n is in the range $0<n<2047$. If n falls outside this range, the n; command

is ignored and "string2" is executed next. The semicolon may be preceded by an argument such as Qq (the value of the numeric part of Q-register q), or the argument may be omitted if the semicolon command is preceded by a command that generates an argument, such as any colon-modified search command.

Note that all unmodified search commands entered within command loops are executed as though they were preceded by a colon and followed by a semicolon. If the search command is preceded by a colon modifier, however, it will be executed as it stands.

The conditional execution commands will accept user-supplied numeric arguments (n and m Table 2-39) of the same form as most other TECO commands. This is generally a trivial case, however, because the user will know in advance whether the condition is satisfied, and need not use the conditional execution command. The following section introduces run-time numeric quantities computed by TECO which may also be used as numeric arguments.

Numeric Arguments

Almost all TECO commands may be preceded by a numeric argument which generally indicates the number of iterations, or how many times the command should be executed. Some numeric arguments must be positive, while others may be negative or zero. In any case, every numeric argument is stored as a single, 12-bit word.

This leads to an important restriction on the maximum size of any numeric argument. Commands which require positive arguments must have an argument in the range $0<n<4095$, since 4095 is the largest number which may be stored in one 12-bit word. Commands which may have positive or negative arguments require an argument in the range $-2048<n<2047$, because -2048 is the smallest number which may be stored in 12 bits using 2's complement notation, while 2047 is the largest number which may be stored in this manner.

TECO maintains several internal counters which record conditions within the text buffer. Each of the counters has a one-character name which is equivalent to the current contents of the counter. These characters may be entered as numeric arguments to TECO commands. When the command is executed, the current value of the designated counter is substituted for the character and used in the numeric argument of the command.

Some of the characters which stand for specific values associated with the text buffer have been introduced earlier in this section. For example, the dot character (.), which references a counter that always contains the number of characters between the beginning of the buffer and the current pointer position, may be used in the argument of a T command. The command ". , .+5T" causes the 5 characters following the buffer pointer to be typed out. When this command is executed, the number of characters preceding the buffer pointer is substituted (twice) for the character "dot." The addition is then carried out, and the command is executed as though it were of the form "m,nT".

Table 2-40 lists all of the characters which have special numeric values. Any of these characters may be used as numeric arguments in place of the values they represent.

Table 2-40 Characters Associated with Numeric Quantities

Character	Function
B	Always equivalent to zero. Thus, B represents the position at the beginning of the buffer, preceding the first character in the buffer.
Z	Equivalent to the number of characters currently contained in the buffer. Thus, Z represents the position at the end of the buffer, following the last character in the buffer.
	Equivalent to the number of characters between the beginning of the buffer and the current position of the pointer, thus represents the current position of the pointer.
H	Equivalent to the numeric argument pair, "B,Z", or "from the beginning of the buffer up to the end of the buffer." Thus, H represents the whole buffer.
↑Z	CTRL/Z (or uparrow/Z) is equivalent to the number of characters presently stored in the entire Q-register storage area, including storage requirements for the command string containing the ↑Z character. Maximum capacity of the Q-register storage area is 2000 characters on an 8K system, or 2944 characters on a 12K system.
nA	Where n is a positive integer. Equivalent to the ASCII code for the n+1th character following the current position of the pointer.

Table 2-40 Characters Associated with Numeric Quantities (Cont.)

Character	Function
↑E	CTRL/E (or uparrow/E) is equivalent to 4095 (-1) if the buffer currently contains a full page of text (which was terminated by a form feed in the input file) or 0 if the buffer contains only part of a page of text (which filled the buffer to capacity before the terminating form feed was read). The ↑E flag is tested by N, EC and EX commands to determine whether a form feed should be appended to the content of the buffer on output.
↑F	CTRL/F (or uparrow/F) is equivalent to the current value of the console switch register.
↑↑X	The combination of CTRL-shift-N (or a double uparrow) followed by any character is equivalent to the value of the ASCII code for the character. The "X" in this example may be any character except CTRL/C and CTRL/P.
\	A backslash (shift-L) character which is not preceded by a numeric argument is equivalent to the value of the digit string (if any) that begins with the character immediately following the buffer pointer and is terminated by the next character that is not a digit. The first character may be a digit or one of the characters + or —. As each backslash is evaluated, TECO moves the buffer pointer to a position immediately following the digit string. If there is no digit string following the pointer, the backslash is equivalent to zero and the pointer position remains unchanged.
↑T	CTRL/T (or uparrow/T) is equivalent to the ASCII code for the next character typed at the terminal. CTRL/T (or uparrow/T) may be entered in the numeric argument of any command. When TECO executes a command string, every ↑T character encountered causes it to pause and accept one character typed at the terminal. The ASCII code for this character is then substituted for the ↑T.
↑V	CTRL/V (or uparrow/V) is equivalent to the version number of the version of TECO which is currently being run. This manual describes TECO version 3.
↑H	CTRL/H (or uparrow/H) is equivalent to zero.
Mq	The Mq command (execute the content of the text storage area of Q-register "q" as a command string) may return a numeric value if the last command in the string returns a numeric value and is not followed by an ALT MODE.

The numeric argument of a TECO command may consist of a single integer, any of the characters listed in Table 2-40, or an arithmetic combination of integers and the characters listed in Table 2-40. If an arithmetic expression is supplied as a numeric argument, TECO will evaluate the expression. All arithmetic expressions are evaluated from left to right. Parentheses may be used to override the normal order of evaluating an expression. If parentheses are used, all operations within the parentheses are performed, from left to right, before performing operations outside the parentheses. Parentheses may be nested, in which case the innermost expression contained by parentheses will be evaluated first. Table 2-41 lists all of the arithmetic operators that may be used in arithmetic expressions. All arithmetic is two's complement arithmetic modulo 4096.

Table 2-41 Arithmetic Operators

Operator	Example	Function
+	+2=2	Ignored if used before the first term in an expression.
+	5+6=11	Addition, if used between terms.
−	−2=4094	Negation, if used before the first term in an expression.
−	8−2=6	Subtraction, if used between terms.
*	8*2=16	Multiplication. Used between terms.
/	8/3=2	Integer divide and drop the remainder. Used between terms.
&	12&10=8	Bitwise logical AND of the binary representation of the two terms. Used between terms.
#	12#10=14	Bitwise logical OR of the binary representation of two terms. Used between terms.

Table 2-42 lists three commands which may be used to facilitate entering arithmetic expressions into TECO command strings.

Table 2-42 Radix Control Commands

Command	Function
n=	Where n is an arithmetic expression which may contain the operators listed in Table 2-12B. Upon execution, this command causes the value of the expression to be typed out at the terminal.
CTRL/O	The CTRL/O command causes all subsequent numeric input to be accepted as octal numbers and all subsequent numeric output to be transmitted in octal. This command must not be typed while TECO is executing a T command. The octal radix will continue to be used until the next CTRL/D command is executed or a command error is encountered. All error messages are printed as decimal numbers, and the decimal radix remains set after any error message is printed.
CTRL/D	The CTRL/D command causes all subsequent numeric input to be accepted as decimal numbers and all subsequent numeric output to be transmitted in decimal. This is the initial setting.
n\	Where n is any arithmetic expression. The backslash command (preceded by an expression) inserts the value of n into the text buffer at the current position of the pointer, leaving the pointer positioned after the last digit of the insertion. The octal value of n will be inserted if the octal radix is set.

Some TECO commands generate numeric arguments which they pass on to subsequent commands. An example is any colon-modified search command, which causes the next sequential command to be executed with an argument of -1 or 0, depending upon the outcome of the search. Commands of this sort are very useful, but occasionally it may be undesirable to have arguments passed in this manner. A single ALTMODE character may be inserted between any two commands in a command string, as long as it is not placed adjacent to another ALTMODE character. This ALTMODE has no effect on the individual commands, however a numeric argument will never be passed across the extraneous ALTMODE.

Programming Aids

The characters carriage return, line feed, vertical tab and space are ignored in command strings, except when they appear as part

of a text argument. These characters may be inserted between any two TECO commands to lend clarity to a long command string. The carriage return/line feed combination is particularly useful for typing command strings which are too long to fit on a single line.

One of the most powerful features of TECO is its ability to store very long command strings so that a given sequence of commands may be executed whenever needed. Long command strings may be thought of as edited programs and, like any other type of program, they should be documented by means of comments.

Comments may be inserted between any two commands by using a tag construction of the form:

!THIS IS A COMMENT!

Comments may contain any number of characters and any characters except the special characters. It is often convenient to include carriage return and line feed characters within the comments so that the command string looks like:

```
TECO commands!          This comment describes line 1
!TECO commands!         This comment describes line 2
!more commands!
!$$!                    end of command string!
```

The CTRL/A command may be used to print out a statement at any point during the execution of a command string. The CTRL/A command has the general form:

↑Atext↑A

where the first ↑A is the actual command, which may be entered by striking the control key and the A key simultaneously or by typing an uparrow followed by an A character. The second CTRL/A character is the command terminator, which must be entered by typing the control key and the A key simultaneously. The message which appears between the CTRL/A characters may contain any characters except the special characters and the ↑A character. Upon execution, this command causes TECO to print the specified message at the terminal.

The ↑Amessage↑A command is particularly useful when it precedes a command whose numeric argument contains ↑T or ↑F

characters. The message may contain instructions notifying the computer operator what sort of input is required.

A question mark character entered between any two commands in a command string causes TECO to print all subsequent commands at the terminal as they are executed. Commands will be printed as they are executed until another question mark character is encountered. The second question mark character may be in the same command string as the first question mark, or it may appear in a later command string. It may not be first character typed after a TECO-generated error message, however.

If an error is typed while entering a command string, the error may be corrected at any time before the double ALT MODE which terminates the command string is typed. Characters may be deleted individually by striking the RUBOUT key. Each depression of the RUBOUT key deletes one character, beginning with the last character typed, and causes the deleted character to be printed at the terminal. If an entire command string is deleted in this manner, TECO responds by printing a new asterisk at the left margin.

Typing two successive CTRL/G characters causes the current command string to be erased completely. The double CTRL/G command should not be confused with the single ↑G command. The double CTRL/G must be produced by holding the control key depressed while striking the G key twice (if the terminal has a bell, it will ring). The uparrow form of CTRL/G may not be used for the double CTRL/G command.

Error Messages

When TECO encounters an illegal command or a command that cannot be executed, a numeric error message is printed at the terminal. Error messages are of the form:

?n

where "n" is a 1- or 2-digit decimal number that references an error message from the list contained at the end of this section. When an error message is generated, the command to which it refers is not executed, the rest of the current command string is ignored, and TECO prints an asterisk at the left margin to indicate that it is ready to accept further commands.

The ↑S character can be very useful. For instance, if a long insertion is typed without the "I" in front of it and an error results, ↑S can be used to save the insertion, and the GZ command

can be used to put it into the text buffer (including the trailing ALT MODEs).

In some cases it may be difficult to determine which command in a long command string resulted in an error message. Typing a question mark immediately after the TECO-generated error message causes TECO to print current command string up to and including the erroneous character. When used in this manner, the question mark must be the first character typed after the error message is printed. It is not necessary to follow the question mark with an ALT MODE.

In general, TECO command strings should be limited to a maximum of 2000 characters. Command strings exceeding 2000 characters in length should be split into smaller strings. Long command strings are impractical because the probability of a command error is increased and because a string which contains more than 2000 characters is too long to be stored in a Q-register.

TECO reserves a limited amount of core for command string storage. If a very long command string (or a long insertion command) uses all but the last 10 command string storage locations, TECO prints one CTRL/G character as soon as only 10 storage locations remain and another CTRL/G after every additional character that is entered. (CTRL/G "prints" as a bell ring if the terminal has a bell.) Should this occur, the current command string must be terminated. Attempting to enter more than 10 additional characters results in a fatal error.

Manipulating Large Pages

TECO is designed to operate most efficiently when edited files that contain no more than 3800 characters per page. If any page of an input file contains more than 3800 characters, the various TECO input commands will terminate reading that page into core when the first line feed following the 3800th character is read or when the 4000th character is read, whichever occurs first. Thus, it is never possible for a page which contains more than 4000 characters to reside entirely within the text buffer.

Most of the TECO output commands append a form feed to the content of the buffer whenever a page of text is written onto the output file. If an input file contains pages which are more than 4000 characters long, these output commands will cause form feed characters to be inserted into the file at locations where they may

not be desired. To prevent this, the user must understand exactly how the output commands operate. These commands are described briefly in Table 2-43.

Table 2-43 Form Feed Processing Output Commands

Output Command	Form Feeding Processing
P, nP, PW and nPW	Always append a form feed character to the text contained in the buffer, regardless of whether this text actually constitutes a complete page of the file. That is, a form feed is appended on output even though one may not have been present upon input.
HP, HPW, m,nP and m, nPW	Never appends a form feed character to the text contained in the buffer.
N, EC and EX	If the text contained in the buffer was followed by a form feed character in the input file, a form feed will be appended to this text upon output. If this text was not followed by a form feed character (i.e. if input was terminated because the buffer had reached the prescribed capacity), no form feed will be appended.

If it becomes necessary to edit text that consists of large pages without introducing extraneous form feed characters into the output, this may be accomplished by avoiding all output commands except the N, EC and EX commands. For example, if use of a P command would introduce an extraneous form feed, use an N command, instead, to search for a character string contained in the next page of the input file.

Techniques and Examples

TECO may be used in three ways. The most elementary application involves using TECO to create and edit ASCII files on-line. The user enters short command strings, often consisting of a single command, and proceeds from task to task until the file is completely edited.

Since every edited job is simply a sequence of TECO commands, an entire job may be accomplished with one long command string consisting of all the short command strings placed end to end with the intervening double ALT MODE characters removed. This leads to the concept of a TECO editing program, which is

simply a long command string that performs a certain editing task. Editing programs may be written (using TECO) and stored in the same manner as any other ASCII file. Whenever the program is needed, it may be read into the buffer as text, stored in a Q-register, and executed by an Mq command (where "q" is the Q-register name).

This is fine for clear-cut editing assignments, such as converting from one format to another or editing certain characters out of a file, but many editing jobs are so complex that a given editing program will only solve a small class of problems. The solution, in this case, is to write very specialized "editing subroutines." TECO subroutines might perform such elementary functions as replacing every occurrence of two or more consecutive spaces with a tabulation character, for example, or ensuring that words are not hyphenated across a page boundary. When an editing problem arises, the right combination of subroutines may be loaded into various Q-registers, augmented with additional commands if necessary, and called by a "mainline" command string.

Editing subroutines are essentially macros; that is, sequences of commands which perform commonly required editing functions. Thus, the third and most powerful application of TECO is the creation and use of a macro library. As each editing job is undertaken, the user may look for sequences of operations which might be required in future editing asignments. All of the TECO commands required to perform such an operation may be loaded into a Q-register and executed by means of an Mq or nMq command. When the job is finished, the content of any Q-register which contains a useful macro may be written onto an output file (via the buffer) and saved in the macro library. The nMq command, which was designed to facilitate use of macros, permits one run-time numeric argument to be passed to the macro.

The following examples are intended to illustrate some of the techniques discussed earlier. It would not be practical to include examples of the use of every TECO command, since most of the commands admit to many diverse applications. Instead, users are encouraged to experiment with the individual commands.

Example 1: Splitting, Merging, and Rearranging Files

Assume that the user has a file named PGM.PA on the system device and this file contains data in the following form:

AB FORM CD FORM EF FORM GH FORM IJ FORM KL FORM MN FORM OP where each of the letters A, B, C, etc., represents 20 lines of text and FORM represents a form feed character. The user intends to rearrange the file so that it appears in the following format:
AOB FORM D FORM MN FORM EF FORM ICJ FORM KL FORM P FORM GH
The following sequence of commands will achieve this rearrangement. (Search command arguments are not listed explicitly.)

.R TECO	Call TECO.
*EBPGM.PA$Y$$	Specify input file and get first page.
*NC$$	Search for a character string in C to write A and B on the output file.
*J20X1$$	Save all of C in Q-register 1.
*20K$$	Delete C from the buffer.
*NG$$	Search for a character string in G to write D, E and F on the output file.
*HX2$$	Save G and H in Q-register 2.
*Y$$	Delete GH from the buffer and read IJ.
*20L$$	Move pointer to the beginning of J.
*G1$$	Insert C, which was stored in Q-register 1.
*NM$$	Search for a character string in M to write ICJ and KL on the output file.
*HX1$$	Save MN in Q-register 1 (the previous content is overwritten).
*Y$$	Delete MN and read OP.
*J20X3$$	Save all of O in Q-register 3.
*20K$$	Delete O from the buffer.
*P$$	Write P onto the output file, leaving the buffer cleared (the input file is exhausted).
*G2$$	Bring GH into the buffer from Q-register 2.
*HPEF$$	Write GH on the output file and close it.
*EBPGM.PA$Y$$	Open the partially revised file.
*20L$$	Move the pointer to the beginning of B.
*G3$$	Insert all of O from Q-register 3.
*ND$$	Search for a character string in D to write AOB on the output file.
*PWHK$$	Write D on the output file and clear buffer.
$G1$$	Bring all of MN from Q-register 1 into the buffer.
*EX$$	Write MN onto the output file, then close the file and exit to the OS/8 monitor.

At this point, the file has been rearranged in the desired format. Of course, this rearrangement could have been accomplished in

fewer steps if the commands listed above had been combined into longer command strings. Note that the asterisks shown at the left margin in this example are generated by TECO, and not typed by the user.

Assume, now, that the same input file mentioned earlier, containing data in the form:

AB FORM CD FORM EF FORM . . . FORM OP

is to be split into two separate files, with the first file containing AB FORM CD and the second file containing KL FORM M, while the rest of the data is to be discarded. The following commands could be used to achieve this rearrangement:

.R TECO	Call TECO.
*ERFILE$EWFILE.1$$	Open the input file and the first output file.
*Y$$	Read AB into the buffer.
*P$$	Write AB FORM onto the output file and read CD into the buffer.
*HPEF$$	Write CD onto the output file (without appending a form feed), and close the first output file.
*←K$$	Search for a character string in K. After this command has been executed, the buffer will contain KL. No output is generated by the search.
*EWFILE.2$P$$	Open the second output file and write KL onto it. Read MN into the buffer.
*20L0,.P$$	Move the pointer to the end of M, then write M onto the output file.
*EF↑C$$	Close the second output file and exit to the OS/8 monitor.

As a final example of file manipulation techniques, assume that the user has two files. One file is MATH.BK, which contains information in the form:

AB FORM CD FORM EF FORM GH FORM IJ FORM KL

and the other is MATH.FT, which contains:

MN FORM OP FORM QR

If both of these files are stored on DECtape unit 1, the following sequence of commands may be used to merge the two files into a single file, MATH.NW, which contains all of MATH.FT followed by the latter half of file MATH.BK in the following format:

MN FORM OP FORM QR FORM GH FORM IJ FORM KL

*R TECO	Call TECO.
*ERDTA1:MATH.FT$$	Open the first input file.
*EWMATH.NW$$	Open the output file on the OS/8 default device.
*Y$$	Read MN into the text buffer.
*NR$$	Search for a character string in R to write MN and OP onto the output file.
*PW$$	Write QR onto the output file, appending a form feed.
*ERDTA1:MATH.BK$$	Open the second input file.
*Y$$	Read AB into the buffer. QR is overwritten.
*←G$$	Search for a character string in G to delete AB, CD and EF, leaving GH in the buffer.
*NK$$	Search for a character string in K to write GH and IJ on the output file, leaving KL in the buffer.
*HPEF↑G$$	Write KL onto the output file (without appending a form feed) and close the file, then exit to the OS/8 monitor.

Example 2: Alphabetizing by Binary Search

Assume that TECO is running and the buffer contains many short lines of text, each beginning with an alphabetic character at the left margin (i.e. immediately following a line feed). The lines might consist of names in a roster, for example, or entries in an index. Figure 2-3 shows a command string which will rearrange the lines into rough alphabetical order. This command string groups all lines which begin with the character "A" at the beginning of the page, followed by all lines beginning with "B," and so on.

Figure 2-4 is a flowchart showing the sequence of operations performed by this command string. The algorithm could be extended to place the entries in strict alphabetical order by having it loop back to perform the same binary sorting operation on successive characters in each line.

Example 3: An Elementary TECO Macro

Figure 2-5 shows a TECO macro which right justifies the content of the text buffer on a 60-space line. This macro assumes that the buffer contains paragraphs of text in manuscript form, and that every line which is not the last line of a paragraph contains between 40 and 60 characters.

When the macro is run, it counts the number of spaces and the

number of characters in each line. It then adds spaces between words until the line contains a total of 60 characters. Lines which contain fewer than 40 characters are assumed to be paragraph terminators. These lines are not justified. Figure 2-6 shows how the macro may be stored, loaded and executed using DECtape unit 1 as the storage device. In this example, DECtape file "TEXT.AS" is the file to be justified.

```
TECO FIGURE 1

!START! J ØAUA!

!!CONT! L ØAUB!

!QA-QB"G XA K -L GA 1UZ'!

!QBUA!

!L Z-."G -L OCONT$'!

!QZ"G ØUZ OSTART$'!

!$$
```

Figure 2-3 Command String for Example 2

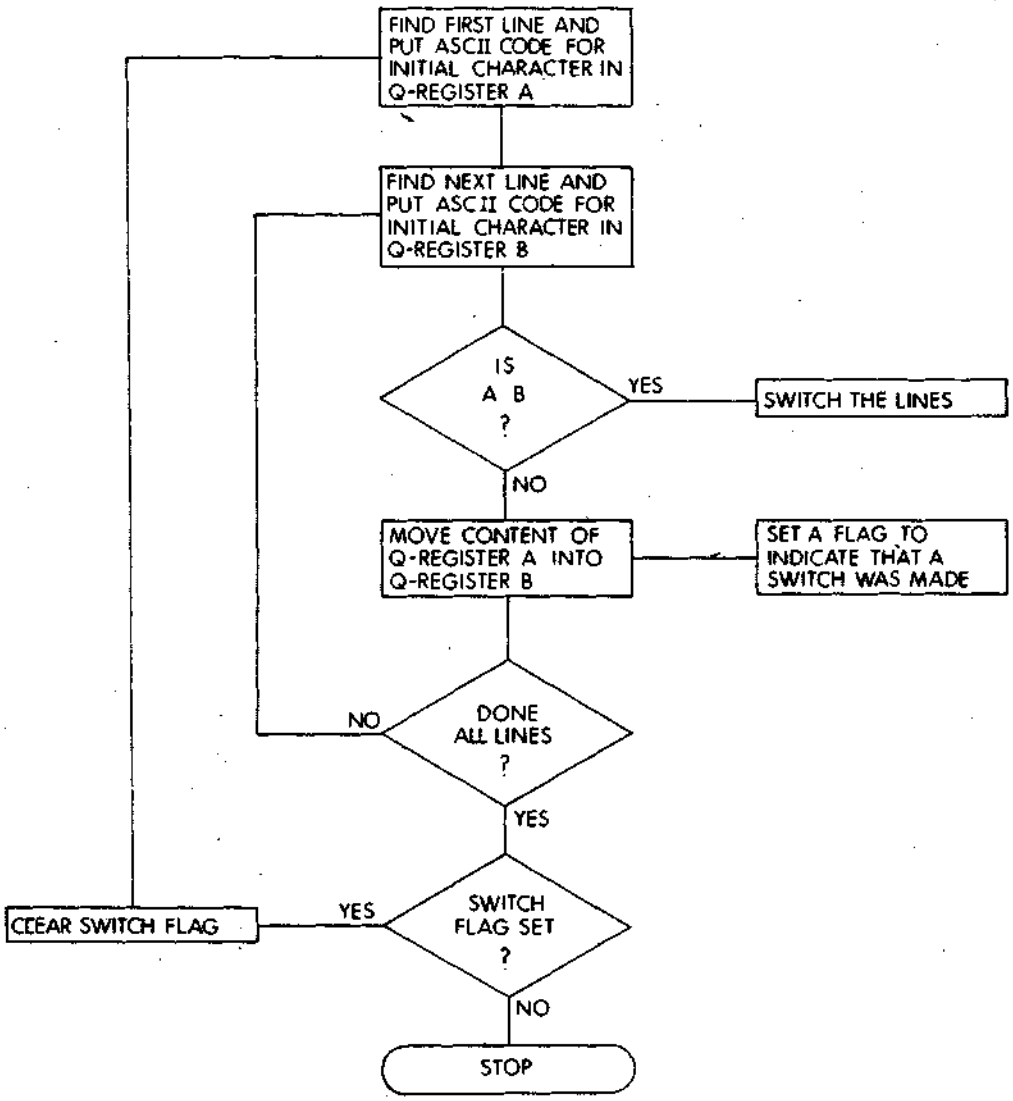


Figure 2-4 Flowchart for Example 2

```
J !1! ØUN ØUS!


!Z-."G!
!<ØNA-32"E 1%S$'!
!ØNA-13"E ØJUSTIFY$'!
!1%N$>!

!!JUSTIFY! ØN-40"G!
!60-ØN-ØS"G ØS<S $I $S↑N $>!
!ØL ØS%N$ ØS%S$ ØJUSTIFY$'!
!60-ØN"G 60-ØN<S $I $S↑N $>''!
!L Ø1$'$$!
```

Figure 2-5 TECO Macro for Example 3

```
.R TECO
*ERDTA1:MACRO.TE$ Y HXI HK$$
*ERDTA1:TEXT.AS$ Y M1$$
*
```

Figure 2-6 Loading and Executing a TECO Macro

Example 4: Managing a Macro Library

A TECO macro library is most conveniently stored with TECO on the OS/8 system device. Macros are usually short enough to require a small amount of storage space, however it is impractical to store each macro in a separate named file, because a large macro library stored in this manner would make the device directly unmanageably large and might even exhaust the available directory entries.

Figure 2-7 illustrates a macro that packs the user's TECO macro library (or any other set of short ASCII files) into a single file requiring only one directory entry. This macro could be stored on the system device in a file named PACK.TE (the extension indicates a TECO command string file). The user must also create a separate file containing the name of each file to be packed. This file must be formatted as follows:

file1.ex
file2.ex
file3.ex
.
.
.
filen.ex

where each file specification after the first is preceded and followed by a carriage return/line feed combination. Assume that such a file is created and stored as INDEX.AS on the system device. If macro PACK.TE is also on the system device, the following commands will pack all files listed in INDEX.AS into file MACLIB.PK on the system device.

```
Y 10<A> HX0 HK 0U1 0U2
<G0 Q1J :S
$; .U1 2R 0X4 HK
IERDSK:$ G4 @I.$. HX3
M3 HK I\$ G4 I\$ 0U5
!A! AZ"N PW HK 0U5 0A$'
%2$> Q2"E OB$' EF
!B! HK Q2\ I FILES PACKED
$ HT HK
```

Figure 2-7 File Packing Macro

```
.R TECO
*ERSYS:PACK.TE$Y HXP HK$$
*ERSYS:INDEX.AS$EWSYS:MACLIB.PK$MP$$
N FILES PACKED
*
```

Figure 2-8 Loading and Running the File Packing Macro

```
0U2 <Y -Z; 0A-92"E
:2S\$"L .-13"L 1,.-1X4
0,.K Q2"E OA$' EF
!A! %2$ IEWDSK:$ G4
@I.$. 0,.X3 M3 0,.K'''
PW> Q2"E OB$' EF
!B! Q2\ I FILES UNPACKED
$ HT HK
```

Figure 2-9 Unpacking Macro

```
.R TECO
*ERSYS:UNPACK.TESY HXU HK$$
*ERINCEX.AS$MP$$
N FILES UNPACKED
*
```

Figure 2-10 Loading and Running the Unpacking Macro

The packing macro prints a message, as shown, where "n" is the number of files that were packed. The files to be packed will be taken from the system device. Files PACK.TE, INDEX.AS and MACLIB.PK may reside on any file-structured device if the file designations in the above command summary are changed accordingly.

Once the packing macro has packed all the files into MACLIB.PK, the individual files may be deleted. Alternatively, macros could be saved in individual files on, say, DTA1 and the packing macro could be used to pack the files into one system device file simply by replacing the imbedded "ERDSK:" command in the macro body with "ERDTA1:". If the library index is also saved on the system device, an unpacking macro may be used to create an unpacked copy of the macro library whenever required, and the original library tape may be saved as a backup.

Figure 2-9 illustrates a macro that unpacks the output file produced by the packing macro. This macro accepts a packed ASCII file (such as MACLIB.PK), then unpacks the file and restores each entry as a discrete file with the appropriate specification.

Assume that a user desires to access a macro or other ASCII file that was packed into file MACLIB.PK, as shown in the previous example. If file UNPACK.TE contains the unpacking macro, the following commands will unpack the entries and restore them as individual, named files.

The unpacking macro prints a message, as shown, where "n" is the number of files that were unpacked. Once the files are unpacked, they will be directed to the system disk. Alternately, the unpacked files could be directed to, say, DECtape unit 5 by modifying the "EWDSK:" command in the macro body to read "EWDTA5:".

Using TECO to Retrieve Lost Files

Inevitably, through user error, hardware error, or operating system error, valuable files may be deleted or directories destroyed.

A two-word patch to TECO creates a program known as SUPER TECO which may be a considerable aid in these situations. The patch is:

```
.GET SYS TECO

.ODT

2034/7420 7610
2117/7450 7410
↑C
.SAVE SYS STECO
```

To use STECO, mount the device on which you want to retrieve the file, then type:

```
.R STECO
*ERDEV:$$
*<STRING$$
```

where "string" is part of the first page of the desired file (for instance, the title line). STECO will search the entire device for the first occurrence of the specified strings. The device may contain many old copies of the desired file. The user should examine the text following each occurrence of the string; if it is an earlier version, or a listing file, the user should continue searching until the correct occurrence is found. Once the correct file is found, type:

```
*ERDEV:$EWDEV2:FILE$$
*N<STRING$NENDSTRING$PWEF$$
```

where n is the number of times you had to search for the specified string on your investigation pass, and "endstring" is a string located at the end of the file. This operation retrieves your file and copies it onto another device. There may be meaningless characters preceding the first good line of your file, if so, delete them.

Incompatabilities Between OS/8 TECO and DECsystem-10 TECO

OS/8 TECO is a proper subset of DECsystem-10 TECO with the following exceptions:

1. The ↑U and ↑R commands do not exist on DECsystem-10 TECO.
2. The ↑S command as described on page 2-154 is implemented differently on DECsystem-10 TECO (refer to the DECsystem-10 Users Handbook).

3. OS/8 TECO assumes a semicolon after failing search commands in interation brackets, DECsystem-10 TECO does not.
4. The EC, ↑D, ↑O, and ↑W command do not exist in DECsystem-10 TECO.
5. DECsystem-10 TECO ignores the number n in the nA command and is equivalent to the OS/8 TECO 0A command.
6. The "A and "B compares are not needed by DECsystem-10 TECO because "G and "L are adequate to compare character pointers.
7. The ↑Z and ↑V numeric values are not implemented by DECsystem-10 TECO.

Table 2-44 TECO Command Summary

Command	Function
ERdev:filnam.ex$	Input file selection.
EWdev:filnam.ex$	Output file selection.
EBdev:filnam.ex$	I/O file selection with backup protection.
Y	Clear buffer and read one page of input file.
A	Read one page of input file and append to current buffer content.
BUFFER POINTER POSITIONS	
B	Before first character.
.	Current pointer position (number of characters to left of pointer).
Z	After last character (number of characters in buffer.
m,n	From $m+1^{th}$ character through and including n^{th} character.
H	Entire buffer; equivalent to B,Z.
ARITHMETIC OPERATORS	
−n	Negation.
m+n	Addition.
m−n	Subtraction.
m*n	Multiplication.
m/n	Divide and truncate.
m&n	Bitwise logical AND.
m#n	Bitwise logical OR.
( )	Perform enclosed operations first.

Table 2-44 TECO Command Summary (Cont.)

Command	Function
POINTER POSITIONS	
nJ	Position pointer between n^{th} and $n+1^{th}$ characters.
nC	Move pointer forward across n characters.
nR	Move pointer backward across n characters.
mL	Position pointer at beginning of n^{th} line following current position.
TYPE-OUT COMMANDS	
nT	Type buffer content from pointer position to beginning of n^{th} following line.
m,nT	Type $m+1^{th}$ character through and including n^{th} character.
n=	Type the integer equivalent of expression n.
↑Atext↑A	Type the enclosed text.
↑O	Inhibit typeout.
DELETION COMMANDS	
nD	Delete the n characters following the pointer.
−nD	Delete the n characters preceding the pointer.
nK	Delete the n lines following the pointer.
m,nK	Delete the $m+1^{th}$ character through and including the n^{th} character.
INSERTION COMMANDS	
Itext$	Insert text delimited by I and ALT MODE.
<I>text$	Insert tabulation, then text. <I> is a TAB (control-I) character.
nI	Insert character whose ASCII code is n.
@I/text/	Insert text delimited by arbitrary character shown as a slash.
n\	Insert the ASCII code for integer n.
OUTPUT AND EXIT	
PW	Write current page and append form feed.
P	Write current page, append form feed, clear buffer, and read next page.
m,nP	Write $m+1^{th}$ through n^{th} characters without appending a form feed.
EF	Close the current output file.
↑G	Close the current output file and exit to the OS/8 monitor.
↑C	Immediate exit to the OS/8 monitor.
↑P	Exit to the monitor and do a START 200.
EX	Write the rest of the input file on the output file and exit to the monitor.
EC	Write the remainder of the input file on the output file and close the file.

Table 2-44 TECO Command Summary (Cont.)

Command	Function
SEARCH COMMANDS	
nStext$	Begin at the pointer and search for the n^{th} occurrence of the text delimited by the S and the ALT MODE on the current page.
nNtext$	Equivalent to nStext$ except that the search is continued across page boundaries.
n←text$	Equivalent to nNtext$ except that no output is generated.
nFNtest1 $text2$	Do nNtext1$ and then replace text1 with text2.
:nStext$	Equivalent to nStext$ except that it returns a value of −1. If the search succeeds, or 0, if the search fails. The colon may be used with N and ← searches.
n@S/text/$	Equivalent to nStext$ except that the text is delimited by the arbitrary character following the S, instead of ALT MODE.
↑X	Accept any character in this position.
↑S	Accept any separator in this position. Save last typed command.
↑N	Accept any character except the following character in this position.
↑Q	Interpret the next character literally, rather than as a command.
ITERATION AND FLOW CONTROL	
n< >	Perform enclosed commands n times.
n;	If n is positive, jump out of the current iteration field.
!tag!	Define a position named "tag" at this location.
Otag$	Jump to the position defined by "tag."
n"E	If n=0, execute the following command string.
n"N	If n≠0, execute the following command string.
n"L	If n is less than zero, execute the following command string.
n"G	If n is greater than zero, execute the following command string.
n"C	If n is the ASCII code for an alphanumeric character, execute the following commands.
n−m"A	If n is greater than or equal to m, execute the following commands.
n−m"B	If n is less than m, execute the following commands.
Q-REGISTER COMMANDS	
nUq	Store n in Q-register q.
Qq	Equivalent to the value stored in Q-register q.

Table 2-44 TECO Command Summary (Cont.)

Command	Function
n%I	Add n to the content of Q-register q and return this value.
↑Uqtext$	Insert text into Q-register q.
nXq	Load the n following lines into Q-register q.
m,nXq	Load the m+1th character through the nth character into Q-register q.
Gq	Insert the content of Q-register q into the buffer.
Mq	Execute the content of Q-register q as a command string.
NUMERIC VALUES	
nA	ASCII value of nth character following pointer.
↑E	Form feed flag.
↑F	Console data switches.
↑H	Always equals zero.
↑↑X	Equivalent to the ASCII code for character "X."
↑Z	Command and Q-register storage words in use.
↑O	Set octal radix.
↑D	Set decimal radix.
\	Equivalent to the value of the digit string following the pointer.
↑T	Equivalent to the ASCII code for the next character typed.
↑V	Equivalent to the number of the version of TECO being run.
PROGRAMMING AIDS	
?	After an error message, identifies erroneous character.
?	Except after an error message, toggles in and out of trace mode.
↑G↑G	Erases current command string.

Running TECO On The PDP-12

When TECO is run on a PDP-12, part of the content of the text buffer is displayed on the console scope. Initially, TECO displays the three lines immediately preceding and following the buffer pointer. An uparrow character (↑) is displayed below the current position of the pointer.

The n↑W command, where n is a small positive integer and ↑W is a control-W or uparrow/W character, causes TECO to display the n lines preceding and following the current position of the pointer on the scope. If a value of n greater than 7 is specified, the

display will wrap around the scope and produce flicker on it. N is assigned an initial value of 3.

The ↑W command (CTRL/W with no numeric argument) causes TECO to execute one scope display cycle. This command may be entered into long command strings for the purpose of displaying part of the buffer at a given point in the command string.

When a ↑F character is entered in a numeric expression on the PDP-12, TECO considers the ↑F to be equivalent to the current value of the right-hand switch register.

Assembly Instructions

The source tape of TECO may be assembled with the PAL8 assembler, in the same manner as any other PAL8 source program. For example, if a TECO source DECtape is mounted on unit 1, typing:

```
.R PAL8
*TECO.BN<DTA1:TECO.PA
```

will produce a TECO binary file on the system device.

Once a TECO binary file has been created on the system device, the following commands will create a core image file called TECO. SV on the system device:

```
.R ABSLDR
*TECO.BN$
.SAVE SYS TECO
```

The binary paper tape of TECO (DEC-S8-UEXTB-A-PB8) may be loaded and saved on the system device by the following sequence of commands:

```
.R ABSLDR
*PTR:$
.SAVE SYS TECO
```

The system will print an uparrow after the ALT MODE character to indicate that a paper tape should be loaded into the reader. Strike any key at the terminal to continue.

KSR-35 Teletype users who want to take advantage of the KSR-35 hardware tabulation feature should change the starting address of TECO to location 5200. This may be accomplished by the commands:

```
.GET SYS TECO
.SAVE SYS TECØ;5200
```

Error Messages

TECO error messages consist of a question mark followed by a number. Typing a second question mark immediately after an error message printout causes the command string to be printed up to and including the character which caused the error message.

Table 2-45 TECO Error Messages

Error	Cause
1	Illegal command.
2	Incomplete command. Can mean either: a. Character missing from command. b. Iteration brackets do not match. c. Conditional delimiters (double quote and apostrophe) do match. d. O command references nonexistent tag.
3	Non-alphanumeric Q-register name.
4	Command iterations or macro calls nested too deeply.
5	Text buffer overflow.
6	Search string longer than 31 characters.
7	Numeric argument missing before comma, equals sign, U, or quote (").
8	Illegal file name in ER, EW or EB command.
9	Semicolon or failing search encountered on command level.
10	Iteration close (>) without matching open (<).
11	Attempt to move pointer outside of text buffer.
12	Q-register storage overflow.
13	Incomplete command.
14	Output file too large, or else output parity error.
15	Input file parity error.
16	File error; can mean either: a. Input file not found by ER command. b. Cannot enter output file with EW or EB command. c. Device specified for file does not exist. d. EB command specifies a file on a non-file-structured device.
17	An output command was encountered which would have caused TECO to overflow its current output file. User should close the current output file and write all further output onto one (or more) additional files. These files may be combined if necessary.
18	Attempt to execute an output command without opening an output file.

OS/8 assemblers

pal 8
sabr
flap/ralf

chapter 3

pal 8

INTRODUCTION

PAL8 is an 8K, two pass assembler designed to run under the OS/8 Operating System. Pass 1 reads the input file and sets up the symbol table. Pass 2 reads the input file and uses the symbol table created in pass 1 to generate the binary (object) file. The binary file is an absolute binary tape and may be loaded into core with the Absolute Loader or Binary Loader. As an optional third pass, a side-by-side octal and symbolic listing and the symbol table are output. (Using the options available, the three passes may be automatically executed. However, if the source file is to be read from the paper tape reader, the user must reload the tape for each pass.) The listing file may be used as an input to the Cross Reference Program (CREF), and the symbol table may be requested to be in a form suitable for input to DDT. If a listing file, but no binary file or /L or /G option was specified, PAL8 does not execute pass 2, but instead goes directly from pass 1 to pass 3.

PAL8 can handle I/O from any OS/8 device which handles ASCII text, and has pseudo-ops and options not available in the other PDP-8 assemblers. It is loaded and saved by way of the OS/8 Monitor and Absolute Loader. It will accept input generated by the Editor and will generate output acceptable to Absolute Loader and CREF.

CALLING AND USING PAL8

PAL8 is called from the system device by typing:

R PAL8

in response to the dot printed by the Keyboard Monitor. The system replies by activating the Command Decoder, which in turn prints an asterisk (*) at the left margin of the teleprinter paper. At this point a command string is entered which indicates the binary

and listing output devices and file names, the input devices and file names, and any options selected by the user. 1 to 9 input files may be specified. The format of the command string is:

```
*DEV:BINARY,DEV:LISTING,DEV:CREFLS<DEV:INPUT/OPTIONS
```

If the extension to the file name is omitted, the following assumed extensions are assigned:

.PA for input file.
.BN for binary output file.
.LS for listing output file.
.TM for intermediate CREF file (if the /C option was specified).

A null output file indicates no output file of that type is to be generated. For example, to assemble, load and run a PAL8 program named PROGRM which is stored on DECtape unit 1, the user would type:

```
.R PAL8
*BIN<DTA1:PROGRM/G
```

After the assembly, the program will be loaded and run with the starting address assumed to be location 0200 in field 0, and the binary stored on the system device as BIN.BN.

The assembler prints any error messages encountered in the program on the teleprinter. Typing CTRL/O at the keyboard during an assembly will suppress the printing of error messages on the teleprinter; however, messages are still printed in the output file and occur immediately before the line that is in error.

PAL8 OPTIONS

Table 3-1 lists the options available in PAL8 which can be indicated in the command string typed to the Command Decoder.

When the /L or /G option is specified, the user can also include any option to the Absolute Loader in the I/O specification line for PAL8, such as = starting address option. If no address is specified, execution begins at 200. If no binary output file is specified with /L or /G a temporary file, PAL8BN.TM, is created and loaded.

Table 3-1 PAL8 Run-Time Options

Option	Meaning
/B	This option makes the operator ! a 6-bit left shift instead of an inclusive OR. (A!B equals A↑100+B)
/C	Chain to SYS:CREF.SV after assembly. The second output file specified is the output file passed to CREF. The third output file is where PAL8 generates its output. If no third output file is given, SYS:CREFLS.TM is assumed. The /C option supersedes the /G and /L options if specified in the same command string.
/D	Generate a DDT compatible symbol table (applicable only if a listing file is specified).
/E	Enable error messages if a link is generated. The LG error message would be generated as well as the link being flagged.
/F	Disable extra zero fill in TEXT pseudo-op. If the text in the TEXT pseudo-op contains an even number of characters, no word of zeroes will be added to the end.
/G	Call the Absolute Loader, load the binary file, and begin execution at the indicated starting address. If no starting address is indicated, start at 200.
/H	Generate non-paginated output. Header, page numbers and page format are suppressed (applicable only if a listing file is specified).
/J	Do not list lines containing code in conditional brackets which is conditionalized out.
/K	Used in assembling very large programs; causes systems containing 12K or more of core to use field(s) 2 and up as symbol table storage.
/L	Call the Absolute Loader at the end of the assembly and load the binary file (applicable only if a binary file was specified).
/N	Generate the symbol table, but not the listing (applicable only if a listing file is specified. The /H option is assumed).
/O	Disable origining to 200 after pseudo-op. The origin remains what it was before the FIELD pseudo- op.
/S	Omit the symbol table normally generated with the listing (applicable only if a listing file is specified).

Table 3-1 PAL8 Run-Time Options (Cont.)

Option	Meaning
/T	Output a carriage return/line feed in place of the form feed character(s) in the program (applicable only if a listing file is specified).
/W	Do not remember the number of literals that were previously stored on a page after origining off page and then back on again.

EXAMPLES OF SPECIFICATION STRINGS

Example 1:

```
.R PAL8
*PTP:,LPT:<SOURCE
```

The above lines cause the PAL8 assembler to be loaded from the system device and the program SOURCE.PA (or SOURCE) to be assembled. The binary output of the assembly is put onto the paper tape punch, and the listing and symbol table on the line printer.

Example 2:

```
.R PAL8
*,LISTIN<PROG/S
```

The above specification line causes PAL8 to assemble PROG.PA (or PROG), putting the listing only into the file LISTIN.LS on the default device DSK. No binary output and no symbol table are generated.

Example 3:

```
.R PAL8
*BIN<INPUT.XY/G=600
```

The above specification line assembles INPUT.XY, putting the binary output into a file named BIN.BN, and then calls the Absolute Loader, which loads the file BIN.BN and starts it at 600. (=600 is an option to the Absolute Loader specifying the starting address.)

Example 4:

```
.R PAL8
*DTA1:PROG
```

The preceding lines will assemble the file PROG from device DTA1, checking for errors, which are listed on the teleprinter. There are no output files.

RESTARTING AND TERMINATING PAL8

PAL8 may only be restarted if the Command Decoder has not been dismissed. For example:

```
.R PAL8
*↑C
.ASSIGN DTA7 DSK
.ST
*
```

If a restart is attempted after the Command Decoder has been dismissed, NO! ! is typed and control returns to the Keyboard Monitor. The user must call PAL8 for each assembly.

CHARACTER SET

The following characters are acceptable as input to PAL8:

1. The alphabetic characters: A through Z.
2. The numeric characters: 0 through 9.
3. The characters described in following sections as special characters and operators.
4. Characters which are ignored during assembly such as LINE FEED, FORM FEED, and RUBOUT.

All other characters are illegal (except when used in a comment) and cause the error message:

IC nnnn

to be printed during pass 1; nnnn represents the location at which the illegal character occurred. (As assembly proceeds, each instruction is assigned a location determined by the current location counter, detailed later in this chapter. When an illegal character or any other error is encountered during assembly, the value of the current location counter is returned in the error message.) Illegal characters do not generally cause assembly to halt. If an illegal character occurs in the middle of a symbol, the symbol is terminated at that point.

STATEMENTS

PAL8 source programs are usually prepared on the console terminal (using the OS/8 EDITOR) as a sequence of statements. Each statement is written on a single line and is terminated by typing the RETURN key. There are four types of elements in a PAL8 statement which are identified by the order of their appearance in the statement and by the separating (or delimiting) character which follows or precedes the element. These are:

1. label,
2. instruction
3. operand
4. /comment

A statement must contain at least one of these elements and may contain all four types. The assembler interprets and processes the statements, generating one or more binary instructions or data words, or performing an assembly process.

Labels

A label is the symbolic name created by the programmer to identify the location of a statement in the program. If present, the label is written first in a statement. It must begin with an alphabetic character, contain only alphanumeric characters, and be terminated by a comma; there must be no intervening spaces between any of the characters and the comma.

Instructions

An instruction may be one or more of the mnemonic machine instructions or a pseudo-operation which directs assembly processing. (Assembly pseudo-ops are described later in this chapter.) Instructions are terminated with one or more spaces (or tabs if an operand follows) or with a semicolon, slash, or carriage return.

Operands

Operands are the octal or symbolic addresses of an assembly language instruction or the argument of a pseudo-operator, and can be any expression. In each case, interpretation of an operand depends upon the instruction or the pseudo-op. Operands are terminated by a semicolon, slash, or carriage return.

Comments

The programmer may add notes or comments to a statement by separating these from the remainder of the line with a slash. Such comments do not affect assembly processing or program execution but are useful in the program listing for later analysis or debugging. The assembler ignores everything from the slash to the next carriage return.

It is possible to have only a carriage return on a line, resulting in a blank line in the final listing. No error message is given.

FORMAT EFFECTORS

The following characters are useful in controlling the format of an assembly listing. They allow a neat readable listing to be produced by providing a means of spacing through the program.

Form Feed

The form feed code causes the assembler to output blank lines in order to skip to a new page in the output listing during pass 3; this is useful in creating a page-by-page listing. The form feed is generated by typing a CTRL/L on the console terminal.

Tabulations

Tabulations are used in the body of a source program to separate fields into columns. For example, a line written:

```
GO, TAD TOTAL/MAIN LOOP
```

is much easier to read if tabs are inserted to form:

```
GO,     TAD TOTAL     /MAIN LOOP
```

Statement Terminators

The RETURN key is used to terminate a statement and causes a carriage return/line feed combination to occur in the listing. The semicolon (;) may also be used as a statement terminator and is considered identical to a carriage return except that it will not terminate a comment. For example:

```
TAD A     /THIS IS A COMMENT;     TAD B
```

The entire expression between the slash and the carriage return is considered a comment. Thus in this case the assembler ignores the TAD B. If, for example, the user wishes to write a sequence of

instructions to rotate the contents of the accumulator and link six places to the right, it might look like the following:

```
RTR
RTR
RTR
```

However, the programmer can alternatively place all three instructions on a single line by separating them with the special character semicolon and terminating the entire line with a carriage return. The above sequence of instructions can then be written:

```
RTR;RTR;RTR
```

NOTE

If an OS/8 CREF listing is desired, there are certain restrictions on the use of semicolons. Refer to the section on CREF in Chapter 2 of this handbook.

These multi-statement lines are particularly useful when setting aside a section of data storage for use during processing. For example, a 4-word cleared block could be reserved by specifying either of the following:

```
LIST,   0;   0;   0;   0
```

or

```
LIST,   0
        0
        0
        0
```

Either format may be used to input data words (data words may be in the form of numbers, symbols, or expressions, explained next). Each of the following lines generates one storage word in the object program:

```
DATA,   7777
        A+C-B
        S
        123+B2
```

NUMBERS

Any sequence of digits delimited by either a SPACE, TAB, semicolon, or carriage return forms a number. PAL8 initially interprets numbers in octal (base 8). This can be changed to decimal using a special pseudo-operator (explained later in this chapter). Numbers are used in conjunction with symbols to form expressions.

SYMBOLS

A symbol is a string of alphanumeric characters beginning with a letter and delimited by a non-alphanumeric character. Although a symbol may be any length only the first six characters are recognized; since additional characters are ignored, symbols which are identical in their first six characters are considered identical.

Permanent Symbols

The assembler contains a table (called its permanent symbol table) which lists the symbols for all PDP-8 pseudo-op codes, memory reference instructions, operate and IOT (input/output transfer) instructions. These instructions are symbols which are permanently defined by PAL8 and need no further definition by the user; they are summarized at the end of the chapter. For example:

HLT — This is a symbolic instruction assigned the value 7402 by the assembler and stored in its permanent symbol table.

User-Defined Symbols

All symbols not defined by the assembler (and represented in its permanent symbol table) must be defined within the source program.

A symbol may be used as a statement label, in which case it is assigned a value equal to the current location counter; it is called a symbolic address and can be used as an operand or as a reference to an instruction. Permanent symbols (instructions, special characters, and pseudo-ops) may not be used as symbolic addresses. The following are examples of legal symbolic addresses:

ADDR,
TOTAL,
SUM,
A1,

The following are illegal symbolic addresses:

AD>M,	(contains an illegal character)
7ABC,	(first character must be alphabetic)
LA BEL,	(must not contain imbedded spaces)
D+TAG,	(contains a legal but non-alphanumeric character)
LABEL ,	(must be terminated by a comma with no intervening spaces)

Current Location Counter

As source statements are processed, PAL8 assigns consecutive memory addresses to the instructions and data words of the object program.

The current location counter contains the address in which the next word of object code will be assembled and is automatically incremented each time a memory location is assigned. A statement which generates a single object program storage word increments the location counter by one. Another statement might generate six storage words, incrementing the location counter by six.

The user sets or resets the location counter by typing an asterisk followed by the octal absolute address value in which the next program word is to be stored. If the origin is not set by the user, PAL8 begins assigning addresses at location 200.

```
        *300      /SET CURRENT LOCATION COUNTER
                    TO 300
TAG,    CLA
        JMP A
B,      0
A,      DCA B
        .
        .
```

The symbol TAG (in the preceding example) is assigned a value of 0300, the symbol B a value of 0302, and the symbol A a value of 0303. If a symbol is defined more than once in this manner, the assembler will print the illegal definition diagnostic:

ID address

where address is the value of the location counter at the second occurrence of the symbol definition. The symbol is not redefined.

(For an explanation of diagnostic messages refer to the section on PAL8 Error Conditions.) For example:

```
            *300
START,      TAD A
            DCA COUNTER
CONTIN,     JMS LEAVE
            JMP START
A,          -74
COUNTER,    0
START,      CLA CLL
            .
            .
            .
```

The symbol START would have a value of 0300, the symbol CONTIN would have a value of 0302, the symbol A would have a value of 0304, the symbol COUNTER (considered COUNTE by the assembler) would have a value of 0305. When the assembler processed the next line it would print (during pass 1):

```
ID COUNTE+0001
```

Since the first pass of PAL8 is used to define all symbols, the assembler will print a diagnostic during pass 2 if reference is made to an undefined symbol. For example:

```
            *7170
A,          TAD C
            CLA CMA
            HLT
            JMP A1
C,          0
```

This would produce the undefined symbol diagnostic:

```
US A+0003
```

Symbol Table

Initially, the assembler's symbol table contains the mnemonic op-codes of the machine instructions and the assembler pseudo-op codes; this is its permanent symbol table. As the source program is processed, user-defined symbols along with their binary values are added to the symbol table. The symbol table is listed in alphabetic order at the end of pass 3.

During pass 1, if PAL8 detects that the symbol table is full (in other words, there is no more memory space in which to store symbols and their associated values), the symbol table exceeded diagnostic is printed:

SE address

and control returns to the OS/8 Monitor. If the system contains more than 8K of memory, the user may choose the /K option with the Run command, or more address arithmetic may be used to reduce the number of symbols. It is also possible to segment a program and assemble the segments separately, taking care to generate proper links between the segments (see LINK GENERATION AND STORAGE). PAL8's symbol capacity is 992 symbols. The permanent symbol table contains 24 pseudo-operations and 71 symbols, leaving space for 897 possible user-defined symbols. Each additional 4K allows 992 new symbols.

Instructions concerning altering the permanent symbol table are discussed later in this chapter should the user wish to add instructions more suitable to his programming needs.

Direct Assignment Statements

The programmer may insert new symbols with their assigned values directly into the symbol table by using a direct assignment statement in the form:

SYMBOL=VALUE

VALUE may be a number or expression. No spaces or tabs may appear between the symbol to the left of the equal sign and the equal sign itself. The following are examples of direct assignment statements:

```
A=6
EXIT=JMP I O
C=A+B
```

All symbols to the right of the equal sign must be already defined. The symbol to the left of the equal sign is subject to the same restrictions as a symbolic address, and its associated value is stored in the user's symbol table. The use of the equal sign does not increment the location counter; it is, rather, an instruction to the assembler itself.

A direct assignment statement may also equate a new symbol to the value assigned to a previously defined symbol. For example:

```
BETA=17
GAMMA=BETA
```

The new symbol, GAMMA, is entered into the user's symbol table with the value 17. The value assigned to a symbol may be changed as follows:

```
ALPHA=5
ALPHA=7
```

The second line of code shown changes the value assigned to ALPHA from 5 to 7.

Symbols defined by use of the equal sign may be used in any valid expression. For example:

```
        *200
A=100       /DOES NOT UPDATE CLC
B=400       /DOES NOT UPDATE CLC
A+B         /THE VALUE 500 IS ASSEMBLED AT
              LOC. 200
TAD A       /THE VALUE 1200 IS ASSEMBLED AT
              LOC. 201
```

If the symbol to the left of the equal sign is in the permanent symbol table, the redefinition diagnostic:

```
RD address
```

will be printed as a warning, where address is the value of the location counter at the point of redefinition. The new value will be stored in the symbol table; for example:

```
CLA=7600
```

will cause the diagnostic:

```
RD +200
```

Whenever CLA is used after this point, it will have the value 7600.

Symbolic Instructions

Symbols used as instructions must be predefined by the assembler or defined in the assembly by the programmer. If a statement has no label, the instructions may appear first in the statement and must

be terminated by a space, tab, semicolon, slash, or carriage return. The following are examples of legal instructions:

TAD	(a mnemonic machine instruction)
PAGE	(an assembler pseudo-op)
ZIP	(an instruction defined by the user)

Symbolic Operands

Symbols used as operands normally have a value defined by the user. The assembler allows symbolic references to instructions or data defined elsewhere in the program. Operands may be numbers or expressions. For example:

```
TOTAL, TAD ACI + TAG
```

The values of the two symbols ACI and TAG (already defined by the user) are combined by a two's complement add (see the section on Operators). This value is then used as the address of the operand.

Internal Symbol Representation for PAL8

Each permanent and user-defined symbol occupies four words in the symbol table storage area. A PDP-8 instruction has an operation code of three bits as well as an indirect bit, a page bit, and seven address bits. The PAL8 assembler distinguishes between pseudo-ops, memory reference instructions, other permanent symbols, and user-defined symbols in the symbol table.

EXPRESSIONS

Expressions are formed by the combination of symbols, numbers, and certain characters called operators, which cause specific arithmetic operations to be performed. An expression is terminated by either a comma, carriage return, or semicolon.

Operators

There are seven characters in PAL8 which act as operators:

+	Two's complement addition
−	Two's complement subtraction
↑	Multiplication (unsigned, 12-bit)
%	Division (unsigned, 12-bit)
!	Boolean inclusive OR
&	Boolean AND

Space (or TAB)	Treated as a Boolean inclusive OR except in a memory reference instruction

Two's complement addition and subtraction are explained in detail in Chapter 1 of INTRODUCTION TO PROGRAMMING; the user should refer to that handbook if he wishes more information. No checks for overflow are made during assembly, and any overflow bits are lost from the high order end. For example:

7755+24 will give a result of 1

The operators + and − may be used freely as prefix operators.

Multiplication is accomplished by repeated addition. No checks for sign or overflow are made. All 12 bits of each factor are considered as magnitude. For example:

3000↑2 will give a result of 6000

Division is accomplished by repeated subtraction. The number of subtractions which are performed is the quotient. The remainder is not saved and no checks are made for sign. Division by 0 will arbitrarily yield a result of 0. For example:

7000%1000 will yield a result of 7

This could be written as:

−1000%1000

In this case the answer might be expected to be −1 (7777), but all 12 bits are considered as magnitude and the result is still 7.

Use of the multiplication and division operators requires an attention to sign on the part of the programmer beyond that which is required for simple addition and subtraction. Table 3-2 contains examples of operators.

The ! operator causes a Boolean inclusive OR to be performed bit by bit between the left-hand term and the right-hand term. (The inclusive OR is explained in Chapter 1 of INTRODUCTION TO PROGRAMMING.) There is an option which can be given to the assembler to have "!" interpreted as a 6-bit left shift of the left term prior to the inclusive OR of the right. According to this interpretation:

if A=1 and B=2
then A!B=0102

Table 3-2 Use of Operators

Expression	Also written as:	Result
7777+2	−1+2	+1
7776−3	−2−3	7773 or −5
0↑2		0
2↑0		0
1000↑7		7000 or −1000
0%12		0
12%0		0
7777%1	−1%1	7777 or −1
7000%1000	−1000%1000	7
1%2		0

Under normal conditions A!B would be 0003. The & operator causes a Boolean AND to be performed bit by bit between the left and right values. The operation is the same as that indicated by the memory reference instruction AND.

SPACE has special significance depending on the context in which it is used. When the symbol preceding the space is not a memory reference instruction as in the following example:

```
SMA CLA
```

it causes an inclusive OR to be performed between them. In this case, SMA=7500 and CLA=7600. The expression SMA CLA is assembled as 7700. When SPACE is used following pseudo-operators it merely delimits the symbol. When it is used after memory reference operators it also signals the assembler that a memory reference instruction must be assembled.

User-defined symbols are treated as operate instructions. For example:

```
      A=333
      *200
B,    CLA
```

Possible expressions and their values using the symbols just defined are shown below. Notice that the assembler reduces each expression to one 4-digit (octal) word:

```
A       0333
B       0222
A+B     0555
A-B     0111
-A      7445
1-B     7557
B-1     0221
A!B     0333    (an inclusive OR is performed)
-71     7707
```

If the information generated is to be loaded, the current location counter is incremented. For example:

```
B-7;A+4;A-B
```

produces three words of information; the current location counter is incremented after each expression. The statement:

```
HLT=HLT CLA
```

produces no information to be loaded (it produces an association in the symbol table) and hence does not increment the current location counter.

```
        *4721
TEMP,
TEM2,   0
```

The location counter is not incremented after the line TEMP,; the two symbols TEMP and TEM2 are assigned the same value, in this case 4721.

Since a PDP-8 instruction has an operation code of three bits as well as an indirect bit, a page bit, and seven address bits, the assembler must combine memory reference instructions in a manner somewhat differently from the way in which it combines operate or IOT instructions. The assembler differentiates between the symbols in its permanent symbol table and user-defined symbols. The following symbols are used as memory reference instructions:

AND	0000	Logical AND
TAD	1000	Two's complement addition
ISZ	2000	Increment and skip if zero
DCA	3000	Deposit and clear accumulator
JMS	4000	Jump to subroutine
JMP	5000	Jump

When the assembler has processed one of these symbols, the space following it acts as an address field delimiter.

```
        *4100
        JMP  A
A,      CLA
```

A has the value 4101, JMP has the value 5000, and the space acts as a field delimiter. These symbols are represented as follows:

```
A       100  001  000  001
JMP     101  000  000  000
```

The seven address bits of A are taken, e.g.:

```
000  001  000  001
```

The remaining bits of the address are tested to see if they are zeros (page zero reference); if they are not, the current page bit is set:

```
000  011  000  001
```

The operation code is then ORed into the JMP expression to form:

```
101  011  000  001
```

or, written more concisely in octal:

```
5301
```

In addition to the above tests, the page bits of the address field are compared with the page bits of the current location counter. If the page bits of the address field are nonzero and do not equal the page bits of the current location counter, an out-of-page reference is being attempted and the assembler will take action as described in the section on Link Generation and Storage.

Special Characters

In addition to the operators described in the previous section, PAL8 recognizes several special characters which serve specific functions in the assembly process. These characters are:

=	equal sign
,	comma
*	asterisk
.	dot
"	double quote

( )	parentheses
[ ]	square brackets
/	slash
;	semicolon
< >	angle brackets
$	dollar sign

The equal sign, comma, asterisk, slash, and semicolon have been previously described. The remainder will be described next.

The special character dot (.) always has a value equal to the value of the current location counter. It may be used as any integer or symbol (except to the left of an equal sign), and must be preceded by a space when used as an operand. For example:

```
*200
JMP .+2
```

is equivalent to JMP 0202. Also,

```
*300
.+2400
```

will produce in location 0300 the quantity 2700. Consider:

```
*2200
CALL=JMS I.
0027
```

The second line (CALL=JMS I.) does not increment the current location counter, therefore, 0027 is placed in location 2200 and CALL is placed in the user's symbol table with an associated value of 4600 (the octal equivalent of JMS I).

If a single character is preceded by a double quote ("), the 8-bit value of ASCII code for the character is used rather than interpreting the character as a symbol (ASCII codes are listed in Appendix A). For example:

```
CLA
TAD     ("A
```

The constant 0301 is placed in the accumulator. The code:

```
".
```

will be assembled as 0256. The character must not be a carriage return or one of the characters which is ignored on input (discussed at the end of this section).

Left and right parentheses ( ) enclose a current page literal (closing member is optional).

```
*200
        .
        .
        CLA
        TAD  INDEX
        TAD  (2)
        DCA  INDEX
        .
        .
```

The left parenthesis is a signal to the assembler that the expression following is to be evaluated and assigned a word in the constants table of the current page. This is the same table in which the indirect address linkages are stored. In the above example, the quantity 2 is stored in a word in the linkage and literals list beginning at the top of the current memory page. The instruction in which the literal appears is encoded with an address referring to the address of the literal. A literal is assigned to storage the first time it is encountered; subsequent reference to that literal from the current page is made to the same register. The use of literals frees symbol storage space for variables and makes programs much more readable.

If the programmer wishes to assign literals to page zero rather than to the current page, he may use square brackets, [and], in place of parentheses. This enables the programmer to reference a single literal from any page of memory. For example:

```
*200
        TAD  [2]
        .
        .
        .
*500
        TAD  [2]
        .
        .
        .
```

The closing member is optional. Literals may take the following forms: constant term, variable term, instruction, expression, or another literal.

NOTE

Literals can be nested, for example:

```
*200
        TAD (TAD (30
```

This type of nesting may be continued in some cases to as many as 6 levels, depending on the number of other literals on the page and the complexity of the expressions within the nest. If the limits of the assembler are reached, the error messages BE (too many levels of nesting) or PE (too many literals) will result.

Angle brackets are used as conditional delimiters. The code enclosed in the angle brackets is to be assembled or ignored contingent upon the definition of the symbol or value of the expression within the angle brackets. (The IFDEF, IFNDEF, IFZERO, and IFNZRO pseudo-operators are used with angle brackets and are described later in this chapter.)

NOTE

Programs which use conditionals should avoid angle brackets in comments as they may be interpreted as beginning or terminating the conditional.

The dollar sign character ($) is optional at the end of a program and is interpreted as an unconditional end-of-pass. It may however occur in a text string, comment or " term, in which case it is interpreted in the same manner as any other character.

The following characters are handled by the assembler for the pass 3 listing, but are otherwise ignored:

FORM FEED	Used to skip to a new page
LINE FEED	Used to create a line spacing without causing a carriage return
RUBOUT	Used by the EDITOR to allow corrections in the input file

Nonprinting characters include:

SPACE
TAB
RETURN

INSTRUCTIONS

There are two basic groups of instructions: memory reference and microinstructions. Memory reference instructions require an operand, microinstructions do not.

Memory Reference Instructions

In PDP-8 computers, some instructions require a reference to memory. They are appropriately designated memory reference instructions, and take the following format:

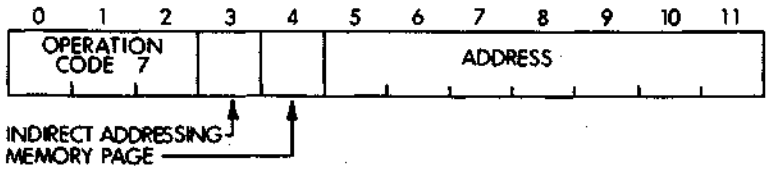


Figure 3-1 Memory Reference Bit Instructions

Bits 0 through 2 contain the operation code of the instruction to be performed. Bit 3 tells the computer if the instruction is indirect. Bit 4 tells the computer if the instruction is referencing the current page or page zero. This leaves bits 5 through 11 (7 bits) to specify an address. In these 7 bits, 200 octal (128 decimal) locations can be specified; the page bit increases accessible locations to 400 octal or 256 decimal. A list of the memory reference instructions and their codes is given at the end of the chapter.

In PAL8 a memory reference instruction must be followed by a space(s) or tab(s), an optional I or Z designation, and any valid expression. It may be defined with the FIXMRI instruction. (See pg. 3-33; Altering the Permanent Symbol Table.) Permanent symbols may be defined using the FIXTAB instruction and may be used in address fields as shown below:

```
A=1234
FIXTAB
TAD A
```

Indirect Addressing

When the character I appears in a statement between a memory reference instruction and an operand, the operand is interpreted as the address (or location) containing the address of the operand to be used in the current statement. Consider:

```
TAD 40
```

which is a direct address statement, where 40 is interpreted as the location on page zero containing the quantity to be added to the accumulator. References to locations on the current page and page zero may be done directly. For compatibility with older paper-tape assemblers the symbol Z is also accepted as a way of indicating a page zero reference, as follows:

```
TAD Z 40
```

This is an optional notation, not differing in effect from the previous example. Thus, if location 40 contains 0432, then 0432 is added to the accumulator. Now consider:

```
TAD I 40
```

which is an indirect address statement, where 40 is interpreted as the address of the location containing the quantity to be added to the accumulator. Thus, if location 40 contains 0432, and location 432 contains 0456, then 456 is added to the accumulator.

NOTE

Because the letter I is used to indicate indirect addressing, it is never used as a variable. Likewise the letter Z, which is sometimes used to indicate a page zero reference, is never used as a variable.

Microinstructions

Microinstructions are divided into two groups: operate and Input/Output Transfer (IOT) microinstructions. Operate microinstructions are further subdivided into Group 1, Group 2, and Group 3 designations.

NOTE

If a programmer mistakenly specifices an illegal combination of microinstructions, the assembler will perform an inclusive OR between them; for example:

CLL SKP is interpreted as SPA
(7100) (7410) (7510)

OPERATE MICROINSTRUCTIONS

Within the operate group, there are three groups of microinstructions which cannot be mixed. Group 1 microinstructions perform clear, complement, rotate and increment operations, and are designated by the presence of a 0 in bit 3 of the machine instruction word.

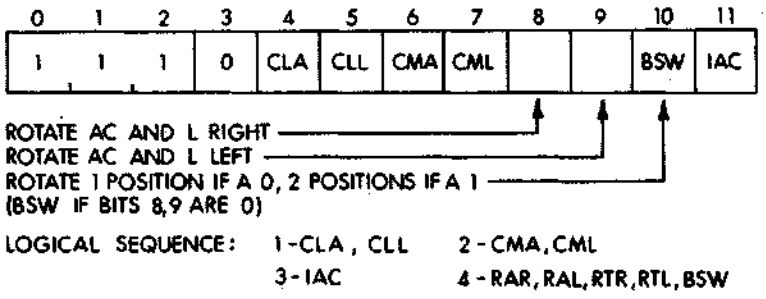


Figure 3-2 Group 1 Operate Microinstruction Bit Assignments

Group 2 microinstructions check the contents of the accumulator and link and, based on the check, continue to or skip the next instruction. Group 2 microinstructions are identified by the presence of a 1 in bit 3 and a 0 in bit 11 of the machine instruction word.

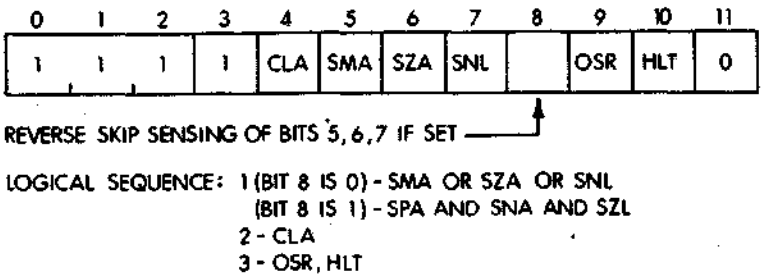


Figure 3-3 Group 2 Operate Microinstruction Bit Assignments

Group 3 microinstructions reference the MQ register. They are differentiated from Group 2 instructions by the presence of a 1 in bits 3 and 11. The other bits are part of a hardware arithmetic option.

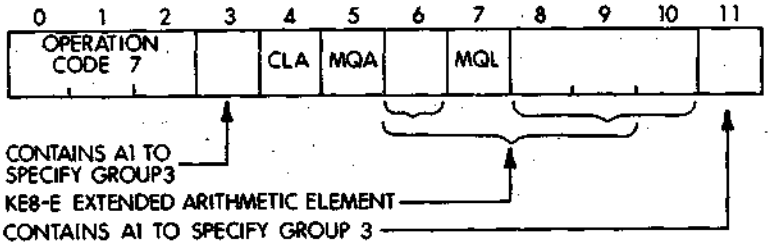


Figure 3-4 Group 3 Operate Microinstruction Bit Assignments

Group 1 and Group 2 microinstructions cannot be combined since bit 3 determines either one or the other. Within Group 2, there are two groups of skip instructions. They can be referred to as the OR group and the AND group.

OR Group	AND Group
SMA	SPA
SZA	SNA
SNL	SZL

The OR group is designated by a 0 in bit 8, and the AND group by a 1 in bit 8. OR and AND group instructions cannot be combined since bit 8 determines either one or the other.

If the programmer does combine legal skip instructions, it is important to note the conditions under which a skip may occur.

1. OR Group—If these skips are combined in a statement, the inclusive OR of the conditions determines the skip. For example:

   SZA SNL

   The next statement is skipped if the accumulator contains 0000 or the link is a 1 or both.

2. And Group—If the skips are combined in a statement, the logical AND of the conditions determines the skip. For example:

   SNA SZL

   The next statement is skipped only if the accumulator differs from 0000 and the link is 0.

INPUT/OUTPUT TRANSFER MICROINSTRUCTIONS

These microinstructions initiate operation of peripheral equipment and effect an information transfer between the central processor and the Input/Output device(s); i.e., console terminal, and line printer.

Autoindexing

Interpage references are often necessary for obtaining operands when processing large amounts of data. The PDP-8 computers have facilities to ease the addressing of this data. When one of the absolute locations from 10 to 17 (octal) is indirectly addressed, the contents of the location is incremented before it is used as an address and the incremented number is left in the location. This allows the programmer to address consecutive memory locations using a minimum of statements. It must be remembered that initially these locations (10 to 17 on page 0) must be set to one less than the first desired address. Because of their characteristics, these locations are called autoindex registers. No incrementation takes place when locations 10 to 17 are addressed directly. For example, if the instruction to be executed next is in location 300 and the data to be referenced is on the page starting at location 5000, autoindex register 10 can be used to address the data as follows:

```
0276    1377    TAD C4777     /=5000-1
0277    3010    DCA 10        /SET UP AUTO INDEX
0300    1410    TAD I 10      /INCREMENT TO 5000
.       .       .             /BEFORE USE AN AN
.       .       .              ADDRESS
.       .       .
0377    4777    C4777,4777
```

When the instruction in location 300 is executed, the contents of location 10 will be incremented to 5000 and the contents of location 5000 will be added to the contents of the accumulator. When the instruction TAD I 10 is executed again, the contents of location 5001 will be added to the accumulator, and so on.

PSEUDO-OPERATORS

The programmer uses pseudo-operators to direct the assembler to perform certain tasks or to interpret subsequent coding in a

certain manner. Some pseudo-ops generate storage words in the object program, other pseudo-ops direct the assembler how to proceed with the assembly. Pseudo-ops are maintained in the permanent symbol table. The function of each PAL8 pseudo-op is described below.

Indirect and Page Zero Addressing

The pseudo-operators I and Z are used to specify the type of addressing to be performed. These were discussed earlier in the chapter.

Radix Control

Numbers used in a source program are initially considered to be octal numbers. However, the programmer may change or alternate the radix interpretation by the use of the pseudo-operators DECIMAL and OCTAL. The DECIMAL pseudo-op interprets all following numbers as decimal until the occurrence of the pseudo-op OCTAL. The OCTAL pseudo-op resets the radix to its original octal base.

Extended Memory

The pseudo-op FIELD instructs the assembler to output a field setting so that it may recognize more than one memory field. This field setting is output during pass 2 and is recognized by the Absolute Loader which in turn causes all subsequent information to be loaded into the field specified by the expression. The form is:

FIELD n

n is an integer, a previously defined symbol, or an expression within the range 0 to 7.

This field setting is output on the binary file during pass 2 followed by an origin setting of 200. This word is read by the ABSLDR when it is executed and begins loading information into the new field.

The field setting is never remembered, in binary, by the assembler and no initial field setting is output. However, it appears as the high-order digit of the Location Counter on the listing. A binary file produced without field settings will be loaded into field 0 when using the ABSLDR.

A symbol in one field may be used to reference the same location in any other field. The field to which it refers is determined by the

use of the CDF and CIF instructions. (The programmer who is unfamiliar with the IOTs but wishes to use them should refer to the PDP/8E SMALL COMPUTER HANDBOOK and experiment with several short test programs to satisfy himself as to their effect.) CDF and CIF instructions must be used prior to any instruction referencing a location outside the current field, as shown in the following example:

```
        *200
        TAD P301
        DCF 00
        CIF 10
        JMS PRINT
        CIF 10
        JMP NEXT
P301,   301
        FIELD 1
        *200
NEXT,   TAD P302
        CDF 10
        JMS PRINT
        HLT
P302,   302
PRINT,  0
        TLS
        TSF
        JMP .-1
        CLA
        RDF
        TAD PCDIF
        DCA .+1
        000
        JMP I PRINT
PCDIF,  CDF CIF 0
```

When FIELD is used, the assembler follows the new FIELD setting with an origin at location 200. For this reason, if the programmer wants to assemble code at location 400 in field 1 he must write:

```
FIELD 1         /CORRECT EXAMPLE
*400
```

The following is incorrect and will not generate the desired code:

```
*400            /INCORRECT
FIELD 1
```

Specifying the /O option to PAL8 inhibits the origin to 200 after a FIELD pseudo-op.

End-Of-File

PAUSE signals the assembler to stop processing the file being read. The current pass is not terminated, and processing continues with the next file. The PAUSE pseudo-op is present mainly for compatability with paper tape assemblers, and its use is optional.

Resetting The Location Counter

The PAGE n pseudo-op resets the location counter to the first address of page n, where n is an integer, a previously defined symbol, or a symbolic expression, whose terms have been defined previously and whose value is from 0 to 37 inclusive. If n is not specified, the location counter is reset to the next logical page of memory. For example:

```
PAGE 2 sets the location counter to 00400
PAGE 6 sets the location counter to 01400
```

If the pseudo-op is used without an argument and the current location counter is at the first location of a page, it will not be moved. In the following example, the code TAD B is assembled into location 00400:

```
*377
        JMP .-3
PAGE
        TAD B
```

If several consecutive PAGE pseudo-ops are given, the first will cause the current location counter to be reset as specified. The rest of the PAGE pseudo-ops will be ignored.

Entering Text Strings

The TEXT pseudo-op allows a string of text characters to be entered as data and stored in 6-bit ASCII by using the pseudo-op TEXT followed by a space or spaces, a delimiting character (must be a printing character), the string of text, and the same delimiting character. Following the last character, a 6-bit zero is inserted as a stop code. For example:

```
TAG,    TEXT/123*/
```

The string would be stored as:

```
6162
6352
0000
```

The IF option inhibits the generation of the extra 6-bit zero character.

Suppressing The Listing

Those portions of the source program enclosed by XLIST pseudo-ops will not appear in the listing file; the code will be assembled, however.

Two XLIST pseudo-ops may be used to enclose the code to be suppressed in which case the first XLIST with no argument will suppress the listing, and the second will allow it again. XLIST may also be used with an expression as an argument; a listing will be inhibited if the expression is equal to zero, or allowed if the expression is not equal to zero. XLIST pseudo-ops never appear in the assembly listing.

Reserving Memory

ZBLOCK instructs the assembler to reserve n words of memory containing zeroes, starting at the word indicated by the current location counter. It is of the form:

```
ZBLOCK n
```

For example:

```
ZBLOCK 40
```

causes the assembler to reserve 40 (octal) words. The n may be an expression. If n=0, no locations are reserved.

Conditional Assembly Pseudo-Operators

The IFDEF pseudo-op takes the form:

```
IFDEF symbol <source code>
```

If the symbol indicated is previously defined, the code contained in the angle brackets is assembled; if the symbol is undefined, this code is ignored. Any number of statements or lines of code may be contained in the angle brackets. The format of the IFDEF statement requires a single space before and after the symbol.

The IFDEF pseudo-op is similar in form to IFDEF and is expressed:

IFNDEF symbol <source code>

If the symbol indicated has not been previously defined, the source code in angle brackets is assembled. If the symbol is defined, the code in the angle brackets is ignored. The IFZERO pseudo-op is of the form:

IFZERO expression <source code>

If the evaluated (arithmetic or logical) expression is equal to zero, the code within the angle brackets is assembled; if the expression is non-zero, the code is ignored. Any number of statements or lines of code may be contained in the angle brackets. The expression may not contain any imbedded spaces and must have a single space preceding and following it. IFNZRO is similar in form to the IFZERO pseudo-op and is expressed:

IFNZRO expression <source code>

If the evaluated (arithmetic or logical) expression is not equal to zero, the source code within the angle brackets is assembled; if the expression is equal to zero, this code is ignored. Pseudo-ops can be nested, for example:

IFDEF SYM <IFNZRO X2 <. . .> >

The evaluation and subsequent inclusion or deletion of statements is done by evaluating the outermost pseudo-op first.

Controlling Binary Output

NOPUNCH causes the assembler to cease binary output but continue assembling code. It is ignored except during pass 2.

ENPUNCH causes the assembler to resume binary output after NOPUNCH, and is ignored except during pass 2. For example, these two pseudo-ops might be used where several programs share the same data on page zero. When these programs are to be loaded and executed together, only one page zero need be output.

Controlling Page Format

The EJECT pseudo-op causes the listing to jump to the top of the next page. A page eject is done automatically every 55 lines;

EJECT is useful if the user requires more frequent paging. If this pseudo-op is followed by a string of characters, the first 50 (octal) characters of that string will be used as a new header line.

Typesetting Pseudo-Operator

DTORG is used in typesetting to output a two frame DECtape block number (4 digits) in the binary tape. The form of this pseudo-op is as follows:

```
DTORG expression
```

The first frame on the binary tape includes channels 7 and 8 punched (in the same manner as a FIELD setting) as a signal to a special typesetting loader that the following data is to be loaded into DECtape block n. The DTORG setting is added into the checksum, unlike the FIELD setting, which is not included. DTORG and FIELD should not be used in the same program.

Calling OS/8 User Service Routine

The pseudo-operators DEVICE and FILENAME may be used by calls to the OS/8 User Service Routine, but have no other meaning to the assembler. The form for these pseudo-ops is:

```
DEVICE name
FILENAME name.extension
```

When using DEVICE, the name can be from 1 to 4 alphanumeric characters. These are trimmed to 6-bit ASCII and packed into 2 words, filled in with zeroes on the right if necessary. With FILENAME (FILENA is also acceptable) the name (or name.-extension) may be from 1 to 6 alphanumeric characters and the optional extension may be 1 or 2 characters. The characters are trimmed to 6-bit ASCII and packed 2 to a word. Three words are allocated for the filename, filled with zeroes on the right if less than 6 characters are specified, followed by one word for the extension. For example:

```
L,      FILENAME ABC.DA
```

is equivalent to the following coding:

```
L,      0102
        0300
        0000
        0401
```

Relocation Pseudo-Op

It is sometimes desirable to assemble code at a given location and then move it to another location for execution. This may result in errors unless the relocated code is assembled in such a way that the assembler assigns symbols their execution-time addresses rather than their load-time addresses. The RELOC pseudo-op establishes a virtual location counter without altering the actual location counter. The line:

```
RELOC expr
```

sets the virtual location counter to expr. The line:

```
RELOC
```

sets the virtual location counter equal to the actual location counter and terminates the relocation section.

Example:

```
          0400              *400
          2000              RELOC 2000
02000*    1377    CODE,     TAD (CODE
02001*    3005              DCA 5
02177*    2000              PAGE
          0600              RELOC
```

The location marked CODE is loaded into location 400, but the assembler treats it as if it were loading into location 2000. The asterisks after the location values indicate that the virtual and the actual location counters differ for that line of code. RELOC always causes current page literals to be dumped.

Altering The Permanent Symbol Table

PAL8 contains a table of symbol definitions for the PDP-8 and OS/8 peripheral devices. These are symbols such as TAD, DCA, and CLA, which are used in most PDP-8 programs. This table is considered to be the permanent symbol table for PAL8.

If the user purchases one or more optional devices whose instruction set is not defined among the permanent symbols (for example EAE or an A/D converter), he would want to add the necessary symbol definitions to the permanent symbol table in every program he assembles.

Conversely, the user who needs more space for user-defined symbols would probably want to delete all definitions except the ones used in his program. For such purposes, PAL8 has three pseudo-ops that can be used to alter the permanent symbol table. These pseudo-ops are recognized by the assembler only during pass 1. During either pass 2 or pass 3 they are ignored and have no effect.

EXPUNGE deletes the entire permanent symbol table, except pseudo-ops.

FIXTAB appends all presently defined symbols to the permanent symbol table. All symbols defined before the occurrence of FIXTAB are made part of the permanent symbol table for the current assembly.

To append the following instructions to the symbol table, the user generates an ASCII file called SYM.PAL containing:

```
MUY=7405     /MULTIPLY
DVI=7407     /DIVIDE
CLSK=6131    /SKIP ON CLOCK INTERRUPT
FIXTAB       /SO THAT THESE WON'T BE
             /PRINTED IN THE SYMBOL TABLE
```

The ASCII file is then entered in PAL8's input designation. The user may also place the definitions at the beginning of the source file. This eliminates the need to load an extra file. Each time the assembler is loaded, PAL8's permanent symbol table is restored.

The third pseudo-op used to alter the permanent symbol table in PAL8 is FIXMRI. FIXMRI is used to define a memory reference instruction and is of the form:

```
FIXMRI name=value
```

The letters FIXMRI must be followed by one space, the symbol for the instruction to be defined, an equal sign, and the value of the symbol. The symbol will be defined and stored in the symbol table as a memory reference instruction. The pseudo-op must be repeated for each memory reference instruction to be defined. For example:

```
EXPUNGE
FIXMRI  TAD=1000
FIXMRI  DCA=3000
CLA=7200
FIXTAB
```

When the preceding program segment is read into the assembler during pass 1, all symbol definitions are deleted and the three symbols listed are added to the permanent symbol table. Notice that CLA is not a memory reference instruction. This process can be performed to alter the assembler's symbol table so that it contains only those symbols used at a given installation or by a given program. This may increase the assembler's capacity for user-defined symbols in the program.

LINK GENERATION AND STORAGE

In addition to handling symbolic addressing on the current page of memory, PAL8 automatically generates links for off-page references. If reference is made to an address not on the page where an instruction is located, the assembler sets the indirect bit (bit 3) and an indirect address linkage will be generated on the current memory page. If the off-page reference is already an indirect one, the error diagnostic II (illegal indirect) will be generated. For example:

```
*2117
A,      CLA
        .
        .
        .
*2600
        JMP A
```

In the example above, the assembler will recognize that the register labelled A is not on the current page (in this case 2600 to 2777) and will generate a link to it as follows:

1. In location 2600 the assembler will place the word 5777 which is equivalent to JMP I 2777.

2. In address 2777 (the last available location on the current page) the assembler will place the word 2117 (the actual address of A).

During pass 3, the octal code for the instruction will be followed by an apostrophe (') to indicate that a link was generated.

Although the assembler will recognize and generate an indirect address linkage when necessary, the programmer may indicate an explicit indirect address by the pseudo-op I. The assembler cannot

generate a link for an instruction that is already specified as being an indirect reference. In this case, the assembler will print the error message II (illegal indirect). For example:

```
*2117
A,      CLA
        .
        .
        .
*2600
        JMP I A
```

The above coding will not work because A is not defined on the page where JMP I A is attempted, and the indirect bit is already set.

Literals and links are stored on each page starting at page address 177 (relative) and extending toward page address 0 (relative). Whenever the origin is then set to another page, the literal buffer for the current page is output. This does not affect later execution. There is room for 160 (octal) literals and links on page zero and 100 (octal) literals on each other page of memory. Literals and links are stored only as far down as the highest instruction on the page. Further attempts to define literals will result in a PE (page exceeded) or ZE (page zero exceeded) error message.

CODING PRACTICES

A neat printout (or program listing, as it is usually called) makes subsequent editing, debugging, and interpretation much easier than if the coding were laid out in a haphazard fashion. The coding practices listed below are in general use, and will result in a readable, orderly listing.

1. A title comment begins with a slash at the left margin.
2. Pseudo-ops may begin at the left margin; often, however, they are indented one tab stop to line up with the executable instructions.
3. Address labels begin at the left margin. They are separated from succeeding fields by a tabulation.
4. Instructions, whether or not they are preceded by a label field, are indented one tab stop.

5. A comment is separated from the preceding field by one or two tabs (as required) and a slash; if the comment occupies the whole line it usually begins with a slash at the left margin.

PROGRAM PREPARATION AND ASSEMBLER OUTPUT

The following program was generated using the OS/8 EDITOR and was assembled with PAL8.

```
/SAMPLE PAL8 PROGRAM
/GETS INPUT FROM KBD,HALTS WHEN "E" IS TYPED
        *200
BEGIN,  KCC
        KSF
        JMP .-1             /WAIT FOR FLAG
        KRB                 /READ IN CHARACTER
        TAD (-"E
        SNA CLA             /IS IT E?
        HLT
        JMP BEGIN+1
/END OF EXAMPLE
$
```

The program consists of statements and pseudo-ops and is terminated by the dollar sign ($). If the program is large, it can be segmented by placing it into several files; this often facilitates the editing of the source program since each section will be physically smaller.

The assembler initially sets the current location counter to 0200. This counter is reset whenever the asterisk (*) is processed.

The assembler reads the source file for pass 1 and defines all symbols used. During pass 2, the assembler reads the source file and generates the binary code using the symbol table equivalences defined during pass 1. The binary file that is output may be loaded by the Load command. This binary file consists of an origin setting and data words.

During pass 3, the assembler reads the source file and generates the code from the source statements. The assembly listing is output in ASCII code. It consists of the current location counter, the generated code in octal, and the source statement. Unless options are chosen to suppress paging or to change the header, the first 50 (octal) characters of the first line of the source program will be used

as a heading for each page followed by the assembler version number, the date and the listing page number. The 5-digit first column is the field number and 4-digit octal address (current location counter); the 4-digit second column is the assembled object code. The symbol table is printed at the end of the pass. The pass 3 output is:

```
/SAMPLE PAL8 PROGRAM

                /SAMPLE PAL8 PROGRAM
                /GETS INPUT FROM KBD,HALTS WHEN "E" IS TYPED
        0200            *200
00200   6032    BEGIN,  KCC
00201   6031            KSF
00202   5201            JMP .-1             /WAIT FOR FLAG
00203   6036            KRB                 /READ IN CHARACTER
00204   1377            TAD (-"E
00205   7650            SNA CLA             /IS IT E?
00206   7402            HLT
00207   5201            JMP BEGIN+1
                /END OF EXAMPLE
00377   7473
                $

/SAMPLE PAL8 PROGRAM

BEGIN  0200
```

TERMINATING ASSEMBLY

PAL8 will terminate assembly and return to the Monitor under any of the following conditions:

1. Normal exit: The end of the source program was reached on pass 2 (or pass 3 if a listing is being generated).

2. Fatal error: One of the following error conditions was found and flagged (see the next section):

   BE DE DF PH SE

3. CTRL/C: If typed by the user, control returns to the Monitor.

PAL8 ERROR CONDITIONS

PAL8 will detect and flag error conditions and generate error messages on the console terminal. The format of the error message is:

```
CODE address
```

where code is a 2-letter code which specifies the type of error, and address is either the absolute octal address where the error occurred or the address of the error relative to the last symbolic tag (if there was one) on the current page. For example, the following code:

```
BEG,    TAD LBL
        %TAD LBL
```

would produce the error message:

```
IC BEG+0001
```

since % is an illegal character.

On the pass 3 listing, error messages are output as 2-character messages on the line just prior to the line in which the error occurred. The following table lists the PAL8 error codes. Those labeled Fatal Error are followed immediately by an effective CTRL/C.

Table 3-3 PAL8 Error Codes

Error Code	Meaning
BE	Two PAL8 internal tables have overlapped. This situation can usually be corrected by decreasing the level of literal nesting or the number of current page literals used prior to this point on the page. Fatal error: assembly cannot continue.
CF	Chain to CREF error. CREF.SV was not found on SYS: .
DE	Device error. An error was detected when trying to read or write a device. Fatal error assembly cannot continue.
DF	Device full. Fatal error: assembly cannot continue.
IC	Illegal character. The character is ignored and the assembly is continued.

Table 3-3 PAL 8 Error Codes (Cont.)

Error Code	Meaning
ID	Illegal redefinition of a symbol. An attempt was made to give a previous symbol a new value by means other than the equal sign. The symbol is not redefined.
IE	Illegal equals. An attempt was made to equate a variable to an expression containing an undefined term. The variable remains undefined.
II	Illegal indirect. An off-page reference was made; a link could not be generated because the indirect bit was already set.
IP	Illegal pseudo-op. A pseudo-op was used in the wrong context or with incorrect syntax.
IZ	Illegal page zero reference. The pseudo-op Z was found in an instruction which did not refer to page zero. The Z is ignored.
LD	The /L or /G options have been specified and the Absolute Loader is not present on the system.
LG	Link generated. This code is printed only if the /E option was specified to PAL8.
PE	Current non-zero page exceeded. An attempt was made to: 1. Override a literal with an instruction. 2. Override an instruction with a literal. 3. Use more literals than the assembler allows on that page. This can be corrected by decreasing either the number of literals on the page or the number of instructions on the page.
PH	Phase error. A conditional assembly bracket is still in effect at the end of the input stream. This is caused by non-matching < and > characters in the source file.
RD	Redefinition. A permanent symbol has been defined with =. The new and old definitions do not match. The redefinition is allowed.
SE	Symbol table exceeded. Too many symbols have been defined for the amount of memory available. Fatal error: assembly cannot continue.

Table 3-3 PAL8 Error Codes (Cont.)

Error Code	Meaning
UO	Undefined origin. An undefined symbol has occurred in an origin statement.
US	Undefined symbol. A symbol has been processed during pass 2 that was not defined before the end of pass 1.
ZE	Page 0 exceeded. This is the same as PE except with reference to page 0.

PAL8 PERMANENT SYMBOL TABLE

The following are the most commonly used elements of the PDP-8 instruction set and are found in the permanent symbol table within the PAL8 Assembler. For additional information on these instructions and for a description of the symbols used when programming other optional devices, see THE SMALL COMPUTER HANDBOOK, available from the DIGITAL Software Distribution Center. (All times are in microseconds and representative of the PDP-8/E.)

Mnemonic	Code	Operation	Time
Memory Reference Instructions			
AND	0000	Logical AND	2.6
TAD	1000	Two's complement add	2.6
ISZ	2000	Increment and skip if zero	2.6
DCA	3000	Deposit and clear AC	2.6
JSM	4000	Jump to subroutine	2.6
JMP	5000	Jump	1.2
IOT	6000	In/Out transfer	—
OPR	7000	Operate	1.2
Group 1 Operate Microinstructions (1 cycle = 1.2 microseconds)			
NOP	7000	No operation	–
IAC	7001	Increment AC	3
BSW	7002	Byte swap	3
RAL	7004	Rotate AC and link left one	4
RTL	7006	Rotate AC and link left two	4

Mnemonic	Code	Operation	Sequence
RAR	7010	Rotate AC and link right one	4
RTR	7012	Rotate AC and link right two	4
CML	7020	Complement the link	2
CMA	7040	Complement the AC	2
CLL	7100	Clear link	1
CLA	7200	Clear AC	1
Group 2 Operate Microinstructions (1 cycle)			
HLT	7402	Halts the computer	3
OSR	7404	Inclusive OR SR with AC	3
SKP	7410	Skip unconditionally	1
SNL	7420	Skip on non zero link	1
SZL	7430	Skip on zero link	1
SZA	7440	Skip on zero AC	1
SNA	7450	Skip on non zero AC	1
SMA	7500	Skip on minus AC	1
SPA	7510	Skip on positive AC (zero is positive)	1
Group 3 Operate Microinstructions			
MQA	7501	Multiplier Quotient OR into AC	
MQL	7421	Load Multiplier Quotient	
SWP	7521	Swap AC and Multiplier Quotient	
Combined Operate Microinstructions			
CIA	7041	Complement and increment AC	2.3
STL	7120	Set link to 1	1.2
GLK	7204	Get link (put link in AC, bit 11)	1.4
STA	7240	Set AC to −1	2.0
LAS	7604	Load AC with SR	2.3

Mnemonic	Code	Operation	Time
Internal IOT Microinstructions			
SKON	6000	Skip with interrupts on and turn them off	
ION	6001	Turn interrupt processor on	1.2
IOF	6002	Turn interrupt processor off	1.2
GTF	6004	Get flags	
RTF	6005	Restore flag, ION	
SGT	6006	Skip if "Greater Than" flag is set	
CAF	6007	Clear all flags	
Keyboard/Reader (1 cycle)			
KCF	6030	Clear keyboard flags	
KSF	6031	Skip on keyboard/reader flag	1.2
KCC	6032	Clear keyboard/reader flag and AC; set reader run	1.2
KRS	6034	Read keyboard/reader buffer (static)	1.2
KIE	6035	Set/clear interrupt enable	
KRB	6036	Clear AC, read keyboard buffer (dynamic), clear keyboard flags	1.2
Teleprinter/Punch (1 cycle)			
TFL	6040	Set teleprinter flag	
TSF	6041	Skip on teleprinter/punch flag	1.2
TCF	6042	Clear teleprinter/punch flag	1.2
TPC	6044	Load teleprinter/punch and print	1.2
TSK	6045	Skip on keyboard or teleprinter flag	1.2
TLS	6046	Load teleprinter/punch, print, and clear teleprinter/punch flag	1.2

Mnemonic	Code	Operation	Time
High Speed Perforated Tape Reader			
RPE	6010	Set Reader/Punch interrupt enable	1.2
RSF	6011	Skip if reader flag=1	1.2
RRB	6012	Read reader buffer and clear flag	1.2
RFC	6014	Clear flag and buffer and fetch character	1.2
High Speed Perforated Tape Punch			
PCE	6020	Clear Reader/Punch interrupt enable	1.2
PSF	6021	Skip if punch flag=1	1.2
PCF	6022	Clear flag and buffer	1.2
PPC	6024	Load buffer and punch character	1.2
PLS	6026	Clear flag and buffer, load buffer and punch character	1.2

chapter 4

sabr

INTRODUCTION

The OS/8 SABR assembler is a modified version of the 8K SABR assembler which is designed to run under the OS/8 Operating System.

The OS/8 SABR assembler can be used as the automatic second pass of the FORTRAN compiler, called separately to do assemblies of FORTRAN compiled files, or used as an independent assembler with its own assembly language. In addition, SABR statements may be used in an OS/8 FORTRAN program, expanding the capabilities of the FORTRAN language.

Calling and Using OS/8 SABR

Unless otherwise specified, the SABR assembler is called automatically by the system to assemble the output of a FORTRAN compilation. At other times the user can call SABR by typing:

```
R SABR
```

in response to the dot printed by the Keyboard Monitor. When the Command Decoder prints an asterisk at the left margin, the user types the appropriate device assignations, I/O files, and any of the acceptable options.

The line to the Command Decoder consists of 0 to 3 output device and file designations, 1 to 9 input device and file designations, and the desired option(s). The form is:

*BINARY,LISTING,MAP<INPUT FILE(S)/OPTION(S)

where BINARY represents the binary output, LISTING the listing output, and MAP the Linking Loader loading map input. Unless alternate extensions are indicated, SABR assumes the following extensions:

File Type	Extension
input file	.SB
binary output	.RL
listing output	.LS

If no binary output file is indicated, no binary output will be generated. However, if the /L or /G options are specified, a binary file will be generated under the assigned name SYS:FORTRL.TM.

OS/8 SABR OPTIONS

The options which can be included in a command string to OS/8 SABR are listed in Table 4-1.

Table 4-1 SABR Options

Option	Meaning
/F	Indicates that the input file is an 8K FORTRAN output file.
/G	Calls the Linking Loader, loads the program into core and begins execution. If a binary output file is not specified, then FORTRL.TM is loaded into core and deleted from the file device. If a starting address is not specified (using the options to the Linking Loader), control is sent to the program entry point MAIN (the FORTRAN compiler gives this name automatically to the main program).
/L	Calls the Linking Loader at the end of the assembly and loads the specified binary file. If a binary output file is not specified, then the temporary file FORTRL.TM is loaded into core and deleted from the file device. The Loader then either returns to the Keyboard Monitor with a core image or asks for more input, depending on whether an ALT MODE or RETURN key has terminated the input line.
/N	Outputs the symbol table but not the rest of the listing (applicable only if a listing file is specified).
/S	Omits the symbol table from the listing (applicable only if a listing file is specified).

When the /L or /G options are specified, any options to the Linking Loader (described in the section concerning the Linking Loader) can be included in the command string for SABR. This does not include the /L (Library) option of the Linking Loader, since it would conflict with the SABR /L option.

NOTE

The FORTRAN compiler automatically generates an entry point named MAIN whose address is the beginning of the program. When writing a main program in SABR, the user should specify the entry point MAIN with the entry pseudo-op in order to symbolically specify the starting address to the Linking Loader. (Otherwise the starting address must be specified to the Loader as a five digit address.)

EXAMPLES OF OS/8 SABR I/O SPECIFICATION COMMANDS

Example 1:

```
.R SABR
*FORTRN.TM/F/G
```

DSK:FORTRN.TM is assembled as a FORTRAN output file and the relocatable binary is loaded and started at the entry point MAIN.

Example 2:

```
.R SABR
*SYS TEERL,TTY:<TEE/S
```

The input file TEE.SB (or TEE) on DSK: is assembled. The relocatable binary goes to the output file TEERL.RL on SYS:, the listing without a symbol table goes to the terminal.

THE CHARACTER SET

ALPHABETIC

In addition to the letters A through Z, the following are considered by SABR to be alphabetic:

[left bracket
] right bracket
\ back slash
↑ up arrow

NUMERIC

SABR recognizes the numbers:

0–9

SPECIAL CHARACTERS

The following printing and non-printing characters are legal:

,	Comma	delimits a symbolic address label
/	Slash	indicates start of a comment
(	Left parenthesis	indicates a literal
"	Quote	precedes an ASCII constant
–	Minus sign	negates a constant
#	Number sign	increases value of preceding symbol by one
	RETURN (carriage return)	terminates a statement
;	Semicolon	terminates an instruction
	LINE FEED	ignored
	FORM FEED	ignored
	SPACE	separates and delimits items on the statement line
	TAB	same as space
	RUBOUT	ignored

All other characters are illegal except when used as ASCII constants following a quote ("), or in comments or text strings.

Legal characters used in ways different from the above, and all illegal characters, cause the error message C (Illegal Character) to be printed by SABR.

STATEMENTS

SABR symbolic programs are written as a sequence of statements and are usually prepared on the terminal, on-line, with the aid of the Symbolic Editor program. SABR statements are virtually format free. Each statement is terminated by typing the RETURN key. (Editor automatically provides a line feed). Two or more statements can be typed on the same line using the semicolon as a separator.

A statement line is composed of one or all of the following elements: label, operator, operand and comment, separated by spaces or tabs (labels require a following comma). The types of elements in a statement are identified by the order of appearance in the line and by the separating or delimiting character which follows or precedes the element.

Statements are written in the general form:

label, operator operand /comment (preceded by slash)

SABR generates one, or possibly more, machine (binary) instructions or data words for each source statement.

An input line may be up to 72_{10} characters long, including spaces and tabs. Any characters beyond this limit are ignored.

The RETURN key (CR/LF) is both an instruction and a line terminator. The semicolon may be used to terminate an instruction without terminating a line. If, for example, the programmer wishes to write a sequence of instructions to rotate the contents of the accumulator (AC) and link (L) six places to the right, it might look like this:

```
.
.
RTR
RTR
RTR
.
.
```

Using the semicolon, the programmer may place all three RTR's on a single line, separating each RTR with a semicolon and terminating the line with the RETURN key. The preceding sequence of instructions could then be written:

```
RTR;RTR;RTR
```
(terminated with the RETURN key)

This format is particularly useful when creating a list of data:

```
0200  0020        LIST,   20;50;-30;62
0201  0050
0202  7750
0203  0062
```

Null lines may be used to format program listings. A null line is a line containing only a carriage return and possibly spaces or tabs. Such lines appear as blank lines in the program listing.

Labels

A label is a symbolic name or location tag created by the programmer to identify the address of a statement in the program. Subsequent references to the statement can be made merely by referencing the label. If present, the label is written first in a statement and terminated with a comma.

```
0200  0000        SAVE,   0
0201  1200        ABC,    TAD SAVE
```

SAVE and ABC are labels referencing the statements in location 0200 and 0201, respectively.

Operators

An operator is a symbol or code which indicates an action or operation to be performed, and may be one of the following:

1. A direct or indirect memory reference instruction
2. An operate or IOT microinstruction
3. A pseudo-operator

All SABR operators, microinstructions and memory reference instructions are summarized in Appendix C.

Operands

An operand represents that part of the statement which is manipulated or operated upon, and may be a numeric constant, a literal or a user-defined address symbol.

In the example last given, SAVE represents an operand.

CONSTANTS

Constants are data used but not changed by a program and are of two types: numeric and ASCII. ASCII constants are used only as parameters. Numeric constants may be used as parameters or as operand addresses, for example:

```
0200  1412              TAD I   12
```

Constant operand addresses are treated as absolute addresses, just as a symbol defined by an ABSYM statement (see Symbol Definition). References to them are not generally relocatable, therefore, they should be used only with great care. The primary use of constant operand addresses is to reference locations on page 0 (see Linkage Routine Locations for free locations on page 0 of each field). All constant operand addresses are assumed to be in the field into which the program is loaded by the Linking Loader.

Constants may not be added to or subtracted from each other or from symbols.

Numeric Constants

A numeric constant consists of a single string of from one to four digits. It may be preceded by a minus sign (−) to negate the constant. The digit string will be interpreted as either octal or decimal according to the latest permanent mode setting by an OCTAL or DECIM pseudo-operator (explained under Assembly Control). Octal mode is assumed at the beginning of assembly. The digits 8 and 9 must not appear in an octal string.

```
0200  5020      A,      5020
0201  7575              -203
                        DECIM
0202  0120              80
```

ASCII Constants

Eight-bit ASCII values may be created as constants by typing the ASCII character immediately following a double quotation

marks ("). A minus sign may be used to negate an alphabetic constant. The minus sign must precede the quotation mark.

```
0200  0273      A,      ";
0201  7477              -"A       /-301
0202  0207              "         /BELL FOLLOWS "
```

The following are illegal as alphabetic constants: carriage return, line feed, form feed and rubout.

LITERALS

A literal is a numeric or ASCII constant preceded by a left parenthesis. The use of literals provides a special and convenient way of generating constant data in a program. The value of the literal will be assembled in a table near the end of the core page on which the instruction referencing it is assembled. The instruction itself will be assembled as an appropriate reference to the location where the numeric value of the literal is assembled. Literals are normally used by TAD and AND instructions, as in the following examples:

```
0200  0376      A,      AND (777
0201  1375              TAD (-50
0202  1374              TAD ("C
      .
      .
      .
0374  0303
0375  7730
0376  0777
```

The numeric conversion mode is initially set to octal, but is controllable with the DECIM and OCTAL pseudo-operators. This mode can be changed on a local basis by inserting a D (decimal) or a K (octal) between the left parenthesis and the constant. For example:

(D32 becomes 0040 (octal)
(K-32 becomes 7746 (octal)

This usage is confined only to the statement in which it is found and does not alter the prevailing conversion mode.

A literal may also be used as a parameter (i.e., with no operator). In this case the numeric value of the literal is assembled as

usual in the literal table near the end of the core page currently being assembled, and a relocatable pointer to the address of the literal is assembled in the location where the literal parameter appeared.

```
0200  0376 01    A,       (20
      .
      .
      .
0376  0020
```

This feature is intended primarily for use in passing external subroutine arguments with the ARG pseudo-operator, which is explained in greater detail later in the chapter.

PARAMETERS

A parameter is generally either a numeric constant, a literal or a user-defined address symbol, which is intended to represent data rather than serve as an instruction. It appears as an operand in a statement line containing no operator. (An exception to this is a parameter used in conjunction with the ARG pseudo-operator, explained in Subroutines.) In the following example, 200 and −320, M, and PGOADR all represent parameters.

```
0200  0200      ABC,     200;-320;"M
0201  7460
0202  0315
0203  0176      POINTR, PGOADR
```

SYMBOLS

Symbols are composed of legal alphanumeric characters and are delimited by a non-alphanumeric character. There are two major types of symbols: permanent, and user-defined.

Permanent Symbols

Permanent symbols are predefined and maintained in SABR's permanent symbol table. They include all of the basic instructions and pseudo-operators in Appendix C. These symbols may be used without prior definition by the user.

User-Defined Symbols

A user-defined symbol is a string of from one to six legal alphanumeric characters delimited by a non-alphanumeric character. User-defined symbols must conform to the following rules:

1. The characters must be legal alphanumerics—ABCD . . . XYZ, [] \ ↑ and 0123456789.
2. The first character must be alphabetic.
3. Only the first six characters are meaningful. A symbol such as INTEGER would be interpreted as INTEGE. Since the symbols GEORGE1 and GEORGE2 differ only in the seventh character, they would be treated as the same symbol: GEORGE.
4. A user-defined symbol cannot be the same as any of the pre-defined permanent symbols.
5. A user-defined symbol must be defined only once. Subsequent definitions will be ineffective and will cause SABR to type the error message M (Multiple Definition).

A symbol is defined when it appears as a symbolic address label or when it appears in an ABSYM, COMMN, OPDEF or SKPDF statement (see Pseudo-Operators). No more than 64 different user-defined symbols may occur on any one core page.

Equivalent Symbols

When an address label appears alone on a line—with no instruction or parameter—the label is assigned the value of the next address assembled.

```
TAG1,
TAG2,    30
TAG3,
```

TAG1 and TAG2 are equivalent symbols in that they are assigned the same value. Therefore, a TAD TAG1 will reference the data at TAG2. TAG3, however, is not equivalent to TAG2. TAG3 would be defined as 1 greater than TAG2.

Comments

A programmer may add notes to a statement by preceding them with a slash mark. Such comments do not affect assembly or program execution but are useful in interpreting the program listing

for later analysis and debugging. Entire lines of comments may be present in the program.

None of the special characters or symbols have significance when they appear in a comment.

```
/THIS IS A COMMENT LINE.
/THIS ALSO. TAD;CALL;#"-2C+=!
A,      TAD SAVE          /SLASH STARTS COMMENT
```

INCREMENTING OPERANDS

Because SABR is a one-pass assembler and also because it sometimes generates more than one machine instruction for a single user instruction, operand arithmetic is impossible. Statements of the form:

```
TAD TAG+3
TAD LIST-LIST2
JMP .+6
```

are illegal. However, by appending a number sign to an operand the user can reference a location exactly one greater than the location of the operand (the next sequential location): TAD LOC# is equivalent to the PAL language statement TAD LOC+1.

```
0200  0020      LOC,    20
0201  0030              30
0202  1200      START,  TAD LOC    /GET 20
0203  1201              TAD LOC#   /GET 30
                        PAGE
0400  0200      A,      LOC
0401  0201      B,      LOC#
```

In assembling #-type references SABR does not attempt to determine if multiple machine code words are generated at the symbolic address referenced.

```
START,  TAD I   LOC     /LOC IS OFF-PAGE
        NOP             /USER HOPES TO MODIFY
         .
         .
        TAD     (7500   /SMA
        DCA     START#
```

In the preceding example the user wishes to change the NOP instruction to an SMA. However, this is not possible because TAD I LOC will be assembled as three machine code words; if START is at 0200, the NOP will be at 0203. The SMA will be inserted at 0201, thus destroying the second word of the TAD I LOC execution.

To avoid this error, the user should carefully examine the assembly listing before attempting to modify a program with #-type references. In the previous example the proper sequence is:

```
0202  4067       START,  TAD I LOC
0203  0200 01
0204  1407
0205  7000       VAR,    NOP
0206  1377               TAD (7500
0207  3205               DCA VAR
0377  7500
```

The #-sign feature is intended primarily for manipulating DUMMY variables when picking up arguments from external subroutines and returning from external subroutines (see Passing Subroutine Arguments).

PSEUDO-OPERATORS

Table 4-2 lists all the pseudo-operators available in SABR, whether used as a free-standing assembler, or in conjunction with the Fortran compiler. The pseudo-operators are categorized and explained in the following paragraphs.

Table 4-2 SABR Pseudo-Operators

Mnemonic Code	Operation
ABSYM	Direct absolute symbol definition, used to indicate an absolute core address. For example: `ABSYM TEM 177  /PAGE ZERO ADDRESS`

ARG	Argument for subroutine call, indicating a value to be transmitted, one value per ARG statement. Used only with CALL. For example:
	
	`N1,  ARG (50`
	`N2,  ARG LOCATN`

BLOCK	Reserve storage block; reserves n words of core by placing zeros in them. For example:
	
	`BLOCK 200  /RESERVE 300`
	`BLOCK 100  /(OCTAL) LOCATIONS`

CALL	Call external subroutine. For example: `CALL 2,SUBR` where 2 is the number of arguments to be passed and SUBR is the subroutine name.
COMMN	Common storage definition, used to name locations in field 1 as externals to be referenced by any program. For example: `A,  COMMN 20  /20 WORDS IN COMMON`
CPAGE	Check if page will hold data, followed by the number of words of code which must be kept together in a unit on a page. That number of words following the CPAGE will be assembled as a unit on the next available core page.
DECIM	Decimal conversion, numeric conversion interprets all numbers input as being decimal numbers.

Table 4-2 SABR Pseudo-Operators (Cont.)

Mnemonic Code	Operation

DUMMY	Dummy argument definition, used in passing arguments to and from subroutines. DUMMY variables are defined in the subprograms which reference them. For example:
	
	`ENTRY A1`
	`DUMMY X`
	`DUMMY Y`

EAP	Enter automatic paging mode, restore automatic paging (See LAP).
END	End of program or subprogram.

ENTRY	Define program entry point, used at beginning of subprograms to give name of entry point for the Linking Loader. For example:
	
	`ENTRY SUBROU`
	`SUBROU, BLOCK 2`

FORTR	Assemble FORTRAN tape.
I	Symbolic representation for indirect addressing. For example: `DCA I ADD`
IF	Conditional assembly, of form: `IF NAME, 7` If the symbol NAME has been previously defined, the statement has no effect. If NAME is not defined, the next 7 symbolic instructions are not assembled.
LAP	Leave automatic paging. Assembler is initially set for automatic jumps to the next core page when the current page is full (or upon REORG or PAGE statements). This feature can be suppressed with LAP.

Table 4-2 SABR Pseudo-Operators (Cont.)

Mnemonic Code	Operation
OCTAL	Octal conversion, numeric conversion is originally set to octal and can be changed back to octal after a DECIM pseudo-op has been used.
OPDEF	Define non-skip operator. For example: `OPDEF DTRA 6761`
PAGE	Terminate current page, begin assembly of succeeding instructions on next core page.
PAUSE	Pause for next tape, designed to allow large source tapes to be broken into several smaller segments. Assembly is continued by pressing the CONT switch.
REORG	Terminate page and reset origin; origin settings are always to the first address of a page. For example: `REORG 1000`
RETRN	Return from external subroutine, the name of the subroutine being left must be specified. Before the RETRN statement is used, the pointer in the second word of the subprogram entry must be incremented to the point following all arguments in the calling program (after the CALL statement).
SKPDF	Define skip-type operator. For example: `SKPDF DTSF 6771`

TEXT	Text string similar to BLOCK, except that the argument is a text string. Characters are stored in six-bit stripped ASCII with a printing character used to delimit the string. For example:
	
	`TAG,  TEXT /123*/`
	
	the string would be stored as:
	
	`6162`
	`6352`

Table 4-2 SABR Pseudo-Operators (Cont.)

Mnemonic Code	Operation
	Odd characters are filled with zeros on the right.
	The floating-point accumulator (in field 1).
ACH	High-order word.
ACM	Middle word.
ACL	Low-order word.

Assembly Control

END — Every program or subprogram to be assembled must contain the END pseudo-op as its last line. If this requirement is not met, an error message (E) is given.

PAUSE — The PAUSE pseudo-op causes assembly to halt and is designed to allow the programmer to break up a large source tape into several smaller segments. To do this, the programmer need only place a PAUSE statement at the end of each section of his source program except the last. Each of these sections of the program is then output as an individual tape. When assembly halts at a PAUSE, the user removes the source tape just read from the reader and inserts the next one. Assembly may then be continued by pressing the CONTinue switch.

WARNING

The PAUSE pseudo-op is designed specifically for use at the end of partial tapes and should not be used otherwise.

The reason for this is that the reader routine may have read data from the paper tape into its buffer that is actually beyond the PAUSE statement.

Consequently, when CONTinue is pressed after the PAUSE is found by the line interpreting routine, the entire content of the reader buffer following the PAUSE is destroyed, and the next tape begins reading into a fresh buffer. Thus, if there is any meaningful data on the tape beyond the PAUSE statement, it will be lost.

DECIM — Initially the numeric conversion mode is set for octal conversion. However, if the user wishes, he may change it to decimal by use of the DECIM pseudo-op.

OCTAL — If the numeric conversion mode has been set to decimal, it may be changed back to octal by use of the OCTAL pseudo-op.

No matter which conversion mode has been permanently set, it may always be changed locally for literals by use of the (D or (K syntax described earlier. For example:

```
0200  0320       START,  320
                           DECIM
0201  0500               320
0202  0377 01             (K320
0203  1000               512
                           OCTAL
0204  0512               512
0205  0376 01             (D512
0206  0320               320
  .
  .                        END
  .
0376  1000
0377  0320
```

LAP — The assembler is initially set for automatic generation of jumps to the next core page when the page being assembled fills up (Page Escapes), or when PAGE or REORG pseudo-ops are encountered. This feature may be suppressed by use of the LAP (Leave Automatic Paging) pseudo-op.

EAP — If the user has previously suppressed the automatic paging feature, it may be restored to operation by use of the EAP (Enter Automatic Paging) pseudo-op.

PAGE

The PAGE pseudo-op causes the current core page to be assembled as is. Assembly of succeeding instructions will begin on the next core page. No argument is required.

REORG

The REORG pseudo-op is similar to the PAGE pseudo-op, except that a numerical argument specifying the relative location within the subprogram where assembly of succeeding instructions is to begin must be given. A REORG below 200 may not be given. A REORG should always be to the first address of a core page. If a REORG address is not the first address of a page, it will be converted to the first address of the page it is on.

```
0200   7200        START,  CLA
                           PAGE
0400   7040                CMA
                           REORG  1000
1000   7041                CIA
```

CPAGE

The CPAGE pseudo-op followed by a numerical argument N specifies that the following N words of code[1] must be kept together in a single unit and not be split up by page escapes and literal tables. If the N words of code will not fit on the current page of code, the current page is assembled as if a

PAGE pseudo-op had been encountered. The N words of code will then be assembled as a unit on the next core page. An example follows.

NOTE

N must be less than or equal to 200 (octal) in nonautomatic paging mode or less than or equal to 176 octal in automatic paging mode.

[1]Normally data. However, if these N words are instructions, for example a CALL with arguments, it is the user's responsibility to count extra machine instructions which must be inserted by SABR.

```
0200  7200      START,  CLA
                        LAP          /INHIBIT PAGE ESCAPE
                        CPAGE 200    /CLOSES THE
0400  0000              NAME1        /CURRENT PAGE
0401  0000              NAME2        /AND ASSEMBLES
                                     /THE NEXT PAGE
                        .
                        .
```

IF

The conditional pseudo-op, IF, is used with the following syntax:

```
IF NAME, 7
```

The action of the pseudo-op in this case is to first determine whether the symbol NAME has been previously defined. If NAME is defined, the pseudo-op has no effect. If NAME is not defined, the next seven symbolic instructions (not counting null lines and comment lines) will be treated as comments and not assembled.

```
                        /ABSYM NAME 176
                        IF NAME, 2          /THE NEXT LINE
                                CLL RTL     /TO BE ASSEMBLED
                                RAL         /WILL BE "DCA LOC"
                        /IF THE SLASH BEFORE "ABSYM NAME 176"
                        /IS REMOVED, THE "CLL RTL" AND "RAL"
                        /WILL BE ASSEMBLED.

0200  3201              DCA LOC
0201  0000      LOC,    0
                        .
                        .
                        .
```

Normally the symbol referenced by an IF statement should be either an undefined symbol or a symbol defined by an ABSYM statement. If this is done, the situation mentioned below cannot occur.

WARNING

In a situation such as the following, a special restriction applies.

```
NAME,   Ø
.
.
.
IF NAME, 3
```

The restriction is that if the line NAME, 0 happens to occur on the same core page of instructions as the IF statement, then, even though it is before the IF statement, NAME will not have been previously defined when the IF statement is encountered, and on the first pass (though not in the listing pass) the three lines after the IF statement will not be assembled. The reason for this is that location tags cannot be defined until the page on which they occur is assembled as a unit.

Symbol Definition

ABSYM — An absolute core address may be named using the ABSYM pseudo-op. This address must be in the same core field as the subprogram in which it is defined. The most common use of this pseudo-op is to name page zero addresses not used by the operating system. These addresses are listed under Linkage Routine Locations.

OPDEF
SKPDF — Operation codes not already included in the symbol table may be defined by use of the OPDEF or SKPDF pseudo-ops. Non-skip instructions must be defined with the OPDEF pseudo-op and skip-type instructions must be defined with the SKPDF pseudo-op.

Examples of ABSYM, OPDEF and SKPDF syntax:

```
0177    ABSYM TEM       177   /PAGE Ø ADDRESSES
ØØ1Ø    ABSYM AX        1Ø
6761    OPDEF DTRA      6761  /NON-SKIP INSTR.
6771    SKPDF DTSF      6771  /SKIP-TYPE INSTR.
7540    SKPDF SMZ       7540
```

NOTE

ABSYM, OPDEF and SKPDF definitions must be made before they are used in the program.

COMMN

The COMMN pseudo-op is used to name locations in field 1 as externals so that they may be referenced by any program. If any COMMN statements are used, they must occur at the beginning of the source, before everything else including the ENTRY statement. Common storage is always in field 1 and is allocated from location 0200 upwards. Since the top page of field 1 is reserved, no more than 3840_{10} words of common storage may be defined.

A COMMN statement normally takes a symbolic address label, since storage is being allocated. However, common storage may be allocated without an address label.

A COMMN statement always takes a numerical argument which specifies how many words of common storage are to be allocated; however, a 0 argument is allowed. A COMMN statement with 0 argument allocates no common storage; it merely defines the given location symbol at the next free common location.

The syntax of the COMMN statement is shown as follows:

```
0200      A,      COMMN 20
0220      B,      COMMN 10
0230              COMMN 300
0530      C,      COMMN 0
0530      D,      COMMN 10
                  ENTRY SUERUT
```

In this example 20 words of common storage are allocated from 0200 to 0217, and A is defined at location 0200. Then, 10 words are allocated

from 0220 to 0227, and B is defined at 0220. Notice that if A is actually a 30 word array, this example equates B(1) with A(21).

The example continues by allocating common storage from 0230 to 0527 with no name being assigned to this block. Then 10 words are allocated from 0530 to 0537 with both C and D being defined at 0530.

Data Generating

BLOCK

The BLOCK pseudo-op given with a numerical argument N will reserve N words of core by placing zeros in them. This pseudo-op creates binary output, and thus may have a symbolic address label.

Before the N locations are reserved, a check is made to see if enough space is available for them on the current core page. If not, this page is assembled and the N locations are reserved on the next core page. The action here is similar to that of the CPAGE pseudo-op. Similar restrictions on the argument apply.

```
/EXAMPLE OF HOW LARGE BLOCK STORAGE
/MAY BE ACHIEVED WITHIN A SUBPROGRAM AREA

        LAP             /INHIBIT PAGE ESCAPES
        BLOCK 200       /RESERVE 500
        BLOCK 200       /(OCTAL) LOCATIONS
        BLOCK 100
        EAP             /RESUME NORMAL CODING
```

As a special use, if the BLOCK pseudo-op is used with a location tag (but with no argument or a zero argument), no code zeros are assembled; instead the symbolic address label is made equivalent to the next relative core location assembled. (This is equivalent to using a symbolic address label with no instruction on the same line.)

```
0200  0000      LIST,   BLOCK 3   /ASSEMBLES AS
0201  0000
0202  0000                        /THREE ZEROS
                                  /WITH "LIST"
                                  /DEFINED AT THE
                                  /FIRST LOCATION
                NAME1,  BLOCK     /DEFINES NAME1=
                NAME2,  BLOCK 0   /NAME2=NAME3=
                NAME3,            /NAME4
0203  0000      NAME4,  BLOCK 2
0204  0000
```

TEXT

The TEXT pseudo-op is used to obtain packed six-bit ASCII text strings. Its function and use are almost exactly the same as for the BLOCK pseudo-op except that instead of a numerical argument, the argument is a text string. In particular, a check is made to be sure that the text string will fit on the current page without being interrupted by literals, etc.

The text string argument must be contained on the same line as the TEXT pseudo-op. Any printing character may be used to delineate the text string. This character must appear at both the beginning and the end of the string. Carriage return, line feed and form feed are illegal characters within a text string (or as delineators). All characters in the string are stored in simple stripped six-bit form. Thus, a tab character (ASCII 211) will be stored as an 11, which is equivalent to the coding for the letter I. In general, characters outside the ASCII range of 240–337 should not be used.

```
0200  2405      TAG,    TEXT  /TEXT EXAMPLE 123*;?/
0201  3024
0202  4005
0203  3001
0204  1520
0205  1405
0206  4061
0207  6263
0210  5273
0211  7700
```

SUBROUTINES

A subroutine is a subprogram which performs a specific operation and is generally designed so that it can be used more than once or by more than one program. Direction of flow goes from the main, or calling, program to the subroutine, where the action is performed, followed by a return back to the address following the subroutine call in the main program.

Internal subroutines are those subroutines which can only be called from within a program. This type of subroutine is used extensively in nearly all PDP-8 programs, and is handled through the use of the JMS, JMS I, and JMP I instructions. An example of an internal subroutine call follows:

```
0200   7300       START,  CLA CLL
0201   1204               TAD N       /GET NUMBER IN AC
0202   4206               JMS TWO     /TRANSFER TO SUB-
                                      /ROUTINE
0203   3205               DCA RESLT   /STORE NUMBER
                                      /(CONTROL RETURNS
                                      /HERE)
0204   0001       N,      1
0205   0000       RESLT,  0

                  /SUBROUTINE
0206   0000       TWO,    0
0207   7104               CLL RAL     /ROTATE LEFT AND
                                      /MULTIPLY BY 2
0210   7430               SZL         /CHECK FOR OVERFLOW
0211   7402               HLT         /STOP IF OVERFLOW
0212   6201 05            JMP I TWO   /RETURN TO MAIN
0213   5606
                                      /PROGRAM
                          END
```

The main program picks up a number (N) and jumps to the subroutine (TWO) where N is multiplied by two. A check is made, and if there is no overflow, control returns to the main program through the address stored at the location TWO.

External subroutines are distinguished from internal subroutines by the fact that they may be called by a program which has been compiled, or assembled, without any knowledge of where the subroutine will be located in core memory. Thus, external subroutines must be loaded with a relocatable linking loader. This makes it possible for a programmer to build a library of frequently

used programs and subroutines which can be combined in various configurations, and eliminates the need to reassemble, or recompile, each individual program when a minor change is made in the system.

A call to an external subroutine can be illustrated using the following FORTRAN programs:

```
      IPARM=5                                    (Calling Program)
      CALL TWO(IPARM)
      WRITE (1,100) IPARM
100   FORMAT (I5)
      END

      SUBROUTINE TWO(IARG)                       (Subroutine)
      IARG=IARG+IARG
      RETURN
      END
```

NOTE

Care should be exercised when naming a function or subroutine. It must not have the same name as any of the assembler mnemonics or pseudo-ops or FORTRAN/SABR library functions or subroutines, as errors are likely to result. The symbol table for SABR Assembler is listed in Appendix C, and the library functions are described in the section The Subprogram Library.

Any time a subroutine is called, it must have data to process. This data is contained in parameters in the calling program which are then passed to the subroutine. The data is picked up by the subroutine where it is referred to as arguments. (The subroutine actually picks up the arguments by a series of TAD I's, and one final TAD I for an integer argument, or by a call to the IFAD subroutine if a floating point argument. This is illustrated in the section entitled SABR Programming Notes.) SABR has special pseudo-operators which facilitate the passing/handling of arguments, and each will be explained in turn.

CALL and ARG

The CALL pseudo-op is used by the main program to transfer control to the subroutine and is of the form:

where n represents a one or two-digit number (62_{10} maximum) indicating the number of parameters to be passed to the subroutine, and NAME (separated from n by a comma) represents the symbolic name of the subroutine entry point.

The Assembler must know the number of parameters which follow the call so that enough room on the current page can be allowed. The CALL pseudo-op and its corresponding parameters must always be coded on the same memory page; that is, there must be no intervening page escapes. (Page format and page escapes are discussed later in the chapter.)

The ARG pseudo-op is used only in conjunction with CALL and consists of the symbol ARG followed by one of the parameters (referred to as arguments in the subroutine) to be passed. One ARG statement must be coded for each parameter.

In the previous FORTRAN example, the main program (or it may have been a subroutine) called a subroutine named TWO, and supplied one argument:

```
CALL 1,TWO
ARG  IPARM
.
.
.
```

SABR actually assembles the above instructions as follows (the user may wish to consult the section concerning the Loader Relocation Codes):

```
0200  0000       IPARM,  BLOCK 1
.
.
.
0206  4033                CALL 1,TWO
0207  0103 06
0210  6201 05             ARG  IPARM
0211  0200 01
.
.
.
                  END
```

ENTRY and RETRN

In the subroutine, the ENTRY statement must occur before the name of the entry point appears as a symbolic address label. The actual entry location must be a two-word reserved space so that both the return address and field can be saved when the routine is called. Execution of the subroutine begins at the first location following the two-word ENTRY block. For example, the TWO subroutine mentioned in the previous example would begin as follows:

```
                              ENTRY TWO
0200   0000          TWO,     BLOCK 2
0201   0000
.
.
.
0227   4040                   RETRN TWO
0230   0001 06
                              END
```

When a subroutine is referenced in a CALL statement, the Run-Time Linkage Routine LINK executes the transfer to the subroutine. It assumes that the entry point to the routine is a two-word block. Into the first word of this block it places a CDF instruction which specifies the field of the calling program. In the second word it places the address from which the CALL occurred. (This is analogous to the operation of the JMS instruction.) In the previous example, if the MAIN program had been in field 0, a 6201 would have been deposited in the location at TWO, and a 0210 at TWO #.

The RETRN statement allows the user to return to the calling program from the subroutine. The name of the subroutine being returned from must be specified in the RETRN statement so that the Return Linkage Routine can determine the action required, and also because a subroutine may have differently named ENTRY points. (This is analagous to the operation of a JMP I instruction.)

When a subroutine is entered, the second word of the entry name block contains the address of the argument or next instruction immediately following the subroutine call in the calling program, and it is to this address that control returns.

Example

A user wishes to write a long main program, MAIN[2], which uses two major subroutines, S1 and S2. S1 requires two arguments and S2 one argument. The user writes MAIN, S1, and S2 as three separate programs in the following manner:

```
        ENTRY MAIN
MAIN,   CLA             /START OF MAIN
        .
        .
        CALL 2,S1
        ARG X
        ARG Y
        CALL 1,S2
        ARG Z
        .
        .
        END


        ENTRY S1
S1,     BLOCK 2
        .
        .
        RETRN S1
        END

        ENTRY S2
S2,     BLOCK 2
        .
        .
        .
        RETRN S2
        END
```

S1 could also contain calls to S2, or S2 calls to S1. Each of these programs is independently assembled with SABR and loaded with the Linking Loader. During the loading process, all of the proper addresses will be saved in tables so that when the user begins execution of MAIN, the Run-Time Linkage Routines (see SABR

[2] A useful procedure in SABR programming is to provide an ENTRY point named MAIN in the main program at the address where execution is to begin. This assures that the starting address of the program will appear in the Linking Loader's symbol print-out where it may be easily referenced. If using OS/8, execution will begin at this address automatically, eliminating the need to specify a 5-digit starting address.

Operating Characteristics), which were automatically loaded, will be able to execute the proper reference. Thus, MAIN will be able to fully use S1 and S2 and be able to pass data to and receive it from them.

Passing Subroutine Arguments

DUMMY

A DUMMY pseudo-op is used in SABR to define a two word block which contains an argument address. Indirect instructions are used to pass arguments to and from subroutines through these DUMMY variables. If a DUMMY variable is referenced indirectly, it causes a CALL to the DUMMY Variable Run-Time Linkage Routine (see Run-Time Linkage Routines) which assumes that the DUMMY variable is a two-word reserved space where the first word is a 62N1 (CDF N), with N representing the field of the address to be referenced, and that the second word contains a 12-bit address.

As an example, consider the FORTRAN subroutine TWO shown earlier. This could be written in SABR as follows (the user may wish to refer to the section concerning the Subprogram Library):

```
                    /CALLED BY:  CALL TWO (IARG)

                             ENTRY TWO     /DEFINE THE
                                           /ENTRY PT. USED
                             DUMMY IARG    /TO PICK UP ARG.
0200  0000        IARG,    BLOCK 2
0201  0000
0202  0000        TWO,     BLOCK 2         /ENTRY POINT
0203  0000
0204  4067                 TAD I TWO
0205  0202 01
0206  1407
0207  2203                 INC TWO#        /GET ARG ADDRESS
0210  3200                 DCA IARG
0211  4067                 TAD I TWO
0212  0202 01
0213  1407
0214  2203                 INC TWO#
0215  3201                 DCA IARG#
0216  4067                 TAD I IARG      /GET ARGUMENT
0217  0200 01
0220  1407
```

```
                                        /INTO AC
0221  4067              TAD I IARG      /ADD IT AGAIN
0222  0200 01
0223  1407
0224  4067              DCA I IARG      /RETURN ARG. TO
0225  0200 01
0226  3407
                                        /CALLING PROGRAM
0227  4040              RETRN TWO
0230  0001 06
                        END
```

A second example may be one in which a user has written a FORTRAN program which contains a call to a SABR subroutine ADD:

```
      A=2
      N=3
      CALL ADD(A,N,C)
      WRITE (1,20)C
20    FORMAT (' THE SUM IS',F6.1)
      STOP
      END
```

The FORTRAN program is compiled and the resulting SABR code translates the subroutine call as follows:

```
0223  4033       CALL 3,ADD
0224  0305 06
0225  6201 05    ARG A
0226  0200 01
0227  6201 05    ARG N
0230  0203 01
0231  6201 05    ARG C
0232  0204 01
```

The CALL statement defines 3 parameters—A, N, and C, and the subroutine name ADD. The subroutine itself would appear as follows (the DUMMY variables X, K, and Z facilitate the passing of the arguments to and from the subroutine):

```
                        /CALLED BY:  CALL ADD (X,K,Z)
                               ENTRY ADD
                               DUMMY X
                               DUMMY K
                               DUMMY Z
0200  0000        X,       BLOCK 2
0201  0000
0202  0000        K,       BLOCK 2
0203  0000
0204  0000        Z,       BLOCK 2
0205  0000
0206  0200 01     XPNT,    X
0207  0000        PNTR,    0
0210  0000        CNTR,    0
0211  0000        ADD,     BLOCK 2        /ENTRY POINT
0212  0000
0213  1206                 TAD XPNT
0214  3207                 DCA PNTR
0215  1377                 TAD (-6
0216  3210                 DCA CNTR
0217  4067        A1,      TAD I ADD
0220  0211 01
0221  1407
0222  2212                 INC ADD#
0223  6201 05              DCA I PNTR
0224  3607
0225  2207                 INC PNTR
0226  2210                 ISZ CNTR
0227  5217                 JMP A1
0230  4067                 TAD I K        /GET 2ND ARG
0231  0202 01
0232  1407
0233  4033                 CALL 0,FLOT    /CONVERT TO
0234  0002 06
                                            /FLOATING PT.
0235  4033                 CALL 1,IFAD    /ADD 1ST ARG
0236  0103 06
0237  6201 05              ARG X
0240  0200 01
0241  4033                 CALL 1,ISTO    /RETURN RESULT
0242  0104 06
0243  6201 05              ARG Z
0244  0204 01
0245  4040                 RETRN ADD
0246  0001 06
0377  7772
                           END
```

The COMMN pseudo-op may be used to specify variables as externals so that they may be referenced by any program. This pseudo-op has been explained under Symbol Definition; an example of its usage is included here.

```
      0200      C,      COMMN 3          /RESERVES COMMON
                                         /STORAGE
                        ENTRY CSQR       /DEFINES ENTRY PT.
0200  0000      CSQR,   BLOCK 2          /ACTUAL ENTRY POINT
0201  0000
0202  4033              CALL 1,FAD       /GET THE ARGUMENT
0203  0102 06
0204  6211              ARG C
0205  0200
0206  4033              CALL 1,FMP       /MULTIPLY IT
0207  0103 06
0210  6211              ARG C
0211  0200
0212  4033              CALL 1,STO       /REPLACE WITH RESULT
0213  0104 06
0214  6211              ARG C
0215  0200
0216  4040              RETRN CSQR       /RETURN TO CALLING
0217  0001 06
                                         /PROGRAM
                        END
```

This subroutine computes the square of a variable C. C resides in field 1 in common storage where it can be referenced by any calling program through argument passing. The above is equivalent to the FORTRAN subroutine:

```
SUBROUTINE CSQR
COMMON C
C=C*C
RETURN
END
```

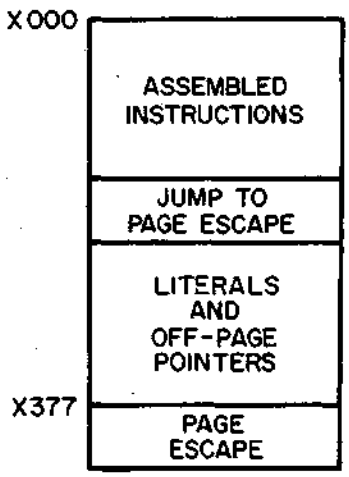
SABR OPERATING CHARACTERISTICS

Page-by-Page Assembly

SABR assembles page-by-page rather than one instruction at a time. To accomplish this it builds various tables as instructions are read. When a full page of instructions has been collected (counting literals, off-page pointers and multiple word instructions) the page is assembled and punched. Several pseudo-operators are available to control page assembly.

PAGE FORMAT

A normal assembled page of code is formatted as below:



Literals and off-page pointers are intermingled in the table at the end of the page.

PAGE ESCAPES

SABR is normally in automatic paging mode: it connects each assembled core page to the next by an appropriate jump. This is called a page escape. For the last page of code, SABR leaves the Automatic Paging Mode and issues no page escape. The LAP (Leave Automatic Paging) pseudo-operator turns off the automatic paging mode. EAP (Enter Automatic Paging) turns it back on if it has been turned off.

Two types of page escape may be generated depending on whether or not the last instruction is a skip. If the last instruction is not a skip, the page escape is as follows:

```
        last instruction (non-skip)
        5377 (JMP to x177)
        literals
        and
        off-page
        pointers
x177/NOP
```

If the last instruction on the page is a skip type, the page escape takes four words, as follows:

```
        last instruction (a skip)
        5376 (JMP to x176)
        5377 (JMP to x177)
        literals
        etc.
x176/SKP
x177/SKP
```

Multiple Word Instructions

Certain instructions in the source program require SABR to assemble more than one machine language instruction (e.g., off-page indirect references and indirect references where a data field re-setting may be required). In the listing, the source instruction will appear beside the first of the assembled binary words.

A difficulty arises when a multiple word instruction follows a skip instruction. The user need be aware that extra instructions are automatically assembled to enable the skip to be effected correctly.

Run-Time Linkage Routines

These routines are loaded by the Linking Loader and perform their tasks automatically when certain pseudo-ops or coding sequences are encountered in the user program. The user needs knowledge of them only to better understand the program listing. (The user may wish to refer to the section entitled Loader Relocation Codes.)

There are seven linkage routines:

1. Change data field to current and skip	CDFSKP
2. Change data field to 1 (common) and skip	CDZSKP
3. Off-page indirect reference linkage	OPISUB
4. Off-bank (common) indirect reference linkage	OBISUB
5. Dummy variable indirect reference linkage	DUMSUB
6. Subroutine call linkage	LINK
7. Subroutine return linkage	RTN

The individual linkage routines function as follows:

1. CDFSKP is called when a direct off-page memory reference follows a skip-type instruction requiring the data field to be reset to the current field.

Program	Assembled Code	Meaning
SZA	7440	
DCA LOC	4045	call CDFSKP
	7410	SKP in case AC = 0 at .−2
	3776	execute the DCA via a pointer near the end of the page.

2. CDZSKP is called when a direct memory reference is made to a location in common (which is always in Field 1). The action of CDZSKP is the same as that of CDFSKP except that it always executes a CDF 10 instead of a CDF current (see Loader Relocation Codes).

Program	Assembled Code	Meaning
SZA	7440	
DCA CLOC	4051	call CDZSKP
	7410	SKP in case AC = 0 at .−2
	3776	execute the DCA via a pointer near the end of the page.

3. OPISUB is called when there is an indirect reference to an off-page location.

Program	Assembled Code	Meaning
DCA I PTR	4062	call OPISUB
	0300 01	relative address of PTR
	3407	execute the DCA I via 0007

4. OBISUB is called when there is an indirect reference to a location in common storage. In such a case it is assumed that the

location in common which is being indirectly referenced points to some location that is also in common.

Program	Assembled Code	Meaning
DCA I CPTR	4055	call OBISUB
	1000	address of CPTR in Field 1
	3407	execute the DCA I via 0007

5. DUMSUB is called when there is an indirect reference to a DUMMY variable. In such a case, DUMSUB assumes that the DUMMY variable is a two-word vector in which the first word is a 62N1, where N = the field of the address to be referenced, and the second word is the actual address to be referenced.

Program	Assembled Code	Meaning
DCA I DMVR	4067	call DUMSUB
	0300 01	relative address of DMVR
	3407	execute DCA I via pointer in location 0007

6. LINK is called to execute the linkage required by a CALL statement in the user's program. When a CALL statement is used, it is assumed that the entry point of the subprogram is named in the CALL and that this entry point is a two-bit word, free block followed by the executable code of the subprogram. LINK leaves the return address for the CALL in these two words in the same format as a DUMMY variable.

Program	Assembled Code	Meaning
CALL 2, SUBR	4033	call LINK
	0205 06	code word
ARG X	62M1	X resides in field M
	0300 01	relative address of X
ARG C	6211	C is in common
	1007	absolute address of C

7. RTN is called to execute the linkage by a RETRN statement in the user's program.

Program	Assembled Code	Meaning
RETRN SUBR	4040	call RTN
	0005 06	number of the subprogram being returned from (SUBR)

Skip Instructions

In page escapes and multiple word instructions, skip-type instructions must be distinguished from non-skipping instructions. For this reason both ISZ and INC are included in the permanent symbol table. ISZ is considered to be a skip instruction and INC is not. INC should be used to conserve space when the programmer desires to increment a memory word without the possibility of a skip.

The first example below shows the code which is assembled for an indirect reference to an off-page location following an INC instruction. The second example shows the same code following an ISZ instruction.

Example 1:

```
INC POINTR  0220  2376
TAD I LOC2  0221  4062
            0222  0520 01  /OFF PAGE INDIRECT EXECUTION
            0223  1407
```

Example 2:

```
ISZ COUNTR  0220  2376
TAD I LOC2  0221  7410      /SKIP TO EXECUTION
            0222  5226      /JUMP OVER EXECUTION
            0223  4062
            0224  0520 01  /OFF PAGE INDIRECT EXECUTION
            0225  1407
```

A special pseudo-operator, SKPDF, must be used to define skip instructions used in source programs but not included in the permanent symbol table. For example:

```
SKPDF DTSF 6771
```

Program Addresses

Since each assembly is relocatable, the addresses specified by SABR always begin at 0200, and all other addresses are relative to this address. At loading time, the Linking Loader will properly adjust all addresses. For example, if 0200 and 1000 are the relative addresses of A and B, respectively, and if A is loaded at 2000, then B will be loaded at 2000 + (1000-0200) or 2600.

All programs to be assembled by SABR must be arranged to fit into one field of memory, not counting page 0 of the field, or the top page (7600 – 7777). If a program is too large to fit into one field, it should be split into several subprograms.

Explicit CDF or CIF instructions are not needed by SABR programs because of the availability of external subroutine calling and common storage. Explicit CDF or CIF instructions cannot be assembled properly.

The Symbol Table

Entries in the symbol table are variable in length. A one or two-character symbol requires three symbol table words. A three- or four-character symbol requires four words, and a five- or six-character symbol, five words. Thus, for long programs it may be to the user's advantage to use short symbols whenever possible.

The symbol table, not counting permanent symbols, contains 2644_{10} words of storage. However, this space must be shared when there are unresolved forward and external references temporarily stored as two-word entries.

If we may assume that a program being assembled never has more than 100_{10} of these unresolved references at any one time, this leaves 2464_{10} words of storage for symbols. Using an average of four words per symbol, this allows room for 616_{10} symbols.

The OS/8 version of SABR has a smaller space for symbol tables, leaving 1364_{10} words of storage, or 1620_{10} if used as the second pass of FORTRAN II.

Symbol table overflow is a fatal condition which generates the error message S.

SYMBOL TABLE FLAGS

Symbols are listed in alphabetic order at the end of the assembly pass 1 with their relative addresses beside them. The following flags are added to denote special types of symbols:

ABS	The address referenced by this symbol is absolute.
COM	The address is in common.
OP	The symbol is an operator.
EXT	The symbol is an external one and may or may not be defined within this program. If not defined, there is no difficulty; it is defined in another program.
UNDF	The symbol is not an external symbol and has not been defined in the program. This is a programmer error. No earlier diagnostic can be given because it is not known that the symbol is undefined until the end of pass 1. A location is reserved for the undefined symbol, but nothing is placed in it.

THE SUBPROGRAM LIBRARY

The Library is a set of subprograms which may be CALLed by any FORTRAN/SABR program. These subprograms are automatically loaded with the OS/8 FORTRAN/SABR system; in the paper tape system they are provided on two relocatable binary paper tapes with part 1 containing those subprograms used by almost every FORTRAN/SABR program. This allows the user to load only those routines which his program makes use of, thus conserving symbol space.

Many of the subprograms reference the Floating-Point Accumulator located at ACH, ACM, ACL (20,21,22 of field 1). The OS/8 Subprogram Library is summarized in the FORTRAN II chapter. The organization of the library programs, as they are provided in the paper tape system, is described in the following pages.

Part 1.	"IOH"	contains	IOH, READ, WRITE
	"FLOAT"	contains	FAD, FSB, FMP, FDV, STO, FLOT, FLOAT, FIX, IFIX, IFAD, ISTO, CHS, CLEAR
	"INTEGER"	contains	IREM, ABS, IABS, DIV, MPY, IRDSW
	"UTILITY"	contains	TTYIN, TTYOUT, HSIN, HSOUT, OPEN, CKIO
	"ERROR"	contains	SETERR, CLRERR, ERROR

Part 2.	"SUBSC"	contains	SUBSC
	"POWERS"	contains	IIPOW, IFPOW, FIPOW, FFPOW, EXP, ALOG
	"SQRT"	contains	SQRT
	"TRIG"	contains	SIN, COS, TAN
	"ATAN"	contains	ATAN

Input/Output

READ is called to initialize the I/O handler before reading data. WRITE is called to initialize the I/O handler before writing data. IOH is called for each item to be read or written. IOH must also be called with a zero argument to terminate an input-output sequence.

All of the programs require that the Floating-Point Accumulator be set to zero before they are called.

```
CALL    2, READ
ARG     (n              /n=DEVICE NUMBER
ARG     fa              /fa=ADDR OF FORMAT
•••
CALL    1, IOH
ARG     data 1          /data 1=ADDR OF HIGH
                        /ORDER WORD OF
                        /FLOATING POINT
                        /NUMBER
CALL    1, IOH
ARG     data 2
•••
•••
CALL    1, IOH          /TERMINATES READ
ARG     0
•••
CALL    2, WRITE        /INITIALIZES WRITE
ARG     (n
ARG     fa
```

The following device numbers are currently implemented:

1 (Teletype keyboard/printer)
2 (High-speed reader/punch)
3 (Card reader/line printer)
4 (Assignable device)

Floating Point Arithmetic

FAD is called to add the argument to the Floating-Point Accumulator.

```
CALL    1, FAD
ARG     addres
```

FSB is called to subtract the argument from the Floating-Point Accumulator.

```
CALL    1, FSB
ARG     addres
```

FMP is called to multiply the Floating-Point Accumulator by the argument.

```
CALL    1, FMP
ARG     addres
```

FDV is called to divide the Floating-Point Accumulator by the argument.

```
CALL    1, FDV
ARG     addres
```

CHS is called to change the sign of the Floating-Point Accumulator.

```
CALL    0, CHS
```

All of the above programs leave the result in the Floating-Point Accumulator. The address of the high-order word of the floating-point number is "addres".

STO is called to store the contents of the Floating-Point

Accumulator in the argument address. The floating-point accumulator is cleared.

```
CALL    1, STO
ARG     storag      /storag=ADDRESS WHERE
                    /RESULT IS TO BE PUT
```

IFAD is called to execute an indirect floating-point add to the Floating-Point Accumulator.

```
CALL    1, IFAD
ARG     ptr         /ptr=2 WORD POINTER
                    /TO HIGH ORDER
                    /ADDRESS OF FLOATING
                    /POINT ARGUMENT
```

ISTO is called to execute an indirect floating-point store.

```
CALL    1, ISTO
ARG     ptr
```

CLEAR is called to clear the Floating-Point Accumulator. The AC is unchanged.

```
CALL    0, CLEAR
```

FLOAT and FLOT are called to convert the integer contained in the AC (processor accumulator) to a floating-point number and store it in the Floating-Point Accumulator.

```
CALL    0, FLOT     or      CALL    1, FLOAT
                            ARG     addr
```

IFIX and FIX are called to convert the number in the Floating-Point Accumulator to a 12-bit signed integer and leave the result in the AC.

```
CALL    0, FIX      or      CALL    1, IFIX
                            ARG     addr
```

ABS leaves the absolute value of the floating-point number at "addr" in the Floating-Point Accumulator.

```
CALL    1, ABS
ARG     addr
```

Integer Arithmetic

MPY is called to multiply the integer contained in the AC by the integer contained in "addr." The result is left in the AC.

```
CALL    1, MPY
ARG     addr
```

DIV is called to divide the integer contained in the AC by the integer contained in "addr." The result is left in the AC.

```
CALL    1, DIV
ARG     addr
```

IREM leaves the remainder from the last executed integer divide in the AC.

```
CALL    1, IREM
ARG     0
```

(The argument is ignored.)

IABS leaves the absolute value of the integer contained in "addr" in the AC.

```
CALL    1, IABS
ARG     addr
```

IRDSW reads the value set in the console switch register into the AC.

```
CALL    0, IRDSW
```

Subscripting

SUBSC is called to compute the address of a subscripted variable, and can be used for doubly or singly subscripted arrays. On entry, the AC should be negative for floating-point variables—any negative number for singly subscripted variables, and 1's complement of the first dimension for doubly subscripted variables. For doubly subscripted integer variables, the AC must be the first dimension.

The general calling sequence for SUBSC is as follows:

```
*TAD (M          /1ST DIMENSION (USED ONLY
                 /IF 2 DIMENSIONS)
*CMA             /USED ONLY IF ARRAY IS
                 /FLOATING POINT
      ┌2,SUBSC┐  /SINGLE SUBSCRIPT
 CALL │       │
      └3,SUBSC┘  /DOUBLE SUBSCRIPT
*ARG  J          /2ND DIMENSION
 ARG  I          /1ST DIMENSION
 ARG  BASE       /BASE ADDRESS OF ARRAY
 LOCA            /ADDRESS OF TWO WORD DUMMY
                 /ADDRESS LOCATION
```

* Optional Statements.

For example, to load the I,Jth element of a floating-point array whose dimensions are 5 by 7:

```
TAD (5
CMA                /DIMENSIONS ARE 5 BY 7
CALL 3,SUBSC
ARG J              /ADDRESS OF 2ND SUBSCRIPT
ARG I              /ADDRESS OF 1ST SUBSCRIPT
ARG ARRAY          /BASE ADDRESS OF ARRAY
LOC                /MUST BE A DUMMY VARIABLE
CALL 1,IFAD
ARG LOC
```

Functions

SQRT leaves the square root of the floating-point number at "addr" in the Floating-Point Accumulator.

```
CALL    1, SQRT
ARG     addr
```

SIN, COS, TAN leave the specified function of the floating-point argument at "addr" in the Floating-Point Accumulator.

```
CALL    1, SIN
ARG     addr
```

ATAN leaves the arctangent of the floating-point number at "addr" in the Floating-Point Accumulator.

```
CALL    1, ATAN
ARG     addr
```

ALOG leaves the natural logarithm of the floating-point number at "addr" in the Floating-Point Accumulator.

```
CALL    1, ALOG
ARG     addr
```

EXP raises "e" to the power specified by the floating-point number at "addr" and leaves the result in the floating-point accumulator.

```
CALL    1, EXP
ARG     addr
```

All of these subprograms require that the floating-point accumulator be set to zero before they are called.

The POWER routines (IIPOW, IFPOW, FIPOW, FFPOW) are called by FORTRAN to implement exponentiation. The first operand is in the AC (floating-point or processor depending on mode), and the address of the second is an argument. The address of the result is in the appropriate AC upon return.

FUNCTION NAME	MODE OF OPERAND 1 (BASE)	MODE OF OPERAND 2 (EXPONENT)	MODE OF RESULT
IIPOW	INTEGER	INTEGER	INTEGER
IFPOW	INTEGER	FLOATING POINT	FLOATING POINT
FIPOW	FLOATING POINT	INTEGER	FLOATING POINT
FFPOW	FLOATING POINT	FLOATING POINT	FLOATING POINT

```
CALL    2, FFPOW
ARG     addr 2          /ADDRESS OF OPERAND 2
```

Utility Routines

OPEN is called at the beginning of every FORTRAN program to start the high-speed reader/punch and teleprinter, and to initialize the I/O routines for device code 4 if using the OS/8 FORTRAN/SABR system. The form is:

```
CALL Ø,OPEN
```

When an error is encountered in a program, the ERROR routine is called. The program passes to the ERROR routine the address of the error message to be printed. The format of the error message is 4 characters in stripped ASCII and packed into 2 words:

```
                              ENTRY ABC
2343   0102        XYZ,       0102;0304
2344   0304
2345   0000        ABC,       BLOCK 2
2346   0000        .
                   .
                   .
                              CALL I,ERROR
                              ARG XYZ
```

When control passes to the ERROR routine, the parameters passed are picked up. In the case above, the parameters are as follows:

```
62N1                    ARG XYZ
2343
```

where N is the field that XYZ is in, and 2343 is the address of XYZ. The ERROR routine then prints the message at location 2343 plus a 5-digit address which is 2 greater than 2343.

```
ABCD ERROR AT N2345
```

Since XYZ is 2 locations before ABC, the address printed will be the address of ABC.

The error message is usually placed just before the entry point of the routine in which the error was detected—thus the address printed by ERROR will be the address of the entry point. This provides a convenience to the programmer since the entry point will appear in the Loader Map.

CKIO is a subroutine which waits for the TTY flag to be set. It is called by the OS/8 EXIT subroutine to eliminate the possibility of a garbled TTY output. It may be used in FORTRAN for possible expansion with interrupts, and is of the form:

```
CALL Ø,CKIO
```

The following subroutines—IOPEN, OOPEN, OCLOSE, CHAIN, EXIT, and GENIO—are used by the OS/8 FORTRAN/SABR Operating System for device independent I/O and chaining.

DECtape I/O Routines

RTAPE and WTAPE (read and write tape) are the DECtape read and write subprograms for the 8K FORTRAN and 8K SABR systems. The subprograms are furnished on one relocatable binary-coded paper tape which must be loaded into field 0 by the 8K Linking Loader, where they occupy one page of core.

RTAPE and WTAPE allow the user to read and write any amount of core-image data onto DECtape in absolute, non-file-structured data blocks. Many such data blocks may be stored on a single tape, and a block may be from 1 to 4096 words in length.

RTAPE and WTAPE are subprograms which may be called with standard, explicit CALL statements in any 8K FORTRAN or SABR program. Each subprogram requires four arguments separated by commas. The arguments are the same for both subprograms and are formatted in the same manner. They specify the following:

1. DECtape unit number (from 0 to 7)
2. Number of the DECtape block at which transfer is to start. The user may direct the DECtape service routine to begin searching for the specified block in the forward direction rather than the usual backward direction by making this argument the two's complement of the block number.
3. Number of words to be transferred ($1 \leqslant N \leqslant 4096$)
4. Core address at which the transfer is to start.

DECtape I/O Routines for the FORTRAN II system are explained in Chapter 7. In 8K SABR, the CALL statements to RTAPE and WTAPE are written in the following format (arguments may be either octal or decimal numbers):

```
CALL 4,WTAPE      /WOULD BE SAME FOR RTAPE
ARG (6            /DATA UNIT NUMBER
ARG (200          /STARTING BLOCK NUMBER
                  /IN OCTAL
ARG (604          /WORDS TO BE TRANSFERRED
                  /IN OCTAL
ARG LOCB          /CORE ADDRESS, START OF
                  /TRANSFER
```

In these examples, LOCA and LOCB may or may not be in common.

As a typical example of the use of RTAPE and WTAPE, assume that the user wants to store the four arrays A, B, C, and D on a tape with word lengths of 2000, 400, 400, and 20 respectively. Since PDP-8 DECtape is formatted with 1474 blocks (numbered 0-2701 octal) of 129 words each (for a total of 190,146 words), A, B, C, and D will require 16, 4, 4, and 1 blocks respectively. (The block numbers used by RTAPE and WTAPE should not be confused with the record numbers used by OS/8. A OS/8 record is 256 words—roughly twice the size of a DECtape block.)

Each array must be stored beginning at the start of some DECtape block. The user may write these arrays on tape as follows:

```
CALL WTAPE (0,1,2000,A)
CALL WTAPE (0,17,400,B)
CALL WTAPE (0,21,400,C)
CALL WTAPE (0,25,20,D)
```

The user may also read or write a large array in sections by specifying only one DECtape block (129 words) at a time. For example, B could be read back into core as follows:

```
CALL RTAPE (0,17,258,B(1))
CALL RTAPE (0,19,129,B(259))
CALL RTAPE (0,20,13,B(388))
```

As shown above, it is possible to read or write less than 129 words by starting at the beginning of a DECtape block. It is impossible, however, to read or write starting in the middle of a block. For example, the last 10 words of a DECtape block may not be read without reading the first 119 words as well.

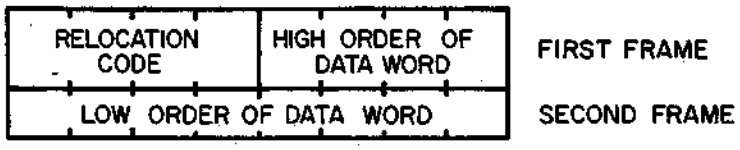
A DECtape read or write is normally initiated with a backward

search for the desired block number. To save searching time, the user may request RTAPE or WTAPE to start the block number search in the forward direction. This is done by specifying the negative of the block number. This should be used only if the number of the next block to be referenced is at least ten block numbers greater than the last block number used. For example, if the user has just read array A and now wants array D, he may write:

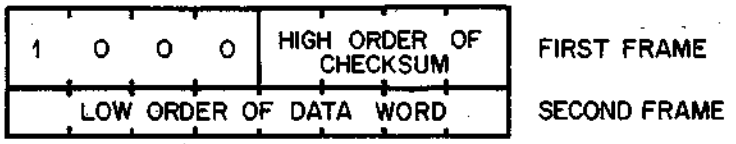
```
CALL RTAPE (0,1,2000,A)
CALL RTAPE (0,-27,20,D)
```

THE BINARY OUTPUT TAPE

SABR outputs each machine instruction on binary output tape as a 16-bit word contained in two 8-bit frames of paper tape. The first four bits contain the relocation code used by the Linking Loader to determine how to load the data word. The last 12 bits contain the data word itself.



The assembled binary tape is preceded and followed by leader/trailer code (code 200). The checksum is contained in the last two frames of tape before the trailer code. It appears as a normal 16-bit word, as shown below.



All assembled programs have a relative origin of 0200.

Loader Relocation Codes

The four-bit relocation codes issued by SABR for use by the Linking Loader are explained below. The codes are given in octal.

00	Absolute	Load the data word at the current loading address. No change is required.

```
0205    5277          JMP LOC  /WHERE LOC IS
                               /AT 0077 (OF
                               /CURRENT PAGE)
```

01	Simple Relocation	Add the relocation constant to the word before loading it. (The relocation constant is 200 less than the actual address where the first word of the program is loaded.) Items with this code are always program addresses.

```
0376  0520 01   A,      LOC2
```

In the above example, LOC2 is at relative address 0520. If the first word of the program (relative address 0200) is loaded at 1000, then the actual address of A is 1176 and location 1176 will be loaded with the value 1320, which will be the actual address of LOC2 when loaded.

03	External Symbol Definition*	The data word is the relative address of an entry point. Before entering this definition in the Linkage Tables so that the symbol may be referenced by other programs at run-time, the Linking Loader must add the relocation constant to it. The six frames of paper tape following the two-frame definition are the stripped ASCII code for the symbol.

* Does not appear in assembly listings.

03	ADDRESS
ADDRESS LOW ORDER	
L	
O	
C	
2	
SPACE	
SPACE	

04 Re-origin*

Change the current loading address to the value specified by the data word plus the relocation constant.

05 CDF Current

The data word is always a 6201 (CDF) instruction which has been generated automatically by SABR. The code 05 indicates to the Linking Loader that the number of the field currently being loaded into must be inserted in bits 6-8 before loading.

```
0300   6201 05    A,      TAD LOC2
0301   1776                       /WHERE LOC2 IS
                           /OFF PAGE SO
                           /THAT THE TAD
                           /INSTR. MUST BE
                           /INDIRECT
0376   0520 01
```

If the program containing this code is being loaded into field 4, relative location 0300 will be loaded with 6241.

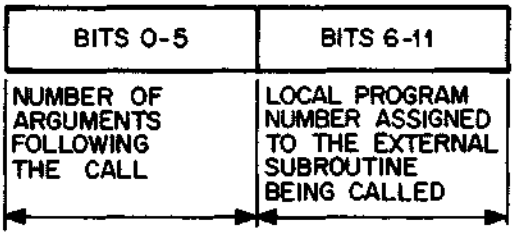
Such an instruction is referred to in this document as CDF Current. It is generated automatically by

* Does not appear in assembly listings.

SABR when a direct reference instruction must be assembled as an indirect, and there is the possibility that the current data field setting is different from the field where the indirect reference occurs.

06 Subroutine Linkage Code

The data word is a special constant enabling the Linking Loader to perform the necessary linking for an external subroutine call. (c.f., CALL Pseudo-op). The structure of the data word is shown below.



Before the 12-bit, two-part code word is loaded into memory, a global external number will be substituted for the local external symbol number in the right half of the data word.

```
0200   4033       CALL 3,SUB
0201   0307 06
                  ARG X
                  ARG Y
                  ARG Z
```

Here, SUB has been assigned the local number 06 during assembly. At loading time this number will be changed to the global number (for example, 23) which is assigned to SUB. In this example, 0323 would actually be loaded at relative address 0201.

10	Leader/Trailer* and Checksum	This code represents normal leader/trailer. The checksum is contained in the last two frames of paper tape preceding the trailer code.
12	High Common*	The data word is the highest location in Field 1 assigned to common storage by the program. This item will occur exactly once in every binary tape and it must be the first word after the leader. If no common storage has been allocated in the program, the data word will be 0177.
17	Transfer* Vector	Signifies that reference to an external symbol occurs in the assembled program. The 12-bit data word is meaningless. The next six frames contain the ASCII code for the symbol. The Linking Loader uses this definition to create a transfer table, whereby local external symbol numbers assigned during assembly of this particular program can be changed to the global external symbol number when several programs are being loaded.

SAMPLE ASSEMBLY LISTINGS

The following examples are offered to illustrate many of the features and formats of the SABR Assembler.

When a multiple word instruction occurs, the actual instruction line is typed beside the first instruction.

---

* Does not appear in assembly listings.

```
0650   6201 05    LOC2,    JMP NAME  /OFF PAGE
0651   5774
0652   7106               CLL RTL;RTL;RTL
0653   7006
0654   7006
```

When there is an erroneous instruction, the error flag appears in the address field. The instruction is not assembled.

```
0700   7200       N2,      CLA
    I                      CLL SKP
0701   7402                HLT
```

The page escape and literal and off-page pointer table are typed with nothing except the correct address, value and loader code.

```
0770   7006       N3,      RTL
0771   7500                SMA
0772   5376
0773   5377
0774   0200 01
0775   0020
0776   7410                          /SKP TO 1ST LOC.-
                                     /NEXT PAGE (AC IS
                                     /NOT MINUS)
0777   7410                          /SKP TO 2ND LOC.-
                                     /NEXT PAGE (AC IS
                                     /MINUS)
                                                    /
```

Locations 0772, 0773, 0776 and 0777 make up the page escape since the last instruction is a skip instruction (SMA). Refer to the section concerning Page Escapes.

The following program has been assembled and listed. It cannot be run without first debugging and editing it.

During the first pass, SABR outputs the binary tape and prints error messages as they occur. In this case, none of the errors are fatal, and assembly continues. The symbol table is printed, and undefined symbols, external symbols, or any other special types of symbols which cannot be determined until the end of the pass are flagged in the symbol table.

The optional second pass of the Assembler produces a listing. The 4-digit first column contains the octal address, while the

second column contains the octal code for each line of instructions. Errors are also printed during the listing pass at the line in which they occur. Meanings of error codes are described later in the chapter.

The reader is also referred to Demonstration Program Using Library Routines.

```
C AT  PUNCH +0003

COUNT   0302
DECIMA  0000UNDF
LT      0264
MAIN    0000EXT
MESG    0243
ORG     0303
PTAPE   0201EXT
PUNCH   0274
REF     0177ABS
RPT     0267
START   0205
TYPE    0000EXT

                    /PROGRAM TO PUNCH RIM FORMAT PAPER TAPES

        6026                OPDEF PLS 6026  /DEFINE HI SPEED
        6021                SKPDF PSF 6021  /IOTS
        0177                ABSYM REF 177
                            ENTRY MAIN
0200    0000                DECIMAL
                            LAP
0201    0000        PTAPE,  BLOCK 2         /PUNCH LEADER
0202    0000                                /TAPE (200 CODE)
0203    1377                TAD (-32        /32 LOCATIONS
0204    3302                DCA COUNT
                            OCTAL
0205    1303        START,  TAD ORG
0206    7132                CLL CML RTR;RTR;RTR
0207    7012
0210    7012
```

```
0211  0376              AND (177
0212  4274              JMS PUNCH         /PUNCH LEADING
0213  1303              TAD ORG           /DIGITS OF ADDRESS
0214  0376              AND (177          /PUNCH SECOND
0215  4274              JMS PUNCH         /DIGITS OF ADDRESS
0216  1703              TAD I ORG         /NOW PUNCH CONTENTS
0217  7112              CLL RTR;RTR;RTR   /OF THAT LOCATION
0220  7012
0221  7012
0222  0375              AND (77
0223  4274              JMS PUNCH
0224  1703              TAD I ORG         /GET SECOND DIGITS
0225  0375              AND (77           /OF THAT LOCATION
0226  4274              JMS PUNCH
0227  2303              INC ORG           /POINT TO NEXT
                                          /CORE LOCATION
0230  2302              ISZ COUNT         /DONE YET?
0231  5205              JMP START         /NO
0232  4033              CALL 1,TYPE       /YES, TYPE MESSAGE
0233  0102 06
0234  6201 05           ARG MESG
0235  0243 01
0236  4264              JMS LT            /ENDING 200 CODE
0237  7404              OSR               /GET NEW ADDRESS
0240  3303              DCA ORG           /FROM SWITCH REGISTER
                                          /PUT IT IN ORG
0241  7402              HLT               /PAUSE
0242  5774              JMP MAIN          /PUNCH NEW TAPE

0243  2401      MESG,   TEXT "TAPE PUNCHED. ENTER ORIGIN & CONT."
0244  2005
0245  4020
0246  2516
0247  0310
0250  0504
0251  5640
0252  0516
0253  2405
0254  2240
0255  1722
0256  1107
0257  1116
0260  4046
0261  4003
0262  1716
0263  2456

0264  0000      LT,     0
                        OCTAL
0265  1373              TAD (-40
0266  3302              DCA COUNT         /32 FRAMES OF
0267  1372      RPT,    TAD (200          /LEADER/TRAILER
0270  4274              JMS PUNCH         /PUNCH IT
0271  2302              ISZ COUNT         /DONE?
0272  5267              JMP RPT           /NO
0273  5664              JMP I LT          /RETURN
```

```
0274  0000       PUNCH,  0
0275  6026               PLS             /PUNCH
0276  6021               PSF             /WAIT FOR FLAG
    C                    JMP .-1
0277  4045               JMP I PUNCH     /EXIT
0300  7410
0301  5674

0302  0000       COUNT,  0
0303  7300       ORG,    7300
0304  4040               RETRN PTAPE
0305  0003 06
0372  0200
0373  7740
0375  0077
0376  0177
0377  7746
                         END
```

SABR PROGRAMMING NOTES

Optimizing SABR Code

There are generally two types of programmers who will use the SABR Assembler—those who like the convenience of a page-boundary-independent code and need not be concerned with program size, and those who need a relocatable assembler, but are still very location conscious. These optimizing hints are directed to the latter user.

One way to circumvent the cost of non-paged code is to make use of the LAP (Leave Automatic Paging) pseudo-op and the PAGE pseudo-op to force paging where needed. This saves 2 to 4 instructions per page by elimination of the page escape. In addition, the fact that the program must be properly segmented may save a considerable amount.

Extra core may be reduced by eliminating the CDF instructions which SABR inserts into a program. This is done by using "fake indirects". Define the following op codes:

```
OPDEF ANDI 0400
OPDEF TADI 1400
OPDEF ISZI 2400
OPDEF DCAI 3400
```

These codes correspond to the PDP-8 memory reference instructions but they include an indirect bit. The difference can best be illustrated by an example:

If X is off-page, the sequence:

```
LABEL,  SZA
        DCA X
```

is assembled by SABR into:

```
LABEL,  SZA
        JMS 45
        SKP
        DCA I (X)
```

or four instructions and one literal.

The sequence:

```
PX,     X
        .
        .
        .
LABEL,  SZA
        DCAI PX
```

assembles into three instructions for a saving of 40 percent. Note, however, that the user *must* be sure that the data field will be correct when the code at LABEL is encountered. Also note that SABR assumes that the Data Field is equal to the Instruction Field after a JMS instruction, so subroutine returns should not use the JMP I op code.

The standard method to fetch a scalar integer argument of a subroutine in SABR is:

```
                              DUMMY X
0200  0000        IARG,       0
0201  0000        X,          BLOCK 2
0202  0000
0203  0000        SUBR,       BLOCK 2
0204  0000
0205  4067                    TAD I SUBR
0206  0203 01
0207  1407
0210  3201                    DCA X
0211  2204                    INC SUBR#
0212  4067                    TAD I SUBR
0213  0203 01
0214  1407
0215  3202                    DCA X#
0216  2204                    INC SUBR#
0217  4067                    TAD I X
0220  0201 01
0221  1407
0222  3200                    DCA IARG
                              .
                              .
                              .
```

This is the method the FORTRAN compiler uses, and although it is standard, it is also the slowest. This code requires 19 words of core and takes several hundred microseconds to execute.

The fastest way to pick up arguments within a SABR coded external subroutine is as follows (this takes approximately one fifth of the time of the previous method and four less locations):

```
0200  0000        IARG,       0
0201  0000        SUBR,       BLOCK 2
0202  0000
0203  1201                    TAD SUBR
0204  3205                    DCA X1
0205  7402        X1,         HLT              /REPLACED
                                               /BY CDF
0206  1602                    TADI SUBR#
0207  3214                    DCA X2
0210  2202                    INC SUBR#
0211  1602                    TADI SUBR#
0212  3200                    DCA IARG
0213  2202                    INC SUBR#
0214  7402        X2,         HLT              /REPLACED
                                               /BY CDF
0215  1600                    TADI IARG
0216  3200                    DCA IARG
                              .
                              .
```

To pick up multiple arguments, the locations from X1 to X2+1 inclusive can be made into a subroutine.

Calling the OS/8 USR and Device Handlers

One important point to remember is that any code which calls the USR must not reside in locations 10000 to 11777. Therefore, any SABR routine which calls the USR must be loaded into a field other than field 1 or above location 2000 in field 1. To call the USR from SABR use the sequence:

```
CPAGE N          /N=7+(# OF ARGUMENTS)
6212             /CIF 10
JMS 7700         /OR 200 IF USR IN CORE
REQUEST
ARGUMENTS        /OPTIONAL DEPENDING ON REQUEST
ERROR RETURN     /OPTIONAL DEPENDING ON REQUEST
```

To call a device handler from SABR use the sequence:

```
         CPAGE 12        /10 IF "HAND" IN PAGE 0
         6202            /CIF 0
         JMS I HAND      /DO NOT USE JMSI
         FUNCT
         ADDR
         BLOCK
         ERROR RETURN
         SKP
HAND,    0               /"HAND" MUST BE ON SAME PAGE
                         /AS CALL, OR IN PAGE 0
```

SABR ERRORS

In case of error, SABR prints the following codes in the address field of the instruction line:

Table 4-3 SABR Error Codes

Error Code	Meaning
A	Too many or too few ARG statements follow a call statement.
C	An illegal character appears on the line.
D	A device handler has returned a fatal condition.
L	/L or /G option was indicated, but the LOADER.SV file does not exist on the system device.
M	A symbol is multiply defined. Listing of programs with multiple definitions have unmarked errors.
I	An illegal syntax has been used, (as one of the following): 1. a pseudo-op with improper arguments, 2. a quote mark with no argument, 3. a non-terminated text string, 4. an improper address, 5. an illegal combination of micro-instructions.
E	There is no END statement.
S	Either the symbol table has overflowed, common storage has been exhausted, more than 64 different user-defined symbols occurred in a core page, or more than 64 external symbols have been declared. Could also indicate a system error such as overflowed output file.
U	No symbol table is being produced, but there is at least one undefined symbol in the program.
UNDF	Undefined symbol, printed in the symbol table listing.

LINKING LOADER

The Linking Loader is the system program used to load and link a user's program and subprograms in any field(s) of memory. It can be called automatically to load or load and start a FORTRAN or SABR program, or independently to load or load and start a relocatable binary file stored on a device. It is capable of loading programs over itself, and has options which allow the user to obtain storage map listings of core availability.

The Linking Loader has the capability of searching program libraries for subroutines which are referenced by the program in core and to load those subroutines needed. (A library is a collection of relocatable subroutines—FORTRAN or SABR output—with a directory at the beginning to facilitate searching.) Any library can be searched by using the /L option to the Loader, but the system library, LIB8.RL, is searched automatically just before the Loader completes the building of a core image of the user's program. If LIB8.RL is not on the system device, there is no automatic library search. (The system program LIBSET is available to allow the user to build his own subroutine library.)

The Linking Loader is capable of loading any number of user and library programs into any field of memory. Several programs are usually loaded into each field. Because of the space reserved for the Linkage Routines, the available space in field 0 is three pages smaller than in all other fields.

Any common storage reserved by the programs being loaded is allocated in field 1 from location 200 upwards. The space reserved for common storage is subtracted from the available loading area in field 1. The program reserving the largest amount of common storage must be loaded first.

The Run-Time Linkage Routines necessary to execute SABR programs are automatically loaded into the required areas of every field by the Linking Loader as part of its initialization. The user needs to know nothing more about these routines than the particular areas of core they occupy.

Calling and Using the Linking Loader

The user can automatically call the Linking Loader following assembly of either a SABR program or a SABR-assembled FORTRAN program by use of the /L or /G options. For details on

automatic calling of the Linking Loader, see the FORTRAN section of this chapter.

When the user wishes to call the Linking Loader specifically to load or load and start a relocatable binary file, he issues the command:

```
R LOADER
```

in response to the dot printed by the Keyboard Monitor. The Command Decoder replies by printing an asterisk at the left margin; the user then indicates input and output files and any desired options. 0 to 1 output files and 1 to 9 input files are possible. Only one binary program per file is permitted. The assumed extension for input files is .RL. The output file, if indicated, is used to hold a map of the loaded program.

The user has the ability to either specify all options and operations to be performed on one line or to have various operations performed individually. Where all options are being specified at one time the line to the Command Decoder contains the complete instructions for the Linking Loader. If operations are to be done individually, the user can type a command, enter it with the RETURN key, and that command will be executed, with another command expected when the first is completed. To indicate the last command, the user types an ALT MODE character, or ends the last command with a /G option to start the program.

LINKING LOADER OPTIONS

The options to the Linking Loader are as shown in Table 4-4.

Table 4-4 Linking Loader Options

Option	Meaning
/I	A program doing device-independent input is to be loaded. (This feature costs the user 3 pages of core.)
/O	A program doing device-independent output is to be loaded. (This feature costs the user 3 pages of core.)

Table 4-4 Linking Loader Options (Cont.)

Option	Meaning

	If both /I and /O are indicated, 6 pages of core are used to handle device-independent I/O.
	
	/I and /O, if used, must be given before or on the first input line specifying files to be loaded. For example:
	
	`*INPUT,FILES/O$`
	
	is acceptable, but
	
	`*INPUT`
	`*/O FILES`
	
	is not legal and will generate an error message.

/H	A program doing device-independent I/O requires two-page device handlers at run-time. (This feature costs the user one additional page if he is doing just input or output, and two additional pages if he is doing input and output. If /I, /O, and /H are indicated, 8 pages of core are used to handle device-independent I/O. /H, if used, must be indicated on or before the first line containing /I or /O, and is meaningless without /I or /O also being specified.
/G	Start the program after processing the rest of the command string. Execution starts at the symbol MAIN unless otherwise indicated.
=n	Specifies the starting address of the program if other than the entry point MAIN; n is an octal number up to 5 digits long.
/M	Output a map of the loaded programs onto the output file specified, followed by a count of the free pages in each field. If no output is specified, the map is put onto the teleprinter. The assumed extension for map output file is .MP. The map is printed after the rest of the command line is processed.

Table 4-4 Linking Loader Options (Cont.)

Option	Meaning
/U	Similar to /M, but only outputs undefined symbols.
/P	Similar to /M, but only outputs counts of free pages in each field.
/n	Search through the available fields starting at field n for space large enough to hold each input file; n is an integer in the range 0 to 7, inclusive. Only one binary program can be in each input file. If n is not specified, the Loader starts looking at field 0.
/R	Restart loading process (forget all previously loaded programs). This command is equivalent to restarting the Linking Loader, but is much faster for DECtape systems since no tape motion is involved.
/L	Load the first input file as a library file (Loader expects a Library Directory as the first block of the file). All other input files on the line are ignored.

The Core Availability option (/P) causes the number of free pages of memory in every field of memory to be printed in a list on the teleprinter. For example, if the user has a 16K configuration, a list like the following might be printed:

```
0002    (number of free pages in field 0)
0010    (number of free pages in field 1)
0030    (number of free pages in field 2)
0036    (number of free pages in field 3)
```

The number of pages initially available in field 0 is 0033 and in all other fields is 0036.

The Storage Map option (/M), when selected, causes a list of all program entry points to be printed along with the actual address at which they have been loaded. Entry points of programs which have been called but which have not been loaded are also

listed along with U flag for "undefined". Such flagged programs must be loaded before execution of the user's programs are possible. The core availability list is automatically appended to the storage map. A sample is shown below for an 8K machine:

```
MAIN     10200
READ     01050
WRITE    01066
IOH      03031
ERROR    00000 U
GENIO    00000 U
FDV      04722
CLEAR    05247
IFAD     05131
FMP      04632
ISTO     05074
STO      04447
FLOT     05210
FAD      04010
DIV      00000 U
IREM     00000 U
FSB      04000
FLOAT    05046
FIX      04513
IFIX     04561
CHS      05231
0011
0033
```

EXAMPLES OF I/O COMMAND STRINGS

The following are examples of possible input command strings:

```
*PROG,DTA2:SUB1,SUB2/G
```

This string loads DSK:PROG.RL, DTA2:SUB1.RL, DTA2:SUB2. RL, loads any necessary library routines requested, and starts the program at the entry point MAIN. The same process could have been done as follows:

Load DSK:PROG.RL ;

Get a list of undefined symbols on the teleprinter;

```
*PROG

*/U
.       (Symbols go here)
.
.
.

*DTA2:SUBR1,SUBR2
```

Load DTA2:SUBR1.RL,SUB2.RL ;

```
*LPT:/M<$
```

Put loading map on the line printer, load the binary of any library routines requested by the program, and exit ($ is printed by the ALT MODE key);

```
.SAVE DTA2 FORTPG
```

Save the core image on DTA2 as FORTPG.SV;

Start the core image at its starting address (entry point MAIN in this case).

```
.START
```

Linking Loader Error Messages

The Linking Loader outputs error messages in the form

ERROR nnnn

where nnnn represents a 4-digit error code. Table 4-5 lists the meanings of these error codes.

Table 4-5 Linking Loader Error Messages

Error Code	Meaning
0000	/I or /O specified too late.
0001	Symbol table overflow; more than 64 subprogram names.
0002	Program will not fit into core.
0003	Program with largest common storage area was not loaded first.
0004	Checksum error in input tape.
0005	Illegal relocation code.
0006	An output error has occurred.
0007	An input error has occurred (either a physical device error, or an attempt was made to read from a write-only device such as LPT:).
0010	No starting address has been specified and there is no entry point named MAIN.
0011	An error occurred while the Loader attempted to load a device handler.
0012	I/O error on system device.

LIBRARY SETUP (LIBSET)

LIBSET, the FORTRAN Library Setup program, creates a library of subroutines from the relocatable binary output of SABR. These library files can be quickly and effectively scanned by the Linking Loader, thus saving a great deal of the time involved in loading frequently used subroutines. (Refer to the section concerning the Linking Loader for information pertaining to relocatable library files, automatic loading of the LIB8.RL file, and the /L option.)

Calling and Using LIBSET

To call LIBSET from the system device, the user types

```
R LIBSET
```

in response to the dot printed by the Keyboard Monitor. The Command Decoder then prints an asterisk at the left margin of the teleprinter paper and waits to receive a line of input. The general form of input required to build a library file is:

```
*DEV:OUTPUT FILE <DEV:INPUT FILE(S)
*(additional input files) $
```

No more than nine input files are allowed on any one line, but several input lines can be entered. The last input line must end with the user typing the ALT MODE key (which echoes as $). Only the first line can contain an output file. If no output file is specified, a file named LIB8.RL is created on the system device. The assumed extension for both input and output files is .RL.

NOTE

Files output from LIBSET are in a special relocatable library format and must *not* be copied with the /B option in PIP. Instead, they should be copied by PIP in image (/I) mode.

LIBSET OPTIONS

Only one option is allowed in the use of LIBSET, and this is described below:

Option	Meaning
/S	The /S option means that all input files on a line are to be regarded as containing more than one relocatable binary file. (This is analogous to the /S option in ABSLDR.

NOTE

If /S is used on a line that contains no input files, input from PTR: is assumed.

EXAMPLES OF LIBSET USAGE

Example 1:

```
*DTA2:SUBS<DTA1:SUB1,SUB2,SUB3,PTR:
↑*SYS:FUNC1,FUNC2.V5$
```

This example creates a relocatable library file on DTA2 named SUBS.RL. This library will contain six FORTRAN (or SABR) subroutines built by combining the relocatable binary file SUB1.RL, SUB2.RL, and SUB3.RL from DTA1 together with one relocatable binary paper tape (note the ↑ printed by OS/8 before loading from PTR:) and the files FUNC1.RL and FUNC2.V5 from the system device.
Example 2:

```
*ASIN,ACOS
*/S$↑
```

Since no output file was specified, this example creates a relocatable library file LIB8.RL on the system device. This produces a new FORTRAN library including the subroutines contained in the files ASIN and ACOS on device DSK, and several subroutines combined on a single paper tape loaded from the high-speed reader.

Subroutine Names

It is important to distinguish between the OS/8 file name of a relocatable binary program and its assigned Entry Point name. The file name has meaning only to the Command Decoder; the Entry Point name (or names) are the true subroutine names that are meaningful to the Loader.

Further details on the format of relocatable binary files and relocatable library files can be found in the *OS/8 Software Support Manual* (DEC-S8-OSSMA-A-D).

Sequence for Loading Subroutines

LIBSET can combine files in any sequence to form a relocatable library file. However, the subroutines in any single library are loaded by the Loader in the order in which they were originally specified to LIBSET. Therefore, it is important to make sure that subroutines are specified in order of size, with the largest subroutine being loaded first. If this is not done, cases can occur in which insufficient core is available in any single field to load a subroutine, whereas space would have been available if the subroutine had been loaded earlier.

LIBSET Error Messages

All errors are fatal. LIBSET recalls the Keyboard Monitor upon encountering any of the following error conditions, and must be recalled in order to enter another command string.

Table 4-6 LIBSET Error Messages

Error Message	Meaning
BAD FORMAT OR CHECKSUM— TRY AGAIN	Error in reading relocatable binary file.
ERROR WHILE WRITING OUTPUT FILE	Fatal output error occurred.
INPUT ERROR	Parity error on input.
LIBRARY DIRECTORY OVERFLOW	Too many subroutines were specified. Every subroutine name in the input file requires four words, and every relocatable binary file read requires two words. If the total number of words exceeds 250, the library must be split into two separate files.

LIBRARY PROGRAMS

During execution, the Library programs check for certain errors and type out the appropriate error messages in the form:

XXXX ERROR AT LOC NNNN

where XXXX specifies the type of error, and NNNN is the location of the error. When an error is encountered, execution stops, and the error must be corrected.

When multiple error messages are typed, the location of the last error message is relevant to the user program. The other error messages are relevant to subprograms called by the statement at the relevant location.

Table 4-7 Library Error Messages

Error Message	Explanation
ALOG	Attempt to compute log of negative number
ATAN	Result exceeds capacity of computer
DIVZ	Attempt to divide by 0
EXP	Result exceeds capacity of computer
FIPW	Error in raising a number to a power
FMT1	Multiple decimal points
FMT2	E or . in integer
FMT3	Illegal character in I, E, or F field
FMT4	Multiple minus signs
FMT5	Invalid FORMAT statement
FLPW	Negative number raised to floating power
FPNT	Floating-point error; may be caused by division by zero; floating-point overflow; attempt to fix too large a number.
SQRT	Attempt to take root of a negative number

OS/8 includes, in addition, the error message:

```
USER ERROR 1 AT 00537
```

which means that the user tried to reference an entry point of a program which was not loaded.

To pinpoint the location of a Library execution error:

1. From the Storage Map, determine the next lowest numbered location (external symbol) which is the entry point of the program or subprogram containing the error.
2. Subtract in octal the entry point location of the program or subroutine containing the error from the LOC of the error in the error message.
3. From the assembly symbol table, determine the relative address of the external symbol found in step 1 and add that relative address to the result of step 2.
4. The sum of step 3 is the relative address of the error, which can then be compared with the relative addresses of the numbered statements in the program.

DEMONSTRATION PROGRAM USING LIBRARY ROUTINES

The following demonstration program is a SABR program showing the use of the library routines. The program was written to add two integer numbers, convert the result into floating-point, and type the result in both integer and floating-point format. The source program was written using the Symbolic Editor, assembled with SABR, and loaded with the Linking Loader, under the OS/8 Operating System.

```
A        0257
B        0260
C        0261
D        0262
FLOAT    0000EXT
FORMT    0240
IOH      0000EXT
N        0256
OPEN     0000EXT
START    0200EXT
STO      0000EXT
WRITE    0000EXT

                          ENTRY START

0200  4033       START,   CALL 0,OPEN      /INITIALIZE
0201  0002 06
                                            /I/O DEVICES
0202  1257                TAD A            /COMPUTE C=A+B
0203  1260                TAD B
0204  3261                DCA C
0205  4033                CALL 1,FLOAT     /CONVERT TO
0206  0103 06
                                            /FLOATING POINT
0207  6201 05            ARG C
0210  0261 01
0211  4033                CALL 1,STO
0212  0104 06
0213  6201 05            ARG D
0214  0262 01
0215  4033                CALL 2,WRITE     /INITIALIZE
0216  0205 06
                                            /I/O HANDLER
0217  6201 05            ARG N            /DEVICE NUMBER
0220  0256 01
                                            /1=TELETYPE
0221  6201 05            ARG FORMT        /FORMAT SPECI-
0222  0240 01
                                            /FICATION
0223  4033                CALL 1,IOH       /TYPE INTEGER
0224  0106 06
                                            /NUMBER
```

```
0225  6201 05           ARG C
0226  0261 01
0227  4033              CALL 1,IOH       /TYPE FLOATING
0230  0106 06                            /POINT NUMBER
0231  6201 05           ARG D
0232  0262 01
0233  4033              CALL 1,IOH       /COMPLETE THE I/O
0234  0106 06
0235  6211              ARG 0
0236  0000
0237  7402              HLT

0240  5047       FORMT,  TEXT "('THE ANSWERS ARE',IS,F7.2)"
0241  2410
0242  0540
0243  0116
0244  2327
0245  0522
0246  2340
0247  0122
0250  0547
0251  5411
0252  6554
0253  0667
0254  5662
0255  5100
0256  0001       N,      1
0257  0002       A,      2
0260  0002       B,      2
0261  0000       C,      0
0262  0000       D,      BLOCK 3
0263  0000
0264  0000

                         END
```

The binary tape produced by the assembly was then run using OS/8 with the following results:

```
THE ANSWERS ARE    4   4.00
```

chapter 5

FLAP/RALF

INTRODUCTION

FLAP and RALF are assemblers that translate PDP-8 or PDP-12 processor and floating point processor (FPP) operation codes in a source program into binary codes in two or three passes. The first pass assigns numeric values to the symbols and places them in the symbol table, the second pass generates the binary coding, and the third pass generates the program listing. FLAP/RALF is used to assemble programs using the FPP instructions and capabilities. Numeric values can be calculated as 12-bit integers, 15-bit integers, 24-bit double precision fractions, 3-word floating point values, or 6-word extended precision floating point values. Refer to the *FPP User's Guide,* DEC-12-GQZA-D, for detailed information on the floating point processor and its instruction set.

FLAP is designed to run on an OS/8 System with a Floating Point Processor (FPP) without any other supporting programs. It generates absolute binary output which is legal input to the OS/8 Absolute Loader (ABSLDR). RALF, an extension of FLAP, is part of the OS/8 FORTRAN IV System. It accepts assembly language files, or FORTRAN compiler output and generates relocatable binary modules that can be loaded by the relocatable loader LOAD (also part of the OS/8 FORTRAN IV System).

The following sections describe the syntax, instruction formats, addressing modes, and pseudo-operators in the assemblers. The special features of RALF involving relocatable assembly are described in the section entitled "RALF Features."

HARDWARE REQUIREMENTS

The minimum hardware configuration for FLAP is a PDP-8 or PDP-12 with a Floating Point Processor (FPP). The minimum hardware configuration for RALF is a PDP-8 or PDP-12 OS/8 System.

STATEMENT SYNTAX

A source program is a sequence of coding statements in the general format:

tag,instruction (space)expression (space)/ comment

A physical line of coding may be up to 127 characters long and is terminated by a carriage return. A semicolon can be used in a line of code (except in the comment field) to terminate a logical statement, thus permitting several statements to be typed on a single line. However, a set of logical statements separated by semicolons must not exceed the 127 character limit.
A space is required in a statement:

- after an instruction mnemonic
- before a slash (/) used to indicate a comment
- as an OR operator

Multiple spaces or tabs are equivalent to a single space. These characters are optional after the comma defining a tag, after the = sign that sets a value, and before a statement.

Tags

A statement tag is indicated by preceding the statement to be labeled with a user-defined symbol followed by a comma. This format assigns the current value of the location counter to the tag.

Instructions

An instruction may be a PDP-8 operation code, an FPP12 operation code, a FLAP pseudo-operator, or a RALF pseudo-operator.

Expressions

An expression can contain:

1. A user-defined symbol (equated symbol or tag).
2. The symbol ".", which has a value equal to the current location counter.
3. A numeric constant.
4. Two or more of the above combined by operators.

FPP and PDP-8 instructions are illegal symbols in expressions. User symbols can be 1 to 6 alphanumeric characters in length and

must start with a # or an alphabetic character. Any additional characters are ignored. Thus, the symbols:

```
#100
A
A1234
```

are acceptable, but in the symbol:

```
ASYMBOLMAYBEMORETHAN6CHARACTERS
```

only the first six characters are stored as the symbol name. In this case, all characters after ASYMBO are ignored. Up to 500 symbols may be defined by the user in an assembly.

All integer expressions are computed in 15-bit 2's complement arithmetic and then truncated if necessary (15 bits for 2-word FPP memory reference instructions and 12 bits for expressions). The following are examples of legal integer (address) expressions:

```
START+1
123
BUFSIZ*2+7600+300
(ADDRES+2
```

The radix pseudo-ops OCTAL and DECIMAL control the interpretation of numbers used in expressions. Decimal numbers larger than 32,767 and octal numbers larger than 77777 will be incorrectly converted, and cause the NE error. (Error messages are listed at the end of the chapter.)

Comments

A comment is a note added by the programmer at the end of a line of code, usually to indicate the logical sequence of the program. A slash (/), preceded by one or more spaces or tabs,is typed to specify the start of a comment. Comments must not contain angle brackets (or).

ARITHMETIC AND LOGICAL OPERATIONS

The operators used by FLAP/RALF and their functions in combining numbers or symbols to form expressions are:

Operator	Function
+	2's complement addition
−	2's complement subtraction
*	multiplication
/	division
space or tab	inclusive OR used to separate two instructions
!	inclusive OR
"	precedes an ASCII constant; e.g., "A has the octal value 301

Expressions are evaluated from left to right. They may not contain floating point constants.

PDP-8 OPERATION CODES

PDP-8 operation codes are legal defined mnemonics for use with FLAP/RALF. The following table lists the mnemonic, octal value, and operation of each PDP-8 operation code. PDP-8 code must be executed by the PDP-8 or PDP-12 processor. Assembler statements using these codes are said to be coded (or executed) in PDP-8 mode.

Table 5-1 PDP-8 Operation Codes

Mnemonic	Octal	Operation
Memory Reference Instructions:		
AND	0000	logical AND
TAD	1000	2's complement add
ISZ	2000	increment and skip if zero
DCA	3000	deposit and clear AC
JMS	4000	jump to subroutine
JMP	5000	jump
Group 1 Operate Microinstructions:		
NOP	7000	no operation
CLA	7200	clear AC
CLL	7100	clear link
CMA	7040	complement AC
CML	7020	complement link
RAR	7010	rotate AC and link right one
RAL	7004	rotate AC and link left one
RTR	7012	rotate AC and link right two

Table 5-1 PDP-8 Operation Codes (Cont.)

Mnemonic	Octal	Operation
RTL	7006	rotate AC and link left two
IAC	7001	increment AC
Group 2 Operate Microinstructions:		
SMA	7500	skip on minus AC
SZA	7440	skip on zero AC
SPA	7510	skip on positive AC
SNA	7450	skip on non-zero AC
SNL	7420	skip on non-zero link
SZL	7430	skip on zero link
SKP	7410	skip
OSR	7404	inclusive OR switch register with AC
HLT	7402	halt
Combined Microinstructions:		
CIA	7401	CMA IAC
LAS	7604	CLA OSR
IOT Microinstructions:		
Keyboard/Reader		
KSF	6031	skip if keyboard/reader flag = 1
KCC	6032	clear AC and keyboard/reader flag
KRS	6034	read keyboard/reader buffer
KRB	6036	clear AC and read keyboard buffer and clear keyboard flag
Teleprinter/Punch		
TSF	6041	skip if teleprinter/punch flag = 1
TCF	6042	clear teleprinter/punch flag
TPC	6044	load teleprinter/punch buffer, select and print
TLS	6046	load teleprinter/punch buffer, select and print, and clear teleprinter/punch flag
Program Interrupt		
ION	6001	turn interrupt on
IOF	6002	turn interrupt off
Extended Memory (Type MC8/I)		
CDF	62n1	change to data field n
CIF	62n2	change to instruction n
RDF	6214	read data field into AC
RIF	6224	read instruction field into AC
RMF	6244	restore memory field
RIB	6234	read interrupt

PDP-8 MODE ADDRESSING

In PDP-8 Mode, addressing is specified by the contents of the Memory Reference Instruction modified by the Data Field and Instruction Field Registers. Direct addressing, specified by bit 3=0, causes reference to the address given in bits 5-11 in page 0 of the current field, if bit 4=0, or to the current page, if bit 4=1. Indirect addressing, specified by bit 3=1, causes reference to the indirect address contained in the location specified by bits 4-11, used as above. The indirect address for AND,TAD,ISZ, and DCA refers not to the current field, but to the field specified in the Data Field Register. The JMP and JMS instructions refer to locations in the field specified in the Instruction Field Register.

The Data Field Register and Instruction Field Register are originally set through the console switches, and can be set under program control by means of the CIF and CDF instructions. The CIF instruction causes the Instruction Field Buffer to be set to the specified field. The CDF instruction causes the Data Field Register to be changed immediately. Other instructions allow the program to read, save, and restore the Data Field and Instruction Field Registers. Completion of execution of a JMP or JMS instruction causes the Instruction Field Register to be set to the contents of the Instruction Field Buffer. This procedure permits a program to choose a new field, then execute a jump from the current field to a chosen address in the new field.

The character % appended to the end of a memory reference instruction indicates indirect addressing and the character Z indicates a page 0 reference:

CURRENT PAGE		PAGE ZERO	
DIRECT	INDIRECT	DIRECT	INDIRECT
TAD A	TAD% A	TADZ A	TADZ% A
DCA B	DCA% B	DCAZ B	DCAZ% B

Spaces are not allowed between Memory Reference Instructions and either the Z or % characters. The Z must precede the % when both are used, i.e., do not write "DCA%Z".

FPP OPERATION CODES

The Floating Point Processor recognizes the following operation code formats. There are three forms of Data Reference Instructions

analogous to the Memory Reference Instructions, and three Special Format instruction forms analogous to the Operate Micro-Instructions.

Data Reference Instructions

Data Reference Instructions cause transfer between memory and the floating point accumulator, a 36-bit register in the FPP. The transfer may be 36 bits of floating point data or 24 bits of double precision fixed point fraction data, depending upon where STARTF or STARTD was most recently executed. In fixed point mode, the last 24 bits of the FAC or memory are used, and the exponent is unchanged. All eight of the Data Reference Instructions can be used in any of the three forms.

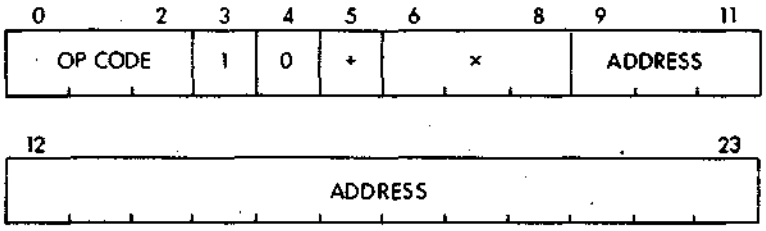
Throughout the description of the instructions that follow, these conventional symbols are used:

C( )	contents of enclosed quantity
FAC	floating accumulator
M	a variable multiplier =2 in Double Precision Mode =3 in Floating Point Mode
X	an indexing variable X=0, do not index $1 \leqslant X \leqslant 7$, use specified index register
X0	origin of index registers
Y	address computed
+	an increment bit =0, no incrementing =1, increment before using index
$\delta(X)$	symbol to avoid indexing X=0 $\quad \delta(X)=0$ $X \neq 0 \quad \delta(X)=1$

Op Code	Mnemonic	Data Function
0	FLDA	C(Y)→FAC
1	FADD	C(Y) + C(FAC)→FAC
2	FSUB	C(FAC) − C(Y)→C FAC
3	FDIV	C(FAC)/C(Y)→FAC
4	FMUL	C(FAC * C(Y)→FAC

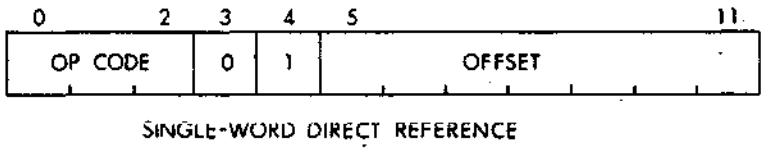
Op Code	Mnemonic	Data Function
5	FADDM	C(Y) + C(FAC)→Y
6	FSTA	C(FAC)→Y
7	FMULM	C(FAC) * C(Y)→Y

DATA REFERENCE INSTRUCTION FORMATS



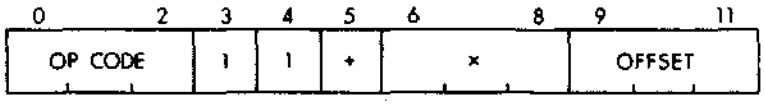
DOUBLE-WORD DATA REFERENCE INSTRUCTIONS

$$Y = C(\text{bits } 9\text{-}23) + M * (C(X + X0) + C(\text{bit } 5)) * \delta(X)$$



SINGLE-WORD DIRECT REFERENCE

$$Y = C(\text{base register}) + 3 * (\text{offset})$$



SINGLE-WORD INDIRECT REFERENCE

$$Y = C(\text{bits } 21\text{-}36 \text{ of } C((\text{base register}) + 3 * \text{offset})) + (M) * (C(X + X0) + C(\text{bit } 5)) * \delta(X)$$

$\delta(X) = 1$ if $X \neq 0$ and 0 if $X = 0$

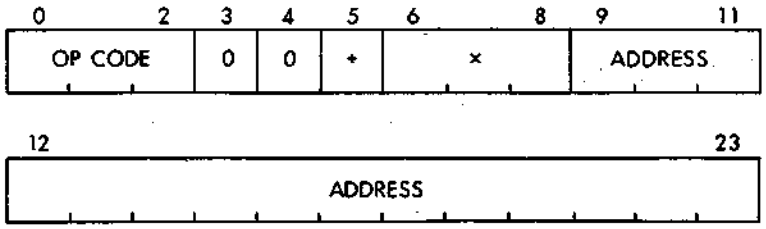
$M = 2$ if fixed-point mode

$= 3$ if floating-point mode

SPECIAL FORMAT 1 – Jump on count + trap

Op Code	Mnemonic	Function
2	JXN	The index register X is incremented if bit 5=1 and a jump is executed to the address contained in bits 9-23, if index register X is nonzero.
3 4 5 6 7		The instruction-trap status bit is set and the FPP12 exits causing a PDP interrupt. The unindexed operand address is dumped into the APT.

The two trap instructions with op codes 3 and 4 are assigned a special meaning by RALF, and the mnemonics TRAP3 and TRAP4 respectively. TRAP3 acts as a JMP to PDP-8 Mode; TRAP4 acts as a JMS to PDP-8 Mode. See the *FORTRAN IV SOFTWARE SUPPORT MANUAL* for details.



SPECIAL FORMAT 1

SPECIAL FORMAT 2 – Load index and add index

Op Code	Extension	Mnemonic	Function
0	10	LDX	The contents of the index register specified by the bits 9-11 are replaced by the contents of bits 12-23.
0	11	ADDX	The contents of bits 12-23 are added to the index register specified by bits 9-11.

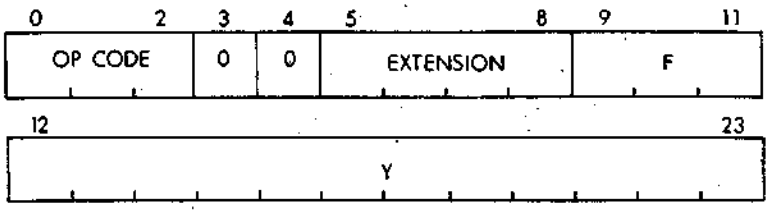
SPECIAL FORMAT 2 – Conditional jumps

Jumps, if performed, are to the location specified by bits 9-23 of the instruction.

Op Code	Extension	Mnemonic	Function
1	0	JEQ	Jump if FAC = 0
1	1	JGE	Jump if FAC $\geqslant$ 0
1	2	JLE	Jump if FAC $\leqslant$ 0
1	3	JA	Jump always
1	4	JNE	Jump if FAC $\neq$ 0
1	5	JLT	Jump if FAC < 0
1	6	JGT	Jump if FAC > 0
1	7	JAL	Jump if impossible to fix the floating point number contained in the FAC; i.e., if the exponent is greater than $(23)_{10}$.

SPECIAL FORMAT 2 – Pointer moves

Op Code	Extension	Mnemonic	Function
1	10	SETX	Set X0 to the address contained in bits 9-23 of the instruction.
1	11	SETB	Set the base register to the address contained in bits 9-23.
1	13	JSR	Jump and save return. Jump to the location specified in bits 9-23 and the return is saved in bits 21-35 of the first entry of the base page.
1	12	JSA	An unconditional jump is deposited in the address and address+1, where address is specified by bits 9-23. The FPC is set to address+2.



SPECIAL FORMAT 2

SPECIAL FORMAT 3 — NORMAL SIZE

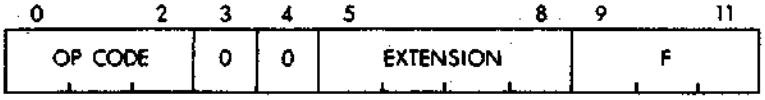
Op Code	Extension	Mnemonic	Function
0	1	ALN	The mantissa of the FAC is shifted until the FAC exponent equals the contents of the index register specified by bits 9-11. If bits 9-11 are zero, the FAC is aligned so that the exponent = $(23)_{10}$. Setting the exponent = $(23)_{10}$ integerizes or fixes the floating-point number. The JAL instruction tests to see if fixing is possible. In double-precision mode, an arithmetic shift is performed on the FAC fraction. The number of shifts is equal to the absolute value of the contents of the specified index register. The direction of shift depends on the sign of the index register contents. A positive sign indicates a shift toward the least significant bit, while a negative sign indicates a shift toward the most significant bit. The FAC exponent is not altered by the ALN instruction in double-precision mode.
0	2	ATX	The contents of the FAC are fixed and the least significant 12 bits of the mantissa are

Op Code	Extension	Mnemonic	Function
			loaded into the index register specified by bits 9-11. In double-precision mode the least significant 12 bits of the FAC are loaded into the specified index register. The FAC itself is not altered by the FATX instruction.
0	3	XTA	The contents of the index register specified by bits 9-11 are loaded right-justified into the FAC mantissa. The FAC exponent is loaded with $(23)_{10}$ and then the FAC is normalized. This operation is typically termed floating a 12-bit number. In double-precision mode, the FAC is not normalized.
0	4	NOP	The single-word instruction performs no operation.
0	5-7	reserved	These codes are reserved for instruction set expansion and should not be used.
0	12-17		
1	14-17		

SPECIAL FORMAT 3 – OPERATE

Op Code	Extension	9-11 Bits	Mnemonic	Function
0	0	0	FEXIT	Dump active registers into the APT, reset the FPP RUN flip-flop to the 0 state, and interrupt the PDP-8 processor.
0	0	1	FPAUSE	Wait for synchronizing signal. IOT FFST (6555) will restart the instruction following FPAUSE.

Op Code	Extension	Mnemonic		Function
0	0	2	FCLA	Zero the FAC mantissa and exponent.
0	0	3	FNEG	Complement FAC mantissa. This instruction produces the true negative, not the bit-by-bit complement.
0	0	4	FNORM	Normalize the FAC. In double-precision mode FNORM is a NOP.
0	0	5	STARTF	Start floating-point mode.
0	0	6	STARTD	Start double-precision mode.
0	0	7	JAC	Jump to the location specified by the least significant 15 bits of the FAC mantissa.



SPECIAL FORMAT 3

FPP MODE ADDRESSING

The FLAP/RALF assembler is able to interact with and effectively use the rather complex addressing scheme of the FPP. This addressing scheme allows the FPP to access a full 32k words of core through 15-bit addresses. It also allows the FPP to access a movable base page through 7-bit addresses. The FPP can also use 2 or 3 bits to specify an index register from a movable set that can modify the address. The FORTRAN compiler makes extensive use of this addressing freedom, particularly in the subroutine calls.

The base page is a block of 128 floating point variables, or 384 12-bit words. The Special Format 2 instruction SETB gives the FPP the origin of the base page. The pseudo-op BASE is used to

pass the base page origin to the FLAP/RALF assembler. The origin of the base page may be changed as often as necessary. The first 8 locations of the base page serve as a pointer to memory.

The index registers are a block of seven 12-bit words in memory. The Special Format 2 instruction SETX gives the FPP the origin of the index registers. The locations used for the index registers may be changed as often as necessary.

The three forms of Data Reference Instructions compute the address of the data referenced in three different ways. The line of print below the diagram of each instruction shows symbolically how each address is computed. The address computation for the first form begins with the 15-bit address in bits 9-23 of the instruction. If X (bits 6–8) is zero, this is the address used. If X is nonzero, the contents of the specified memory location, X+X0 (where X0 is the beginning of the index registers, set by SETX), is used as an index. If bit 5 of the instruction is equal to one, the index value is incremented by one. The index value remains incremented after the instruction is completed. The resulting index value is multiplied by either two or three, depending upon whether the FPP is in Double Precision Fixed Point Mode (STARTD) or Floating Point Mode (STARTF). This index is then added to the original address (bits 9-23 to form the address used.

The second data reference form is used to address the locations on the base page. The contents of bits 5-11 of the instruction are multiplied by three and added to the origin of the base page, set by the SETB instruction.

Note that the offset on the base page always assumes Floating Point (3-word) variables. It is wise to prevent use of the base page for storage of double precision fixed point variables or instructions.

The third form of data reference instruction provides an indirect or indexed indirect mode of address. The offset, bits 9-11 of the instruction, are multiplied by three and added to the origin of the base page, to give the address of a 3-word variable. The last 15 bits of this word are used for the address of the data. This address may be modified by the index register exactly the same as in the first form.

The FLAP/RALF Assembler will choose the form of the data reference instruction that is generated. The second form (one word direct) is used instead of the first form (two word direct) whenever

the data lies on the base page; no indexing is involved. The indirect form is used whenever indirect addressing is called for by a % in the assembler source statement.

LITERALS

Only FLAP allows literals in PDP-8 code. By starting an expression in a PDP-8 memory reference instruction with a left parenthesis or square bracket (as explained below), the value after it is taken "literally" by FLAP. There is then no need to specify an address or tag that contains the value. Internally the value of the literal expression is the address of the word generated by FLAP that contains the evaluated expression.

If the expression starts with a left parenthesis, (, then the literal is placed at the end of the current page. If it starts with a left bracket, [, the literal is placed at the end of page 0. Literal tables are built backwards from the end of the page so that the most recently defined literal has the lowest core address.

If the origin is changed to a new page, the previous page's literals are output and the literal table is reset. If the origin is reset to a previous page which contained literals, those literals may be overlayed by any new literals. The previously-defined literals will not be available for reference. For this reason, it is best to complete all coding on any non-zero page before moving to another.

If the field is changed, the literals on page 0 of the previous field are output and the page 0 literal table is reset. For this reason, it is best to complete all coding in any one field before moving to another.

Because locations 0-17 are generally used for interrupts and autoindex registers, there may be only 112_{10} (160_8) literals on page 0.

The following examples illustrate the use of literal expressions with memory reference instructions:

TAD (POINTER	generates a literal with the lower 12 bits of the address of POINTER at the end of the current page.
TAD [10	generates a literal containing 0010 at the end of page 0.

The left bracket, [, is typed as a SHIFT/K on an ASR-33.

Literals may not be nested. For example, the expression:

```
TAD (TAD [10
```

LINKS

Links are only generated by the FLAP assembler. If a PDP-8 memory reference is made to an address that is not on the same page as the instruction, FLAP creates an indirect address linkage on the current page. The address can, therefore, be accessed during the second pass of the Assembler. For example, the coding:

```
                    ORG 200
00200 1777'                 TAD A

00377 0400
                    PAGE
00400 1025          A,      1025
```

is equivalent to

```
                    ORG 200
00200 1777                  TAD I X

                    ORG 377
00377 0400          X,      A

                    PAGE
0400 1025           A,      1025
```

All instructions generating links are flagged in the listing with an apostrophe (') following the generated code. The total number of links is printed at the completion of assembly.

DATA SPECIFICATION

A logical line of code may consist of only an expression. Such expressions can function as flags, pointers, constants, or symbols. If the expression is larger than 12 bits, it will be truncated to 12 bits.

PSEUDO-OPERATORS

A pseudo-operator is a defined mnemonic code that is included in the source program as a logical line to control some functions of the assembler. Binary code may or may not be generated by a

pseudo-op, depending on its function. The FLAP/RALF pseudo-ops and their functions are listed below.

EQUATE (=)

The symbol to the left of the = is assigned the value of the expression to the right of it.

OCTAL

All integers which follow are assumed to be in octal radix. The digits 8 and 9 are flagged if they occur in octal radix. The radix is initially set to octal by FLAP.

DECIMAL

All integers which follow are assumed to be in decimal radix.

PAGE

The current location counter is set to the beginning of the next core page. This pseudo-op is not in the RALF assembler.

BASE n

The location of the base page, n, is placed in FLAP/RALF base register to be used in calculating single-word addresses. The argument, n, is an expression denoting a 15-bit address. The expression may not contain any symbols that are defined after the BASE pseudo-op occurs. A correct sequence is illustrated below.

```
ORG 400

A,      F 2,0

B,      F 3,0

        BASE A          /SET ASSEMBLER BASE REGISTER
        SETB A          /SET FPP BASE REGISTER

        FLDA A
```

If no BASE pseudo-op is included, all FPP memory reference instructions will be 2 words. Refer to the sections on FPP addressing, and on referencing memory.

TEXT

A string of text may be entered by using the pseudo-op TEXT followed by a space or tab, a delimiting character, a string of text, and the same delimiting character, issued in that order. The first

printing character after TEXT is the delimiter and the text string is all the characters that follow it until the next occurrence of the delimiter or a carriage return. The characters space, tab, , , and / cannot be delimiters. For example:

TEXT %DATA%

causes the word DATA to be printed with the code at assembly time as:

```
00200 0401        TEXT %DATA%
00201 2401
```

END

Input is terminated. (This pseudo-op is optional, and is never printed on the listing.)

INDEX n

Set the location of the first FPP index register to n.

ORG expr

The current location counter is assigned the value of the lower 15 bits of the address expression expr. The expression should contain only symbols which have previously been defined. For example, to set the origin at location 400 of field 1, the pseudo-op used is ORG 10400.

If the ORG pseudo-op is omitted, an origin of 200 in field 0 is assumed, but the origin setting is not included in the binary output file. For useful results, the user program must begin with an ORG pseudo-op.

ZBLOCK n

Assemble a block of n words containing 0.

LISTOF

Continue assembly but inhibit further listing. There is no effect on the first two passes or if the listing is currently inhibited. This pseudo-op never appears in the listing.

LISTON

Cease to inhibit the listing. There is no effect on the first two passes if the listing is not currently inhibited.

IFnnn (conditional assembly)

FLAP/RALF has ten conditional pseudo-ops. Four of them require an argument expression:

pseudo-op	function
IFZERO n <	assemble if n is zero
IFNZRO n <	assemble if n is not zero
IFPOS n <	assemble if n is positive
IFNEG n <	assemble if n is negative

where n is an integer expression. For each of the above conditional pseudo-ops, the expression n is evaluated and, if it fulfills the conditions of the pseudo-op (e.g., n equals zero for IFZERO), the subsequent coding is assembled. If the condition is not met, the subsequent coding is ignored until a matching > is encountered. Assembly is continued after the >.

The fifth and sixth pseudo-ops are used in the format:

IFREF symbol < — assemble if symbol was previously defined or referenced.

IFNDEF symbol < — where symbol may be defined or undefined. When an IFREF statement is encountered, subsequent coding is assembled if the symbol after the pseudo-op has been defined or referenced in a previous statement. Note that use of a symbol with an IFREF pseudo-op or in a statement that was skipped during assembly because the condition required by a preceding conditional pseudo-op was not met does not constitute a reference to the symbol. If the symbol has not been previously defined or referenced assembly is continued after the matching > is found.

The seventh through tenth pseudo-ops are:

IFSW n < — assemble the enclosed code if the switch n was set in the input/output file specification to the command decoder, i.e. /n or (n).

IFNSW n < assemble the enclosed code if the switch n was not set.

IFFLAP < assemble the enclosed code if the assembler is FLAP. This pseudo-op is intended for use in programs which may be assembled either by RALF or by FLAP.

IFRALF < do not assemble the enclosed code if the assembler is FLAP.

Conditionals may be nested. A possible nested conditional is

```
IFFLAP < IFREF  A < A=263>>
```

Use of some of the conditional assembly pseudo-ops is illustrated in the next example.

```
                        IFPOS .1 <
                A,      F 0,0
                        >
                        IFNEG .1 <
00200 0000      B,      F 0,0
00201 0000
00202 0000
                        >
                        IFREF A <
                        TAD A
                        >
00203 1200              TAD B
                        >
                        IFREF C <
                        TAD C
                        >
                        IFNDEF D <
                        D=5
                        >

NO ERRORS
2 SYMBOLS, NO LINKS

 B      00200   D       00005
```

REPEAT n

Assemble the following line n times.

S n

Generate a 1-word constant with value n. RALF does not support this pseudo-op.

F n

Generate a 3-word floating point constant with value n. The argument n may be written as a decimal floating point number, for example 2.0, or in standard exponential format, for example 2E10. In standard exponential format, 2E10 is equal to 2×10^{10}.

E n

Generate a 6-word extended precision floating point constant with value n. The argument n may be written as a decimal floating point number or in standard exponential format.

ADDR

Generates a two-word address corresponding to the value of the argument.

COMMON

Causes the assembler to enter the COMMON section whose name follows the pseudo-op. Subsequent output is placed in the named COMMON section until another section defining pseudo-op is encountered.

COMMZ

Define Field 1 8-mode page 0 section. Used to give PDP-8 page 0 section for the Loader.

DPCHK

Indicates that the current module requires double precision hardware in order to execute.

ENTRY

Defines program entry point. The symbol whose name follows the ENTRY pseudo-op can be used as an external symbol by other programs. Multiple entry points with the same name are accepted by the assembler but cause an error from the loader.

EXTERN

The symbol following this pseudo-op is defined to be external to this assembly.

FIELD1

Define FIELD1 8-mode section. Used to give field 1 name of section for the Loader.

SECT

Define program section, used at the beginning of subprograms to give the name of section for the Loader. For example:

```
        SECT    SUBROU
        JA      START
        BASE     .
B0,     F       0.
        etc.
```

SECT8

Define 8-mode program section. Used at the beginning of 8-mode subprograms.

REFERENCING MEMORY

A PDP-8 computer with an FPP is basically a 32K machine. All of this memory may be referenced through the 15 bit address field provided by the 2-word memory reference instructions. When it is necessary to conserve memory, the base page and the short form (1 word) of the memory reference instructions can be used. Those instructions that have a floating point operand can use this short form:

FADD	FDIV	FMUL	FSTA
FADDM	FLDA	FMULM	FSUB

The base page is a movable page 0 assigned by the user. To determine the location referred to by the operand of the single word instruction the displacement field (address expression) is multiplied by 3 and added to the contents of the base register. Thus, using the single word form of the instruction, any location within 128*3 locations of the base register can be referenced. (Only 128*3 locations can be accessed because the displacement field has only 7 bits.) The location of the base page (via BASE) and the operands (via ORG,=,etc.) must be defined in the coding before the FPP instruction. Then the short form of the instruction will be executed unless the suffix # is added, forcing the long (2 word) form.

Consider the following example of the BASE pseudo-op:

```
                        ORG 200
00200 0002      A,      F 2.0
00201 2000
00202 0000
00203 0002      B,      F 3.0
00204 3000
00205 0000
00206 0003      C,      F 5.0
00207 2400
00210 0000
00211 0000      D,      F 0.0
00212 0000
00213 0000


                        BASE 200
                        SETB 200
00214 1110
00215 0200
00216 0200              FLDA A
00217 1201              FADD B
00220 4202              FMUL C
00221 6203              FSTA D          /D=(A+B)*C
```

This same program can be written with a subroutine:

```
                ORG 200
00200 0002      A,      F 2.0
00201 2000
00202 0000
00203 0002      B,      F 3.0
00204 3000
00205 0000
00206 0003      C,      F 5.0
00207 2400
00210 0000
00211 0000      D,      F 0.0
00212 0000
00213 0000

00214 1110              SETB 200
00215 0200
00216 1120              JSA SUBR
00217 0400
00220 7402              HLT
                        BASE 0

                ORG 400

00400 0000      SUBR,   0;0             /LEAVE 2 WORDS FOR JSA
00401 0000
00402 0200              FLDA 0          /A
00403 1201              FADD 3          /B
00404 4202              FMUL 6          /C
00405 6203              FSTA 11         /D
00406 1030              JA SUBR         /--RETURN--
00407 0400
```

This routine performs the same operation as the first one. The values 0, 3, 6, and 11 are used with BASE 0 so that the assembler generates the correct 1 word instructions.

RALF FEATURES

RALF symbols may be absolute, relocatable, or external. When a relocatable symbol appears in an assembled value, an indicator is placed in the binary output file so that the relocating loader (LOAD) will add the base loading address of the assembled value to arrive at the value to be loaded. If an external symbol appears, the loader will look up the name of the symbol in its symbol table and substitute the value found there for the symbol. The loader symbol table contains all symbols defined by the SECT, SECT8, FIELD1, COMMON, COMMZ and ENTRY pseudo-ops of RALF. Expressions using both absolute and relocatable terms are evaluated as follows (where "op" is one of the set [+ – * / & !] and "op1" is one of the set [* / & !]):

Expression	Evaluated
numeric constant	absolute
label	relocatable
absolute op absolute	absolute
relocatable + absolute	relocatable
relocatable – relocatable	absolute relocatable
absolute – relocatable	ERROR
expression op1 relocatable	ERROR
relocatable op1 expression	ERROR

RALF code is divided into sections. each section is a separately loadable entry within the assembly. These sections are defined via one of the five pseudo-ops: SECT, SECT8, FIELD1, COMMON and COMMZ. Section names are placed in the External Symbol Dictionary (ESD) which is used by the relocating loader to build its symbol table. The pseudo-ops ENTRY and EXTERN allow RALF programs to insert other symbols into the ESD and to refer to these symbols in other RALF programs at load time. Table 5-3 lists the RALF pseudo-ops and their meanings.

Core Allocation

The user who wishes to link RALF modules containing PDP-8 mode code must be aware of the core allocation algorithm of the loader. Five RALF pseudo-ops may be used to specify a section: SECT, COMMON, SECT8, FIELD1, and COMMZ. These sections are loaded independently by the loader, including those in the same RALF module. SECT is used to begin a section of RALF code that can be loaded into any level and overlay and anywhere in field 1 and above. COMMON is used to begin a section with a given name available to COMMON statements in FORTRAN or other RALF modules. SECT8 is used to begin a section of RALF code that is loaded into level MAIN and is required to begin and end on a page boundary. FIELD1 is used to begin a section subject to all the restrictions of SECT8 and in addition must be loaded into field 1. COMMZ is used to begin a section subject to all the restrictions of FIELD1 and must be loaded into page 0.

The first COMMZ section encountered is forced to begin at location 10000, thus enabling a page 0 in field 1. COMMZ sections of the same name are handled like COMMON sections of the same name (i.e., they are combined into one common section). This feature allows 8-mode code in different modules to share page 0, provided that the modules do not destroy each other's page 0 allocations. Suppose two modules were to share page 0, with the first using location 0-17 and the second using locations 20-37:

```
                                    /Module A
        COMMZ SHARE
P1,     1
P2,     2
KSUBA1, SUBA1
KSUBA2, SUBA2
        .
        .
        .                           /Should not go over
LASTA,  -1                          /20 locations

FIELD1  A

        TADZ P1
        JMSZ% KSUBA1
        .
```

```
                .
                .                       /Module B
                COMMZ  SHARE
                ORG  .+20               /ORG past module A's
                                        /Page 0
P3,             3
P4,             4
KSUBB,          SUBB
                .
                .
                .
LASTB           -2
FIELD1          B
                TADZ  P3
                .
                .
                .
```

The two COMMZ sections will be put on top of one another, however, because of the ORG .+20 in module B, they will effectively reside back to back. When the image is loaded, the COMMZ sections will look as follows:

LOC	CONTENTS	
1 0000	1	
0001	2	
2	SUBA1	
3	SUBA2	
.		
.		
.		
1 0017	-1	/LASTA
1 0020	3	
21	4	
22	SUBB	
.		
.		
.		
37	-2	/LASTB

If module A is to reference module B's page 0, the procedure is:

```
P3=20
TADZ P3
```

Alternately, a duplicate of the source code for COMMZ SHARE may be included in module B. Modules that are using the same COMMZ section must be aware of how it is divided up. Although COMMZ SHARE takes only 40 locations, the loader allocates a full 200 locations to it. All 8-mode section core allocations are always rounded up so that they terminate on a page boundary. If COMMZ sections of different names exist, they are accepted by the loader and inserted into field 1, but only one COMMZ is the real page 0. In general, it is unwise to have more than 1 COMMZ section name.

If there is more than one COMMZ pseudo-op in a module, they are stacked one behind the other, but there is no way of specifying which one starts at absolute location 0 of field 1. COMMZ sections are allocated by the loader before FIELD1 sections.

For users who intend to write 8-mode code that will execute in conjunction with certain 8-mode library routines, the layout of PDP-8 FIELD1 #PAGE 0 is:

LOCATION	USE
0-1	Temps for any non-interrupt time routine.
2-13	User locations.
14-157	System locations.
160-177	User locations.

1. Do not define any COMMZ sections other than the system COMMZ which is #PAGE0.
2. If the system page 0 is desired, it will be pulled in from the library if EXTERN #DISP appears in the code.
3. Do not use any part of page 0 reserved for the system.

FIELD1 sections are identical to COMMZ sections in most respects. Memory allocation for FIELD1 sections is assigned after COMMZ sections. however, and FIELD1 sections are combined with FORTRAN COMMON sections of the same name as well as other FIELD1 sections of the same name. The first difference ensures that COMMZ will be allocated page 0 storage even in the presence of FIELD1 sections. The second allows PDP-8 code to be loaded into COMMON, making it possible to load initialization

code into data buffers.Two FIELD1 sections with the same name may be combined in the same manner as two COMMZ sections.

The primary purpose of COMMZ is to provide a PDP-8 page 0; the primary purpose of FIELD1 is to ensure that 8-mode code will be loaded into field 1 and that generating CIF CDF instructions in-line is not necessary. SECT8 sections may not be combined in the manner of a COMMON and are not ensured of being placed into field 1.

A section begins when a pseudo-op with its name first appears. A SECT8 section is not combined with another of the same name in another RALF module. However, the second use of the same name in the same module continues a section. For example:

```
SECT8  PARTA
   .
   .
   .
SECT8  PARTB
   .
   .
   .
SECT8  PARTA
```

The second mention of PARTA in the same module continues the source where the first mention of PARTA ended. (There is a location counter for each section.)

An 8-mode section does not have to be less than a page in length; however, the programmer should be aware that a SECT8 section which exceeds one page may be loaded across a field boundary and could thereby produce disastrous results at execution time. For this reason, it is generally unwise to cross pages in SECT8 code. This situation will never occur on an 8K configuration. If the total amount of COMMZ and FIELD1 code exceeds 4K, the loader generates an OVER CORE message. The loader generates an MS error for any of the following:

1. A COMMZ section name is identical to some entry point or some non-COMMZ section name.
2. A FIELD1 section name is identical to some entry point or a SECT, SECT8 or COMMZ section name.
3. A SECT8 section name is identical to an entry point or some other section name.

COMMZ sections, like FORTRAN COMMONS, are never entered in the library catalog.

RALF Programming Notes

The best means of creating RALF modules that can be called from FORTRAN programs is to write a skeleton FORTRAN subroutine. The subroutine should be written so that it can be called with the same "call" statement to be used for the RALF subroutine. This FORTRAN subroutine is then compiled with the RALF output sent to a mass storage file. This file may be modified using EDIT or TECO to create the desired module.

The address pseudo-op (ADDR) which generates a two word relocatable 15 bit address (i.e., JA TAG without use of JA) might prove useful in 8-mode routines. The following example demonstrates a way in which an 8-mode routine in one RALF module calls an 8-mode routine in another module:

```
        EXTERN SUB
        .
        .
        RIF             /Set DF to current
        TAD     ACDF    /IF for return
        DCA     .+1
        0               /CDF X
        TAD     KSUB    /Make a CIF from
        RTL     CLL     /Field bits
        RAL
        TAD     ACIF
        DCA     .+1
        0               /CIF to field
                        /Containing SUB
        JMS%    KSUB+1
KSUB,   ADDR    SUB     /Pseudo-op to
                        /Generate 15 bit
                        /ADDR of subroutine
                        /SUB
ADCF,   CDF
ACIF,   CIF
```

In general the address pseudo-op can be used to supply an 8-mode section with an argument or pointer external to the section.

FPP and 8-mode code may be combined in any RALF section. PDP-8 mode routines must be called in FPP mode by either:

```
        TRAP3 SUB
or      TRAP 4 SUB
```

A TRAP3 SUB causes FRTS to generate a JMP SUB with interrupts on and the FPP hardware (if any) halted. TRAP4 generates a JMS SUB under the same conditions. The return from TRAP4 is:

```
CDF CIF 0
JMP% SUB
```

The return from TRAP3 is:

```
CDF CIF 0
JMP% RETURN+1
```

```
        EXTERN  #RETRN
RETURN, ADDR    #RETRN
```

It is not possible to call PDP-8 mode subroutines from FORTRAN. A RALF subroutine called from FORTRAN will be entered in FPP-mode, it may branch into PDP-8 mode code using a TRAP3 or TRAP4.
Communication between FPP and 8-mode routines is best done at the FPP level because of greater flexibility in both addressing and relocation in FPP mode. The following routine demonstrates how to pass an argument to, and retrieve an argument from, an 8-mode routine:

```
EXTERN SUB
EXTERN SUBIN
EXTERN SUBOUT
.
.
.
FLDA    X       /Arg for SUB
FSTA    SUBIN
TRAP4   SUB     /Call SUB
FLDA    SUBOUT  /Get result
FSTA    Y
```

If the 8-mode routine SUB were in the same module as the FPP routine, the EXTERNs would not be necessary. In practice it is common for FPP and 8-mode routines that communicate with one another to be in the same section. A number of techniques can be used to pass arguments. For example, an FPP routine could move the index registers to an 8-mode section and pass single precision arguments via ATX.

Because 8-mode routines are commonly used in conjunction with FPP code (generated by the compiler), the 8-mode programmer should be familiar with OS/8 FORTRAN IV subroutine calling conventions. The general code for a subroutine call is a JSR, followed by a JA around a list of arguments, followed by a list of pointers to the arguments. The FPP code for the statement:

```
CALL SUB (X,Y,Z)
```

would be

```
        EXTERN SUB
        JSR    SUB
        JA     BYARG
        JA     X
        JA     Y
        JA     Z
BYARG,  .
        .
        .
        .
```

The general format of every subroutine obeys the following scheme:

```
        SECT   SUB
        JA     #ST          /Jump to start of
                            /Routine
        TEXT   +SUB+        /Needed for
                            /Trace back
RTN,    SETX   XSUB         /Reset SUB's index
        SETB   BSUB         /And base page
BSUB,   FNOP                /Start of base page
        JA     .
        .
```

```
        ORG   BSUB+30    /Restart for SUB
        FNOP:JA RTN
GOBAK,  FNOP:JA .        /Return to
                         /Calling program
```

Location 0000 of the calling routine's base page points to the list of arguments, if any, and may be used by the called subroutine provided that it is not modified. Location 0003 of the calling routine's base page is free for use by the called subroutine. Location 0030 of the calling routine's base page contains the address where execution is to continue upon exit from the subroutine, so that a subroutine should not return from a JSR call via location 0 of the calling routine:

```
CORRECT          INCORRECT
FLDA 30          FLDA 0
JAC              JAC
```

This return allows the calling routine to reset its own index registers and base page before continuing in-line execution. General initialization code for a subroutine would be:

```
        SECT    SUB
        JA      #ST
        .
        .
        .
        BASE    0
#ST,    STARTD                /So only 2 words
                              /Will be picked up
        FLDA    30            /Get return JA
        FSTA    GOBAK         /Save it
        FLDA    0             /Get pointer to list
        SETX    XSUB          /Set SUB's XR
        SETB    BSUB          /Set SUB's Base
        BASE    BSUB
        INDEX   XSUB
        FSTA    BSUBX         /Store pointer
                              /Somewhere on Base
```

```
.
.
.
STARTF                  /Set F mode before
JA          GOBAK       /Return
```

The above code can be optimized for routines that do not require full generality. The JA #ST around the base page code is a convenience which may be omitted. The three words of text are necessary only for error traceback and may also be omitted. If the subroutine is not going to call any general subroutines, the SETX and SETB instructions at location RTN and the JA RTN at location 0030 are not necessary. If the subroutine does not require a base page, the SETB instruction is not necessary in subroutine initialization; similar remarks apply to index registers. If neither base page nor index registers are modified by the subroutine, the return sequence:

```
FLDA 0
JAC
```

is also legal. In a subroutine call, the JA around the list of arguments is unnecessary when there are no arguments. A RALF listing of a FORTRAN source will provide a good reference of general FPP coding conventions.

The AMOD routine is listed in Figure 5-1 to illustrate an application of the formal calling sequence. It also includes an error condition check and picks up two arguments. When called from FORTRAN, the code is AMOD(X,Y).

If a PDP-8 mode subroutine is longer than one page and values are to be passed across page boundaries, the address pseudo-op, ADDR, is required. The format is:

```
AVAR1, ADDR VAR1
```

```
/
/
/
/       A M O D
/       - - - -
/
/SUBROUTINE     AMOD(X,Y)
        SECT    AMOD            /SECTION NAME(REAL NUMBERS)
        ENTRY   MOD             /ENTRY POINT NAME(INTEGERS)
        JA      #AMOD           /JUMP TO START OF ROUTINE
        TEXT    +AMOD  +        /FOR ERROR TRACE BACK
AMODXR, SETX    XRAMOD          /SET INDEX REGISTERS
        SETB    BPAMOD          /ASSIGN BASE PAGE
BPAMOD, F 0.0                   /BASE PAGE
XRAMOD, F 0.0                   /INDEX REGS.
AMODX,  F 0.0                   /TEMP STORAGE
        ORG     10*3+BPAMOD     /RETURN SEQUENCE
        FNOP
        JA      AMODXR
        0
AMDRTN, JA      .               /EXIT
        EXTERN  #ARGER
AMODER, TRAP4   #ARGER          /PRINT AN ERROR MESSAGE
        FCLA                    /EXIT WITH FAC=0
        JA      AMDRTN
        BASE    0               /STAY ON CALLER'S BASE PG
/LONG ENOUGH TO GET RETURN ADDRESS
MOD,                            /START OF INTEGER ROUTINE SAME AS
#AMOD,  STARTD                  /START OF REAL NUM. ROUTINE
        FLDA    10*3            /GET RETURN JUMP
        FSTA    AMDRTN          /SAVE IN THIS PROGRAM
        FLDA    0               /GET POINTER TO PASSED ARG
        SETX    XRAMOD          /ASSIGN MOD'S INDEX REGS
        SETB    BPAMOD          /AND ITS BASE PAGE
        BASE    BPAMOD
        LDX     1,1
        FSTA    BPAMOD
        FLDAZ   BPAMOD,I        /ADDR OF X
        FSTA    AMODX
        FLDAZ   BPAMOD,1+       /ADDR OF Y
        FSTA    BPAMOD
        STARTF
        FLDAZ   BPAMOD          /GET Y
        JEQ     AMODER          /Y=0 IS ERROR
        JGT     .+3
        FNEG                    /ABS VALUE
        FSTA    BPAMOD
        FLDAZ   AMODX           /GET X
        JGT     .+5
        FNEG                    /ABS VALUE
        LDX     0,1             /NOTE SIGN
        FSTA    AMODX           /SAV IN A TEMPORARY
        FDIV    BPAMOD          /DIVIDE BY Y
        JAL     AMODER          /TOO BIG.
        ALN     0               /FIX IT UP NOW.
        FNORM
        FMUL    BPAMOD          /MULITPLY IT.
        FNEG                    /NEGATE IT.
        FADD    AMODX           /AND ADD IN X.
        JXN     AM,1            /CHECK SIGN
        FNEG
AM,     JA      AMDRTN          /DONE
```

Figure 5-1 AMOD Routine

This generates a two-word (15 BIT) reference to the proper location on another page, here VAR1. For example, to pass a value to VAR1, possible code is:

```
00124  1244  TAD      VAR2        /Value on this page
00125  3757  DCA%     AVAR1+1     /Pass through 12-bit
                                  /location
00156  0000  AVAR1,ADDR VAR1      /Field and
00157  0322                       /location of VAR1
```

Any reference to an absolute address can be effected by the ADDR pseudo-op.
If it is doubtful that the effective address is in the current data field, it is necessary to create a CDF instruction to the proper field. In the above example, suitable code to add to specify the data field is:

```
        TAD   AVAR1     /Get field bits
        RTL             /Rotate to bits 6-8
        RAL
        TAD   (6201     /Add a CDF
        DCA   .+1       /Deposit in line
        0               /Execute CDFn
```

If the subroutine includes an off-page reference to another RALF module (e.g., in FORLIB), it can be addressed by using an EXTERN with an ADDR pseudo-op. For example, in the display program, a reference to the non-interrupt task subroutine ONQB is coded as

```
        EXTERN          ONQB
ONQBX,  ADDR            ONQB
```

and is called by

```
        JMS%            ONQBX+1
```

No field change instruction is necessary here, because both library modules are defined by *field 1* pseudo-op's, and so are both in the same field.

RALF does not recognize LINC instruction or PDP-8 laboratory device instructions. Such instructions can be included in the subroutine by defining them by equate statements in the program.

For example, adding the statements:

```
PDP = 2
LINC = 6141
DIS = 140
```

takes care of all instructions for coding the PDP-12 display subroutine.

When writing a routine that is going to be longer than a page, it can be useful to have a non-fixed origin in order not to waste core and to facilitate modification of the code. A statement such as

```
IFPOS .-SECNAM&177-K<ORG
.-SECNAM&7600+200+SECNAM>
```

will start a new page only if the value [current location less section name] is greater than some K (start of section has a relative value of 0) where K≤177 and is the relative location on the current page before which a new page should be started. The ORG statement includes an AND mask of 7600 to preserve the current page. When added to 200 for the next page and the section name, the new origin is set.

When calculating directly in a module, the following rules apply to relative and absolute values.

relative − relative = absolute
absolute + relative = relative
OR (!), AND (&) and ADD (+) of relative symbols
generate the RALF error message RE.

When passing arguments (single precision) from FPP code to PDP code, using the index registers is very efficient. For example,

```
.
.
.
FLDA%   ARG1    /Get argument in FPP mode
SETX    MODE8   /Change index registers so XRO is
                /At MODE8
ATX     MODE8   /Save argument
.
.
.
TRAP4   SUB8    /Go to PDP-8 routine
```

```
SUB8,   0                       /PDP-8 routine
        .
        .
        .
        TAD     MODE8   /Get argument
        .
        .
        .
MODE8,  0                       /Index registers set here
        .
        .
        .
```

The source of FORTRAN Library is the best collection available of useful coding techniques in RALF. Working examples include subroutine linkage, 8-mode trap sequences, background task inclusion, interrupt handling, laboratory peripheral interfacing,and mathematical calculation.

Using The Assembler

FLAP/RALF is run as a standard OS/8 program by typing:

```
.R FLAP (or RALF)
*binary,listing←input1,input2,.....
```

where binary is the binary output file, default extension .RL; listing is the listing output file, default extension .LS; input1, input2, etc. are up to 9 source input files, default extensions .RA . The source files must contain only one FLAP/RALF source module (i.e., one END statement).

All error messages are printed on the terminal during pass 2, without affecting the binary output file, along with the line which caused the error. This output may be inhibited by typing CTRL/O. The error messages are also printed above the error line on the listing. FLAP/RALF error codes are listed in the next section.

Assembly may be aborted by typing CTRL/C. Each page of a FLAP/RALF listing has a one line header in the form:

FLAP (or RALF) V nn mo da, yr PAGE r

where nn is the assembler version number, mo da, yr is the date, and r is the page number.

The /S option, in FLAP, may be used to suppress the listing file and generate only the symbol map on pass 3. If no listing file is specified, this option is ignored. The /T option performs the same function in RALF.

Error Messages

During pass 2, error messages are printed at the terminal as they occur, followed by the statement in which the error occurred.

During pass 3, error codes are printed in the listing immediately preceding the line in which the error occurred, except the EG message, which is printed after the line. If the line of code includes statements terminated by a semicolon, then the error message for a statement precedes the printing of its octal value on the next line.

A fatal error causes an immediate return to the OS/8 monitor after the message is printed. The following table lists the error codes and their meanings.

Table 5-2 FLAP/RALF Error Codes

Error Code	Meaning
BE	Illegal equate. The symbol had been defined previously.
BI	Illegal index register specification.
BX	Bad expression. Something in the expression is incorrect or the expression is not valid in this context.
DV	An attempt was made in an expression evaluation to divide by zero.
EG	The preceding line contains extra code which could not be used by the assembler.

Table 5-2 FLAP/RALF Error Codes (Cont.)

Error Code	Meaning
ES	External symbol error. (RALF only)
FL	An error has occurred in the FPP or software floating conversion routines. This could be due to an attempt to convert an excessively large or small number, or an internal error in the assembler occurred.
FP	A syntax error was encountered in a floating point or extended precision constant.
IC	The symbol or expression in a conditional is improperly used, or left angle bracket is missing. The conditional pseudo-op is ignored.
IE	An entry point has not been defined, or is absolute, or is also defined as a common, section, or external. (RALF only)
IL	A literal was used in an instruction which cannot accept one. (FLAP only)
IO	Input/output error (fatal error).
IR	Invalid reference in a PDP-8 instruction.
IX	An index register was specified for an instruction which cannot accept one.
LT	The line is longer than 127 characters. The first 127 characters are assembled and listed.
MD	The tag on the line has been previously encountered at another location or has been used in a context requiring an absolute expression.
NE	Number error. A number out of range was specified or an 8 or 9 occurred in octal radix.
PO	Page overflow. Literals and instructions have been overlapped. (FLAP only)
RE	Relocatability error. A relocatable expression has been used in context requiring an absolute expression. (RALF only)

Table 5-2 FLAP/RALF Error Codes (Cont.)

Error Code	Meaning
ST	User symbol table overflow (fatal error).
US	Undefined symbol in an expression.
XS	External symbol table overflow. Control returns to the OS/8 Keyboard Monitor. (RALF only)

FLAP/RALF Pseudo-Operators

The following table lists the FLAP/RALF pseudo-ops and gives a brief description of each pseudo-op.

Table 5-3 FLAP/RALF Pseudo-Operators

Pseudo-op	Meaning
ADDR	Place the 15-bit address of the symbol into two words of core at the current position of the location counter.
BASE expr	Assign base register for 1-word instructions.
COMMON name	Causes the assembler to enter the common section whose name follows the pseudo-op.
COMMZ name	Define name as a special common section restricted to load into page 0 of field 1.
DECIMAL	Set radix for integer conversion to decimal.
E xxx	Generate 6-word extended precision floating point constant.
END	End of input.
ENPUNC	Re-enable binary output (FLAP only).
ENTRY name	Insert name into the ESD as an entry point. The symbol name must be defined as a relocatable symbol in the current assembly.
EXTERN name	Insert name into the ESD as an external reference. The symbol name must not be defined in the current assembly.
F xxx	Generate 3-word floating point constant.
FIELD1 name	Similar to SECT8, but this section is restricted to load into field 1 only.

Table 5-3 FLAP/RALF Pseudo-Operators (Cont.)

Pseudo-op	Meaning
IFFLAP	Assemble if the assembler is FLAP.
IFNDEF n	Assemble if n is not defined.
IFNEG n	Assemble if n is negative.
IFNSW n	Assemble if switch n was not set in Command Decoder input.
IFNZRO n	Assemble if n is not zero.
IFPOS n	Assemble if n is positive.
IFRALF	Assemble if the assembler is RALF.
IFREF symbol	Assemble if symbol has already been defined or referenced.
IFSW n	Assemble if symbol was set in Command Decoder input.
IFZERO n	Assemble if n is zero.
INDEX n	Assign index register location.
LISTOF	Inhibit program listing.
OCTAL	Set radix for integer conversion to octal.
ORG expr	Set current location counter to lower 15 bits of expr.
PAGE	Set current location counter to the beginning of next core page (FLAP only).
REPEAT n	Repeat next line n times.
S xxx	Generate 1-word constant (FLAP only).
SECT name	Define name as a section and begin that section. Subsequent SECT name commands will resume the section wherever it left off.
SECT8	Similar to SECT, but this section is restricfed to load in level MAIN, on a 200_8 word boundary. SECT8 is used to define sections that contain PDP-8 mode code.
TEXT	Assemble the text between delimiters as packed 6-bit ASCII characters.
ZBLOCK n	Assemble n words containing 0.
=	Equate symbol on left of = to value of expression on right.

os/8 languages

basic
fortran ii
fortran iv

chapter 6

basic

INTRODUCTION TO OS/8 BASIC

BASIC[1] is an interactive programming language with a variety of applications. It is used in scientific and business environments to solve both simple and complex mathematical problems with a minimum of programming effort. It is used by educators and students as a problem-solving tool and as an aid to learning through programmed instruction and simulation.

In many respects the BASIC language is similar to other programming languages (such as FOCAL and FORTRAN), but BASIC is aimed at facilitating communication between the user and the computer. The BASIC user types in the computational procedure as a series of numbered statements, making use of common English words and familiar mathematical notations. Because of the small number of commands necessary and its easy application in solving problems, BASIC is one of the simplest computer languages to learn. With experience, the user can add the advanced techniques available in the language to perform more intricate manipulations or express a problem more efficiently and concisely.

OS/8 BASIC has a greater capability than 8K BASIC and contains such added features as chaining, string manipulation, and file-oriented input/output.

Running BASIC

Once the Keyboard Monitor has responded with a period to indicate that it is ready to receive a monitor command, the user types the following command:

```
.R BASIC
```

BASIC responds with the following:

[1] BASIC is a registered trademark of the trustees of Dartmouth College.

NEW OR OLD—

The user types in:

NEW FILE.EX

if the user is going to create a new program, where FILE.EX is the name and extension of the new program. If the user wants to work with a previously created program that he saved on a storage device, he types in the following:

OLD DEV:FILE.EX

where DEV: is the name of the OS/8 device on which his old file is stored.
For example:

OLD DTA6:SAMPLE.BA

This response to NEW or OLD—retrieves the file named SAMPLE from DECtape 6 and replaces the current contents of user core with the file SAMPLE. If the user specifies a device that does not exist or that is not available for use, BASIC returns an error message.

ENTERING THE NEW PROGRAM

After the user types in his filename, OS/8 BASIC responds with the following:

READY

The user can begin to type in his new program or make changes to his old program. When the new program is being typed, the user must make sure that each line begins with a line number containing no more than five digits and containing no spaces or non-digit characters. The RETURN key must be pressed at the completion of each line. If, in the process of typing a statement, the user makes a typing error and notices it before he terminates the line, he can correct it by pressing the RUBOUT key or SHIFT/O keys once for each character to be erased, going backward until the character in error is reached. Then he may continue typing, beginning with the character in error. Using the RUBOUT key or SHIFT/O keys echoes a backarrow (←) for each character deleted. The following is an example of this correcting process (note that a ← is typed for spaces as well as characters):

```
20 DEN F←←←F FNA(X,Y)=↑2+3*Y
```

The corrected version of the above example would appear on a subsequent listing of the program as:

```
20 DEF FNA(X,Y)=X↑2+3*Y
```

Program listings can be generated using the LIST or LISTNH commands.

EXECUTING THE PROGRAM

After typing the complete program (do not forget to end with an END statement), type RUN or RUNNH, followed by the RETURN key. OS/8 BASIC prints the name of the program, the version of OS/8 BASIC, the current date (unless RUNNH is specified), and then it analyzes the program. If the program can be run, OS/8 BASIC executes it and, via PRINT statements, prints out any results that were requested. The printout of results does not guarantee that the program is correct (the results could be wrong), but it does indicate that no syntactical errors exist (e.g., missing line numbers, misspelled words, or illegal syntax). If errors of this type do exist, OS/8 BASIC prints a message (or several messages) to the user. A list of these diagnostic messages, with their meanings, is given at the end of the chapter.

NOTE

RUN and RUNNH are control commands, and like all other OS/8 BASIC edit and control commands, they do not require a line number preceding the command.

CORRECTING THE PROGRAM

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

```
110 (followed by a carriage return)
```

If he wishes to insert a statement between lines 60 and 70, the user types a iine number between 60 and 70 (e.g., 65), followed by the statement.

INTERRUPTING EXECUTION OF THE PROGRAM

If the results being printed seem to be incorrect and the user wants to stop execution of his program, the user types CTRL/C which is echoed by ↑C. The OS/8 BASIC editor responds with the READY message whereupon the user can modify or add statements and rerun his program.

LEAVING THE COMPUTER

When the user's program is finished and he no longer requires the use of OS/8 BASIC, he types the BYE command (or CTRL/C) to return control to the Keyboard Monitor.

EXAMPLE OF OS/8 BASIC RUN

The following is a simple example of the use of OS/8 BASIC.

```
READY
SCRATCH

READY

.R BASIC                          Instruct monitor to bring
NEW OR OLD--NEW SAMPLE.BA         BASIC into core and start
                                  its execution
READY
10 FOR N=1 TO 7                   BASIC asks whether new
20 PRINT N,SQR(N)                 or old program is to be run
30 NEXT N
40 PRINT "DONE"
50 END                            BASIC is now ready to
RUN                               receive statements

SAMPLE  BA     3.0

                                  Type in statements
 1             1
 2             1.41421
 3             1.73205
 4             2                  Run program
 5             2.23607
 6             2.44949
 7             2.64575            Program heading and re-
DONE                              sults of program are printed

READY
```

GENERAL SYSTEM DESCRIPTION

The OS/8 BASIC system is divided into five discrete parts:

1. Editor
2. Compiler
3. Loader
4. Runtime System
5. Runtime System Overlays

The OS/8 BASIC Editor is used to create and edit the source program. On receipt of a RUN command, the Editor stores the program in a temporary file and chains to the Compiler. The Compiler compiles the program into pseudo-instructions which are then loaded into core with the Run-time System by the Loader. The Run-time System interprets each pseudo-instruction, calling each of the Overlays into core as needed. On completion of the program, the Run-time System chains back to the Editor.

OS/8 BASIC STATEMENTS AND COMMANDS

OS/8 BASIC consists of program statements and system control commands which are needed to write programs. A number of the elementary OS/8 BASIC statements and commands are:

OS/8 BASIC Statements	
LET	Assign a value to a variable.
PRINT	Print out the indicated information.
READ	Initialize variables to values from the data list.
DATA	Provide initial data for a program.
GOTO	Change order of program execution.
IF GOTO IF THEN	Conditionally change order of program execution.
FOR TO STEP	Set up a program loop.
NEXT	End a program loop.
GOSUB	Go to a subroutine.
RETURN	Return from a subroutine.
INPUT	Get initial values from the terminal.
REM	Insert a program comment.
RESTORE	Restore the data list.

DEF	Define a function.
STOP	Stop program execution.
END	End a program.
DIM	Define subscripted variables.
UDEF	Define user-coded function.

OS/8 BASIC Edit and Control Commands

LIST	List all stored program statements.
RUN	Run the currently stored program.
SCRATCH	Delete the currently stored program.
SAVE	Save the currently stored program.
OLD	Retrieve the old program.
NEW	Prepare for a new program.
NAME	Rename the currently stored program.
BYE	Exit from BASIC.

These statements and commands are explained in detail with actual computer output in this manual. For the convenience of the user, a detailed OS/8 BASIC Statement, Command and Function Summary is included at the end of the chapter.

The experienced BASIC programmer may elect to skip the first six sections of this chapter since they are rather fundamental. However, he should familiarize himself with the remaining sections as they provide information specifically related to OS/8 BASIC.

OS/8 BASIC ARITHMETIC

Numbers

An OS/8 BASIC number may be any number in the range of $10^{-616}<N<10^{616}$. OS/8 BASIC treats all numbers as decimal numbers; that is, it accepts any number containing a decimal, and assumes a decimal point after an integer. The advantage of treating all numbers as decimal numbers is that the programmer can use any number or symbol in any mathematical expression without regard to its type.

In addition to integer and decimal formats, a third format is recognized and accepted by OS/8 BASIC and is used to express numbers outside the range $.000001 \leqslant \times \leqslant 999999$. This format is

called exponential or E-type notation and in this format, a number is expressed as a decimal number times some power of 10. The form is:

xxEn

where E represents "times 10 to the power of," thus the number is read: "xx times 10 to the power of n." For example:

$23.4E2=23.4*10^2=2340$

Data may be input in any one or all three of these forms. Results of computations are output as decimals if they are within the range previously stated; otherwise, they are output in E format. OS/8 BASIC handles six significant digits in normal operation and input/output, as illustrated below:

Value Typed in	Value Output By OS/8 BASIC
.01	0.0099999
.0099	0.0099
999999	999999
1000000	.100000E+007
.0000009	.899999E−006

OS/8 BASIC automatically suppresses the printing of leading and trailing zeros in integer numbers and all but one leading zero in decimal numbers. As can be seen from the preceding examples, OS/8 BASIC formats all exponential numbers in the form:

sign .xxxxxxE(+or−)n

where x represents the number carried to six decimal places, E stands for "times 10 to the power of," and n represents the exponential value.
For example:

−.347021E+009 is equal to −347,021,000
.726000E−003 is equal to 0.000726

Variables

A simple variable in OS/8 BASIC is an algebraic symbol representing a number, and is formed by a single letter or a letter followed by a digit. For example:

Acceptable Variables	Unacceptable Variables
I	2C—a digit cannot begin a variable
B3	AB—two or more letters cannot form a variable
X	

The user may assign values to variables either by indicating the values in a LET statement, or by inputting the values as data.

```
10 LET I=53721
20 LET B3=456.9
30 LET X=20E9
40 INPUT Q
```

These operations, as well as subscripted variables, are discussed in detail in the section entitled LISTS AND TABLES. A discussion of subscripted and unsubscripted string variables is provided in the section entitled ALPHANUMERIC INFORMATION.

Arithmetic Operations

OS/8 BASIC performs addition, subtraction, multiplication, division and exponentiation, as well as more complicated operations explained in detail later in the manual. The five operators used in writing most formulas are:

Symbol Operator	Meaning	Example
+	Addition	A + B
−	Subtraction	A − B
*	Multiplication	A * B
/	Division	A / B
↑ (or **)	Exponentiation (Raise A to the B Power)	A ↑ B or (A**B)

PRIORITY OF ARITHMETIC OPERATIONS

In any given mathematical formula, OS/8 BASIC performs the arithmetic operations in the following order:

1. Parentheses receive 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 (of equal priority)
   c. Addition and Subtraction (of equal priority)
3. If either 1 or 2 above does not clearly designate the order of priority, then the evaluation of expressions proceeds from left to right.

The expression A↑B↑C is evaluated from left to right as follows:

1. A↑B = step 1
2. (result of step 1)↑C = answer

The expression A/B*C is also evaluated from left to right since multiplication and division are of equal priority:

1. A/B = step 1
2. (result of step 1)*C = answer

PARENTHESES

Parentheses may be used by the programmer to change the order or priority (as listed in rule 2 above), because expressions within parentheses are always evaluated first. Thus, by enclosing expressions appropriately, the programmer can control the order of evaluation. Parentheses may be nested, or enclosed by a second set (or more) of parentheses. In this case, the expression within the innermost parentheses is evaluated first, and then the next innermost, and so on, until all have been evaluated.
Consider the following example:

```
A=7*((B↑2+4)/X)
```

The order of priority is:

1. B↑2 = step 1
2. (result of step 1)+4 = step 2

3. (result of step 2)/X = step 3
4. (result of step 3)*7 = A

Parentheses also prevent any confusion or doubt as to how the expression is evaluated. For example:

```
A*B↑2/7+B/C+D↑2
((A*B↑2)/7)+((B/C)+D↑2)
```

Both of these formulas will be executed in the same way. However, the inexperienced programmer or student may find that the second is easier to understand. Spaces may also be used to increase readability. Since the OS/8 BASIC compiler ignores spaces, the two statements:

```
10 LET B = D↑2 + 1
10 LETB=D↑2+1
```

are identical, but spaces in the first statement provide ease in reading.

RELATIONAL OPERATORS

A program may require that two values be compared at some point to discover their relation to one another. To accomplish this, OS/8 BASIC makes use of the following relational operators:

= equal to	> greater than
< less than	=> or >= greater than or equal to
=< or <= less than or equal to	> < or < > not equal to

Depending upon the result of the comparison, control of program execution may be directed to another part of the program. Relational operators are used in conjunction with the IF-THEN statement which is discussed in the next section.

The meaning of the (=) sign should be clarified. In algebraic notation, the formula X=X+1 is meaningless. However, in OS/8 BASIC (and most computer languages), the equal sign designates replacement rather than equality. Thus, this formula is actually translated "add one to the current value of X and store the new result back in the same variable X." Whatever value has previously been assigned to X will be combined with the value 1. An expression such as A=B+C instructs the computer to add the

values of B and C and store the result in a third variable A. The variable A is not being evaluated in terms of any previously assigned value, but only in terms of B and C. Therefore, if A has been assigned any value prior to its use in this statement, the old value is lost; it is instead replaced by the value of B+C.

RULES FOR EXPONENTIATION

The following rules apply in evaluating the expression A↑B.

Rule	Example
1. If B=0, then A↑B=1	3↑0=1
2. If A=0 and B>0, then A↑B=0	0↑2=0
3. If A=0 and B<0, then A↑B=0 and a DV error message is printed (See Appendix C)	0↑−2=0
4. If B is an integer >9, then A↑B=A_1*A_2*A_3 . . . *A_n, where n=B	3↑5=3*3*3*3*3=243
5. If B is an integer <0 then A↑B=1/(A_1*A_2*A_3 . . .*A_n), where n=B	3↑−5=1/243
6. If B is a decimal (non-integer) and A>0, then A↑B= EXP(B*LOG(A))	$2\uparrow 3.6=e^{B\ln A}=e^{3.61n2}$
7. If B is a positive or negative decimal (non-integer) and A<0, program halts due to fatal error	−3↑2.6 is illegal. Fatal error message EM printed.

OS/8 BASIC STATEMENTS

The following Example Program is included at this point as an illustration of the format of an OS/8 BASIC program, the ease in running it, and the type of output that may be produced. This program and its results are for the most part self-explanatory. Following sections cover the program statements and system commands used in OS/8 BASIC programming.

```
10  REM - PROGRAM TO TAKE AVERAGE OF
15  REM - STUDENT GRADES AND CLASS GRADES
20  PRINT "HOW MANY STUDENTS, HOW MANY GRADES PER STUDENT ";
30  INPUT A,B
40  LET I=0
50  FOR J=I TO A-1
55  LET V=0
60  PRINT "STUDENT NUMBER = ";J
75  PRINT "ENTER GRADES"
76  LET D=J
80  FOR K=D TO D+(B-1)
81  INPUT G
82  LET V=V+G
85  NEXT K
90  LET V=V/B
95  PRINT "AVERAGE 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
RUNNH
HOW MANY STUDENTS, HOW MANY GRADES PER STUDENT ?5,4
STUDENT NUMBER =  0
ENTER GRADES
?78
?86
?88
?74
AVERAGE GRADE - 81.5

STUDENT NUMBER =  1
ENTER GRADES
?59
?86
?70
?87
AVERAGE GRADE - 75.5

STUDENT NUMBER =  2
ENTER GRADES
?58
?64
?75
?80
AVERAGE GRADE - 69.25

STUDENT NUMBER =  3
ENTER GRADES
?88
?92
?85
?79
AVERAGE GRADE - 86
```

```
STUDENT NUMBER =  4
ENTER GRADES
?60
?78
?85
?80
AVERAGE GRADE - 75.75


CLASS AVERAGE = 77.6

READY
```

Statement Numbers

A program is made up of statements. Each line of the program begins with a line number of 1 to 5 digits that serves to identify the line as a statement. The largest allowable line number is 99999. Line numbers serve to specify the order in which these statements are to be performed. Before the program is run, OS/8 BASIC sorts out and edits the program, putting the statements into the order specified by their line numbers; thus, the program statements can be typed in any order, as long as each statement is prefixed with a line number indicating its proper sequence in the order of execution. Each statement starts after its line number with an English word (except the LET statement where LET is optional) which denotes the type of statement. Unlike program statements, system commands are not preceded by line numbers and are executed immediately after they are typed in. Spaces have no significance in BASIC programs or commands, except in messages or literal strings which are printed out, and in line numbers. Thus, spaces may, but need not be, used to modify a program and make it more readable.

A common programming practice is to number lines by fives or tens, so that additional lines may be inserted in a program without the necessity of renumbering lines already present. Renumbering a program can be accomplished by using the RESEQ program described in the section entitled EDITING AND CONTROL COMMANDS.

Multiple statements may be placed on a single line by separating each statement from the preceding statement with a backslash (SHIFT/L on some terminals). For example:

```
10 A=5\B=.2\C=3\PRINT "ENTER DATA"
```

All of the statements in line 10 will be executed before BASIC continues to the next line. Only one statement number at the beginning of the entire line is necessary. However, it should be remembered that program control cannot be transferred to a statement within a line, only to the first statement of the line in which it is contained.

REMARK—The Commenting Statement

The REM or REMARK statement allows the programmer to insert comments or remarks into a program without these comments affecting execution. The OS/8 BASIC compiler ignores everything between REM and the end of the line. The form is:

(line number) REM (message)

In the Example Program, lines 10 and 15 are REMARK statements describing what the program does. It is often useful to put the name of the program and information relating to its use at the beginning where it is available for future reference. Remarks throughout the body of a long program will help subsequent debugging by explaining the purpose of each statement within the program.

Statements For Terminating A Program

END

The END statement (line 140 in the Example Program) should be the last statement of the entire program. The form is:

(line number) END

STOP

The STOP statement is used synonymously with the END statement to terminate execution; but while END occurs only once at the end of a program, STOP may occur any number of times. The format of the STOP statement is:

(line number) STOP

This statement signals that execution is to be terminated at that point in the program where it is encountered.

LET—The Assignment Statement

The Assignment (LET) statement is probably the most commonly used OS/8 BASIC statement and is used whenever a value is to be assigned to a variable. It is of the form:

(line number) LET x = expression

where x represents a variable, and the expression is either a number, another variable, or an arithmetic expression. The word "LET" is optional; thus the following statements are treated the same:

```
100 LET A=A↑B+10      110 LET L=L+1
100 A=A↑B+10          110 L=L+1
```

The LET statement is not strictly an equality. LET means "evaluate the expression to the right of the equal sign and assign this value to the variable on the left." Thus, the statement L=L+1 means "set L equal to a value one greater than it was before."

Input/Output Statements and Functions

Input/output statements allow the user to bring data into a program and output results or data at any time during execution.

THE INPUT STATEMENT

The INPUT statement is used when data is to be supplied by the user from the terminal keyboard while a program is executing and is of the form:

(line number) INPUT x1, x2, . . . , xn

where x1 through xn represent variable names. For example:

25 INPUT A

This statement will cause the program to pause during execution, print a question mark on the terminal console, and wait for the user to type in a numerical value.
The following rules apply to the use of the INPUT statement.

Rule

1. The following characters are recognized as acceptable when inputting numeric data:

   + or − sign
   digits 0 through 9

the letter E
leading spaces (ignored)
. (first decimal point)

All other characters are treated as delimiters for separating numeric data.

```
10 INPUT A,B,C,D,E
•
•
•
RUNNH
?10,32A16,8 1

READY
```

In the above example, A=10, B=32, C=16, D=8, and E=1.

2. When inputting numeric data, two delimiters read in succession imply that the data between the delimiters is 0.

```
10 INPUT A,B,C,D,E
•
•
•

RUNNH
?5,10,,12,15

READY
```

In the above example A=5, B=10, C=0, D=12, and E=15.

3. In response to an INPUT statement the user can provide more data than is requested by the INPUT statement. The remaining or unused data is saved for subsequent use by the next INPUT statement. The question mark (?) is not printed until the program is out of data.
4. When inputting string data, all characters are recognized as part of the string. See ALPHANUMERIC INFORMATION (Strings) for further information relating to strings.

THE PRINT STATEMENT

General

The PRINT statement is used to output results of computations,

comments, values of variables, or plot points of a graph on a terminal. The format is:

(line number) PRINT expression

When used without an expression, a blank line will be output on the terminal. For more complicated uses, the type of expression and the type of format control characters (comma or semicolon) following the word PRINT determines which formats will be created.

In order to have the computer print out the results of a computation, or the value of a variable at any point in the program, the user types the line number, PRINT, and the variable name(s) separated by a format control character, in this case, commas:

```
5 A=16   B=5   C=4
10 PRINT A, C+B, SQR(A)
15 END
```

The PRINT statement may also be used to output a message or line of text. The desired message is simply placed in quotation marks in the PRINT statement as follows:

```
10 PRINT "THIS IS A TEST"
```

When line 10 is encountered during execution, the following will be printed:

```
THIS IS A TEST
```

A message may be combined with the result of a calculation or a variable as follows:

```
80 PRINT "AMOUNT PER PAYMENT=";R
```

Assuming R=344.961 when line 80 is encountered during execution, this will be output as:

```
AMOUNT PER PAYMENT = 344.961
```

The PRINT statement can also cause a constant to be printed on the console. For example:

```
10 PRINT 1.234, SQR(10014)
```

will cause the following to be output at execution time:

```
1.234        100.07
```

Any algebraic expression in a PRINT statement will be evaluated using the current value of the variables. Numbers will be printed according to the format specified in the section entitled PRINTING NUMBERS.

Format Control Characters

In OS/8 BASIC, a terminal line is formatted into five fixed zones (called print zones) of 14 columns each. A program such as:

```
5 A=2.3\B=21\C=156.75\D=1.134\E=23.4
10 PRINT A,B,C,D,E
15 END
```

where the control character comma (,) is used to separate the variables in the PRINT statement, will cause the values of the variables to be printed using all five zones.

```
RUNNH
 2.3            21             156.75         1.134          23.4

READY
```

It is not necessary to use the standard five zone format for output. The control character semicolon (;) causes the text or data to be output immediately after the last character printed.

The following example program illustrates the use of the control characters in PRINT statements.

```
5  READ A,B,C
10 PRINT A,B,C,A↑2,B↑2,C↑2
15 PRINT
20 PRINT A;B;C;A↑2;B↑2;C↑2
25 DATA 4,5,6
30 END
RUNNH
 4              5              6              16             25
 36

 4  5  6  16  25  36

READY
```

As this example illustrates, when more than five variables are listed in the PRINT statement, OS/8 BASIC automatically moves the sixth number to the beginning of the next line.

Printing Numbers

For any format (integer, decimal, or E-type) OS/8 BASIC prints numbers in the form:

sign number space

where sign is either minus (−) or blank (for plus) and a blank space always trails the number.

```
10 A=64\B=-32\C=72
20 PRINT A;B;C
21 END

READY
RUNNH
 64 -32  72

READY
```

PRINT Used With INPUT

Another use of the PRINT statement is to combine it with an INPUT statement so as to identify the data expected to be entered. As an example, consider the following program:

```
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
60 LET N=N*M
65 LET I=J/M
70 LET B=1+I
75 LET R=A*I/(1-1/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 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
200 END

READY
RUNNH
INTEREST IN PERCENT?9
AMOUNT OF LOAN?2500
NUMBER OF YEARS?2
NUMBER OF PAYMENTS PER YEAR?4

AMOUNT PER PAYMENT =  344.965
TOTAL INTEREST     =  259.724

 INTEREST       APP TO PRIN    BALANCE
 56.25          288.715        2211.28
 49.7539        295.212        1916.07
 43.1116        301.854        1614.22
 36.3199        308.645        1305.57
 29.3754        315.59         989.982
 22.2746        322.691        667.291
 15.0141        329.951        337.34
 7.59015        337.375
LAST PAYMENT   344.93

READY
```

As can be noticed in this example, the question mark is grammatically useful in a program in which several values are to be input by allowing the programmer to formulate a verbal question which the input value will answer.

THE TAB(X) FUNCTION

The TAB function, which may only be used in a PRINT statement, allows the user to position the printing of characters anywhere on the terminal line (or other printing device line when used with PRINT#). Print positions can be thought of as being numbered from 1 to 72 across the Teletype from left to right. The form of this function is:

```
TAB(X)
```

where the argument X represents the position (from 1 to 72 columns available on the terminal) in which the next character will be typed.

Each time the TAB function is used in a PRINT statement, positions are counted from the beginning of the line, not from the current position of the printing head. For example, the TAB function in the following program causes the character "/" to be printed at 24 equally spaced positions along the line.

```
10 FOR K=3 TO 72 STEP 3
20 PRINT TAB(K); "/";
30 NEXT K
40 END
```

If the argument X in the TAB function is less than the current position of the printing head, printing is started at the current position. If the argument X is greater than 72 (the number of columns available in an output line), a carriage return-line feed is executed and printing resumes at position 1.

THE PNT(X) FUNCTION

OS/8 BASIC provides an additional function, PNT(X), to increase input/output flexibility. The function is primarily used for outputting non-printing characters such as the "bell", but can be used for more sophisticated applications. The PNT(X) function, like the TAB(X) function, may only be used in either a PRINT or PRINT# statement. The form of the function is:

```
PNT(X)
```

where the argument X represents the decimal value of the 7-bit ASCII character to be output. For example, the statement:

```
10 PRINT "X=";3.14159;PNT(13);TAB(14);"/"
```

will print the slash (/) on top of the equal sign after executing a carriage return (CR=13_{10}) and a TAB to column 2 as shown below:

```
X=␣3.14159␣
```

Notice that a TAB(14) is required since OS/8 BASIC remembers the print head to be at column 12 after the carriage return (11 columns for X=␣3.14159␣ and 1 column for the PNT function). A tab to column 2 after the carriage return provides a total of 14 columns. The PNT(13) carriage return does not zero the column count but, in fact, adds to the column count. (This example may not work on some terminals.)

The READ and DATA Statements

READ and DATA statements are used to provide data to a program. One statement is never used without the other. The form of the READ statement is:

(line number) READ x1,x2 , . . . ,xn

where x1 through xn represent variable names. For example:

```
10 READ A,B,C
```

A, B, and C are variables to which values will be assigned. Variables in a READ statement must be separated by commas. READ statements are generally placed at the beginning of a program, but must at least logically occur before that point in the program where the value is required for some computation.

Values which will be assigned to the variables in a READ statement are supplied in a DATA statement of the form:

(line number) DATA x1,x2 . . . ,xn

where x1 through xn represent values. The values must be separated by commas and occur in the same order as the variables which are listed in the corresponding READ statement. A DATA statement appropriate for the preceding READ statement is:

```
70 DATA 1,2,3
```

Thus, at execution time A=1, B=2, and C=3.

The DATA statement is usually placed at the end of a program (before the END statement) where it is easily accessible to the programmer should he wish to change the values.

A READ statement may have more or fewer variables than there are values in any one DATA statement. The READ statement causes OS/8 BASIC to search all available DATA statements in the order of their line numbers until values are found for each variable in the READ. A second READ statement will begin reading values where the first stopped. If at some point in the program an attempt is made to read data which is not present, OS/8 BASIC will stop and print the following message at the console:

```
DA AT LINE YYYYY
```

where YYYYY indicates the line which caused the error.

RESTORE

If it is desired to use the same data more than once in a program, the RESTORE statement will make it possible to recycle through the DATA list beginning with the first DATA statement. The RESTORE statement is of the form:

```
(line number) RESTORE
```

An example of its use follows:

```
15 READ B,C,D
.
.
.
55 RESTORE
60 READ E,F,G
.
.
.
80 DATA 6,3,4,7,9,2
.
.
.
100 END
```

The READ statements in lines 15 and 60 will both read the first three data values provided in line 80. (If the RESTORE statement had not been inserted before line 60, then the second READ would pick up data in line 80 starting with the fourth value.)

The programmer may use the same variable names the second time through the data or not, as he chooses, since the values are being read as though for the first time. In order to skip unwanted values, the programmer may insert replacement, or dummy variables. Consider:

```
1 REM - PROGRAM TO ILLUSTRATE USE OF RESTORE
20 READ N
25 PRINT "VALUES OF X ARE:"
30 FOR I=1 TO N
40 READ X
50 PRINT X,
60 NEXT I
70 RESTORE
80 READ M
185 PRINT
190 PRINT "SECOND LIST OF X VALUES"
200 PRINT "FOLLOWING RESTORE STATEMENT:"
210 FOR I=1 TO N
220 READ X
230 PRINT X,
240 NEXT I
250 DATA 4,1,2
251 DATA 3,4
300 END

READY
RUNNH
VALUES OF X ARE:
 1             2             3             4
SECOND LIST OF X VALUES
FOLLOWING RESTORE STATEMENT:
 1             2             3             4
READY
```

The second time the data values are read, the first X picks up the value originally assigned to N in line 20, and as a result, OS/8 BASIC prints:

```
1             2             3             4
```

To circumvent this, the programmer could insert a dummy variable which would pick up and store the first value, but would not be represented in the PRINT statement, in which case the output would be the same each time through the list.

Control Statements

Certain control statements cause the execution of a program to jump to a different line either unconditionally or depending upon some condition within the program. The following statements give the programmer capabilities in this area.

GOTO

The GOTO (or GO TO) statement is an unconditional statement used to direct program control either forward or backward in a program. The form of the GOTO statement is:

(line number) GOTO n

where n represents a statement number. When the logic of the program reaches the GOTO statement, the statement(s) immediately following will not be executed; instead execution is transferred to the statement beginning with the line number indicated.

The following program never ends; it does a READ, prints something, and jumps back to the READ via a GOTO statement. It 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 ENDING WITH ERROR
11 REM - MESSAGE WHEN OUT OF DATA
20 READ X
25 PRINT "X=";X, "X↑2=";X↑2
30 GO TO 20
35 DATA 1,5,10,15,20,25
40 END
```

```
READY
RUNNH
X=  1            X↑2=  1
X=  5            X↑2=  25
X=  10           X↑2=  100
X=  15           X↑2=  225
X=  20           X↑2=  400
X=  25           X↑2=  625

DA AT LINE 00020

READY
```

IF-THEN and IF-GOTO

If a program requires that two values be compared at some point, control of program execution may be directed to different procedures depending upon the result of the comparison. In computing, values are logically tested to see whether they are equal, greater than, less than another value, or possibly a combination of the three. This is accomplished by use of the relational operators.

IF-THEN and IF-GOTO statements allow the programmer to test the relationship between two variables, numbers, or expressions. Providing the relationship described in the IF statement is true at the point it is tested, control will transfer to the line numbers specified. If the relationship described in the IF statement is not true at the point it is tested, control will transfer to the line following the IF statement. The statements are of the form:

(line number) IF v1 <relation>v2 $\left\{\begin{matrix}\text{GOTO}\\ \text{or}\\ \text{THEN}\end{matrix}\right\}$ x

where v1 and v2 represent variable names, numbers, or expressions, and x represents a line number. The use of either THEN or GOTO is acceptable.

In the following example, the value of the variable A is changed or remains the same depending on A's relation to B.

```
100 IF A>B THEN 120
110 A=A↑ B-1
120 C=A/D
.
.
.
```

When using non-integer arithmetic in the IF-THEN statement, the test for zero may not always be appropriate due to the nature of the floating-point arithmetic used by the computer. To avoid this problem, the programmer should either avoid using non-integer arithmetic in the IF-THEN statement, or test for fractional values less than the tolerance desired and set the value to zero.

IF-THEN statements that test the running variable in FOR-NEXT loops (see the next section) are particularly sensitive to this problem. For example:

```
10 FOR A=-5 TO 5 STEP .1
20 IF A=0 THEN 50
30 NEXT A
40 STOP
50 PRINT "EQUAL TO ZERO"
60 END
```

The above program will never go to line 50.

LOOPS

Frequently programmers are interested in writing a program in which one or more portions are executed a number of times, usually with slight variations each time. To write the simplest program in which the portion of the program to be repeated is written just once, a loop is used. A loop is a block of instructions that the computer executes repeatedly until a specified terminal condition is met. BASIC provides two statements to specify a loop: FOR and NEXT.

FOR and NEXT Statements

The FOR statement is of the form:

(line number) FOR v=x1 TO x2 STEP x3

where v represents a variable name, and x1, x2, and x3 all represent expressions (a numerical value, variable name, or mathematical expression). v is termed the index, x1 the initial value, x2 the terminal value, and x3 the incremental value. For example:

```
15 FOR K=2 TO 20 STEP 2
```

This means that the loop will be repeated as long as K is less than or equal to 20. Each time through the loop, K is incremented by 2, so the loop will be repeated a total of 10 times.

A variable used as an index in a FOR statement must not be subscripted, although a common use of loops is to deal with subscripted variables, using the value of the index as the subscript of a previously defined variable (this is illustrated in the section concerning Subscripted Variables).
The NEXT statement is of the form:

(line number) NEXT v

and signals the end of the loop. When execution of the loop reaches the NEXT statement, the computer adds the STEP value to the index and checks to see if the index is less than or equal to the terminal value. If so, the loop is executed again. If the value of the index exceeds the terminal value, control falls through the loop to the statement following the NEXT statement, with the value of the index equaling the value it was assigned the final time through the loop.

If the STEP value is omitted, a value of +1 is assumed. Since +1 is the usual STEP value, that portion of the statement is frequently omitted. The STEP value may also be a negative number.
The following example illustrates the use of loops. This loop is executed 10 times: the value of I is 10 when control leaves the loop. +1 is the assumed STEP value.

```
READY
10 FOR I=1 TO 10
20 NEXT I
30 PRINT I
40 END
RUNNH
 10

READY
```

If line 10 had been:

```
10 FOR I=10 TO 1 STEP -1
```

the value printed by the computer would be 1.

As indicated earlier, the numbers used in the FOR statement are expressions; these expressions are evaluated upon first encountering the loop. While the index, initial, terminal, and STEP values may be changed within the loop, the value assigned to the initial expression remains as originally defined until the terminal condition is reached. To illustrate this point, consider the last example program. The value of I (in line 10) can be successfully changed as follows:

```
10 FOR I=1 TO 10
15 LET I=10
20 NEXT I
```

The loop will only be executed once since the value 10 has been reached by the variable I and the terminal condition is satisfied.

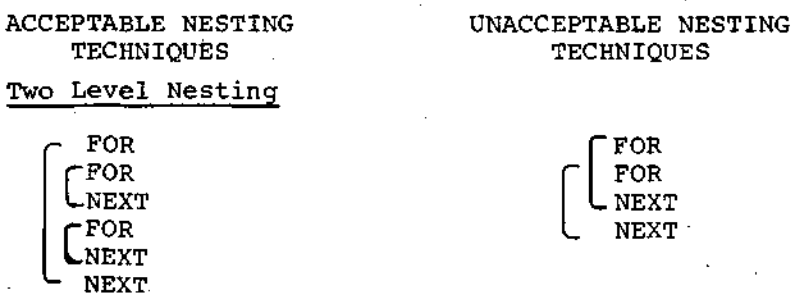
If the value of the counter variable is originally set equal to the terminal value, the loop will execute once, regardless of the STEP value. If the starting value is beyond the terminal value, the loop will never execute because an initial check is made of the starting and terminal values before the loop is executed. The following statement is executed but the loop it describes would never be executed:

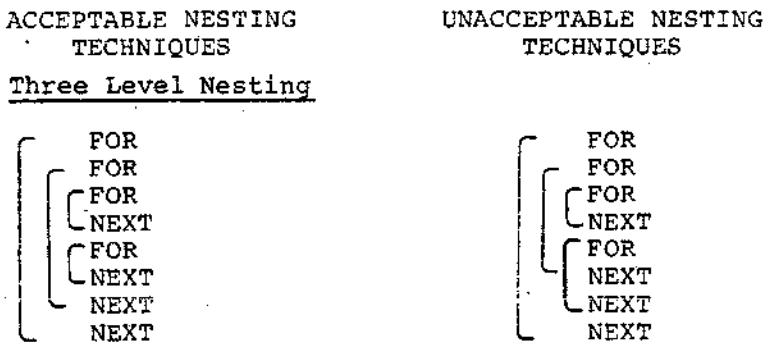
```
10 FOR I=10 TO 20 STEP -2
```

It is possible to exit from a FOR-NEXT loop without the index reaching the terminal value via an IF statement. Control may only transfer into a loop which has been left earlier without being completed, ensuring that the terminal and STEP values are assigned.

Nesting Loops

It is often useful to have one or more loops within a loop. This technique is called nesting, and 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:





LISTS AND TABLES

Subscripted Variables

In addition to single variable names, OS/8 BASIC accepts another class of variables called subscripted variables. Subscripted variables provide the programmer with additional computing capabilities for handling lists, tables, mattrices, or any set of related variables. Variables are allowed one or two subscripts. A single letter or a letter followed by a digit forms the name of the variable; this is followed by one or two integers in parentheses and separated by commas, indicating the place of that variable in the list. Up to 31 arrays are possible in any 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 follows:

A(1),A(2),A(3),A(4),A(5)

This allows the programmer to reference each of the five elements in the list A. A two dimensional matrix A(I,J) can be defined in a similar manner, but the subscripted variable A can only be used once (i.e., 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 an unsubscripted variable. Both A and A(I) are valid variable names and can be used in the same program.

Subscripted variables allow data to be input quickly and easily, as illustrated in the following program (the index of the FOR statement in lines 20, 42, and 44 is used as the subscript):

```
42 FOR I=1 TO 2
LIST

BAS16    BA     3.0      18-MAR-74

10 REM - PROGRAM DEMONSTRATING READING
11 REM - OF SUBSCRIPTED VARIABLES
15 DIM A(5),B(2,3)
16 PRINT "A(I) WHERE A=1 TO 5;"
20 FOR I=1 TO 5
25 READ A(I)
30 PRINT A(I);
35 NEXT I
38 PRINT
39 PRINT
40 PRINT "B(I,J) WHERE I=1 TO 2;"
41 PRINT "          AND J=1 TO 3;"
42 FOR I=1 TO 2
43 PRINT
44 FOR J=1 TO 3
48 READ B(I,J)
50 PRINT B(I,J);
55 NEXT J
56 NEXT I
60 DATA 1,2,3,4,5,6,7,8
61 DATA 8,7,6,5,4,3,2,1
65 END

READY
RUNNH
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;

 6   7   8
 8   7   6
READY
```

The DIM Statement

From the preceding example, it can be seen 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 indicates an array variable name and n and m are integer numbers indicating the largest subscript value required during the program. For example:

15 DIM A(6,10)

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:

$A_{0,0}\ A_{0,1} \ldots A_{0,10}$
$A_{1,0}\ A_{1,1} \ldots A_{1,10}$
$A_{2,0}\ A_{2,1} \ldots A_{2,10}$
. . .
. . .
. . .
$A_{6,0}\ A_{6,1} \ldots A_{6,10}$

and is illustrated in the following program:

```
10 REM - MATRIX CHECK PROGRAM
15 DIM A(6,10)
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

READY
RUNNH
 0  1  2  3  4  5  6  7  8  9  10
 1  0  0  0  0  0  0  0  0  0  0
 2  0  0  0  0  0  0  0  0  0  0
 3  0  0  0  0  0  0  0  0  0  0
 4  0  0  0  0  0  0  0  0  0  0
 5  0  0  0  0  0  0  0  0  0  0
 6  0  0  0  0  0  0  0  0  0  0

READY
```

Notice that a variable assumes a value of zero until another value has been assigned. If the user wishes to conserve core space by not making use of the extra variables set up within the array, he should set his DIM statement to one less than necessary, DIM A(5,9). This results in a 6 by 10 array which may then be referenced beginning with the A(0,0) element.

More than one array can be defined in a single DIM statement:

```
10 DIM A(20), B(4,7)
```

This dimensions 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 will result in an error message by BASIC at compile 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 TB error (too big) 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.

NOTE

If a subscripted variable is not defined by a DIM statement, the variable is assigned an array size of ten.

OS/8 BASIC FUNCTIONS AND SUBROUTINES

General Information On OS/8 BASIC Functions

OS/8 BASIC provides a number of functions, as part of the language, which perform calculations. The use of these functions eliminates the need for writing small programs to perform the calculations. Functions have a three letter call name, followed by an argument, X, which can be a number, variable, expression or another function. Generally, functions may be used anywhere a number or a variable is legal in a mathematical expression.
The following OS/8 BASIC functions are discussed in this chapter.

Function	Meaning
SIN(X)	Sine of X (X is expressed in radians)
COS(X)	Cosine of X (X is expressed in radians)
ATN(X)	Arctangent of X (result expressed in radians)
EXP(X)	e^x (e=2.718282)
LOG(X)	Natural log of X ($\log_e X$)
RND(X)	Random number
ABS(X)	Absolute value of X ($\|X\|$)
INT(X)	Integer value of X
SGN(X)	Sign of X — assign a value of +1 if X is positive, 0 if X is zero, or −1 if X is negative
SQR(X)	Square root of X (X)
FNA(X)	User-defined function
TRC(X)	Trace function — Used for debugging OS/8 BASIC programs.

In addition, there are a number of other functions provided by OS/8 BASIC, which include printing functions and string handling functions.

Function	
PNT(X)	Printing functions
TAB(X)	
LEN(X$)	String handling functions
ASC(X)	
CHR$(X)	
VAL(X)	
STR$(X)	
POS(X$,Y$,Z)	
SEG$(X$,Y,Z)	
DAT$(X)	

Arithmetic Functions

THE RANDOM NUMBER FUNCTION — RND(X)

The RND(X) function produces pseudo-random numbers between 0 and 1. The argument X is a dummy argument and can be any number.

If the user wants the first 20 random numbers, he can write the program shown below and get 20 six-digit decimals.

```
10 FOR L=1 TO 20
20 PRINT RND(X),
30 NEXT L
40 END

READY
RUNNH
 0.361572    0.332764    0.633057    0.350342    0.670166
 0.539795    0.8479      0.026123    0.54126     0.934326
 0.125244    0.389404    0.974853    0.516357    0.465088
 0.440186    0.970947    0.285889    0.867432    0.178467

READY
```

A second RUN gives exactly the same sequence of numbers as the first RUN; this is done to facilitate the debugging of programs.

If the user wants 20 random one-digit integers, he can change line 20 to read as follows:

```
20 PRINT INT(10*RND (X));
RUNNH
```

The results will be as follows:

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

READY
```

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

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

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

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

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

The RANDOMIZE Statement

As noted in the example program, the same numbers in the same order resulted both times the program was run. However, a different set will be produced with the RANDOMIZE statement, as in the following program:

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

READY
```

```
RUNNH
 0 7  0 0 2 7 7 3 2 5 7 0 0 3 0 1 6 0 6 7
READY
RUNNH
 1 4  6 5 7 6 0 6 2 2 9 4 7 1 2 1 1 3 0 6
READY
```

RANDOMIZE resets the numbers based on elapsed time spent waiting for terminal I/O. For example, if RANDOMIZE appears after a PRINT or INPUT instruction but before a statement with the RND (X) function, then repeated RUNs of the program produce different results. If the instruction is absent, then the official list of random numbers is obtained in the usual order. It is suggested that a simulated model should be debugged without this instruction so that one always obtains the same random numbers in test runs. After the program is debugged, and before starting production runs, the user inserts the following:

(line number) RANDOMIZE

at the appropriate place in the program.

THE SIGN FUNCTION — SGN (X)

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

25 LET X=SQR (A↑2+2*B*C) *SGN (A)

assigns the sign of X to the sign of A.

THE INTEGER FUNCTION — INT (X)

The integer function returns the value of the nearest integer not greater than X. For example, INT (34.67) = 34. By specifying INT (X+.5) the INT function can be used to round numbers to the nearest integer; thus, INT (34.67+.5) = 35. INT can also be used to round numbers to any given decimal place by specifying:

INT (X*10↑D+.5)/10↑D

where D is the number of decimal places desired. The following program illustrates this function; execution has been stopped by typing a CTRL/C:

```
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 =";B
80 GO TO 20
90 END

READY
RUNNH
NUMBER TO BE ROUNDED?55.65342
NO. OF DECIMAL PLACES:?2
A ROUNDED = 55.65
NUMBER TO BE ROUNDED?78.375
NO. OF DECIMAL PLACES:?-2
A ROUNDED = 100
NUMBER TO BE ROUNDED?67.89
NO. OF DECIMAL PLACES:?-1
A ROUNDED = 70
NUMBER TO BE ROUNDED?↑C
READY
```

If the argument is a negative number, the value returned is the largest negative integer (rounded to the higher value) contained in the number. For example, INT (−23) =−23 but INT (−14.39) =−15.

THE ABSOLUTE VALUE FUNCTION — ABS (X)

The absolute value function is used to obtain the absolute (positive) value of an expression. For example:

```
5 PRINT ABS(-66)
10 END

READY
RUNNH
 66

READY
```

THE SQUARE ROOT FUNCTION — SQR (X)

The square root function is used to compute the square root of an expression. For example:

```
5 LET B=4\A=2.5\C=.5
10 PRINT SQR(B↑2-4*A*C)
20 END
RUNNH
 3.31662

READY
```

If the argument of the SQR (X) function is <0, the absolute value of the argument is used.

Transcendental Functions

THE SINE FUNCTION — SIN (X)

The sine function is used to calculate the sine of an angle specified in radians. For example:

```
5  REM - CALCULATE SINE 30 DEGREES
10 LET P=3.14159
20 PRINT SIN(30*P/180)
25 END
RUNNH
 0.5

READY
```

THE COSINE FUNCTION—COS(X)

The cosine function is used to calculate the cosine of an angle specified in radians. For example:

```
5  REM - CALCULATE THE COSINE OF 45 DEGREES
10 PRINT COS(45*3.14159/180)
20 END
RUNNH
 0.707108

READY
```

THE ARCTAN FUNCTION—ATN(X)

This function calculates the angle (in radians) whose tangent is given as the argument of the function. For example:

```
5  REM - CALCULATE ATN(.57735)
10 PRINT ATN(.57735)
20 END
RUNNH
 0.523598

READY
```

THE EXPONENTIAL FUNCTION—EXP(X)

The EXP(X) function calculates the value of e raised to the X power, where e is equal to 2.71828. For example:

```
5 REM - CALCULATE EXPONENTIAL VALUE OF 1.5
10 PRINT EXP(1.5)
20 END
RUNNH
 4.48169

READY
```

THE NATURAL LOGARITHM FUNCTION—LOG(X)

The LOG(X) function calculates the natural logarithm of X. For example:

```
5 REM - CALCULATE THE LOG OF 959
10 PRINT LOG(959)
20 END
RUNNH
 6.86589

READY
```

User Defined Functions

THE FNA(X) FUNCTION AND THE DEF STATEMENT

In addition to the standard functions OS/8 BASIC provides, the user may define up to 26 functions of his own with the DEF statement. The name of the defined function must be three letters, the first two of which are FN, e.g., FNA, FNB, . . . , FNZ. Each DEF statement introduces a single function and is of the form:

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

where A may be any letter and X is a dummy variable, but must be the same on each side of the equal sign. The DEF statement may appear anywhere in the program so long as it appears before the first use of the function it defines. The function itself can be defined in terms of numbers, several variables, other functions, or mathematical expressions. For example, if the user repeatedly uses the function $e^{-x^2}+5$, he can introduce the function by the following:

```
30 DEF FNE(X)=EXP(-X↑2)+5
```

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

```
DEF  FNA(S)=S↑2
```

will cause the later statement:

```
20 LET R=FNA(4)+1
```

to be evaluated as R=17.
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 function might appear as follows:

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

where D, L, and R have been defined in the program.

THE UDEF FUNCTION CALL AND THE USE STATEMENT

OS/8 BASIC has the capability for adding one or more user-coded assembly language functions. These user functions may use four numeric and two string arguments and once properly interfaced to OS/8 BASIC, they can be used as any other OS/8 BASIC function. Complete instructions for writing and interfacing such

functions are provided later in this chapter. A user-coded function, if present, is specified in an OS/8 BASIC program as:

(line number) UDEF function name argument)

For example:

```
10 LET R=4
15 LET B=6
20 LET Q=10
25 UDEF PLT(X,Y,Z)
30 LET D=PLT(R, B,0)
35 PRINT 4*D
40 END
```

Line 25 introduces the function PLT to OS/8 BASIC and indicates the number and type of arguments associated with the function. In line 30 the function is used as any other standard function might be used in an OS/8 BASIC program. If the function requires the use of an array, a USE statement identifying the array must precede the statement that calls the function.

```
10 DIM S(15,5)
  .
  .
20 LET Q=10
22 USE S
25 UDEF PLT(X,Y,Z)
  .
  .
  .
```

NOTE

A UDEF function name may consist of alphabetic characters only and must have at least one argument (a dummy argument if necessary).

The Debugging Function—TRC(X)

The TRC(X) function is used by the programmer to follow the progress of a program and is, therefore, a useful debugging aid. The form of this function is:

(line number) vl=TRC(X)

where vl is a dummy variable, X=1 turns the function on and X=0 turns the function off. When TRC(1) is encountered in a program, OS/8 BASIC prints the line number of each line in the program as it is executed. The line numbers are printed between a pair of percent signs so as to be distinguishable from other material that is printed by the program. Program execution time is slowed down considerably to accommodate the function and the extra printing which it causes. When TRC(0) is encountered by the program the function is turned off and normal program operation resumes.

The following example shows the effect of using the TRC(X) function in a program to check the operation of a loop. The same program with the TRC(X) function removed from the program, is also shown.

With TRC(X) Function

```
5 REM BASIC
6 REM FACTORIAL PROGRAM
10 FOR J=1 TO 5
20 GOSUB 60
30 NEXT J
40 STOP
60 LET S=1
62 T=TRC(1)
65 FOR K=1 TO J
70 LET S=S*K
75 NEXT K
77 T=TRC(0)
80 PRINT J,S
85 RETURN
90 END

READY
RUNNH
% 65 %
% 70 %
% 77 %
 1            1
% 65 %
% 70 %
% 70 %
% 77 %
```

Without TRC(X) Function

```
5 REM BASIC
6 REM FACTORIAL PROGRAM
10 FOR J=1 TO 5
20 GOSUB 60
30 NEXT J
40 STOP
60 LET S=1
65 FOR K=1 TO J
70 LET S=S*K
75 NEXT K
80 PRINT J,S'
85 RETURN
90 END

READY
RUNNH
 1            1
 2            2
 3            6
 4            24
 5            120

READY
```

With TRC(X) Function

```
 2                2
% 65 %
% 70 %
% 70 %
% 70 %
% 77 %
 3                6
% 65 %
% 70 %
% 70 %
% 70 %
% 70 %
% 77 %
 4                24
% 65 %
% 70 %
% 70 %
% 70 %
% 70 %
% 70 %
% 77 %
 5                120

READY
```

Subroutines

A subroutine is a part of the program performing some operation that is required at more than one point in the program. Subroutines are generally placed physically at the end of a program, usually before DATA statements, if any, and always before the END statement.

GOSUB AND RETURN

Two statements are used exclusively in OS/8 BASIC to handle subroutines; these are the GOSUB and RETURN statements.

When a program encounters a GOSUB statement of the form:

(line number) GOSUB x

where x represents the first line number of the subroutine, control then transfers to that line. For example:

```
.
.
.
50 GOSUB 200
```

When program execution reaches line 50, control transfers to line 200; the subroutine is processed until execution encounters a

RETURN statement of the form:

(line number) RETURN

which causes control to return to the statement following the GOSUB statement. Before transferring to the subroutine, OS/8 BASIC internally records the next statement to be processed after the GOSUB statement; thus the RETURN statement is a signal to transfer control to this statement. In this way, no matter how many different subroutines are called, or how many times they are used, OS/8 BASIC always knows where to go next.
The following program demonstrates a simple subroutine:

```
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)
60 LET C=FNA(C)
70 PRINT
80 GOSUB 100
90 STOP
100 REM - THIS SUBROUTINE PRINTS OUT THE SOLUTIONS
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*B-4*A*C
140 IF D<>0 THEN 170
150 PRINT "ONLY ONE SOLUTION...X=";-B/(2*A)
160 RETURN
170 IF D<0 THEN 200
180 PRINT "TWO SOLUTIONS... X =";
185 PRINT (-B+SQR(D))/(2*A);"AND X=";(-B-SQR(D))/(2*A)
190 RETURN
200 PRINT "IMAGINARY SOLUTIONS... X = (";
205 PRINT -B/(2*A);",";SQR(-D)/(2*A);") AND (";
207 PRINT -B/(2*A);",";-SQR(-D)/(2*A);")"
210 RETURN
900 END

READY
RUNNH
?1,.5,-.5
THE EQUATION IS 1 *X↑2 +  0.5 *X + -0.5
TWO SOLUTIONS... X = 0.5 AND X=-1

THE EQUATION IS 1 *X↑2 +  0 *X +  1
IMAGINARY SOLUTIONS... X = ( 0 , 1 ) AND ( 0 ,-1 )

READY
```

Line 100 begins the subroutine. There are several places in which control may return to the main program, depending upon a certain condition being satisfied. 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 execution is terminated.

It is important to remember that subroutines should generally be kept distinct from the main program. The last statement in the main program should be a STOP or GOTO statement, and subroutines are normally placed following this statement.

More than one subroutine may be used in a single program in which case these 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 the main program is numbered with line numbers up to 199, 200 and 300 could be used as the first numbers of two subroutines.

NESTING SUBROUTINES

Nesting of subroutines occurs when one subroutine calls another subroutine. If a RETURN statement is encountered during execution of a subroutine, control returns to the statement following the GOSUB which called it. From this point, it is possible to transfer to the beginning or any part of a subroutine, even back to the calling subroutine. Multiple entry points and RETURN statements make subroutines more versatile.

The maximum level of GOSUB nesting is ten levels, which should prove more than adequate for all normal uses. Exceeding this limit results in the message:

GS AT LINE YYYYY

where YYYYY represents the line number where the error occurred. An example of GOSUB nesting follows (execution has been stopped by typing a CTRL/C, as the program would otherwise continue in an infinite loop).

```
10 REM FACTORIAL PROGRAM USING GOSUB TO
15 REM COMPUTE RECURSIVELY THE FACTORS
40 INPUT N
50 IF N>20 THEN 120
60 X=1
70 K=1
80 GOSUB 200
90 PRINT "FACTORIAL";N;"=";X
110 GOTO 40
```

```
120 PRINT "MUST BE 20 OR LESS"
130 GOTO 40
200 IF N=1 THEN 230
210 N=N-1
220 GOSUB 200
225 N=N+1
227 X=X*N
230 RETURN
240 END

READY
RUNNH
?2
FACTORIAL  2 =  2
?4
FACTORIAL  4 =  24
?5
FACTORIAL  5 =  120
?↑C
READY
```

ALPHANUMERIC INFORMATION (STRINGS)

In previous sections we have dealt only with numerical information. However, OS/8 BASIC also processes, or manipulates, alphanumeric information called strings. A string is a sequence of characters, each of which is a letter, a digit, a space, or some character other than a statement terminator (backslash or carriage return).

String Conventions

CONSTANTS AND VARIABLES

Strings may appear as constants or variables just as numerics may. We have already used string constants in PRINT statements. For example:

```
100 PRINT "THIS IS A STRING CONSTANT"
```

where the alphanumerics enclosed in quotes are the string constant.

Naming a string variable is similar to naming a numeric variable. It consists of a letter followed by a dollar sign ($) or a letter and a single digit followed by $. A$ and A1$ are both legitimate string variable names; 2A$ and AA$ are not legitimate string variable names.

DIMENSIONING STRINGS

OS/8 BASIC assumes that a string length is 8 characters or less unless a string has been dimensioned in the form:

```
10 DIM A$(I)
```

where "I" is the length of string variable A$. "I" cannot exceed 72.

String lists (equivalent to single subscripted numeric variables) are permitted in OS/8 BASIC and must be dimensioned in the form:

```
20 DIM A$(K,L)
```

where K is the number of strings in the list and L is the length of each string.

When referencing a subscripted string variable in a LET or IF-THEN statement, for example:

```
25 LET B$(I) = "YES"
```

the expression I represents the place of that string variable in the list B$.

Double subscripted string variables (string tables) are not permitted in OS/8 BASIC.

INPUTTING STRING DATA

String data may be included in a DATA list but must always be enclosed by quotation marks. In fact any string written into a program must be enclosed by quotation marks to be recognized by the OS/8 BASIC Compiler.

```
10 READ A$,B$,C$
20 PRINT C$;B$;A$
25 DATA "NG","RI","ST"
30 END
```

The program above prints STRING.

Quotation marks may be included in strings by indicating 2 quotation marks in succession. For example the string A"B would appear in a program as:

```
10 LET A$ = "A" "B"
```

Both string data and numeric data may be intermixed in a DATA list but the burden falls on the programmer to assemble the list in the correct sequence, since all READ statements for both string and numeric data remove data serially from the DATA list. If he does not, the results of the READ statement are unpredictable.

The INPUT statement may also be used for inputting string data to a program. Quotation marks are not necessary when inputting

string data in response to the question mark (?) generated by the INPUT statement unless the quotation marks are deliberately meant to be part of the string.

```
    .
    .
    .
330 PRINT "DO YOU WISH TO CONTINUE?"
340 INPUT A$
350 IF A$="YES" THEN 410
360 PRINT "ARE YOU SURE?"
370 INPUT B$
380 IF B$="NO" THEN 410
390 PRINT "PROGRAM STOPPED"
400 STOP
410 PRINT "LET'S CONTINUE"
    .
    .
    .
    .
    .
    .
490 END
```

Each string literal requested by an INPUT statement must be terminated by a carriage return which acts as the data delimiter. This is necessary since all characters, except for the carriage return, are recognized as part of the data string.

```
10 INPUT A$,B$,C$
.
.
.

RUNNH
?ABCD
?EFGH
?IJ
```

In the above example A$="ABCD", B$="EFGH" and C$="IJ".

STRINGS IN LET AND IF-THEN STATEMENTS

Strings may be used in both LET and IF-THEN statements as already indicated by some of the previous examples. Any of the relational operators may be used in an IF-THEN statement to compare strings. Strings are compared on the basis of the ASCII numeric value of each character in the string.

When comparing strings in an IF-THEN statement, the relational operators have the following significance:

Operator	Meaning
<	earlier in ASCII numeric order than
>	later in ASCII numeric order than
=< or <=	same ASCII numeric order as or earlier in ASCII numeric order than
=	same ASCII numeric order
>< or <>	different ASCII numeric order than
=> or >=	same ASCII numeric order as or later in ASCII numeric order than

For example:

```
10 IF "ABCD"<"ABC@" THEN 50
20 STOP
.
.
.
50 LET A$="ABCD"
```

Each character in string ABCD is compared, left-to-right, with the respective character in string ABC@. A, B, and C match but D and @ do not. The character @ has a lower numeric value than the character D. Therefore the string ABCD is not earlier in ASCII numeric sequence than ABC@ and the program stops at line 20.

If the strings in an IF-THEN comparison are of unequal length, then OS/8 BASIC lengthens the shorter string to make it equal in length to the longer string by appending an appropriate number of ASCII space characters. In the following example:

```
10 IF "ABCD"<"AB" THEN 50
20 STOP
.
.
50 LET A$="ABCD"
```

string "AB" is treated as "AB␣␣". Since the character C is earlier in ASCII numeric order than the character "space," the IF-THEN statement is true and control is transferred to line 50.

STRING CONCATENATION

Strings can be concatenated by means of the operator ampersand (&). The ampersand can be used to concatenate string expressions wherever a string expression is legal, with the exception that information cannot be stored by means of a LET statement in concatenated string variables. That is, concatenated string variables cannot appear to the left of the equal sign in a LET statement. For example, LET A$=B$&C$ is legal, but LET A$&B$=C$ is not. An example of string concatenation is:

```
10 READ A$,B$,C$
20 PRINT C$&B$&A$
25 DATA "NG," "RI," "ST"
30 END
```

Running this program causes STRING to be printed.

String Handling Functions

A number of functions have been implemented that perform manipulations on strings. These functions are LEN, ASC, CHR$, VAL, STR$, POS, SEG$, and DAT$. Functions that return strings have names that end in a dollar sign ($); those functions that return numbers have names that do not end in a dollar sign.

THE LEN FUNCTION

The LEN function returns the number of characters in a string. It has the form:

```
LEN(X$)
```

Example:

```
SCRATCH

READY
5 DIM B$(10)
10 READ A$,B$
20 PRINT LEN(A$&B$&"AROUND")
30 DATA "UP, ","DOWN, AND "
40 END
RUNNH
 20

READY
```

THE ASC AND CHR$ FUNCTIONS

The ASC and CHR$ functions perform conversion from and to ASCII, respectively. The ASC function converts a one character string to its ASCII decimal equivalent, and the CHR$ function converts a decimal number to its equivalent ASCII character.
The ASC function has the form:

ASC (argument)

The argument is a one character string. ASC returns the equivalent ASCII decimal number for the character.
The CHR$ function has the form:

CHR$ (numeric expression)

The value of the numeric expression is truncated to an integer that is in the range 0 to 63. Integers greater than 63 are treated modulo 64. That is, they are divided by 64 and the remainder becomes the new integer. This integer is then interpreted as an ASCII decimal number that is converted to its equivalent character (refer to Appendix A for the ASCII decimal numbers and the equivalent characters).
An example of the ASC and CHR$ functions follows:

```
5 FOR T=ASC("A") TO ASC("A")+3
10 PRINT "THIS IS TEST "&CHR$(T)
15 NEXT T
20 END
```

This is the beginning of a FOR loop that successively prints:

```
READY
RUNNH
THIS IS TEST A
THIS IS TEST B
THIS IS TEST C
THIS IS TEST D

READY
```

THE VAL AND STR$ FUNCTIONS

The VAL and STR$ functions perform conversions from strings to numbers and numbers to strings. The form of the VAL function is:

VAL (string expression)

The string expression must look like any number which may be legally typed in response to an INPUT statement. VAL returns the actual number that the string represents. The VAL function does not return the ASCII value of the number that the string represents, it returns the number. For example, VAL ("25") returns the number 25. The 25 that is the argument to VAL is a *string*, the 25 that VAL returns is a *number*.
Example:

```
10 INPUT A$
20 PRINT VAL(A$)*2
25 END

READY
RUNNH
?2.46111
 4.92222

READY
```

The STR$ function returns the string representation (as a number) of its argument. The form of STR$ is:

STR$ (numeric expression)

The string that is returned is in the form in which numbers are output in BASIC. For example, PRINT STR$(1.76111124) prints the string 1.76111.

THE POS FUNCTION

The POS function is of the form:

POS(X$,Y$,Z)

The function returns the location in string X$ of the first occurrence of string Y$ starting with Zth character in string X$. For example:

```
20 LET X$="MONDAY"
25 LET X=POS(X$,"DAY,"1)
```

After line 25, X will be equal to 4. The arguments of the POS function may be constants, variables, or expressions.

The following rules apply in evaluating the POS(X$,Y$,Z) function.

1. If Y$ is a null string (no characters) then POS(X$,Y$,Z)=1
2. If X$ is a null string (no characters) then POS(X$,Y$,Z)=0
3. If Z is less than zero or greater than the actual string length, a fatal error (PA) is detected and program execution stops
4. If Y$ is not found, then POS(X$,Y$,Z)=0

THE SEG$ FUNCTION

This function is of the form:

SEG$(X$,Y,Z)

The function returns the substring of X$ which is between positions Y and Z inclusively. For example:

```
20 LET X$="MONDAY"
25 LET B=6
30 LET A$=SEG$(X$,2*B/3,B)
```

After line 30, A$ is equal to "DAY". The arguments of the SEG$ function may be variables, constants, or expressions.

The following rules apply in evaluating the SEG$(X$,Y,Z) function.

1. If Y≤0, Y is set equal to 1
2. If Y> length X$, then SEG$(X$,Y,Z) = null string (no characters)
3. If Z≤0, then SEG$(X$,Y,Z) = null string
4. If Z > length X$, then Z is set equal to length of X$
5. If Z<Y, then SEG$(X$,Y,Z) = null string

THE DAT$ FUNCTION

The DAT$ function is of the form:

DAT$(X)

The function returns an eight character string giving the current date in the form MM/DD/YY. For example:

```
SCRATCH

READY
20 PRINT DAT$(X)
30 END
RUNNH
04/08/74

READY
```

The use of DAT$ function assumes the user has specified the date in the OS/8 monitor command DATE. If the DATE command was not used, the DAT$ function outputs a null string (no characters).

EDITING AND CONTROL COMMANDS

Several commands for editing OS/8 BASIC programs and for controlling their execution enable the user to perform such operations as:

- erase characters or lines
- list part or all of a program
- save programs on various storage devices, and
- call program from storage devices.

Correcting Programs

ERASING CHARACTERS AND LINES

Errors made while typing programs at the terminal are easily corrected. Typing a SHIFT/O or pressing the RUBOUT key causes deletion of the last character typed, and echoes as a back arrow (←) on the terminal. One character is deleted each time the key is typed. For example:

```
20 DEN F←←F FNA(X,Y)=X↑2+3*Y
```

The user types N instead of F and immediately notices his mistake. He presses the RUBOUT key (or SHIFT/O) three time, which is once for as many characters including spaces to be deleted. He makes the correction and continues typing the line. The typed line enters the computer only when the RETURN key is pressed.

```
20 DEF FNA(X,Y)=X↑2+3*Y
```

Sometimes it is easier to delete a line being typed and retype the line rather than attempt a correction using rubouts. Typing CTRL/U or pressing the ALTMODE key will delete the line currently being worked on and echoes DELETED and a carriage return-line feed. Use of the CTRL/U or ALTMODE command is equivalent to typing rubouts back to the beginning of the line.

To delete a line that has already been entered into the computer the user simply types the line number followed by a carriage return. Both the line number and the line are removed from his program.

The user may change individual lines by simply typing them in again. Whenever a line is entered, it replaces any existing line which has the same line number. New lines may be inserted anywhere in the program by giving them line numbers which are between two other existing line numbers. Using these editing capabilities, the program may be modified and re-run until it works properly.

THE RESEQ PROGRAM

After the user has extensively modified his program, he may find that some portions of the program have line numbers spaced so closely together that they do not permit any further addition of statements should he wish to do so. Renumbering the lines in the program so as to provide a practical increment between line num-

bers can be accomplished automatically by using the RESEQ program. It should also be noted that the RESEQ program modifies the line numbers in GOSUB and IF-THEN statements to agree with the new line numbers assigned to statements by the program. Line lengths must not exceed 72 characters.
Typically, the program would be used as follows:

SAVE DSK:SAMPLE	User saves program SAMPLE which requires renumbering.
READY	BASIC is ready for next command.
OLD DSK:RESEQ	User calls for program RESEQ.
READY	BASIC is ready for next command.
RUNNH	User runs program.
FILE? DSK:SAMPLE.BA	Program asks for filename. User responds with name of program to be renumbered.
START,STEP?100,10	Program asks for a starting line number (START) and for the increment between line numbers (STEP). User requested that SAMPLE start with line number 100 and each line be incremented by 10.
READY	Renumbering is accomplished. BASIC ready for next command.
OLD DSK:SAMPLE	User calls back his program.
READY	BASIC ready for next command.
LISTNH	User gets listing of program SAMPLE for further modification.

The LIST and LISTNH Commands

An entire program can be listed on the terminal by typing LIST. A heading is printed before the program itself and is of the form:

FILE EX VERSION NO. DATE

For example, if the user is working on a program named USER.BA and wants a listing he types:

```
LIST
```

and BASIC responds with:

```
USER   BA   1.0   26-JUL-72
100 LET X=1
    .
    .
    .
    .
    .
    .
200 END
READY
```

NOTE

When any OS/8 BASIC command is completed, the message READY is printed at the terminal. OS/8 BASIC is then ready to accept any other commands from the user.

A part of a program may be listed by typing LIST followed by a line number. This causes that line and all following lines in the program to be listed. For example:

```
LIST 100
```

will list line 100 and all remaining lines in the program. Typing CTRL/O while the listing is being printed terminates the printing and outputs the READY message.

The LISTNH command may be used exactly as the LIST command, but it eliminates the heading from the listing.

The SCRATCH Command

The command:

```
SCRATCH
```

is provided to allow the programmer to clear his storage area, deleting any commands, or a program which may have been previously entered, and leaving a clean area in which to work. If the storage area is not cleared before entering a new program, lines from previous programs may be executed along with the new program, causing errors or misinformation. The SCRATCH command eliminates

all old statements and numbers and should be used before new programs are created.

The NEW Command

The NEW Command is used to name a program to be created and performs an inherent SCRATCH on the storage area. The command is in the form:

```
NEW FILE.EX
```

It assigns the filename to the program to be created. For example:

```
NEW USERA.BA
```

creates file USERA.BA.
An alternate method of naming programs is to type NEW followed by the RETURN key. BASIC responds with:

```
FILE NAME—
```

The user types the filename and extension followed by the RETURN key.

```
NEW                          (.BA is assumed and need not
FILE NAME—USERA.BA           be typed)
```

NOTE

Extension .BA is assumed in the OLD, NEW, NAME, and SAVE commands unless otherwise specified. If no extension is specified in the OLD command, OS/8 BASIC first tries to load FILE.BA and if unsuccessful tries to locate and load FILE without the extension.

The OLD Command

This command retrieves a previously created file from a storage device and places the file in the storage area of the computer. The command is of the form:

```
OLD DEV:FILE.EX
```

For example:

```
OLD DTA1:SAMPLE.BA
```

retrieves file SAMPLE from DECtape number 1 and places it in the storage area of the computer.
An alternate method of retrieving a file is to type OLD followed by the RETURN key. BASIC responds with:

```
FILE NAME—
```

The user types the device, filename and extension followed by the RETURN key.

```
OLD
FILE NAME—DTA1:SAMPLE.BA
```

If the device is omitted, DSK: is assumed.

The NAME Command

This command permits the user to rename the program in the storage area of the computer. The command is of the form:

```
NAME FILE.EX
```

Since this command changes only the name of the storage area of the computer—not its contents, it can be used to create two almost identical versions of the same program. This is accomplished by retrieving the first file, making modifications to it, and then SAVE-ing the modified version under the new name.

The SAVE Command

The SAVE command saves on the specified device the file currently in the storage area of the computer. The command is of the form:

```
SAVE DEV:FILE.EX
```

If device is omitted, DSK: is assumed. If filename and extension are omitted, the current filename and extension are used.

The SAVE command also provides a convenient method for listing large programs quickly on the line printer rather than the terminal. For example:

```
SAVE LPT:
```

will list the current program on the line printer.

The RUN and RUNNH Commands

After a BASIC program has been typed and is in core, it is ready to be run. This is accomplished by simply typing the command:

```
RUN
```

The program heading is printed and the program will begin execution. If errors are encountered, appropriate error messages will be printed on the keyboard; otherwise, the program will run to completion, printing whatever output was requested. When the END statement is reached, OS/8 BASIC stops execution and prints:

```
READY
```

If the program does not run properly, or contains an infinite loop and, hence, will never stop, it may be terminated by typing CTRL/C, which returns control to the OS/8 BASIC editor (READY).

The RUNNH command suppresses printing of the heading and may be used in place of the RUN command.

NOTE

If a program that is not SAVEd is RUN, and for some reason is not retrievable by LISTing, the program may be retrieved by calling for OLD file BASIC.WS.

The BYE Command

The BYE command instructs the computer to exit from OS/8 BASIC and returns control to the OS/8 Keyboard Monitor. Typing a CTRL/C while in the OS/8 BASIC editor (READY) mode also returns control to the OS/8 Keyboard Monitor.

NOTE

Never type BYE before SAVEing a newly typed program. Unless the SAVE command is used, the program will be lost.

FILES, FILE STATEMENTS AND CHAINING

General Information On OS/8 BASIC Files

SYSTEM DEVICES

The file capability provided by OS/8 BASIC allows the user to write information into (PRINT#) or read information from

(INPUT#) files on devices other than the terminal. Devices such as disks, DECtapes, and line printers permit the user the flexibility and advantage of mass storage and high speed I/O, when writing OS/8 BASIC programs. Each device in the OS/8 system is referenced by means of its standard OS/8 permanent device name.

Files are referenced symbolically by a name of up to six alphanumeric characters followed, optionally, by a period and an extension of two alphanumeric characters. The extension to a file name is generally used as an aid for remembering the format of a file.

A fixed length file is one which is already in existence. That is, it has been created and CLOSEd. The length of a fixed length file is equal to the number of blocks in the file and cannot be changed.

A variable length file is a newly created file. Until the file is CLOSEd, it is equal in length to the largest free space on the device. When the file is CLOSEd it becomes a fixed length file equal in length to the actual number of blocks it occupies. Unless the file is CLOSEd, the CHAIN, STOP or END statements will cause a loss of the file.

Data in numeric files are stored as successive three-word floating-point numbers (85 to each OS/8 block) with the last word in each block unused.

Data in ASCII files are stored in standard OS/8 format (three 8-bit characters to every two words). Refer to the next section of this chapter and the PS/8 Software Support Manual (DEC-08-MEXB-D).

File Statements

OS/8 BASIC provides a number of statements for operating on files. They include:

```
FILE#
PRINT#
INPUT#
RESTORE#
CLOSE#
IF END#
```

The statements are distinguishable from other OS/8 BASIC statements by the number sign (#) which appends the statement name.

The FILE # statement is used to open a file and is of the form:

(line number) FILEtype # numeric expression: string expression

where:

a. FILEtype is one of four possibilities:

FILEtype	Definition
FILE	Fixed length—ASCII
FILEV	Variable length—ASCII
FILEN	Fixed length—numeric
FILEVN	Variable length—numeric

b. The numeric expression has a value of from one to four and represents the internal file number for the file number being opened and is used in any other statements referencing this file. FILE #0 references the Teletype.

c. The string expression is either a string variable or string constant which has a value of the form:

```
DEV:FILE.EX
```

For example, the following program:

```
10 LET A$="DTA1:NETSAK.BA"
20 FILEN #1:A$
```

is equivalent to:

```
10 FILEN #1:"DTA1:NETSAK.BA"
```

for opening fixed length numeric file NETSAK on DECtape number 1 and assigning it internal file number 1. If DEV: is missing from the string expression, the device DSK is assumed by default. Only four files, (numbered 1 through 4) besides the Teletype (FILE #0) may be open in a program at any time. However, the ability to open and close (CLOSE#) files under program control permits the user to access an unlimited number of files.

When selecting a FILEtype, the user should keep in mind that variable length files (FILEV and FILEVN) are restricted to outputting only. That is, variable length files should be used in conjunction with the PRINT# statement only. An attempt to read

(INPUT#) from a variable length file results in error message VR. Only one FILEV or FILEVN may be active per device at a given time.

The use of fixed length ASCII files (FILE) should be restricted to use with the INPUT# statement, but may be used with the PRINT# statement when the user is certain that the ASCII or numeric data he is PRINTING (i.e., the number of characters) replaces, exactly, the ASCII or numeric data on the file. It is recommended that the use of fixed length ASCII files (FILE) for output be entirely avoided.

The PRINT# statement writes data into files and is of the form:

(line number) PRINT#N: list of expressions and delimiters

where N is the numerical expression for a file number. For ASCII files, the expressions in the list can be string or numeric, and the TAB and PNT functions can both be used. The delimiters can be commas or semicolons and have the same meanings that they have in the PRINT statement for the terminal. For numeric files, the expressions may only be numeric variables separated by commas or semicolons.

```
10 FILEV #1:"LPT:"
20 LET F=1
30 PRINT #F: TAB(28);DAT$(X)
40 CLOSE #F
50 END
```

This routine prints the date, starting at column 28 on the lineprinter.

The INPUT# statement reads data from files and is of the form:

(line number) INPUT #N: list of variables

where N is the numerical expression for a file number. The INPUT# statement does not expect a line number on each line of data in the file. If one is present, it is read as data:

```
10 DIM N$(19,15)
20 FILE #1:"LARRY"
30 FOR I=1 TO 19
40 INPUT #1: N$(I)
50 NEXT I
```

The above routine reads 19 strings from DSK: file LARRY.

The previous paragraph indicated that numbers may be written into an ASCII file. The reading of numbers from an ASCII file requires some precaution if the data delimiter is other than a comma or semicolon. In line 30 of the example below, the data written into file number 1 will be separated by a carriage return and line feed which are both written into the file. The subsequent reading of numbers from the file in line number 80 shows the use of a pair of dummy arguments (C and L) to compensate for the carriage return and line feed since they would otherwise be read as numeric data with a value of 0.

```
10 FILEV #1:A$
20 FOR I = 1 TO 10
30 PRINT #1:I
40 NEXT I
50 CLOSE #1
60 FILE #1:A$
70 FOR I = 1 TO 10
80 INPUT #1:J,C,L
90 NEXT I
```

The RESTORE# statement is of the form:

(line number) RESTORE #N

where N is a numerical expression for the file number to be reset to the beginning. If N is equal to zero, or if #N is missing from the statement, the DATA list in the program is reset to the beginning.

```
10 FILE #2: "SUSAN"
20 INPUT #2: A,B,C,D
25 RESTORE #2
30 INPUT #2: E,F,G,H
```

This program uses the same values from disk file SUSAN for variables A, B, C, and D as it does for variables E, F, G, and H. The CLOSE# statement is of the form:

(line number) CLOSE #N

where N is the numerical expression for the file number to be closed. For example :

```
10 DIM A$(5,10)
15 FOR I=1 TO 4
20 LET A$(I) = "SHERY" & CHR$(I+48)
25 FILE #I:A$(I)
30 INPUT #I:B$
35 IF B$="SANDY" THEN 60
40 CLOSE #I
45 NEXT I
50 PRINT "CANNOT FIND SANDY"
55 STOP
60 PRINT "FOUND SANDY"
   .
   .
   .
   .
   .
   .
90 END
```

This program searches through four disk files (SHERY1 through SHERY4) for the file that has SANDY as the first entry. If the first entry in the file is not SANDY, the file is closed (statement 40) and the next file is opened.

NOTE

The user must CLOSE# all output files before ending the program in order to prevent the loss of data.

The IF END# statement allows the user to determine whether or not there has been an End-of-File detected for the ASCII file in question. The statement has the form:

(line number) IF END #N THEN line number

where N is a numerical expression for the ASCII file number. The line number must refer to a line in the program.

```
   .
   .
   .
   .
100 IF END #1 THEN 170
```

```
.
.
.
.
170 END
```

If the IF END# statement is found true, the last INPUT# or PRINT# (read or write) should be discarded.

The IF END# statement is not used to determine if more data is available, but rather to determine if the last read or write was valid. The first attempt, in an INPUT# or PRINT# statement, to read or write past an end-of-file causes an abort of the I/O associated with that read or write, and the program passes control to the next operation to be performed. The second, and any subsequent attempts, to read or write past an end-of-file, causes an RE or WE runtime error to be printed for each syntactical item in the INPUT# or PRINT# list. To avoid a lengthy list of error messages, avoid using long INPUT# or PRINT# lists in situations which may approach an end-of-file.

Core image files have been implemented under OS/8 BASIC V3. To create a core image file from a BASIC language source, type:

```
.R BCOMP
*DEV:PROG.BA/K
```

where PROG.BA is the source. The K switch indicates that a core image file is to be created. The following additional switch/option designations apply:

/N If the resulting core image will *never* be executed on a 12K TD8E system. This saves 400 words of memory but reduces configuration independence.

/B If a copy of the run-time system should be loaded into the core image. This increases the size of the core image file by 10-50% (exactly 15 blocks) but eliminates the need for a file access to read in BRTS at run-time. BRTS and overlays must still exist on the system device, when the program is run.

=n Where n is an octal digit in the range $1 \leq n \leq m$ and m is the highest memory field available on the host machine. The highest memory field on the target machine will be n. This may reduce configuration independence, since the

resulting core image will not load correctly on a machine with fewer than n+1 memory banks. If n is not specified, n= 1 is assumed. If n is specified larger than m, n=m is assumed.

In the absence of error conditions, the compiler post-processor (BLOAD) will exit to OS/8. At this time, type:

```
.SA DEV:PROG
```

to create an executable core image. Additional arguments to the save command must not be specified. The core image is executed by typing:

```
.R PROG
  or
.RUN DEV:PROG
```

as appropriate. Any errors flagged during compilation will prevent creation of an executable core image file.
In contrast to normal BASIC program execution, which requires a minimum of 6 file access operations, core image file execution requires no more than two file accesses; one to read the core image file and one to read BRTS if /B was not specified. Compiler/loader overhead is also eliminated, so that the reduction in execution time is very significant, especially on DECtape systems. The following error messages may occur during execution of a BASIC core image file:

USER ERROR 1 AT nnnn
One of the files:

```
BRTS.SV
BASIC.AF
BASIC.SF
BASIC.FF
```

was missing from the system device.

USER ERROR 2 AT nnnn
An attempt was made to load a core image file produced under the /N option on a 12K TD8E system (without ROM).

USER ERROR 3 AT nnnn
Insufficient core to load this core image file.

When executing BASIC core image files on a DECtape system, the following techniques will ensure minimum execution time:

1. Follow the recommended procedure for grouping calls to functions according to the overlay in which the function resides, to minimize overlaying at run-time. (Refer to the section entitled THE OS/8 BASIC RUN-TIME SYSTEM)
2. Prepare a system DECtape which contains all of the BASIC core image files, followed by:

   BRTS.SV
   BASIC.AF
   BASIC.FF
   BASIC.SF
   BASIC.UF (optional)

The BASIC core image files should reside near the beginning of the DECtape. If chaining is employed, the least frequently run programs should appear first on the DECtape.
The CHAIN statement provides a convenient means for dividing large programs into a series of smaller programs which are written and stored separately, and executed in a chain. The CHAIN statement is of the form:

(line number) CHAIN "DEV:Filename.extension"

When BASIC encounters a CHAIN statement in a program, it stops execution of that program, retrieves the program named in the CHAIN statement from the specified device and file, compiles the chained program and begins execution of the program. The use of the CHAIN statement, therefore, is the automatic equivalent of running an OLD with no header (RUNNH). The file BASIC.WS will contain the original program in the chain and when execution is complete, the BASIC storage area will contain the original program.
Since BASIC removes the program which contains the CHAIN statement from core before retrieving the chained program, the user should make certain to CLOSE# all output files that are opened by FILE statements in the program which contains the CHAIN statement in order to avoid the loss of data generated by the program.

A BASIC core image file may also execute CHAIN statements, subject to the following restrictions:

1. A BASIC language source may not CHAIN to a BASIC core image and vice versa.
2. When a BASIC language source chains to another BASIC language source as described above, the extension of the target file must not be ".SV". When a BASIC core image chains to another BASIC core image, the extension of the target file must be ".SV". This is the standard extension for BASIC core image files.

In general, any departure from these procedures will cause a "CL" error.

NOTES

1. Control commands OLD, RUNNH, and SAVE are described in Chapter 8.
2. If DEV: is not specified, DSK: is assumed by default.

CREATING ASSEMBLY LANGUAGE FUNCTIONS

Introduction

OS/8 BASIC has a facility which allows *experienced* PDP-8 assembly language programmers to interface their own assembly language routines to OS/8 BASIC. This facility permits the user to add functions to OS/8 BASIC which can operate directly on special purpose peripheral devices. This section describes in some detail the organization and internal characteristics of the OS/8 BASIC Run-time System (BRTS)and is intended to serve as a programming guide for the creation of such user-coded assembly language functions. This material assumes the user to be familiar with OS/8 and PDP-8 assembly language.

In addition to this section the programmer would find most useful a listing of the OS/8 BASIC Run-time System (DEC-S8-LBASA-A-LA).

The OS/8 BASIC System

The OS/8 BASIC system is divided into the following discrete parts:

1. The BASIC editor (BASIC.SV)
2. The BASIC compiler (BCOMP.SV)
3. The BASIC loader (BLOAD.SV)
4. The BASIC run-time system (BRTS.SV)
5. The run-time system overlays (BASIC.AF) (BASIC.SF) (BASIC.FF) (BASIC.UF) (if needed)

The OS/8 BASIC editor is used to create and edit the source program. On receipt of a RUN command, the editor creates a temporary file called BASIC.WS, stores the source in that file, then chains to the compiler. The compiler compiles the program into a 12-bit pseudo-code which is loaded into core along with the run-time system by the loader. The run-time system interprets each pseudo-instruction, calling each of the overlays into core as needed. On completion of the program, the run-time system chains back to the editor, and the cycle is repeated. Following is a diagram showing the files on the systems device associated with or used by each system component.

BASIC Component	Associated Files	Usage
Editor	BASIC.WS	program storage
Compiler	BASIC.WS	program storage
	BASIC.TM	compiled code storage
Loader	BASIC.TM	compiled code storage
Run-time System	BASIC.AF BASIC.SF BASIC.FF BASIC.UF	overlays (if needed)

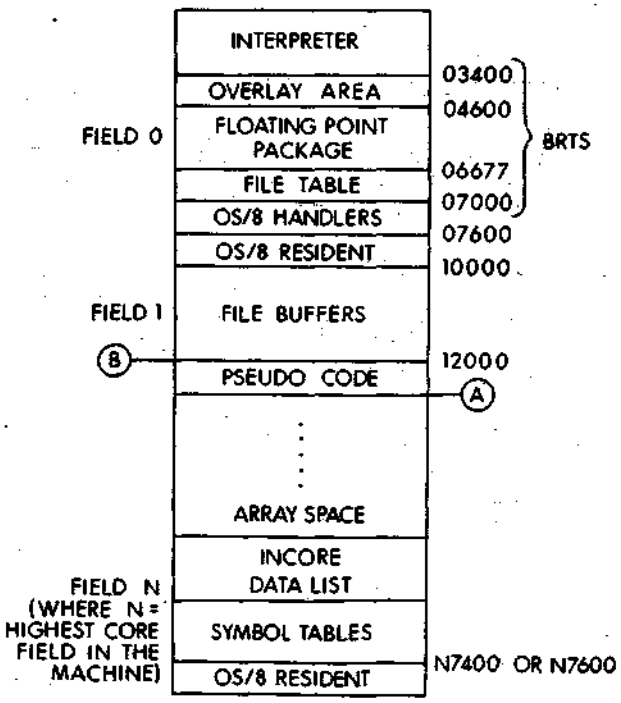
The user must avoid the filenames above; they are reserved for the OS/8 BASIC System.

The OS/8 BASIC Run-time System

At the time the user's BASIC program is actually being executed, the portion of the BASIC system in control is the Run-time System (BRTS). BRTS is also in core when user-coded functions are executed, therefore, a knowledge of it is essential to writing an OS/8 BASIC assembly language function. Note that the following sections refer frequently to specific core locations in BRTS by symbolic names (always capitalized). The actual value of these symbols can be obtained from the symbol table for the version of BASIC being used. Note that this symbol table is for a non-EAE system; if the EAE overlay is used some of the minor symbols will change. The major routine entry points mentioned in this chapter, however, are the same for both systems. All diagrams in this chapter have the lowest core address at the bottom. This chapter also makes use of the variable names A, A(0,0), A$, and A$(0) to represent the general case. All references in this chapter to "page 0" refer to the BRTS page 0 (Page 0, Field 0).

BRTS CORE LAYOUT

When executing, BRTS has the following configuration:



The highest core field is used for BRTS symbol tables, storing of the field 1 and field 2 (if non-ROM TD8/E) resident portions of the OS/8 Monitor, the incore DATA list (data generated by DATA statements in the program) and pseudo-code (generated by the compiler). The bottom of the array space (marked by line A) can exceed the field boundary and proceed into lower fields, but this will only happen for large programs. Note that if the bottom of the pseudo-code extends below line B (12000), some file buffer space must be sacrificed, with corresponding loss of runtime file capabilities. As the bottom of the pseudo-code approaches 10000, the number of files which may be simultaneously open at runtime approaches 0. At least 400 words of buffer must be free for each file opened at runtime. The 8 file buffers and OS/8 handler areas are allocated dynamically at runtime in response to FILE commands in the BASIC program, and if not fully used may be used as buffer space by the user function.

BRTS OVERLAYS

Locations 3400-4577 of field 0 serve as an overlay area, into which the currently needed overlay is read. The overlays consist mainly of functions which are infrequently used, and are constructed as follows:

BASIC.AF	Arithmetic Functions
	SIN, COS, ATN, EXP, FIX, FLOAT, INT, RND, EXPONENTIATION, SGN, SQR, LOG
BASIC.SF	String Functions
	ASC, CHR$, DAT$, LEN, POS, SEG$, STR$, VAL, Error processing, TRC
BASIC.FF	File Functions
	CHAIN, CLOSE, FILE, STOP
BASIC.UF	User Function

This last overlay, BASIC.UF is reserved for user-written assembly language subroutines. Each time a call to the user routine is issued, the overlay BASIC.UF is read into the overlay buffer, and control is dispatched to the appropriate routine.

Note that the overlay driver reads in a new overlay only if the overlay currently resident does not contain the function specified in a given function call. If the function call is for a function which is found in the currently resident overlay, no overlay I/O takes place.

BRTS SYMBOL TABLES

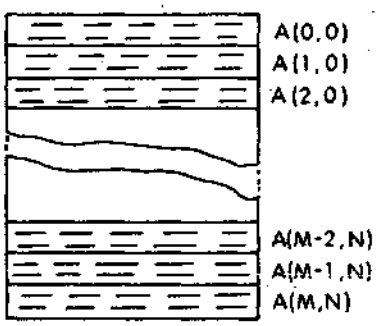
BRTS locates variables and strings at runtime via four permanently resident symbol tables. These tables, which always reside in the highest core field, are the Scalar Table (for variables like A, B1), the Scalar Array Table (A(1),B(1,1)), the String Symbol Table (A$, A1$), and the String Array Table, (B1$(2)).

Data Formats

VARIABLES

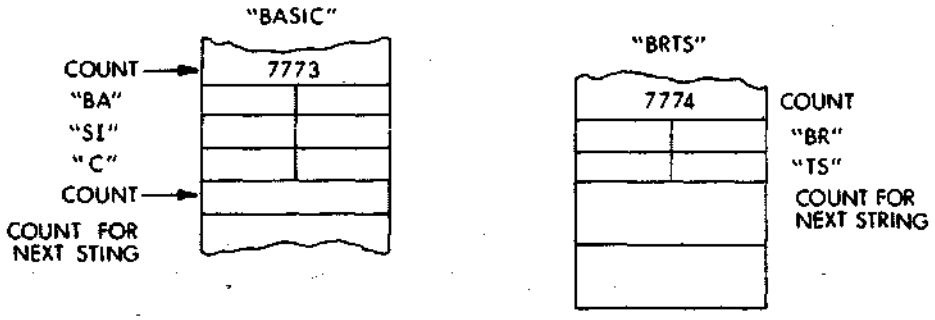
Variables are stored in core as standard three word floating point numbers. The first word is a signed, two's complement exponent, while the second and third word represent the signed, two's complement mantissa.

Single variables are stored as three word entries in the Scalar Table. Arrays are stored in core as successive three word entries, with the first subscript varying the fastest, and A(0,0) occupying the lowest core address. The address and field of A(0,0) are specified in the Scalar Array Table.



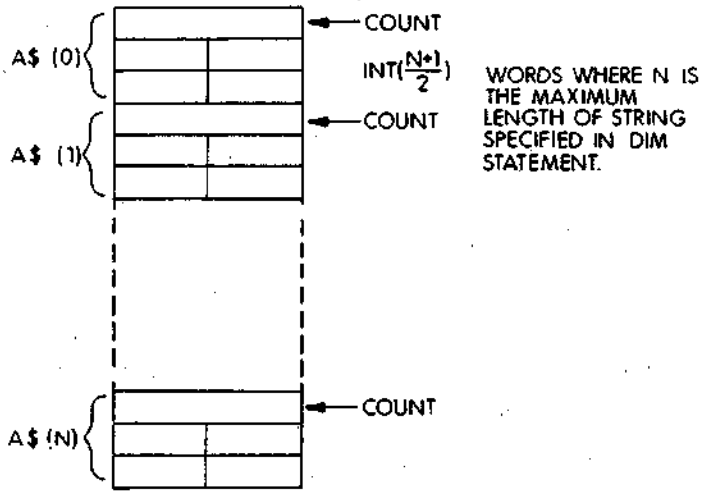
STRINGS

Strings are stored as 6-bit ASCII characters, with a character count as the first word of the string. The left half of each character word is used first, with unused characters padded with spaces (40_8). The character count is a signed, two's complement number representing the actual number of characters in the string, not the number of words devoted to that string. Each string is allocated $[INT(\frac{n+1}{2})+1]$ words, where n is the maximum length specified in a DIM statement, whether that many words are actually used or not.



The minimum string is one character long. The address of the count word for each string is pointed to by its entry in the String Symbol Table.

String arrays are stored as successive strings, with A$(0) occupying the lowest core address. Each string is allocated enough space for its maximum length, even though all of this space may not be used.



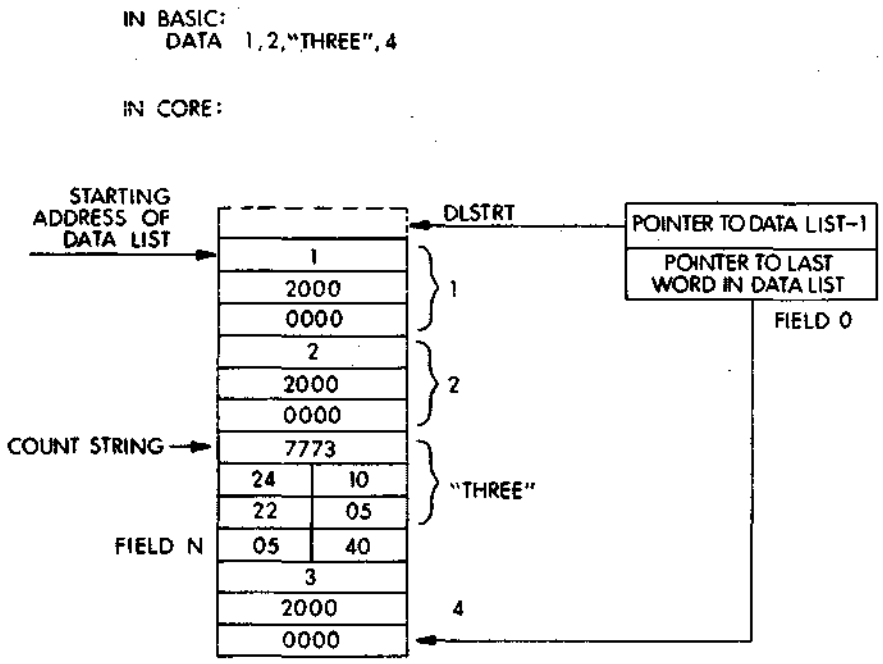
NOTE

For any of the above data types, a field boundary may fall anywhere within any individual item. Routines that use successive words in any data item must be careful to check for a field boundary within that item.

INCORE DATA LIST

The incore DATA list is stored as sequential data items in core. Strings again are devoted even numbers of words, and are prefixed by a count. There is no separator or identifier of DATA items and the DATA list is always in the highest core field. A page 0, field 0 pointer to the starting address of the DATA list *less 1* is maintained at DLSTRT, and the address of the last word of the list can be found at DLSTOP.

Example:



THE STRING ACCUMULATOR (SAC)

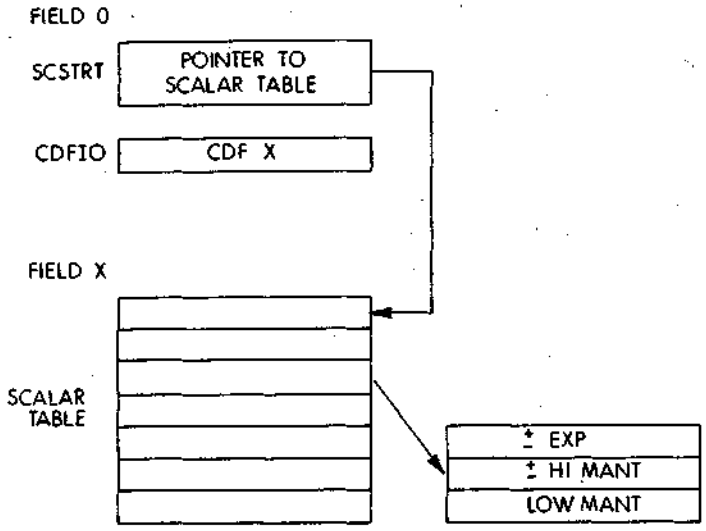
All BRTS string operations use the String Accumulator (SAC) as one of the operations, and the result is always left in the SAC. The string accumulator is to strings as the hardware AC is to PDP-8 instructions. The SAC starts at location SAC for 36 words (72 characters), and the length of the string currently in the SAC is stored as a negative number in STRLEN. A page 0 pointer to the start of the SAC *less 1* is maintained at SACPTR.

BRTS Symbol Table Structure

The BRTS symbol tables all reside in the highest core field. A CDF to the symbol table field can be found in location CDFIO of field 0.

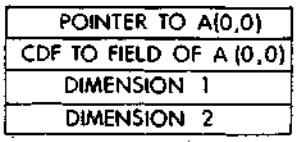
THE SCALAR TABLE

The Scalar Table is the highest symbol table in core, and it consists of successive three-word entries, each entry containing a 3-word floating-point number. One entry exists for each variable used in the program, and a few extra entries are used as temporaries. A pointer to the start of the Scalar Table can be found at location SCSTRT of field 0. The scheme for scalar variables is as follows:



THE ARRAY SYMBOL TABLE

The Array Symbol Table consists of successive 4-word entries, each entry specifying the location and size of an array used in the program. Each entry is as follows:

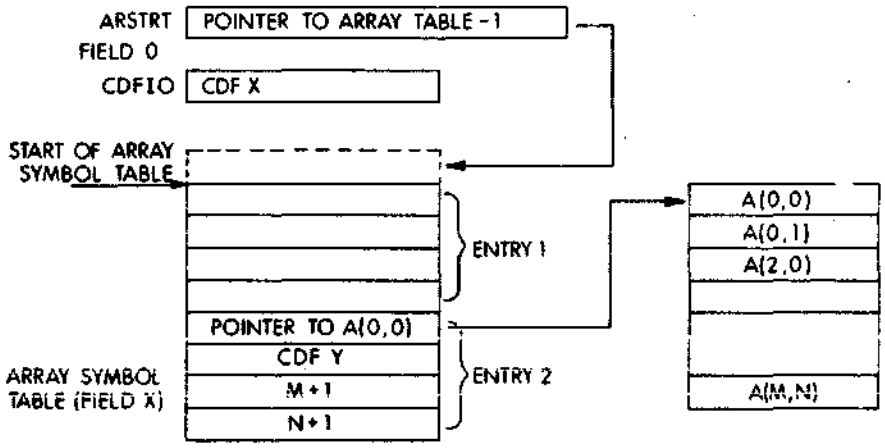


The first word of each entry is a 12-bit pointer to the location of the exponent word of the first element in the array. The second word is a CDF N where N is the field for the pointer in the first word. The third word is the first dimension of the array (obtained by adding 1 to the M in a DIM A(M,N) statement because the

first subscript is always 0), and the last word is the second dimension of the array (obtained by adding 1 to the N in the aforementioned DIM statement for the same reason). If the array is unidimensional, the second dimension is zero. To locate the n^{th} element in the array, BRTS performs the following calculation:

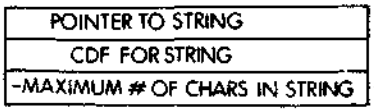
Addr of A(M,N)=3*[M+(DIM1+1)*N]+Addr of A(0,0)

A pointer to the start of the Array Symbol Table *less 1* (for use in an index register) can be found in field 0 at location ARSTRT. The scheme for arrays is:

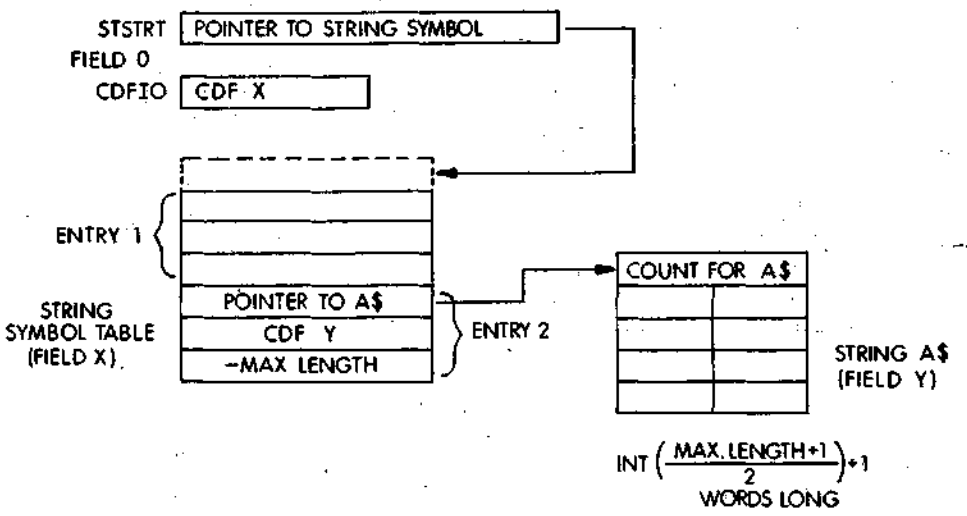


THE STRING SYMBOL TABLE

The String Symbol Table has successive three-word entries as follows:



The first word is a 12-bit pointer to the count word of the string. The second word of each entry is a CDF for that count word, and the third word of the entry is the maximum length of the string (in number of characters) stored as a two's complement negative number. A pointer to the start of the String Symbol Table (*less 1*) can be found in field 0 location STSTRT. Note that the maximum number of characters in the string represents the amount of space allocated for the string; the amount of space actually used is represented by the count word which is stored with the string.
The scheme for simple strings is:



THE STRING ARRAY TABLE

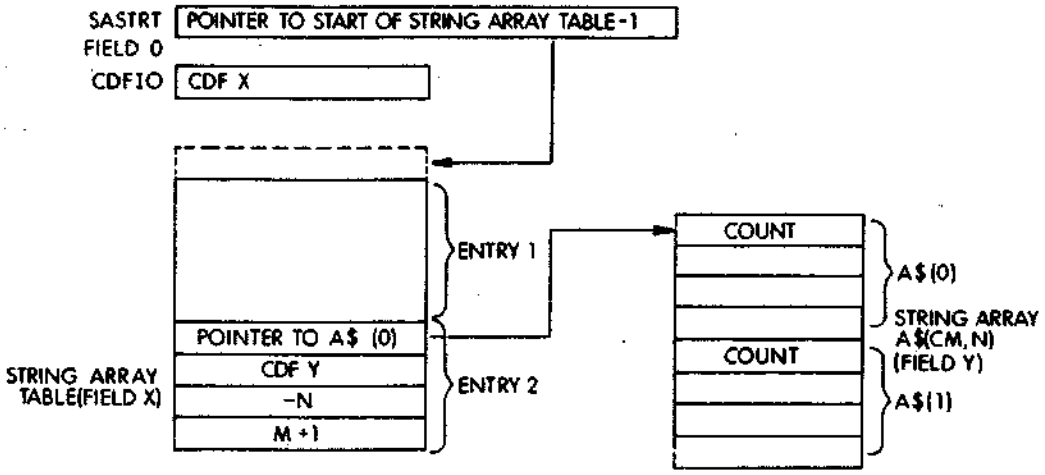
The String Array Table consists of consecutive 4-word entries, with each entry as follows:

POINTER TO A$(0)
CDF FOR A$(0)
-MAX # OF CHARS IN A$(0)
DIMENSION OF A$(0)

The first word contains a pointer to the count word of string A$(0), and the second word contains a CDF for this count. The third word has the maximum length (in characters) of each element

in the array stored as a two's complement negative number. The last word contains the dimension of the string array, obtained by adding 1 to the M in a DIM statement of the form DIM A$ (M,N) because the first element is always A$(0). A pointer to the start of the String Array Table *less 1* can be found in field 0 at location SASTRT.

The scheme for string arrays is:



To locate the n^{th} element of the string array, BRTS performs the following calculation:

$$\text{addr of A\$ (N)=addr of A\$ (0)+(INT}\frac{(\text{ABS(Z)+1})}{\text{N}}\text{+1)*N}$$

where Z = individual element length.

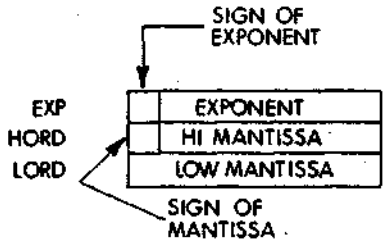
Floating-Point Operations

The BRTS floating-point package is permanently resident, and as such it is readily available for use by assembly language routines for floating-point calculations.

FLOATING-POINT ACCUMULATOR

One of the operands of every floating-point operation is the Floating Accumulator (FAC), and the result of all floating-point operations (except FPUT) is always left in the FAC. The FAC is

found at EXP, HORD and LORD on page 0 with standard PDP-8 23-bit floating-point format:



The floating-point accumulator is to floating-point instructions what the hardware accumulator is to PDP-8 machine language instructions.

FLOATING-POINT ROUTINES

The following floating-point routines are available for user subroutine use:

Function	Starting Address	Operation
ADD	FFADD	FAC←FAC+OPERAND
SUBTRACT	FFSUB	FAC←FAC−OPERAND
MULTIPLY	FFMPY	FAC←FAC*OPERAND
DIVIDE	FFDIV	FAC←FAC/OPERAND
INVERSE SUBTRACT	FFSUB1	FAC←OPERAND−FAC
INVERSE DIVIDE	FFDIV1	FAC←OPERAND/FAC
LOAD FAC	FFGET	FAC←OPERAND
STORE FAC	FFPUT	OPERAND←FAC

The symbol "←" means "is replaced by".

Note that the store function (FFPUT) is the only operation in which the result is not left in the FAC. Note also that FFPUT is a nondestructive store, i.e., the FAC is the same after the store operation as before.

There are two calling sequences for the floating-point routines, each with a different method for passing the address of the operand. Mode 1 is the most efficient, and can be used whenever the operand is in field 0. Mode 2 is the field independent call, but is more core expensive than mode 1.

The mode being used is determined as follows:

1. If the contents of the AC is non-zero on entry, the mode used is mode 2.
2. If the contents of the AC is zero on entry, the location FF is examined. If FF is also zero, mode 1 is the calling mode. If FF is non-zero, mode 2 is used.

The calling modes are as follows:
Mode 1—address of operand follows call to floating-point routine.

```
        CLA
        DCA FF              /SWITCH FF=0 FOR MODE 1
        JMS I POINTR        /JUMP TO FLOATING-POINT ROUTINE
        (OPERAND ADDR)      /12 BIT ADDRESS OF OPERAND
        .                   /RETURNS HERE
        .
        .
POINTR, (STARTING ADDR)
                            /FLOATING-POINT ROUTINE
                            /STARTING ADDRESS.
```

Mode 2—address of operand in AC on call to floating-point routine.

```
       CLA IAC
        DCA FF              /FF SWITCH NOT EQUAL TO 0 FOR MODE 2
        CDF N               /DF TO FIELD OF OPERAND
        TAD OPADDR          /ADDRESS OF OPERAND
        JMS I POINTR        /JUMP TO FLOATING-POINT ROUTINE
        (UNUSED)            /THIS LOCATION UNUSED
        .                   /RETURNS HERE.
POINTR, (STARTING ADDR)     /ADDRESS OF FLOATING-POINT ROUTINE
OPADDR, (OPERAND)           /ADDRESS OF OPERAND
```

Both modes return with a clear AC and the data field set to 0. Note that the switch FF is not altered by the routines themselves, hence it is only necessary to set it when desired to change modes, not before every call.

The mode 2 call always returns to the second instruction following the JMS call, skipping the word following the JMS. Since this word is completely unused, it is a good location for constant storage.

The FF switch is necessitated by the special case when it is desired to reference an operand located at location 0 in a field other than field 0. If the FF switch were not present, the floating-point package would examine the AC, find it empty, and use the address in the word following the call, since there is no way of distinguishing an empty AC from an operand address of 0 loaded into the AC. The FF switch, then, is used to tell the floating-point package whether the zero AC means "mode 1 call" or "operand at 0."

BRTS maintains links for FGET and FPUT on page 0 of field 0, providing convenient access to these frequently used routines.

Page 0 Link Name	Routine Linked
FGETL	FFGET
FPUTL	FFPUT

Examples:
Some examples of BRTS floating-point code:

1. Routine to calculate X^2+2X+1

```
.
.
CLA
DCA FF          /OPERAND ADDRESS WILL
                /FOLLOW CALLS (MODE 1)
JMS I FGETL     /LINK IS ON PAGE 0
X
JMS I FMPYLK    /X * X
X
JMS I FPUTL     /SAVE X^2
Y
JMS I FGETL     /LOAD X AGAIN
X
```

```
        JMS I FMPYLK    /2X
        TWO
        JMS I FADDLK    /2X+1
        ONE
        JMS I FADDLK    /X^2+2X+1
        Y
        .
        .               /RESULT NOW IN FAC
        .
FADDLK, FFADD           /LINK TO ADD ROUTINE
FMPYLK, FFMPY           /LINK TO FLOATING MULTIFLY
TWO,    0002            /FLOATING POINT CONSTANT
        2000            /2.0
        0000
ONE,    0001            /FLOATING POINT CONSTANT
        2000            /1.0
        0000
X,      ...             /VARIABLE
        ...
        ...
Y,      0               /FLOATING POINT TEMPORARY
        0
        0
```

2. Routine to add 5 successive floating-point numbers starting at location 0 of field 2.

```
START,  CLA
        DCA OPADDR      /FIRST OPERAND AT LOCATION 0
        JMS I FCLR      /ZERO FAC
        IAC
        DCA FF          /CALLS ARE MODE 2
ALOOP,  CDF 20
        TAD OPADDR      /OPERAND ADDR IN AC
        JMS I FADDLK    /CALL ADD ROUTINE
MINUS5, -5              /LOCATION UNUSED, SO WE USE
        TAD OPADDR      /IT AS A COUNTER
```

```
        TAD K3          /UPDATE OPERAND ADDRESS
        DCA OPADDR
        ISZ MINUS5      /DONE?
        JMP ALOOP       /NO
        HLT             /YES-ANSWER IN FAC.
FADDLK, FFADD           /POINTER TO ADD ROUTINE
OPADDR, 0               /POINTER TO OPERAND
K3,     3               /EACH OPERAND IS 3 WORDS LONG.
```

FLOATING-POINT OPERATIONS

There are also four simple floating-point operations that operate on the FAC and are available to user subroutines.

Function	Starting Address	Operation
NEGATE	FFNEG	FAC←—FAC
NORMALIZE	FFNOR	NORMALIZE←FAC
SQUARE	FFSQ	FAC←FAC*FAC
CLEAR	FACCLR	FAC←0

These functions are all called by simple JMS, and return with the hardware AC=0. Page 0 links are maintained for negate, normalize, and clear.

Page 0 Link	Routine
FNEGL	FFNEG
FNORL	FFNOR
FCLR	FACCLR

BRTS Subroutines

There are several subroutines in BRTS which can be useful to assembly language functions. A discussion of each of these routines follows. They are identified in the discussion by the tag for their starting address, and all tags referred to can be found in the symbol table.

SUBROUTINE ARGPRE

Subroutine ARGPRE is used to locate scalar variables in the Scalar Table. When called, it uses the rightmost 8 bits (0-255 decimal) of location INSAV as the entry number to be found, and

on return, the data field is set to the field of the variable and the AC points to the exponent word of the variable. ARGPRE is called via a JMS, and is used most often in passing arguments to and from the user subroutine.
Example: Load the FAC with the third variable in the Scalar Table.

```
        CLA
        TAD C2          /WE WAND ENTRY #3,BUT
                        /SINCE THE FIRST ONE IS 0,
                        /LOAD INSAVE WITH 2
        DCA INSAVE
        IAC
        DCA FF          /SET FF SWITCH
        JMS I ARGPRL    /CALL ARGPRE
        JMS I FGETL     /THE AC AND DATA FIELD
        (UNUSED)        /ARE SET, SO THIS IS A
        HLT             /MODE 2 CALL.
C2,     2
ARGPRL, ARGPRE
```

SUBROUTINE XPUTCH

Subroutine XPUTCH is used to put ASCII characters into the terminal ring buffer. When called, the 8-bit ASCII character is in the right-most 8 bits of the AC. On return, the AC is cleared. Note that unless the ring buffer is full, XPUTCH does not cause any characters to be printed; it merely places the character in the terminal ring buffer. A page 0 link to XPUTCH is maintained at location XPUT.
Example: Put a carriage return/line feed combination in the teminal buffer.

```
        CLA             /LOAD CR INTO AC
        TAD K215        /CALL XPUTCH VIA PAGE 0 LINK
        JMS I XPUT      /LOAD LINE FEED INTO AC
        TAD K212        /PUT IN BUFFER
        JMS I XPUT
        HLT
K215,   215             /ASCII CODE FOR CR
K212,   212             /ASCII CODE FOR LF
```

SUBROUTINE XPRINT

Subroutine XPRINT is used to print the next character in the terminal ring buffer. If there are characters waiting to be printed in the ring buffer, XPRINT returns to the instruction following the calling JMS. If the ring buffer is empty, XPRINT skips the instruction following the JMS on returning. XPRINT will actually print a character only if the terminal is not busy; i.e., a call to XPRINT means "print a character if possible" rather than "print a terminal character." For a more detailed description of how the terminal ring buffer I/O operates, see terminal I/O. BRTS maintains a page 0 link to XPRINT called PRINT.

Example: In the middle of a compute bound loop a call placed to XPRINT keeps the terminal busy. At the end of the compute loop a call is placed to XPRINT that empties the terminal ring buffer before proceeding.

```
LOOP,   .
        .                   /COMPUTING INSTRUCTIONS
        .
        JMS I PRINT         /CALL XPRINT VIA PAGE 0 LINK
        NOP                 /THIS INSTRUCTION WILL BE
                            /SKIPPED IF RING BUFFER IF EMPTY
        .
        .
        ISZ LOOPCN          /LOOP CONTROLLING INSTRUCTION
        JMP LOOP            /
        JMS I PRINT         /LOOP IS DONE - EMPTY RING
        JMP .-1             /BUFFER BEFORE CONTINUING
```

SUBROUTINE PSWAP

Under normal conditions, BRTS runs with the OS/8 page 17600 portion of the resident monitor moved to the highest page of core (second highest page TD8/E system). PSWAP is used to swap this page back and forth prior to doing any operations with OS/8. Prior to calling OS/8, PSWAP should be used to restore the

page 17600 resident to 17600, and when OS/8 operations are complete, PSWAP should be called again to swap the 17600 resident back up to high core. A page 0 link to PSWAP is maintained at location P1SWAP.
Example: The following code uses the USR in OS/8 to perform a LOOKUP on the file BASIC.DA on the system device.

```
.
.
.
CLA              /AC SHOULD BE 0 ON CALL
JMS I P1SWAP     /RESTORE OS/8 PAGE 17600 RESIDENT
CLA IAC          /DEVICE # FOR SYS: IS 1
CIF 10
JMS I K7700      /CALL USR
2                /LOOKUP
FNAME            /POINTER TO FILE NAME
0                /CONTAINS LENGTH ON RETURN
HLT              /ERROR RETURN
JMS I P1SWAP     /SWAP OS/8 RESIDENT BACK
.                /TO HIGH CORE
.
.
```

NOTE

If PSWAP is used, it must be executed an *even* number of times. When the assembly language function is called, the page 17600 resident is at high core; when the function returns to BRTS, the 17600 resident must be back in high core.

SUBROUTINE UNSFIX

Subroutine UNSFIX is used to fix a positive, 12-bit, magnitude only integer from the FAC and return with the result in the hardware AC. The range of the fixed integer is 0-4095; an attempt to fix a number larger than 4095 or a negative number will cause an "FO" or "FM" error, respectively. UNSFIX is called via simple

JMS, and a page 0 link to UNSFIX is maintained, called INTL. UNSFIX destroys the contents of the FAC.
Example: The following code uses the FAC as a count of the number of times to ring the bell on the terminal.

```
        .
        .
        .
        CLA
        JMS I INTL      /FIX THE FAC TO 12 BIT INTEGER
        CIA             /NEGATE THE INTEGER
        DCA COUNTR      /AND STORE AS COUNT
BELLOP, TAD K207        /ASCII FOR BELL
        JMS I XPUT      /PUT IN RING BUFFER
        ISZ COUNTR      /RIGHT NUMBER YET?
        JMP BELLOP      /NO-RING ANOTHER BELL
        .
        .
        .
K207,   207
```

SUBROUTINE STFIND

Subroutine STFIND is used to locate a string variable or the first element of a string array. When called, if the link is non-zero, STFIND looks for an entry in the String Array Table. If the link is zero, STFIND uses the String Symbol Table. For standard string variables, the rightmost 8 bits of location INSAV are used as the number of the entry to be obtained; for string array variables the last 5 bits are used. On returns from STFIND, the AC contains a CDF to the field of the string specified, location STRPTR points to the first word (count word) of the string, location STRMAX holds the maximum length of the string (as a negative number), and location STRCNT contains the actual number of characters in the string (as a negative number). STFIND is used most often in passing arguments to and from user functions.

Examples:

1. To find string number 7

```
        TAD K6          /THE NUMBERING STARTS WITH 0
        DCA INSAV       /SET UP STFIND POINTER
        CLL             /WE WANT SIMPLE STRING
        JMS I STGINL    /CALL STFIND
        .
        .
K6,     6
STFINL, STFIND
```

2. To find the first element of string array number 2.

```
        TAD K1          /THE SECOND ENTRY
        DCA INSAV
        CLL CML         /WE WANT STRING ARRAY
        JMS I STFINL    /CALL STFIND
        .
        .
K1,     1
STFINL, STFIND
```

SUBROUTINE BSW

Subroutine BSW is used to swap the two halves of the hardware AC. BSW is called by a simple JMS, and a page 0 link called BSWL is maintained. Obviously, PDP-8/E users would perform a hardware BSW instruction rather than use this subroutine.

SUBROUTINE MPY

Subroutine MPY is a 12 by 12 bit binary multiply routine. The AC is multiplied by the contents of location TEMP3 (both numbers are treated as 12-bit, unsigned integers), and on return, the high-order bits of the result are in TEMP6, and the low-order bits of the result are in the AC. The page 0 link to MPY is MPYLNK.

SUBROUTINE DLREAD

Subroutine DLREAD is used to read the next word of the incore DATA list into the AC. If there is no more data in the DATA list, a DA error message results.

Example: Read the next number from the DATA list into the FAC.

```
        CLA
        JMS I DLREAL    /READ EXPONENT WORD INTO AC
        DCA EXP         /STORE IN FAC
        JMS I DLREAL    /READ HIGH MANTISSA FROM LIST
        DCA HORD        /STORE HIGH MANTISSA WORD
        JMS I DLREAL    /READ LOW MANTISSA FROM LIST
        DCA LORD        /STORE LOW MANTISSA WORD
        .
        .
        .
DLREAL, DLREAD
```

SUBROUTINE ABSVAL

Subroutine ABSVAL is used to take the absolute value of the FAC. If the FAC is positive, ABSVAL is essentially a NOP; if the FAC is negative, it is negated before return.

Passing Arguments To The User Function

BRTS calls the user assembly language function with a JMS instruction. Prior to executing that JMS, it places the first numeric argument in the FAC, the second in Scalar Table entry 0, the third in Scalar Table entry 1, etc., until the argument list is satisfied. If any string arguments are used, the first is found in the SAC and the second is pointed to by String Table entry 0. The user function obtains these arguments as needed by calling the routines ARGPRE and STFIND appropriately.

Example: The following function takes the first two numeric arguments and performs the operation on them specified in A$:

```
UDEF EXM (X,A$, Y)
LET Z=EXM (2,"PLUS",1)
```

Legal values for A$ are strings beginning with "PL" for "PLUS" and "MI" for "MINUS".

```
EXM,    0               /ENTRY POINT
                        /INDEX REGISTER 5 POINTS TO SAC
        TAD I SACP      /GET FIRST 2 CHARS OF A$ FROM SAC
        TAD PL          /COMPAR THEM TO "PL"
        SZA CLA
        JMP EMINUS      /NOT "PLUS"-CHECK FOR "MINUS"
        DCA INSAV       /OPERATION IS PLUS-INIT ARGRE TO GET
                        /SCALAR 0
        JMS I ARGPRL    /FIND Y, X IS ALREADY IN FAC
        JMS I FADDL     /X+Y
ARGPRL, ARGPRE          /THIS LOC SKIPPED BY FADD
        JMP I EXM       /DONE-RETURN WITH RESULT IN FAC
EMINUS, TAD I SACP      /FIRST TWO CHARS OF SAC AGAIN
        TAD MI          /COMPAR TO MI
        SZA CLA         /IS IT "MINUS"?
        JMP I IAL       /NO-ERROR
        DCA INSAV       /YES-SET UP ARGPRE FOR ENTRY 0
        JMS I ARGPRL    /FIND Y, X IS ALREADY IN FAC
        JMS I FSUBL     /X-Y
SACP,   SAC             /THIS LOC SKIPPED BY FSUB
        JMP I EXM       /RETURN WITH VALUE IN FAC
PL,     -2014
MI,     -1511
FADDL,  FFADD
FSUBL,  FFSUB
IAL,    IA
```

If the function is to return any value, that value should be left in the FAC on return. The user function must always return by a JMP I through the entry point.

To generate a fatal IA (illegal argument) error message, perform a JMP to location IA in BRTS.

USING THE USE STATEMENT

If the assembly language function needs to know the location of an array (for buffer space, multiple argument passing, array argument), the USE statement is necessary. The USE statement places the octal number for the array specified into location USECON. By using this value as an index into the Array Symbol Table, the array specified can be located and used by the assembly language function, as necessary.

For example: The hypothetical assembly language function PLT requires a 100 word buffer. To assure allocation of this buffer, the BASIC user of PLT is instructed to dimension a 34 element array and use it in a USE statement before calling the PLT function. In BASIC:

```
10 REM DEFINE THE USER FUNCTION
20 UDEF PLT (X,Y)
30 REM ALLOCATE A 34 ELEMENT (102 WORDS) ARRAY FOR A BUFFER
40 DIM B(34)
     .
     .
```

The function PLT finds B as follows:

```
     .
100 USE B
110 Y=PLT(3,2.8)
     .
     .
     .
```

```
PLT,    0
        TAD USECON      /GET ENTRY NUMBER OF B
        CLL TRL         /MULTIPLY BY 4 (EACH ARRAY TABLE ENTRY
                        /IS 4 WORDS LONG)
        TAD ARSTRT      /MAKE POINTER INTO ARRAY TABLE
        DCA XR5         /AND SAVE IT
        TAD CDFIO       /GET CDF TO SYMBOL TABLE FIELD
        DCA /+1         /PUT INTO LINE
        .               /CHANGE DF TO SYMBOL TABLE FIELD
        TAD I XR5       /GET POINTER TO B(0)
        DCA BPTR        /SAVE FOR LATER
        TAD I XR5       /GET ARRAY DIMENSION 1
        DCA DIM1
        TAD I XR5       /GET ARRAY DIMENSION 2
        .
        .
        .
```

Note that the USE statement merely passes an array entry number to the assembly language function; all actual parameters must be obtained from the Array Symbol Table using that entry number as an index. Note also that the physical location of arrays passed in such a fashion can be almost anywhere in core, and a field boundary may fall within the array.

BRTS I/O

TERMINAL I/O

BRTS drives the terminal asynchronously by maintaining a 40 character terminal output buffer and regularly calling subroutine XPRINT. The procedure is as follows:

1. Characters are inserted into the terminal ring buffer by calling subroutine XPUT. If the ring buffer is full, XPUT waits until a character is printed and a slot is free.
2. BRTS regularly (at least once every pseudo-instruction) calls subroutine XPRINT. XPRINT acts as follows:
   a. If the terminal flag is not set, XPRINT returns.
   b. If the terminal flag is set, the buffer is checked for any more characters. If it is not empty, the next character is printed (via TLS) and XPRINT returns.

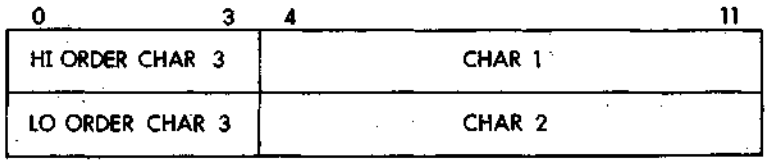
XPRINT returns to the instruction following the calling JMS if there are characters waiting in the ring buffer. If the ring buffer is empty, XPRINT skips the next instruction before returning. This scheme allows BRTS to do other things for most of the 100 milliseconds necessary to print a character, without turning the interrupt on. It requires periodic calls to XPRINT, but the overhead is still considerably less than the time wasted waiting for the terminal flag.

Assembly language functions are free to use the ring buffer (it is emptied before the function is called), or they may choose to perform simple terminal I/O (TLS, TSF, JMP.−1). If terminal I/O is performed other than ring buffered I/O, the terminal flag must be set when the user function returns to BRTS. Note that the assembly language function need not call XPRINT; it may choose to place characters in the ring buffer, and let BRTS empty it after the assembly language function has returned.

BRTS FILE FORMATS

BASIC files are formatted as follows:

1. Numeric files—Numeric files are formatted as consecutive 3 word floating-point numbers, 85 to each 256 word OS/8 block. The last word in each block is unused. There is no end-of-file marker.
2. ASCII Files—ASCII files are stored in OS/8 ASCII format, that is, 3 8-bit characters packed to every two words as follows:



The end of the file is marked with a CTRL/Z character.

BRTS BUFFER SPACE

Locations 10000-12000 in BRTS are devoted to file buffer space. Buffers are allocated as they are needed, the lowest free buffer always being allocated first. A map of currently allocated

buffers is maintained on page 0, called BMAP. Bits in the map are on if the buffer is allocated, off if the buffer is free. Bit 11 represents the buffer from 10000-10377, Bit 10 for 10400-10777; Bit 9 for 11000-11377; and Bit 8 for 11377-11777. If any of the buffers are not available because the pseudo-code or variable space extends below 12000, the corresponding BMAP bits are set when BRTS is started.

BRTS DEVICE DRIVER SPACE

Locations 7000-7577 are devoted to space for OS/8 device drivers. Both one and two page handlers may be loaded; a map of these three pages is maintained at DMAP. Page 7000-7177 is represented in Bit 11; Page 7200-7377 in Bit 10, and 7400-7577 in Bit 9.

Note that assembly language functions that are used in programs which do not require more than one or two files open at once may wish to use some of this driver and buffer space for their own purposes. This space can be allocated by setting appropriate bits in BMAP and DMAP. After the bits are set, BRTS will not use this space in subsequent FILE commands.

THE BRTS I/O TABLE

BRTS keeps track of the status of each of the up to four files which may be open simultaneously by means of the I/O table. Starting at FILE1, it has four 13-word entries, labeled FILE1, FILE2, FILE3 and FILE4. Each name corresponds to the number specified in the file statement which opened that file, and the format of each entry is as follows:

HEADER WORD
STARTING ADDRESS OF BUFFER (IN FIELD 1)
CURRENT BLOCK IN BUFFER
READ/WRITE POINTER INTO BUFFER
HANDLER ENTRY POINT
STARTING BLOCK NUMBER FOR FILE
ACTUAL FILE LENGTH
MAXIMUM FILE LENGTH
POSITION OF PRINT HEAD (FOR COLUMN FORMATTING)

FILE NAME
FILE NAME
FILE NAME
FILE NAME

The header word bits have significance as follows:

Bit Positions	Meaning
0-3	OS/8 number for device
4-5	Current character number for unpacking ASCII files
6	0 if the current buffer load has not been changed 1 if current buffer load has been altered
7	0 if device is file structured 1 if device is read/write only
8	0 if the handler is 1 page long 1 if it is a 2 page handler
9	0 if file is fixed length 1 if variable length
10	0 if more data in file 1 if EOF has been seen
11	0 if file numeric 1 if file ASCII

Interfacing The Assembly Language Function To BRTS

All assembly language functions are subroutines, called by a JMS through the User Function Table. This table, which begins at location 1560 in BRTS, contains absolute pointers to the starting addresses of each of the user assembly language functions. User functions must be origined to run between 3400 and 4577, and must return to BRTS via a JMP I through the user function starting address. To interface a set of user functions to BRTS, perform the following operations:

1. Assemble all the user assembly language functions (up to 16_{10}) together. They must fit between 3400 and 4577, but may be anywhere within that space.

```
.R PAL8
*USER.BN←USER.PA
```

2. Load the user functions into core with the Absolute Loader, and save locations 3400-4577 as the file BASIC.UF, which is the user overlay.

   .R ABSLDR
   *USER.BN$
   .SAVE SYS:BASIC.UF 3400-4577

3. Using OS/8 ODT, modify the User Function Table in BRTS which starts at 1560, entering pointers to the user assembly language functions. Unmodified table entries are 240_8; replace these entries with the starting addresses (pointers) to the user assembly language function. Starting at location 1560, enter the pointers in the table in the exact corresponding order in which the functions appear in the UDEF statement which defines them.

   .GET SYS:BRTS
   .ODT
   1560/240 3400 (LF)
   1561/240 3410
   ↑C
   .SAVE SYS:BRTS

   On the procedure above two functions are interfaced which start at locations 3400 and 3410 respectively. LF indicates pressing the LINE FEED key.

Example: There are three assembly language functions in our package, called PLT, HI, and LO. The BASIC user is instructed that when he uses this function package, PLT, HI and LO must be defined in that order. The function files, then, look like:
Function Source (USER.PA)

```
        *3400
HLT,    0                   /ENTRY POINT FOR HI
        .                   /ORDER OF ENTRY POINTS IS
        .                   /NOT CRITICAL
        .
        .
        JMP I HI
```

```
PLT,    0               /ENTRY POINT FOR PLT
        .
        .
        JMP I PLT
LO,     0               /ENTRY POINT FOR LO
        .
        .
        JMP I LO
```

To enter these three functions into the User Function Table in BRTS, the procedure is:

```
.GET SYS:BRTS
.ODT
1560/240   PPPP    (LF)
1561/240   HHHH    (LF)
1562/240   LLLL
↑C
.SAVE SYS:BRTS
```

where PPPP, HHHH, and LLLL represent octal starting addresses for PLT, HI, and LO respectively. LF indicates pressing the LINE FEED key.

NOTE

BRTS establishes calls to the user function by setting up a one-to-one correspondence between the pointers at 1560 and the function names present in a UDEF statement. Therefore, the order of the pointers must exactly correspond to the order of the function definitions in UDEF. If the BASIC user wants to use only the n^{th} function in a given user package, he must still define n functions in the UDEF statement, though the first n-1 may be dummies.

For example: A package of eight assembly language functions that use arguments as follows:

```
ONE (X)
TWO (X,Y)
THR (X,Y,Z)
FOU (X,Y,Z,A)
FIV (X,Y,Z,A,A$)
SIX (X,Y,Z,A,A$,B$)
SEV (X)
EIG (Y)
```

If a BASIC user wishes to use only function ONE, the UDEF would look like:

```
10 UDEF ONE (X)
```

If the BASIC user wants to use functions ONE and EIG, the UDEF would look like:

```
10 UDEF ONE(X),DUA(D),DUB(D),
DUC(D),DUD(D),DUE(D),DUF(D),
EIG(Y)
```

For this user, DUA through DUF are dummy user function names which are never called; they merely set up the right correspondence between names and pointers.
The easiest way to assure that the pointers are established correctly is to provide the user of an assembly language function package with a set of complete UDEF statements that define all functions correctly, and instruct him to use the complete set of UDEF's each time.

General Considerations And Hints

ROUTINES UNUSABLE BY ASSEMBLY LANGUAGE FUNCTIONS

Because the assembly language functions reside in the overlay buffer when executing, they cannot use any routines that reside in any of the three other overlays. Following is a list of the BRTS functions and routines that are not callable from the assembly language function overlay.

Routine Name	Function
FFATN	Arctangent Function
FFCOS	Cosine Function
FFEXP	Exponential Function (e^x)
EXPON	Power Function (A^B)
INT	Signed integer Function
FFLOG	Naperian log Function
SGN	Sign Function
FFSIN	Trigonometric Sine Function
RND	Random Number generator
FROOT	Square root Function
ASC	String Function ASC
CHR	CHR$ Function
DATE	DAT$ Function
LEN	String length Function
POS	String search Function
SEG	String segmenting Function
STR	STR$ Function
VAL	VAL Function
TRC	Trace Function

Routine Name	Function
CHAIN	File manipulation Routines
CLOSE	
OPENAF	
OPENAV	
OPENNF	
OPENNV	

USING OS/8

So long as the assembly language function is carefully designed to protect all core areas being used by BRTS, there are no restrictions on the function's use of OS/8. Once the page 17600 resident monitor has been restored, the OS/8 User Service Routine (USR) may be called at will, and files may be located, used, and closed again. If the user's BASIC program does not need full file capabilities, the assembly language function is free to use the driver space from 7000-7577 and the buffer space from 10000-17777. The assembly language function should be careful, however, to check the bit maps and status words on page 0 to make certain a given

area is free before using it. Note that the system device driver may be used without restoring the page 17600 resident: restoration is only required when it is desired to use the USR.

OS/8 runs with the interrupt off, and as an OS/8 based program, BASIC does not use the interrupt facility. Locations 0-2, however, in BRTS have been left free for any assembly language functions which wish to use the interrupt facility. Prior to turning on the interrupt system, an assembly language function must clear all the flags left around by the OS/8 handlers, and before returning the function must turn off the interrupt system and set the TTY flag.

PAGE 0 USAGE

Following is a map of the BRTS page 0 usage. Locations marked with an * may be used by the assembly language function without saving the contents.

Locations

0-2 *	Interrupt vector
3-7	System parameters and temps
10-15 *	Index registers
16-17	System pointers
20-30	Compiler-BRTS communication
30-36	System registers
37-62	Floating-point package area
63-67	System registers
73-107	Constants
110-161	Links to BRTS subroutines
162-177	I/O Table pointers

Assembly language functions are free, of course, to use any of the pointers or constants, but they must be intact when control is returned to BRTS.

Assembly Language Function Examples

To illustrate the material in the previous sections, two examples of a complete assembly language function follow. The first example, which we call PLY, evaluates a second degree polynomial of the form $ax^2 \pm bx \pm c$. It requires six arguments: X,A,R1$,B,R2$,C. X is the variable; A,B and C are the coefficients, and R1$ and R2$ are the signs of the first order and constant term, respectively.

Legal values for R1$ and R2$ are "+" and "−"; any other string causes an error message to be printed. Note that this example is tutorial in nature; it is meant to illustrate argument passing and floating-point operations rather than a typical use for assembly language functions.

The second example, ADC, is an example of an actual assembly language function which samples an AD8-EA analog to digital converter. It was written as part of a user function package intended for laboratory use.

Example 1: PLY

In BASIC:

```
10 UDEF PLY (X,A,R1$,B,R2$,C)
      .
      .
      .
1000 PRINT PLY (Z,1,"+",2,"+",1)
```

The pointer for PLY is 3460 and the function is coded as follows:

```
/LOCATIONS IN BRTS
/VALUES WERE OBTAINED FROM SYMBOL

        XR1=10
        XPUT=122
        FFADD=6000
        FFSUB=6117
        FFMPY=5600
        STRLEN=32
        SAC=321
        INSAV=64
        STFIND=1666
        STRCNT=71
        STRPTR=72
        FPUTL=135
        FGETL=134
        ARGPRE=307
        FFSQ=6347

        *1560
        PLY             /LINK TO FUNCTION FOR BRTS

        *3400           /ASSEMBLY LANGUAGE FUNCTION OVERLAY STARTS
                        /AT 3400

/SUBROUTINES USED BY THE FUNCTION PLY

/SUBROUTINE SEVAL=USED TO DETERMINE IF LEFT HALF OF AC CONTAINS
/A "+" OR "-". IF "+" , LOADS THE OCTAL EQUIVALENT OF "JMS I FADDL"
/INTO THE AC AND RETURNS. IF "-" , LOADS THE EQUIVALENT OF
/JMS I FSUBL INTO THE AC AND RETURNS. IF THE CHARACTER IS EQUAL
/TO NEITHER, IT TYPES AN ERROR MESSAGE ON THE TERMINAL AND EXITS
/FROM THE FUNCTION

SEVAL,  0
        DCA SCNTV       /SAVE AC IN TEMPORARY
        TAD SCNTV
        AND K7700       /STRIP TO LEFT CHARACTER
```

```
        TAD MPLUS       /COMPARE TO "+"
        SZA CLA         /IS IT A "+"?
        JMP MC          /NO-CHECK FOR A "-"
        TAD JMSADD      /YES-LOAD AC WITH "JMS I FADDL"
        JMP I SEVAL     /RETURN
MC,     TAD SCNTV       /GET ORIGINAL AC VALUE AGAIN
        AND K7700       /STRIP TO LEFTMOST CHAR
        TAD MMINUS      /COMPAR TO "-"
        SZA CLA         /IS IT "-"?
        JMP SERR        /NO-AN ILLEGAL STRING ARGUMENT HAS BEEN PASSED
        TAD JMSSUB      /YES-LOAD AC WITH "JMS I FSUBL"
        JMP I SEVAL     /AND RETURN
SERR,   TAD EMESSA      /ADDRESS OF ERROR MESSAGE -1
        DCA XR1         /INDEX REGISTER 1 POINTS TO ERROR MESSAGE
ELOOP,  TAD I XR1       /GET CHAR OF ERROR MESSAGE
        SNA             /IS IT 0? (0 TERMINATES MESSAGE)
        JMP I PLY       /YES-FUNCTION COMPLETE-RETURN TO BRTS
        JMS I XPUT      /NO-PUT CHARACTER IN RING BUFFER
        JMS ELOOP

JMSADD, JMS I FADDL
JMSSUB, JMS I FSUBL

MPLUS,  -5300
MMINUS, -5500

EMESSA, EMESS-1
EMESS,  311             / I
        314             / L
        314             / L
        305             / E
        307             / G
        301             / A
        314             / L
        240             /
        323             / S
        311             / I
        307             / G
        316             / N
        215
        212             /CR,LF
        0               /TERMINATES MESSAGE

/SUBROUTINE SCNTV-MAKES CERTAIN THE AC IS = -1. IF NOT, THE STRING
/WHOSE COUNT IS IN THE AC IS LONGER THAN 1 CHARACTER, MAKING IT
/AN ILLEGAL SIGN SPECIFICATION. AND THE ABOVE ERROR MESSAGE IS
/PRINTED. IF SO, IT CLEARS THE AC AND RETURNS.

SCNTV,  0
        IAC             /BUMP AC BY 1
        SZA CLA         /SHOULD NOW BE 0
        JMP SERR        /IT IS NOT, SO PRINT ERROR MESSAGE
        JMP I SCNTV     /RETURN

/THE BODY OF THE FUNCTION ITSELF

PLY,    0               /ENTRY POINT
        TAD STRLEN      /GET LENGTH OF THE STRING IN THE SAC (R1$)
        JMS SCNTV       /MAKE SURE IT IS ONLY 1 CHARACTER LONG
        TAD I SACP      /GET THAT 1 CHARACTER FROM THE SAC
        JMS SEVAL       /GET THE CORRESPONDING ROUTINE CALL INTO THE AC
        DCA OP1         / AND PLACE IT IN LINE
        DCA INSAVE      /SET UP STFIND TO GET STRING TABLE ENTRY 0(R2$)
        CLL             /WE WANT REGULAR STRINGS, NO STRING ARRAYS
        JMS I STFINL    /FIND STRING ARGUMENT 2 (R2$)
        DCA STCDF       /PUT THE CDF IN LINE WHERE ITS NEEDED
        TAD STRCNT      /GET COUNT FOR STRING R2$
        JMS SCNTV       /MAKE SURE IT'S -1
STCDF,  .               /CHANGE DF TO FIELD OF STRING R2$
        ISZ STRPTR      /BUMP STRING POINTER PAST COUNT WORD
        JMP NOSKIP      /IF ISZ DOESN'T SKP,THE STRING IS WITHIN 1 FIELD
                        /IF ISZ DOES SKP, THERE IS A FIELD BOUNDARY
```

```
                        /WITHIN THE STRING AND WE MUST BUMP THE DATA
                        /FIELD BY 1
        RDF             /READ CURRENT DATA FIELD
        TAD CDF10       /MAKE A CDF INSTRUCTION,INCREASING OF BY 1
        DCA .+1         /STORE THE NEW CDF IN LINE
        .               /CHANGE BY 1
NOSKIP, TAD I STRPTR    /GET CHARACTER IN STRING
        CDF
        JMS SEVAL       /AND GET THE OPERATION IT REPRESENTS
        DCA OP2         /PLACE THAT OPERATION IN LINE
        JMS I FPUTL     /SAVE X,WHICH WAS IN THE FAC ON CALL
         X
        JMS I FFSQL     /X**2
        DCA INSAV       /ARG A CAN BE FOUND AT SCALAR TABLE ENTRY 0
        JMS I ARGPRL    /FIND THE ADDRESS OF A
        JMS I FMPYL     /USE IT AS THEADDRESS TO PASS TO THE MULTIPLY
                        /ROUTINE
ARGPRL, ARGPRE          /THIS WORD IS SKIPPED BY MULTIPLY ON RETURN
        JMS I FPUTL     /SAVE A*X**2
        AXSQD           /
        IAC
        DCA INSAV       /THE THIRD NUMERIC ARGUMENT (B) IS FOUND AT
                        /SCALAR TABLE ENTRY 1
        JMS I ARGPRL    /FIND THE ADDRESS OF B
        JMS I FGETL     /AND USE IT IN A MODE CALL TO FGET
CDF10,  CDF 10          /THIS LOCATION IS SKIPPED BY FGET ON RETURN
        JMS I FMPYL     /B*X
         X
        JMS I FPUTL     /SAVE B*X
         BX
        JMS I FGETL     /LOAD FAC WITH A*X**2
        AXSQD
OP1,    .               /THIS LOCATION IS LOADED WITH A CALL TO THE ADD
        BX              /OR SUBTRACT ROUTINE, DEPENDING ON THE VALUE
                        /OF R1$, IN EITHER CASE, THE OPERAND IS B$
        IAC
        CLL RAL         /2 IN THE AC
        DCA INSAV       /THE FOURTH NUMERIC ARGUMENT (C) IS FOUND
                        /IN SCALAR TABLE ENTRY 2
        JMS I ARGPRL    /FIND THE ADDRESS OF C
OP2,    .               /THIS LOCATION IS LOADED WITH A CALL TO ADD
                        /OR SUBTRACT, DEPENDING UPON THE VALUE OF R2$
                        /IN EITHER CASE, IT IS A MODE TWO CALL WITH
                        /THE OPERAND BEING C
K7700,  7700            /THIS INSTRUCTION IS SKIPPED ON THE RETURN
        JMP I PLY       /RETURN FROM THE FUNCTION-ANSWER IS IN THE FAC

FADDL,  FFADD
FSUBL,  FFSUB
FMPYL,  FFMPY
SACP,   SAC
STFINL, STFIND
FFSQL,  FFSQ

/FLOATING POINT STORAGE

X,      0
        0
        0

BX,     0
        0
        0

AXSQD,  0
        0
        0
        0
        $
```

Example 2: ADC

Assembly language function ADC comes from a laboratory package of 11 routines, of which ADC is the eighth. ADC accepts one argument, the number of the channel to be sampled, and returns the channel reading.

In BASIC:

```
10 UDEF DUA(D),DUB(D),DUC(D),DUD(D),DUE(D),DUF(D),DUG(D),ADC(N)
20 REM SINCE ADC IS EIGHTH FUNCTION IN PACKAGE,
30 REM IT MUST BE EIGHTH FUNCTION IN UDEF
40 REM FIRST SEVEN ARE DUMMIES
30  Y=ADC(0)
40 PRINT Y
50 END
```

Modify the User Function Table in BRTS as follows:

```
.GET SYS BRTS

.ODT

1560/0240
01561 /0240
01562 /0240
01563 /0240
01564 /0240
01565 /0240
01566 /0240
01567 /0240 4416
↑C
.SAVE SYS:BRTS

.
```

ADC is coded as follows:

```
        *4416
        INTL = 114      /BRTS LOCATIONS
        EXP = 44        /TAKEN FROM
        HORD = 45       /SYMBOL TABLE
        LORD = 46
        FNORL = 136
ADC,    0               /ENTRY POINT
        JMS I INTL      /FIX ARG TO INTEGER
        ADLM            /LOAD MULTIPLEXER
        ADST            /START CONVERSION
        ADSK            /WAIT FOR COMPLETION FLAG
        JMP .-1
        ADRB            /READ VALUE
        DCA EXP
        JMS FFLOT       /CONVERT TO FLOATING POINT
        JMP I ADC       /RETURN WITH READING IN FAC
/AD8-EA IOTS
        ADLM = 6531     /LOAD MULTIPLEXER FROM AC
        ADST = 6532     /START CONVERSION
        ADRB = 6533     /READ A/D BUFFER
        ADSK = 6534     /SKIP ON A/D DONE

/ROUTINE TO CONVERT 12-BIT INTEGER IN EXP INTO
/FLOATING POINT NUMBER IN FAC

FFLOT,  0
        TAD EXP
        DCA HORD        /PUT NUMBER IN HI MANTISSA
        DCA LORD        /ZERO LOW MANTISSA
        TAD K13
        DCA EXP         /INITIALIZE EXPONENT TO 11(10)
        JMS I FNORL     /NORMALIZE
        JMP I FFLOT     /RETURN
K13,    13
```

Table 6-1 OS/8 BASIC Language Summary

ELEMENTARY OS/8 BASIC STATEMENTS

Statement	Example Form	Meaning
CHAIN	CHAIN DEV: Filename.ex	Stops execution of the current program, retrieves the program named in the CHAIN statement from the specified device and file, compiles the chained program and begins execution of the program.
DATA	DATA x1,x2,...,xn	Values x1 through xn are to be associated with corresponding variables in a READ statement. The values may be either numerics or strings. Strings must be enclosed by quotation marks.
DEF	DEF FNB(x) = f(x) DEF FNB(x,y) = f(x,y)	The user may define his own function 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 argument list in the function may contain as many as 4 numeric and 2 string arguments.
DIM	For numerics: DIM v1(n1), v2(n2,m2)	Enables the user to create a table or array with the specified number of elements where v is the variable name and n and m are the maximum subscript values. Any number of arrays can be dimensioned in a single DIM statement.

Table 6-1 OS/8 BASIC Language Summary (Cont.)

Statement	Example Form	Meaning
	For strings: DIM v1$(I),v2$(K,L)	I is the length of string variable v1$, K is the number of strings and L is the length of strings of string variable v2$. Strings longer than 8 characters must be dimensioned. String tables are illegal.
END	END	Last statement in the program. Signals completion of the program.
FOR-TO-STEP	FOR v=x1 TO x2 STEP x3	Used to implement loops; the variable v is set equal to the expression x1. From this point the loop cycle is completed following which v is incremented after each cycle by x3 until its value is greater than x2. If STEP x3 is omitted, x3 is assumed to be +1. x3 may also be negative. If x3 is positive and x1>x2, the loop is never executed.
GOSUB	GOSUB x	Allows the user to enter a subroutine at several points in the program. Control transfers to line x.
GO TO or GOTO	GO TO n or GOTO n	Transfers control to line n and continues execution from there.
IF-GOTO	IF f1 r f2 GOTO n	Same as IF-THEN.
IF-THEN	IF f1 r f2 THEN n	If the relationship r between the expressions f1 and f2 is true, transfers control to line n; if not, continues in regular sequence. If f1 and f2 are strings they are compared on the basis of the ASCII numeric value of each character in the string.

Table 6-1 OS/8 BASIC Language Summary (Cont.)

Statement	Example Form	Meaning
INPUT	INPUT v1,v2,...,vn	Causes typeout of a ? to the user and waits for the user to supply the values of the variables v1 through vn.
LET	LET v = f	Assigns the value of the expression f to the variable v. The word LET is optional.
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 terminal value in FOR statement (or v is ⩽ terminal value if increment <0).
PRINT	PRINT al,a2,...,an	Prints the values of the specified arguments, which may be variables, text, or format control characters (, or ;).
RANDOMIZE	RANDOMIZE	Generates new sets of random-numbers.
READ	READ v1,v2,...,vn	Variables v1 through vn are assigned the value of the corresponding numbers or strings in the DATA list.
REM	REM	When typed as the first three letters of a line everything between REM and end of line is ignored to allow typing of remarks within the program.
RESTORE	RESTORE	Sets pointer back to the beginning of the list of DATA values.

Table 6-1 OS/8 BASIC Language Summary (Cont.)

Statement	Example Form	Meaning
RETURN	RETURN	Transfers control to the statement following the last GOSUB.
STOP	STOP	Terminates execution at that point at which the statement is reached in the program.
UDEF	UDEF function name (arguments)	The UDEF statement is used to define the syntax of a call to a user-coded machine language function (function name) with its associated arguments.
USE	USE v1,v2,...,vn	The USE statement identifies lists and arrays referenced by a user-coded machine language function.
OS/8 BASIC FILE STATEMENTS		
CLOSE #	CLOSE #N:	Closes a file N previously opened by a FILE#N statement where N is the numerical expression for the file number.
FILE # FILEV # FILEN # FILEVN #	FILE #n:s FILEV #n:s FILEN #n:s FILEVN #n:s	These statements, respectively, open a fixed length ASCII, variable length ASCII, fixed length numeric, and variable length numeric file, where n has a value of from 0 to 4 and s is a string expression with a value of DEV:FILE.EX.
INPUT #	INPUT #N: v1,v2,...,vn	Reads v1 through vn from file number N.
IF END #	IF END #N THEN n	If an attempt has been made to read or write beyond the last datum in file number N, control passes to line number n.

Table 6-1 OS/8 BASIC Language Summary (Cont.)

Statement	Example Form	Meaning
PRINT #	PRINT #N: a1,a2,...,an	Writes the values of the arguments into file number N.
RESTORE #	RESTORE #N	Sets pointer back to beginning of file number N.
OS/8 BASIC CONTROL COMMANDS		
BYE	BYE	Exits from BASIC and returns control to Keyboard Monitor.
CTRL/C	CTRL/C	Stops execution of program and returns control to OS/8 BASIC editor. In editor mode returns control to OS/8 Keyboard Monitor.
CTRL/O	CTRL/O	Stops the listing of text and returns control to BASIC editor.
LIST LI	LIST LI	Lists program with heading.
LIST n LI n	LIST n LI n	Lists program starting from line n, with heading.
LISTNH LISTNH n	LISTNH LISTNH n	Same as LIST and LIST n, but heading suppressed.
NAME NA	NAME FILE.EX NA FILE.EX	This statement renames the current program in user core.
NEW NE	NEW FILE.EX NE FILE.EX	Used to name a program to be created. Performs an inherent SCRATCH.
OLD OL	OLD DEV: FILE.EX OL DEV: FILE.EX	Performs inherent SCRATCH and retrieves a previously created file from the device specified.
RUN RU	RUN RU	Compiles and executes the program currently in core, with heading.

Table 6-1 OS/8 BASIC Language Summary (Cont.)

Statement	Example Form	Meaning
RUNNH	RUNNH	Compiles and executes the program currently in core, with heading suppressed.
SAVE SA	SAVE DEV:FILE.EX SA DEV:FILE.EX	Saves the current program on the device specified.
SCRATCH SC	SCRATCH SC	Deletes all program statements from user core.

OS/8 BASIC FUNCTIONS

Function	Meaning
ABS(X)	This function returns the absolute value of the argument X.
ASC(X)	This function returns the decimal ASCII number (see Appendix A) corresponding to the character X.
ATN(X)	This function calculates the angle (in radians) whose tangent is given by the argument X.
CHR$(X)	X is a numeric expression (modulo 64) which is truncated to an integer. The decimal integer is converted to its equivalent ASCII character (see Appendix A).
COS(X)	The cosine function is used to calculate the cosine of an angle specified in radians.
DAT$(X)	This function returns the data in the form MM/DD/YY. The argument X is a dummy argument.
EXP(X)	This function calculates the value of e(2.71828) raised to the X power.
FNA(X)	Used with a DEF statement to define a user function. Thereafter used as any other function.
INT(X)	This function returns the greatest integer less than the value of the argument X.
LEN(X$)	This function returns the number of characters in string X$.
LOG(X)	The LOG(X) function calculates the natural logarithm of X.

Table 6-1 OS/8 BASIC Language Summary (Cont.)

Function	Meaning
PNT(X)	This function, which can only be used in a PRINT statement, outputs the character whose decimal ASCII value is X. This function is useful for outputting non-printing characters.
POS(X$,Y$,Z)	This function returns the location in string X$ of the first occurrence of string Y$ starting with the Zth character in string X$.
RND(X)	This function returns a random number between 0 and 1.
SEG$(X$,Y,Z)	This function returns the substring of X$ which is between positions Y and Z inclusively.
SGN(X)	The sign function returns the value 1 if X is any positive number, 0 if zero, and −1 if any negative number.
SIN(X)	This function is used to calculate the sine of an angle specified in radians.
SQR(X)	The square root function computes the square root of the absolute value of an expression.
STR$(X)	This function converts the numeric value of X to a string which is its decimal representation.
TAB(X)	This function which can only be used in a PRINT statement, moves the print head to position X.
TRC(X)	This function turns on the trace feature if x = 1 and turns off the trace feature if x = 0. When the trace feature is on, line numbers are printed between percent signs as the lines are encountered in the program. The feature is useful when debugging programs.
VAL(X$)	This function returns the number represented by the string X$ which is the decimal representation of a number.

Compile-Time Diagnostics

Compile-time diagnostic messages typed out by OS/8 BASIC are in the form:

XX YY

where XX is the diagnostic code and YY is the line number at which the error occurred.

Table 6-2 Compile-Time Diagnostics

Diagnostic Code	Explanation
CH	Error in CHAIN statement.
DE	Error in DEF statement.
DI	Error in DIM statement syntax or string dimension greater than 72, or array dimensioned twice.
FN	Error in file number or filename designation.
FP	Incorrect FOR loop parameters or FOR loop syntax.
FR	Error in function arguments or function not defined.
IF	THEN or GOTO missing from IF statement, or bad relational operator.
IO	I/O error.
LS	Missing equal sign in LET statement.
LT	Statement too long (greater than 80 characters).
MD	Line number defined more than once. YY equals line number before line in error.
ME	Missing END statement.
MO	Operand expected, not found.
MP	Missing parenthesis or error in expression within parentheses.
MT	Operand of mixed type or operator does not match operands (e.g., A$=1 and A&2 are both incorrect).
NF	NEXT statement without corresponding FOR statement.
NM	Line number missing after GOTO, GOSUB, or THEN.

Table 6-2 Compile-Time Diagnostics (Cont.)

Diagnostic Code	Explanation
OF	Output file error.
PD	Pushdown stack overflow. Result of either too complex a statement (or statements) or too many nested FOR-NEXT loops.
QS	String literal too long or does not end in quote.
SS	Subscript or function argument error.
ST	Symbol table overflow due to too many variables, line numbers, or literals. Combine lines using backslash (\) to condense program.
SY	System incomplete. System files such as BASIC.SV, BCOMP.SV, and BRTS.SV missing.
TB	Program too big. Condense or CHAIN.
TD	Too much data in program.
TS	Too many total characters in the string literals.
UD	Error in UDEF statement.
UF	FOR loop without corresponding NEXT statement.
US	Undefined statement number. (i.e., statement number mentioned in statement is not in program.)
UU	Incorrect or missing array designator in USE statement.
XC	Extra characters after the logical end of line. (E.g., LET A=B.D—the dot after the B suggests that B is the end of the line and the characters .D appear extraneous.)

Run-time Diagnostics

Run-time diagnostic messages typed out by OS/8 BASIC are in the form:

XX AT LINE YYYYY

where XX is the diagnostic code and YYYYY is the line number at which the error occurred. Most runtime errors stop execution of the program. Those errors which do not stop the program are termed non-fatal (NF) and are indicated below.

Table 6-3 Run-time Diagnostics

Diagnostic Code	Explanation
BO	No more file buffers available.
CI	Inquire failure in CHAIN. Device not found.
CL	Lookup failure in CHAIN. Filename not found.
DA	Attempt to read past end of data list.
DE	Device driver error. Caused by hardware I/O failure.
DO	No more room for drivers. Too many different devices used in file commands.
DV	Attempt to divide by 0. Results is set to zero. (NF)
EF	Logical end of file. Usually caused when I/O device runs out of medium.
EM	Attempt to exponentiate a negative number to a power.
EN	Enter error in opening file. Device is read only or there is already one variable file open on that device or file not found.
FB	FILE busy. Attempt to use a file already in use.
FC	OS/8 error while closing variable file. Device is read only on file closed already.
FE	Fetch error in opening file. Device not found, or device handler too big for available space.
FI	Attempt to close or use unopened file.
FM	Attempt to fix minus number. Usually caused by negative subscripts or file numbers.
FN	Illegal file number. Only 0,1,2,3,4 are legal.
FO	Attempt to fix number greater than 4095. Usually caused by negative subscripts of file numbers.
GR	RETURN without a GOSUB.
GS	Too many nested GOSUBs. The limit is 10.
IA	Illegal argument in UDEF function call.
IF	Illegal DEV:filename specification.
IN	Inquire failure in opening file. Device not found.

Table 6-3 Run-time Diagnostics (Cont.)

Diagnostic Code	Explanation
IO	TTY input buffer overflow. Cause input buffer to be cleared and outputs another ? (NF).
LM	Attempt to take log of negative number or 0.
OE	Driver error while overlaying. Caused by SYS device hardware error.
OV	Numeric or input overflow.
PA	Illegal argument in POS function.
RE	Atempt to read past end of file. (NF)
SC	String too long (greater than 72 characters) after concatenating.
SL	String too long or undefined.
SR	Attempt to read string from numeric file.
ST	String truncation on input. Stores maximum length allowed. (NF)
SU	Subscript out of DIM statement range.
SW	Attempt to write string into numeric file.
VR	Attempt to read variable length file.
WE	Attempt to write past end of file (NF).

OS/8 BASIC System Build Instructions

OS/8 BASIC is distributed on DECtape and paper tape. The DECtape version of OS/8 BASIC contains SAVE images (ready-to-run) for each of the OS/8 BASIC system components as well as binaries for each system component. The paper tape distribution includes binaries for each of the system components. OS/8 BASIC, then, is distributed as the following files:

File	Component	Distributed on:
BASIC.BN	Binary for editor	DECtape and paper tape
BASIC.SV	Editor save image	DECtape only
BCOMP.BN	Compiler binary	DECtape and paper tape
BCOMP.SV	Compiler save image	DECtape only

BLOAD.BN	Loader binary	DECtape and paper tape
BLOAD.SV	Loader save image	DECtape only
BRTS.BN	Run-time system binary (any PDP-8 or PDP-12)	DECtape and paper tape
EAEOVR.BN	Overlay for KE8/E EAE (8/E with KE-8E-EAE)	DECtape and paper tape
BRTS.SV	Run-time system save image (from BRTS.BN)	DECtape only
BASIC.AF	Arithmetic function overlay	DECtape only
BASIC.SF	String function overlay	DECtape only
BASIC.FF	File manipulation overlay	DECtape only

Making SAVE Images from Binary Files:
To create SAVE images of each of the OS/8 BASIC binaries, perform the following OS/8 commands.

1. For the editor:

```
.R ABSLDR
*DEV:BASIC.BN$
.SAVE SYS:BASIC;3011
```

2. For the compiler:

```
.R ABSLDR
*DEV:BCOMP.BN$
.SAVE SYS:BCOMP;7000
```

3. For the loader:

```
.R ABSLDR
*DEV:BLOAD.BN$
.SAVE SYS:BLOAD;7605
```

4. For the run-time system:

```
.R ABSLDR
*DEV:BRTS.BN$          (without KE8/E EAE option)
        or
*DEV:BRTS.BN,DEV:EAEOVR.BN$
                       (PDP-8/E, PDP-8M or
                       PDP-8F with KE-8E EAE)
```

```
.SAVE SYS:BRTS 0-6777
.SAVE SYS:BASIC.AF 3400-4577
.SAVE SYS:BASIC.SF 12000-13177
.SAVE SYS:BASIC.FF 13400-14577
```

NOTE

All BASIC system files must reside on the systems device (SYS:).

5. At this point, BASIC is ready to run.

Assembling the BASIC sources

Instructions for assembling each of the OS/8 BASIC sources follow. PAL-8 (under OS/8) is used, and the descriptions represent OS/8 commands. To assemble OS/8 BASIC, a 12K machine is required.

The OS/8 BASIC sources are named as follows:

NAME.MM

where MM represents the version number. For the first release, the files are named:

Name	Component
BASIC.03	Editor Source
BCOMP.03	Compiler Source
BLOAD.03	Loader Source
BRTS.03	Runtime System Source

1. To assemble the editor:

```
.R PAL8
*DEV:BASIC.BN<DEV:BASIC.03
```

2. To assemble the compiler:

```
.R PAL8
*DEV:BLOAD.BN<DEV:BLOAD.03
```

3. To assemble the loader:

```
.R PAL8
*DEV:BCOMP.BN<DEV:BCOMP.03
```

4. The run-time system source is conditionalized for PDP-8/E with EAE. Assembly instructions for each of the supported configurations follow.

To assemble for PDP-12, PDP-8, PDP-8/I or PDP-8/L, or PDP-8E without EAE, create a source file named NOEAE.PA with EDIT that works as follows:

```
EAE=0
PAUSE
```

Then

```
.R PAL8
*DEV:BRTS.BN<DEV:NOEAE,DEV:BRTS.03/K
```

To assemble the run-time system overlay for PDP-8E, PDP-8F or PDP-8/M with KE-8/E EAE option, prepare a file called EAE.PA that looks as follows:

```
EAE=1
PDP8E=1
PAUSE
```

Then:

```
.R PAL8
*DEV:EAEOVR.BN<DEV:EAE,DEV:BRTS.03/K
```

Optimizing System Performance

There are several steps the OS/8 BASIC user can take to speed up BASIC execution and compilation times, thus speeding up OS/8 BASIC throughput rates.

BYPASSING THE EDITOR

The OS/8 BASIC compiler is constructed such that it will accept any source file for input. Thus, it is possible to execute an already existing BASIC program directly, saving the overhead of an OLD and RUN command to the editor. The format is as follows:

```
.R BCOMP
*DEV:FILE.BA
```

If OS/8 BASIC is used in this fashion, it returns to the OS/8 Monitor on completion, rather than the OS/8 BASIC editor.

Normal Usage	Faster Equivalent
.R BASIC	.R BCOMP
NEW OR OLD—FILE	*FILE
READY	
RUNNH	
READY	

In general, use R BASIC when:

a. Creating new programs or modifying old programs
b. Debugging old programs

Use R BCOMP:

a. To run existing programs
b. In BATCH stream to run BASIC programs

Source files for use by BCOMP must conform to the following rules:

a. There should be no blank lines.
b. Statements must be in the order in which they are to be executed.
c. Line numbers are only required for statements that are referenced in IF, GOSUB, and GOTO statements. In other words, if the only way a statement may be reached is for the preceding statement to be executed, it does not require a line number. In the following example, there are no unnecessary line numbers.

```
      FOR I=1 TO 10
      IF I=2 THEN 400
      PRINT I
      GO TO 410
400   PRINT "TWO".
410   NEXT I
      END
```

Note that the source file can be created in one of two ways: it may be created in the normal fashion with the OS/8 BASIC editor and saved (in which case all lines will contain line numbers), or it may be prepared using any of the other OS/8 editors (EDIT, TECO). In this second case, the user can take advantage of the extra features supported by these sophisticated editors over the OS/8 BASIC editor.

PLACEMENT OF BASIC OVERLAYS ON SYS:

For DECtape system users, the performance of the system can be improved by two simple steps:

a. Use a DECtape drive other than DTA0 for DSK: (via the ASSIGN statement).
b. Place the OS/8 BASIC system files as close together on the SYS tape as possible. The best approach is to make a "BASIC tape" containing only the OS/8 system, PIP, and the BASIC system image files.

Both actions have the effect of speeding up OS/8 BASIC by the simple reduction of the tape motion required for overlaying and compiling.

PLACEMENT OF FUNCTION CALLS WITHIN BASIC PROGRAMS

Most of the BASIC functions and file operations reside in one of the three system overlays. Since the system overlay driver reads in an overlay only if the function desired is not present in the currently resident overlay, overlaying overhead can be reduced by the simple mechanism of placing calls to functions that reside in the same overlay as close as possible in the BASIC program. For example:

```
10 INPUT A$
20 Z$= SEG$(A$,1,6)
30 FILEN #1: Z$
40 INPUT A$
50 Z$= SEG$(A$,1,6)
60 FILEN #2: Z$
```

The above BASIC program uses the first six characters of a string typed by the user as a file name to open a BASIC file. It uses the SEG$ function, a File command, the SEG$ function, and the File command again. Since SEG$ and FILE are in different overlays, the overlayer will be used four times. A faster way to accomplish the same operations follows:

```
10 INPUT A$,B$
20 Z$=SEG$(A$,1,6)
30 X$=SEG$(B$,1,6)
40 FILEN #1: Z$
50 FILEN #2: X$
```

The above only overlays twice, saving considerable time in the program execution. The functions are grouped in the overlays as follows:

Overlay 1 (BASIC.AF):	SIN,COS,ATN,LOG,EXP,RND, SQR,SGN,POWER(A↑B)
Overlay 2 (BASIC.SF):	ASC,CHR$,DAT$,LEN,POS, SEG$,STR$,VAL
Overlay 3 (BASIC.FF):	CLOSE,FILE,FILEN,FILEV, FILEVN

LAB8/E FUNCTIONS FOR OS/8 BASIC

Introduction

The addition of LAB8/E functions to OS/8 BASIC enables the user to solve a range of real-time and pseudo real-time problems using a higher-level language. The benefits of approaching real-time problems using BASIC are numerous. A novice programmer can solve problems with little or no assembly language expertise, and in general, the programming effort required for specific problems is dramatically reduced.

The approach taken for specifying each function was to maximize functional flexibility rather than to stress simplicity. Slaving the computer to external events is accomplished by recognizing Schmitt trigger firings. One of the design goals for the LAB8/E functions was to utilize memory efficiently for single precision and displayable data arrays. Another design goal was to incorporate a masking ability for the recognition of bit patterns when reading digital data. This feature allows easy conversion of decimal data into floating-point format when data is received from decimal devices interfaced to the LAB8/E's digital input registers (DR8-E's).

General Description

This program contains a set of 12 functions which enable a user of OS/8 BASIC to utilize the following peripherals on a LAB8/E: A/D converter, VC8-E display control, DK8-ES real-time clock,

and DR8-EA 12-channel buffered digital I/O. All functions, contained in an overlay called BASIC.UF, reside in the overlay area of BASIC (3400-4577) with the understanding that the entire set of functions is in core whenever a given function is in use. Each function is called by a suitable three-character name, followed by any necessary arguments.

General regulations on arguments passed by the user functions in this package:

1. All arguments must be within the following range:

   $0<\text{ARGUMENT}\leqslant 4095$

   Hence, negative arguments (<0) will cause a fatal error, FM; and positive arguments greater than 4095 (>4095) will cause the fatal error, FO. Fatal errors terminate program execution and return the user to command mode.
2. Furthermore, certain functions in this package require that the arguments be further restricted. These restrictions will be stated along with the discussion of each function later on. Argument errors due to these added restrictions will cause the fatal error, IA (illegal argument).

Preparing Basic for LAB8/E Functions

The Basic Run-Time System (BRTS) provides for one overlay area and divides a set of infrequently used functions into three separate overlays; namely, BASIC.AF, BASIC.SF and BASIC.FF. Since a logical need for user-written assembly language subroutines exists, a last overlay, BASIC.UF was reserved. It is this last overlay that contains the 12 functions for LAB8/E support. Since the subroutines of this last overlay are determined apart from BRTS, it is necessary that BRTS be given a list of core addresses for each of the user subroutines. It is critical that the order of specifying these links or addresses be in the same order that the UDEF statements will appear in the program that calls the functions.

Before writing any program using these functions, it is absolutely necessary to modify BRTS. The following example illustrates how this is done. Notice that in the test programs at the end, the order in the UDEF statements is the same as the ordering of the addresses here. A list of the names of the functions associated with each address is specified to the right for the sake of clarity only.

```
.GET SYS BRTS.SV

.OD

1/****  5402                    used for interrupts
00002 /****  4456
1560/****  3400                  INI
01561 /****  3454                PLY
01562 /****  3473                DLY
01563 /****  3600                DIS
01564 /****  4000                SAM
01565 /****  4100                CLK
01566 /****  3541                CLW
01567 /****  3521                ADC
01570 /****  4400                GET
01571 /****  4432                PUT
01572 /****  4271                DRI
01573 /****  4313                DRO
^C

.SAVE SYS BRTS.SV
```

Since many of BASIC's functions also reside in overlays, the user is cautioned about using a function that will cause the current set of functions to be overlayed and thereby destroy any useful information. For example, the user cannot calculate a set of cosine values and pass them to the PLY function to be stored, because COS resides in BASIC.AF overlay and PLY resides in BASIC.UF.

Definition of LAB/8E Support Functions

Once BRTS has been modified to recognize the user function from the BASIC.UF overlay, BASIC programs making use of these functions may be written. If a program requires the use of the Nth function in the ordered list of links, the first (N-1) functions of the list must be defined by UDEF statements or a set of (N-1) dummy-named functions must precede the defining of the Nth function. For example:

In reference to the ordered list of functions in the previous section, if the ADC function is the only one to be used in a particular BASIC program, the UDEF statements must be:

```
.
.
.
10  UDEF INI(N),PLY(Y),DLY(N),DIS(S,E,N,X)
11  UDEF SAM(C,N,P,T),CLK(R,O,S),CLW(N),ADC(N)
.
```

OR

```
10  UDEF DUA(N),DUB(N),DUC(N),DUD(N)
11  UDEF DUE(N),DUF(N),DUG(N),ADH(N)
```

However, it is recommended that the user always use the complete set of UDEF's each time one requires one or more functions in a program. This is recommended solely to keep careless omissions to a minimum.

INI(N)

The *initialize* function has a twofold purpose. Its main purpose is to locate the address of the array specified by BASIC's USE statement and retain that address until BASIC.UF is overlayed by one of the other three overlays.

A secondary purpose is to set a pointer to the first location of the array. Consequently, an array may be used to store one set of data followed immediately by a second set of data, provided the INI function was called only once. This means that displayable data (10 bits), and fixed point data (12 bits) may share the user array at the user's discretion. If, however, the INI function was again specified at the end of the first data run, the first set of data is overwritten by the second set of data. Hence, INI effectively zeros the array in this case. Whenever an array is to be used in conjunction with one or more of the functions in the BASIC.UF overlay, user first dimensions the array and eventually employs the USE statement before the INI function can have meaning. For example:

```
DIM A(3)
.
.
.
USE A
.
.
```

```
.
X=INI(0)
.
.
.
```

The argument N, for INI, is a dummy argument, and may be any integer; 0, 1, 2, . . .

Whenever the functions PLY, DIS, SAM, GET, and PUT are used, make sure that the INI function has been previously called at least once. When an array is given the dimension N, BASIC allocates (N+1) *floating point* words of memory which is actually 3(N+1) *single memory* locations. Thus, in the example above, BASIC allocates 4 floating point words or 12 single memory locations for the array. Each data value deposited into the user's array by the user functions is a single precision value (uses one memory word).

PLY(Y)

The purpose of the *plot* function is to enable a BASIC program to create y-data values and enter them into the user array sequentially, beginning with the first unused location of the array. Each floating point value is fixed to a ten (10) bit single precision value before it is put into the array. The range of the y-data values must be:

$0 \leqslant y \leqslant 1.0$

This is easily accomplished by inserting a scaling factor. (Refer to line numbers 26 and 64 of the example program TEST0A.PG at the end of this chapter.)

The data in the user array can be displayed as it is being passed to the array (see DLY function) and/or be refreshed continuously once all values have been entered into the array (see DIS function).

DLY(N)

The *delay* function is used only in conjunction with the PLY function. It causes the scope to be refreshed with the contents of the user array after each point is processed, so that the graphical progress of data can be observed.

N is an integer such that $1 \leqslant N \leqslant 1024$. It specifies the maximum number of points to be eventually displayed. Implied here is the fact that the display will contain only the first N points even if the arrays contain more than N points.

DIS(S,E,N,X)

. The *display* function is used to set up parameters for the displaying of y-data stored in the user array. The display will begin with the desired starting point, S of the array and display every Nth point while not exceeding the desired endpoint, E (where N = 1, 2, 3, . . .).

Depending on the value of X, the DIS function has two separate operations. Operation when X equals zero (X=0): Indication is given to the user-overlay-functions that a SAM function will be the next BASIC instruction. Consequently the parameters mentioned above are set up so that exactly one of the sampled channels can be displayed 'on the fly'. To understand the use of the arguments S,E,N,X; it is necessary to know how the A/D data is stored in the user array. For example assume 100 samples/channel in each case:

ARRAY	CASE 1 SAM CH#0	CASE 2 SAM CH #3,4,5
WD1	CH#0	CH#3
WD2	CH#0	CH#4
WD3	CH#0	CH#5
WD4	CH#0	CH#3
WD5	CH#0	CH#4
WD6	CH#0	CH#5
.	.	.
.	.	.
.	.	.
WD100	CH#0	CH#3

To display CASE1, once sampling begins:

DIS(1,100,1,0)

To display CH#4 of CASE2, once sampling begins:

DIS(2,100,3,0)

Operation when X is greater than zero (X>0): A user array of y-data is to be displayed immediately. The display is continually refreshed (no return to BASIC) until the operator types CTRL/N on the keyboard.
Displayable y-data values are assumed to be 10-bit single precision data words.
The x-coordinate for each y-data value is determined by a DELTAX value found as follows:

DELTAX = 1023/|[E-S)/N]

Due to the outcome of DELTAX, the display may not always use the full width of the scope. However, the display is always centered.

S≥1; E≥S; (E-S)/N≤1023. At least one point must be displayed and no more than 1024 points may be displayed.

SAM(C,N,P,T)

The *sample* function is used solely to set up parameters for subsequent sampling of the ADC's or for subsequent sampling of digital input registers (0,1,2) depending on the value of T.

TASK 1 (T=0): Sample the ADC's

C = First channel # to be sampled; $0 \leq C \leq 17_8$.

N = Number of consecutive channels to sample; $1 \leq N \leq (20_8\text{-}C)$.

P = Number of sample points/channel; P=0.

TASK 2 (T=0): Sample digital input registers.

C = First register # to be sampled; 0≤C≤2.

N = Number of consecutive input registers to sample; 1≤N≤(3-C).

P = Number of samples/register; P=0.

Anytime a SAM instruction is used to sample the ADC's, exactly one channel must be displayed on the fly. However, the sampling rate is not slowed down by this requirement. Hence a DIS function call *must* precede a SAM function call whenever TASK 1 is chosen.

It is possible to display digital input data so long as only the least significant 10 bits will be displayed. However, this data can not be displayed 'on the fly' and can only be displayed via the DIS function once all data is in the array.

CLK(R,O,S)

The clock function sets up the clock to be used for A/D sampling, for digital input sampling, or as a simple timing device.

R(rate) = desired frequency at which to run the clock

Value of R	Frequency
1	External input
2	100 HZ
3	1K HZ
4	10K HZ
5	100K HZ
6	1M HZ

O(overflow CNT) = number of clock ticks per interrupt with the clock running at the desired frequency, R. $0 \leqslant O \leqslant 4095$

S (Schmitt trigger) ($S \neq 0$) = Activate all Schmitt triggers and start the clock when any one of the three Schmitt triggers fires. ($S=0$) Do not activate any Schmitt triggers and start up the clock immediately.

As mentioned above, this single clock function is used to set the clock for one of three separate tasks.

TASK1: Sample the ADC's.

The interrupts are turned on[1] and the program waits in the display loop for a clock overflow; at which time the A/D channel(s) is (are) sampled. The display loop will display the data for the channel specified by the user in the DIS function. When all channels have been sampled the requested number of times, the CLK function returns to BASIC.

TASK2: Sample digital input registers.

At each clock overflow, the digital input register(s) is (are) sampled. When all registers have been sampled the requested number of times, the CLK function returns to BASIC.

[1] When interrupts are turned on, the only possible valid interrupts can be caused by the keyboard or the clock. Hence, any other interrupt is an uncontrollable, spurious interrupt (faulty hardware) which will cause a HLT at location 4466. If this happens, do the following:

1. Set SWITCH REGISTER to 4476 and press ADDR LOAD.
2. Next, press the CLEAR and CONT switches to return to BASIC.
3. Type CTRL/C to return to the OS/8 Monitor.

NOTE

The sampled data from the ADC's or the digital input registers is stored sequentially in the user's array.

TASK3: A simple timing device.

The clock is set up and started (unless it is to be started when a Schmitt trigger fires) and then returns to BASIC.

The following illustrates what sequence of instructions are needed for each task.

```
TASK1                TASK2                TASK3
  .                    .                    .
  .                    .                    .
  .                    .                    .
DIM A(n)             DIM A(n)             Z=CLK(R,O,S)
USE A                USE A                  .
  .                    .                    .
  .                    .                    .
  .                    .
W=INI(0)             W=INI(0)
X=DIS(C,N,P,T)       Y=SAM(C,N,P,1)
Y=SAM(C,N,P,0)       Z=CLK(R,O,S)
Z=CLK(R,O,S)           .
  .                    .
  .                    .
  .
```

CLW(N)

With the clock having been set up by CLK as a simple timer, this *clock wait* function, when called, simply returns to BASIC whenever a clock overflow occurs; and/or whenever a Schmitt trigger fires,provided S was a non-zero argument in CLK.

Upon return to BASIC, a number is returned to the caller indicating whether the return was due to a clock overflow, a Schmitt trigger, or a clock overflow and the firing of a Schmitt trigger simultaneously. The number also indicates whether one of the above conditions occurred before or after the CLW function was called. N is a dummy argument (N=0,1,2, . . .) .

The following table illustrates the various numbers returned.

Case 1: Clock overflowed or a Schmitt trigger fired *after* CLW is called.

Overflow only	Schmitt Trigger Only	Simultaneously
0	1 (Trigger 1 fired)	−1
	2 (Trigger 2 fired)	−2
	3 (Trigger 1 & 2 fired)	−3
	4 (Trigger 4 fired)	−4
	5 (Trigger 1 & 4 fired)	−5
	6 (Trigger 2 & 4 fired)	−6
	7 (Trigger 1,2 & 4 fired)	−7

Case 2: Clock overflowed or a Schmitt trigger fired *before* CLW is called.

Overflow only	Schmitt Trigger only	Simultaneously
−8	9	−9
	10	−10
	11	−11
	12	−12
	13	−13
	14	−14
	15	−15

The TEST4A.PG and TEST5A. PG examples make use of the CLW function.

The CLW function has many useful applications. Subroutine timing may be accomplished by starting the clock with a specific rate and overflow count. The subroutine is called, and at the end of the subroutine the CLW function is called to see if an immediate return is obtained. This timing is empirical in that the user would keep changing the rate and/or overflow count until Case 2 occurred. Secondly, Schmitt trigger firing may be used to branch to a particular subroutine or to notify the program to proceed with specific tasks such as reading digital data or sampling an analog input. Thirdly, time interval histograms and post stimulus histograms are also possible (see TST20A.PG).

ADC(N)

This function is issued any time one wishes to sample A/D channel N. The 10 bit data value is floated and returned to the caller for immediate examination. $0 \leqslant N \leqslant 17_8$.

The BASIC statement W=ADC(3) asks that A/D channel #3 be sampled and the floating point value be assigned to W.

The TEST5A.PG example illustrates one use of the ADC function.

GET(M,L)

This function is used to get one 12 bit word from the user array, mask out certain bits and return the result as a floating point number to the caller.

L is Lth location of the user array. Hence, if an array has N single precision words, L can take on meaningful values of 1,2,3, . . . ,N.

NOTE

Although BASIC allows 0 to be a meaningful value in a dimension statement such as DIM A(0), it must be understood that L always begins with 1, where 1 stands for the *first* single-word location of the array. Thus DIM A(0) specifies an array of one floating point word (three one-word locations).

M is a masking number such that $0 \leqslant M \leqslant 4095$. This floating point number is converted to a 12 bit binary number between 0 and 7777. Those bits that are zero will mask out or eliminate those bits in the array value. If M=0, then no masking is done and the 12 bit array value is returned intact. M=0 and M=4095 have the same meaning.

The BASIC statement Y=GET(15,2) gets the second word of the user array, masks out all bits *except* bits 8,9,10,11 and assigns the floating point result to Y. Consequently, if an array is as follows:

single prec WD1	5678	Fl. pt. word 0
single prec WD2	1234	
single prec WD3	4455	

.
.
.

$WD2 = 1234_8 = 001010011100_2$
$MASK = 15_{10} = 17_8 — 000000001111_2$
The 12 bit value after masking is:

$000000001100_2 = 12_{10}$

Hence, Y=12

NOTE

For user assistance in understanding decimal to octal to binary conversions, refer to *Introduction to Programming.*

PUT(M,L)

This function enables a floating point number to be fixed to a single 12 bit word and put into the user's array.

L is Lth location of the user's array. For an array of N single precision words, L can take on meaningful values of 1,2,3, . . . ,N.

M is the floating point number to be fixed and stored in the array. $0 \leqslant M \leqslant 4095$.

NOTE

Both GET and PUT functions imply that a user's array must not exceed 4096 memory locations, because of the general restriction on any argument for these user functions.

The BASIC statement Y=PUT(128,4) means fix 128 to 12 bits ($000\ 010\ 000\ 000_2$) and put the value into the 4th word of the user array. TST15A.PG, TST16A.PG, TST17A.PG and TST18A.PG illustrate the use of functions GET and PUT.

DRI(N)

This function is issued any time one wishes to sample a digital input register, N ($0 \leqslant N \leqslant 2$). The 12 bit digital value is returned to the user as a floating point number. Basic statement: X=DRI(0) means that input register #0 is sampled and the floating point result is assigned to X.

DRO(M,N)

This function is issued any time one wishes to set the bits of a digital output register, N($0 \leqslant N \leqslant 2$). The output register bits are set via the value of M ($1 \leqslant M \leqslant 4095$). If M=0, the output register is

cleared, otherwise the bits of the register remain set. Hence, additional bits of the register can be set while maintaining those set earlier.

Basic statement: Z=DRO(9,1) means set bits 8 and 11 of output register # 1 if not already set.

9_{10} — 000000001001_2

TST13A.PG and TST15A.PG illustrate the use of the DRI and DRO functions.

LAB8/E Examples

The following set of BASIC programs illustrates a number of ways the user functions may be implemented. Each program has been kept as simple as possible.

Note that for TST12A.PG, TST13A.PG and TST15A.PG a battery powered 'black box' was used to interact with the digital I/O registers. The box contained a set of 12 switches which could set any combination of bits for the digital input register, and it also contains a row of 12 lights that were lighted by the contents of the 12 bit digital output register. When running TST18A.PG, use the data from TST17A.PG.

```
1 REM -            PROGRAM NAME:  TEST0A.PG
2 REM -
3 UDEF INI(N),PLY(Y),DLY(N),DIS(S,E,N,X)
4 UDEF SAM(C,N,F,T),CLK(R,O,S),CLW(N),ADC(N)
5 UDEF GET(M,L),PUT(M,L),DRI(N),DRO(M,N)
6 DIM A(342)
9 REM -
10 REM - CALC 1024 PTS & DISPLAY ON FLY.
11 REM - WHEN DONE DISPLAY EVERY 10TH PT.
12 REM -
20 USE A
22 Z=INI(0)
24 FOR N=1 TO 1024
26 Y=(3*N-2)/3071
28 X=PLY(Y)
30 W=DLY(1024)
32 NEXT N
34 V=DIS(1,1024,10,1)
49 REM -
50 REM - CALC 30 PTS & DISPLAY ONLY
51 REM - WHEN DONE.
60 Z=INI(0)
62 FOR N=1 TO 30
```

```
64 Y=(2+N+1)/61.1
66 Z=PLY(Y)
68 NEXT N
70 V=DIS(1,30,1,1)
80 END
```

```
1 REM -          PROGRAM NAME:  TEST1A.PG
2 REM -
3 UDEF INI(N),PLY(Y),DLY(N),DIS(S,E,N,X)
4 UDEF SAM(C,N,P,T),CLK(R,O,S),CLW(N),ADC(N)
5 UDEF GET(M,L),PUT(M,L),DRI(N),DRO(M,N)
6 DIM A(342)
10 REM -
11 REM - SAMPLE CHAN 0 (1024 TIMES);DISPLAY
12 REM - ALL PTS ON THE FLY.
13 REM - 10 INTERRUPTS/SEC
14 REM -
20 USE A
21 W=INI(0)
22 W=DIS(1,1024,1,0)
24 X=SAM(0,1,1024,0)
26 Y=CLK(3,100,0)
28 Z=DIS(1,1024,1,1)
40 REM -
41 REM - SAMPLE CHANNELS 0,1 (100 TIMES EACH).
42 REM - 10 INTERRUPTS/SEC;DISPLAY CHAN 0 WHILE
43 REM - SAMPLING,WHEN DONE SHOW THREE DIFF
44 REM - DISPLAYS: DISPLAY CHAN 0--HIT ↑N DISPLAY
45 REM - CHAN 1--HIT ↑N DISPLAY CHANS 0&1.
50 USE A
51 W=INI(0)
52 W=DIS(1,200,2,0)
54 X=SAM(0,2,100,0)
56 Y=CLK(3,100,0)
58 Z=DIS(1,200,2,1)
60 U=DIS(2,200,2,1)
62 V=DIS(1,200,1,1)
70 END
```

```
1 REM -          PROGRAM NAME:  TEST2A.PG
2 REM -
3 UDEF INI(N),PLY(Y),DLY(N),DIS(S,E,N,X)
4 UDEF SAM(C,N,P,T),CLK(R,O,S),CLW(N),ADC(N)
5 UDEF GET(M,L),PUT(M,L),DRI(N),DRO(M,N)
6 DIM A(342)
10 REM -
11 REM - CALC A PARABOLA OF 601 PTS AND DISPLAY
12 REM - ON THE FLY. WHEN DONE DISPLAY EVERY 10TH
13 REM - PT OF PARABOLA.
```

```
14 REM -
20 USE A
22 Z=INI(0)
24 FOR N=-300 TO 300
26 Y=(N*N)/100000
28 X=PLY(Y)
30 W=DLY(601)
32 NEXT N
34 V=DIS(1,601,10,1)
50 REM -
51 REM - CALC A CUBIC OF 601 PTS & DISPLAY ON FLY
52 REM - WHEN DONE DISPLAY EVERY 10TH PT.
53 REM -
60 Z=INI(0)
62 FOR N=-300 TO 300
64 Y=(N*N*N+27000000)/54000010
66 X=PLY(Y)
68 W=DLY(601)
70 NEXT N
72 V=DIS(1,601,10,1)
80 END
```

```
1 REM -       PROGRAM NAME:  TEST3A.PG
2 REM -
3 UDEF INI(N),PLY(Y),DLY(N),DIS(S,E,N,X)
4 UDEF SAM(C,N,P,T),CLK(R,O,S),CLW(N),ADC(N)
5 UDEF GET(M,L),PUT(M,L),DRI(N),DRO(M,N)
6 DIM A(342)
10 REM -
11 REM - ILLUSTRATE ABILITY TO ACCESS USER BUFFER.
12 REM - PUT NUMBERS 1-10 INTO BUF IN THAT ORDER.
13 REM - & READ THEM OUT IN THE REVERSE ORDER.
14 REM -
20 Z=INI(0)
22 FOR N=1 TO 10
24 PRINT N
26 T=N
28 R=PUT(T,N)
30 NEXT N
32 FOR N=1 TO 10
34 N=11-N
36 P=GET(0,M)
38 PRINT P
40 NEXT N
50 END
```

```
1 REM -        PROGRAM NAME:  TEST4A.PG
2 REM -
3 UDEF INI(N),PLY(Y),DLY(N),DIS(S,E,N,X)
4 UDEF SAM(C,N,P,T),CLK(R,O,S),CLW(N),ADC(N)
5 UDEF GET(M,L),PUT(M,L),DRI(N),DRO(M,N)
6 REM - SAMPLE CHAN 0 IF CLOCK O.F.
7 REM - SAMPLE CHAN 1 IF SCHMITT ONLY
8 REM - SAMPLE CHAN IF BOTH FIRE
9 REM - IF EARLY, TELL USER
10 REM - ROUTINE ALSO OUTPUTS Z
11 X=CLK(3,4000,1)
12 FOR N=1 TO 10
14 Z=CLW(0)
15 PRINT "Z=";Z
16 IF Z=0 GOTO 30
18 IF Z<0 GOTO 24
19 IF Z<8 GOTO 34
20 IF Z=8 GOTO 40
21 GOTO 40
24 IF Z<-8 GOTO 40
26 W=ADC(2)
28 GOTO 36
30 W=ADC(0)
31 GOTO 36
34 W=ADC(1)
36 PRINT W
37 GOTO 42
40 PRINT "EARLY"
42 NEXT N
50 END
```

```
1 REM -        PROGRAM NAME:  TEST5A.PG
2 UDEF INI(N),PLY(Y),DLY(N),DIS(S,E,N,X)
3 UDEF SAM(C,N,P,T),CLK(R,O,S),CLW(N),ADC(N)
4 UDEF GET(M,L),PUT(M,L),DRI(N),DRO(M,N)
5 DIM A(342)
10 REM -
11 REM - USE CLK AS A SIMPLE TIMER.
12 REM - SAMPLE CHAN 0 EVERY 4TH SEC AND PUT VAL TO TTY
13 REM - DO THIS 10 TIMES
14 REM -
20 X=CLK(3,4000,0)
22 FOR I=1 TO 10
24 Y=CLW(0)
26 Z=ADC(0)
28 PRINT Z
30 NEXT I
40 REM -
41 REM - USE CLK AS A SIMPLE TIMER
42 REM - SAMPLE CHAN 1 TEN TIMES & SYNC OFF ANY
```

```
43 REM - SCHMITT TRIGGER
44 REM -
50 X=CLK(4,4000,1)
52 FOR I=1 TO 10
54 Y=CLW(0)
56 Z=ADC(0)
58 PRINT Z
60 NEXT I
70 END
```

```
1 REM -     PROGRAM NAME:  TEST7A.PG
2 REM -
3 UDEF INI(N),PLY(Y),DLY(N),DIS(S,E,N,X)
4 UDEF SAM(C,N,P,T),CLK(R,O,S),CLW(N),ADC(N)
5 UDEF GET(M,L),PUT(M,L),DRI(N),DRO(M,N)
6 DIM A(342)
7 USE A
8 REM - DISPLAY A TRIANGLE
10 Z=INI(0)
12 FOR N=1 TO 30
14 Y=N/30.1
16 W=PLY(Y)
18 Z=1/30.1
20 U=PLY(Z)
22 P=DLY(118)
24 NEXT N
26 FOR N=1 TO 29
27 M=30-N
28 Y=N/30.1
30 W=PLY(Y)
32 Z=1/30.1
34 U=PLY(Z)
36 P=DLY(118)
38 NEXT N
40 V=DIS(1,118,1,1)
42 END
```

```
1 REM -        PROGRAM NAME:   TEST8A.PG
2 REM -
3 UDEF INI(N),PLY(Y),DLY(N),DIS(S,E,N,X)
4 UDEF SAM(C,N,P,T),CLK(R,O,S),CLW(N),ADC(N)
6 DIM A(342)
10 REM -
11 REM - SAMPLE CHAN 0 100 TIMES; DISPLAY;
12 REM - HOWEVER SYNC OFF SCHMITT TRIGS.
```

```
14 REM -
32 USE A
34 W=INI(0)
36 W=DIS(1,100,1,0)
38 X=SAM(0,1,100,0)
40 Y=CLK(3,100,1)
42 Z=DIS(1,100,1,1)
50 END
```

```
1 REM -        PROGRAM NAME:    TEST9A.PG
2 REM -
3 UDEF INI(N),PLY(Y),DLY(N),DIS(S,E,N,X)
4 UDEF SAM(C,N,P,T),CLK(R,O,S),CLW(N),ADC(N)
5 UDEF GET(M,L),PUT(M,L),DRI(N),DRO(M,N)
6 DIM A(342)
10 REM -
11 REM - CALC A PARABOLA OF 401 PTS AND DISPLAY ON FLY
13 REM -
20 USE A
22 Z=INI(0)
24 FOR N=-200 TO 200
26 Y=(N*N)/40001
28 X=PLY(Y)
30 W=DLY(401)
32 NEXT N
50 REM -
51 REM - CALC A CUBIC OF 401 PTS & DISPLAY ON FLY
52 REM - SHOW PARABOLA. WHEN DONE DISPLAY EVERY PT
53 REM - & THEN EVERY 10TH PT
54 REM -
62 FOR N=-200 TO 200
64 Y=(N*N*N+8000000)/16000010
66 X=PLY(Y)
68 W=DLY(802)
70 NEXT N
72 V=DIS(1,802,1,1)
74 V=DIS(1,802,10,1)
80 END
```

```
1 REM -          PROGRAM NAME:    TST10A.PG
2 REM -
3 UDEF INI(N),PLY(Y),DLY(N),DIS(S,E,N,X)
4 UDEF SAM(C,N,P,T),CLK(R,O,S),CLW(N),ADC(N)
5 UDEF GET(M,L),PUT(M,L),DRI(N),DRO(M,N)
6 DIM A(342)
```

```
7 REM - THIS ROUTN RETURNS 4 DIGITS-3BITS/DIGIT
10 USE A
11 Z=INI(0)
12 PRINT "VALUE"
14 INPUT Y
16 Z=PUT(Y,1)
18 P=GET(7,1)
19 PRINT P
20 P=GET(56,1)
21 PRINT P
22 P=GET(448,1)
23 PRINT P
24 P=GET(3584,1)
25 PRINT P
26 GOTO 12
30 END
```

```
1 REM -       PROGRAM NAME:    TST12A.PG
2 REM -
3 REM - THIS ROUTN SAMPLES DIGITAL BOARD
4 REM - #1 TEN TIMES, ONCE EVERY 4 SECS & PUTS
5 REM - THE VALUES INTO USER BUF THEN IT PRINTS
6 REM - OUT THE 10 VALUES
10 UDEF INI(N),PLY(Y),DLY(N),DIS(S,E,N,X)
11 UDEF SAM(C,N,P,T),CLK(R,O,S),CLW(N),ADC(N)
12 UDEF GET(M,L),PUT(M,L),DRI(N),DRO(M,N)
20 DIM A(342)
22 USE A
23 W=INI(0)
24 X=SAM(1,1,10,1)
26 Y=CLK(3,4000,0)
28 FOR N=1 TO 10
30 W=GET(0,N)
32 PRINT W
34 NEXT N
40 END
```

```
1 REM -        PROGRAM NAME:    TST13A.PG
2 REM -
3 REM - TEST THE OUTPUT REG-SEE THE LIGHTS LITE
4 REM - UP .  OCTAL INPUT LIGHTS THE LIGHTS AND
5 REM - THE LAMP? AN INPUT OF 0 CLEARS THE OUTPUT REG
10 UDEF INI(N),PLY(Y),DLY(N),DIS(S,E,N,X)
11 UDEF SAM(C,N,P,T),CLK(R,O,S),CLW(N),ADC(N)
12 UDEF GET(M,L),PUT(M,L),DRI(N),DRO(M,N)
```

```
14 W=DRO(0,1)
16 PRINT "NUMBER"
18 INPUT Y
19 IF Y=0 GOTO 14
20 W=DRO(Y,1)
22 GOTO 16
30 END
```

```
1 REM -       PROGRAM NAME:   TST15A.PG
2 REM -
3 UDEF INI(N),PLY(Y),DLY(N),DIS(S,E,N,X)
4 UDEF SAM (C,N,P,T),CLK(R,O,S),CLW(N),ADC(N)
5 UDEF GET(M,L),PUT(M,L),DRI(N),DRO(M,N)
6 DIM A(342)
7 REM - THIS ROUTN RETURNS 3 DIGITS-4 BITS/DIGIT
8 REM - (MASKING) IT FIRST OUTPUTS THE DECIMAL
9 REM - EQUIV OF THE NUMBER
10 USE A
11 Z=INI(0)
12 W=DRI(1)
13 PRINT W
16 X=PUT(W,1)
18 P=GET(15,1)
19 PRINT P
20 P=GET(240,1)
21 PRINT P
22 P=GET(3840,1)
23 PRINT P
24 PRINT "WASTE TIME"
25 INPUT R
26 GOTO 12
30 END
```

```
1 REM -       PROGRAM NAME:  TST16A.PG
2 REM -
3 UDEF INI(N),PLY(Y),DLY(N),DIS(S,E,N,X)
4 UDEF SAM(C,N,P,T),CLK(R,O,S),CLW(N),ADC(N)
5 UDEF GET(M,L),PUT(M,L),DRI(N),DRO(M,N)
6 DIM A(3)
7 REM - THIS ROUTN SHOWS THAT ANY N;0<=N<=4095
8 REM - PUT INTO A USER BUF IS RETURNED AS THE
9 REM - SAME VALUE.
10 USE A
11 Z=INI(0)
```

```
12 PRINT "NUMBER"
14 INPUT Y
16 X=PUT(Y,1)
18 Z=GET(0,1)
20 PRINT Z
26 GOTO 12
30 END
```

```
1 REM -      PROGRAM NAME:    TST17A.PG
2 REM - FILL AN ARRAY OF 30 WORDS WITH THE
3 REM - FIRST 30 INTEGERS. WRITE THE ARRAY
4 REM - OUT TO DECTAPE.
5 UDEF INI(N),PLY(Y),DLY(N),DIS(S,E,N,X)
6 UDEF SAM(C,N,P,T),CLK(R,O,S),CLW(N),ADC(N)
7 UDEF GET(M,L),PUT(M,L),DRI(N),DRO(M,N)
8 DIM A(9)
9 USE A
10 X=INI(0)
11 FOR N=1 TO 30
12 PRINT N
13 X=PUT(N,N)
14 NEXT N
16 FILEVN#1:"DTA1:DATA.PG"
22 FOR I=0 TO 9
24 PRINT #1:A(1)
26 NEXT I
28 CLOSE #1
30 END
```

```
1 REM -      PROGRAM NAME:    TST18A.PG
2 REM - READ INTO AN ARRAY 10 FL PT WDS
3 REM - (30 INTEGERS FROM MS) WRITE OUT THE
4 REM - 30 INTEGERS ON TTY
5 UDEF INI(N),PLY(Y),DLY(N),DIS(S,E,N,X)
6 UDEF SAM(C,N,P,T),CLK(R,O,S),CLW(N),ADC(N)
7 UDEF GET(M,L),PUT(M,L),DRI(N),DRO(M,N)
8 DIM A(9)
9 USE A
20 FILEN #1:"DTA1:DATA.PG"
22 FOR I=0 TO 9
24 INPUT #1:A(I)
26 NEXT I
28 CLOSE #1
29 X=INI(0)
```

```
30 FOR N=1 TO 30
32 X=GET(0,N)
34 PRINT X
36 NEXT N
40 END
```

```
1 REM -      PROGRAM NAME:    TST19A.PG
2 REM -
3 UDEF INI(N),PLY(Y),DLY(N),DIS(S,E,N,X)
4 UDEF SAM(C,N,P,T),CLK(R,O,S),CLW(N),ADC(N)
5 UDEF GET(M,L),PUT(M,L),DRI(N),DRO(M,N)
6 DIM A(16)
10 REM - SAMPLE CHAN 0 50 TIMES;SYNC OFF SCHMITT;
11 REM - 10 INTERRUPTS/SEC;WHEN DONE DISPLAY TILL ↑N;
12 REM - THEN WRITE OUT DATA TO DTA1;
20 USE A
21 W=INI(0)
22 W=DIS(1,50,1,0)
24 X=SAM(0,1,50,0)
26 Y=CLK(3,104,1)
28 Z=DIS(1,50,1,1)
29 FILEVN #1:"DTA1:SAM.DA"
30 FOR I=0 TO 16
32 PRINT #1:A(I)
34 NEXT I
36 CLOSE #1
38 REM - DISPLAY A PARABOLA
40 P=INI(0)
4R FOR N=-25 TO 25
44 Y=(N*N)/625.1
46 X=PLY(Y)
48 W=DLY(51)
50 NEXT N
52 V=DIS(1,51,1,1)
54 REM - READ DATA BACK IN & DISPLAY IT AS BEFORE
56 FILEN #1:"DTA1:SAM.DA"
58 FOR I=0 TO 16
60 INPUT #1:A(I)
62 NEXT I
64 W=INI(0)
66 Z=DIS(1,50,1,1)
68 END
```

```
1 REM -      PROGRAM NAME:   TST20A.PG
2 REM -
3 UDEF INI(N),PLY(Y),DLY(N),DIS(S,E,N,X)
4 UDEF SAM(C,N,P,T),CLK(R,O,S),CLW(N),ADC(N)
5 UDEF GET(M,L),PUT(M,L),DRI(N),DRO(M,N)
10 DIM X(100),Y(100),A(67)
```

```
11 REM - J1=BINS IN LATENCY(#EPOCHS TILL DONE)
12 REM - T1=BIN WIDTH(TIH) IN MS(#MS/CLK 0.F.)
13 REM - T2=BIN WIDTH OF LATENCY(#CLK 0.F./EPOCHS)
16 PRINT "J1,T1,T2?"
18 INPUT J1,T1,T2
20 I=0
21 J=0
22 K=0
23 Y=CLK(3,T1,1)
25 Z=CLW(0)
30 IF Z=0 GOTO 100
32 IF Z<0 GOTO 36
34 IF Z<8 GOTO 200
35 GOTO  38
36 IF Z>-8 GOTO 200
37 REM - INCR UNDERFLO BIN 0
38 I=0
39 GOTO 300
99 REM - CLK 0.F. ONLY;BMP HIST BIN
100 I=I+1
102 IF I<>100 GOTO 110
103 REM - END OF TIME,BMP HIST BIN
104 X(100)=X(100)+1
105 I=0
109 REM - BMP LATENCY CTR
110 K=K+1
112 IF K<>T2 GOTO 25
113 REM - AN EPOCH IS DONE
114 K=0
116 J=J+1
118 IF J=J1 GOTO 500
119 REM - MORE EPOCHS TO GO?
120 GOTO 25
199 REM - CLK 0.F. AND SCHMITT TRIG
200 X(I)=X(I)+1
202 Y(J)=Y(J)+1
204 GOTO 100
299 REM - SCHMITT TRIG ONLY
300 X(I)=X(I)+1
302 Y(J)=Y(J)+1
304 GOTO 25
498 REM - GET LARGEST BIN VALUE TO BE USED AS A
499 REM - SCALE FACTOR FOR DISPLAY
500 USE A
503 Q=0
504 FOR I=0 TO 100
506 Z=X(I)
508 IF Q>=Z GOTO 516
510 Q=Z
516 NEXT I
549 REM - SCALE ALL BIN VALUES FOR MAX DISPLAY
550 W=INI(0)
551 FOR I=0 TO 100
```

```
552 Z=X(I)
554 Y=Z/(Q+1)
555 W=PLY(Y)
556 NEXT I
598 REM - GET LARGEST LATENCY VAL TO BE
599 REM - USED AS A SCALE FACTOR FOR DISPLAY
600 Q=0
602 FOR I=0 TO 100
604 Z=Y(I)
606 IF Q>=Z GOTO 610
608 Q=Z
610 NEXT I
699 REM - SCALE ALL LATENCY VALS FOR MAX DISPLAY
700 FOR I=0 TO 100
702 Z=Y(I)
704 Y=Z/(Q+1)
706 W=PLY(Y)
708 NEXT I
710 REM - DISPLAY 'TIH'
711 V=DIS(1,101,1,1)
712 REM - DISPLAY LATENCY
720 V=DIS(102,202,1,1)
725 REM - DISPLAY BOTH 'TIH' & LATENCY SIDE BY SIDE
726 V=DIS(1,202,1,1)
800 END
```

Getting on the Air with BASIC

A. DECtape users:

Transfer the user overlays, BASIC.UF, from the DECtape provided with the software kit to the OS/8 system device.

```
.R PIP
*SYS:BASIC.UF<DTAn:BASIC.UF/I
  (where n=0,1,2, . . . , 7)
*↑C
```

B. Papertape users:

Use the ABSLDR to read into core the user overlays which are in binary format on the paper tape, provided with the software kit. Then create a 'save file' on the system device.

```
.R ABSLDR
*PHR:$↑ (where $ symbolizes striking the ALT MODE
  key)
.SAVE SYS BASIC.UF 3400-4577
```

LAB8/E Function Summary

Table 6-4 LAB8/E Function Summary

Function	Explanation
INI(N)	Locate the address of the user array and initialize a pointer to start of the array. N is a dummy argument.
PLY(Y)	Y-data created via the BASIC program is deposited into the user array sequentially. $0 \leqslant Y < .0$
DLY(N)	Used in conjunction with PLY, the scope is refreshed with the contents of the user array after each point is processed. $1 \leqslant N \leqslant 1024$ and N specifies the maximum number of points to be eventually displayed.
DIS(S,E,N,X)	Meaning #1 (X=0). Set up parameters to display ADC data once sampling begins. Meaning #2 (X=0). An array of y-data is to be displayed immediately. In both cases, the display begins with point S of the array, and every Nth point is displayed while not exceeding the desired point E.
SAM(C,N,P,T)	Used to set up parameters for subsequent sampling of the ADC's (T=0) or sampling of digital input registers ($T \neq 0$). C is the first channel # or digital input register #. N is the number of consecutive channels or registers to sample. P is the number of samples per channel or register.
CLK(R,O,S)	Set up the clock for A/D sampling, digital input sampling or for use as a simple timer. R is the desired rate, O is the overflow count and S activates the Schmitt triggers.
CLW(N)	This function returns to the caller a number, indicating whether the clock overflowed or a Schmitt trigger fired and whether these occurred before or after CLW was called.
ADC(N)	This function is issued any time the user wishes to sample A/D channel N.
GET(M,L)	A twelve (12) bit number from the user array at location L is masked with the number M and returned to the caller.
PUT(M,L)	A floating point number, M, is fixed to 12 bits and stored in the user array at location L.

Table 6-4 LAB8/E Function Summary (Cont.)

Function	Explanation
DRN(N)	This function is used any time the user wishes to sample a digital input register N.
DRO(M,N)	The bits of digital output register N are set via the value of M.

chapter 7

fortran II

INTRODUCTION

OS/8 FORTRAN is an improved version of the paper tape 8K FORTRAN. OS/8 FORTRAN contains such added features as Hollerith constants, implied DO loops, chaining, mixing of SABR and FORTRAN statements, and device-independent I/O.

It is assumed that the reader is familiar with the basic concepts of FORTRAN programming. Several excellent elementary texts are available (such as *FORTRAN Programming* by Frederic Stuart, published by John Wiley and Sons, New York, 1969, and *A Guide to FORTRAN Programming* by Daniel D. McCracken, published by John Wiley and Sons, New York) if review is needed.

Calling and Using the OS/8 FORTRAN Compiler

The user calls the FORTRAN compiler by typing:

```
R FORT
```

in reply to the dot generated by the Keyboard Monitor. When the Command Decoder prints an asterisk at the left margin, the user types the appropriate device designations, I/O files, and any of the acceptable specification options allowed for 8K FORTRAN. A carriage return is used to terminate a command string and begin compilation.

The line to the Command Decoder consists of 0 to 3 output files, 1 to 9 input files, and any of the available options. The format of the command line is:

*BINARY,LISTING,MAP<INPUT/OPTION(S)

The first output file holds the binary output in relocatable binary format. If no extension is specified, the extension .RL is assumed.

If a binary output file is not indicated in the command line, then no binary output will be generated. (An exception to this occurs when either the /L or /G options are used; this is explained in the section describing the individual options). The second output file contains the listing; if no extension is specified, the extension .LS is assumed. If no listing file is specified, a listing will not be generated. The third output file is the Linking Loader output, and, unless otherwise specified, this file assumes the extension .MP. (This output is produced by use of the /M, /U and /P options, which are discussed in the section of this chapter concerning the Linking Loader.) 1 to 9 input files are available with OS/8 FORTRAN, although ordinarily only 1 is used. The default extension for input files is .FT.

FORTRAN OPTIONS

The following table provides a list of the options which are available under OS/8 FORTRAN. In addition to these, the /N and /S options to the SABR Assembler may be specified to the FORTRAN compiler, and options to the Linking Loader other than /L may be used. (The user is referred to the respective sections for details.)

Table 7-1 FORTRAN Options

Option	Meaning
/G	Load and execute the file. The Linking Loader is called, the binary output file is loaded and executed. (If a binary file is not specified, a temporary file named FORTRL.TM is created and stored on the file device. This file is loaded into core and then deleted from the file device.) If a starting address is not specified (using the options described under the Linking Loader) control is sent to the program entry point MAIN (the FORTRAN compiler gives this name automatically to the main program.
/K	Keep the file FORTRAN.TM as a permanent file. The FORTRAN compiler produces an output file named FORTRN.TM on the system device. This file is the FORTRAN source program converted into SABR assembly language, and serves as input to the 8K SABR assembler, which is automatically called by

Table 7-1 FORTRAN Options (Cont.)

Option	Meaning
	the compiler. The file FORTRAN.TM is then deleted unless the /K option has been specified. The /K option saves the file as a permanent file, allowing future editing and assembling.
/L	Load, but do not start execution. Call the Linking Loader at the end of the assembly and load the specified binary file. (If a binary output file is not specified, then the temporary file FORTRL.TM is loaded into core and deleted from the file device.) When using the /L option, the user has the choice of terminating the command string with either an ALT MODE or a carriage return. If ALT MODE is typed, the Loader returns to the Keyboard Monitor with a core image in core, while the RETURN key instructs the Loader to ask for more input.

EXAMPLE PROGRAM

The following example illustrates the ease with which a FORTRAN program can be executed under OS/8. The program TEST has been created with the Symbolic Editor and saved on device SYS:

```
C       FORTRAN DEMO 'TEST'
C       COMPUTE AND PRINT POWERS OF TWO

        DIMENSION A(16)
        WRITE (1,15)
15      FORMAT (/'POWERS OF TWO..EXAMPLE PROGRAM'/)
        DO 20 N=1,16
20      A(N)=2.**N
        WRITE (1,25) (N,A(N),N=1,16)
25      FORMAT ('2** 'I2'='F10.2)
        CALL EXIT
        END
```

By issuing the following commands, TEST is loaded and executed; execution is automatic with the /G option:

```
.R FORT
*TEST/G

POWERS OF TWO..EXAMPLE PROGRAM

2**  1=       2.00
2**  2=       4.00
2**  3=       8.00
2**  4=      16.00
2**  5=      32.00
2**  6=      64.00
2**  7=     128.00
2**  8=     256.00
2**  9=     512.00
2** 10=    1024.00
2** 11=    2048.00
2** 12=    4096.00
2** 13=    8192.00
2** 14=   16384.00
2** 15=   32768.00
2** 16=   65536.00
```

FORTRAN assembles one main program or subroutine per call. A job with multiple subprograms is run by compiling each routine separately and combining them with the Linking Loader.

Typing a CTRL/C (↑C) at run time during a non-compute bound job will return control to the Keyboard Monitor. Typing .ST at this point will restart the user's FORTRAN program. If ↑C is typed when compiling a program, FORTRAN will have to be recalled.

EXAMPLES OF FORTRAN I/O SPECIFICATION COMMANDS

Example 1:

```
.R FORT
*DTA1:TEST/G
```

The input file TEST.FT (or TEST) on DTA1 is compiled, the output stored in FORTRN.TM on the system device, and SABR is called. SABR uses FORTRN.TM as input and outputs the assembled file into FORTRL.TM, deleting the old FORTRN.TM. The /G option specifies that the Linking Loader then loads

FORTRL.TM and the Library Subroutines, deletes FORTRL.TM upon loading, and sends control to the entry point MAIN.
Example 2:

```
.R FORT
*MATRIX<MATRIX.AB/G/U
```

The input file MATRIX.AB on DSK is compiled and the output stored in SYS:FORTRN.TM. SABR is called and assembles SYS:FORTRN.TM, putting the relocatable binary output into DSK:MATRIX.RL, deleting the file FORTRN.MT. The /G option specifies that the Linking Loader then loads MATRIX.RL and the Library Subroutines, and then prints on the teleprinter (via /U) a list of undefined external symbols and a count of the unused pages in each memory field.
Example 3:

```
.R FORT
*,,LPT:<INPUT/L/M
*
```

The FORTRAN Compiler compiles and SABR assembles the file DSK:INPUT.FT (or INPUT), outputting the binary file as SYS:FORTRL.TM. The Linking Loader is automatically called (/L) to load SYS:FORTRL.TM into core and delete that file from SYS. The Linking Loader puts a full loading map on the LPT device (/M). The Loader then asks for another command string. If the line had been terminated with the ALT MODE key instead of the RETURN key, control would be returned to the Keyboard Monitor after loading.
Example 4:

```
.R FORT
*SUB1<SUB1
.R FORT
*SUB2<SUB2
.R FORT
*MAIN/L
*SUB1,SUB2/G
```

The subroutines and the MAIN program are each compiled separately, and the MAIN program is loaded but not executed (as the /L option indicates). The Linking Loader is called at the end of

the assembly and waits for more input. The /G option is used to load the FORTRAN Library Subroutines and initiate execution of the MAIN program.
Example 5:

```
.R FORT
*DTA5:SOURCE/L
```

The file SOURCE on DTA5 is compiled, assembled, and loaded but not executed.
Example 6:

```
.R FORT
*DTA1:PROG,PTP:,PTP:<DTA1:PROG(NMG)
```

For those users with DECtape systems, keeping the source program on a non-system DECtape and putting the binary on a non-system DECtape gives the best possible results in terms of minimizing tape motion. The above file, PROG, is loaded and executed. The binary is stored on DTA1 under the name PROG.RL, and the symbol table, the map of the loaded program and the count of the free pages in each field are punched onto paper tape.

In DECtape systems, excessive DECtape motion can also be eliminated by storing LIB8.RL on a non-system tape. The user would then specify to the Loader:

```
*DTA2:LIB8.RL/L
```

Using FORTRAN or SABR with the Interrupt On

SABR code can be run with the interrupt on, providing the user supplies his own interrupt handling code. That code which is executed when the interrupt is off must not call any of the SABR subroutines and must be independent of all SABR or library subroutines and linkage subroutines. With the interrupt on, the user should not call exit routines or do any generalized (device-independent) I/O, unless those routines are modified to make allowances for interrupts.

Using PAL8 with SABR or FORTRAN

It is possible to call PAL8 subroutines from a SABR or FORTRAN program. The user should build a core image of the running FORTRAN or SABR program and return to the Keyboard Monitor by typing $ (ALT MODE key) on the last Linking Loader Command. He should then save the core image. The core image file (.SV) can be used as input to the Absolute Loader (ABSLDR) with the /I option, followed by the binary of the PAL8 routine. For example:

```
.R ABSLDR
*DTA7:CHAIN2.SV/I
*PALSUB.BN/G$
```

The above calls the Absolute Loader, loads the core image CHAIN2.SV and then merges the PALSUB.BN program with it. Execution starts at location 200 and, when completed, the system returns to the Keyboard Monitor for further instructions.

FORTRAN Data Files

When doing FORTRAN output onto DECtape or disk into a file which is to be read only as a data file by another FORTRAN program, a significant time saving can be obtained by using A6 format to output floating-point variables and A2 format to output integer values. The same format specifications must be used when the data is read. The data file is not an ASCII file and should not be edited with EDIT. The file should only be moved by PIP in image mode (/I option).

The following caution should be observed concerning programs which may have been written and compiled with a previous version of OS/8 FORTRAN:

CAUTION

A FORTRAN compiler and its corresponding Library constitute an interlocking set of programs. No user should attempt to compile a program under OS/8 and load it with the paper tape FORTRAN, or vice versa. Similarly, programs developed with the current FORTRAN compiler should not be run under an old FORTRAN system.

Character Set

The following characters are used in the FORTRAN language.[1]

1. The alphabetic characters, A through Z.
2. The numeric characters, 0 through 9.
3. The special characters:[2]

!	'	↑
"	(	[
$	)	]
%	+	\
&	—	←
*	/	
=	.	
#	,	
;	<	
:	>	
?	(space)	

FORTRAN Constants

Constants are self-defining numeric values appearing in source statements and are of three types: integer, real, and Hollerith.

INTEGER CONSTANTS

An integer (fixed point) constant is represented by a digit string of from one to four decimal digits, written with an optional sign,

---

[1] Appendix A lists the octal and decimal representations of the FORTRAN character set.

[2] Of these, the characters " ! $ % & # : ? < > ↑ [ ] \ ← may only appear inside FORMAT statements or Hollerith constants.

and without a decimal point. An integer constant must fall within the range −2047 to +2047. For example:

```
47
+47     (+ sign is optional)
−2
0434    (leading zeros are ignored)
 0      (zero)
```

REAL CONSTANTS

A real constant is represented by a digit string, an explicit decimal point, an optional sign, and possibly an integer exponent to denote a power of ten (7.2×10^3 is written 7.2E+03). A real constant may consist of any number of digits but only the leftmost eight digits appear in the compiled program. Real constants must fall within the range of $\pm\ 1.7\times10^{38}$.

HOLLERITH CONSTANTS

A Hollerith constant is a string of up to 6 characters (including blanks) enclosed in single quotes. A Hollerith constant is treated like a real constant, except that it cannot be used in arithmetic expressions other than for simple equivalence (A=B). Any character except the quote character itself can be used in a Hollerith constant. For example:

```
'MOM'
'A+B=C'
'5 & 10'
```

FORTRAN Variables

A variable is a named quantity whose value may change during execution of a program. Variables are specified by name and type. The name of a variable consists of one or more alphanumeric characters the first of which must be alphabetic. Although any number of characters may be used to make up the variable name, only the first five characters are interpreted as defining the name;

the rest are ignored. For example, DELTAX, DELTAY, and DELTA all represent the same variable name.

The type of variable (integer or real) is determined by the first letter of the variable name. A first letter of I, J, K, L, M, or N indicates an integer variable, and any other first letter indicates a real variable. Variables of either type may be either scalar or array variables. A variable is an array variable if it first appears in a DIMENSION statement.

INTEGER VARIABLES

The name of an integer variable must begin with an I, J, K, L, M, or N. An integer variable undergoes arithmetic calculations with automatic truncation of any fractional part. For example, if the current value of K is 5 and the current value of J is 9, J/K would yield 1 as a result.

Integer variables may be converted to real variables by the function FLOAT (see Function Calls) or by an arithmetic statement (see Arithmetic Statements). Integer variables must fall within the range −2047 to +2047.

Integer arithmetic operations do not check for overflow. For example, the sum 2047+2047 will yield a result of −2. For more information refer to Chapter 1 of *Introduction to Programming* or any text on binary arithmetic.

REAL VARIABLES

A real variable name begins with any alphabetic character other than I, J, K, L, M, or N. Real variables may be converted to integer variables by the function IFIX (see Function Calls) or by an arithmetic statement. Real variables undergo no truncation in arithmetic calculations.

SCALAR VARIABLES

A scalar variable may be either integer or real and represents a single quantity. For example:

```
LM
A
G2
TOTAL
J
```

ARRAY VARIABLES

An array (subscripted) variable represents a single element of a one- or two-dimensional array of quantities. The array element is denoted by the array name followed by a subscript list enclosed in parentheses. The subscript list may be any integer expression or two integer expressions separated by a comma. The expressions may be arithmetic combinations of integer variables and integer constants. Each expression represents a subscript, and the values of the expressions determine the referenced array element. For example, the row vector A_i would be represented by the subscripted variable A(I), and the element in the second column of the first row of the matrix A, would be represented by A (1, 2).

Examples of one-dimensional arrays are:

```
Y(1)
PORT(K)
```

while a two-dimensional array appears as follows:

```
A(3*K+2,1)
```

Any array must appear in a DIMENSION statement prior to its first appearance in an executable statement. The DIMENSION statement specifies the number of elements in the array.

Arrays are stored in increasing storage locations with the first subscript varying most rapidly (see Storage Allocation). The two-dimensional array B (J, K) is stored in the following order:

B(1, 1), B(2, 1), . . . , B(J, 1), B(1, 2), B(2, 2), . . . , B(J, 2), . . . , B(J, K)

For representation of arrays of more than two dimensions, refer to the section entitled Representation of N-Dimensional Arrays toward the end of this chapter.

SUBSCRIPTING

Since excessive subscripting tends to use core memory inefficiently, it is suggested that subscripted variables be used judiciously. For example, the statement:

```
A=((B(I)+C2)*B(I)+C1)*B(I)
```

could be rewritten with a considerable saving of core memory as follows:

```
T=B(I)
A=((T+C2)*T+C1)*T
```

Expressions

An expression is a sequence of constants, variables, and function references separated by arithmetic operators and parentheses in accordance with mathematical convention and the rules given below.

Without parentheses, algebraic operations are performed in the following descending order:

**	exponentiation
−	unary negation
* and /	multiplication and division
+ and −	addition and subtraction
=	equals or replacement sign

Parentheses are used to change the order of precedence. An operation enclosed in parentheses is performed before its result is used in other operations. In the case of operations of equal priority, the calculations are performed from left to right.

Integers and real numbers may be raised to either integer or real powers. An expression of the form:

A**B

means A^B and is real unless both A and B are integers. Exponential (e^x) and natural logarithmic ($\log_e(x)$) functions are supplied as subprograms and are explained later.

Excluding ** (exponentiation), no two arithmetic operators may appear in sequence unless the second is a unary plus or minus.

The mode (or type) of an expression may be either integer or real and is determined by its constituents. Variable modes may not be mixed in an expression with the following exceptions:

1. A real variable may be raised to an integer power:

```
A**2
```

2. Mode may be altered by using the functions IFIX and FLOAT (see Function Calls):

```
A*FLOAT(I)
```

The I in example 2 above, indicates an *integer* variable; it is changed to *real* (in floating point format) by the FLOAT function.

Zero raised to a power of zero yields a result of 1. Zero raised to any other power yields a zero result. Numbers are raised to integer powers by repetitive multiplication. Numbers are raised to floating point powers by calling the EXP and ALOG functions. A negative number raised to a floating point power does not cause an error message but uses the absolute value. Thus, the expression (−3.0)**3.0 yields a result of +27.

Any arithmetic expression may be enclosed in parentheses and be considered a basic element.

```
IFIX(X+Y)/2
(ZETA)
(COS(SIN(PI*EM)+X))
```

An arithmetic expression may consist of a single element (constant, variable, or function call). For example:

```
2.71828
Z(N)
TAN(THETA)
```

Compound arithmetic expressions may be formed using arithmetic operators to combine basic elements. For example:

```
X+3.
TOTAL/A
TAN(PI*EM)
```

Expressions preceded by a + or a − sign are also arithmetic expressions. For example:

```
+X
-(ALPHA*BETA)
-SQRT(-GAMMA)
```

As an example of a typical arithmetic expression using arithmetic operators and a function call, the expression for the largest root of the general quadratic equation:

$$\frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

is coded as:

```
(-B+SQRT(B**2-4.*A*C))/2.*A)
```

FORTRAN STATEMENTS

A FORTRAN source program consists of a series of statements, each of which must start on a separate line. Any FORTRAN statement may appear in the statement field (columns 7 through 72) and may be preceded by a positive number, called a statement number, of from 1 to 4 digits which serves as an address label and is used when referencing the statement. When used, statement numbers are coded in columns 1 through 5 of the 72 column line. Statement numbers need not appear in sequential order, but no two statements should have the same number. Statement numbers are limited to a value of 2047 or less.

When using the Symbolic Editor to create the source program, typing a CTRL/TAB (generated by holding down the CTRL key and depressing TAB) causes a jump over the statement number columns and into the statement field. Except for data within a Hollerith field (see Input/Output Statements), spaces are ignored by the compiler. The programmer may use spaces freely, however, to make the program listing more readable and to organize data into columns.

Line Continuation Designator

Statements too long for the statement field of a single terminal line may be continued on the next line. The continued portion must not be given a line number, but must have an alphanumeric character other than 0 in column 6. If the Symbolic Editor is used, the programmer may type a CTRL/TAB followed by a digit from 1 to 9 before continuing the line. The continuation character is not treated as part of the statement.

For example, using spaces, a continued statement would look as follows:

```
      WRITE (3,30)
30    FORMAT (1X,'THE FOLLOWING DATA IS GROUPED INTO THREE
     1 PARTS UNDER THE HEADINGS X, Y, AND Z.')
```

Using tabs, the same statement would be typed:

```
      WRITE (3,30)
30    FORMAT (1X,'THE FOLLOWING DATA IS GROUPED INTO THREE
      1 PARTS UNDER THE HEADINGS X, Y, AND Z.')
```

There is no limit to the number of continuation lines which may appear. However, one restriction is that an implied DO loop must not be broken, but must be on one line. For ease in program correction, it is recommended that continuation lines be minimized.

Comments

The letter C in column 1 of a line designates that line as a comment line. A comment appears in a program listing but has no effect on program compilation. Any number of comment lines may appear in a given program, and comments that are too long for one line may be continued by placing a C in the first column of the next line. A comment line may not appear between another line and its continuation.

FORTRAN statements are of five types:

1. Arithmetic, defining calculations to be performed;
2. Input/Output, directing communication between the program and input/output devices;
3. Control, governing the sequence of execution of statements within a program;
4. Specification, describing the form and content of data within the program;
5. Subprogram, defining the form and occurrence of subprograms and subroutines.

Each of these five types is explained in the following paragraphs.

Arithmetic Statements

Constants and *variables,* identified as to type and connected by logical and arithmetic operators form *expressions*: one or more expressions form an *arithmetic statement.* Arithmetic statements are of the general form:

V=E

where V is a variable name (subscripted or nonsubscripted), E is an expression, and = is a replacement operator. The arithmetic statement causes the FORTRAN object program to evaluate the expression E and assign the resultant value to the variable V. Note that = signifies replacement, not equality. Thus, expressions of the form:

```
A=A+B

A=A*B
```

are quite meaningful and indicate that the value of the variable A is to be changed.

For example:

```
Y=1.1*Y
P=X**2+3.*X+2.0
X(N)=EN*ZETA*(ALPHA+EM/PI)
```

The expression value is made to agree in type with the variable before replacement occurs. In the statement:

```
META=W*(ABETA+E)
```

since META is an integer and the expression is real, the expression value is truncated to an integer before assignment to META.

Input/Output Statements

Input/Output (I/O) statements are used to control the transfer of data beween computer memory and peripheral devices and to specify the format of the output data. I/O statements may be divided into two categories:

1. Data transmission statements, READ and WRITE, specify transmission of data between computer memory and I/O devices.
2. Nonexecutable FORMAT statements enable conversion between internal data (within core memory) and external data.

DATA TRANSMISSION STATEMENTS

The two data transmission statements, READ and WRITE, accomplish input/output transfer of data listed in a FORMAT statement. The two statements are of the form:

READ (unit, format) I/O list
WRITE (unit, format) I/O list

where *unit* is a device designation which can be an integer constant or an integer variable, *format* is a FORMAT statement line number, and the *I/O* list is a list specifying the order of transmission of the variable values. During input, the new values of listed variables may be used in subscript or control expressions for variables appearing later in the list.

For example:

```
READ(2,1000)L,A(L),B(L+1)
```

reads a new value of L and uses this value in the subscripts of A and B; where 2 is the device designation code, and 1000 is a FORMAT statement number.

An element in an I/O list can take one of the following forms:

1. Arithmetic expression: expressions more complicated than a single variable (which can be subscripted) are meaningless in an input operation.
2. The name of an array (1 or 2 dimensional) : this indicates that every element of the array is to be transmitted. Elements are transmitted in the order in which they are stored in core.

   For example:

```
DIMENSION A(2,2)
READ (1,100) A
```

reads:

```
A(1,1),A(2,1),A(1,2),A(2,2)
```

3. Implied DO Loops of the form:

$$(s_1,s_2,\ldots,s_n,i=m_1,m_2,m_3)$$

repeat the list elements (s_n) with the value of i being equal to m_1 through m_2 having an optional step value of m_3. The m's are integer constants or variables, i is an integer variable, and s_1-s_n are the I/O list elements (possibly including an implied DO loop). For example:

```
DIMENSION A(3,6)
WRITE (1,100) I,(A(J,I)J=1,3)
```

will output the values:

```
I,A(1,I),A(2,I),A(3,I)
```

It is important to remember that when using implied DO loops, the entire implied DO loop must be on the same input line or card. An implied DO loop cannot be continued onto the next line with a continuation character.

If no I/O list is specified for a WRITE statement, then information is read directly from the specified FORMAT statement and written on the device designated.

Data appears on the external device in the form of records.[3] All information appearing on input is grouped into records. On output to the printer a record is one line. The amount of information contained in each ASCII record is specified by the FORMAT statement and the I/O list.

[3] This should not be confused with the OS/8 record, which is equal to 256_{10} words (2 DECtape blocks with the 129th word of each block ignored.)

Each execution of an I/O statement initiates the transmission of a new data record. Thus, the statement:

```
READ(1,100)FIRST,SECOND,THIRD
```

is not necessarily equivalent to the statements below where 100 is the FORMAT statement referenced:

```
READ(1,100)FIRST
READ(1,100)SECOND
READ(1,100)THIRD
```

In the second case, at least three separate records are required, whereas, the single statement

READ (d, f) FIRST, SECOND, THIRD

may require one, two, three, or more records depending upon FORMAT statement f.

If an I/O statement requests less than a full record of information, the unrequested part of the record is lost and cannot be recovered by another I/O statement without repositioning the record.

If an I/O list requires more than one ASCII record of information, successive records are read.

READ Statement

The READ statement specifies transfer of information from a selected input device to internal memory, corresponding to a list of named variables, arrays or array elements. The READ statement assumes the following form:

READ (d, f) list

where d is a device designation which may be an integer constant or an integer variable, f is a FORMAT statement line number, and list is a list of variables whose values are to be input.

The READ statement causes ASCII information to be read from the device designated and stored in memory as values of the variables in the list. The data is converted to internal form as specified by the referenced FORMAT statement.

For example:

```
READ(1,15)ETA,PI
```

WRITE Statement

The WRITE statement specifies transfer of information from the computer to a specified output device. The WRITE statement assumes one of the following forms:

WRITE (d, f) list
WRITE (d, f)

where d is a device designation (integer constant or integer variable), f is a FORMAT statement line number, and list is a list of variables to be output.

The WRITE statement followed by a list causes the values of the variables in the list to be read from memory and written on the designated device in ASCII form. The data is converted to external form as specified by the designated FORMAT statement.

The WRITE statement without a list causes information (generally Hollerith type) to be read directly from the specified format and written on the designated device in ASCII form.

The I/O device designations used in the READ and WRITE statements are described in Table 7-2.

Table 7-2 Device Designations

Device Code	Input Designation	Output Designation
1	Teletype keyboard or low-speed reader	Teleprinter
2	High-speed reader	High-speed punch
3	Card reader (CR8/I)	Line printer (LP08)
4[4]	Assignable device (see Device Independent I/O and Chaining)	Assignable device

Device code 3 is assigned to the card reader (for all READ statements), and the line printer (for all WRITE statements). The card reader uses a two-page device handler, which is too large to

[4] If using device code 4, the /I or /O options to the Linking Loader must be given. If the assignable device is a two-page handler, the /H option must be given also.

be used with the device independent I/O feature (device code 4). Therefore, the card reader has its own device code.

The line printer is a separate output device because it can require special formatting, such as inserting a Form Feed to skip to the top of a page. The contents of the first column of any line is a control character. These control characters are never printed. They are as follows:

Character in Column 1	Resulting Spacing
space	single space
0	double space
1	skip to top of next page (Form Feed)
all others	single space

FORMAT STATEMENT

The nonexecutable FORMAT statement specifies the form and arrangement of data on the selected external device. FORMAT statements are of the form:

$$m \text{ FORMAT } (S_1,S_2,...S_n)$$

where m is a statement number and each S is a data field specification. Both numeric and alphanumeric field specifications may appear in a FORMAT statement. The FORMAT statement also provides for handling multiple record formats, skipping characters, space insertion, and repetition.

FORMAT statements may be placed anywhere in the source program. Unless the FORMAT statement contains only alphanumeric data for direct I/O transmission, it will be used in conjunction with the list of a data transmission statement.

During transmission of data, the object program scans the designated FORMAT statement; if a specification for a numeric field is present, and the data transmission statement contains items remaining to be transmitted, transmission takes place according to the specification. This process ceases and execution of the data transmission statement is terminated as soon as all specified items have been transmitted. The FORMAT statement may contain specifications for more items than are indicated by the data transmission

statement. The FORMAT statement may also contain specifications for fewer items than are indicated by the data transmission statement, in which case, format control reverts to the rightmost left parenthesis in the FORMAT statement. If an input list requires more characters than the input device supplies for a given record, blanks are inserted.

Numeric Fields

Numeric field specification codes and the corresponding internal and external forms of the numbers are listed in Table 7-3.

Table 7-3 Numeric Field Codes

Conversion Code	Internal Form	External Form
E	Binary floating point	Decimal floating point[5] with E exponents: 0.324E+10
F	Binary floating point	Decimal floating point with no exponent: 283.75
I	Binary integer	Decimal integer: 79

Conversions are specified by the form:

rEw.d
rFw.d
rIw

where r is a repetition count, E, F, and I designate the conversion code, w is an integer specifying the field width, and d is an integer specifying the number of decimal places to the right of the decimal point. For E and F input, the position of the decimal point in the external field takes precedence over the value of d. For example:

[5] When using E format, or with numbers less than 1.0 when using F format in a WRITE statement, a zero will be typed to the left of the decimal point.

```
FORMAT (I5,F10.2,E16.8)
```

could be used to output the line

```
   32     -17.60   0.59624575E+03
```

on the output listing.

The field width should always be large enough to include the decimal point, sign, and exponent (plus a leading zero in OS/8 FORTRAN). In all numeric field conversions, if the field width is not large enough to accommodate the converted number, asterisks will be printed; the number is always right-justified in the field.

Numeric Input Conversion

In general, numeric input conversion is compatible with most other FORTRAN processors. A few exceptions are listed below:

1. Blanks are ignored except to determine in which field digits fall. Thus, numbers are treated as if they are right-justified within a field. In an F5.2 format, the following:

   bbb12
   12bbb
   00012

   are read as the number 0.12 (where 'b' represents a blank space).

2. A null line delimited by two carriage return/line feed (CR/LF) combinations is treated as a line of blanks, and blanks are appended to the right of a line (if necessary) to fill out a FORMAT statement. Thus:

   12 (CR/LF)
   12bbb
   bbb12

   are identical under an F5.2 format. If an entire line is blank, numeric data from that line is read as zeros.

3. No distinction is made between E and F format on input. Thus:

100.
100E2
1.E2
10000

are all read identically under either an F5.2 or E5.2 format.

Alphanumeric Fields

Alphanumeric data can be transmitted in a manner similar to numeric data by use of the form

rAw

where r is a repetition count, A is the control character, and w is the number of characters in the field. Alphanumeric characters are transmitted as the value of a variable in an I/O list; the variable may be either integer or real.

Although w may have any value, the number of characters transmitted is limited by the maximum number of characters which can be stored in the space allotted for the variable. This maximum depends upon the variable type; for a real variable the maximum is six characters, for an integer variable the maximum is two characters. The characters are stored in stripped ASCII format. If not enough data is supplied as input to the variables, the data is padded with blanks on the right. For example:

```
      READ (1,20) M1,M2,M3,M4,M5,M6,M7,M8
20    FORMAT (8A1)
```

if the user types at this point:

```
123ABC
```

followed by a carriage return, the following are the values of the variables:

Variable	Decimal	Octal	ASCII
M1	−928	6140	1
M2	−864	6240	2
M3	−800	6340	3
M4	96	0140	A
M5	160	0240	B
M6	224	0340	C
M7	−2016	4040	blank
M8	−2016	4040	blank

If the above had been read in 4A2 format, the values would be as follows:

Variable	Decimal	Octal	ASCII
M1	−910	6162	1 2
M2	−831	6301	3 A
M3	131	0203	B C
M4	−2016	4040	blanks
	. . . . .		
M8	−2016	4040	blanks

As a second example:

```
      READ (1,20) ALPHA
20    FORMAT (A6)
```

the user types:

```
123AB
```

and a carriage return, and the octal value of ALPHA is:

```
6162   6301   0240
```

NOTE

The numeric value of alphanumeric characters stored in floating point variables is generally not meaningful.

Hollerith Conversion

Alphanumeric data may be transmitted directly from the FORMAT statement by using Hollerith (H) conversion. H-conversion format is normally referenced by WRITE statements only.

In H-conversion, the alphanumeric string is specified by the form

$$nH\ h_1, h_2, \ldots, h_n$$

where H is the control character and n is the number of characters in the string, including blanks. For example, the statement below can be used to print PROGRAM COMPLETE on the output listing.

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

A Hollerith string may consist of any characters capable of representation in the processor. The space character is a valid and significant character in a Hollerith string.

An attempt to use H format specifications with a READ statement will cause characters from the format field to be either printed or punched. This can be a useful feature since it provides a simple way of identifying data that is to be read from the Teletype keyboard. For example, the following instructions:

```
      READ (1,30)A,B
30    FORMAT (4HA = ,F7.2/4HB = ,F7.2)
```

cause A = and B = to be printed out before the data is read.

By merely enclosing the alphanumeric data in single quotes, the same result is achieved as in H-conversion; on input, the characters between the single quotes are typed as output characters, and on output, the characters between the single quotes (including blanks) are written as part of the output data. For example, when referred to from a WRITE statement:

```
50    FORMAT (' PROGRAM COMPLETE')
```

causes PROGRAM COMPLETE to be printed. This method eliminates the need to count characters.

Blank or Skip Fields

Blanks can be introduced into an output record or characters skipped on an input record by use of the nX specification. The number n indicates the number of blanks or characters skipped and must be greater than zero. For example:

```
FORMAT(5H STEPI5,10X2HY=F7.3)
```

can be used to output the line:

```
 STEP   28           Y=  3.872
```

Mixed Fields

A Hollerith format field may be placed among other fields of the format. The statement:

```
FORMAT(I5,7H FORCE=F10.5)
```

can be used to output the line:

```
   22 FORCE=  17.68901
```

The separating comma may be omitted after a Hollerith format field, as shown above.

Repetition of Fields

Repetition of a field specification may be specified by preceding the control character E, F, or I by an unsigned integer giving the number of repetitions desired.

```
FORMAT(2E12.4,3I5)
```

is equivalent to:

```
FORMAT(E12.4,E12.4,I5,I5,I5)
```

Repetition of Groups

A group of field specifications may be repeated by enclosing the group in parentheses and preceding the whole with the repetition number.

For example:

```
FORMAT(2I8,2(E15.5,2F8.3))
```

is equivalent to:

```
FORMAT(2I8,E15.5,2F8.3,E15.5,2F8.3)
```

Multiple Record Formats

To handle a group of output records where different records have different field specifications, a slash is used to indicate a new record. For example, the statement:

```
FORMAT(3I8/I5,2F8.4)
```

is equivalent to:

```
FORMAT(3I8)
```

for the first record and

```
FORMAT(I5,2F8.4)
```

for the second record.

The separating comma may be omitted when a slash is used. When n slashes appear at the end or beginning of a format, n blank records may be written on output (producing a CR/LF for each record) or ignored on input. When n slashes appear in the middle of a format, n-1 blank records are written or n-1 records skipped. Both the slash and the closing parenthesis at the end of the format indicate the termination of a record. If the list of an I/O statement dictates that transmission of data is to continue after the closing parenthesis of the format is reached, the format is repeated

from the last open parenthesis of level one or zero. Thus, the statement:

```
FORMAT(F7.2,(2(E15.5,E15.4),I7))
```

causes the format:

```
F7.2,2(E15.5,E15.4),I7
```

to be used on the first record, and the format:

```
2(E15.5,E15.4),I7
```

to be used on succeeding records.

As a further example, consider the statement:

```
FORMAT(F7.2/(2(E15.5,E15.4),I7))
```

The first record has the format:

```
F7.2
```

and successive records have the format:

```
2(E15.5,E15.4),I7
```

Control Statements

The control statements GO TO, IF, DO, PAUSE, STOP, and END alter the sequence of statement execution, temporarily or permanently halt program execution, and stop compilation.

GO TO STATEMENT

The GO TO statement has two forms: unconditional and computed.

Unconditional GO TO

Unconditional GO TO statements are of the form:

GO TO n

where n is the number of an executable statement. Control is transferred to the statement numbered n.

Computed GO TO

Computed GO TO statements have the form:

$$\text{GO TO } (n_1, n_2, \ldots, n_k), J$$

where $n_1, n_2, \ldots, n_k$ are statement numbers and J is a nonsubscripted integer variable. This statement transfers control to the statement numbered $n_1, n_2 \ldots, n_k$ if J has the value 1, 2, . . . , k, respectively. The index (J in the above example) of a computed GO TO statement must never be zero or greater than the number of statement numbers in the list (in the example above, not greater than k). For example, in the statement:

```
GO TO(20,10,5),K
```

the variable K acts as a switch, causing a transfer to statement 20 if K = 1, to statement 10 if K = 2, or to statement 5 if K = 3.

IF STATEMENT

Numerical IF statements are of the form:

$$\text{IF (expression) } n_1, n_2, n_3$$

where n_1, n_2, n_3 are statement numbers. This statement transfers control to the statement numbered n_1, n_2, n_3 if the value of the numeric expression is less than, equal to, or greater than zero, respectively. The expression may be a simple variable or any arithmetic expression.

```
IF (ETA)4,7,12
IF(KAPPA-L(10))20,14,14
```

DO STATEMENT

The DO statement simplifies the coding of iterative procedures. DO statements are of the form:

$$\text{DO n i} = m_1, m_2, m_3$$

where n is a statement number, i is a scalar integer variable, and m_1, m_2, m_3 are integer constants or nonsubscripted integer variables. If m_3 is not specified, it is understood to be 1.

The DO statement causes the statements which follow, up to and including the statement numbered n, to be executed repeatedly. This group of statements is called the range of the DO statement. In the example above, the integer variable i is called the index, the values of m_1, m_2, m_3 are, respectively, the initial, terminal, and increment values of the index.

For example:

```
DO 10 J=1,N
DO 20 I=J,K,5
DO 30 L=I,J,K
```

The index is incremented and tested before the range of the DO is executed. After the last execution of the range, control passes to the statement immediately following the terminal statement in what is called a *normal exit.* An exit may also occur by a transfer out of the range taking place before the loop has been executed the total number of times specified in the DO statement.

DO loops may be nested, or contained within one another, provided the range of each contained loop is entirely within the range of the containing DO statement. Nested DO loops may contain the same terminal statement, however. A transfer into a DO loop from outside the range is not allowed.

Within the range of a DO statement, the index is available for use as an ordinary variable. After a transfer from within the range, the index retains its current value and is available for use as a variable.[6] The values of the initial, terminal, and increment variables for the index and the index of the DO loop may not be altered within the range of the DO statement.

---

[6] After a normal exit from a DO loop, the index of the DO statement has the value of the index the final time through the loop plus whatever increment was assigned. For example:

```
DO 10 I=1,5
```

after a normal exit the value of the index is 6. However, it is good programming practice to avoid using the index as a variable following a normal exit until it has been redefined, as according to *ANSI FORTRAN Standards* the value is undefined.

The last statement of a DO loop must be executable, and must not be an IF, GO TO or DO statement.

CONTINUE STATEMENT

This is a dummy statement, used primarily as a target for transfers, particularly as the last statement in the range of a DO statement. For example, in the sequence:

```
      DO 7 K=INIT,LIMIT
           .
           .
           .
      IF (X(K)) 22,13,7
           .
           .
           .
7     CONTINUE
```

a positive value of X(K) begins another execution of the range. The CONTINUE provides a target address for the IF statement and ends the range of the DO statement.

PAUSE, STOP, AND END STATEMENTS

The PAUSE and STOP statements affect FORTRAN object program operation; the END statement affects assembler operation only.

Pause Statement

The PAUSE statement enables the program to incorporate operator activity into the sequence of automatic events. The PAUSE statement assumes one of two forms:

```
          PAUSE
or        PAUSE n
```

where n is an unsigned decimal number.

Execution of the PAUSE statement causes the octal equivalent of the decimal number n to be displayed in the accumulator on the user's console. Program execution may be resumed (at the next executable statement) by depressing the CONTinue key on the console.

In some cases the PAUSE statement may be used to give the operator a chance to change data tapes or to remove a tape from the punch. When this is done it is necessary to follow the PAUSE

statement with a call to the OPEN subroutine. This subroutine initializes the I/O devices and sets hardware flags that may have been cleared by pressing the tape feed button. For example:

```
PAUSE.
CALL OPEN
```

NOTE

The CALL OPEN statement in OS/8 FORTRAN also resets all I/O on unit 4, the assignable channel. Any further READs or WRITEs on unit 4 without an intervening IOPEN or OOPEN will print an error message and abort.

Stop Statement

The STOP statement has the form:

STOP

It terminates program execution. STOP may occur several times within a single program to indicate alternate points at which execution may cease. Program control is either directed to a STOP statement or transferred around it.

End Statement

The END statement is of the form:

END

and signals the compiler to terminate compilation. The END statement must be the last statement of every program. (In OS/8 FORTRAN, the END statement generates a STOP statement as well.)

Specification Statements

Specification statements allocate storage and furnish information about variables and constants to the compiler. The specification statements are COMMON, DIMENSION, and EQUIVALENCE and, when used, must appear in the program prior to any executable statement.

COMMON STATEMENT

The COMMON statement causes specified variables or arrays to be stored in an area available to other programs. By means of COMMON statements, the data of a main program and/or the data of its subprograms may share a common storage area. Varibles in COMMON statements are assigned to locations in ascending order in field 1 beginning at location 200 storage allocation. The COMMON statement has the general form:

$$\text{COMMON } v_1, v_2, \ldots, v_n$$

where v is a variable name. See the section entitled Common Storage Allocation for greater detail.

DIMENSION STATEMENT

The DIMENSION statement is used to declare array identifiers and to specify the number and bounds of the array subscripts. The information supplied in a DIMENSION statement is required for the allocation of memory for arrays. Any number of arrays may be declared in a single DIMENSION statement. The DIMENSION statement has the form:

$$\text{DIMENSION } s_1, s_2, \ldots, s_n$$

where s is an array specification. For example:

```
DIMENSION A(100)
DIMENSION Y(10),PORT(25),B(10,10),J(32)
```

Dimension statements are used for the purpose of reserving sufficient storage space for anticipated data; it is the user's responsibility to see that his subscripting does not conflict with the DIMENSION statement declarations. For example:

```
DIMENSION I(10),J(10),K(10)
I(2,4)=2
J(12)=3
```

The above statements would assemble without error; at run time I(8) would be set equal to 2 and K(2) would be set equal to 3.

NOTE

When variables in common storage are dimensioned, the COMMON statement must appear before the DIMENSION statement.

EQUIVALENCE STATEMENT

The EQUIVALENCE statement causes more than one variable within a given program to share the same storage location. This is useful when the programmer desires to conserve storage space. The form of the statement is:

EQUIVALENCE (v_1, v_2 . . .) , . . .

where v represents a variable name. The inclusion of two or more variables within the parenthetical list indicates that these variables are to share the same memory location and thus have the same value. For example:

```
EQUIVALENCE(RED,BLUE)
```

The variables RED and BLUE are now of equal value. The subscripts of array variables must be integer constants. For example:

```
EQUIVALENCE(X,A(3),Y(2,1)),(BETA(2,2),ALPHA)
```

Because of core memory restrictions within the compiler, variables cannot appear in EQUIVALENCE statements more than once.

```
EQUIVALENCE(A,B,C)
```

is valid, but the statement:

```
EQUIVALENCE(A,B),(B,C)
```

would not compile correctly.

Variables may not appear in both EQUIVALENCE and COMMON statements.

Subprogram Statements

External subprograms are defined separately from the programs that call them, and are complete programs which conform to all the rules of FORTRAN programs. They are compiled as closed subroutines; that is, they appear only once in core memory regardless of the number or times they are used. External subprograms are defined by means of the statements FUNCTION and SUBROUTINE. Functions and subroutines must be compiled independently of the main program and then loaded together with the main program by the Linking Loader.

NOTE

Care should be exercised when naming a subprogram or subroutine. It must not have the same name as any of the FORTRAN library functions or subroutines, or assembler mnemonics or pseudo-ops, as errors are likely to result. The Library Functions are listed in this chapter, and the symbol table for the SABR Assembler is listed in Appendix C.

Subprogram definition statements may optionally contain dummy arguments representing the arguments of the subprogram. They are used as ordinary identifiers within the subprogram and are replaced by the actual arguments when the subprogram is executed.

FUNCTION SUBPROGRAMS

A function subprogram is a subprogram which is called from an arithmetic expression within the main program and returns a single numeric value. A function subprogram begins with a FUNCTION statement and ends with an END statement. It returns control to the calling program by means of one or more RETURN statements. The FUNCTION statement has the form:

FUNCTION identifier (a_1, a_2 . . . , a_n)

where FUNCTION (or FUNC) declares that the program which follows is a function subprogram, and identifier is the name of the function being defined. The identifier must appear as a scalar variable and be assigned a value during execution of the subprogram. This value is the function's value.

Arguments appearing in the list enclosed in parentheses are dummy arguments representing the function arguments. A function must have at least one dummy argument. The arguments must agree in number, order and type with the actual arguments used in the calling program. Function subprograms may be called with expressions and array names as arguments. The corresponding dummy arguments in the FUNCTION statement would then be scalar and array identifiers, respectively. Those representing array names must appear within the subprogram in a DIMENSION statement. Dimensions must be indicated as constants and should be smaller

than or equal to the dimensions of the corresponding arrays in the calling program. Dummy arguments to FUNCTION cannot appear in COMMON or EQUIVALENCE statements within the function subprogram.

A function should not modify any arguments which appear in the FORTRAN arithmetic expression calling the function. The only FORTRAN statements not allowed in a function subprogram are SUBROUTINE and other FUNCTION statements.

The type of function is determined by the first letter of the identifier used to name the function, in the same way as variable names.

The following short example calculates the gross salary of an individual on the basis of the number of hours he has worked (TIME) and his hourly wage (RATE). The function calculates time and a half for overtime beyond 40 hours. The function name is SUM.

```
      FUNCTION SUM(TIME,RATE)
      IF (TIME-40.)10,10,20
10    SUM = TIME * RATE
      RETURN
20    SUM = (40.*RATE) + (TIME-40.)*1.5*RATE
      RETURN
      END
```

Depending upon which path the program takes, control will return to the main program at one of the two RETURN statements with the answer. Assume that the main program is set up with a statement to read the employee's weekly record from a list of information prepared on the high-speed reader:

```
READ(2,5) NAME, NUM, NDEP, TIME, RATE
```

This statement reads the person's name, number, department number, time worked, and hourly wage. The main program then calculates his gross pay with a statement such as the following:

```
GROSS = SUM(TIME,RATE)
```

and goes on to calculate withholdings, etc.

SUBROUTINE SUBPROGRAMS

A subroutine subprogram is a subprogram which is called by the main program via a CALL statement, and may return several

or no values. The subprogram begins with a SUBROUTINE statement and returns control to the calling program by means of one or more RETURN statements. The SUBROUTINE statement has the form:

SUBROUTINE identifier (a_1, a_2 . . . a_n)

where SUBROUTINE declares the program which follows to be a subroutine subprogram and the identifier is the subroutine name. The arguments in the list enclosed in parentheses are dummy arguments representing the arguments of the subprogram. The dummy arguments must agree in number, order, and type with the actual arguments, if any, used by the calling program.

Subroutine subprograms may have expressions and array names as arguments. The dummy arguments may appear as scalar or array identifiers. Dummy identifiers which represent array names must be dimensioned within the subprogram by a DIMENSION statement. The dummy arguments must not appear in an EQUIVALENCE or COMMON statement in the subroutine subprogram.

A subroutine subprogram may use one or more of its dummy identifiers to represent results. The subprogram name is not used for the return of results. A subroutine subprogram need not have any arguments, or may use the arguments to return numbers to the calling program. Subroutines are generally used when the result of a subprogram is not a single value.

Example SUBROUTINE statements are as follows:

```
SUBROUTINE FACTO (COEFF,N,ROOTS)
SUBROUTINE RESID (NUM,N,DEN,M,RES)
SUBROUTINE SERIE
```

The only FORTRAN statements not allowed in a subroutine subprogram are FUNCTION and other SUBROUTINE statements.

The following short subroutine takes two integer numbers from the main program and exchanges their values. If this is to be done at several points in the main program, it is a procedure best performed by a subroutine.

```
SUBROUTINE ICHGE (I,J)
ITEM=I
I=J
J=ITEM
RETURN
END
```

The calling statement for this subroutine might look as follows:

```
CALL ICHGE (M,N)
```

where the values for the variables M and N are to be exchanged.

CALL Statement

The CALL statement assumes one of two forms:

CALL identifier

or CALL identifier (a_1, a_2 . . . , a_n)

The CALL statement is used to transfer control to a subroutine subprogram. The identifier is the subroutine name.

The arguments (indicated by a_1, through a_n) may be expressions or array identifiers. Arguments may be of any type, but must agree in number, order, type, and array size with the corresponding arguments in the SUBROUTINE statement of the called subroutine. Unlike a function, a subroutine may produce more than one value and cannot be referred to as a basic element in an expression.

A subroutine may use one or more of its arguments to return results to the calling program. If no arguments at all are required, the first form is used. For example:

```
CALL EXIT
CALL TEST (VALUE,123,275)
```

The identifier used to name the subroutine is not assigned a type and has no relation to the types of the arguments. Arguments which are constants or formed as expressions must not be modified by the subroutine.

RETURN Statement

The RETURN statement has the form:

RETURN

This statement returns control from a subprogram to the calling program. Each subprogram must contain at least one RETURN statement. Normally, the last statement executed in a subprogram is a RETURN statement; however, any number of RETURN statements may appear in a subprogram. The RETURN statement may not be used in a main program.

FUNCTION CALLS

Function calls are provided to facilitate the evaluation of functions such as sine, cosine, and square root. A function is a subprogram which acts upon one or more quantities (arguments) to produce a single quantity called the function value. A function call may be used in place of a variable name in any arithmetic expression.

Function calls are denoted by the identifier which names the function (i.e., SIN, COS, etc.) followed by an argument enclosed in parentheses as shown below:

IDENT (ARG, ARG, . . . , ARG)

where IDENT is the identifying function name and ARG is an argument which may be any expression. A function call is evaluated before the expression in which it is contained.

FUNCTION LIBRARY

The standard FORTRAN library contains built-in functions, including user-defined functions and subroutine subprograms.

Table 7-4 lists the built-in functions. These are open subroutines: they are incorporated into the compiled program each time the source program names them.

Function and subroutine subprograms are closed routines; their coding appears only once in the compiled program. These routines are entered from various points in a program through jump-type linkages.

NOTE

A FORTRAN compiler and its corresponding Library constitute an interlocking set of programs. No user should attempt to compile a program under OS/8 and load it with the paper tape FORTRAN, or vice versa. Similarly, programs developed with the current FORTRAN compiler should not be run under an old FORTRAN system.

Table 7-4 FORTRAN Function Library

Function	Definition	Type of Argument(s)
ABS(x)	the absolute value of x	real
IABS(x)	the absolute value of x	integer
FLOAT(x)	convert x from integer to real format	integer
IFIX(x)	convert x from real to integer format	real
IREM(0)	remainder of last integer divide is returned	integer
IREM(x/y)	remainder of x/y is returned	integer
EXP(x)	exponential of x, e^x	real
ALOG(x)	natural logarithm of x, $\log_e x$	real
SIN (x)	sine of x, where x is given in radians	real
COS (x)	cosine of x, where x is given in radians	real
TAN(x)	tangent of x, where x is given in radians	real
ATAN(x)	arc tangent of x, where x is given in radians	real
SQRT(x)	square root of x is returned	real
IRDSW(0)	read the console switch register, returning a decimal equivalent of the octal integer in the switch register. The switch register can be set before executing the FORTRAN program or, using the PAUSE statement, during execution.	integer

FLOATING POINT ARITHMETIC

In general, floating point arithmetic calculations are accurate to seven digits with the eighth digit being questionable. Subsequent digits are not significant even though several may be typed to satisfy a field width requirement. With the exception of the arctangent function, which is accurate to seven places over the entire range, results of function operations are accurate to six decimal places.

The floating point arithmetic routines check for both overflow and underflow. Overflow will cause the OVFL error message to be printed and program execution will be terminated. Underflow is detected but will not cause an error message. The arithmetic operation involved will yield a zero result.

DEVICE INDEPENDENT I/O AND CHAINING

OS/8 FORTRAN provides for device independent, file-oriented, formatted I/O through use of the device number 4 in the READ and WRITE statements and several utility subroutines. These are described below.

The IOPEN Subroutine

The subroutine IOPEN prepares the system to accept input from a specified device when device code 4 is used in a READ statement. IOPEN takes two arguments which are interpreted as Hollerith strings. After a

```
CALL IOPEN(A,B)
```

any READ statement reading from device 4 will read from the file specified by B (which must have the extension .DA) on the device specified by A. For example:

```
CALL IOPEN('DTA5','INPUT')
```

will prepare for input from the file DTA5:INPUT.DA

```
CALL IOPEN('F1',0)
```

will prepare for input from the device F1, which, in this case, is a non-file-structured device.

If the file and device names are input via READ statements which use A format in their FORMAT statements, then A6 format must be used. @ signs rather than spaces should be used to fill in empty characters. For example, the following statements are contained in a program:

```
        WRITE (1,20)
20      FORMAT ('ENTER FILE NAME')
        READ (1,22)FNAME
22      FORMAT (A6)
        CALL IOPEN('DSK',FNAME)
        .
        .
        .
```

The Teletype prints:

```
ENTER FILE NAME
```

and the user responds:

```
ABC@@@
```

The OOPEN Subroutine

The subroutine OOPEN prepares the system to send output to a specified device when device code 4 is used in a WRITE statement. The arguments of OOPEN are treated like those of IOPEN. Future WRITE statements using device 4 write on the device and file specified in the call to OOPEN. An error message is printed if the program has previously issued a CALL OOPEN without issuing a subsequent CALL OCLOSE. For example:

```
CALL OOPEN('PTP',0)
```

prepares device 4 to output on device PTP.

```
CALL OOPEN('SYS','LADE')
```

prepares device 4 to output to the file SYS:LADE.DA.

The OCLOSE Subroutine

The subroutine OCLOSE is called with no arguments. Its function is to terminate output on the output file opened by OOPEN. If OCLOSE is not called after a file has been written, that output file will never exist on the specified device.

The CHAIN Subroutine

A call to the subroutine CHAIN terminates execution of the calling program and starts execution of the core image on the system device as specified by the argument to CHAIN. Variables in common storage are not disturbed. For example:

```
CALL CHAIN('PROG2')
```

causes the file SYS:PROG2.SV to be loaded and started. Notice that PROG2 *must* be compiled and stored on the system device as a core image (.SV) file in order to be successfully accessed.

The EXIT Subroutine

To return to the Keyboard Monitor from a FORTRAN program, the EXIT subroutine is used, as follows:

```
CALL EXIT
```

DECTAPE I/O ROUTINES

RTAPE and WTAPE (read tape and write tape) are the DECtape read and write subprograms for the 8K FORTRAN and 8K SABR systems. For the paper tape FORTRAN system, these subprograms are furnished on one relocatable binary-coded paper tape which must be loaded into field 0 by the 8K Linking Loader, where they occupy one page of core.

RTAPE and WTAPE allow the user to read and write any amount of core-image data onto DECtape in absolute, non-file-structured data blocks. Many such data blocks may be stored on a single tape, and a block may be from 1 to 4096 words in length.

RTAPE and WTAPE are subprograms which may be called with standard, explicit CALL statements in any 8K FORTRAN or

SABR program. Each subprogram requires four arguments separated by commas. The arguments are the same for both subprograms and are formatted in the same manner. They specify the following:

1. DECtape unit number (from 0 to 7)
2. Number of the DECtape block at which transfer is to start. The user may direct the DECtape service routine to begin searching for the specified block in the forward direction

   rather than the usual backward direction by making this argument the two's complement of the block number. For additional information on this and other features the reader is referred to the *DECtape Programmer's Reference Manual* (DEC-08-SUCO-D).
3. Number of words to be transferred (1<N<4096).
4. Core address at which the transfer is to start.

The general form is:

CALL RTAPE (n_1, n_2, n_3, n_4)

where n_1 is the DECtape unit number, n_2 is the block number, n_3 is the number of words to be transferred, and n_4 is the starting address.

In 8K FORTRAN, an example CALL statement to RTAPE could be written in the following format (arguments are taken as decimal numbers):

```
CALL RTAPE(6,128,388,LOCA)
```

In this example, LOCA may or may not be in common.

As a typical example of the use of RTAPE and WTAPE, assume that the user wants to store the four arrays A, B, C, and D on a tape with word lengths of 2000, 400, 400, and 20 respectively.

Since PDP-8 DECtape is formatted with 1474 blocks (numbered 0-2701 octal) of 129 words each (for a total of 190,146 words) [7], A, B, C, and D will require 16, 4, 4, and 1 blocks respectively.

```
DIMENSION IDIR(258)
CALL RTAPE(5,2,258,IDIR)
```

would read Block 2 (OS/8 Block 1) of DECtape #5.

Each array must be stored beginning at the start of some DECtape block. The user may write these arrays on tape as follows:

```
CALL WTAPE(0,1,2000,A)
CALL WTAPE(0,17,400,B)
CALL WTAPE(0,21,400,C)
CALL WTAPE(0,25,20,D)
```

The user may also read or write a large array in sections by specifying only one DECtape block (129 words) at a time. For example, B could be read back into core as follows:

```
CALL RTAPE(0,17,258,B(1))
CALL RTAPE(0,19,129,B(259))
CALL RTAPE(0,20,13,B(388))
```

As shown above, it is possible to read or write less than 129 words starting at the beginning of a DECtape block. It is impossible, however, to read or write starting in the middle of a block. For example, the last 10 words of a DECtape block may not be read without reading the first 119 words as well.

---

[7] The block numbers used by RTAPE and WTAPE should not be confused with the record numbers used by OS/8. An OS/8 record is 256 words—roughly twice the size of a DECtape block. An RTAPE or WTAPE record number is exactly twice the corresponding OS/8 record number. For example, to read the first segment of the OS/8 directory on DECtape #5, the statements:

A DECtape read or write is normally initiated with a backward search for the desired block number. To save searching time, the user may request RTAPE or WTAPE to start the block number search in the forward direction. This is done by specifying the negative of the block number. This should be used only if the number of the next block to be referenced is at least ten block numbers greater than the last block number used. For example, if the user has just read array A and now wants array D, he may write:

```
CALL RTAPE(0,1,2000,A)
CALL RTAPE(0,-27,20,D)
```

The following section of a program demonstrates the use of DECtape I/O. Assume that values are already present on the DECtape.

```
      DIMENSION DATA(500)
      .
      .
      .
      NB=0
      SUM=0.
      DO 100 N=1,10
      CALL RTAPE(1,-NB,1500,DATA)
      TEM=0.
      DO 50 K=1,500
50    TEM=TEM+DATA(K)
      SUM=SUM+TEM
100   NB=NB+24
      AMEAN=SUM/5000.
      WRITE (1,110) SUM, AMEAN
      CALL EXIT
110   FORMAT ('SUM=',E15.7' MEAN=',E15.7///)
      END
```

OS/8 FORTRAN LIBRARY SUBROUTINES

Table 7-5 contains a summary of the OS/8 FORTRAN library subroutines. This list describes the routines available under OS/8 FORTRAN, their functions, and other routines which must also

be present in order for them to be used. The Subroutine Names listed are the files which comprise OS/8 Source DECtape #3 (available from the Software Distribution Center upon request).

Table 7-5 FORTRAN II Library Subroutines

Subroutine Name	Entry Points, or Defined External Symbols	Routines That are Pre-requisites	Core Require-ments (Pages)	Function the Routine Performs
IOH	'READ' 'WRITE' 'IOH'	FLOAT UTILTY INTEGR	11	Handles Input and Output Conversion
FLOAT	'FAD' 'FSB' 'FMP' 'FDV' 'STO' 'FLOT' 'FLOAT' 'FIX' 'IFIX' 'IFAD' 'ISTO' 'ABS' 'CHS'	UTILTY	5	Floating Point Arith-metic Package
UTILTY	'OPEN' 'GENIO' 'EXIT' 'ERROR' 'CKIO'	INTEGR	3	FORTRAN De-vice Routines, Error Exit, Normal Exit
POWERS	'IFPOW' 'FFPOW' 'EXP' 'ALOG'	FLOAT UTILTY IPOWRS INTEGR	3	Handles Num-bers to Floating Powers

Table 7-5 FORTRAN II Library Subroutines (Cont.)

Subroutine Name	Entry Points, or Defined External Symbols	Routines That are Pre-requisites	Core Require-ments (Pages)	Function the Routine Performs
INTEGR	'IREM' 'IABS' 'DIV' 'MPY' 'IRDSW' 'CLEAR' 'SUBSC'	UTILTY	2	Integer Math Package
TRIG	'SIN' 'COS' 'TAN'	FLOAT	2	Handles Sine, Cosine, and Tangent
ATAN	'ATAN'	FLOAT	2	Handles Arc-tangents
SQRT	'SQRT'	FLOAT UTILTY	1	Handles Square Roots
IPOWRS	'IIPOW' 'FIPOW'	FLOAT INTEGR	1	Handles Num-bers to Integer Powers
IOPEN	'IOPEN' 'OOPEN' 'OCLOS' 'CHAIN'	UTILTY	1	OS/8 Device-Independent I/O, and Chaining Routines
RWTAPE	'RTAPE' 'WTAPE'	UTILTY	1	OS/ Indepen-dent DECtape I/O Routines

MIXING SABR AND FORTRAN STATEMENTS

An S in column 1 of an input line identifies that line as containing SABR code. This feature is very useful for performing instructions which are undefined in the FORTRAN language. For example:

```
      DIMENSION M(10)
      .
      .
      J=M(1)
      DO 55 K=2,10
      L=M(K)
S     TAD     \L
S     AND     \J
S     DCA     \J
55    CONTINUE
```

This section of code will form the logical AND of M(1) through M(10) in the variable J.

Notice that whenever a FORTRAN variable is used in a SABR statement, the variable name is preceded by a backslash (\). FORTRAN line numbers referenced in SABR statements are also preceded by a backslash for identification purposes. (A backslash is produced by typing a SHIFT/L.)

Information on calling subroutines which are written in SABR assembly language from a FORTRAN program may be found in the SABR chapter.

SIZE OF A FORTRAN PROGRAM

The maximum size of any FORTRAN program is 36 octal or 30 decimal pages of code.

OS/8 can run FORTRAN programs in 8 to 32K of core. No one program or subprogram can be longer than 4K, however.

The user can estimate the size of his program as follows: Take the amount of core available on the system (at least 8K) and from it subtract 4K for the linkage subroutines, external symbol table, and I/O, math, error, and utility subroutines. From the remainder subtract the amount of storage required for data. The remaining space can be used to hold FORTRAN coding, at the rate of 50-70 FORTRAN statements per 1K of core.

One way to have a longer FORTRAN program in core than is usually possible is to divide a FORTRAN program into three chained segments:

Segment 1—inputs data into common storage
Segment 2—FORTRAN program for data processing
Segment 3—does output to desired device(s)

This gives two space advantages:

1. The entire program does not have to fit into available core, only the largest segment.
2. If no I/O statements are used in the middle (computational) segment, the I/O conversion routines will not be loaded with that segment. Since these routines occupy over 1100_{10} words, this technique allows the computational segment to be from 50 to 80 statements longer than a similar program containing I/O statements.

When chaining to a subroutine, the user must be sure he has compiled, loaded, and saved a complete runnable main program on the *system device*. This program is brought into core by the FORTRAN CHAIN subroutine.

FORTRAN STATEMENT SUMMARY

A summary of the statements available under OS/8 FORTRAN follows.

Table 7-6 FORTRAN Language Summary

Statement	Definition
Arithmetic Statements	
v=e	v is a variable (scalar or array); e is an expression.
Control Statements	
GOTO n	Transfer control to the statement numbered n.

Table 7-6 FORTRAN Language Summary (Cont.)

Statement	Definition
GOTO (n_1,n_2,...,n_i)j	Where n_1-n_i are statement numbers and j is a scalar integer variable. This statement transfers control to the j^{th} member of the series of n_i.
IF(expression)n_1,n_2,n_3	This statement transfers control to the statement numbered n_1,n_2, or n_3 if the value of the numeric expression is less than, equal to, or greater than zero, respectively. The expression can be simple or complex.
DO n i=m_1,m_2,m_3	Repeat execution through statement n, beginning with i=m_1, incrementing by m_3, while i is less than or equal to m_2. If m_3 is omitted, it is assumed to be 1. m's and i's cannot be subscripted. m's can be either integer numbers or integer variables; i is an integer variable.
CONTINUE	Dummy statement, used primarily as a target for transfers, particularly the last statement in the range of a DO loop. A DO loop need not end with a CONTINUE statement.
PAUSE PAUSE n	Temporarily suspend execution. The octal equivalent of the decimal number n is displayed in the accumulator. Program execution can be resumed by following the statement with a call to the OPEN subroutine.
STOP	Terminate execution.
END	Terminate compilation; must be the last statement in a program.
Input/Output Statements	
FORMAT (s_1,s_2,...,s_n)	Where S_1-S_n are data field specifications, this statement is used with either a READ or WRITE statement.

Table 7-6 FORTRAN Language Summary (Cont.)

Statement	Definition
READ (u,f) list	Where u is a device designation (integer constant or integer variable), f is a FORMAT statement number, and list is a list of variables.
WRITE (u,f) list	Where u is a device designation (integer constant or integer variable), f is a format statement number, and list is a list of variables.
Specification Statements	
COMMON $v_1, v_2,...,v_n$	Specified variables or arrays are stored in an area available to other programs.
DIMENSION $a_1,a_2,...,a_n$	Used to declare variable names to be array names and specify the number and bounds of each one and two dimensional array.
EQUIVALENCE $(v_1,v_2,...,)$, $(v_i,v_{i+1},...)$	The inclusion of two or more variable or array names in a parenthetical list indicates that the quantities in the list are to share the same memory location and hence have the same value. Subscripts of array variables must be integer constants. Names must not appear in both EQUIVALENCE and COMMON statements.
Subprogram Statements	
FUNCTION $v(a_1,a_2,...,a_n)$	Declares the program which follows to be a function subprogram. v is the name of the function being defined. v must appear as a scalar variable and be assigned a value during execution of the subprogram.

Table 7-6 FORTRAN Language Summary (Cont.)

Statement	Definition
SUBROUTINE v(a_1,a_2,...,a_n)	Declares the program which follows to be a subroutine subprogram. The arguments in the list(s) are dummy arguments representing the arguments of the subprogram. Dummy arguments must agree in number, order, and type with the arguments used by the calling program.

CALL v
CALL v (a_1,a_2,...,a_n)
	Statement used to transfer control to a subroutine subprogram. v is the subroutine name in the SUBROUTINE statement. The arguments can be of any type, but must agree in number, order, type and array size with the arguments in the SUBROUTINE statement. One or more of the arguments can be used to return results to the calling program. For example:
	`CALL EXIT`
	`CALL TEXT (VALUE,123,275)`
	`CALL TECK ('MAS',3)`

RETURN	Returns control from a subprogram to the calling program. Each subprogram must contain at least one RETURN statement. RETURN cannot be used in the main program.

FORTRAN ERROR MESSAGES

Compiler Error Messages

The following OS/8 FORTRAN Compiler error messages are self-explanatory.

ARITHMETIC EXPRESSION TOO COMPLEX
EXCESSIVE SUBSCRIPTS
ILLEGAL ARITHMETIC EXPRESSION
ILLEGAL CONSTANT
ILLEGAL CONTINUATION
ILLEGAL EQUIVALENCING
ILLEGAL OR EXCESSIVE DO NESTING
ILLEGAL STATEMENT
ILLEGAL STATEMENT NUMBER
ILLEGAL VARIABLE
MIXED MODE EXPRESSION
SYMBOL TABLE EXCEEDED
SYNTAX ERROR (usually indicates illegal punctuation)
SUBR. OR FUNCT. STMT. NOT FIRST

In addition, OS/8 FORTRAN contains the following error messages:

Message	Explanation
COMPILER MALFUNCTION	The meaning of this message has been extended to cover various unlikely Monitor errors.
IO	A device handler has signalled an I/O error.
NO END STATEMENT	The input to the Compiler has been exhausted.
NO ROOM FOR OUTPUT	The file FORTRN.TM cannot fit on the system device.
SABR.SV NOT FOUND	The SABR assembler is not present on the system device.

Library Error Messages

During execution, the various library programs check for certain errors and print error messages in the form:

XXXX ERROR AT LOC NNNNN

where XXXX is the error code and NNNNN is the location of the error.

Table 7-7 FORTRAN Library Error Messages

Error Code	Meaning
The following errors are fatal and cause a return to the Keyboard Monitor.	
ALOG	Attempt to compute log of negative number.
IOER	One of the following has occurred: 1. Device-independent input or output attempted without /I or /O options, or user attempted to specify a device requiring a two-page handler for device-independent I/O without using the /H option. 2. Bad arguments to IOPEN or OOPEN, or 3. Transmission error while doing I/O.
CHER	File specified as argument to CHAIN not found on system device.
FMT1	Invalid Format statement.
The following input errors are fatal unless input is coming from the Teletype, in which case the entire READ statement is tried again.	
FMT2	Illegal character in I format.
FMT3	Illegal character in F or E format.
The following errors do not termniate execution of the user's program.	
DIVZ	Division by zero—very large number is returned.
EXP	Argument to EXP too large—very large number is returned.
OVFL	Floating point overflow—very large number is returned.
FLPW	Negative number raised to floating point power—absolute value taken.
SQRT	Attempt to take square root of negative number —absolute value used.
FIX	Attempt to fix a number >2047; 2047 is returned.

In addition, the error message:

USER ERROR 1 AT XXXX

means that the user tried to reference an entry point of a program which was not loaded, or possibly that he failed to define a subscripted variable in a DIMENSION statement. XXXX has no meaning.

To pinpoint the location of a library program execution error:

1. Determine, from the storage map, the next lowest numbered location (external symbol) which is the entry point of the program or subprogram containing the error.
2. Subtract, in octal, the entry point location of the program or subprogram containing the error from the location of the error indicated in the error message.
3. From the assembly symbol table, determine the relative address of the external symbol found in step 1 and add that relative address to the result of step 2.
4. The sum of step 3 is the relative address of the error, which can then be compared with the relative addresses of the numbered statements in the program.

Undefined statement numbers are not detected until the assembly phase, at which time a U error message is given. (Refer to the list of SABR error messages.)

chapter 8

FORTRAN IV

FORTRAN IV SYSTEM OVERVIEW

OS/8 FORTRAN IV provides full standard ANSI FORTRAN IV under the OS/8 operating system. The FORTRAN IV package requires a minimum hardware environment consisting of a PDP-8 family processor with at least 8K of mainframe memory, a console terminal, and at least 96K of mass storage. The system is automatically self-expanding to employ a KE8-E Extended Arithmetic Element, FPP-12 Floating Point Processor, up to 32K of mainframe memory, and any bulk storage or peripheral I/O devices that may be present in the system.

Although such factors as maximum program size and minimum execution time depend heavily on the hardware configuration on which any program is run, OS/8 FORTRAN IV affords the full capability of the FORTRAN IV language, even on a minimum configuration, subject only to the restriction that double precision and complex number operations require an FPP-12 with extended precision option. The system is highly optimized with respect to memory requirements, and an overlay feature is included that can permit programs requiring up to 300K of virtual storage to run on a PDP-8 or PDP-12. The library functions permit the user to access a number of laboratory peripherals, to evaluate a number of transcendental functions, to manipulate alphanumeric strings, and to output to a standard incremental plotter.

A FORTRAN IV program written by the user is called a source program, to distinguish it from the various object programs generated by the OS/8 FORTRAN IV system. Source programs may be prepared off line on punched cards or low-speed paper tape; however, it is usually most convenient to prepare source programs on line by means of an editing program such as TECO or EDIT. (See Figure 8-1.) The source file produced in this manner is an image of the corresponding punched-card file, with carriage

return and line feed characters separating adjacent statements (that would otherwise appear on adjacent punched cards) and ASCII spaces or tabs entered in place of blank columns. Because of the close analogy between punched card source files and other types of source files, the terms "character" and "column" are used interchangeably in this manual.

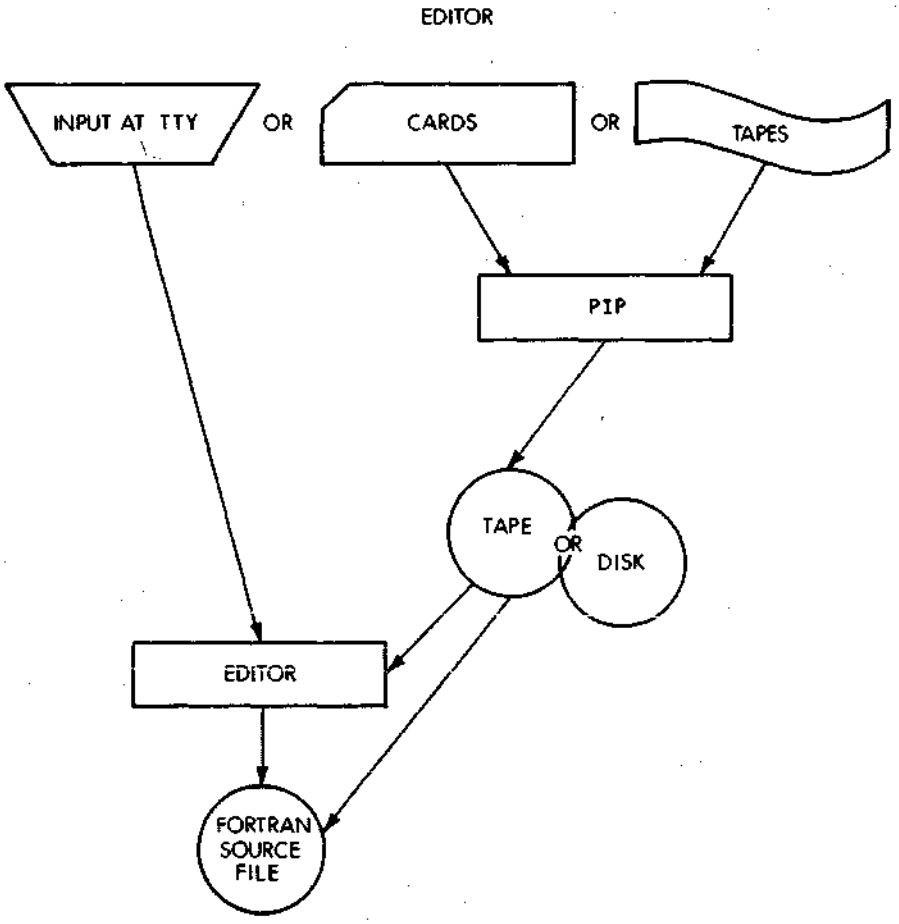


Figure 8-1 Preparing a FORTRAN IV Source File

Once a source program has been prepared, it is supplied as input to the FORTRAN IV compiler, which translates each FORTRAN statement into one or more RALF (Relocatable Assembly Language, Floating-point) statements and produces an output file containing an assembly language version of the source program, plus an optional annotated listing of the source. (See Figure 8-2.)

This is accomplished in three passes. System program F4.SV begins compilation by building a symbol table and generating intermediate code. F4 chains to PASS2.SV automatically, and PASS2 calls PASS20.SV to complete the translation into assembly language during compilation pass 2. If a source listing was requested, PASS20 chains to PASS3.SV automatically, and PASS3 generates the listing during pass 3. Like PASS2, PASS20 and PASS3 are never accessed directly by the user.

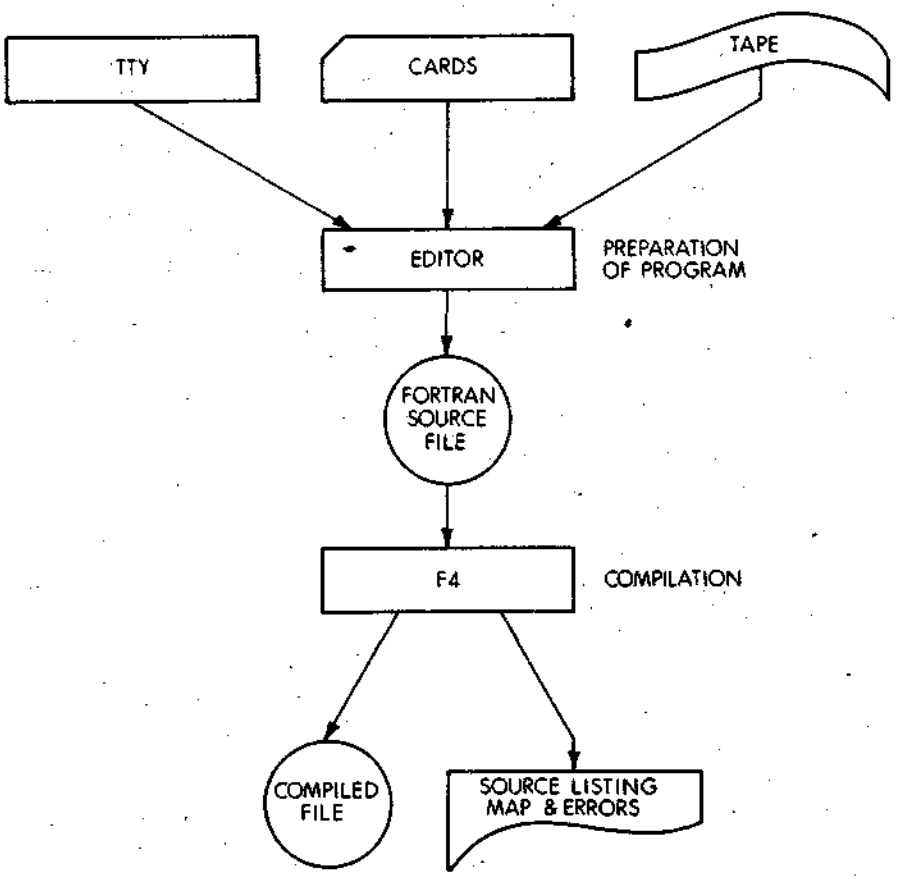


Figure 8-2 Compiling a Source File

The RALF assembly language output produced by the compiler must be assembled by system program RALF.SV, the RALF assembler. (See Chapter 5 for a description of the RALF assembler.) During assembly, each RALF assembly language statement is translated into one or more instructions for either the PDP-8 processor or the FPP and an output file is created containing a

relocatable binary version of the assembly language input. This is accomplished in two passes; a third pass is executed to generate an annotated listing of the assembly language input file, if requested.

The relocatable binary file produced by the RALF assembler is a machine language version of a single program or subroutine. This file, called a RALF module, must be linked with its main program (if it is a subroutine) and with any other subroutines, including subroutines from the system library (e.g., FORLIB.RL) that it requires in order to execute. System program LOAD.SV, the OS/8 FORTRAN IV loader, accepts a list of RALF module specifications from the console terminal and builds a loader image file containing a relocated main program linked to relocated versions of all subroutines and library components that the mainline requires in order to execute. (See Figure 8-3.)

The loader image file is an executable core load, complete except for run-time I/O specifications. It may be stored on any mass storage (directory) device and executed whenever desired. The loader also produces an optional symbol map that indicates the core storage requirements of the linked and relocated program. The overlay feature of the loader permits certain segments of a program to be stored in the loader image file during execution and read into core memory only as needed, which effectively provides a tenfold increase in maximum program size.

The loader image file produced by the loader is read and executed by system program FRTS.SV, the OS/8 FORTRAN IV run-time system, which also configures an I/O supervisor to handle any FORTRAN input or output in accordance with run-time I/O specifications. This makes the full I/O device independence of the OS/8 operating system available to every FORTRAN IV program, and permits FORTRAN programs to be written without concern for, or even knowledge of, the hardware configuration on which they will be executed. The run-time system assigns I/O device handlers to the I/O unit numbers referenced by the FORTRAN program, allocates I/O buffer space, and also diagnoses certain types of errors that occur when the loader image file is read into core. If no errors of this sort are encountered, the run-time system starts the FORTRAN program and monitors execution to check for run-time errors involving data I/O, numeric overflow, hardware malfunctions, and the like. Run-time errors are identified at the console terminal, and, when a run-time error occurs, the

system also provides complete error traceback to identify the full sequence of FORTRAN statements that terminated in the error condition.

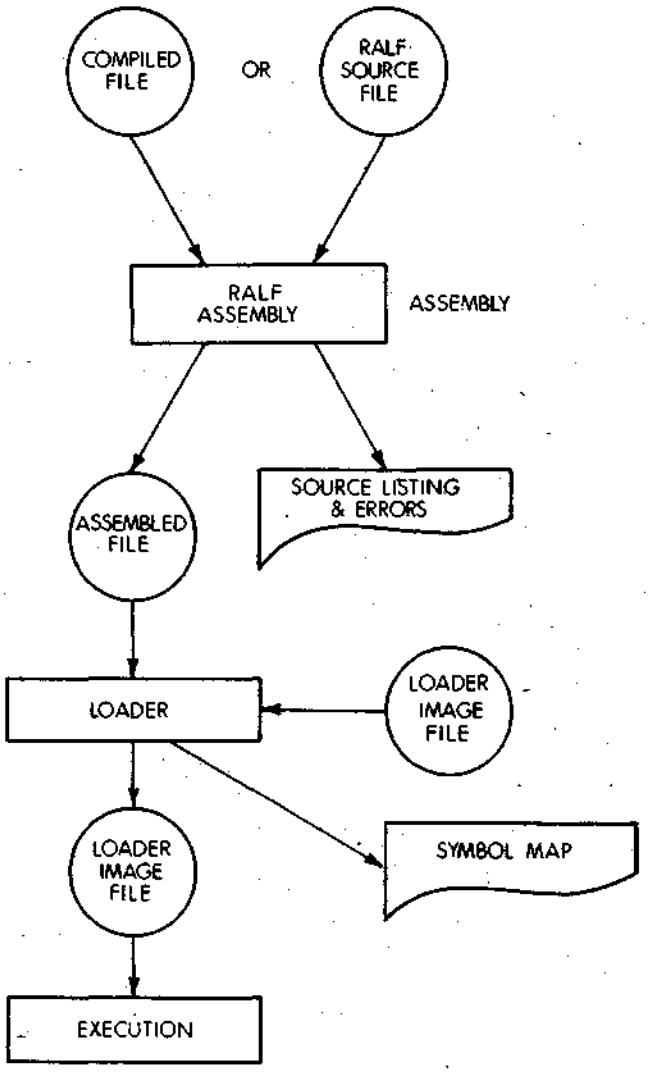


Figure 8-3 Assembling, Loading, and Executing a RALF File

The compiler, assembler, loader, and run-time system each accept standard OS/8 Command Decoder option specifications, as do most OS/8 programs. The option specifications are alphanumeric characters which may be thought of as switches that, by their presence or absence, enable or disable certain program

features and conventions. For example, specifying the N option to the compiler suppresses compilation of internal sequence numbers, thereby reducing program memory requirements (at the cost of preventing full error traceback during execution). Thus, N is one of the compiler run-time option specifications which may be requested to modify the usual compilation procedure. In this context, run-time refers to the time at which the compiler, or other system program is executed, rather than the time at which the FORTRAN program is executed.

A FORTRAN source program may be executed by calling the compiler to convert the source into RALF assembly language, then calling the assembler to produce a relocatable binary file, calling the loader to link and relocate the binary file, and finally calling the run-time system to load the program and supervise execution. OS/8 FORTRAN IV provides a program chaining feature that can simplify or eliminate this sequence of program calls in most cases. When chaining is requested, the first system program to be executed automatically calls the next program in the compiler/assembler/loader/run-time system sequence. When the compiler chains to the assembler, for example, the five programs (the compiler consists of four programs) function as a single unit that accepts FORTRAN source language input and generates relocatable binary output, suitable for use as input to the loader. In this manner, simple FORTRAN programs may be compiled, assembled, relocated, loaded, and executed, all as the result of a single Keyboard Monitor or CCL command. More complicated programs involving subroutines and, perhaps, overlays, do not admit to a high degree of chaining because a great deal of user input in the form of run-time option specifications may be required at some point in the chain. In general, however, it is usually most convenient to chain from the compiler to the assembler (combining compilation and assembly into a single operation) and from the loader to the run-time system (combining relocation, loading, and execution).

Errors encountered by the various system programs do not result in termination of program chaining unless the error is such that it is impossible for execution to continue. This permits the system to locate and identify as many errors as possible before returning control to the Keyboard Monitor. When chaining is

requested, intermediate output files produced by one system program are deleted automatically after they have been read as input by the next program in the chain sequence. This serves to optimize storage requirements and minimize access time, particularly on DECtape and LINCtape based systems.

The OS/8 FORTRAN IV system also includes FORLIB.RL, a library of FORTRAN functions and subroutines, plus LIBRA, the system librarian program. Almost every FORTRAN program executes calls to library functions and subroutines which perform such tasks as mathematical function evaluation, data I/O and numeric conversion. When the loader recognizes that a program or subroutine has called a library component, it copies a relocated version of the referenced library routine into the loader image file and links it to the calling routine. LIBRA is used to maintain the library by inserting or deleting library functions or subroutines, which are simply assembled FORTRAN files or specially coded RALF modules. LIBRA may also be used to create alternate libraries for use in place of the standard system library.

Because it affords full I/O device independence, highly optimized memory and bulk storage, program chaining, and a variety of run-time options, OS/8 FORTRAN IV is necessarily somewhat complicated. In order to use the system most efficiently, it is important to identify the four processes that must be performed, in the proper sequence, to execute a FORTRAN source program:

Process	Performed by
COMPILATION	FORTRAN IV compiler (F4, PASS2, PASS20 and PASS3).
ASSEMBLY	RALF assembler (RALF).
RELOCATION	FORTRAN loader (LOAD) using system library.
EXECUTION	FORTRAN run-time system (FRTS).

It is also important to identify the types of input that must be supplied to each process listed above and the types of output that will be produced. The OS/8 FORTRAN IV system accepts user-generated FORTRAN source programs (supplied as input to the compiler) and user-written RALF assembly language files (supplied to the assembler) as input. It generates four types of output files:

a. RALF assembly language files generated by the compiler and read as input by the assembler. Compiler output is functionally equivalent to user-written RALF language input.
b. Relocatable binary files generated by the assembler and read as input by the loader.
c. Loader image files generated by the loader and read as input by the run-time system. Once a program has been written and debugged, it may be stored as a loader image file and executed whenever required without the necessity for further compilation, assembly, or relocation.
d. Optional listing files including the FORTRAN source listing produced by the compiler, the RALF language listing produced by the assembler, and a symbol map produced by the loader.

In addition, the FORTRAN program itself usually reads and writes data files under the supervision of the run-time system; FORTRAN I/O files are treated separately in the section on the FORTRAN IV Run-Time System.

Every FORTRAN source program thus generates up to three object files, aside from any I/O files that may be read or written during execution, and up to three listing files. System-generated files are most conveniently identified by assigning them the same file name as the source from which they were produced and a file extension that identifies them by type. Table 8-1 lists the standard file extensions used to identify various types of source and system-generated files. The standard extensions are called default extensions because, when any output file name is specified with a null extension, the appropriate standard extension is appended by default. Thus, specifying file "PROG" or "PROG." to the RALF assembler, for example, causes the relocatable binary output from the assembly to be written on file "SYS:PROG.RL" where "SYS:" is the default device when a file name is explicitly defined and ".RL" is the default extension for relocatable binary files. Specifying a null file causes this output to be routed to file "DSK:FORTRN.RL" where "DSK:" is the OS/8 default device and "FORTRN" is the default output file name. For clarity, all examples in this chapter will use either null or default extensions, although the user may explicitly specify any extension desired.

Table 8-1 Standard FORTRAN IV File Extensions

Extension	File Type
.FT	FORTRAN language source file.
.RA	RALF assembly language file.
.RL	Relocatable binary (assembler output).
.LD	Loader image.
.LS	Listing or symbol map.
.TM	System temporary file. Created by certain multipass programs and normally deleted automatically after use.

This chapter assumes that the reader is familiar with the OS/8 operating system; however, all material has been presented in a manner that requires minimal experience with OS/8. Every reader should understand the use of the OS/8 Keyboard Monitor (although only the monitor R command is referenced here) and the OS/8 Command Decoder. In particular, notice that all Command Decoder file/option specifications presented here are illustrated in a standard format which may not be the most convenient format for an experienced user's particular application. In addition, the Command Decoder provides file storage optimization features which may be invaluable in many applications, but are not covered in this chapter. DECtape and LINCtape users will benefit from an understanding of the OS/8 file structure, so that they may assign I/O files in a manner that minimizes access time on tape-based systems.

The FORTRAN IV system of programs may be entered through the CCL commands COMPILE, EXECUTE, and LOAD. These commands are described in the CCL section of Chapter 1.

THE FORTRAN IV COMPILER

The OS/8 FORTRAN IV compiler accepts one FORTRAN source language program or subroutine as input, examines each FORTRAN statement for validity, and produces a list of error diagnostics plus a RALF assembly language version of the source program, along with an optional annotated source listing, as output. A job containing one or more subroutines is run by compiling and assembling the main program and each subroutine separately, then combining them with the loader. F4 terminates a compilation by chaining to the RALF assembler automatically unless it was

requested to return to the Keyboard Monitor. The compiler is called by typing:

```
R F4
```

(terminated by a carriage return) in response to the dot generated by the Keyboard Monitor. F4 may also be called via the CCL command COMPILE (see the CCL section in Chapter 1). It replies by loading the OS/8 Command Decoder, which accepts and decodes a standard command line that designates 0 to 3 output files, 1 to 9 input files, and any run-time option specifications. The file/option specification command line is entered by typing:

```
DEV:RALF.RA,DEV:LIST.LS,DEV:MAP.LS<DEV:IF1.FT, . . . ,DEV:IF9.FT(options)
```

(terminated by a carriage return) in response to the asterisk generated by the Command Decoder, where DEV:RALF.RA, DEV:LIST.LS, and DEV:MAP.LS are output files, RALF assembly source file, listing file, and loader symbol map file, respectively. The files DEV:IF1.FT, . . . ,DEV:IF9.FT are input files 1 to 9. Options is a string of alphabetic characters, enclosed in parentheses, that designates any run-time options desired. The "←" character may be used in place of the "<" character to separate output file specifications from input file specifications. The parentheses may be omitted if each run-time option specification character is preceded by a "/" character.

When any input file name is entered with a null extension, the compiler will search for the indicated file name with an assumed extension of ".FT", and, if this is unsuccessful, it will then search for the indicated file with a null extension. If the first output file RALF.RA is entered with a null extension, the compiler appends the default extension ".RA". If the second output file is a directory device file with a null extension, the compiler appends the default extension ".LS". Note that unless chaining to RALF, the first output file is always written onto the OS/8 system device; any user device specification entered for this file will be ignored when the A option is specified. When there is more than one input file, all of the input files are assumed to contain a single FORTRAN program or subroutine.

After accepting and decoding the file/option specification command, the compiler reads the input files in the order they were entered, and compiles each FORTRAN source statement until an END statement is encountered. Any text following the first END statement is ignored. The compiler then writes a RALF assembly language version of the source program onto the first output file, or onto file SYS:FORTRN.RA if no first output file was specified. It also copies an annotated source program listing onto the second output file; however, this listing is not produced unless a second output file was specifically defined. The third output file is not used by the compiler; it receives a loader symbol map only when chaining to the loader.

An internal statement number (ISN) is assigned to each FORTRAN IV statement sequentially, in octal, beginning with ISN 2 at the first FORTRAN statement. When an error is encountered during compilation, the compiler prints a 2-character error code, followed by the ISN of the offending statement, on the console terminal during pass 2. An extended error message is printed below every erroneous statement in the listing, provided that a listing is produced. Certain errors cause an immediate return to the Keyboard Monitor, however, in which case the listing file is never produced. Table 8-3 lists the FORTRAN compiler error messages and describes the error condition indicated by each message.

The compiler accepts four run-time option specifications, listed in Table 8-2, any combination of which may be requested by entering the appropriate alphabetic character(s) in the Command Decoder file/option specification line. Any run-time options recognized by the RALF assembler, the loader, or the run-time system may be entered along with the compiler options; they will be passed to the assembler automatically unless chaining is suppressed (by an error condition or the A option) in which case they will be ignored.

Table 8-2 FORTRAN IV Compiler Run-Time Options

Option	Operation
A	Return to the Keyboard Monitor when compilation is complete. If the A option is not requested, the compiler will automatically chain to the RALF assembler.
F	Produce an annotated listing of the RALF assembly language output file. The listing is actually produced by the assembler; thus, the F option is only valid when chaining to RALF. The listing is routed to the same output file as the FORTRAN source listing. It will overwrite the FORTRAN listing if the second output file resides on a directory device. It will not be produced if a second output file was not specifically defined.
N	Suppress compilation of ISNs. This reduces program memory requirements by two words per executable statement; however it also prevents full error traceback at run time.
Q	Optimize cross-statement subscripting during compilation. This option should not be requested when any variable which appears in a subscript is modified either by referencing a variable equivalent to it or via a SUBROUTINE or FUNCTION call (whether as an argument or through COMMON).

Examples

Compile, assemble, load, and execute a FORTRAN IV source program:

```
.R F4
*PROG/G
```

Compiles DSK:PROG.FT or DSK:PROG into DSK:FORTRN.RA, assembles it into DSK:FORTRN.RL, links it into DSK:FORTRN.LD, then loads it into core and executes it. No listing files are produced.

Compile any source program by calling F4 and specifying the file (or files) containing the source as input:

```
.R F4
*PROG/A
```

Compiles DSK:PROG.FT or else DSK:PROG. into SYS:FORTRN.RA. The back-arrow is optional when there are no output file specifications.

```
.R F4
*SYS:PROG.FT(NA)
```

Compiles SYS:PROG.FT into SYS:FORTRN.RA under the N option.

Obtain a source listing with error messages by specifying a listing output file as the second output file. In these examples, the first output file is a null file.

```
.R F4
*,LPT:<PROG/A
```

Identical to the first example above, except that a listing is produced on the line printer.

```
.R F4
*,DTA1:PROG<DTA2:PROG.FT/A/N
```

Compiles DTA2:PROG.FT into SYS:FORTRN.RA and writes a source listing onto file DTA1:PROG.LS under the N option.

Designate a specific output file to receive the compiler output by specifying it as the first output file:

```
.R F4
*PROG<PROG/A
```

Compiles DSK:PROG.FT or else DSK:PROG. into SYS:PROG.RA.

```
.R F4
*WHEN.RA,WHERE.LS<LTA0:WHAT(AQ)
```

Compiles LTA0:WHAT.FT or else LTA0:WHAT. into SYS:WHEN.RA with a listing routed to DSK:WHERE.LS under the Q option.

Compiler Error Messages

During compilation pass 2, error messages are printed at the console terminal as a 2-character error message followed by the ISN of the erroneous statement. Typing CTRL/O at the terminal suppresses the printing of error messages. If a listing was requested, an extended error message is appended to the listing, immediately following the erroneous statement, during pass 3. Except where indicated in Table 8-3, errors located by the compiler do not halt processing.

Table 8-3 FORTRAN IV Compiler Error Messages

Error Code	Meaning
AA	More than six subroutine arguments are arrays.
AS	Bad ASSIGN statement.
BD	Bad dimensions (too big, or syntax) in DIMENSION, COMMON, or type declaration.
BS	Illegal in BLOCK DATA program.
CL	Bad COMPLEX literal.
CO	Syntax error in COMMON statement.
DA	Bad syntax in DATA statement.
DE	This type of statement illegal as end of DO loop (i.e., GO TO, another DO).
DF	Bad DEFINE FILE statement.
DH	Hollerith field error in DATA statement.
DL	Data list and variable list are not same length.
DN	DO-end missing or incorrectly nested. This message is not printed during pass 3. It is followed by the statement number of the erroneous statement, rather than the ISN.
DO	Syntax error in DO or implied DO.
DP	DO loop parameter not integer or real.
EX	Syntax error in EXTERNAL statement.
GT	Syntax error in GO TO statement.
GV	Assigned or computed GO TO variable must be integer or real.
HO	Hollerith field error.
IE	Error reading input file. Control returns to the Keyboard Monitor.
IF	Logical IF statement cannot be used with DO, DATA, INTEGER, etc.
LI	Argument of logical IF is not type Logical.
LT	Input line too long, too many continuations.
MK	Misspelled keyword.
ML	Multiply defined line number.
MM	Mismatched parenthesis.
MO	Expected operand is missing.
MT	Mixed variable types (other than integer and real).
OF	Error writing output file. Control returns to the Keyboard Monitor.
OP	Illegal operator.
OT	Type / operator use illegal (e.g., A.AND.B where A and / or B not typed Logical).
PD	Compiler stack overflow; statement too big and/or too many nested loops.
PH	Bad program header line.

Table 8-3 FORTRAN IV Compiler Error Messages (Cont.)

Error Code	Meaning
QL	Nesting error in EQUIVALENCE statement.
QS	Syntax error in EQUIVALENCE statement.
RD	Attempt to re-define the dimensions of a variable.
RT	Attempt to re-define the type of variable.
RW	Syntax error in READ/WRITE statement.
SF	Bad arithmetic statement function.
SN	Illegal subroutine name in CALL.
SS	Error in subscript expression, i.e., wrong number, syntax.
ST	Compiler symbol table full, program too big. Causes an immediate return to the Keyboard Monitor.
SY	System error, i.e., PASS20.SV or PASS2.SV missing, or no room on system for output file. Causes an immediate return to the Keyboard Monitor.
TD	Bad syntax in type declaration statement.
US	Undefined statement number. This message is not printed during pass 3. It is followed by the statement number of the erroneous statement, rather than the ISN.
VE	Version error. One of the compiler programs is absent from SYS: or is present in the wrong version.

THE RALF ASSEMBLER

The RALF assembler accepts one RALF assembly language program or subroutine as input and produces a relocatable binary file, called a RALF module, as output. An optional annotated listing of the input file may also be produced. RALF terminates an assembly by returning to the Keyboard Monitor unless it was requested to chain to the loader.

A RALF module is composed of an external symbol dictionary (ESD table) and associated text. The ESD table lists all symbols defined in the RALF input file, which may be sections, entry points, or externs. Each of these symbols is assigned a relative address to be used by the loader when it relocates the relative code by assigning absolute core addresses. The text produced by RALF is a relocatable binary version of the assembly language input file. All text addresses are relative to the ESD table symbols.

A section can be thought of as a contiguous block of relocatable code having a definite beginning and end, which is temporarily

assigned a relative starting address of 00000. A RALF file can have more than one section defined in its ESD table. For example, consider a subroutine containing a COMMON section which is assembled by RALF. Both COMMON and the subroutine itself are sections. An entry point is a location within a given section that is referenced by code in other sections. An extern is a section or entry point in some other module that is referenced within the module currently being assembled.

Unless the A option is specified to the FORTRAN IV compiler, the RALF assembler is called automatically to assemble the output of a successful compilation. In this case, RALF reads the assembly language file just produced by the compiler as input and routes its output, consisting of the assembled RALF module, to the first output file that was specified to the compiler. If this file had a null extension, the default extension ".RL" is supplied. If no first output file was specified, the module is written onto default file SYS:FORTRN.RL.

The RALF language output produced by the compiler is then deleted, and an annotated listing of the RALF assembly language input is written on the second output file specified to the compiler, provided that a second output file and the F option were both specified. This listing will overwrite the compiler source listing if the second output file is a directory device file. Note, however, that the RALF language listing is rarely required for most applications, and should not be routinely requested.

The RALF assembler might also be called separately to assemble the output of a compilation produced under the A option or to assemble a user-generated file written in RALF assembly language. This is accomplished by typing:

```
R RALF
```

(terminated by a carriage return) in response to the dot generated by the Keyboard Monitor. RALF replies by loading the OS/8 Command Decoder, which accepts and decodes a standard command line that designates 0 to 3 output files, 1 to 9 input files, and any run-time option specifications. The file/option specification command line is entered by typing:

```
DEV:RALF.RA,DEV:LIST.LS,DEV:MAP.LS<DEV:IF1.R A, . . . ,DEV:IF9.RA(options)
```

(terminated by a carriage return) in response to the asterisk generated by the Command Decoder, where DEV:RALF.RA, DEV:LIST.LS and DEV:MAP.LS are the relocatable binary RALF module, annotated listing of RALF source, and loader symbol map, respectively; DEV:IF1.RA, . . . ,DEV:IF9.RA are input files 1 to 9; and "options" is a string of alphabetic characters that designates any run-time options desired. If any input file name is entered with a null extension, the assembler will search for the indicated file name with an assumed extension of ".RA" and, if this is unsuccessful, it will then search for the indicated file with a null extension. If the first output file is entered with a null extension, the assembler appends the default extension ".RL". If the second output file is a directory device file with a null extension, the assembler appends the default extension ".LS".

When there is more than one input file, all of the input files are assumed to contain the assembly language source for a single RALF module. After accepting and decoding the file/option specification command, RALF reads the input files in the order they were entered, and assembles every RALF language statement. RALF terminates the assembly by writing a relocatable binary version of the input program or subroutine onto the first output file, or onto file SYS:FORTRN.RL if no output files were specified. It also copies an annotated source listing and symbol table onto the second output file; however, this listing is not produced unless a second output file was specifically defined. The third output file is not used by the assembler; it receives a loader symbol map only when chaining to the loader.

When an error is encountered during assembly, the assembler prints a 2-character error code, followed by the label associated with the erroneous statement, on the console terminal during pass 2. Error codes are also appended to the listing, on a line by themselves immediately preceding the statement to which they apply (except EG, which follows the line in error). Certain errors cause an immediate return to the Keyboard Monitor, however, in which case the listing is never produced. Table 5-2 lists the RALF assembler error messages and describes the error condition indicated by each message.

The assembler accepts the three run-time option specifications listed in Table 8-4, any combination of which may be requested by entering the appropriate alphabetic character(s) in the Command

Decoder file/option specification line. Any options recognized by the loader or the run-time system may be entered along with the assembler options; they will be passed to the loader automatically unless chaining is suppressed (by an error condition or omission of the L option specification), in which case they will be ignored.

Table 8-4 RALF Assembler Run-Time Options

Option	Operation
G	Chain to the loader when assembly is complete and chain to the run-time system following creation of a loader image file.
L	Chain to the loader when assembly is complete. If the L option is not specified, RALF will return to the Keyboard Monitor upon completion.
T	Suppress the RALF assembly language listing and produce only a symbol table. The T option is ignored by the assembler when a second output file was not specifically defined. When chaining from the compiler, it is ignored unless the F option and a listing output file were both specified.

The symbol table produced by RALF and appended to the RALF language listing includes the assembler version number, system date, and the listing page number, followed by the number of errors encountered during assembly, the number of symbols defined in the program, and the number of absolute references encountered in FPP instructions. All symbols referenced during the assembly are then listed in alphabetical order, from left to right across the page. An alphabetic code follows certain classes of symbols and identifies them by type. The alphabetic codes are:

C = symbol names a COMMON section
F = symbol names a FIELD1 section
S = symbol is the name of a section
U = symbol is undefined
X = symbol is external to this assembly
Z = symbol names a COMMZ section
8 = symbol names an 8-mode section

If no alphabetic code is shown, the symbol is an ordinary address symbol. A numeric code is also printed after each symbol in the list. The numeric code indicates the relative octal value of the symbol except for the case of:

C, F, S, Z, or 8 codes	where the numeric code indicates the length of the section or common block.
U or X codes	where 00000 indicates undefined or external symbols.

Examples

When chaining from the compiler to the assembler, RALF deletes the compiler output after reading it as input:

```
.R F4
*PROG
```

Produces RALF module SYS:FORTRN.RL and deletes compiler output file SYS:FORTRAN.RA.

```
.R F4
*PROG.V3,LPT:<PROG/F
```

Produces RALF module SYS:PROG.V3 and lists both the FORTRAN source and the RALF language compiler output on the line printer.

```
.R F4
*DTA2:OBJ,DTA1:LIST<DTA2:PROG(TF)
```

Produces RALF module DTA2:OBJ.RL and writes a symbol map onto file DTA1:LIST.LS. The FORTRAN source listing is overwritten and destroyed.

When calling the assembler to assemble and relocate the output of a successful compilation produced under the A option or a user-written RALF language source, the procedure is closely analogous to that for running the compiler:

```
.R RALF
*PROG
```

Assembles DSK:PROG.RA or else DSK:PROG. into SYS:FORTRN.RL.

```
.R RALF
*,SYS:LIST<DTA1:FILE.RA
```

Assembles DTA1:FILE.RA into SYS:FORTRN.RL and writes a listing on SYS:LIST.LS.

```
.R RALF
*DTA1:TEMP.TM,LPT:<RALF.RA
```

Assembles DSK:RALF.RA into DTA1:TEMP.TM and writes a listing on the line printer.

RALF Assembler Error Messages

During assembly pass 2, error messages are printed at the console terminal as a 2-character error code followed by the label associated with the erroneous statement. If a listing was requested, error codes are printed during pass 3 on a line by themselves immediately preceding the statement to which they apply (except for EG, which follows the line in error). RALF error messages are listed in Table 5-2.

THE LOADER

The OS/8 FORTRAN IV loader accepts up to 128 RALF modules as input and links the modules, along with any necessary library components, to form a loader image file that may be loaded and executed by the run-time system. This is accomplished by replacing the relative starting location (00000) of each section with an absolute core address. Absolute addresses are also assigned to all entry points defined in the input modules. Once all RALF modules and library components have been assigned to some portion of memory and linked, absolute addresses are assigned to the relocatable binary text and the externs.

The overlay feature of the loader facilitates running programs which are too large to be contained in available memory, making it possible to run programs that require up to 300K words of storage in less than 32K of actual core memory. This is accomplished by dividing very large FORTRAN programs into a set of subroutines linked by one mainline. Unlike the subroutines, each of which has a section name by which it is called, the mainline does not have a name and is therefore assigned section name #MAIN by the system. An overlay scheme is then designed in such a way that the memory requirement of those subroutines that are core-resident at any given time does not exceed the available core memory.

An overlay is a set of subroutines stored on a bulk storage device. When any subroutine in an overlay is called by the mainline or another subroutine, the entire overlay is read into core, where it generally replaces another overlay of equivalent size.

Levels are variable-size portions of memory reserved for specific sets of overlays. OS/8 FORTRAN IV permits up to 8 levels, designated level 0, level 1, and so on up to level 7. Level 0 is always present and always contains only one overlay, called overlay MAIN, which always includes section #MAIN (the

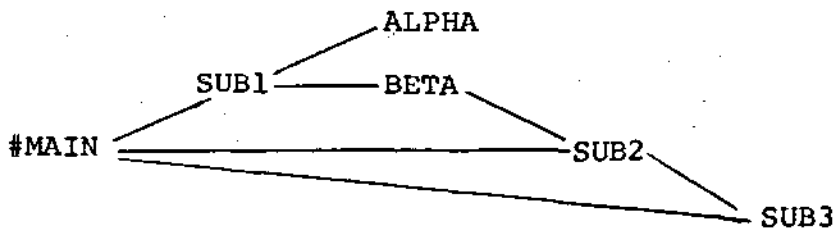
FORTRAN or RALF mainline) as well as all COMMON sections, 8-mode sections and library components. Additional subroutines may also reside in overlay MAIN; in fact, the entire program should be loaded into level 0 if there is sufficient core available.

Levels 1 to 7 may each contain up to 16 overlays, only one of which is core-resident at any given time during program execution. If no subroutines are loaded into a given level, that level does not exist for the current execution and no memory is allocated to it. As execution begins, overlay MAIN is loaded into level 0 (where it remains throughout execution) and started at the entry point of section #MAIN. Other overlays are read into the block of memory reserved for their particular level whenever one of their constituent subroutines is called. As an overlay is read into a given level, it overwrites any other overlay which may have been resident in that level. Thus, no two overlays from the same level are ever core-resident simultaneously.

When section #MAIN or any subroutine calls another subroutine, the flow of execution from calling routine to called routine is referred to as part of a calling sequence. Every calling sequence begins with a call from section #MAIN and ends with a call to some subroutine that does not contain any further CALL statements. Calling sequences generally contain branches, and they may be very intricate. For example, assume that:

Routine/Subroutine	Contains Calls To
mainline (#MAIN)	SUB1, SUB2, SUB3
SUB1	ALPHA, BETA
SUB2	SUB3
SUB3	
ALPHA	
BETA	SUB2

Then the calling sequences could be mapped as:



When any subroutine CALL is executed, the system determines whether the overlay containing the called routine is core-resident and, if not, reads this overlay into its proper level in core, overwriting any overlay which was previously resident in that level. No such determination is possible for RETURN statements, however. For this reason, it is extremely important to ensure that, at the end of a calling sequence, all subroutines in the calling sequence are still core-resident. In other words, no subroutine may execute a CALL that will cause it, or any subroutine which called it, to be overlaid. In the previous example, if SUB1, SUB2 and SUB3 occupy separate overlays in level 1 while ALPHA and BETA reside in level 2, the calling sequence from #MAIN to SUB1 to BETA to SUB2 will cause a fatal error because SUB2 will overwrite SUB1 and prevent control from returning to level 0. The FORTRAN system guards against some errors of this type by enforcing the following rules:

a. Subroutines in a given level cannot call other subroutines in the same level if the called subroutine is in a different overlay.
b. Subroutines in high numbered levels cannot call subroutines in lower numbered levels unless the call is to level 0. (This convention is not enforced when the U option is specified to the run-time system.)

These restrictions will not prevent fatal errors in all cases. In the previous example, if subroutine BETA is placed in level 0 instead of level 1, the calling sequence from #MAIN to SUB1 to BETA to SUB2 still causes a fatal error, even though neither of the enforced conventions is violated. Thus, any overlay scheme must be designed with careful attention to calling sequences.

If the L or G option is specified to F4 or RALF, the loader is called automatically to relocate the output of a successful assembly. When chained to via F4, the loader reacts in one of two ways. If the last Command Decoder file/option line was terminated with a carriage return, it immediately fetches the Command Decoder and proceeds as though it had been called from the monitor, as described below. The only difference, in this case, is that certain loader or run-time system options may have been passed to the loader from RALF, and cannot be suppressed at this point. Also, unless two different files are specified as output files, the loader automatically routes its loader image to the first output file specified

to F4 or RALF at the start of the chain, with default extension ".LD" assigned if this file had a null extension, or to file SYS:FORTRAN.LD if no output files were specified. The relocatable binary output produced by the assembler is deleted after it has been read as input. A loader symbol map is routed to the third output file specified at the start of the chain sequence, if any, or to the second output file, if any, specified to the loader as described below. When this is a directory device file with a null extension, the default extension ".LS" is supplied.

If the last file/option specification supplied to the Command Decoder was terminated with an ALTMODE character instead of a carriage return, the loader reacts differently when chained to from RALF. In this case, the loader assumes that the RALF module just produced is a stand-alone mainline that requires no subroutines (other than library components) in order to execute. The loader does not call the Command Decoder under these circumstances, since level 0 is the only level that will be defined. Output is produced exactly as described above, and the loader either returns to the Keyboard Monitor upon completion or, if a G option specification was previously entered, chains to the run-time system.

The loader may be called separately, to link and relocate a set of previously assembled RALF modules. This is accomplished by typing:

```
R LOAD
```

(terminated by a carriage return) in response to the dot generated by the Keyboard Monitor. The loader replies by calling the OS/8 Command Decoder, which accepts and decodes one or more standard command lines, each of which designates 0 to 9 input files, 0 to 2 output files, and any run-time option specifications desired. Each file/option specification line is entered by typing:

```
DEV:IMAGE.LD,DEV:MAP.LS<DEV:PROGA.RL,...,DEV:PROGX.RL(options)
```

(terminated by a carriage return) in response to the asterisk generated by the Command Decoder. IMAGE.LD is the loader image output file and MAP.LS is the loader symbol map output file. The input files may be either relocatable binary RALF modules or a library file, and "options" is a string of alphabetic characters that designates any run-time options desired.

The loader accepts up to 128 input file specifications, one of which may designate a library file to be used in place of the standard system library. The OS/8 Command Decoder, however, accepts a maximum of only 9 input file specifications per command line. Thus, after each file/option command line is entered, the loader recalls the Command Decoder to accept another command line, and continues this process until the /G option is received or a line is terminated with an ALTMODE. Input file specifications should be entered in sequence, beginning with all RALF files to be loaded into level 0, followed by files for level 1 overlay 1, level 1 overlay 2, and so on until all level 1 overlays are filled. Level 2 overlays are then built in the same manner, using as many file/option specification lines as necessary, and the process continues until all levels are filled. Each line may contain from 0 to 9 input file specifications; null lines will be ignored by the loader.

At some point during this process, two output files and one library (input) file may also be specified. The loader image file built by the loader is routed to the first output file, which must reside on a directory device, or to file SYS:FORTRN.LD if no output files are specified. When the first output file has a null extension, the default extension ".LD" is supplied. The loader symbol map is routed to the second output file, provided that a second file is specifically defined. If this is a directory device file with a null extension, the default extension ".LS" is supplied. One library file may be specified as an input file, to be used in place of the standard system library. This must be a specially formatted file, prepared with LIBRA as described in the section concerning the FORTRAN IV Library, and it must be specified on a command line that contains no other input file names. This command line may appear anywhere in the file/option specification sequence. It is identified by the presence of an L option specification.

If more than one first output file, second output file, or library file is specified to the loader, only the last specification in each category is used. Previous specifications, including those supplied to F4 or RALF when chaining to the loader, are ignored.

Run-time option specifications are used to group the sequence of input files into discrete overlays, allocate overlays to certain levels, and identify the user-generated library file, if any. Table 8-5 lists the run-time options recognized by the loader and describes their

use. The E and H options, recognized by the run-time system, may be entered on the same line as the G option when chaining to the run-time system.

Table 8-5 Loader Run-Time Options

Option	Operation
C	Continue the current line of input on the next line of input. When specifying RALF files to the loader, there may be more than 9 files that belong in a given overlay. Since the Command Decoder will not allow more than nine input files in one file option specification line, the C option permits the additional files to be put on the following line. If the C option is not specified at the end of a line, the current overlay is closed when the terminating carriage return is received and subsequent input files are placed in a new overlay in the current level. An exception to this is level 0, which only contains one overlay. The presence of a C option specification is assumed on every line until level 0 has been closed by an O specification.
G	Treat the current line as the last line of input, and chain to the FORTRAN IV run-time system when finished.
L	Accept the single input file specified on this line as an alternate library to be used in place of the system library, FORLIB.RL.
O	Close the level that is currently open, and open the next sequential level for input. RALF files specified on subsequent lines are assigned to overlays in the new level until the new level is closed by the next O specification (or the end of input).
S	Include system symbols in the loader symbol map. System symbols are identified by an initial "#" character. This option is only valid when a symbol map output file was specifically defined.
U	Ignore the rules governing subroutine calls between overlays. This option should only be used when subroutines making illegal calls will not be accessed during execution since, in general, any illegal subroutine call will cause unpredictable behavior at run time.

Input may be terminated by entering a G option specification on the last line and/or by terminating the last line with an ALTMODE character rather than a carriage return. If the G specification and the ALTMODE both appear, this indicates that the user has no file/option specification input for the run-time system and prevents the run-time system from calling the Command Decoder.

The following sequence of Command Decoder specification lines illustrates the use of option specifications to allocate RALF files to particular overlays.

.R LOAD	Loader is called from Keyboard Monitor.
*SYS:PROG.LD,LPT:<PROG.RL	Loader image file will be routed to SYS:PROG.LD while the symbol map is printed on the line printer. PROG.RL is placed in level 0 overlay MAIN. Since the presence of a C option specification is assumed on every line preceding the first O option specification, level 0 overlay MAIN remains open.
*<ALPHA.RL,BETA.RL	Place subroutines ALPHA and BETA in level 0 overlay MAIN. The presence of a C option specification is assumed.
*/O	Close level 0 and open level 1 overlay 1.
*<SUB1.RL,SUB2.RL,SUB3.RL	Place SUB1, SUB2 and SUB3 in level 1 overlay 1. Close overlay 1 and open overlay 2.
*<SUB4.RL,SUB5.RL,SUB6.RL/C	Place SUB4, SUB5 and SUB6 in level 1 overlay 2. Accept further input for this overlay on the next line.
*<DTA1:SUB7.RL/O	Place SUB7 in level 1 overlay 2. Close level 1 and open level 2 overlay 1.

`*<DTA1:SUB8.RL`	Place SUB8 in level 2 overlay 1. Close overlay 1 and open overlay 2.
`*<SUB9.RL`	Place SUB9 in level 2 overlay 2. Close overlay 2 and open overlay 3.
`*<LIB.RL(LS)`	Use file DSK:LIB.RL in place of SYS:FORLIB.RL as the library file. In spite of its position in the specification list, any library components will be placed in level 0. The S option specification requests an augmented loader symbol map.
`*<SUB10.RL/O`	Place SUB10 in level 2 overlay 3. Close level 2 and open level 3 overlay 1.
`*<SUB11.RL,DTA1:SUB12.RL/G`	Place SUB11 and SUB12 in level 3 overlay 1. Close level 3, terminate input, and chain to the run-time system when finished.

This sequence of commands will provide the following overlay scheme:

Level	Overlay	Contents
0	MAIN	PROG, ALPHA, BETA library subroutines
1	1	SUB1, SUB2, SUB3
1	2	SUB4, SUB5; SUB6, SUB7
2	1	SUB8
2	2	SUB9
2	3	SUB10
3	1	SUB11, SUB12

Note that all of the input files except those containing SUB7, SUB8, and SUB12 are taken from device DSK:, the OS/8 default device. The left angle bracket (or backarrow) character is optional when a file/option specification line contains only input file specifications, but it has been included here for clarity. Obviously, there are many other ways in which the sequence of file/option specifi-

cations shown above could have been entered to produce an identical result.

Considerable foresight is required when designing an overlay scheme. Since an overlay may have to be read into core whenever one of its constituent subroutines is called, a great deal of useless I/O results from inefficient overlay design. The system does verify that an overlay is not already resident before reading it into core.

Levels must be an integral number of system blocks (400 octal words) in size and big enough to accommodate the largest overlay they contain.

Ideally, then, the largest overlay in a level should occupy slightly less than some multiple of 400 (octal) words of storage, and all overlays in a level should be nearly equal in size. For example, if level 1 contains three overlays requiring 300, 100, and 150 octal words of storage, respectively, then the two smaller overlays should be combined because level 1 will be 400 octal words long in any case. If the three overlays require 500, 100, and 150 octal words of storage, all three should be combined because level 1 will be 1000 octal words long in any case.

Frequently called subroutines should be kept core-resident whenever possible, perhaps by placing them in level 0 or in a level that contains rarely accessed overlays. Within the loader image file, subroutines are stored in the order in which they were specified to the loader. Thus, grouping frequently called subroutines into adjacent levels also speeds execution by reducing the access time required to read an overlay into core, particularly from DECtape or LINCtape. When running very large programs with many overlay levels, it may be desirable to make level 0 as small as possible, in spite of the resulting excess I/O. This is accomplished by minimizing COMMON (which always occupies level 0), dividing the mainline into a series of subroutines, and creating a new mainline that contains predominately CALL statements. Note, however, that all library subroutines will reside in level 0, regardless of the location of subroutines that call them.

Any error recognized by the loader during generation of a loader image file results in an error message, printed on the console terminal, immediately following the input specification line that caused the error condition. Table 8-6 lists the loader error messages and describes the error condition indicated by each message.

The optional loader symbol map lists all symbols defined in the loader image file and identifies each symbol by overlay, level, and memory address:

```
LOADER V21  04 /30 /73

SYMBOL VALUE LVL OVLY

A      10400  1  00
ARGERR 00204  0  00
B      10400  1  01
C      11214  1  01
EXIT   00223  0  00
#MAIN  10000  0  00
 12000  = 1ST FREE LOCATION


LVL OVLY LENGTH

 0  00   10143
 1  00   01270
 1  01   01240
```

Following the alphabetical list of symbols, the loader prints the address of the first free memory location and the length, in octal words, of each overlay defined. This information is useful in optimizing memory requirements.

Loader Error Messages

The loader prints error messages on the console terminal during generation of a loader image file. Except where indicated in Table 8-6, loader errors are fatal. The loader returns control to the Keyboard Monitor when a fatal error condition is encountered.

Table 8-6 Loader Error Messages

Error Message	Meaning
BAD INPUT FILE	An input file was not a RALF module.
BAD OUTPUT DEVICE	The loader image file device was not a directory device, or the symbol map file device was a read-only device. The entire line is ignored.
ILLEGAL ORIGIN	A RALF routine tried to store data outside the bounds of its overlay.

Table 8-6 Loader Error Messages (Cont.)

Error Message	Meaning
MIXED INPUT	The L option was specified on a line that contained some file other than a library file. The library file (if any) is accepted. Any other input file specification is ignored.
MULT SECT	Any combination of entry point, COMMON section (with the exception of multiple COMMONs) or program section of the same name causes this error, except as shown in the following table.

	ENTRY POINT	SECT	SECT8	COMMON	COMMZ	FIELD1
SECT	MS	MS	MS	OK	OK	OK
SECT8	MS	MS	MS	OK	OK	OK
COMMON	MS	MS	MS	OK	MS	OK
COMMZ	MS	MS	MS	MS	OK	MS
FIELD1	MS	MS	MS	OK	MS	OK

NO MAIN	No RALF module contained section #MAIN.
OVER CORE	The loader image requires more than 32K of core memory.
OVER IMAG	Output file overflow in the loader image file.
OVER SYMB	Symbol table overflow. More than 253 (decimal) symbols in one FORTRAN job.
TOO MANY LEVELS	The 0 option was specified more than 7 times.
TOO MANY OVERLAYS	More than 16 overlays were defined in the current level.
TOO MANY RALF FILES	More than 128 input files were specified.

The following FATAL error messages occur when the Loader is linking and relocating:

```
SYSTEM ERROR

LOADER I/O ERROR

OS/8 ENTER ERROR
```

and indicate an error detected by OS/8 while trying to perform a USR function.

All errors identified during the loading procedure are followed by a line of the form:

1 oo nnn

where "1" is the level in which the error occurred, "oo" is the overlay in which the error occurred, and "nnn" is the module number, within the referenced overlay, that caused the error. Some errors (e.g., NO MAIN) are attributable to a single module, and the module numbers for this type of error are meaningless.

FORTRAN IV RUN-TIME SYSTEM (FRTS)

The OS/8 FORTRAN IV run-time system reads, loads, and executes a loader image file produced by the loader. It also configures a software I/O interface between the FORTRAN IV program and the OS/8 operating system, then monitors program execution to direct I/O processes and identify certain types of run-time errors. The run-time system is called automatically to load and execute the loader image file produced by the loader whenever the G option is specified to the loader.

When chained to from F4, RALF, or LOAD, the run-time system reacts in one of two ways. If the last Command Decoder file/option line was terminated with a carriage return, it immediately fetches the Command Decoder and proceeds as though it had been called from the Keyboard Monitor, as described below. The only difference, in this case, is that certain run-time system options may have been passed to the run-time system from the loader, and cannot be suppressed at this point. If the last file/option specification line supplied to the Command Decoder was terminated with an ALTMODE character instead of a carriage return, however, the loader assumes that no user input is required. The Command Decoder is not called. The loader image file just produced is read

as input, and, unless the H option was previously specified, it is loaded and executed.

The FORTRAN IV Run-Time System is able to accept file I/O specifications. This allows the user to write a source program which refers to I/O devices as integer constants or variables. This program may be compiled, assembled, and loaded into an image file. This image file may be run any number of times, each time specifying different physical I/O devices. Thus logical unit 8 may refer in one run to the console terminal, in another run to a disk file, and in another run to a paper tape punch.

These run-time specifications allow the FORTRAN program to use the OS/8 file handling capabilities, to use any OS/8 supported I/O device, and potentially to use any I/O device for which an OS/8 device handler can be written.

The following pages explain how the user gives the run-time system the connections between OS/8 device and file names and the FORTRAN logical unit numbers.

FORTRAN IV programs are usually saved as loader image files and executed by calling the run-time system from the Keyboard Monitor to load and execute the saved loader image. This is accomplished by typing:

```
R FRTS
```

(terminated by a carriage return) in response to the dot generated by the Keyboard Monitor. The run-time system replies by calling the OS/8 Command Decoder to accept one or more standard file/option specification lines. It recalls the Command Decoder after processing each line, until a line terminated by an ALTMODE character is received.

The run-time system accepts two classes of Command Decoder file/option specifications. The first class specifies the load module to be executed; the second class specifies the run-time file assignment. When it is called from the Keyboard Monitor, the run-time system loads the Command Decoder to accept one input file name, perhaps followed by the E or H option specifications, described in Table 8-7. This information is not required when the loader chains to the run-time system because the loader image file just produced is automatically read as input, while the E and/or H options could

have been specified to the loader along with the G specification that requested chaining.

Thus, the loader image input file to be executed must be identified on the first file/option specification line when FRTS is called from the Monitor, and must not be specified at all when the loader chains to FRTS. This Command Decoder line has the form:

*DEV:IMAGE.LD(options)

where IMAGE.LD is the loader image input file and "options" is E or H or both. If this line is terminated by an ALTMODE, the program is executed; if it is terminated with a carriage return, the Command Decoder is recalled to accept run-time file specifications.

Once the loader image file to be executed has been identified, the run-time system recalls the Command Decoder to accept any FORTRAN I/O device specifications. Of the nine I/O unit numbers available under FORTRAN IV, four are initially assigned to FORTRAN internal device handlers by the system as follows:

I/O Unit	Internal Handler	Comments
1	paper tape reader	Single character buffer.
2	paper tape punch	Single character buffer.
3	line printer	LP8 and LS8E only. Ring buffered.
4	console terminal	Double buffered output, single character input.

The FORTRAN internal handlers listed above are not the same as the OS/8 device handlers. The FORTRAN internal handlers are designed for ASCII text only and will not execute binary or core image I/O. Also, FORTRAN internal handlers are interrupt driven to execute foreground I/O concurrently with background computation.

FORTRAN internal device handlers may be assigned different unit numbers, in addition to those listed above, by typing:

/n=m

(in response to the asterisk generated by the Command Decoder) where m is the I/O unit number (1 to 4) of one of the internal handlers listed above and n is a different unit number (1 to 9) which is also to be assigned to that internal handler. This specification

causes all program references to logical unit n to perform I/O to device m in the preceding table. For example:

/6=2 Assigns the FORTRAN internal paper tape punch handler as I/O unit number 6, in addition to unit number 2.

/1=2 Assigns I/O unit number 1 to the FORTRAN internal paper tape punch handler instead of the internal paper tape reader handler.

OS/8 device handlers for non-directory devices may be assigned I/O unit numbers by typing:

DEV:/n

(in response to the asterisk generated by the Command Decoder) where n is an I/O unit number (1 to 9) and DEV: is the standard or assigned designation for any supported non-directory device. For example:

LPT:/3 Specifies the OS/8 line printer handler to be used instead of the FORTRAN internal line printer handler, possibly because the line printer is not an LP08 or LS8E.

Existing directory device files may be assigned I/O unit numbers by typing:

DEV:FILE.EX/n

(in response to the asterisk generated by the Command Decoder) where n is an I/O unit number (1 to 9) and DEV:FILE.EX is the standard OS/8 designation for an existing directory device file. For example:

DTA1:FORIO.TM/2 Assigns unit number 2 to DECtape file FORIO.TM rather than to the FORTRAN internal paper tape punch handler, where FORIO.TM is an existing file on DECtape unit 1.

A directory device file that does not presently exist may be assigned a FORTRAN I/O unit number in the same manner by entering it as an output file on the specification line; however, only

one such file may be created on any particular device. For example:

FORIO.TM</9	Assigns unit number 9 to file DSK:FORIO.TM, which has not been created at load time.

In any case, only one device or file specification is permitted on each line, and no more than 6 directory device files may be created by the FORTRAN program. Excess files after the sixth are accepted and written, but they will not be closed. If a file created by the program has the same file name and extension as a pre-existing file, the old file is automatically deleted when the new file is closed.

The Command Decoder "[n]" specification may be used to optimize storage allocation when assigning files that do not yet exist, where n is a decimal number that indicates the maximum expected length of the file, in blocks.

Each time a run-time I/O specification is terminated with a carriage return, the Command Decoder is recalled to accept another specification. When a specification is terminated with an ALT-MODE, the program is run.

Although existing files are specified as though they were input files and nonexistent files are specified as though they were output files, any file that has been assigned a unit number may be used for either input or output. The content of a nonexistent file is undefined until it has been written by the program.

Table 8-7 Run-Time System Option Specifications

Option	Operation
H	Halt after loading but before starting the program. Press the CONTinue switch on the processor to commence execution.
E	Ignore the following run-time system errors, any of which indicates that an error was detected earlier in the compilation/assembly/loading process: a. Illegal subroutine call. b. Reference to an extern in an overlay other than in the form "JSR EXTERN" (i.e., CALL statement). c. Reference to an undefined symbol.

Table 8-7 Run-Time System Option Specifications (Cont.)

Option	Operation
	Any of the above may lead to unpredictable program behavior as, in general, some portion of the program will not be loaded or executed.
C	Carriage control switch. The first character on every output line is processed as a carriage control character (see FORTRAN IV LANGUAGE SUMMARY by all FORTRAN internal handlers and also by the OS/8 hard copy handlers TTY and LPT. The first character on every output line is processed as data, in the same manner as any other character, by all OS/8 handlers except TTY and LPT. Entering a C option specification on the command line that assigns an I/O unit number to a particular handler reverses the processing of carriage control characters for that device. Thus: TEMP(2C) assigns file DSK:TEMP. as I/O unit 2. The C option causes the first character of every output line to be processed as a carriage control character. If C were not specified, these characters would be processed as data. /C/6=3 assigns the FORTRAN internal line printer handler as I/O unit 6, as well as unit 3. The first character of every line will be processed as a carriage control character on unit 3, and as a character of data on unit 6.

The OS/8 FORTRAN IV run-time system executes with the PDP-8/E interrupt system enabled. OS/8 device handlers are not interrupt driven; however, certain handlers may execute with the interrupt system enabled because the devices they control have interrupt enable switches which the handlers do not set. FRTS allows for this by running with the interrupt system enabled when driving handlers of this type, and disabling the interrupt system when a handler that does not run under interrupts is loaded. Handlers that can run with the interrupt system enabled include:

TC08 DECtape system handler and nonsystem handlers DTA0 to DTA7

RF08 system handler

RK8 system handler and nonsystem handlers RKA0 to RKA3
RK8E system handler and nonsystem handlers RKA0 to RKA3 and RKB0 to RKB3
Any FORTRAN internal handlers

These OS/8 handlers do not permit interrupts from these devices, but they do permit other devices, e.g., CLOCK, to interrupt the data transfer. Note that TD8E is absent from this list because the TD8E data transfer cannot be interrupted.

The run-time system recognizes two classes of error conditions. Certain errors are diagnosed while the core image file is being read from a storage device and loaded into core memory. Other errors may occur during execution of the FORTRAN program. Both classes of run-time errors are identified on the console terminal. Table 8-8 lists the FRTS error messages and describes the error condition indicated by each message. The run-time system error traceback feature provides automatic printout of statement numbers corresponding to the sequence of executable statements that terminated in an error condition. At least one statement number is always printed. This number identifies the erroneous statement or, in certain cases, the last correct statement executed prior to the error. When a statement was compiled under the N option, however, the system cannot generate meaningful statement numbers during traceback.

The console terminal serves as FORTRAN I/O unit 4 for both input and output. Terminal input is automatically echoed on the console printer. In addition, the run-time system monitors the keyboard continually during execution of a FORTRAN program. Typing CTRL/C at any time causes an immediate return to the OS/8 Monitor. Typing CTRL/B branches to the system traceback routine, and then exits to the monitor. This traceback routine causes a printout, similar to the error traceback, including the current subroutine, the line number in the next higher level subroutine from which it was called, etc., to the main program. This facilitates locating infinite loops when debugging a program. The following additional special characters are recognized by the console terminal handler and processed as shown:

RUBOUT	Deletes last character accepted.
CTRL/U	Deletes current line of input.

CTRL/I (Tabulation.) Converted to appropriate number of spaces.

CTRL/Z Signals end-of-file on input.

Tentative output files (that is, files created by the FORTRAN program) are closed automatically upon successful completion of program execution provided that either:

1. An END FILE statement referencing the file was executed. FRTS assigns a file length equal to the actual length of the file.
2. The last operation performed on the file was a write operation. FRTS proceeds as though an END FILE statement had been executed.
3. A DEFINE FILE statement referencing the file was executed but an END FILE statement was not executed. Upon completion of program execution, FRTS assigns a file length equal to the length specified in the DEFINE FILE statement.

Execution of a REWIND statement does not close a tentative file, nor does it modify the tentative file length.

Run-Time System Error Messages

The run-time system generates two classes of error messages. Messages listed in Table 8-8 identify errors that may occur during execution of a FORTRAN program and errors that may be encountered when the run-time system is reading a loader image file into memory in preparation for execution, or accepting I/O unit specifications. Except where indicated, all run-time system errors cause full traceback and an immediate return to the monitor. Nonfatal errors cause partial traceback, sufficient to locate the error, and execution continues.

Table 8-8 Run-Time System Error Messages

Error Message	Meaning
BAD ARG	Illegal argument to library function.
CAN'T READ IT!	I/O error on reading loader image file.
CAUTION—NO DP	The present hardware configuration does not include an FPP-12 Floating-Point Processor with double precision option. Execution continues; however,

Table 8-8 Run-Time System Error Messages (Cont.)

Error Message	Meaning
	all double precision operations default to real arithmetic (with unpredictable results) and all complex operations also produce unpredictable results.
D.F. TOO BIG	Product of number of records times number of blocks per record exceeds number of blocks in file. Note that for a random access file the length in OS/8 blocks must be no less than the number of records times the integer but must be greater than the quotient of floating point variables per record divided by 85.
DIVIDE BY 0	Attempt to divide by zero. The resulting quotient is set to zero and execution continues.
EOF ERROR	End of file encountered on input.
FILE ERROR	Any of: a. A file specified as an existing file was not found. b. A file specified as a nonexistent file would not fit on the designated device. c. More than 1 nonexistent file was specified on a single device. d. File specification contained "*" as name or extension.
FILE OVERFLOW	Attempt to write outside file boundaries.
FORMAT ERROR	Illegal syntax in FORMAT statement.
FPP ERROR	Hardware error on FPP start-up.
INPUT ERROR	Illegal character received as input.
I/O ERROR	Error reading or writing a file, tried to read from an output device, or tried to write on an input device.
MORE CORE REQUIRED	The space required for the program, the I/O device handlers, I/O buffers and the resident Monitor exceeds the available core.

Table 8-8 Run-Time System Error Messages (Cont.)

Error Message	Meaning
NO DEFINE FILE	Direct access I/O attempted without a DEFINE FILE statement.
NO NUMERIC SWITCH	The referenced FORTRAN I/O unit was not specified to the run-time system.
NOT A LOADER IMAGE	The first input file specified to the run-time system was not a loader image file.
OVERFLOW	Result of a computation exceeds upper bound for that class of variable. The result is set equal to zero and execution continues. This error is detected only if an FPP is present.
OVERLAY ERROR	Error while reading overlay.
PARENS TOO DEEP	Parentheses nested too deeply in FORMAT statement.
SYSTEM DEVICE ERROR	I/O failure on the system device.
TOO MANY HANDLERS	Too many I/O device handlers are resident in memory, or files have been defined on too many devices.
USER ERROR	Illegal subroutine call, or call to undefined subroutine. Execution continues only if the E option was requested.
UNIT ERROR	I/O unit not assigned, or incapable of executing the requested operation.

FORTRAN IV LIBRARY

The OS/8 FORTRAN IV system contains a general purpose FORTRAN library FORLIB.RL, which may be extended and modified by the librarian LIBRA. The library gives the programmer the capacity to compute arithmetic and transcendental functions, use the complex and double precision options of the FPP, read console switches, and interface with standard laboratory peripherals.

The OS/8 FORTRAN librarian, LIBRA, is used to create and maintain libraries of RALF modules. The loader uses one such library, specified by the user, to resolve undefined external symbols. Each library contains a collection of RALF modules and a catalog, which lists the program section names and entry points defined in the modules, along with sufficient information for the loader to find them.

LIBRA's tasks are to create libraries (and their catalogs) from user-specified sets of modules (RALF output files), to add new modules to existing libraries, to copy the contents of a library to a new library (all with various options on selective deletion and replacement during the copy), and to list the catalogs of libraries.

To create a library, call LIBRA by typing:

```
R LIBRA
```

in response to the dot generated by the Keyboard Monitor. LIBRA loads the OS/8 Command Decoder, which prints an asterisk at the left margin. In response to the Command Decoder's asterisk, type in the following order:

1. The output device and name of the library to be created (LIBRA assigns the extension .RL unless one is specified). If no output file is specified, the default name FORLIB.RL is used and output is to the system device.
2. The desired number of index blocks (decimal, maximum 255) enclosed in square brackets. LIBRA allocates two index blocks if no specification is given.
3. The output device for the catalog listing when the library build is complete (preceded by a comma). If no device is specified, the listing is suppressed.
4. The input files (RALF output modules) or libraries to be included in the library (preceded by a backarrow or left angle bracket).
5. Options:

   /C to continue input specification on next line.

   /I if a decision is to be made on insertion of each entry point or section name.

   /Z if an existing file of the same name is to be replaced by the new library.

/R if a new input file is to replace a module of the same name already in the library.

The = option if extra blocks are to be allowed for library expansion.

```
.R LIBRA
*LIB1.RL[5],TTY:<LIB0.RL,R1,R2,R3,R4,R5,R6/Z=20
```

For the above command, a library named LIB1.RL is created on the system device containing the existing library, LIB0.RL, and the files R1, R2,..., R6. Five blocks are allocated for the index; the catalog is printed on the console terminal and 20 (octal) extra blocks are reserved for future expansion. The /Z indicates that if a file already exists with the name LIB1.RL, it is to be replaced by the newly created library.

If there are more than nine modules to be included, type /C to continue input specification on the next line. Note that the "=" option and the output device for the catalog listing must be specified on the last line (that is, the one without /C). The /Z, if it is used, must appear on the first line.

```
.R LIBRA
*LIB1.RL/Z[5]<R1,R2,R3,R4,R5,R6,R7,R8,R9,R10/C
*,TTY:<R10,R11=20
```

In this case the library now contains the additional files R7, R8, ..., R11. The /I and /R options can be specified at any point in the command line. Both /I and /R apply only to modules listed on the line in which they appear.

To expand a previously created library, call LIBRA as usual. Specify the name of the old library file as the first output file, the catalog listing file, if desired, next, and then the modules or libraries to be added as input. Do not specify /Z.

```
.R LIBRA
*LIB1.RL,TTY:<ROUT,MOD
```

LIBRA adds the contents of ROUT and MOD to LIB1. If the old library file name does not exist, a new library is created using default options if necessary. Since LIBRA cannot change the size of the index or the room left for expansion at this time, it is useless to specify index blocks and expansion blocks.

If a module entry point or section name being added to a library duplicates a name in the library catalog, the duplicate name is printed on the terminal. The name in the catalog continues to refer to the original module, unless:

/R is specified on a command line, then the new module is put in the library and the old module of the same name is deleted (unless there are other names for the old module, in which case only the duplicate name is deleted). For example:

```
*LIB1.RL<LIB0.RL,R1,R2,R3/R
```

causes any of the input modules R1, R2, and R3 to replace existing modules in LIB0.RL with the same entry point or section name.

/I is specified on the LIBRA command line, then input file entry points and section names are listed on the console terminal. If the names duplicate names in the catalog, the message printed is:

xxxx IS DUPLICATE NAME; KEEP OLD OR NEW?

where xxxx is an entry point or section name. LIBRA waits for the user to type OLD and a RETURN (or just a RETURN or O and a RETURN) to keep the old name; NEW and a RETURN (or N and a RETURN) to delete the use of the old name and include the new. The question is repeated if any other character is typed.

If the new names do not appear in the catalog, the message typed is:

xxxx: INCLUDE?

where xxxx is the new entry point or section name.

Type YES and a RETURN (or just a RETURN or Y and a RETURN) to include the name or NO and a RETURN (or N and a RETURN) to omit it. The question is repeated if any other character is typed.

A catalog listing can be obtained at any time by omitting the input file specification in the call to LIBRA. For example:

```
.R LIBRA
*FORLIB.RL,LPT:<
```

prints the catalog of FORLIB on the line printer. LIBRA's version number (Vxx) is output as part of the catalog heading.

Entry points and section names may be deleted from the catalog by combining the I and Z options. Each catalog entry is listed on the console terminal with the message:

name: INCLUDE?

Type Y and RETURN to include the section name or entry point; type N and RETURN to delete it. If all catalog entries corresponding to a particular module are deleted from the catalog in this manner, the module is deleted from the library and the message:

MODULE IS DELETED

is printed on the console terminal.

FORLIB.RL, the standard library supplied with the FORTRAN IV system, contains functions and subroutines that perform mathematical calculations and drive various peripheral devices. This library may be modified with LIBRA to fit the needs of a particular installation. Although at least one copy of the standard library should be maintained as a backup, it may be desirable to delete unwanted routines from FORLIB in order to reduce storage requirements. For example, double precision routines may be deleted if an installation does not include an FPP-12 with extended precision option. Care should be exercised not to delete subroutines that may be called by the various system programs or by other library routines that are not deleted. Table 8-9 lists the library routines that execute calls to entry points in other routines; in general, when an entry in the right column of Table 8-9 is deleted, the corresponding entry in the left column may not be called.

Table 8-9 FORLIB Calling Relationships

Section Name:	Executes Calls to:
SYNC	DISP, ONQI
DISP	ONQB
EXPIR	EXP3
EXP3	ALOG, EXP
ALOG10	ALOG
COS	SIN
TAN	SIN, COS
SIND	SIN

Table 8-9 FORLIB Calling Relationships (Cont.)

Section Name:	Executes Calls to:
COSD	SIN
TAND	TAN
ASIN	ATAN, SQRT
ACOS	ATAN, SQRT
ATAN2	ATAN
SINH	EXP
COSH	EXP
TANH	SINH, COSH

For example, to delete the entry points ABS, IABS, and LSW from the catalog, the proper command to LIBRA is:

```
.R LIBRA
*LIB2,RL<LIB1,RL/I/Z
```

Respond with Y and a carriage return to all of the messages except:

```
IABS: INCLUDE? N
ABS: INCLUDE? N
MODULE IS DELETED
.
.
LSW: INCLUDE? N
.
.
```

The module containing ABS and IABS is deleted from the library because all of its section names and entry points have been deleted from the catalog. Entry point LSW is deleted from the catalog, but the corresponding module remains in the library because other entry points are still present in the catalog. Table 8-10 lists the FORLIB entry points that are contained in modules with different section names.

Table 8-10 FORLIB Multiple Entry Points by Section

Section Name	Other Entry Points
IABS	ABS
SIGN	ISIGN
AMIN0	AMIN1, MIN0, MIN1
AMAX0	AMAX1, MAX0, MAX1
DIM	IDIM
PLOT	SCALE, CLRPLT, #DISP
REALTM	SAMPLE, ADB
CHARS	CGET, CPUT, CHAR
IFIX	AINT, INT
AMOD	MOD
RSW	LSW, SSW, ROPEN, EXTLVL, RCLOSE
ONQI	ONQB
SYNC	CLOCK, TIME, #CLINT

The catalog entries #FIX, #RFDV, #LTR, #EQ, #NE, #GE, #LE, #GT, #LT, #EXPIR, #CLINT, and #EXPII are used by the compiler and should not be deleted.

Library Functions and Subroutines

Library functions and subroutines are called in the same manner as user written functions and subroutines. The following section lists the library components that are available to FORTRAN programs and illustrates calling sequences, where necessary. Arguments must be of the correct number and type, but need not have the same name as those shown in the illustrative examples. Routines that require LAB8/E or PDP-12 hardware are marked with an asterisk*. Routines that will run on the FPP with extended precision option are marked with a dagger sign (†). Neither symbol may be used in the actual FORTRAN program. Certain library routines are used by the FORTRAN system programs and are not available to a user's FORTRAN program. These routines may be identified by the initial "#" character in the entry point or section name, and are not listed in the following section.

ABS (SINGLE-PRECISION ABSOLUTE VALUE)

ABS calculates the absolute value of a real variable by leaving the variable unchanged if it is positive (or zero) and negating the variable if it is negative.

ACOS (SINGLE-PRECISION ARC-COSINE FUNCTION)

ACOS calculates and returns the primary arc-cosine (in radians) of a real argument less than or equal to 1.0 according to the relation:

$$\text{If } x > 0.0,\ \text{ACOS}(x) = \text{ATAN}\left[\frac{\text{SQRT}(1-X^2)}{x}\right]$$

$$\text{If } x < 0.0,\ \text{ACOS}(x) = \pi + \text{ATAN}\left[\frac{\text{SQRT}(1-X^2)}{x}\right]$$

$$\text{If } x = 0.0,\ \text{ACOS}(x) = \pi/2.0$$

ADB* (RETURN NEXT SAMPLE FROM REAL-TIME SAMPLING BUFFER)

ADB gets and returns the next sample in the range [−512, 511] from the real-time sampling buffer. The following program illustrates how ADB may be used to sample 500 points from channel 3 and plot them on the scope:

```
        DIMENSION PLTBUF(400),DATBUF(50)
1       CALL CLRPLT(400,PLTBUF)
        CALL REALTM (DATBUF,50,3,1,500)
        CALL CLOCK (8,10)
        DO 100 I=1,500
100     CALL PLOT(1,I/384.,ADB(X)/1024.+.5)
        READ(1,10)Q
10      FORMAT(I2)
        GO TO 1
        STOP
        END
```

After finishing the plotting, the program waits for the user to type the RETURN key, and then repeats the sampling-display process. Note that REALTM sets up the sampling procedure, while CLOCK actually initiates the sampling.

ADC* (ASYNCHRONOUS SAMPLING)

The ADC function accepts an integer argument in the range [0, 15], assumed to be a channel number. It returns the current value of the referenced channel as a real number in the range [−1, 1]. Sampling employs the fast SAM instruction for one or multiple channels. ADC may not be used in a program that also uses REALTM. The following program illustrates the use of the ADC function.

```
C    EXAMPLE OF ADC FUNCTION
C    REQUIRES PDP12 OR LAB8E HARDWARE
C    SAMPLES AND TYPES ANALOG INPUT
C
   10 CONTINUE
      WRITE(4,100)
  100 FORMAT('   TYPE IN CHANNEL NUMBER '
     1   'AND NUMBER OF SAMPLES')
      READ(4,101) NC,NS
  101 FORMAT(2I3)
      DO 20 I=1,NS
      X=ADC(NC)
      WRITE(4,102) X
  102 FORMAT(F15.5)
   20 CONTINUE
      GOTO 10
      CALL EXIT
      END
```

AIMAG† (COMPLEX TO IMAGINARY CONVERSION)

AIMAG returns the imaginary part of its complex argument as a real variable.

AINT (SINGLE-PRECISION FLOATING-POINT TO INTEGER)

AINT is a floating-point truncation function. Given a real argument, it truncates the fractional part of the argument and returns the integral part as an integer. This is accomplished by taking the absolute value of the argument, aligning and normalizing this result, then restoring the original sign. AINT, IFIX, and INT perform identical functions.

ALOG (SINGLE-PRECISION NATURAL LOGARITHM)

ALOG calculates and returns the natural (Naperian) logarithm of a real argument greater than zero. Any negative or zero argument returns an error message and a value of 0.0. The algorithm used is an 8-term Taylor series approximation.

ALOG10 (SINGLE-PRECISION COMMON LOGARITHM)

ALOG10 calculates and returns the common (base 10) logarithm of a real argument greater than zero. Any negative or zero argument returns an error message and a value of 0.0. The calculation is accomplished by calling ALOG to compute the natural logarithm and executing a change of base.

AMAX0 (SINGLE-PRECISION MAXIMUM VALUE)

AMAX0 accepts an arbitrary number of integer arguments and returns a real value equal to the largest of the arguments.

AMAX1 (SINGLE-PRECISION MAXIMUM VALUE)

AMAX1 accepts an arbitrary number of real arguments and returns a real value equal to the largest of the arguments.

AMIN0 (SINGLE-PRECISION MINIMUM VALUE)

AMIN0 accepts an arbitrary number of integer arguments and returns a real value equal to the smallest of the arguments.

AMIN1 (SINGLE-PRECISION MINIMUM VALUE)

AMIN1 accepts an arbitrary number of real arguments and returns a real value equal to the smallest of the arguments.

AMOD (SINGLE-PRECISION A MODULO B)

AMOD accepts two real arguments and returns a real value equal to the remainder when the first argument is divided by the second argument. If the second argument is not sufficiently large to prevent overflow, an error message and a value of 0.0 are returned.

ASIN (SINGLE-PRECISION ARC-SINE)

ASIN calculates and returns the arc-sine (in radians) of a real argument in the range [−1, 1] according to the relation:

$$\text{ASIN(X)} = \text{ATAN(X/SQRT(1-X**2))}$$

If the argument falls outside the range [−1, 1], an error message results.

ATAN (SINGLE-PRECISION ARC-TANGENT)

ATAN calculates and returns the primary arc-tangent (in radians) of a real argument. The argument is first reduced according to the relations:

(1) If $x<2^{-14}$, $\quad \text{atan}(x) = x$
(2) If $x>2^{-14}$, $\quad \text{atan}(x) = 1/x$
(3) If $x>1.0$, $\quad \text{atan}(x) = \pi/2 - \text{atan}(1/x)$
(4) If $x<0$, $\quad \text{atan}(x) = -\text{atan}(-x)$

and the arc-tangent is then computed by a power series approximation.

ATAN2 (SINGLE-PRECISION ARC-TANGENT OF TWO ARGUMENTS)

ATAN2 accepts two real arguments, assumed to be an abscissa and an ordinate respectively, and calculates the arc-tangent of the quotient of the first argument divided by the second argument. This is accomplished by calling ATAN to find the principal arc-tangent of the quotient and then adjusting the result, depending upon the quadrant in which a point defined by the arguments falls, according to the relations:

argument in first quadrant	atan2(y,x) = atan(y/x)
argument in second quadrant	atan2(y,x) = atan(y/x)−π
argument in third quadrant	atan2(y,x) = atan(y/x)−π
argument in fourth quadrant	atan2(y,x) = atan(y/x)+π

CABS† (COMPLEX ABSOLUTE VALUE)

CABS accepts a complex argument and returns the absolute value of the argument as a real variable defined by:

CABS(X+iY) = SQRT(X**2+Y**2)

CCOS† (COMPLEX COSINE)

CCOS accepts a complex argument and returns the cosine of the argument, a complex number defined by:

CCOS(X+iY) = COS(X)*COSH(Y)−i*SIN(X)*SINH(Y)

CEXP† (COMPLEX EXPONENTIAL)

CEXP accepts a complex argument and returns the exponential function of the argument, a complex variable defined by:

CEXP(X+iY) = EXP(X)*(COS(Y)+i*SIN(Y))

CGET (CHARACTER GET SUBROUTINE)

The calling sequence:

CALL CGET (STRING,N,CHAR)

causes the Nth character to be unpacked from STRING and stored in CHAR as a variable in the range 0, 63, where STRING is a character string in A6 format.

CHKEOF (CHECK FOR END-OF-FILE SUBROUTINE)

CHKEOF accepts one real, integer or logical argument. After the next formatted read operation, this argument will be set to non-zero if the logical end-of-file was encountered, or to 0 if the

logical end-of-file was not encountered. The following is an example of the use of CHKEOF:

```
.
.
.
CALL CHKEOF(EOF)
READ (N,101)DATA
IF (EOF.NE.0) GO TO 999
.
.
.
```

CLOCK* (INITIALIZE CLOCK SUBROUTINE)

The purpose of the CLOCK subroutine is to initialize certain features of the KW12A or DK8ES real-time clock. The calling sequence is:

```
CALL CLOCK (FUNCTN,RATE)
```

Depending upon the arguments FUNCTN and RATE, CLOCK can enable Schmitt triggers, clock controlled A/D conversions, or run the clock at a variable rate. The clock is always run on interrupt. Both arguments may be either integer, real, or logical in type. The first argument indicates a class of clock functions, and the second specifies a clock rate in Hertz. A common use of the clock routine will be in conjunction with the REALTM subroutine. With one exception noted below, the clock routine is independent of hardware type. That is, a program employing the KW12A clock on a PDP-12 does not require modification to run on a PDP-8. The FUNCTN argument controls the enabling of all Schmitt triggers, clock controlled A/D conversions, and clock rate or external input according to the scheme shown in Table 8-11.

Combinations of the conditions in Table 8-11 may be enabled by setting FUNCTN to a value equal to the sum of the values of the desired conditions. For example, to enable all Schmitt triggers, set FUNCTN=7 (the sum of 4, 2, and 1); to enable clocked A/D conversion at an external rate, set FUNCTN=24, etc. If a clock condition is not specified, the clock is disabled. Every call to CLOCK clears any functions which may have been enabled by previous calls to CLOCK and redefines clock conditions according to the new arguments. If the FUNCTN argument is out of range

(e.g., negative), the clock conditions enabled are arbitrary. Once the clock has been started, the calling program may disposition a Schmitt trigger via the SYNC subroutine, read the time of day via TIME, or use the clock in conjunction with some other routine such as REALTM. Schmitt triggers 1, 2 and 3 correspond to the DK8ES events 1, 2 and 4, respectively, or the KW12A clock channels 1, 2 and 3, respectively.

Table 8-11 CLOCK Subroutine FUNCTN Arguments

Value of FUNCTN	Effect
0	none, or enable clocked A/D conversion, more than one channel
1	enable Schmitt trigger 1
2	enable Schmitt trigger 2
4	enable Schmitt trigger 3
8	enable clocked A/D conversion, one channel
16	enable the clock to run under external input

The rate at which the clock runs is specified by the argument RATE, which has two different meanings that are dependent upon the FUNCTN argument. If the caller does not specify an external rate (e.g., FUNCTN less than 16), then RATE is interpreted as a number in Hertz and specifies the rate of clock interrupts.

When the clock is run at a programmable rate, the rate must fall in the range [0.0244, 4096.0], or one tick every 250 microseconds to every 40 seconds. Specifying a rate outside this range causes the clock to be disabled (which may be desirable in some cases). The calling program should not specify an unnecessarily high clock rate, as this slows down program execution. Because the allowable set of programmable clock rates is discrete, the clock may not run at exactly the specified rate but will always be less than or equal to the specified rate and within one percent of it. The actual rate can be computed from the specified rate by:

$R_A = {}^{RB}/_{[RB/RR]}$
R_A = actual rate
R_R = requested rate
[ ] = greatest integer

R_B = base rate — maximum number in the set (100000, 10000, 1000, 100) that satisfies the condition. $R_B/R_R \leqslant 4096$

If an externally driven clock is specified, RATE is interpreted as the number of external ticks between clock interrupts, and must be in the range [1, 4096]. If the argument is outside this range, the interrupt rate will be arbitrary. The RATE argument is actually an overflow count, and the actual rate of the clock can be determined from:

$$RA = RE/RATE$$

where RE is the rate of the external input and RA is the actual clock rate. The advantage of an externally driven clock is that it may run at an arbitrarily high rate; however, specifying too high a rate may hang up the FORTRAN system. The calling sequence to define an external clock for the KW12A differs from that of a call for the DW8ES in that the KW12A calling program must enable Schmitt trigger 1. Optional clock execution is obtained on a KW12A external clock when RATE=1. Note that the arguments for a KW12A external clock are sufficient to enable a DK8ES external clock, but not vice versa.

CLOG† (COMPLEX NATURAL LOGARITHM FUNCTION)

CLOG calculates and returns the natural logarithm of its complex argument, as defined by the relation:

$$LOG(X+iY) = LOG(X**2+Y**2)+i*ATAN(Y/X)$$

CLRPT* (CLEAR PLOT SUBROUTINE)

The calling sequence:

```
CALL CLRPLT (N,BUFFER)
```

clears the current plot, if any, and assigns an N element buffer (designated BUFFER) which will hold 3N/2 points for display. The display is actually created by the PLOT subroutine. The variable BUFFER must be an array with at least N elements.

CMPLX† (REAL-TO-COMPLEX CONVERSION FUNCTION)

CMPLX accepts two real arguments and returns a complex value with real part equal to the first argument and imaginary part equal to the second argument.

CONJG† (COMPLEX CONJUGATE FUNCTION)

CONJG calculates and returns the complex conjugate of its complex argument. This is accomplished by leaving the real part of the argument unchanged, and negating the imaginary part.

COS (SINGLE-PRECISION COSINE FUNCTION)

COS calculates and returns the cosine of a real argument (in radians) by applying the identity:

$$COS(X) = SIN(X+\pi/2)$$

COSD (SINGLE-PRECISION COSINE IN DEGREES)

COSD calculates and returns the cosine of a real argument (in degrees). This is accomplished by adding 90 to the argument, converting the result to radians, and extracting the sine.

COSH (SINGLE-PRECISION HYPERBOLIC COSINE FUNCTION)

COSH calculates and returns the hyperbolic cosine of a real argument according to the relation:

$$\text{If } |x| \leqslant 88.029$$
$$COSH(x) = 1/2\left(EXP(x) + \frac{1.0}{EXP(x)}\right)$$

$$\text{If } |x| > 88.028 \text{ and } |x| - \log_e 2 \leqslant 88.028$$
$$COSH(x) = EXP(|x| - \log_e 2)$$

$$\text{If } |x| - \log_e 2 > 88.028$$
$$COSH(x) = 377737777777_8$$

and an error message is returned.

CPUT (CHARACTER PUT SUBROUTINE)

The calling sequence:

```
CALL CPUT (STRING,N,CHAR)
```

causes CPUT to insert CHAR as the Nth character in STRING, where STRING is a character string stored in A6 format, and CHAR is a number in the range [0, 63] which is interpreted as a character. The following program illustrates the use of CGET and CPUT.

```
      DATA STR/'HEY!'/
      WRITE(4,100) STR
  100 FORMAT('   HEY! IN ASCII   ',A6)
      WRITE(4,101)
  101 FORMAT('   HEY! IN DECIMAL')
      DO 10 I=1,4
      CALL CGET(STR,I,ICHAR)
      WRITE(4,102) ICHAR
   10 CONTINUE
  102 FORMAT(I6)
      DO 20 I=1,6
      J=2*I
      CALL CPUT(STR,I,J)
   20 CONTINUE
      WRITE(4,103) STR
  103 FORMAT('   NEW STRING    ',A6)
      CALL EXIT
      END

.R F4
*TCHRC/G$
  HEY! IN ASCII  HEY!
  HEY! IN DECIMAL
     8
     5
    25
    33
  NEW STRING    BDFHJL
```

CSIN† (COMPLEX SINE FUNCTION)

CSIN calculates and returns the sine of a complex argument according to the relation:

$$SIN(X+iY) = SIN(X)*COSH(Y)+i*COS(X)*SINH(Y)$$

CSQRT† (COMPLEX SQUARE ROOT FUNCTION)

CSQRT calculates and returns the square root of a complex argument.

DABS† (DOUBLE-PRECISION ABSOLUTE VALUE FUNCTION)

DABS returns the absolute value of its double-precision argument by negating the argument if it is negative, or returning it intact if it is positive.

DATAN† (DOUBLE-PRECISION ARC-TANGENT FUNCTION)

DATAN calculates and returns the primary arc-tangent of its double-precision argument. The argument is first reduced to the interval [0, ½] with the identities:

$$ATAN(-X) = -ATAN(X)$$

if $X>1.0$, $ATAN(X) = \pi/2 - ATAN(1/X)$

if $0.5<X<1.0$, $ATAN(X) = ATAN(1/2) + ATAN\left(\frac{2X-1}{X+2}\right)$

and the arc-tangent is then calculated as a continued fraction approximation.

DATAN2† (DOUBLE-PRECISION ARC-TANGENT OF TWO ARGUMENTS)

DATAN2 accepts two double-precision arguments, assumed to be an abscissa and an ordinate respectively, and calculates the arc-tangent of the quotient of the first argument divided by the second argument. The result is then adjusted, depending upon the quadrant in which a point defined by the arguments falls, in the same manner as for the ATAN2 function.

DATE (OS/8 DATE SUBROUTINE)

DATE accepts three integer arguments, accesses the current OS/8 system date, and returns an integer from 1 to 12 corresponding to the current month as the first argument, an integer from 1 to 31 corresponding to the current day as the second argument, and an integer from 1970 to 1977 corresponding to the current year as the third argument.

DBLE† (SINGLE- TO DOUBLE-PRECISION CONVERSION)

DBLE accepts a real argument and returns a double-precision value equal to the argument, filled out with zeroes in the low-order three words.

DCOS† (DOUBLE-PRECISION COSINE FUNCTION)

DCOS calculates and returns the cosine of a double-precision argument (in radians). This is accomplished by adding PI/2 to the argument and passing this result to the DSIN function.

DEXP† (DOUBLE-PRECISION EXPONENTIAL FUNCTION)

DEXP calculates and returns the exponential function of its double-precision argument by applying the method of Kogbetliantz (IBM Journal of Research and Development, April, 1957, pp 110-5).

DIM (SINGLE-PRECISION POSITIVE REAL DIFFERENCE)

DIM calculates and returns the positive difference of two real arguments. That is, if the first argument is larger than the second argument, DIM returns the difference between the arguments; if the first argument is less than or equal to the second argument, DIM returns 0.0.

DLOG† (DOUBLE-PRECISION NATURAL LOGARITHM)

DLOG calculates and returns the natural (Naperian) logarithm of its double-precision argument. This is accomplished by reducing the range of the argument through application of a method described by Ralston and Wilf in their text, Numerical Methods for Digital Computers, and then performing a Taylor series expansion.

DLOG10† (DOUBLE-PRECISION COMMON LOGARITHM)

DLOG10 calculates and returns the common (base 10) logarithm of its double-precision argument by extracting the natural logarithm and executing a change of base.

DMAX1† (DOUBLE-PRECISION MAXIMUM VALUE)

DMAX1 accepts an arbitrary number of double-precision arguments and returns the largest of the arguments.

DMIN1† (DOUBLE-PRECISION MINIMUM VALUE)

DMIN1 accepts an arbitrary number of double-precision arguments and returns the smallest of the arguments.

DMOD† (DOUBLE-PRECISION A MODULO B FUNCTION)

DMOD accepts two double-precision arguments and returns a double-precision value equal to the remainder when the first argument is divided by the second argument. If the second argument is not sufficiently large to prevent overflow, an error message and a value of 0.0 are returned.

DSIGN† (DOUBLE-PRECISION TRANSFER-OF-SIGN)

DSIGN accepts two double-precision arguments, calculates the absolute value of the first argument, and returns this value if the second argument is positive (or zero), or the negative of this value if the second argument is negative.

DSIN† (DOUBLE-PRECISION SINE FUNCTION)

DSIN calculates and returns the sine of a double-precision argument (in radians). The argument is first reduced to the range [0, PI/2], and the sine is then calculated from a Taylor series approximation.

DSQRT† (DOUBLE-PRECISION SQUARE ROOT)

DSQRT calculates and returns the (positive) square root of a positive double-precision argument. Any negative argument results in an error message.

EXP (SINGLE-PRECISION EXPONENTIAL FUNCTION)

EXP calculates and returns the exponential function of a real argument. The algorithm uses a numerical method after Kogbetliantz (IBM Journal of Research and Development, April, 1957, pp 110-5).

EXTLVL* (READ PDP-12 EXTERNAL LEVEL)

EXTLVL accepts two integer, real or logical arguments. The first argument is assumed to be a PDP-12 external level number in the range [0, 12]. If the referenced external level is at +3 volts (floating), the second argument is set equal to 0. If the referenced external level is at 0 volts (ground), the second argument is set equal to 1. If the first argument is outside the range [0, 12], the value returned in the second argument is unpredictable. If EXTLVL is called on a PDP-8, the second argument will always be set to zero.

FLOAT (INTEGER-TO-FLOATING-POINT CONVERSION)

FLOAT accepts an integer argument and returns a real variable equal to the argument.

IABS (INTEGER ABSOLUTE VALUE FUNCTION)

IABS calculates and returns the absolute value of an integer variable by leaving the variable unchanged if it is positive (or zero), and negating the variable if it is negative.

IDIM (INTEGER POSITIVE DIFFERENCE FUNCTION)

IDIM calculates and returns the positive difference of two integer arguments. That is, if the first argument is larger than the second argument, IDIM returns the difference between the arguments; if the first argument is less than or equal to the second argument, IDIM returns a value of 0.

IDINT (DOUBLE-PRECISION INTEGER TRUNCATION)

IDINT accepts a double-precision argument and returns the largest integer that is less than or equal to the argument.

IFIX (SINGLE-PRECISION FLOATING-POINT-TO-INTEGER FUNCTION)

IFIX is a floating-point truncation function. Given a real argument, it truncates the fractional part of the argument and returns the integral part as an integer. IFIX, AINT and INT perform the same function.

INT (SINGLE-PRECISION FLOATING-POINT-TO-INTEGER)

INT is a floating-point truncation function that performs the same function as AINT and IFIX.

ISIGN (INTEGER TRANSFER OF SIGN FUNCTION)

ISIGN accepts two integer arguments, calculates the absolute value of the first argument, and returns this value if the second argument is positive (or zero), or the negative of this value if the second argument is negative.

LSW* (READ PDP-12 LEFT SWITCH REGISTER)

LSW accepts two real, integer or logical arguments. The first argument is assumed to be a PDP-12 left switch register switch number in the range [0, 11]. Upon return, the second argument is set to the logical value of the referenced switch (either 0 or 1). If the first argument is outside the range [0, 11], the result that will be returned in the second argument is unpredictable. If LSW is called on a PDP-8, a value of 0 is always returned.

MAX0 (SINGLE-PRECISION MAXIMUM VALUE)

MAX0 accepts an arbitrary number of integer arguments and returns an integer result equal to the largest of the arguments.

MAX1 (SINGLE-PRECISION MAXIMUM VALUE)

MAX1 accepts an arbitrary number of real arguments and returns an integer result equal to the largest of the arguments.

MIN0 (SINGLE-PRECISION MINIMUM VALUE FUNCTION)

MIN0 accepts an arbitrary number of integer arguments and returns an integer value equal to the smallest of the arguments.

MIN1 (SINGLE-PRECISION MINIMUM VALUE FUNCTION)

MIN1 accepts an arbitrary number of real arguments and returns an integer value equal to the smallest of the arguments.

MOD (INTEGER A MODULO B FUNCTION)

MOD accepts two integer arguments and returns an integer value equal to the remainder when the first argument is divided by the second argument. If the second argument is not sufficiently large to prevent overflow, an error message and a value of 0 are returned.

ONQB (PLACE TASK ON BACKGROUND JOB CHAIN)

ONQB is a subroutine which is called from PDP-8 mode RALF code to place a PDP-8 mode task on the list of background tasks. These background tasks are executed in round-robin order whenever the PDP-8 processor has nothing to do (e.g., while waiting for terminal input). If FPP-12 hardware is present, these background subroutines execute in parallel with the execution of the FORTRAN program by the FPP-12. ONQB is called by a sequence such as:

```
        JMS%    XONQB+1
        ADDR    BRJOB

        EXTERN  ONQB
XONQB,  ADDR    ONQB
```

where BRJOB is the address of the background job, a subroutine which must obey all the conventions of ONQI. ONQB resides in field 1 and should only be called from field 1. See the *FORTRAN IV Software Support Manual* for details.

ONQI (PLACE INTERRUPT HANDLER ON SKIP CHAIN)

ONQI is a subroutine which is called from PDP-8 mode RALF code to put the interrupt handler of a device on the interrupt skip chain. When an interrupt is received by the PDP-8 processor, the processor checks each device on the skip chain, then the FPP, then the standard FORTRAN peripherals, e.g., line printer. If the interrupt was caused by a device with a handler on the skip chain, the PDP-8 processor branches to the handler. ONQI is called by a sequence such as:

```
        JMS%    XONQI+1
        IOT
        ADDR    IHNDLR

XONQI,  ADDR    ONQI
        EXTERN  ONQI
```

where IOT is the actual IOT code for the device skip-on-flag instruction and 1HNDLR is the address of the interrupt handler for this device. ONQI always resides in field 1 and must be called by PDP-8 mode RALF code in field 1 only. The interrupt handler is entered with the AC cleared and the data and instruction fields set to 1. It should return with these registers in the same state. ONQI should not be called more than once for any given IOT.

PLOT* (DISPLAY DATA ON PDP-12 OR LAB-8/E SCOPE)

The calling sequence:

CALL PLOT (M,X,Y)

plots M points whose X coordinates are in the array X and whose Y coordinates are in the array Y into the plot buffer specified by the CLRPLT routine. A background task plots the contents of all points entered into the plot buffer on the scope whenever the PDP-8 processor would otherwise be idle. When X is 1, X and Y are interpreted as scalars. The scope is scaled with (0,0) in the lower left corner and (1.3,1.0) in the upper right corner. These values may be altered by a call to SCALE.

PLOTR* (CHANGE SCOPE BUFFER VALUES)

The calling sequence:

CALL PLOTR (M,X,Y,I)

alters the M entries in the plot buffer beginning at the Ith entry, getting the new X coordinates from the array X and the new Y coordinates from the array Y. Calling this subroutine does not alter the number of points displayed by the background display task.

RCLOSE* (CLOSE A PDP-12 RELAY)

RCLOSE accepts an integer, real, or logical argument assumed to be a PDP-12 relay number in the range [0, 5] and closes the referenced relay. If the argument falls outside the specified range, the result is unpredictable. RCLOSE has no effect when called on a PDP-8.

REAL† (COMPLEX-TO-REAL CONVERSION FUNCTION)

REAL accepts a complex argument and returns a real value equal to the real part of the argument.

REALTM* (BUFFERED/CLOCKED SAMPLING)

REALTM performs buffered/clocked sampling on the PDP-12 or LAB-8/E. The calling sequence is:

```
CALL   REALTM   (BUFFER,LENGTH,CSTART,NCHANL, NPTS)
```

where:

BUFFER = array to be used by REALTM as a ring buffer.

LENGTH = size of BUFFER.

CSTART = first channel to sample at each clock interrupt (0-15).

NCHANL = number of channels to sample at each time step. If NCHANL = 1, then argument 1 of the call to CLOCK may specify clock-initiated A/D sampling (8 images). If NCHANL>1, then arg 1 of CLOCK CALL should not specify clock-initiated sampling. Fetching of the first sample will be initiated in the clock interrupt routines, or 50-100 μs after the clock tick. The other samples are taken as soon as possible, about 100-200 μs later for each sample.

NPTS = total number of samples to take.

Algorithm and Comments

The following program samples 500 points from channel 3 at 10 Hz. and plots them on the scope:

```
        DIMENSION PLTBUF(400),DATBUF(50)
1       CALL CLRPLT(400,PLTBUF)
        CALL REALTM (DATBUF,50,3,1,500)
        CALL CLOCK (8,10)
        DO 100 I=1,500
100     CALL PLOT(1,I/384.,ADB(X)/1024.+.5)
C       NOW PAUSE SO THAT POINTS WILL BE DISPLAYED
        READ(1,10)Q
10      FORMAT(I2)
        GO TO 1
        STOP
        END
```

ROPEN* (OPEN A PDP-12 RELAY)

ROPEN accepts one integer, real or logical argument, assumed to be a PDP-12 relay number in the range [0, 5], and opens the referenced relay. If the argument falls outside the specified range, the result is unpredictable. ROPEN has no effect when called on a PDP-8.

RSW (READ SWITCH REGISTER)

RSW accepts two real, integer or logical arguments. The first argument is assumed to be a switch register switch number in the range [0, 11]. The second argument is set to the logical value of the referenced switch (right switch register on the PDP-12). If the first argument falls outside the range [0, 11], the result that will be returned in the second argument is unpredictable.

SCALE* (DEFINE SCALE OF SCOPE)

SCALE defines the scope screen scaling for calls to PLOT. The calling sequence is:

```
CALL SCALE (XLO, YLO, XHI, YHI)
```

where:

XLO is the value at the left edge of the screen.
YLO is the value at the bottom of the screen.
XHI is the value at the right edge of the screen.
YHI is the value at the top of the screen.

If SCALE is never called, the assumed values are equivalent to:

```
CALL SCALE (0,0,1.3,1.0)
```

SIGN (SINGLE-PRECISION TRANSFER OF SIGN)

SIGN accepts two real arguments, calculates the absolute value of the first argument, and returns this value if the second argument is positive (or zero), or the negative of this value if the second argument is negative.

SIN (SINGLE-PRECISION SINE FUNCTION)

SIN calculates and returns the sine of a real argument (in radians). The argument is reduced to the first quadrant, and the sine is then computed from a Taylor series expansion.

SIND (SINGLE-PRECISION SINE (DEGREES) FUNCTION)

SIND calculates and returns the sine of a real argument (in degrees). This is accomplished by converting the argument to radians and passing this value to the SIN function.

SNGL† (DOUBLE- TO SINGLE-PRECISION CONVERSION)

SNGL accepts a double-precision argument, truncates the low-order bits, and returns the resulting real value.

SINH (SINGLE-PRECISION HYPERBOLIC SIGN)

SINH calculates and returns the hyperbolic sine of a real argument according to the relations:

$$\text{If } 0.10<|x|<87.929, \quad \text{SINH}(x) = 1/2\left[\text{EXP}(x) - \frac{1}{\text{EXP}(x)}\right]$$

$$\text{If } |x|\leq 0.10, \quad \text{SINH}(x) = x + x^{3}/6 + x^{5}/120$$

$$\text{If } |x|>88.028, \quad \text{SINH}(x) = [\text{EXP}(|x| - \log_e 2)] \cdot [\text{signum}(x)]$$

SQRT (SINGLE-PRECISION SQUARE ROOT FUNCTION)

SQRT calculates and returns the (positive) square root of a positive real argument. Any negative argument results in an error message.

SSW* (READ PDP-12 SENSE SWITCH)

SSW accepts two real, integer or logical arguments. The first argument is assumed to be a PDP-12 sense switch number in the range [0, 5]. The second argument is set to the logical value of the referenced sense switch. If SSW is called on a PDP-8, a value of zero is always returned. If the first argument falls outside the range [0, 5], the result that will be returned in the second argument is generally unpredictable. The exception is the calling sequence:

```
CALL SSW (14,RUA12)
```

which returns RUA12=0 on a PDP-8 and RUA12=1 on a PDP-12.

SYNC* (READ A SCHMITT TRIGGER)

SYNC determines whether a Schmitt trigger has been fired, and must not be called unless CLOCK has been called at least once. SYNC accepts two real, integer or logical arguments. The first argument is assumed to be a Schmitt trigger number in the range [1, 3]. The second argument is set to one if the referenced Schmitt

trigger has fired since the last time it was read, or to zero otherwise. The referenced Schmitt trigger is also reset to the not-fired, or zero, state. A call to CLOCK sets all triggers to the zero state, and any trigger that was not enabled by a call to CLOCK is always in the zero state. If the first argument falls outside the range [1, 3], an unpredictable result (either zero or one) is generally returned. If the first argument is zero, however, a value of zero is always returned.

TAN (SINGLE-PRECISION TANGENT FUNCTION)

TAN calculates and returns the tangent of a real argument (in radians). This is accomplished by computing the quotient of the sine of the argument divided by the cosine of the argument; thus, if the cosine of the argument is zero, an error message is returned.

TAND (SINGLE-PRECISION TANGENT, DEGREES)

TAND calculates and returns the tangent of a real argument (in degrees). This is accomplished by converting the argument to radians and passing the resulting value to the TAN routine.

TANH (SINGLE-PRECISION HYPERBOLIC TANGENT)

TANH calculates and returns the hyperbolic tangent of a real argument by computing the quotient of the hyperbolic sine of the argument divided by the hyperbolic cosine of the argument.

TIME* (READ TIME OF DAY)

TIME may be called as a subroutine with one real or integer argument, or as a function with a dummy argument. It returns the elapsed time since the clock was started. This result will be in seconds unless the clock is running under external input, in which case it will be in external ticks, with the interval between ticks specified by the clock rate (see CLOCK).

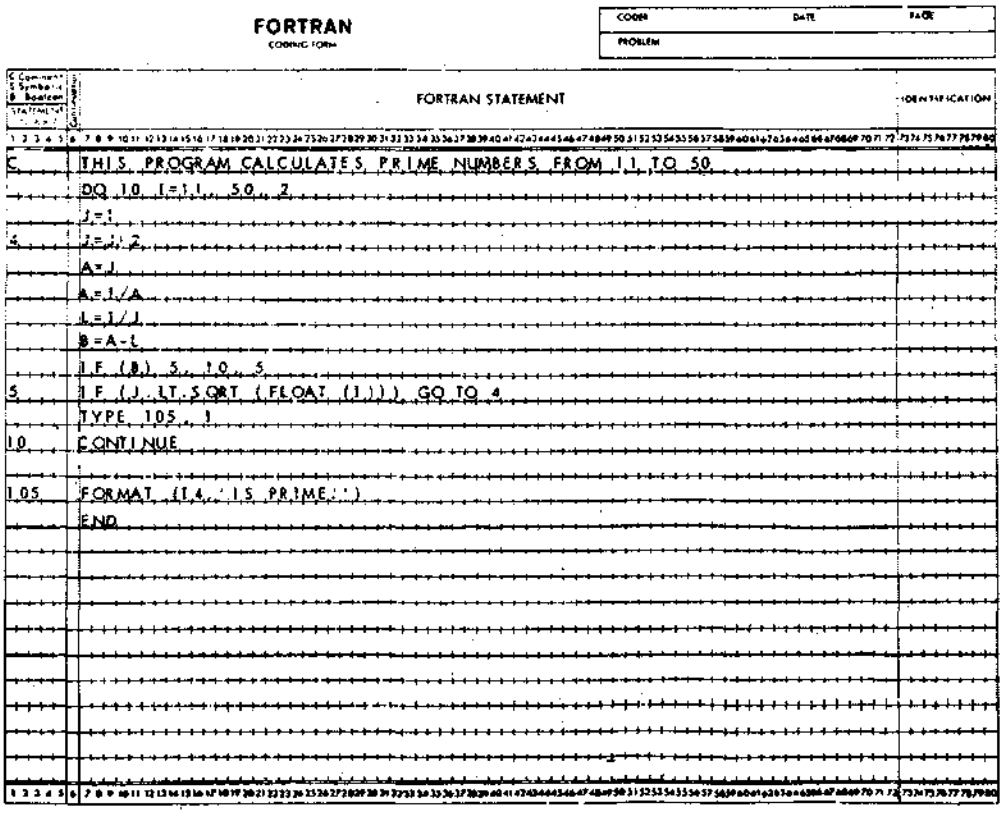
FORTRAN IV SOURCE LANGUAGE

The FORTRAN language is composed of mathematical-form statements constructed in accordance with precisely formulated rules. FORTRAN (source) language programs consist of meaningful sequences of FORTRAN statements that direct the computer to perform specified operations and computations. OS/8 FORTRAN IV is compatible with ANSI Standard FORTRAN; that is, programs written in compliance with the ANSI Standard FORTRAN (3.9-1966) are acceptable to OS/8 FORTRAN IV.

Certain features of OS/8 FORTRAN IV are not defined by the standard, so users intending to write OS/8 FORTRAN IV programs to be used on other machines should ensure either that they do not use the non-standard features, or that the other machines on which the programs are to be run also include these features.

FORTRAN source programs are generally written on a coding sheet such as the one shown in Figure 8-4. Each line of a program contains three fields: statement number field, line continuation field, and statement field. A fourth field, the identification field consisting of columns 73 to 80, is ignored by the compiler. It may be used to number statements sequentially or for any other purpose.

A statement number consists of one to five digits entered in columns 1-5. Leading zeros or blanks (leading and trailing) are retained on the listing, but otherwise ignored in this field. Statement numbers may be assigned in any order, but they must be unique. Any statement referenced by another statement must have a statement number. Statement numbers on specification statements are ignored.


FORTRAN
CODING FORM

CODER | DATE | PAGE
PROBLEM

FORTRAN STATEMENT | IDENTIFICATION

```
C     THIS PROGRAM CALCULATES PRIME NUMBERS FROM 11 TO 50
      DO 10 I=11, 50, 2
      J=1
6     J=J+2
      A=J
      A=I/A
      L=I/J
      B=A-L
      IF (B) 5, 10, 5
5     IF (J.LT.SQRT (FLOAT (I))) GO TO 6
      TYPE 105, I
10    CONTINUE
105   FORMAT (I4,' IS PRIME.')
      END
```

Figure 8-4 FORTRAN IV Coding Form

If a FORTRAN statement is so large that it cannot conveniently fit into one statement field, the statement fields of up to 5 additional lines may be used to specify the complete statement. The first line of a statement must have a blank in column 6. Continuation lines must have some character other than a blank in column 6.

FORTRAN statements define arithmetic operations, call for input or output, and alter the sequence of program execution. Any FORTRAN statement may appear in the statement field (columns 7-72). Except when they occur as alphanumeric data within a FORMAT statement, DATA statement, or literal constant, blanks (spaces) are ignored and may be used freely for appearance purposes. A TAB at the beginning of a line or after the statement number causes spacing to column 7. The first TAB after column 7 is treated as a blank by the compiler. Any input line that does not contain a TAB or at least 6 other characters in columns 1-72 is ignored by the compiler.

Comments explaining the program may be written in any format. A line which contains the letter C in column 1 is interpreted as a line of comments. Comment lines are printed on all listings, but are otherwise ignored by the compiler. A comment line must not immediately precede a continuation line.

Constants, Variables, and Expressions

CONSTANTS

The constants, variables, and expressions described below are basic to expressing data values in the FORTRAN language. Seven types of constants are used in OS/8 FORTRAN IV programs: integer, real, double precision, octal, complex, logical, and Hollerith.

Integer Constants

An integer constant consists of from one to seven decimal digits written without a decimal point. Negative constants must be preceded by a minus (−) sign; however, the plus (+) sign preceding a positive constant is optional. Embedded commas and blanks are not allowed in integer constants.

Examples:

0
+051
−440
6073

Integer constants must fall within the range −2↑23 to 2↑23−1 (−8,388,608 to 8,388,607 decimal). When used as subscripts, integer constants are taken modulo 2↑12 (4096 decimal). The following are illegal as integer constants:

10.3	(decimal point)
5,000	(comma)
9000000	(outside acceptable range)

Real Constants

A real constant is an integer constant followed by a decimal point, a second string of digits and an optional exponent. Only the leftmost six digits, aside from leading zeros, are used by the compiler. A negative constant must be preceded by a minus (−) sign. The plus (+) sign preceding a positive real constant is optional.

Real constants may be entered in exponential notation, as illustrated below, by specifying a positive or negative decimal value followed by the letter E and a 1-3 digit integer which may be positive, negative or zero. The value of the real constant is taken as the value of the decimal number preceding the letter E multiplied by that power of 10 indicated by the integer following the letter E. This notation eliminates leading and trailing zeroes from very large or very small real constants. The absolute value of any real constant must fall within the approximate range 10↑-615 to 10↑615 (or zero). Examples:

0.0	
.579	
−10.794	
5.0E03	(i.e., 5000.)
5.0E+3	(i.e., 5000.)
5.0E−3	(i.e., 0.005)
5.0E0	(i.e., 5.0)
5E0	

The following are not valid real constants.

6,517.6	(comma)
131	(no decimal point or exponent)
1DE	(no exponent)
20E1.5	(exponent must be integer)

Double Precision Constants

A double precision constant is a real constant that contains extra significant digits. Aside from leading zeroes, only the leftmost 17 significant digits of a double precision constant are used by the compiler. The decimal point may be omitted from a double precision constant that does not have a fractional component. In other respects, double precision constants conform to the same format as real constants, except that the letter D is used in place of the letter E, preceding the exponent, when exponential notation is employed. Double precision arithmetic requires the presence of an FPP with extended precision option.

Examples:

```
24.671325982134D0
 3.6D2          (i.e., 360.)
 3.6D−2         (i.e., .036)
 3.0D0
 3D0
```

Octal Constants

An octal constant is a string of octal digits (0-7 only) preceded by the letter O. Only the 12 low-order digits are used by the compiler. Octal constants are valid only in DATA statements where they are generally used to set bits for masking purposes.

Examples:

```
DATA JOB/01032/
DATA BASE/07777/
```

Complex Constants

FORTRAN IV provides for direct operations on complex numbers. A complex constant is written as an ordered pair of real constants separated by a comma and enclosed in parentheses.

Examples:

```
(.70712, −.70712)
(8.763E3, 2.297)
```

The first constant of the pair represents the real part of the complex number, and the second constant represents the imaginary part. The real and imaginary parts may each be signed. The enclosing parentheses are part of the constant and always appear, regardless of context. The two parts are represented internally by

single precision floating-point numbers occupying adjacent positions in memory. Complex arithmetic can only be done on the FPP using the extended precision option.

Logical Constants

The two logical constants (.TRUE. and .FALSE.) have the internal values 1. and 0., respectively. Logical constants may be entered in DATA or input statements as .TRUE. or .FALSE. (or abbreviations .T. or .F.). The enclosing periods are part of the constant and always appear. Logical quantities are operated upon by logical operators only.

Hollerith Constants

A hollerith constant (or literal constant) is a string of ASCII characters. There are two forms by which a Hollerith constant may be represented.

Form 1: nH character string

where n is the number of characters following the H.
Examples:

```
5HWORDS
3H123
```

Form 2: 'character string'
Examples:

```
'WORDS'
'123'
```

The single quote character which delimits a Hollerith constant in form 2 may be included in the character string if immediately preceded by a second single quote character. Thus, 'DON' 'T' will be stored as DON'T.

A Hollerith value may be entered in a DATA statement or FORMAT statement as a string of one to six ASCII characters per integer or real variable, and one to twelve per complex or double precision variable.

VARIABLES

A variable is a quantity which is represented by a symbolic name. Arithmetic statements and ASSIGN statements are used to

change the value of a variable, by computation or assignment, during program execution. I/O statements and subroutine calls can also change the value of a variable. A variable name is a string of one to six alphanumeric characters, the first of which must be alphabetic:

Valid Names		Invalid Names
J	1ACT	(first character number)
ALPHA	STANDARD	(too long)
MAX	FILE 1	(space within name)
A34	#MAIN	(# not alphanumeric)

There are five types of variables: integer, real, double precision, complex, or logical. Definitions for these types correspond to definitions of constants of the same type, i.e., integer variables take on a value of from zero to any positive or negative integer in the range −8,388,607(decimal) to 8,388,607(decimal); real variables contain a decimal point; etc.

Type classification is assigned to a variable explicitly via a type declaration statement or by virtue of the initial letter of its name. A first letter of I, J, K, L, M, or N indicates an integer variable. Any other first letter indicates a real variable. The type declaration statement overrides the type assigned by an initial letter.

Arrays

Variables can be either scalar (representing a single quantity) or array (representing many quantities with one name). An entire array is identified by its name, while a single element of the array is identified by a subscript, in parentheses, following the array name.

Variable	Refers To
ARRAY(1)	The first element of a one-dimensional array named ARRAY.
B(1,3)	The element located in the first row and the third column of a two-dimensional array named B.

Subscripts

The subscripts of an array variable can be integer constants or expressions. For example, A(1), A(ONE), and A(I+1,2*K+3*J) illustrate valid subscripts. The elements of an array must be of the same type, i.e., all real or all logical.

The extent of an array is determined by the dimensions it is assigned. This may be done by means of a DIMENSION or COMMON statement, or as part of a type declaration statement.

EXPRESSIONS

An expression is a combination of elements (constants, subscripted or nonsubscripted variables, and function references), each of which is related to another by operators and parentheses. An expression represents one single value that is the result of calculations specified by the elements and operators that make up the expression. An expression may, itself, function as an element in another expression if it is enclosed in parentheses. The FORTRAN language provides two kinds of expressions: arithmetic and logical.

Arithmetic Expressions

An arithmetic expression is a combination of constants, variables, and function references separated by arithmetic operators and parentheses. In the absence of parentheses, algebraic operations within arithmetic expressions are performed in the following descending order:

**	exponentiation
–	unary minus
* and /	multiplication and division
+ and –	addition and subtraction
=	equals or replacement sign

Parentheses are used to change this order of precedence. An operation enclosed in parentheses is performed before its result is used in other operations. In the case of operations of equal precedence, the calculations are performed from left to right. Additional computations (such as sine, cosine, or square root extraction) may be specified via a function reference.

An arithmetic expression may consist of a single constant, variable, or function call, referred to as a basic element. For example:

```
2.71828
Z(N)
TAN (THETA)
```

Any function reference acts as a basic element in an expression, since all functions return a unique value for any given argument. The reference SQRT(4.), for example, always represents the value 2. in an expression.

Any arithmetic expression may be enclosed in parentheses and considered as a basic element. For example:

```
IFIX(X+Y)/2
(ZETA)
(COS(SIN(PI*EM)+X))
```

Compound arithmetic expressions may be formed using numeric operators to combine basic elements. For example:

```
X+3
TOTAL/A
PI*EM
```

A basic element preceded by a + or − sign is also an arithmetic expression. For example:

```
+X
−(ALPHA*BETA)
—(SQRT(—GAMMA))
```

With the exception of unary minus, no two arithmetic operators may appear in sequence. For instance, X*/Y is illegal.

Parentheses do not imply multiplication, thus (A+B)(C+D) is improper. This expression must be written:

```
(A+B)*(C+D)
```

A typical numeric expression using numeric operators and a function reference, the expression for one of the roots of the general quadratic equation

$$\frac{-b+\sqrt{b\uparrow 2-4ac}}{2a}$$

might be coded as:

```
(−B+SQRT(B**2−4.*A*C))/(2.*A)
```

The following examples illustrate conversion of other mathematical expressions into FORTRAN expressions.

a+5	A+5.0
a.b	A*B
a↑(b+2)	A**(B+2)
$\left(\frac{b+d}{a}\right)^{2.5}$	((B+D)/A)**2.5

In general, only real and integer quantities may be mixed in arithmetic expressions. No other type mixing is legal. Logical variables and constants may only be operated upon by logical operators (.AND., .OR., .NOT., .XOR., .EQV.). Hollerith literals in expressions have type integer, with only the first six characters being used.

Logical Expressions

A logical expression combines logical constants, logical variables, logical function references, and logical expressions, using the logical or relational operators given below.

Logical operators can combine only basic elements whose type is logical. Relational operators compare units of type integer, real, or double-precision. The value of such an expression will be of logical type (that is, .TRUE. or .FALSE.). The relational operators .EQ. and .NE. may also be used with complex expressions. Complex quantities are equal if the corresponding parts are equal.

Logical Operator	Meaning
.NOT. expr	Has the value .TRUE. only if the expression is .FALSE., and has the value .FALSE. only if the expression is .TRUE.
expr1.AND.expr2	Has the value .TRUE. only if expr1 and expr2 are both .TRUE., and has the value .FALSE. if either expr1 or expr2 or both are .FALSE.
expr1.OR.expr2	(Inclusive OR) Has the value .TRUE. if either expr1 or expr2 or both are .TRUE., and has the value .FALSE. only if both expr1 and expr2 are .FALSE.
expr1.XOR.expr2	(Exclusive .OR.) Has the value .TRUE. if either expr1 or expr2, but not both, are

Logical Operator	Meaning
	.TRUE., and has the value .FALSE. otherwise.
expr1.EQV.expr2	(Equivalence) Has the value .TRUE. if expr1 and expr2 are both .TRUE. or both .FALSE., and has the value .FALSE. otherwise.

Relational operator	Relation
.GT.	greater than
.GE.	greater than or equal to
.LT.	less than
.LE.	less than or equal to
.EQ.	equal to
.NE.	not equal to

The enclosing periods are part of the logical and relational operators, and must be present.

A logical expression, like an arithmetic expression, may consist of basic elements or a combination of elements, as in

```
.TRUE.
X.GE.3.14159
```

or

```
TVAL.AND.INDEX
BOOL(M).OR.K.EQ.IMIT
```

where BOOL is a logical function with 1 argument, or a singly-dimensioned logical array. A logical expression may also be enclosed in parentheses and function as a basic element. Thus, the expressions:

```
A.AND.(B.OR.C)
```

and

```
(A.AND.B).OR.C
```

are evaluated differently.

No two logical operators may appear in sequence, except in the case where .NOT. appears as the second of two logical operators. Any logical expression may be preceded by the unary operator .NOT. as in:

```
.NOT.T
.NOT.X+7.GT.Y+Z
BOOL(K).AND..NOT.(TVAL.OR.R)
```

Logical and relational operations (unless overridden by parentheses) are carried out in the following order:

```
.GT.,.GE.,.LT.,.LE.,.EQ.,.NE.
.NOT.
.AND.
.OR.
.EQV.,.XOR.
```

For example, the logical expression

```
.NOT.ZETA**2+Y*MASS.GT.K-2.OR.PARITY.AND.X.EQ.Y
```

is interpreted as

```
(.NOT.(((ZETA**2)+(Y*MASS)).GT.(K-2))).OR.(PARITY. AND.(X.EQ.Y))
```

There are 16 logical operators theoretically possible between the logical expressions. Two of them are constants (true and false) and four are unary operators (that is, the value of one of the two expressions is irrelevant to the value of the operation). These six are marked by asterisks in Table 8-12. The remaining ten operators can be most conveniently represented as shown at the right of the table, with A and B representing the two logical expressions involved.

Assignment Statements

A variable may be assigned a value at any point in the source program. During program execution, the most recent assignment determines the variable's value in subsequent statements. The statements which may be used to assign a value to a variable are the arithmetic and logical statements which assign a numeric or logical value and the ASSIGN statement which assigns a statement number.

ARITHMETIC STATEMENTS

Arithmetic statements indicate computations to be performed by OS/8 FORTRAN IV.

Form	v=e	
where	v	is a variable name
	e	is an expression
	=	is the replacement operator
Effect		The variable v is assigned the value of expression e.

Table 8-12 Truth Table for Logical Expressions

Expressions Involved:						
	A	F	F	T	T	FORTRAN IV
	B	F	T	F	T	Expressions
Function						
*	FALSE	F	F	F	F	.FALSE.
	AND	F	F	F	T	A .AND. B
		F	F	T	F	A .AND. .NOT. B
*	A	F	F	T	T	A
		F	T	F	F	.NOT. A .AND. B
*	B	F	T	F	T	B
	XOR	F	T	T	F	A .XOR. B
	OR	F	T	T	T	A .OR. B
	NOR	T	F	F	F	.NOT. (A .OR. B)
	EQV	T	F	F	T	A .EQV. B
*	NOT B	T	F	T	F	.NOT. B
		T	F	T	T	A .OR. .NOT. B
*	NOT A	T	T	F	F	.NOT. A
		T	T	F	T	.NOT. A .OR. B
	NAND	T	T	T	F	.NOT. (A .AND. B)
*	TRUE	T	T	T	T	.TRUE.

The arithmetic statement associates a variable name with a value. This name may then be used in subsequent expressions to represent the value. Thus, if the arithmetic statement A=2 is executed first, the statement B=A+1 is equivalent to the statement B=2+1, or B=3.

Since the equal sign in the arithmetic statement does not indicate equality but, rather, a replacement; statements of the form:

```
I = I+1
```

are perfectly legal. The arithmetic statement is, in fact, the only means in FORTRAN by which the results of computations represented by expressions may be stored.

In the following examples, the expression to the right of the equal sign is evaluated and converted when necessary to conform to the type of the variable to the left of the equal sign. The converted value is stored in the storage location associated with the variable name to the left of the equal sign. That is, if a real expression is assigned to an integer variable, the value of the expression is converted to an integer before assignment.

Examples:

```
ANS=Y*(X**2+Z)
I=1*N
X(J)=A(J)-B(J)
P=.TRUE.
S=D.LT.5
```

The expression to be assigned must be capable of yielding a value which conforms to the type attribute of the variable to which it is being assigned. The compiler performs conversions in accordance with Table 8-13.

THE GO TO ASSIGNMENT STATEMENT

The ASSIGN statement is used in conjunction with an assigned GO TO statement to permit symbolic referencing of statements.

Form	ASSIGN n to var
Where	n is a statement number var is an integer or real variable
Effect	The variable represents the assigned statement number and may be used in an assigned GO TO statement.

Table 8-13 Conversion Rules for Assignment Statements

TO: / FROM:	REAL	INTEGER	COMPLEX	DOUBLE PRECISION	LOGICAL CONSTANT	LITERAL CONSTANT
Real	D	D	R,D	H,D	D	D,6
Integer	C	D	R,C	H,C	D	D,6
Complex	D,R,I	D,R,I	D	H,D,R,I	D,R,I	D,6
Double Precision	D,H,L	D,H,L	R,D,H,L	D	D,H,L	D,6
Logical	N	N	R,N	H,N	D	N,6

N—Convert non-zero to 1.0 (logical truth)
D—Direct replacement
C—Conversion between integer and floating point
R—Real only (imaginary part set to 0)
I—Set imaginary part to 0
H—High order portion of expression assigned
L—Set low-order part to 0
6—Use the first character in the literal and five characters following

The statement number assigned must be that of an executable statement. If more than one ASSIGN statement refers to the same integer variable name, the value assigned by the last executed statement is the current value.

An integer variable which has obtained its value via an ASSIGN statement must be redefined via an arithmetic statement before it can be used in any context other than the GO TO statement. For example, the statement:

```
ASSIGN 10 TO COUNT
```

associates the variable name COUNT with statement number 10 and the statement:

```
COUNT=COUNT+1
```

is then invalid. The statement becomes valid, however, if preceded by an arithmetic assignment statement such as:

```
COUNT=10
```

which assigns COUNT the integer value of 10. The use of an arithmetic assignment, however, invalidates any future use of the variable COUNT in an assigned GO TO.

An assigned GO TO must not be used to transfer program control outside of the program or subprogram in which it appears.

Control Statements

Statements are normally executed in the sequence in which they appear in the source program. This sequence may be altered by the use of the FORTRAN control statement: GO TO, IF, DO, CONTINUE, PAUSE, STOP, CALL and RETURN. The CALL and RETURN statements, which transfer control to and from subroutines, are described later in this section.

GO TO STATEMENTS

The GO TO statement transfers control directly to a specified statement. There are three forms of the GO TO statement: unconditional, computed, and assigned. A GO TO statement may appear anywhere in the executable portion of the source program except as the terminal statement in a DO loop.

Unconditional GO TO Statement

Form GO TO n

Where n = the statement number of an executable statement

Effect Control is transferred to statement n

When control is transferred by a statement of the form GO TO n, the usual sequential processing continues at the statement whose number is n.
Examples:

```
GO TO 50
GO TO 1020
```

Computed GO TO Statement

Form GO TO (n1,n2,...,nK)e

An optional comma may follow the right parenthesis.

Where n1,n2,...,nk are statement numbers

e is a positive (non-zero) integer expression whose value is less than or equal to the number of statement numbers within the parentheses.

Effect Control is transferred to the statement whose number is e^{th} in the list of statement numbers.

The integer expression in a computed GO TO statement acts as a switch, as in the example given below:

```
GO TO (20,10,6),K
```

If K=1, control will be transferred to statement 20; if K=2, to statement 10; or if K=3, to statement 6. If K has a value less than 1 or greater than the number of statements within the parentheses, unpredictable results occur.

Assigned GO TO Statement

Form GO TO v
or
GO TO v,(n1,n2,...,nk)

Where v is an integer variable

n1,n2,...,nk are statement numbers whose values may have been assigned to v

Effect Control is transferred to the statement whose number is currently associated with the variable v via an ASSIGN statement.

An ASSIGN statement defines an integer or real variable as a statement number. Thus, when the statement:

```
ASSIGN 10 TO LOOP
```

has been executed, control is transferred to statement 10 by the assigned GO TO statements:

```
GO TO LOOP
```

or

```
GO TO LOOP, (10,20,100)
```

either of which may be used to transfer control to whichever statement number is currently associated with LOOP.

An assigned GO TO statement must never be used to transfer program control outside of the program or subprogram in which it appears.

IF STATEMENTS

An IF statement causes control to be transferred on the basis of the value of a specified expression. There are two forms of the IF statement: arithmetic and logical.

Arithmetic IF Statement

Form IF (arithmetic expression) n1,n2,n3

Where n1,n2,n3 are statement numbers

Effect Control is transferred to:
n1 if expression <0
n2 if expression = 0
n3 if expression >0

An IF statement transfers control to one of three statements, as shown in the model, according to the value of the expression given. For example, the statements:

```
ALPHA=3
IF (ALPHA) 10,20,30
```

transfer control to statement number 30. Complex expressions may be used in an arithmetic IF statement; however, only the real part is used in the comparison. If less than three statement numbers are present, control passes to the next sequential statement for each of the missing conditions. Thus:

```
IF (ALPHA) 10
STOP
```

transfers control to 10 if ALPHA is negative, otherwise it executes the STOP statement.

Logical IF Statement

Form IF (logical expression) statement

Where statement may be any executable statement except another logical IF or a DO statement

Effect the statement given is executed if the expression has the value .TRUE., otherwise, the next statement in sequence is executed.

Examples:

```
LOGICAL T,F
IF (T.OR.F)X=Y+1
IF (Z.GT.X) CALL SWITCH (S,Y)
IF (K.EQ.INDEX) GO TO 15
```

DO STATEMENT

DO statements provide for the repeated execution of a statement or series of statements.

Form DO n i=m1,m2,m3

Where n is a statement number

i is a nonsubscripted integer or real variable

m1,m2,m3 are integer or real constants or expressions

Effect i is set to m1 and statements following the DO statement up to and including statement n are executed repeatedly increasing i by m3 at the end of each iteration, until i is greater than m2.

The statements which are executed as a result of a DO statement are called the range. The variable i is called the index. The values m1, m2, and m3 are, respectively, the initial, limit, and increment values of the index. Note that the range of a DO need not be merely a section of straight line code following the DO statement. A control statement which causes instructions elsewhere in the program to be executed is permissible, as long as control eventually comes back to the terminal statement. When the range contains such control statements, it is called an extended range.

If m3 is omitted, an increment of 1 is assumed. A zero or negative increment is not permitted. The range of a DO is always executed at least once, regardless of the values of the limit and increment. After each execution of the range, the increment value is added to the value of the index and the result is compared with the limit value. If the value of the index is not greater than the limit value, the range is executed again using the new value of the index.

Examples:

```
DO 5 I=1,100
(I=100 during last iteration of DO loop)

DO 20 I=5,100,2
(I=99 during last iteration of DO loop)

DO 100 I=0,100,2
(I=100 during last iteration of DO loop)
```

After the last execution of the range, control passes to the statement immediately following it. This exit from the range is called the normal exit. Exit may also be accomplished by the execution of a control statement within the range.

The values of the initial, limit and increment variables or expressions of the DO loop may be altered within the range of the DO statement. Such alteration will not affect the operation of the loop, since the values of m1, m2, and m3 are remembered by the program. Altering the index will affect the number of iterations of the loop, however. This value is available for program use as a variable.

For example:

```
        DO 40 I=1,10
        TEMP=I-1
        ANUMBR=TEMP*0.1
40      ROOT=SQRT(ANUMBR)
```

In this example, the value of the index I is used as the minuend in determining the value of TEMP. Also, when a statement transfers control outside the range of a DO loop, e.g., by a GO TO or IF, the index retains its current value and is available for use as a variable. A transfer into a DO loop from outside its range may cause improper partial execution of the loop unless the transfer into the range is a return from the extended range.

The terminal statement of a DO range must not be a GO TO, DO, RETURN, STOP, PAUSE, or an arithmetic IF statement. A logical IF statement is allowed as the last statement of the range, provided that it does not contain any of the statements mentioned above. As an example:

```
        DO 5 K=1,4
5       IF (X(K).GT.L) Y(K)=X(K)
6       ...
```

In this case, the range is considered ended when, and if, control would normally pass to the statement following the entire logical IF statement. Statement 5 is executed four times whether or not the statement Y(K)=X(K) is executed. Statement 6 is not executed until statement 5 has been executed four times. Note that if statement 5 were:

```
5       IF (X(K).GT.Y(L)) GO TO 10
```

it would be an error.

Any statement which serves as the range limit of a DO loop must not be used as the transfer point for IF or GO TO statements which are outside the DO loop. The range of a DO statement may also include other DO statements. This is referred to as nesting. The range of any nested DO statement must fall entirely within the range of the next outermost DO statement; that is, every statement

in the range of an inner loop must be within the range of its enclosing outer loop. It is possible for a terminal statement to be the terminal statement for more than one DO loop, however. Figure 8-5 illustrates the order in which nested DO's are executed.

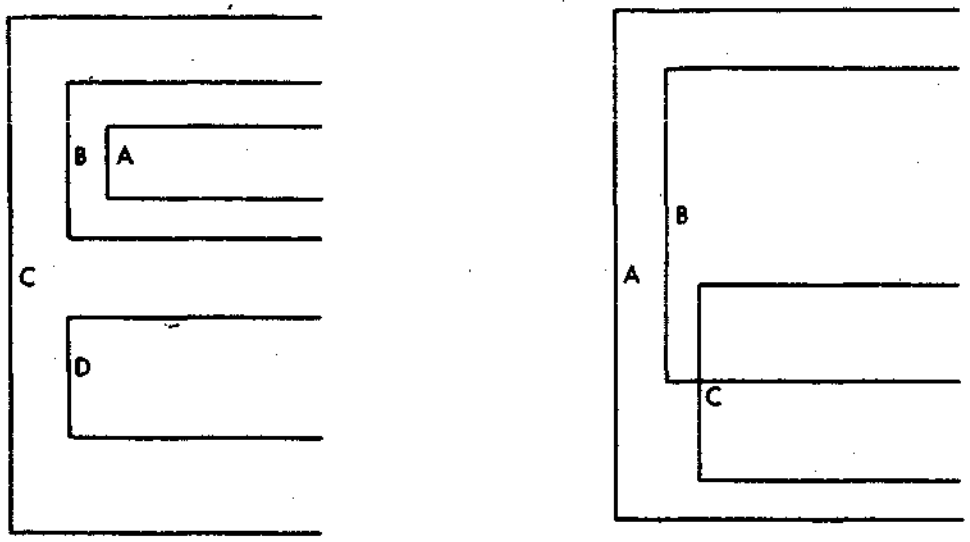


Figure 8-5. Nested DO Loops.

Do loops may be nested to a depth of (at least) ten levels. In calculating this depth, one implied DO in an I/O statement counts as one level, whose range is the single statement and n implied DO's within one I/O statement count as n levels the ranges of which are all within the single statement.

CONTINUE STATEMENT

The CONTINUE statement consists of the text:

```
CONTINUE
```

and causes continuation of the normal sequence of program execution. CONTINUE is principally used as the range limit of DO loops in which the last statement would otherwise be a GO TO, IF, PAUSE, STOP or RETURN statement. The CONTINUE statement is also used as a transfer point for IF and GO TO statements within the DO loop that are intended to begin another repetition of the loop. For example:

```
        DO 25 I=1,20
        D=D+5.0
7       IF (A-B)10,30,25
10      A=A+1.0
        B=B-2.0
        GO TO 7
25      CONTINUE
30      C=A+B
        .
        .
        .
```

A CONTINUE statement used as the range limit of any number of DO loops is compiled as an executable instruction, as in this example:

```
        DO 55 I=3,5
        .
        .
        DO 55 C=1,11
        .
        .
        DO 55 V=2,6,3
        .
        .
55      CONTINUE
        .
        .
```

A CONTINUE statement which serves as the range limit of a DO loop must not be used as the transfer point for IF or GO TO statements which are outside the DO loop. If it serves as the range limit of several DO loops, as above, it must not be used as the transfer point for IF or GO TO statements which are outside the innermost loop. For example:

```
        DO 20 I=1,50
        IF (K.EQ.4) GO TO 10      (incorrect)
        DO 10 I=1,50
        IF (K.EQ.4) GO TO 20      (correct)
        WRITE (5,100)I,J,K
10      CONTINUE
20      CONTINUE
100     FORMAT (3I6)
        END
```

PAUSE STATEMENT

Form PAUSE
PAUSE number

Where number is an integer variable or expression

Effect The number, if any, is typed on the console terminal. Execution is suspended until the user presses CONTinue on the computer console.

The PAUSE statement interrupts program execution.

STOP STATEMENT

The STOP statement is placed at the logical end of a program.

Form STOP

Effect terminates the program and returns control to the OS/8 monitor.

The STOP statement terminates program execution. No continuation is possible. If no STOP statement is present in a program, a STOP occurs when control passes to the END statement in the MAIN program.

A CALL EXIT statement is equivalent to a STOP and closes tentative files at the last block written on the file. Control returns to the OS/8 Monitor.

END STATEMENT

The END statement consists of the text:

END

and is placed at the physical end of a program or subprogram. In the main program, the END statement is equivalent to STOP; in a subroutine, END is equivalent to RETURN. The compiler assumes the presence of an END statement if it fails to find one before the end of the source input file. A program can not reference an END statement.

Data Transmission Statements

Data transmission statements control the transfer of data between computer memory and I/O devices. These include three distinct types of statement: data description (FORMAT) statement, input/

output (READ and WRITE) statements, and device control (BACKSPACE, REWIND, and END FILE) statements.

FORMAT STATEMENT

The FORMAT statement describes the form and arrangement of data on a record.

Form FORMAT (spec1,spec2,.../...)

Where spec1,spec2 define consecutive series of characters within a record.

/ is the end of the record description.

) is the end of a statement.

Effect Specifies either the type of conversion to be performed between the internal and external representation of data or the format of fixed data.

A FORMAT statement must have a statement number which is used in other statements for reference.
The field specification (spec) is one of the following:

nAw
nBw.d
nDw.d
nEw.d
nFw.d
nGw.d
nH
nIw
nLw
−sP
Tw
nX
'string'
"string"

where:

n = an unsigned non-zero integer stating the number of times the field specification is to be repeated.

s = scale factor

A,B,D,E,F,G,H,I,L,P,T,X = type of conversion

w = non-zero, unsigned integer constant specifying width of field. Field width must be large enough to provide for all characters (including decimal point, sign, and exponent) necessary to constitute the data value plus any blank characters needed to separate it from other data values. The data value within a field is right justified; if the value is too large for the field, the field is filled with asterisks.

.d = unsigned integer constant (may be zero) specifying number of digits to the right of the decimal point or, for G conversion, the number of significant digits.

Field specifications must be written in the same sequence as the data record being described, except when the T specification is used.

A FORMAT statement may describe one or more records, each of which can consist of one or more field specifications. The character "/" (slash) indicates that a new record is being described. For example, the statement:

```
FORMAT (G10.2/I5,2F8.4)
```

is equivalent to

```
FORMAT (G10.2)
```

for the first record, and:

```
FORMAT (I5,2F8.4)
```

for the second record. Field specifications are separated by commas as shown above. The separating comma may be omitted when a slash is used. When n slashes appear at the end or beginning of a format, n blank records may be written on output or n records skipped on input. When n slashes appear in the middle of a format, n-1 blank records are written or n-1 records skipped.
For example,

```
FORMAT (I6,///,2F5.1)
```

where /// indicates that two records are to be skipped

A group of field specifications may be repeated by enclosing the group in parentheses and preceding the enclosed group with a repetition number. For example,

```
FORMAT (3(I5,D10.3))
```

Both the slash and the closing parenthesis at the end of the format indicate the termination of a record. If the list of an input/output statement dictates that transmission of data is to continue after the closing parentheses of the format is reached, the format is repeated starting with that group repeat specification terminated by the last preceding right parenthesis, or, if none exists, then to the first left parenthesis of the format specification. Thus, the statement:

```
FORMAT (F7.2,3(I2,2(I3,E9.3)17))
                  ↑              ↑↑
                  group repeat   terminator
                  specification  Last preceding
                                 right parenthesis
```

causes (F7.2,3(I2,2(I3,E9.3)I7)
to be used on the first record, and the format

3(I2,2(I3,E9.3)I7)

to be used on succeeding records.
As a further example, consider the statement:

```
FORMAT (F7.2(2(E15.5,E15.4),I7))
```

The first record has the format

(F7.2)

and successive records have the format

(2(E15.5,E15.4),I7)

FORMAT statements may be placed anywhere within the executable portion of the source program. Unless the FORMAT statement contains only Hollerith data for direct input/output transmission, it will be used in conjunction with the list of a data

transmission statement. Because FORMAT statements are referenced by READ or WRITE statements, each FORMAT statement must have a statement number.

The ASCII character string comprising a format specification may be stored as an array. Input/output statements may then refer to the format by giving the array name, rather than the statement number of a FORMAT statement. The stored format has the same form as a FORMAT statement excluding the word "FORMAT". The enclosing parentheses are required.

Field specifications in a FORMAT statement should be of the same type as the corresponding items in the I/O list; that is, integer quantities require integer (I) conversion, etc. There are three types of field specifications: numeric, logical, and alphanumeric (including Hollerith). In addition, a blank field description may be given to skip portions of an input record or to embed blanks within an output record.

Numeric Fields

Numeric fields are specified by one-letter codes (B, D, E, F, G or I) which designate the type of conversion to be performed. Two parameters may appear in a numeric field description, depending on the field type. These are: an integer (w) specifying the field width (which may be greater than required to provide for blank columns between numbers) and an integer (d) specifying the number of decimal places to the right of the decimal point or, for G conversion, the number of significant digits. Decimal points are not permitted in I conversion. (For B, D, E, F and G input, the position of the decimal point, if present in the external field, takes precedence over the value of d in the format.) Conversion codes and the corresponding internal and external forms of the numbers are listed in Table 8-14. Numeric fields are right justified with the addition of leading spaces and, if necessary, trailing zeroes.

Single precision I/O specifications will transfer a maximum of six decimal digits of accurate data. Double precision I/O specifications will transfer a maximum of 15 decimal digits of accurate data.

Table 8-14 Numeric Field Codes

Conversion Code	Internal Form	External Input Form	External Output Form
D	Double precision	Decimal number with or without a decimal point or exponent field.	Decimal number with a D exponent field and a decimal point.
B	Double precision	Same as D.	Same as F.
E	Real	Decimal number with or without a decimal point or exponent field.	Decimal number and an E exponent with a decimal point field.
F	Real	Decimal number with or without a decimal point or exponent field.	Decimal number with a decimal point.
G	Real	Decimal number with or without a decimal point or exponent field.	Decimal number with a decimal point and with or without an E exponent field.
I	Integer	Decimal number without a decimal point or exponent.	Decimal number without a decimal point or exponent.

The allowable numeric field specification forms are:

1. Bw.d
2. Dw.d
3. Ew.d
4. Fw.d
5. Iw
6. Gw.d

For example:

```
FORMAT (I5,F10.2,D18.10)
```

could be used to output the line

bb32bbbb-17.60bb0.5962547681E+03

on the output listing. (The letter b represents a blank or space.) Since there is no carriage control character in the statement, the first character is interpreted as a carriage control character and is not printed.

Complex quantities are transmitted as two independent real quantities. The format specification consists of two successive real specifications or one repeated real specification. For instance, the statement

```
FORMAT (2E15.4,2(F8.3,F8.5))
```

could be used in the transmission of three complex quantities.

The G format is the general format code that is used to transmit data having a specific number of significant figures, no matter what the magnitude of the number. This format is intended to allow use of the simplest output format which can express the desired value in the space allowed. The rules for input are the same as for E format.

The form of the output conversion (E or F) is a function of the magnitude of the data being converted. Table 8-15 shows the magnitude of the internal data, M, and the resulting method of conversion.

Table 8-15 Conversion Under G Format

Magnitude of Data	Resulting Conversion
0.1<=M<1	F(w−4).d, 4X
1<=M<10	F(w−4).(d−1), 4X
.	.
.	.
.	.
10(d−2)<=M<10(d−1)	F(w−4).1, 4X
10(d−1)<=M<10(d)	F(w−4).0, 4X
All others	Ew.d

Scale factors may be written for B, D, E, F, and G conversion. A scale factor is written:

sP

where s is a signed or unsigned integer that specifies the scale factor and P is the identifying character.

For F type conversions, the scale factor specifies a power of ten, so that

$$\text{external number} = (\text{internal number}) * 10^{(\text{scale factor})}$$

For D and E conversions, the scale factor multiplies the fraction by a power of ten, but the exponent is decreased accordingly leaving the number unchanged except in form. For example, if the statement:

```
FORMAT (F8.3,E16.5)
```

corresponds to the line

bb26.451bbbb-0.41321E-01

then the statement

```
FORMAT (-1PF8.3,2PE16.5)
```

would correspond to the line

bbb2.645bbb-41.3215E-03

For G type output conversions, the scale factor is not used unless the magnitude of the number is such that E format is used. In input operations, the scale factor is not used if there is an exponent in the external field.

When no scale factor is specified, a scale factor of zero is assumed. Once a scale factor has been specified, however, it holds for all subsequent B, D, E, F, and G type conversions within the same format unless another scale factor is encountered. A zero scale factor may be resumed via an explicit specification. Scale factors have no effect on I type conversions.

Logical Fields

Logical data can be described in a manner similar to numeric data. A logical field description has the form:

Lw

where L is the conversion code character and w is an integer

specifying the field width. The data is transmitted as the value of a logical variable in the input/output list. On input, the first non-blank character in the data field must be .T., T, .F. or F; the value of the logical variable will be stored as .TRUE., .TRUE.; or .FALSE., .FALSE., respectively. Leading blanks and the period preceding the T or F are ignored. If the data field is blank, a value of false will be stored. On output, w-1 blanks followed by the letter T or F, according to the variable's value, will be transmitted. For example, if the specification were L10, the output for the value .TRUE. would be:

bbbbbbbbbT

Hollerith Data A Conversion

Hollerith data that is to be stored by the program is specified by:

Aw

where A is the conversion code character and w is the number of characters in the field. The alphanumeric characters are transmitted as the value of a variable in an input/output list. The variable may be of any type. The sequence:

```
      READ (2,5)V
5     FORMAT (A4)
```

causes four characters to be read and placed in memory as the value of the variable V.

The value of w is limited to the maximum number of characters which can be stored in the space allotted for a single variable. If w exceeds this amount, the extraneous rightmost characters are lost on input, and on output the characters after the sixth (twelfth with double precision) are not significant. If w is less than the number of characters which can be stored in the space allotted to the variable, on input the characters are left justified and blank-filled on the right of each list item. On output the leftmost w characters in the variable are transmitted to the output field.

Hollerith Data H Conversion

Hollerith data which is not changed by the program is specified by one of two forms. One, called H conversion, is:

nH

where H is the control character and n is the number of characters in the string (including blanks). For example, the format in the statement below can be used to print PROGRAM COMPLETE on the output listing.

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

Referring to this format in a READ statement causes the 17 characters to be replaced with a new string of characters from the input file.

In the second form, the Hollerith data is simply enclosed in single quotes. The output result is the same as in H conversion; on output the characters between the quotes (including blanks) are written as part of the output data, as with Hollerith constants. This form is illegal on input. A quote character within the data is represented by two successive single quotes. For example:

```
FORMAT (' PROGRAM COMPLETE ')
```

A Hollerith format field may be placed among the other fields of the format. For example, the statement:

```
FORMAT (I5,7H FORCE=F10.5)
```

can be used to output the line:

bbb22bFORCE=bb17.68901

Note that the separating comma may be omitted after a Hollerith format field. The legal characters in a Hollerith field are the 64-character graphic subset of ASCII, with the exception of "@", which must not be used.

Carriage Control

The first character of each ASCII record controls the spacing of the line printer or Teletype. This character may be established by beginning a FORMAT statement for an ASCII record with 1Hc, where c is the desired control character. The line spacing actions, listed below, occur before any printing:

Character		Effect
	blank	Advance carriage to next line.
0	zero	Skip a line (double space).
1	one	Form feed—go to top of next page; equivalent to 0 on TTY.
+	plus	Suppress skipping—overprint previous line.

For example, the program:

```
        A=14.
50      FORMAT (2F3.1)
        X=(A-5.25)/2.5
        WRITE (6,100)X
100     FORMAT (1H1,F4.1)
        STOP
        END
```

moves the line printer paper to the top of the next sheet (1H1) and prints b3.5 on the first line. If any unexpected character appears first in the FORMAT statement, it is processed as a blank.

It is often desirable to print a prompting sequence, such as a question, to which a response is to be entered on the same line. To cause such prompting, a $ is included at the end of a FORMAT statement that is associated with a WRITE statement that has been satisfied. This will inhibit the carriage return and line feed before the next input. For example:

```
        A=5
        WRITE(4,100)A
        READ(4,200)B
100     FORMAT(' SAMPLE NO.',I2,' IS:',$)
200     FORMAT(A6)
        WRITE(4,200)B
        END
```

The output is:

```
SAMPLE NO. 5 IS: RED
RED
```

Record Layout Specification

Input and output can be made to begin at any position within a FORTRAN record by use of a field description of the form:

Tw

where T is the spacing control character and w is an unsigned integer constant specifying the character position in a FORTRAN record where the transfer of data is to begin.

For printed output, w corresponds to the $(w-1)^{th}$ print position, since the first character of the output buffer is a carriage control character and is not printed. (A blank carriage control indicator is assumed.) For example:

```
FORMAT (T30,'BLACK'T50,'WHITE')
```

causes "BLACK" to appear in columns 29-33 and "WHITE" to appear in columns 49-53. The statement:

```
FORMAT (T50,'BLACK',T30'WHITE')
```

causes "BLACK" to appear in columns 49-53 of one line and "WHITE" to appear in columns 29-33 of the following line. The two lines will constitute two separate records if later read as input. On input, the statement:

```
1       FORMAT (T35,F5.0)
        READ (3,1)X
```

causes the first 34 characters of the input data to be skipped, and the next five characters to be used as the value of X. If an input record containing

ABCbbbXYZ

is read with the format specification

```
FORMAT (T7,A3,T1,A3)
```

then the characters XYZ and ABC are read, in that order. The T-code can be used in a FORMAT statement with any other type of format code.

Blanks can be introduced into an output record or characters skipped on an input record by use of the specification:

nX

where the spacing control character is X and n is the number of blanks or characters skipped and must be greater than zero. For example, the statement:

```
FORMAT (5H STEP15,10X2HY=F7.3)
```

can be used to output the line

bSTEPbbb28bbbbbbbbbbY=b-3.872

The preceding blank would not be printed on the terminal or line printer.

DEFINE FILE STATEMENT

The FORTRAN program may read or write chosen records out of a direct-access file without reading intermediate records. In the more common sequential access, the FORTRAN program must read or write each record in turn until the correct record is found. No DEFINE FILE statement is required for sequential access to mass storage files.

The DEFINE FILE statement is required so that mass storage files may be referenced as direct access files by input/output statements.

Form DEFINE FILE a1(b1,c1,U,v1),a2(b2,c2,U,v2), . .

Where a is an integer constant or variable name that is the symbolic designation for this file specified to the run-time system.

b is an integer expression that defines the number of records in the file.

c is an integer expression that defines the length (in floating point variables) of each file record. Each single precision integer or real variable or constant requires one floating point variable; double precision and complex variables or constants require two.

U is a fixed argument designating that the file is unformatted.

v is an integer variable name, called the associated variable, which is set at the conclusion of an input/output operation on the file to point to the next record.

Effect — Describes a mass storage file for use with input/output statements.

The associated variable (v) in a DEFINE FILE statement is used to maintain an index of records processed. It is set automatically after an input/output statement is executed. The statement:

```
DEFINE FILE 1(1000,100,U,IV1)
```

specifies a 1000-record file, each record of which is 100 floating point variables long. The variable IV1 will maintain an index of records processed, providing a pointer to the next record.

The symbolic file designation (a) cannot be passed as a dummy argument to a define file statement in a subroutine.

INPUT/OUTPUT STATEMENTS

The input/output statements, READ and WRITE, govern transfer of data records between internal storage and peripheral devices. Each statement can contain an input/output list naming the variables and array elements to be given values on input or whose values are to be transmitted on output. Both formatted and unformatted records can be transmitted. A formatted record requires the use of a format specification.

Input/Output Lists

An input/output list contains variable names and array elements whose values will be assigned on input or written on output. Constants are not allowed as list items. During input, the new values listed can be used in subscript or control expressions for variables appearing later in the list. For example:

```
READ (6,100)L,A(L),B(L+1)
```

reads a new value for L and uses this value in subscripts of A and B.

The transmission of array variables may be controlled by indexing similar to that used in the DO statement. This is called an implied DO and includes as a list element a parenthesized list of control variables followed by the index control. For example:

```
READ (7,10)  (X(K),K=1,4),A
```

is equivalent to:

```
READ (7,50)X(1),X(2),X(3),X(4),A
```

The indexing may be compounded by nesting the implied DO's as in the following:

```
READ (9,70)  ((MASS(K,L),K=1,4),L=1,5)
```

The above statement reads in the elements of array MASS in the following order:

MASS(1,1),MASS(2,1), . . . ,MASS(4,1), MASS(1,2), . . . ,MASS(4,5)

If an entire array is to be transmitted, the indexing may be omitted and only the array name written. The array is transmitted in order of increasing subscripts with the first subscript varying most rapidly. Thus, the example above could have been written:

```
READ (7,75) MASS
```

assuming that the array MASS is dimensioned MASS(4,5). The same statement can transmit integer and real quantities. If the data to be transmitted exceeds the items in the list, the extra data is ignored.

Input/Output Records

All data is transmitted by an input/output statement in terms of records. The maximum amount of information in one record and the manner of separation between records depends upon the medium. For punched cards, each card constitutes one record; on a terminal, a record is one line; for ASCII records, the amount of information is specified by the FORMAT reference and the I/O

list; for magnetic tape binary records, the amount of information is specified by the I/O list.

Each execution of an input or output statement initiates the transmission of a new data record. If an input/output statement requests less than a full record of information, the unrequested part of the record is lost and cannot be recovered by another input/output statement without repositioning the record. Repositioning is not possible on all devices, however. If an input/output statement requires more than one ASCII record of information, successive records are read in sequence.

The READ Statement

Form	READ (u,f) list READ (u,f)	formatted READ
	READ (u) list READ (u)	unformatted READ
	READ (a'r) list	direct access mass storage READ
Where	u is an input unit designation	
	f is a format statement reference number	
	list is an I/O list of variable names	
	a is a symbolic mass storage file number (unsigned integer constant or integer variable)	
	' designates direct access	
	r is the record number where transfer begins (integer expression)	
Effect	Input is performed according to the arguments of the READ statement.	

The unit designation(u) referred to in READ statements must be an integer in the range 1 to 9. Unit designations are assigned via a device specification command to the run-time system. If device specifications are not made, the system assumes the standard device designations given in the section on the FORTRAN IV Run-Time system. Thus, READ (1,10) could refer to input from the high-speed paper tape reader, and READ (4,11) could refer to input from the terminal. Formatted and unformatted records should not be mixed on a single unit.

A formatted READ statement causes information to be read from the specified unit and put in memory. The data are converted from external to internal form as specified by the referenced FORMAT statement. If an I/O list is provided, the data are stored as the values of the listed variables. The second form of the formatted READ statement is used if the data are transmitted directly into the specified format. For example,

```
READ (5,50)A,B,C
READ (4,100)
```

Detection of end-of-file during input causes a fatal run-time system error unless the library subroutine CHKEOF was called. CHKEOF should only be used with formatted I/O involving a single record.

When a comma is encountered on input, this signals the end of the current field and causes the next input character to be read as the first character of the next input field.

An unformatted READ statement causes binary information to be read from the unit designated and stored in memory as values of the variables in the I/O list, if any. If the record contains more words than the list requires, that part of the record is lost. If more items are in the list than are in one record, additional records are read until the list is completed.

A direct access READ statement provides random access to fixed-length records in a mass storage file. The file whose records are to be read must be defined by a DEFINE FILE statement. For example,

```
DEFINE FILE ATC(100,100,U,PT)
READ (ATC'5)ARRAY
```

The WRITE Statement

Form	WRITE (u,f) list WRITE (u,f) WRITE (u) list	formatted WRITE
	WRITE (u)	unformatted WRITE
	WRITE (a'r) list	Direct access mass storage WRITE

Where u is an output unit designation (unsigned integer constant or variable)

f is a format statement number

list is an I/O list of variable names

a is a symbolic mass storage file number

' designates direct access

r is an associated variable (record pointer)

Effect Output is performed as specified by the arguments of the WRITE statement.

The unit designation (u) referred to in WRITE statements may be an integer in the range 1 to 9. Unit designations are assigned via a device specification command to the run-time system.

A formatted WRITE statement may appear with or without an I/O list. If a list is provided, the values of the variables in the list are read from memory and written on the unit designated in ASCII form. The data is converted to external form as specified by the designated FORMAT statement. If no list is supplied, information is read directly from the specified format and written on the designated unit in ASCII form. For example:

```
        WRITE (2,100)
100     FORMAT (' OUTPUT RECORD')
```

will produce the following record on unit 2:

```
OUTPUT RECORD
```

In the case of an unformatted or direct-access WRITE, the values of the variables in the list are read from memory and written on the unit designated in binary form. A record holds 85 single precision variables. If the list elements fill more than one record, successive records are written until the list is completed. If the list elements do not fill the record, the remaining part of the record contains unknown data. Thus, if there are 100 variables in the list, two records are used; one record contains 85 variables and the

second record contains 15 variables and unknown data. For example:

```
DIMENSION X(200)
WRITE (6) X
```

will produce three records on unit 6, the first will contain X(1) to X(85); the second will contain X(86) to X(170), and the third will contain X(171) to X(200).

A direct access WRITE statement outputs a fixed-length record directly into a mass storage file. The file must have been defined previously via the execution of an appropriate DEFINE FILE statement. For example:

```
        DIMENSION RAY(10)
        DEFINE FILE 1(12,10,U,J)
        J=1
        DO 5 I=1,12
        READ (2,100) RAY
100     FORMAT (10F8.0)
        WRITE (1'J) RAY
  5     CONTINUE
        CALL EXIT
        END
```

DEVICE CONTROL STATEMENTS

There are three device control statements—BACKSPACE, END FILE, and REWIND, which apply to any file-structured device (DECtape, and disk). Their forms and effects are listed in Table 8-16.

Table 8-16 Device Control Statements

Statement	Effect
BACKSPACE u	If the next record which would be read or written on unit (u) is n, BACKSPACE repositions the unit so that the next record to be read or written will be n−1. If the first record is the next record to be read or written, the BACKSPACE statement has no effect.

Table 8-16 Device Control Statements (Cont.)

Statement	Effect
REWIND u	Repositions the designated unit (u) to the beginning of the file. If the unit is at the beginning of the file, the REWIND statement has no effect.
END FILE u	Writes an END-OF-FILE character in the specified file (u), provided that the file has been written on by a formatted WRITE. END FILE does not execute a REWIND.

Specification Statements

Specification statements may be divided into three categories: Storage specification statements, DIMENSION, COMMON and EQUIVALENCE, which give the compiler storage allocation instructions; data specification statements, DATA and BLOCK DATA, which are used to enter values; and type declaration statements, INTEGER, REAL, DOUBLE PRECISION, COMPLEX, and LOGICAL, which specify the type of variable.

STORAGE SPECIFICATION STATEMENTS

DIMENSION Statement

Form DIMENSION name1 (u1,...,u7), name2 (v1,...,v7),...

Where u1,...,u7 and v1,...,v7 are the maximum values of the subscripts they represent, up to a maximum of seven subscripts.

Effect The array name assigns the type to the array. Storage is allocated according to the dimensions given.

Each array specification gives the array name and the maximum size which each of its subscripts may assume. Array size is limited to 4096 elements. Each size specification must be a non-zero positive integer constant. For example:

```
DIMENSION A(10),B(4,6),X(5,5,5)
```

defines A as a one dimensional array variable with storage locations for 10 FPP words. In floating-point FORTRAN, each FPP

word occupies three storage locations. Therefore allocating storage locations for 10 words reserves 30 locations of core. Array B is defined as a two dimensional array with storage for 24 FPP words (4x6); 72 locations are reserved. Array X is a 3 dimensional array with 125 FPP words, reserving 375 locations.

In certain cases involving subroutines, a dimension may be an unsubscripted integer parameter.

Any number of arrays may be declared in a single DIMENSION statement. Each array variable appearing in the program must represent an element of an array declared in a DIMENSION statement, unless the dimension information is given in a COMMON or TYPE statement.

Dimension information may appear only once for a given variable. The DIMENSION statement must precede any reference to the variable including reference in a DATA or EQUIVALENCE statement. TYPE declaration or COMMON statements may appear anywhere in a program unless they include dimension information.

A subprogram can establish adjustable arrays at execution time if both the array name and the subscript size are expressed as dummy arguments in the subroutine, as in:

```
SUBROUTINE WHAT(A,X,Y,Z)
DIMENSION A(X,Y,Z)
```

To do this, the programmer must establish A, X, Y and Z as required arguments The dummy array must not exceed the dimensions of the main program array but may be smaller if the call provides lower subscript sizes than those of the main program dimensioning or if the initial array element referenced is not the beginning of the main program array.

COMMON Statement

Form	COMMON/block1/a,b,c/block2/d,e,f/...
Where	block1,block2,..., are the block names.
	a,b,c, d,e,f are the variables to be assigned to each block.
Effect	Specified variables or arrays are stored in an area available to other programs.

By means of COMMON statements, the data of a main program and/or the data of its subprograms can share a common storage area. The common area can be divided into separate blocks which are identified by block names that may not be the same as any program variable names. A block is specified as follows:

/block name/varl,var2,...

The variables which follow the block name indicate scalar or array variables assigned to the block. They are placed in the block in the order in which they appear in the block specification. For example, the statement:

```
COMMON/R/X,Y,T/C/U,V,W,Z
```

indicates that the elements X,Y, and T are to be placed in block R in that order, and that U, V, W, and Z are to be placed in block C. Variables whose names appear in the formal parameter list must not also appear in COMMON declarations within the subroutine. A COMMON block may not have the same name as a variable in the same program. Also, a COMMON block may not have the same name as any subprogram which is used at the same time as the COMMON block.

Block entries are linked sequentially throughout the program, beginning with the first COMMON statement. For example, the statements:

```
COMMON/D/ALPHA/R/A,B/C/S
COMMON/C/X,Y/R/U,V,W
```

have the same effect as the statement:

```
COMMON/D/ALPHA/R/A,B,U,V,W/C/S,X,Y
```

One block of COMMON storage, referred to as blank COMMON, can be left unlabeled. Blank COMMON is indicated by two consecutive slashes, for example

```
COMMON/R/X,Y//B,C,D
```

indicates that B, C, and D are placed in blank COMMON. The

slashes may be omitted when blank COMMON is the first block of the statement, as in:

```
COMMON B,C,D
```

Storage allocation for blocks of the same name begins at the same location for all programs executed together. For example, if a program contains:

```
COMMON A,B/R/X,Y,Z
X=3
```

as its first COMMON statement, and a subprogram has

```
COMMON/R/U,V,W/D,E,F
```

as its first COMMON statement, the quantities represented by X and U are stored in the same location, i.e., X and U both equal 3. A similar correspondence holds for A and D in blank COMMON.

COMMON blocks may be of any length, subject to the limitations on available memory.

Array names appearing in COMMON statements may have dimension information appended if the arrays have not been declared via a DIMENSION statement or a type declaration. For example:

```
COMMON ALPHA,T(15,10,5),GAMMA
```

specifies the dimensions of the array T while entering T in blank COMMON. If array dimensions are not defined in a COMMON statement, they must be defined in some other type statement.

EQUIVALENCE Statement

Form	EQUIVALENCE(v1,v2,...),(vk,vk+1,...),..
Where	v1,v2,..,vk are the variable names.
Effect	The variables within the parentheses identify the same storage location.

Example:

```
EQUIVALENCE(RED,BLUE)
```

specifies that the values of the variables RED and BLUE are stored in the same location. If used at different times, multiple variables with different values can occupy the same storage location or if used at the same time, multiple variables can be assigned the same value through the use of EQUIVALENCE.

The master variable in an equivalence group is either the variable in the group that is in COMMON (only one such variable per group is legal) or the first variable in the group. All the other variables of an equivalence group are considered slaves and can only appear in one group. The master of an equivalence group should be large enough to encompass all of the slaves equivalenced to it.

The subscripts of array variables in an EQUIVALENCE statement must be integer constants. Example:

```
EQUIVALENCE(X,A(3),Y(2,1,4)),(BETA(2,2),ALPHA)
```

The formal parameters of a subroutine must not appear in EQUIVALENCE statements within that subprogram.

The variables assigned by an EQUIVALENCE statement must be within the same main program or within the same subprogram.

EQUIVALENCE and COMMON

Variables may appear in both COMMON and EQUIVALENCE statements, but no two quantities in COMMON may be set equivalent to one another.

Quantities placed in a COMMON block by means of EQUIVALENCE statements can cause the end of the COMMON block to be extended. For example, the statements:

```
COMMON/R/X,Y,Z
DIMENSION A(4)
EQUIVALENCE(A,Y)
```

cause the COMMON block R to extend from X to A(4), arranged as follows:

```
X
Y   A(1)
Z   A(2)
    A(3)
    A(4)
```

EQUIVALENCE statements which would require extension of the start of a COMMON block are not allowed. For example, the sequence:

```
COMMON/R/X,Y,Z
DIMENSION A(4)
EQUIVALENCE(X,A(3))
```

is not permitted, since it would require A(1) and A(2) to extend the starting location of block R.

Care must be exercised when using EQUIVALENCE and COMMON statements.

THE DATA STATEMENT

Form	DATA,var list1/val list1/var list2/val list2/,...
Where	var list contains a string of variables separated by commas
	/val list/ contains a string of data items separated by commas
Effect	A value from val list is assigned to the corresponding variable in var list.

The DATA statement is used to supply initial or constant values for variables. The specified values are compiled into the object program, and become the values assumed by the variables when program execution begins. Such values may also be provided via a BLOCK DATA subprogram. Initial values for variables in COMMON may not be specified in subprograms which may be overlaid at execution time (refer to Loader description). It is recommended that variables in COMMON be initialized only by means of a BLOCK DATA subprogram.

Variables in the variable list may be either single subscripted or unsubscripted arrays, or the name of an entire array.

When an entire array is given, data values must be specified for each and every element of the array. Data elements are stored in the array in the same order as that used for the data transmission and storage arrays, i.e., in order of increasing subscripts with the first subscript varying most rapidly.

Allocation to memory locations in the array stops when:

a. the data item list is exhausted, or,
b. data items have been allocated to the entire array. If so, additional data items will be allocated to succeeding variables listed.

When Hollerith or literal constants are encountered in the values list, they are assigned to the associated variables in the same manner as in assignment statements.

A Hollerith value in a data statement will occupy (n+5)/6 words of storage (36-bit FPP words), with any partial words filled to the right with blanks.

The data items following each list of variables must have a one-to-one correspondence with the variables of the list, since each item of the data specifies the value given to its corresponding variable.

Data items assigned may be numeric, Hollerith, octal, or logical constants. For example,

```
DATA ALPHA, BETA/5,16,E-2/
```

specifies the value 5 for ALPHA and the value .16 for BETA. Any item of data may be preceded by an unsigned non-zero integer constant followed by an asterisk. This notation indicates that the item is to be repeated. For example,

```
DATA A(1),A(2),A(3),/3*0,/
```

specifies the value zero for array elements A(1)-A(3). As another example:

```
DIMENSION A(2,2),B(3)
DATA A,B/2*1,0,3*2,0,3,0,4,/
```

will initialize

A(1,1) and A(2,1) to 1
A(1,2),A(2,2) and B(1) to 2
B(2) to 3, and B(3) to 4

TYPE DECLARATION STATEMENTS

Form	type v1,v2,v3,...
Where	type may be INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL
	v1,v2,v3,... represent variables
Effect	All variables in the list are assigned the given type.

A variable can appear in only one type statement. Type declaration statements override any implicit type specifications determined by the first letter of a variable. Type statements can be used to give dimension specifications for arrays. Adjustable arrays in subprograms can also be defined via type statements. Each variable or function name in a type statement is defined to be of that specific type throughout the program; the type cannot change.
Examples:

```
INTEGER ABC,IJK,XYZ
REAL A(2,4),I,J,K
DOUBLE PRECISION B,X,DARG1,DARG2
```

Subprogram Statements

Using subprograms, a statement or group of statements can be written once and then referenced whenever the implemented operation is to be executed. The use of subprograms saves programming time and computer memory. There are three categories of subprograms in FORTRAN: FUNCTION subprograms, SUBROUTINE subprograms and BLOCK DATA subprograms. Functions and subroutines consist of one or more FORTRAN statements which may be invoked by name and, as appropriate, with values upon which they are to operate. A function differs from a subroutine in that it is always called with at least one parameter, and always returns at least one value as the value to be used for the function reference in the expression in which it occurs. A subroutine may be called with or without parameters and may return values only through its parameters or via COMMON. BLOCK DATA subprograms contain specification statements only and are used to specify initial values for variables in COMMON.

The transmission of arguments between a subprogram reference and the subprogram itself is accomplished by the use of dummy variables within the subprogram definition. Those variables in the subprogram which are dummy variables are listed in the subprogram definition statement. References to the subprogram may then supply values for these arguments in the same order and be substituted for them whenever they appear in the subprogram.

FUNCTIONS

An internal function is defined via a form of the arithmetic statement and may be referenced only by the program in which it is defined.

Form	name (argl,...)=expression
Where	name is the function name
	argl, ... are dummy arguments. These variables will be altered whenever the function is used and should not be referenced elsewhere in the program.
	expression is the function definition
Effect	Defines an internal function.

An arithmetic statement function definition is a single statement. The expression which defines the function may include dummy arguments, ordinary variables, external functions and previously defined internal functions.

In the following definition:

ACOSH(X) = (EXP(X/A) + EXP(−X/A))/2.

X is a dummy argument and A an ordinary variable. When the function is referenced, the current value of A and the supplied value of X will be used to evaluate it. All function definitions of this type must precede the first executable statement of the program in which they appear, and follow the last specification statement appearing in the program.

FUNCTION Statements

An external function, one which may be referenced by other programs, is defined via the FUNCTION statement. A function reference may only appear within an expression and must, like other elements of expressions, have a specified type. Type may be

specified in the definition itself or via any other FORTRAN type specification facility.

Form	t FUNCTION name (argl,...)
Where	t is an optional type specification (e.g. real)
	name is the function name
	argl,... are dummy arguments
Effect	Defines an external function subprogram.

The function name must be a legal symbol which is assigned a value within the subprogram definition. The last value assigned to this name is the function's value. There must be at least one argument and arguments must agree in number, order, and type with actual arguments given by the calling program. A maximum of six dummy arguments may be given in a FUNCTION statement. Example:

```
        FUNCTION M(X)
        IF (X.GE.12.0) GO TO 50
        M=10.0 * 2.5
        RETURN
50      M=.5 * 5.0
        RETURN
        END
```

Dummy arguments may represent the following elements in the function definition: expressions, alphanumeric strings, array names, or elements and subprogram names. Dummy arguments which represent array names must appear within the subprogram either in a DIMENSION statement, or in one of the type statements that provide dimension information. Dimensions given as constants must not exceed the dimensions of the corresponding arrays in the calling program. Dimensions given as dummy variables may be used to specify adjustable dimensions for array name arguments. For example, in the statement sequence:

```
FUNCTION TABLE (A,M,N,B,X,Y)
.
.
.
DIMENSION A(M,N),B(10),C(50)
```

the dimensions of array A are specified by the dummy arguments M and N, while the dimension of array B is given as a constant. The various values given for M and N by calling the program must be within the limits of the actual arrays which the dummy array A represents. Various arrays may be substituted for A. These arrays may each be of different size. Dummy dimensions may only be given for dummy arrays. Note in the above example that the array C, which is not a dummy argument, must be given absolute dimensions. A dummy argument can not appear in an EQUIVALENCE statement in the function subprogram.

A function may modify any arguments which appear in the FORTRAN arithmetic expression calling the function. The only FORTRAN statements not allowed in a function subprogram are SUBROUTINE, BLOCK DATA, and another FUNCTION statement.

A function is called by placing the function name in the program. Control then passes to the function subroutine. The quantity resulting from the function subroutine replaces the function name in the calling expression.

SUBROUTINE SUBPROGRAMS

A subroutine subprogram is defined external to the program which references it. Subroutine definition is initiated by a SUBROUTINE statement. A subroutine is referenced by a CALL statement and returns control to the calling program by means of one or more RETURN statements.

SUBROUTINE Statement

Form	SUBROUTINE name or SUBROUTINE name(argl,...)
Where	name is a subroutine name
	argl,... are optional dummy arguments
Effect	The program which follows is declared a subroutine program.

The arguments in the parenthesized list are dummy arguments representing the arguments of the subprogram. The dummy arguments must agree in number, order, and type with the actual arguments used by the calling program. A subroutine subprogram

need not have any arguments at all; a maximum of six dummy array arguments are allowed. When supplied, they may be expressions, alphanumeric strings, array names, array elements, scalar variables, and subprogram names.

Dummy variables which represent array names must be dimensioned within the subprogram by a DIMENSION or type declaration statement. As in the case of a function subprogram, either constants or dummy identifiers may be used to specify dimensions in a DIMENSION statement. The dummy arguments may not appear in an EQUIVALENCE or COMMON statement in the subroutine program.

A subroutine or function subprogram may use one or more of its dummy arguments to represent results. For example:

```
SUBROUTINE COMPUT (A,B,ANS)
```

might require the user to supply numeric values for A and B to be computed, and a variable for ANS in which to store the results. The only FORTRAN statements not allowed in a subroutine subprogram are FUNCTION, BLOCK DATA, and another SUBROUTINE statement.

Constants in call lists of subroutines and function subprograms are not protected. Therefore, a function such as the following will result in erroneous values.

```
FUNCTION FNC1 (Y)
FNC1=100.0
Y=Y+2.0
RETURN
END
```

CALL Statement

Form	CALL name or CALL name (arg1,...)
Where	name identifies a subprogram arg1,... are actual arguments
Effect	Control is transferred to the subroutine subprogram.

The arguments of a CALL statement can be expressions, array names, array elements, scalar variables, alphanumeric strings, or

subprogram names; arguments may be of any type, but must agree in number, order, type and array size (except for adjustable arrays, as discussed under the DIMENSION statement) with the corresponding arguments in the SUBROUTINE statement of the called subroutine. A subroutine cannot be referred to as a basic element in an expression.

RETURN STATEMENT

The RETURN statement consists of the text:

```
RETURN
```

This statement returns control from a subprogram (subroutine or function) to the calling program. Normally, the last statement executed in a subprogram is a RETURN statement. Any number of RETURN statements can appear in a subprogram.

BLOCK DATA STATEMENT

The BLOCK DATA statement is used to establish a block data subprogram, a data specification subprogram which is used to enter initial values for variables in common blocks. No executable statements can appear in a block data subprogram. A block data subprogram is established by a BLOCK DATA statement consisting of the text:

```
BLOCK DATA
```

This statement declares the program which follows to be a data specification subprogram and must be the first statement of the subprogram. For example:

```
BLOCK DATA
COMMON A,B,C
COMMON/X/ARRAY(100)
INTEGER A,C
REAL B
DATA A,B,C/5,1.5,0/
LOGICAL ARRAY/100*0/
END
```

This subprogram causes blank common to be initialized with the integer 5, a real variable 1.5, an integer 0 as its first three variables and an array with 100 zeroes.

The subprogram contains only type statements, EQUIVALENCE, DATA, DIMENSION, and COMMON arguments. A complete set of specifications must be given for an entire common block. A single block data subprogram can initialize any number of named COMMON blocks.

EXTERNAL STATEMENT

Form	EXTERNAL identifier,identifier,....,identifier
Where	identifier is the name of a subprogram
Effect	The identifier is declared a subprogram name and may be used as the argument of other subprograms.

Function and subroutine subprogram names can be used as the actual arguments of subprograms. When they are, their names must be distinguished from ordinary variables by their appearance in an EXTERNAL statement.

Any subprogram name given as an argument to another subprogram must have previously appeared in an external declaration in the calling program (i.e., as an identifier in an EXTERNAL).
Example:

```
EXTERNAL SIGMA,THETA
.
.
CALL TRIGF(SIGMA,1.5,ANSWER)
.
.
CALL TRIGF(THETA,187,ANSWER)
.
.
END

SUBROUTINE TRIGF(FUNC,ARG,ANSWER)
.
.
ANSWER=FUNC(ARG)
.
.
RETURN
END
```

Table 8-17 FORTRAN IV Statement Summary

Statement	Form	Effect
Arithmetic	a = b	The value of expression b is assigned to the variable a.
Arithmetic statement function definition	t name(a1...)=x	The value of expression x is assigned to f(a1...) after parameter substitution.
ASSIGN	ASSIGN n TO v	Statement number n is assigned as the value of integer variable v for use in an assigned GO TO statement.
BACKSPACE	BACKSPACE u	Peripheral device u is backspaced one record.
BLOCK DATA	BLOCK DATA	Identifies a block data subprogram.
CALL	CALL prog CALL prog(a1...)	Invokes subroutine named prog, supplying arguments when required.
COMMON	COMMON/block1/a,b,c/...	Variables (a,b,c) are assigned to a common block.
CONTINUE	CONTINUE	No processing, target for transfers.
DATA	DATA var list1/va1 list1/...	Assigns initial or constant values to variables.
DEFINE FILE	DEFINE FILE al(b1,c1,U,v1)...	Describes a mass storage file for direct access I/O.
DIMENSION	DIMENSION array (v1...,v7)...	Storage allocated according to dimensions specified for the array.

Table 8-17 FORTRAN IV Statement Summary (Cont.)

Statement	Form	Effect
DO	DO n i = m1,m2,m3	Statements following the DO up to statement n are iterated for values of integer variable i, starting at i =ml, incrementing by m3, terminating when i>m2.
END	END	Cease program compilation; equivalent to STOP in main program or RETURN in subprogram.
END FILE	END FILE u	Writes END-OF-FILE character in file u.
EQUIVALENCE	EQUIVALENCE (v1,v2,...,),	Identifies same storage location for variables within parentheses.
EXTERNAL	EXTERNAL subprog,...	Declares a subprogram for use by other subprograms.
FORMAT	FORMAT (spec1,spec2,.../...)	Specifies conversions between internal and external representation of data.
FUNCTION	FUNCTION name (a1...)	Indicates an external function definition.
GO TO		Transfers control to:
	(1) GO TO n	(1) statement n
	(2) GO TO (n1,...,nk),e	(2) to statement n1 if e = 1, to statement nk if e=k.

Table 8-17 FORTRAN IV Statement Summary (Cont.)

Statement	Form	Effect
	(3) GO TO v GO TO v(n1,...,nk) GO TO v,(n1,...,nk)	(3) Transfers control to statement number assigned to v optionally checking that v is assigned one of the labels n1 ....nk.
IF	IF (arith expr)n1,n2,n3	Transfers control to n1 if expr <0, n2 if expr = 0, n3 if expr >0.
IF	IF (logical expr) statement	Executes statement if expression has value .TRUE., otherwise executes next statement in sequence.
Logical	V=E	Value of expression E is assigned to variable V.
PAUSE	PAUSE PAUSE number	Program execution interrupted and number printed, if given.
READ	READ (u,f) list READ (u,f) READ (u) list READ (u) READ (a'r) list	Reads a record from a peripheral device according to specifications given in the arguments of the statement.
RETURN	RETURN	Returns control from a subprogram to the calling program.
REWIND	REWIND u	Repositions designated unit to the beginning of the file.
STOP	STOP	Terminate program execution.

Table 8-17 FORTRAN IV Statement Summary (Cont.)

Statement	Form	Effect
SUBROUTINE	SUBROUTINE name(a1,...)	Declares name to be a subroutine subprogram and a1,..., if supplied, as dummy arguments.
type	type v1,v2,v3,...	Where the variables vn are assigned the indicated type, i.e., Real, Integer, etc.
WRITE	WRITE (u,f) list WRITE (u,f) WRITE (u) list WRITE (u) WRITE (a'r) list	Writes a record to a peripheral device according to specifications given in the arguments of the statement.

PAPER TAPE LOADING INSTRUCTIONS

The FORTRAN IV system may be loaded from paper tape using OS/8 EPIC. Of the nine files that make up the system, the following eight:

F4.SV
PASS2.SV
PASS20.SV
PASS3.SV
RALF.SV
LOAD.SV
FRTS.SV
LIBRA.SV

are on separate paper tapes, as indicated, and may be read in any order. After these tapes have been read, the six tapes that comprise the library (FORLIB.RL) must be read in ascending numerical order. A typical procedure might be:

```
.R EPIC
```

Load OS/8 EPIC.

```
*SYS:<
```

Designate the device on which the new FORTRAN IV system will be built and mount the F4.SV tape in the reader.

```
*/Y
*/Y
*/Y
*/Y
*/Y
*/Y
*/Y
END OF TAPE ENTER NEXT
END OF TAPE ENTER NEXT
END OF TAPE ENTER NEXT
END OF TAPE ENTER NEXT
END OF TAPE ENTER NEXT
*↑C
```

Mount the PASS2.SV tape in the reader.
Mount the PASS20.SV tape in the reader.
Mount the PASS3.SV tape in the reader.
Mount the RALF.SV tape in the reader.
Mount the LOAD.SV tape in the reader.
Mount the FRTS.SV tape in the reader.
Mount the LIBRA.SV tape in the reader.
Mount the first FORLIB.RL tape in the reader.
Continue to read the six FORLIB.RL paper tapes in increasing numerical order.

PDP-12 users who create OS/8 FORTRAN IV systems from paper tape and require the real-time capabilities of this system must assemble the RALF modules containing REALTM, ADB, ADC, PLOT, CLRPLT, and SCALE, then add these modules to the system library. The routines to be assembled and inserted are contained on three paper tapes. A typical procedure might be as follows.

```
.ASSIGN SYS DEV
.R PIP
*DEV:FILE1.RA<PTR:
↑
*DEV:FILE2.RA<PTR:
↑
*DEV:FILE3.RA<PTR:
↑
*↑C

.R RALF
*DEV:FILE3.RL<DEV:FILE3.R

.R RALF
*DEV:FILE2.RL<DEV:FILE2.RA

.R RALF
*DEV:FILE1.RL<DEV:FILE1.RA
```

Use OS/8 PIP to read the RALF modules, in ascending numerical order, onto temporary files.

Assemble the temporary files under RALF.

```
.R LIBRA
*DEV:TEMPLB.RL<DEV:FORLIB.RL=13
*↑C
```

Use LIBRA to create a temporary library containing 13 extra blocks for expansion.

```
.R PIP
*DEV:FORLIB.RL</D
*↑C
```

Use PIP to delete the old library.

```
.R LIBRA
*DEV:FORLIB.RL[3]<DEV:TEMPLB.RL,DEV:FILE1.RL,DEV:FILE2.RL,DEV:FILE3.RL/R
```

NOTE

Use LIBRA to merge the temporary library and the new RALF modules. LIBRA will print a list of duplicate module names on the console terminal. When all output has ended and LIBRA prints an asterisk at the left margin, type CTRL/C to return to the monitor.

The following program example provides a simple test to verify that the OS/8 FORTRAN IV system is operating correctly. When duplicating the illustrated procedure, use care to distinguish between user input and machine output. Type CTRL/L at the point marked (1) to return the editor to command mode. Type ALT MODE (ESCAPE on some terminals) at the point marked (2) to generate the dollar sign character ($).

```
.R EDIT
*EXAMPL<

#A
        WRITE (4,100)
100     FORMAT (1H "THIS IS A SIMPLE EXAMPLE OF A PROGRAM")
        X=2.5
        Y=3.14
        Z=X*Y
        WRITE (4,200) X,Y,Z
200     FORMAT (1H "X="F4.2," Y="F4.2," X*Y="F4.2)
        END


#L                                                          (1)
        WRITE (4,100)
100     FORMAT (1H "THIS IS A SIMPLE EXAMPLE OF A PROGRAM")
        X=2.5
        Y=3.14
        Z=X*Y
        WRITE (4,200) X,Y,Z
200     FORMAT (1H "X="F4.2," Y="F4.2," X*Y="F4.2)
        END


#E
```

```
.R F4
*,TTY:,TTY:<EXAMPL/G
  OS/8  FORTRAN IV  3.02              JUN  1  1973          PAGE  ONE

0002              WRITE (4,100)
0003     100      FORMAT (1H "THIS IS A SIMPLE EXAMPLE OF A PROGRAM")
0004              X=2.5
0005              Y=3.14
0006              Z=X*Y
0007              WRITE (4,200) X,Y,Z
0010     200      FORMAT (1H "X="F4.2," Y="F4.2," X*Y="F4.2)
0011              END

        LOADER V21  06 /01 /73

SYMBOL  VALUE LVL OVLY

ARGERR 00204  0  00
EXIT   00223  0  00
#MAIN  10000  0  00
 10400  = 1ST FREE LOCATION


LVL OVLY LENGTH

 0  00   10270


*$
THIS IS A SIMPLE EXAMPLE OF A PROGRAM                    (2)
X=2.50 Y=3.14 X*Y=7.85
```

FORTRAN IV PLOTTER ROUTINES

The X,Y plotter routines control an incremental plotter (Calcomp 563, 565 or similar) for use with OS/8 FORTRAN IV. The routines permit the user to generate a wide variety of plotted information, including:

1. Labelled axes,
2. Textual data,
3. Graphs from data arrays (X and Y), with optional scaling of either array and centered symbols denoting the location of a data point,
4. Variables from the FORTRAN IV program plotted in F format,
5. Individual point and vector plotting.

The user also has control of:

1. Pen position (up or down),
2. Origin of plotted information,
3. Scaling of any plot,
4. Rotation of text and axes.

The routines included are:

Table 8-18 FORTRAN IV Plotter Routines

Name	Function
PLOTS	Initializes all other plotter routines to the user's hardware configuration.
XYPLOT	Moves pen to specified X,Y location with pen in up or down position, permits origin control.
FACTOR	Scales size of subsequent plotting data.
WHERE	Passes current position and factor to the user program.
SYMBOL	Prints textual information (such as titles) at any angle and special symbols to indicate a data point.
NUMBER	Prints each digit in a variable, including optional decimal point and truncation.
PSCALE	Defines parameters for axis annotation and size of final plot for data array.
AXIS	Plots an axis, at any angle, including segment markings and title.
LINE	Generates the graph of data in two arrays (X and Y).
PLEXIT	Terminates all plotting.

The system must support any OS/8 FORTRAN IV configuration plus: XY/8e interface for PDP-8/E, or XY interface for

PDP-12, 8, or 8/I and an incremental plotter suitable for one of the above interfaces.

The system must have OS/8 (QFS8-A) and OS/8 FORTRAN IV (QF008-AB).

Plotter Operation

To optimize the use of the X,Y plot routines, a brief description of the plotter operation is presented here. The plotter permits six basic functions: drum down (+X movement), drum up (−X movement), pen left (+Y movement), pen right (−Y movement), pen up and pen down. Diagonal movement is accomplished by a combination of pen and drum motion. The plotting increment is a function of the plotter itself, generally .005 or .01 inches. Each line plotted is in this incremental unit. Hence upon very close examination vectors plotted at angles other than multiples of 45 degrees may appear sightly non-linear. This effect is unnoticeable at normal viewing distances from the plotter where all vectors appear smooth. If the user requests a vector that exceeds the physical width of the plotter, the pen will move to the physical limit and plot the remaining section at the margin. This may distort subsequent plotting, depending on the user's sequence of commands. Therefore, be sure the pen is either physically located in a useful position at the start of the plot or use the plotting commands to monitor its position to prevent such problems.

Plotter Commands

PLOTS

The routine PLOTS must be called once at the start of each plotting program to initialize internal parameters to the current configuration. The call is:

```
CALL PLOTS(X,Y)
```

where:

X is the increment size of the plotter in inches; generally .01 or .005 inches.

Y is 0 if running on a PDP-8/E;
1 if running on a PDP-8/I, PDP-8, or PDP-12.

PLOTS initializes the factor (for overall plot size) to one and clears old pen location and origin status. Note that although the

plotter may actually move in inches, the code can cause it to behave as if it were millimeters (or any other unit) by including the proper conversion in the FORTRAN code.

XYPLOT

XYPLOT is the routine that actually causes pen and drum movements on the plotter. Routines such as NUMBER and AXIS eventually use XYPLOT. This routine is useful when a plot is to be generated one vector at a time by the user program (rather than saving an array, for example). It also controls the origin, defined as the logical point (0,0) for future plotting.

The call is of the form:

```
CALL XYPLOT (X,Y,I)
```

where:

X,Y is the X,Y coordinate in inches to which the pen is to move relative to the most recently established origin point.

I is an integer of the set (−3,−2,2,3) which controls pen position and establishes the origin point, as follows:

If I= 2, the pen is down during the move.
If I= 3, the pen is up during the move.
If I is negative, the pen moves to point X,Y and this point is then established as the current origin point (0,0).
If a value outside this set is called, the pen defaults to down.

For example:

```
CALL XYPLOT(4,−2,−2)
```

moves from the current position to 4,—2 with the pen down and establishes this location as the origin point (0,0).

```
CALL XYPLOT(−7,3,3)
```

moves the pen in the up position to −7,3. If these two commands are sequential, then this move would be −7 inches of X and +3 of Y, from 0,0 to −7,3.

No single vector can be plotted longer than 4095 plotting increments, or approximately 40.9 inches for a .01 increment plotter or 20.4 for a .005 increment plotter.

FACTOR

Overall plot size can be increased or reduced by using the FACTOR routine. The call is:

CALL FACTOR(Z)

where:

Z is the ratio of the desired plot size to the current size. This value is initialized by PLOTS to 1. Calling FACTOR with Z=1 resets the plot to its initial size. The absolute value of Z is used. For example, to double the size of the plot, the call is CALL FACTOR (2); to halve it, is CALL FACTOR (.5).

WHERE

The WHERE routine passes three values to the user program: current X position, current Y position and current factor. This routine is most commonly used to determine the current location of the pen in a long plotting sequence, or to calculate a delta X or Y value for the next step in a graph.
The call is:

CALL WHERE (X,Y,Z)

where:

X is set to the current X position

Y is set to the current Y position

Z is set to the current factor

Consider the following example:

```
      CALL PLOTS(,01,1)
      CALL XYPLOT(0,0,-3)
      CALL XYPLOT(-5,3,2)
      CALL WHERE(A,B,C)
      WRITE(4,10)A,B
10    FORMAT(1X,'XVAL=',I3,'YVAL=',I3)
      CALL PLEXIT
      END
```

At the completed running of this program the statement XVAL= −5YVAL= 3 will be printed on device 4.

SYMBOL

The SYMBOL routine has two forms:

1. Print any number of letters and symbols
2. Print a single character

The available character set for both forms is found in Tables 8-19 and 8-20.

Table 8-19 Special Symbols

SYMBOL	CODE	SYMBOL	CODE	SYMBOL	CODE
	100		106		112
	101		107		113
	102		108		114
	103		109		115
	104		110		116
	105		111		117

Table 8-20 Regular Characters

SYMBOL	CODE	SYMBOL	CODE	SYMBOL	CODE
A	1	V	22	+	43
B	2	W	23	,	44
C	3	X	24	-	45
D	4	Y	25	.	46
E	5	Z	26	/	47
F	6	[	27	0	48
G	7	\	28	1	49
H	8	]	29	2	50
I	9	↑	30	3	51
J	10	←	31	4	52
K	11		32	5	53
L	12	!	33	6	54
M	13	"	34	7	55
N	14	#	35	8	56
O	15	$	36	9	57
P	16	%	37	:	58
Q	17	π	38	;	59
R	18	'	39	<	60
S	19	(	40	=	61
T	20	)	41	>	62
U	21	*	42	?	63

Multiple Characters

Any of the above characters, except pi, in Table 8-20 can be combined in any order to print titles, legends, labels or the like using a multiple character call:

CALL SYMBOL (X,Y,H,T,A,N)

where:

X,Y is the coordinate in inches of the lower left corner of the first character to be printed.

H is the height in inches of each character. Because characters are considered to be on a 7x7 grid, a multiple of 7 times the increment size is recommended (i.e., a minimum of .07 for .01 increment plotters and .035 for .005 increment plotters). The actual plotting grid occupied by any character is 6x4, the remaining 1x3 being used for spacing between characters.

T is the text in A or Hollerith format.

A is the angle at which the text is to be printed and is specified in degrees from the X axis.

N is the number (positive integer) of characters to be plotted and must be greater than 0 and equal to or less than the number of characters in T.

For example:

```
DIMENSION TEXT(2)
DATA TEXT/*TEXT EXAMPLE*/
CALL PLOTS(,01,1)
CALL XYPLOT(0,0,-3)
CALL SYMBOL(1,1,.21,TEXT,0,12)
CALL PLEXIT
END
```

will, on a non-PDP8/e machine with a .01 increment plotter, initialize the origin point at the current pen location, move from there to 1,1 and print the 12 characters in TEXT, namely TEXT EXAMPLE, in letters .21 inches high at 0 degrees from the X axis, i.e., parallel to the side of the plotter.

The program above is equivalent to:

```
CALL PLOTS(,01,1)
CALL XYPLOT(0,0,-3)
CALL SYMBOL(1,1,.21,12HTEXT EXAMPLE,0,12)
CALL PLEXIT
END
```

Note that the character pi can only be plotted by a single character command because it has no Hollerith representation.

Single Characters

Two types of single characters can be plotted:

1. Characters from the available character set listed in Table 8-20.
2. Special symbols used to denote a data point. The available special symbols are listed in Table 8-19. Their use differs from other characters in that their starting and terminating point is the center of the character, not the lower left corner. These symbols occupy a 4x4 grid.

The call is:

CALL SYMBOL (X,Y,H,I,A,N)

where:

X,Y is the X,Y coordinate of the lower left corner of a regular character, including pi, or the center for a special symbol.

H is the height in inches of the symbol and should be 7 times the increment size for a regular character and 4 times the increment size for a special symbol (i.e., .02 or .04 minimum depending on the plotter).

I is in the range 1-63 for regular characters (Table 8-20) and 100-117 for special symbols (Table 8-19). If a non-acceptable value is used, SYMBOL prints a space in its place.

A is the angle in degrees from the X axis at which the character is printed.

N is −1 if the pen is to be up during the move to X,Y or −2 if the pen is to be down during the move to X,Y.

For example:

```
CALL PLOTS(,01,1)
CALL XYPLOT(0,0,-3)
CALL SYMBOL(-6,2,,35,1,180,-1)
CALL PLEXIT
END
```

This will plot the letter A .35" tall at 180 degrees to the X axis on a PDP-8/E. The pen will be up during the move from 0,0 to −6,2, the lower left corner of the A.

```
CALL PLOTS(.01,1)
CALL SYMBOL(1,4,.20,100,270,-2)
CALL PLEXIT
END
```

This will plot the first special character .2 inches tall centered at point 1,4 at an angle of 270 degrees to the X axis on a non-PDP-8/E. The move of the pen from its current location to the start of the character (1,4) will be visible.

NUMBER

To facilitate handling internal format data (floating point) the NUMBER routine is included. It plots floating point numbers in a format similar to FORTRAN IV F format. One number at a time is plotted using the call:

CALL NUMBER (X,Y,H,Z,A,N)

where:

X,Y is the coordinate of the lower left corner of the first character of the number.

H is the height of each character, preferably 7 times increment size (each number is considered to occupy a 7x7 grid).

Z is the number to be plotted. It may be a real or integer number.

A is the angle to the X axis at which to plot the number.

N is an integer that controls the format of the number Z as follows:

Value of N	Result
0	Z is truncated and plotted as an integer followed by a decimal point
−1	Z is truncated and plotted as an integer

Value of N	Result
=>1	N digits to the right of the decimal point are plotted. The number is rounded based on the value of the $(N+1)^{th}$ digit.
<−1	N−1 digits are truncated from the integer portion of the number.

Note that the accuracy of the number printed cannot exceed 6 digits, however at the user's discretion he may plot up to 19 digits with an expected loss of accuracy. If a bad digit is found in Z, that digit defaults to 0. For Z less than one a leading zero is included. For example:

```
CALL PLOTS(,005,1)
C=0
A=198,678
CALL XYPLOT(0,0,-3)
CALL NUMBER(1,1,.07,A,C,0)
CALL NUMBER(1,2,.07,A,C,-1)
CALL NUMBER(1,3,.14,A,C,-2)
CALL NUMBER(1,4,.14,A,C,2)
CALL PLEXIT
END
```

Statistically the above program will be plotted as follows:

Starting Location	Height (Inches)	Number Plotted	Angle
1,1	.07 * 6/7	198.	0
1,2	.07 * 6/7	198	0
1,3	.14 * 6/7	19	0
1,4	.14 * 6/7	198.68	0

If the number (Z) is out of range of the acceptable number of characters including minus sign and decimal point, the message:

```
NUMBER OF DIGITS NOT 1-19
```

is printed on the console device (unit 0).

PSCALE

For many applications, the data to be plotted is scattered irregularly across the total range and in a manner not neatly related to unit (inch) increments. To permit plotting data in a finite (user specified) length graph with labelled axis, the PSCALE routine is invoked to establish two critical plotting parameters—starting value and scaling increment. The starting value can be positive or negative and a maximum or minimum. It is the value printed at the starting axis annotation. The scaling increment is the delta value between succeeding axis annotations and is the number of data units per inch of plot, adjusted to 1,2,4,5 or 8 * 10↑N. These two values are used by the AXIS and LINE routines to produce a properly annotated axis and a graph whose data includes all points in a user specified length. PSCALE does no plotting; its use is in conjunction with AXIS and/or LINE. It is generally called twice—once for X (abscissa) values and once for Y (ordinate) values.
The call is:

```
CALL PSCALE(A,L,N,I)
```

where:

A is the array containing the data to be plotted. This array must have extra locations at the end in which PSCALE can store the starting value and scaling increment, as explained below.

L is the length (integer) of the axis that the data is to cover. L must be greater than or equal to 1.

N is the number of data values in A to be considered. N must be greater than or equal to 1.

I is the increment between data values to be considered. The first value examined is always A(1), the next is A(1+I). If I is positive, the calculated starting value will be a minimum value.
If I is negative, the calculated starting value will be a maximum and the scaling increment will be negative.

The calculated starting value is stored at A(N*J+1), the scaling increment is stored at A(N*J+J+1) where J is the absolute value

of I. Be sure to dimension A to a length sufficient to include these locations. Consider the data array ARRAY:

Element	Contents
1	.5
2	1
3	.9
4	.9
5	3.4
6	3.2
7	3.9
8	4.5
9	5.2
10	5.9

The statement:

```
CALL PSCALE (ARRAY,5,5,2)
```

PSCALE will use ARRAY(1), ARRAY(3), ARRAY(5), ARRAY(7), and ARRAY(9) in determining the starting value and scaling increment. For the example above, the scaling increment is 2.0 and the starting value is 0.

If an axis length of less than one is supplied, the message:

```
AXIS LENGTH <1
```

is printed on device 0. If all elements of the data array are the same, the message:

```
MAX PT = MIN PT
```

is printed.

AXIS

For most graphs, the presence of labelled axes adds significantly to interpreting the data. The AXIS routine draws an axis with labelled tic marks at one inch intervals and a title or other annotation parallel and centered to the axis. AXIS must be called separately for an X and a Y axis. The starting value and scaling

increment discussed in PSCALE must have been determined previously to calling AXIS.

The call is:

CALL AXIS (X,Y,T,N,L,A,F,D)

where:

X,Y are the coordinates of the start of the axis, in inches, relative to the current origin. Often when two axes are required, X and Y are 0 for both calls. It is suggested that the physical origin of the axis be at least ½" in from any edge of the plotter, as annotation will require that space. This position becomes the new origin for subsequent plotting.

T is the title in Hollerith format. It is printed .14 inches tall (dependent on existing user specified plotting factors) and centered along the axis. If the scaling increment is greater than 99 or less than .01, the notation *10 is added at the end of the title.

N is the number of characters in the title (T) to be printed. Its sign is used to specify on which side of the axis the tic marks and their labels are to be: positive means the positive (counter clockwise) side of the axis, negative is the negative or clockwise side. Positive labeling is generally used for Y axes and negative for X axes.

L is the length of the axis in inches. Note that this value should not exceed the width of the plotter for an axis in that direction. The absolute value of L is used.

A is the angle in degrees at which the axis is to be drawn. X axes are generally at 0 and Y axes at 90.

F is the starting value and will be used as the annotation for the first tic mark. The annotations include two significant places after the decimal point. This value may be determined by PSCALE or supplied by the user. If calculated by PSCALE, F must be the appropriate array element. If the user chooses to calculate his own starting value and scaling increment, be aware that a tiny F and large D or large F and tiny D do not produce a meaningful graph.

D is the scaling increment between tic mark annotations. It may be determined by PSCALE or by the user. If calculated by PSCALE, D must be the appropriate array element.

For best results axes should be drawn at multiples of 45 degrees (including 0). AXIS uses the routine NUMBER.

LINE

Pairs of data points in two arrays can be combined by LINE and plotted according to user specified parameters. Points to be plotted can be indicated by a special symbol and can also be connected by a continuous line. LINE requires a starting value and scaling increment for each array such as those produced by PSCALE.
The call to LINE is:

```
CALL LINE(A,B,N,I,L,J)
```

where:

A is the name of the array whose values are to be the abscissa values.

B is the name of the array whose values are to be the ordinate values.
For A and B, the $(N*I+1)^{th}$ element must contain its starting value and the $(N*I+I+1)^{th}$ element must contain its scaling increment, as supplied by the user or PSCALE.

N is the number of points in each array to be plotted. The same number of points is taken from each array.

I is the increment at which the data in A and B is collected, i.e., every I^{th} point is plotted. I must be greater than 0.

L determines the manner in which the line is plotted, as follows:
If L is positive, each point is connected by a line and a special symbol is plotted at each point.
If L is 0, each point is connected by a line, and no symbols are drawn.
If L is negative, no connecting lines are plotted. Each point is indicated by a special symbol.

J is a value between 100 and 117 according to Table 8-19 indicating the special symbol to be used in the plot.

The pen should be located at the logical 0,0 position of the graph when a call to LINE is issued. If the preceding plot operation was drawing an axis in the usual manner, the pen should be properly positioned. If I or N is less than or equal to 0, the LINE routine returns without plotting.

PLEXIT

In order to permit the plotting routines to finish completely, the routine PLEXIT must be called once when all plotting commands have been issued. PLEXIT does a final pen up operation.

Implementing the Plotter Routines

GETTING STARTED

In order for the plotter to interface properly to OS/8 FORTRAN IV, the following patch must be made to the file FRTS.SV. It adds a clear plotter flag IOT to the run-time device initialization chain. The sequence is:

```
.GET SYS:FRTS.SV
.ODT
4020/7000 6502          /User types 4020 / Response
                        /at terminal is 7000.
                        /User types 6502
↑C                      /Type CTRL/C to exit ODT
.SA SYS:FRTS.SV         /Assumes FRTS.SV on SYS
                        /Device
```

ADDING THE PLOTTING ROUTINES

The FORTRAN plotting routines are supplied as relocatable RALF (.RL) modules that can either be added to the FORTRAN library (FORLIB.RL) or specified explicitly to the loader. To add the files to FORLIB.RL, the procedure is:

```
.R LIBRA
*PLOTLB[3]<FORLIB.RL/Z=40
*PLOTLB<XYPLOT.RL,AXIS.RL,PSCALE.RL,LINE.RL,NUMBER.RL
*↑C
```

PLOTLB may then be used by specifying it as a library to the loader or it may be copied using PIP so that no additional loader specifications are required. If you choose not to add the plotting modules to the library, and prefer to specify them to the loader, it is suggested that only the modules required by the FORTRAN program be specified in order not to waste space. In general the user employing elaborate overlay schemes will not want these in his library, while the user with shorter programs will.

The core requirements to the nearest hundredth location of the files are:

XYPLOT	1000 locations in field 1 and 700 elsewhere (includes FACTOR,PLOTS, WHERE, and PLEXIT).
SYMBOL	500
symbol table	700 (regular and special characters)
NUMBER	1300
PSCALE	1000
AXIS	1500 (requires NUMBER)
LINE	600

Note that the routines PLOTS, XYPLOT, FACTOR, WHERE, PLEXIT, SYMBOL and the symbol table, including the code in field one, are all loaded if any one of those routines is called.

Loading the Plotter Routines from Papertape

If the relocatable plotter routines are supplied on paper tape, they must be loaded into mass storage using the program EPIC. Place each tape in the reader before typing the response to the asterisk. The sequence is:

```
.R EPIC
*/0$                    /Mount XYPLOT.RL
*/Y                     /Mount NUMBER.RL
*/Y                     /Mount AXIS.RL
*/Y                     /Mount PSCALE.RL
*/Y                     /Mount LINE.RL
*↑C
```

The above puts the files on device SYS.

Examples

An example combining several of the commands is shown below. This program requests user input of text and then plots it as a spiral.

```
      DIMENSION NAME(30)
      ITTY=4
      WRITE(ITTY,100)
100   FORMAT(1X,'TYPE IN TEXT(30 CHARACTER MAX)')
      READ(ITTY,200)NAME
200   FORMAT(30A1)
      WRITE(ITTY,150)
150   FORMAT(1X,'HOW MANY CHARACTERS DID YOU TYPE IN?')
      READ(ITTY,250)NN
250   FORMAT(I2)
      CALL PLOTS(,01,1)
      CALL XYPLOT(0,-30,-3)
      CALL XYPLOT(10,10,-3)
      RAD=3
      SIZE=.1225*RAD
      SPIR=.995
      CONV=180./3.1415
      ANG=0
      BANG=1.5707
      DO 300 J=1,NN
300   CALL SYMBOL((J-1-NN)*SIZE,RAD,SIZE,NAME(J),ANG,1)
380   DO 400 J=1,NN
      T=2*ATAN(SIZE/(2.*RAD))
      ANG=ANG-T*CONV
      X=RAD*COS(BANG)
      Y=RAD*SIN(BANG)
      BANG=BANG-T
      RAD=RAD*SPIR
      SIZE=.1225*RAD
400   CALL SYMBOL(X,Y,SIZE,NAME(J),ANG,1)
      IF(SIZE-.07)500,500,380
500   CALL PLEXIT
      END
```

The actual plotter output is shown in Figure 8-6.

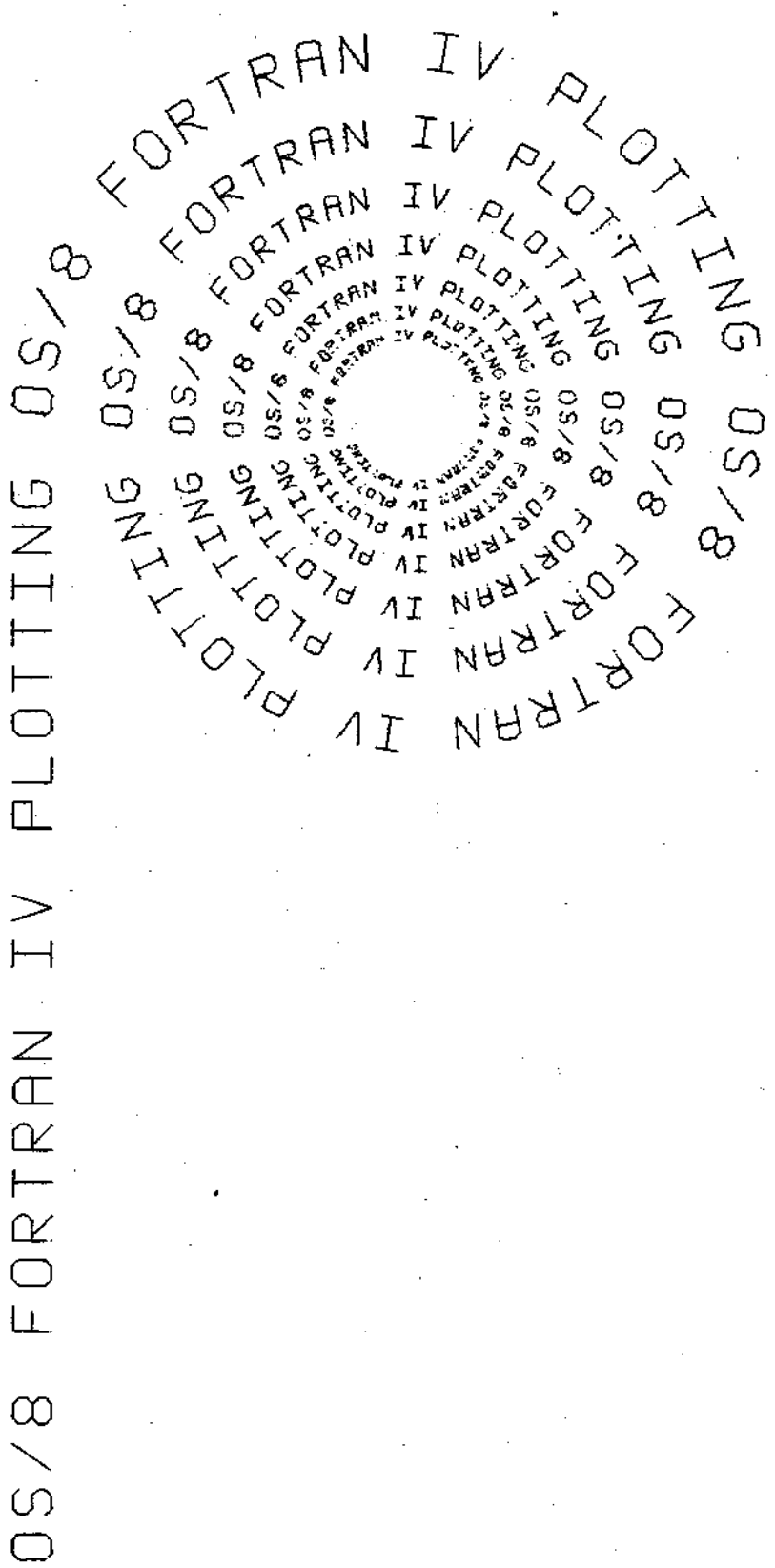


Figure 8-6 Spiral Plotter Example

The next example plots a histogram.

```
      DIMENSION X(50),Y(50),SALES(12),TXT(7)
      DATA SALES/20,30,32,40,50,90,95,90,40,30,20,15/
      DATA TXT/'AIR CONDITIONER SALES EACH MONTH OF 1972 '/
      CALL PLOTS(,01,0)
      CALL XYPLOT(0,-30,-3)
      CALL XYPLOT(1,1,-3)
      DO 5 J=1,12
      I=(J-1)*4+1
      Y(I)=0
      Y(I+1)=SALES(J)
      Y(I+2)=SALES(J)
5     Y(I+3)=0.
      Y(49)=0.
      Y(59)=10.
      DO 10 I=1,48,4
      X(I)=I/4+.75
      X(I+1)=X(I)
      X(I+2)=I/4+.25
10    X(I+3)=X(I+2)
      X(49)=0.
      X(50)=1.
      CALL AXIS(0,0,'MONTH',-5,13,0,0,1)
      CALL AXIS(0,0,'AIR CONDITIONERS (X100)',23,10,90,0,10)
      CALL LINE(X,Y,48,1,0,0)
      CALL SYMBOL(1,10,5,.25,TXT,0,40)
      CALL XYPLOT(12,0,3)
      CALL PLEXIT
      END
```

The actual plotter output is shown in Figure 8-7.

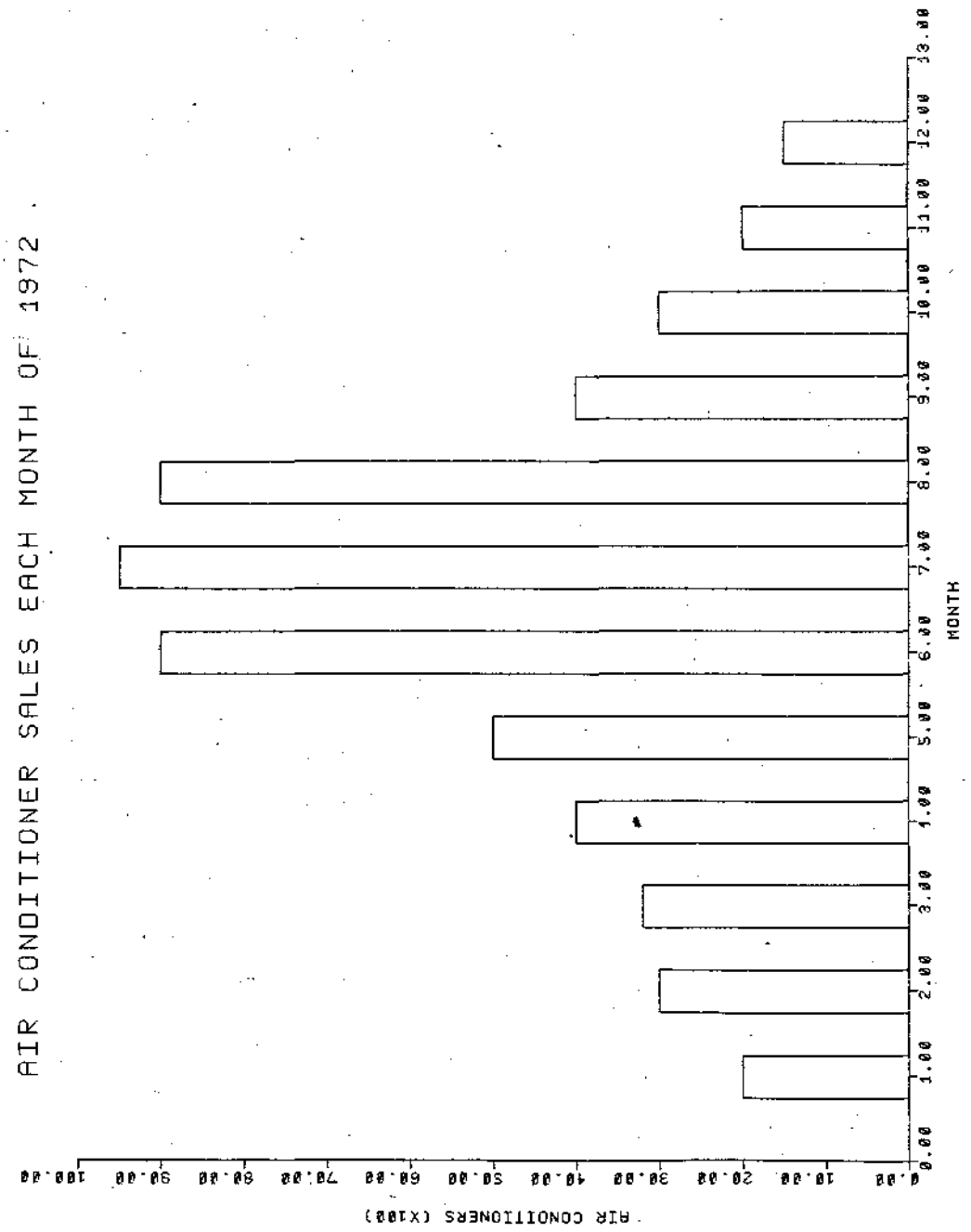


Figure 8-7 Histogram Plotter Example

appendices

appendix a
character codes

ASCII[1] Character Set

Character	8-Bit Octal	6-Bit Octal	Decimal Equivalent (A1 Format)	Character	8-Bit Octal	6-Bit Octal	Decimal Equivalent (A1 Format)
A	301	01	96	!	241	41	−1952
B	302	02	160	"	242	42	−1888
C	303	03	224	#	243	43	−1824
D	304	04	288	$	244	44	−1760
E	305	05	352	%	245	45	−1696
F	306	06	416	&	246	46	−1632
G	307	07	480	'	247	47	−1568
H	310	10	544	(	250	50	−1504
I	311	11	608	)	251	51	−1440
J	312	12	672	*	252	52	−1376
K	313	13	736	+	253	53	−1312
L	314	14	800	,	254	54	−1248
M	315	15	864	-	255	55	−1184
N	316	16	928	.	256	56	−1120
O	317	17	992	/	257	57	−1056
P	320	20	1056	:	272	72	−352
Q	321	21	1120	;	273	73	−288
R	322	22	1184	<	274	74	−224
S	323	23	1248	=	275	75	−160
T	324	24	1312	>	276	76	−96
U	325	25	1376	?	277	77	−32
V	326	26	1440	@	300		32
W	327	27	1504	[	333	33	1760
X	330	30	1568	\	334	34	1824
Y	331	31	1632	]	335	35	1888
Z	332	32	1696	↑(^)[2]	336	36	1952
0	260	60	−992	←(_)[2]	337	37	2016
1	261	61	−928	Leader/Trailer	200		
2	262	62	−864	LINE FEED	212		
3	263	63	−800	Carriage RETURN	215		
4	264	64	−736	SPACE	240	40	−2016
5	265	65	−672	RUBOUT	377		
6	266	66	−608	Blank	000		
7	267	67	−544	BELL	207		
8	270	70	−480	TAB	211		
9	271	71	−416	FORM	214		

[1] An abbreviation for American Standard Code for Information Interchange.

[2] The character in parentheses is printed on some Teletypes.

appendix b
loading procedures

Initializing the System

Before using the computer system, it is good practice to initialize all units. To initialize the system, ensure that all switches and controls are as specified below.

1. Main power cord is properly plugged in.
2. Terminal is turned OFF.
3. Low-speed punch is OFF.
4. Low-speed reader is set to FREE.
5. Computer POWER key is ON.
6. PANEL LOCK is unlocked.
7. Console switches are set to 0.
8. SING STEP is not set.
9. High-speed punch is OFF.
10. DECtape REMOTE lamps OFF.

The system is now initialized and ready for your use.

Loaders

READ-IN MODE (RIM) LOADER

When a computer in the PDP-8 series is first received, it is nothing more than a piece of hardware; its core memory is completely demagnetized. The computer "knows" absolutely nothing, not even how to receive input. However, the programmer can manually load data directly into core using the console switches.

The RIM Loader is the very first program loaded into the computer, and it is loaded by the programmer using the console

switches. The RIM Loader instructs the computer to receive and store, in core, data punched on paper tape in RIM coded format (RIM Loader is used to load the BIN Loader described below.)

There are two RIM loader programs: one is used when the input is to be from the low-speed paper tape reader, and the other is used when input is to be from the high-speed paper tape reader. The locations and corresponding instructions for the low-speed reader are listed in Table B-1. The high-speed reader RIM loader is listed in Table B-2.

For each step in the table, place each of the PDP-8/E console SWITCH REGISTER switches numbered 0 to 11 either in the up position if the corresponding table entry is 1, or in the down position if the corresponding table entry is 0. When all 12 switches have been set to correspond to a line in the table, follow the instructions in the right hand column and proceed to the next line. The tables also include octal values of the binary switch settings for the benefit of users familiar with octal numbers.

Table B-1 RIM Loader for Low-Speed Reader

Step #	Octal Values	Switch Register Setting				And Then
		012	345	678	91011	
1	0000	000	000	000	000	press EXTD ADDR LOAD
2	7756	111	111	101	110	press ADDR LOAD
3	6032	110	000	011	010	lift DEP key
4	6031	110	000	011	001	lift DEP key
5	5357	101	011	101	111	lift DEP key
6	6036	110	000	011	110	lift DEP key
7	7106	111	001	000	110	lift DEP key
8	7006	111	000	000	110	lift DEP key
9	7510	111	101	001	000	lift DEP key
10	5357	101	011	101	111	lift DEP key
11	7006	111	000	000	110	lift DEP key
12	6031	110	000	011	001	lift DEP key
13	5367	101	011	110	111	lift DEP key
14	6034	110	000	011	100	lift DEP key
15	7420	111	100	010	000	lift DEP key
16	3776	011	111	111	110	lift DEP key
17	3376	011	011	111	110	lift DEP key
18	5356	101	011	101	110	lift DEP key

Table B-2 RIM Loader for High-Speed Reader

Step #	Octal Values	Switch Register Setting				And Then
		012	345	678	91011	
1	0000	000	000	000	000	press EXTD ADDR LOAD
2	7756	111	111	101	110	press ADDR LOAD
3	6014	110	000	001	100	lift DEP key
4	6011	110	000	001	001	lift DEP key
5	5357	101	011	101	111	lift DEP key
6	6016	110	000	001	110	lift DEP key
7	7106	111	001	000	110	lift DEP key
8	7006	111	000	000	110	lift DEP key
9	7510	111	101	001	000	lift DEP key
10	5374	101	011	111	100	lift DEP key
11	7006	111	000	000	110	lift DEP key
12	6011	110	000	001	001	lift DEP key
13	5367	101	011	110	111	lift DEP key
14	6016	110	000	001	110	lift DEP key
15	7420	111	100	010	000	lift DEP key
16	3776	011	111	111	110	lift DEP key
17	3376	011	011	111	110	lift DEP key
18	5357	101	011	101	111	lift DEP key

After RIM has been loaded, it is good programming practice to verify that all instructions were stored properly. This can be done by performing the steps illustrated in Figure B-2, which also shows how to correct an incorrectly stored instruction.

When loaded, the RIM Loader occupies absolute locations 7756 through 7776.

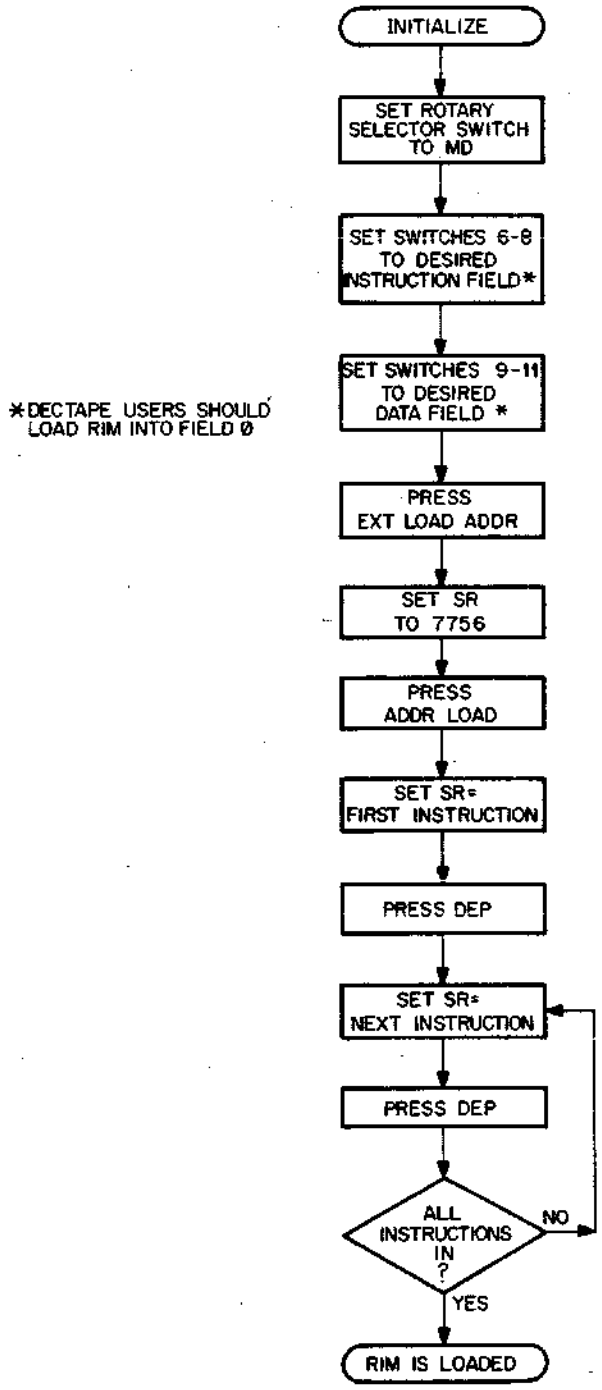


Figure B-1 Loading the RIM Loader

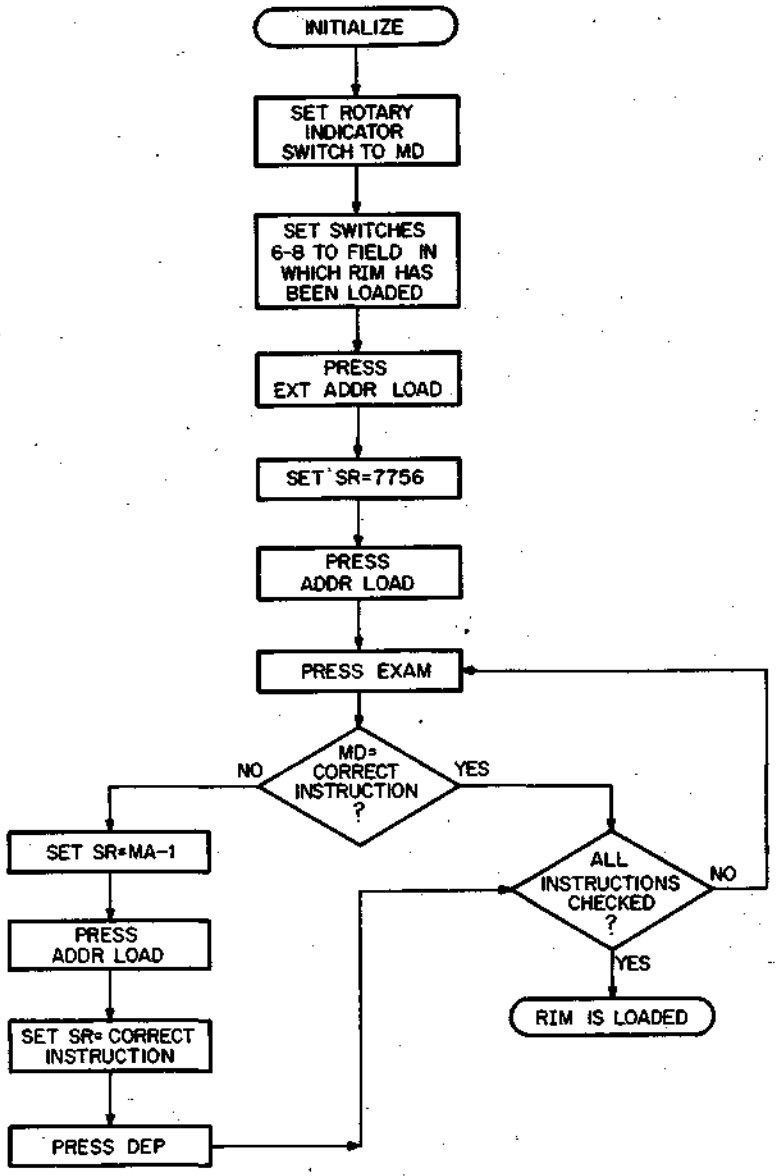


Figure B-2 Checking the RIM Loader

BINARY (BIN) LOADER—

The BIN Loader is a short utility program which, when in core, instructs the computer to read binary-coded data punched on paper tape and store it in core memory. BIN is used primarily to load the programs furnished in the software package (excluding the loaders and certain subroutines) and the programmer's binary tapes.

BIN is furnished to the programmer on punched paper tape in RIM-coded format. Therefore, RIM must be in core before BIN can be loaded. Figure B-3 illustrates the steps necessary to properly load BIN. And when loading, the input device (low- or high-speed reader) must be that which was selected when loading RIM.

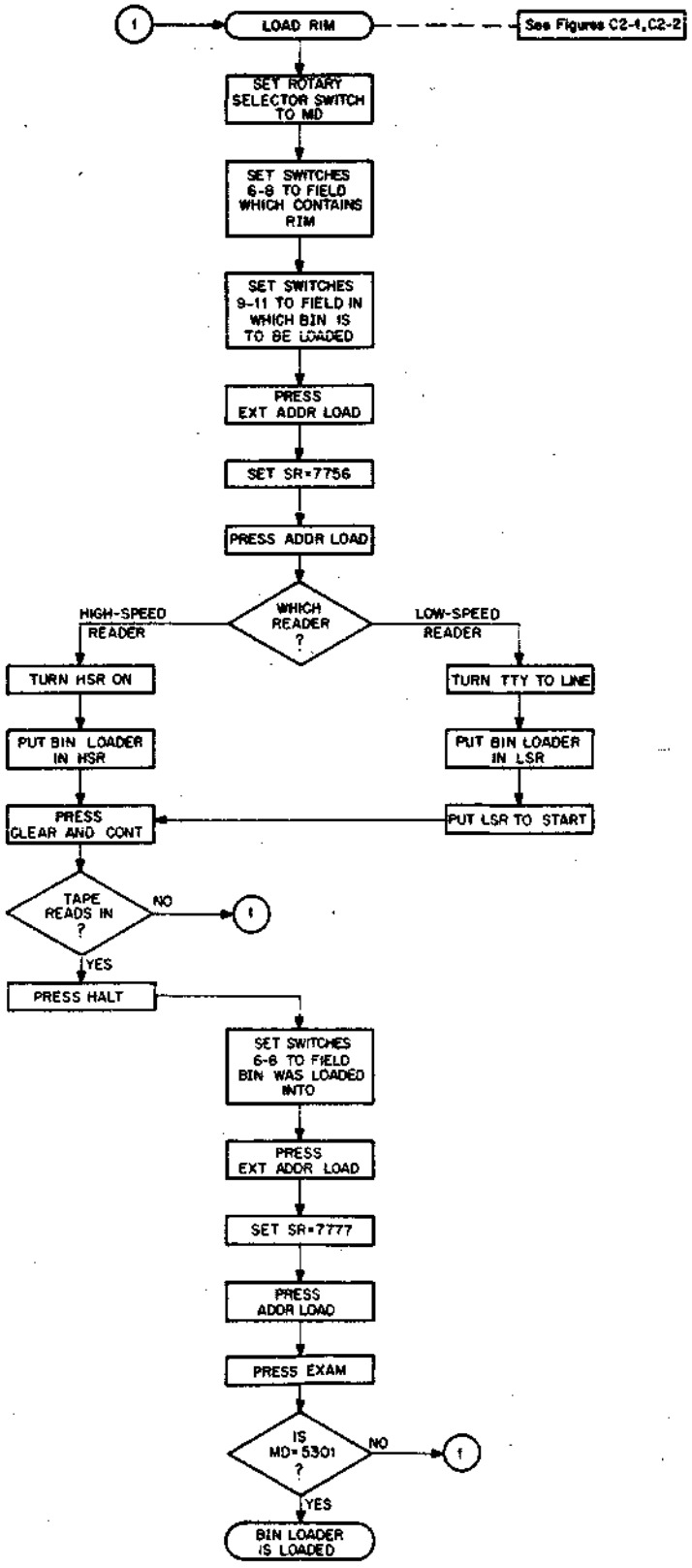


Figure B-3 Loading the BIN Loader

When stored in core, BIN resides on the last page of core, occupying absolute locations 7625 through 7752 and 7777.

BIN was purposely placed on the last page of core so that it would always be available for use—the programs in DEC's software package do not use the last page of core (excluding the Disk Monitor). The programmer must be aware that if he writes a program which uses the last page of core, BIN will be wiped out when that program runs on the computer. When this happens, the programmer must load RIM and then BIN before he can load another binary tape.

Binary tapes to be loaded should be started on the leader-trailer code (Code 200), otherwise zeros may be loaded into core, destroying previous instructions.

Figure B-4 illustrates the procedure for loading binary tapes into core.

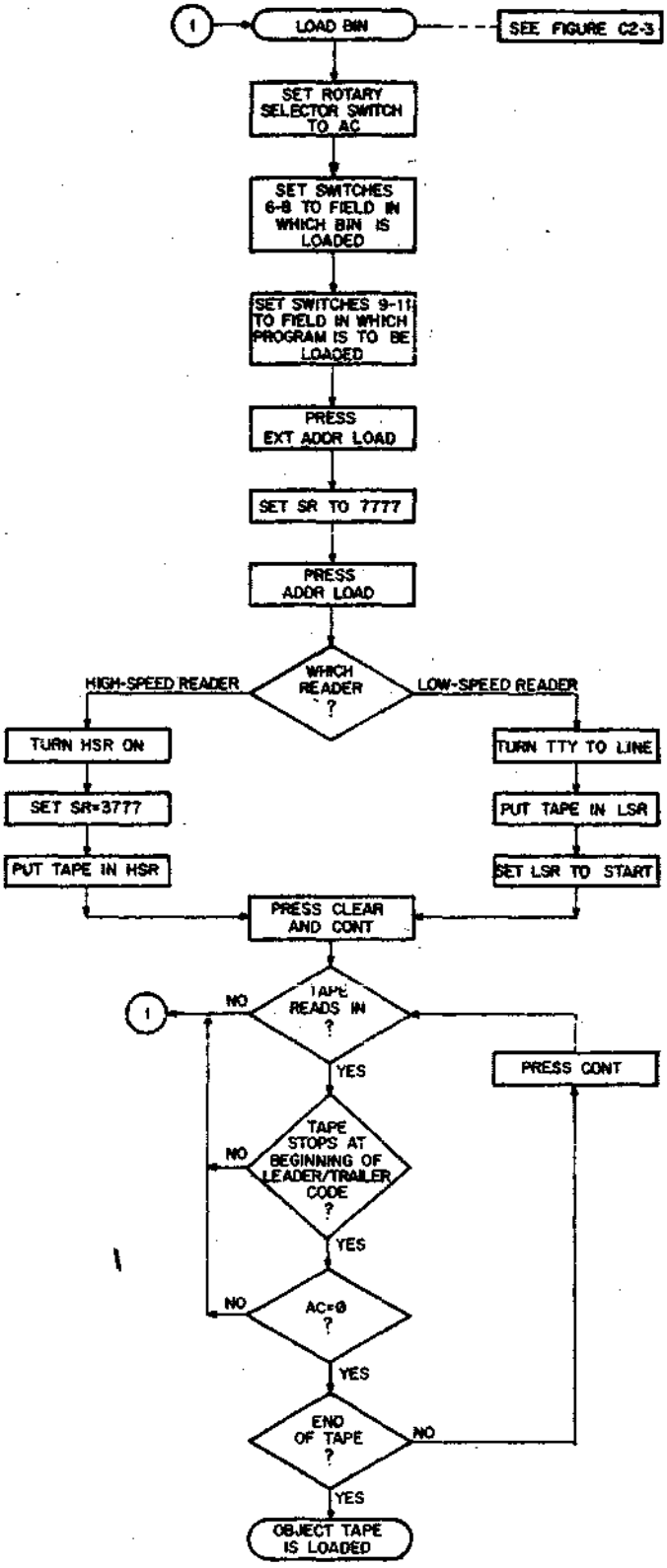


Figure B-4 Loading a Binary Tape Using BIN

appendix C
permanent symbol table

The following are the elements of the PDP-8 instruction set found in the SABR permanent symbol table. These instructions are already defined within the computer. For additional information on these instructions and for a description of the symbols used when programming other, optional, I/O devices, see the *Small Computer Handbook,* available from the DEC Software Distribution Center.

INSTRUCTION CODES

Mnemonic	Code	Operation	Time (μsec.)[1]
Memory Reference Instructions			
AND	0000	Logical AND	2.6
TAD	1000	Two's complement add	2.6
ISZ	2000	Increment and skip if zero	2.6
INC	2000	Nonskip ISZ	2.6
DCA	3000	Deposit and clear AC	2.6
JMS	4000	Jump to subroutine	2.6
JMP	5000	Jump	1.2
			Sequence
Group 1 Operate Microinstructions (1 cycle[2])			
NOP	7000	No operation	•
IAC	7001	Increment AC	3
RAL	7004	Rotate AC and link left one	4
RTL	7006	Rotate AC and link left two	4
RAR	7010	Rotate AC and link right one	4
RTR	7012	Rotate AC and link right two	4
CML	7020	Complemented link	2
CMA	7040	Complement AC	2
CLL	7100	Clear link	1
CLA	7200	Clear AC	1

[1] Times are representative of the PDP-8/E.

[2] 1 cycle is equal to 1.2 microseconds.

Mnemonic	Code	Operation	Sequence
Group 2 Operate Microinstructions (1 cycle)			
HLT	7402	Halts the computer	3
OSR	7404	Inclusive OR SR with AC	3
SKP	7410	Skip unconditionally	1
SNL	7420	Skip on nonzero link	1
SZL	7430	Skip on zero link	1
SZA	7440	Skip on zero AC	1
SNA	7450	Skip on nonzero AC	1
SMA	7500	Skip on minus AC	1
SPA	7510	Skip on positive AC (zero is positive	1
Combined Operate Microinstructions			
CIA	7041	Complement and increment AC	2, 3
STL	7120	Sent link to 1	1, 2
STA	7240	Set AC to − 1	2
Internal IOT Microinstructions			
ION	6001	Turn interrupt processor on	
IOF	6002	Disable interrupt processor	
Keyboard/Reader (1 cycle)			
KSF	6031	Skip on keyboard/reader flag	
KRB	6036	Clear AC, read keyboard buffer (dynamic), clear keyboard flags	
Teleprinter/Punch (1 cycle)			
TSF	6041	Skip on teleprinter/punch flag	
TLS	6046	Load teleprinter/punch, print, and clear teleprinter/punch flag	
High Speed Reader—Type PR8/E (1 cycle)			
RSF	6011	Skip on reader flag	
RRB	6012	Read reader buffer and clear reader flag	
RFC	6014	Clear flag and buffer and fetch character	
High Speed Punch—Type PP8/E (1 cycle)			
PSF	6021	Skip on punch flag	
PLS	6026	Clear flag and buffer, load buffer and punch character	

PSEUDO-OPERATORS

The following is a list of the SABR assembler pseudo-operators.

ABSYM
ACH
ACM
ACL
ARG
BLOCK
CALL
COMMN
CPAGE
DECIM
DUMMY
EAP
END
ENTRY
FORTR
I
IF
LAP
OCTAL
OPDEF
PAGE
PAUSE
REORG
RETRN
SKPDF
TEXT

appendix d

OS/8 demonstration run

The following pages present a demonstration of the use of the OS/8 system. The terminal output is set off by letters (to its left) which correspond to the textual explanations on the facing page. This demonstration illustrates the procedures involved and use of many of the OS/8 system programs and commands.

A The CCL command is used to zero the DECtape on Unit 1, specifying one additional information word in the directory.

B The user then types the DATE command to set the system date to April 10, 1974.

C The ASSIGN command is used to give DTA1 the additional name IN. All subsequent references to IN refer to DTA1.

D DIRECT is called to list the directory of DECtape Unit 1. A directory listing of DTA1 is produced.

E The Keyboard Monitor GET and SAVE commands are used to copy EDIT from the system device to DTA1.

F The FORTRAN compiler is run via the CCL command COMPILE to compile and execute the program TEST1 on the device DSK: An output relocatable binary file named TEST1 is saved by SABR on DECtape Unit 1. The program has an error in it. Control is returned to the Keyboard Monitor after execution and the error message printed on the terminal.

G The program EDIT, located on DTA1, is used to correct the error in TEST1. The old program, TEST1, is input to the Editor, and the new (corrected) program, TEST2, is written by the Editor onto DTA1. The first page is yanked into core.

H The user has noticed a misspelled word in FORMAT line 35 and used the string search feature of the Editor to correct it. An END statement is appended to the program.

A
```
.ZERO DTA1:=1
```

B
```
.DA 4/10/74
```

C
```
.AS DTA1 IN
```

D
```
.DIR IN:

 10-APR-74



 730 FREE BLOCKS
```

E
```
.GET SYS EDIT

.SAVE IN EDIT 0-5000,200=2001
```

F
```
.COMPILE IN:TEST2<IN:TEST2
/       CALL EXIT

NO END STATEMENT
```

G
```
.RUN IN EDIT
*IN:TEST2.FT<TEST1.FT

#Y
```

H
```
#/:0055
#$35      'L
35        FORMAT ('THE AVERATE IS' F20.2/)

#.S
35        FORMAT ('THE AVERAT\GE IS' F20.2/)

#.L
35        FORMAT ('THE AVERAGE IS' F20.2/)

#/L
          CALL EXIT

#A
          END
```

I The user instructs the Editor to list the entire FORTRAN program.

J Note the use of implied DO loops in the READ and WRITE statements . . .

K and device independent I/O. A file named ABCD.DA is opened on the default device DSK and data is written into it. When all the data is entered, the file is closed. Later, this file is again opened, and the data is read and used by the program.

L An S in column 1 of a FORTRAN line indicates that the line contains SABR code.

M CALL EXIT is used to return control to the Keyboard Monitor after execution.

N After listing the program, the E command to the Editor closes the file and returns control to the Keyboard Monitor.

```
#L
C       THIS PROGRAM PRESENTS A FEW OF THE FEATURES
C       OF OS/8 FORTRAN; SPECIFICALLY IT INCLUDES IM-
C       PLIED DO LOOPS, DIRECT INSERTION OF SABR CODE
C       AND EXPANDED I/O.

C       THIS SECTION READS DATA FROM THE TTY AND WRITES
C       IT ONTO THE DSK AS AN ARRAY.

        DIMENSION A(10)
5       CALL OOPEN ('DSK','ABCD')
        WRITE (1,10)
10      FORMAT ('ENTER 10 NUMBERS IN F6.2 FORMAT.')
        WRITE (1,11)
11      FORMAT ('FOLLOW EACH WITH A CARRIAGE RETURN:'//)
        READ (1,15) (A(N), N=1,10)
        WRITE (4,15) (A(N), N=1,10)
15      FORMAT (F6.2)
        CALL OCLOSE

C       THIS SECTION ADDS THE NUMBERS STORED ON THE DSK
C       AND AVERAGES THEM, PRINTING BOTH RESULTS ON
C       THE TELETYPE.

        SUM=0.0
        DO 20 I=1,10
20      A(I)=0.0
        CALL IOPEN ('DSK','ABCD')
        READ (4,15) (A(N), N=1,10)
        DO 25 N=1,10
        SUM=SUM+A(N)
25      CONTINUE
        WRITE (1,30) SUM
30      FORMAT (/'THE SUM IS' F20.2)
        AVR=SUM/10.
        WRITE (1,35)AVR
35      FORMAT ('THE AVERAGE IS' F20.2/)

C       THE SABR CODE FOLLOWING CHECKS FOR A CARRIAGE
C       RETURN CHARACTER TO INITIATE REPEATING THE
C       PROGRAM.  ANY OTHER CHARACTER TERMINATES THE
C       PROGRAM.

        WRITE (1,40)
40      FORMAT ('TO REPEAT, TYPE A CARRIAGE RETURN.'//)
SX,     KSF
S       JMP X
S       KRB
S       TAD MYES
S       SZA
S       JMP \50
        GO TO 5
SMYES,  -215
50      WRITE (1,60)
60      FORMAT (/'PROGRAM DONE'//)
        CALL EXIT
        END

#E
```

O The ASSIGN command is used to change the assigned name of DTA1 from IN to OUT. The FORTRAN compiler is called again, and the program is loaded. An output relocatable binary file named TEST2 is saved by SABR on DECtape Unit 1.

P The FORTRAN program is executed via the CCL command EXECUTE. The /G, /I, and /O options cause automatic loading and execution of the program and the device independent I/O., and results are calculated and returned. Execution is not repeated.

Q The DEASSIGN command is used to delete all user-assigned device names. The ASSIGN command is then used to give the name X to DTA1.

R The CCL command DIR is used to obtain a directory listing of DECtape Unit 1. TEST2.RL is the relocatable binary output file from the FORTRAN compilation.

S Next, the CCL command DIR is used to print the directory of the system device on the line printer. ABCD.DA is the FORTRAN data file created in the preceding program.

T The CCL command DEL is used to delete the unwanted files PROG3 and PROG4, from the system device. Then the ASCII file TEST2 is copied from DECtape Unit 1 to the system device with the CCL command COPY.

O
```
.AS DTA1 OUT
.COMPILE OUT:TEST2<OUT:TEST2
```

P
```
.EXECUTE OUT:TEST2/G/I/O

ENTER 10 NUMBERS IN F6.2 FORMAT.
FOLLOW EACH WITH A CARRIAGE RETURN:


16.23
32.00
171.45
2.15
22.10
77.35
2.91
66.00
.46
27.50
THE SUM IS                418.15
THE AVERAGE IS             41.81

TO REPEAT, TYPE A CARRIAGE RETURN.



PROGRAM DONE
```

Q
```
.DEA

.AS DTA1 X
```

R
```
.DIR X:

 10-APR-74

EDIT  .SV   12 10-APR-74
TEST2 .FT    4 10-APR-74
TEST2 .RL    4 10-APR-74


 710 FREE BLOCKS
```

```
.DIR LPT:<SYS:

 10-APR-74

ABSLDR.SV    5 15-JAN-74
CCL    .SV   17 26-FEB-74
DIRECT.SV    7 18-JAN-74
FOTP   .SV    8 18-JAN-74
PIP    .SV   11 18-JAN-74
LIB8   .RL   29 18-JAN-74
EDIT   .SV   10 18-JAN-74
PAL8   .SV   16 18-JAN-74
CREF   .SV   13 18-JAN-74
BITMAP.SV    5 18-JAN-74
FORT   .SV   25 18-JAN-74
SABR   .SV   24 18-JAN-74
LOADER.SV   12 18-JAN-74
SRCCOM.SV    5 18-JAN-74
BOOT   .SV    5 18-JAN-74
BUILD  .SV   33 18-JAN-74
EPIC   .SV   14 18-JAN-74
PIP10  .SV   17 18-JAN-74
RESORC.SV   10 18-JAN-74
DTFRMT.SV    7 18-JAN-74
TDFRMT.SV    9 18-JAN-74
RK8FMT.SV    9 18-JAN-74
RKEFMT.SV    6 18-JAN-74
CAMP   .SV    8 18-JAN-74
MCPIP  .SV   13 18-JAN-74
DTCOPY.SV    5 18-JAN-74
TDCOPY.SV    7 18-JAN-74
LIBSET.SV    5 18-JAN-74
CCL    .PA 130 26-FEB-74
TEST1  .BK    4 11-APR-74
TEST1  .RL    4
TEST1  .FT    4 10-APR-74
TEST2  .RL    4 10-APR-74
ABCD   .DA    1 10-APR-74
PROG3  .      1 10-APR-74
PROG4  .      1 10-APR-74

2295 FREE BLOCKS
```

S

```
.DEL PROG3,PROG4
FILES DELETED:
PROG3.
PROG4.

.COPY SYS:TEST2.FT<X:TEST2.FT
```

T

appendix e

OS/8 error message summary

The following summary is provided for the user's convenience. Error messages generated by OS/8 programs are listed in alphabetic order and identified by the system program by which they are generated. This appendix is only a summary. Refer to the appropriate chapters for more detailed information about error conditions.

Message	Program	Explanation
?0	SRCCOM	Insufficient core—this means that the differences between the files are too large to allow for effective comparison. Use of the /X option may alleviate this problem.
0	Editor	Editor failed in reading a device. Error occurred in device handler; most likely a hardware malfunction.
0000	Linking Loader	/I or /O specified too late.
0001	Linking Loader	Symbol table overflow; more than 64 subprogram names.
0002	Linking Loader	Program will not fit into core.
0003	Linking Loader	Program with largest common storage area was not loaded first.
0004	Linking Loader	Checksum error in input tape.
0005	Linking Loader	Illegal relocation code.

Message	Program	Explanation
0006	Linking Loader	An output error has occurred.
0007	Linking Loader	An input error has occurred (either a physical device error, or an attempt was made to read from a write-only device such as LPT:).
0010	Linking Loader	No starting address has been specified and there is no entry point named MAIN.
0011	Linking Loader	An error occurred while the Loader was attempting to load a device handler.
0012	Linking Loader	I/O error on system device.
?1	SRCCOM	Input error on file #1 or less than 2 input files specified.
1	Editor	Editor failed in writing onto a device. Generally a hardware malfunction or WRITE-LOCKed device.
1	TECO	Illegal command.
?2	SRCCOM	Input error on file #2.
2	Editor	File close error occurred. The output file could not be closed; the file does not exist on that device.
2	TECO	Incomplete command. (See TECO in Chapter 2.)
2045 REFS	CREF	More than 2044 (decimal) references to one symbol were made.
?3	SRCCOM	Output file too large for output device.
3	Editor	File open error occurred. This error occurs if the output device is a read-only device or if no output file name is specified on a file-oriented output device.

Message	Program	Explanation
3	TECO	Non-alphanumeric Q-register name.
?4	SRCCOM	Output error.
4	Editor	Device handler error occurred. The Editor could not load the device handler for the specified device. This error should never occur.
4	TECO	Command iterations or macro calls nested too deeply.
?5	SRCCOM	Could not create output file.
5	TECO	Text buffer overflow.
6	TECO	Search string longer than 31 characters.
7	TECO	Numeric argument missing before comma, equal sign, U, or quote (").
8	TECO	Illegal filename in ER, EW, or EB command.
9	TECO	Semicolon or failing search encountered on command level.
10	TECO	Iteration close (>) without matching open (<).
11	TECO	Attempt to move pointer outside of text buffer.
12	TECO	Q-register storage overflow.
13	TECO	Incomplete command.
14	TECO	Output file too large, or else output parity error.
15	TECO	Input file parity error.
16	TECO	File error. (See Chapter 2.)
17	TECO	An output command was encountered which would have caused TECO to overflow its current output file. Users should close the current output file and write all further

Message	Program	Explanation
		output onto one (or more) additional files. These files may be combined if necessary.
18	TECO	Attempt to execute an output command without opening an output file.
A	SABR	Too many or too few ARG statements follow a CALL statement.
AA	F4	More than six subroutine arguments are arrays.
ALOG	FORT Library	Attempt to compute log of negative number.
ALREADY EXISTS (filename)	FOTP	An attempt was made to rename an output file with the name of an existing output file.
ARE YOU SURE?	PIP	Occurs when using the /S option. A response of Y will compress the files.
ARITHMETIC EXPRESSION TOO COMPLEX	FORT	Self-explanatory.
AS	F4	Bad ASSIGN statement.
?BAD ARG	BUILD	No device name was included in the LOAD command.
BAD ARG	FRTS	Illegal argument to library function.
BAD ARGS	Keyboard Monitor	The arguments to the SAVE command are not consistent and violate restrictions.
BAD-BLK	EPIC	When EPIC is punching a patch it checks the block specified by =n to see if it is within range. If the block is out of range EPIC outputs this error message and returns to the command decoder.

Message	Program	Explanation
BAD CHECKSUM, FILE #n	ABSLDR	File number n of the input file list has a checksum error.
BAD CHECKSUM, FILE #n	BITMAP	File number n of the input file list had a checksum error.
BAD CORE IMAGE	Keyboard Monitor	The file requested was not a core image file.
BAD DATE	Keyboard Monitor	The date has not been entered correctly, or incorrect arguments were used, or the date was out of range.
BAD DEVICE	CCL	The device specified in a CCL command is not of the correct form.
?BAD DIRECTORY	RESORC	Input device directory cannot be read.
BAD DIRECTORY ON DEVICE #n	PIP	Error message occurs when: 1. PIP is trying to read the directory, but it is not an OS/8 directory. 2. The output device does not have a system directory; i.e., file storage begins at record 7 (occurs during a /Y transfer). n is the number of the file in the input file list.
BAD EXTENSION	CCL	Either an extension was specified without a file name or two extensions were specified.
BAD FORMAT OR CHECKSUM— TRY AGAIN	LIBSET	Error in reading relocatable binary file.
?BAD INPUT	BUILD	An error was detected in the binary file; it is not a proper input for the LOAD command.
BAD INPUT DIRECTORY	DIRECT FOTP	The directory on the specified input device is not a valid OS/8 directory.
BAD INPUT FILE	Loader	An input file was not a RALF module.

Message	Program	Explanation
BAD INPUT, FILE #n	ABSLDR	Attempt was made to load a non-binary file as file number n of the input file list; or a non-core image with /I option.
BAD INPUT, FILE #n	BITMAP	A physical end of file was reached before a logical end of file, or extraneous characters were found in binary file n.
#BAD LINE. JOB ABORTED	BATCH	The BATCH monitor detected a record in the input file that did not have one of the characters dot, slash, dollar sign, or asterisk as the first character of the record. The record is ignored, and BATCH scans the input file for the next $JOB record.
?BAD LOAD	BUILD	An attempt was made to load a binary handler that is not in the correct format.
?BAD MONITOR	RESORC	The input device may be a system device but the Monitor cannot be accessed.
BAD MONITOR	CCL	The version of the Keyboard Monitor being used is not compatible with CCL. A newer version of the monitor must be obtained from Digital before CCL can be used.
BAD NUMBER	CCL	A CCL command which uses the # construction does not have the full 16-digit specification that is required.
?BAD ORIGIN	BUILD	The origin in a binary file is not in the range 200-577.
BAD OUTPUT DEVICE	FOTP	This message usually appears when a non-file structured device is specified as the output device.

Message	Program	Explanation
BAD OUTPUT DEVICE	Loader	The loader image file device was not a directory device, or the symbol map file device was a read-only device. The entire line is ignored.
BAD OUTPUT DIRECTORY	FOTP	The directory on the specified output device is not a valid OS/8 device directory.
BAD RECOLLECTION	CCL	An attempt was made to use a previously remembered argument when no argument was saved.
BAD SWITCH OPTION	CCL	The character used with a slash (/) to indicate an option is not a legal option.
BAD SYSTEM HEAD	PIP	If the /Y option is used and the area being transferred does not contain OS/8, this message results.
BATCH.SV NOT FOUND ON SYS:	BATCH	A copy of BATCH.SV must exist on the system device. Control returns to the OS/8 Monitor.
BD	F4	Bad dimensions (too big, or syntax) in DIMENSION, COMMON or TYPE declaration.
BE	FLAP RALF	Illegal equate. The symbol had been defined previously.
	PAL8	Two PAL8 internal tables have overlapped. Fatal error —assembly cannot continue.
BI	FLAP RALF	Illegal index register specification.
BO	FRTS	No more file buffer available.
BS	F4	Illegal in BLOCK DATA program.
BX	FLAP RALF	Bad expression. Something in the expression is incorrect, or the expression is not valid in this context.

Message	Program	Explanation
C	SABR	An illegal character appears on the line.
CANNOT CHANGE CORE CAPACITY WHILE RUNNING BATCH	CCL	A CORE command was issued while the BATCH program was running.
CANNOT HANDLE VARIABLE LENGTH RECORDS	MCPIP	The records on the input and output files specified are not the same size.
% CAN'T—AT BOF	CAMP	A file mark was read before the specified number of records were read over in a BACKSPACE command. The device is moved forward so that it is positioned at the beginning of the file.
? CAN'T—AT BOT	CAMP	A BACKSPACE command cannot move the device backward the specified number of files because the device is positioned at the beginning of the first file.
% CAN'T—AT EOD	CAMP	The specified number of files cannot be advanced over because the end of data was encountered. The tape is positioned at the end of data.
% CAN'T—AT EOF	CAMP	A file mark was read before the specified number of records were advanced over in a SKIP command. The tape is moved backward one record to leave it positioned at the end of the file.
? CAN'T—DEVICE DOESN'T EXIST	CAMP	The device specified in a CAMP command is not present on the OS/8 system.
? CAN'T—DEVICE IS READ-ONLY	CAMP	The device specified in a CAMP command is a read-only device; e.g., PTR.
? CAN'T—DEVICE IS WRITE-ONLY	CAMP	The device specified in a CAMP command is a write-only device; e.g., TTY.

Message	Program	Explanation
? CAN'T FOR THIS DEVICE	CAMP	The operation specified does not make sense for the device specified.
? CAN'T I/O ERROR	CAMP	This message is followed by a brief explanation of the input/output error that occurred.
CAN'T OPEN OUTPUT FILE	PIP	Message occurs due to one of the following: 1. Output file is on a read-only device. 2. No name has been specified for the output file. 3. A /Y transfer has been attempted to a non-directory device. 4. Output file has zero free blocks.
CAN'T READ IT	FRTS	I/O error on reading loader image file.
%CAN'T REMEMBER	CCL	The argument specified in a CCL command line is too long to be remembered or an I/O error occurred.
CAUTION— DO DP	FRTS	The present hardware configuration does not include an FPP-12 Floating-Point Processor with double precision option.
CCL #x OVERLAY & MONITOR INCOMPATIBLE	CCL	The version of CCL being used is not compatible with the Keyboard Monitor present on the system. Type R CCL to retry.
CH	BCOMP	Error in CHAIN statement.
	PAL8	Chain to CREF error—CREF-SV was not found on SYS:.
CHER	FORT Library	File specified as argument to CHAIN not found on system device.
CI	BRTS	Inquire failure in CHAIN. Device not found.

Message	Program	Explanation
CL	BRTS	Lookup failure in CHAIN. Filename not found.
	F4	Bad COMPLEX literal.
CLOSE ERROR	MCPIP	MCPIP is not able to close the file. A bad file just created on magnetic tape or cassette must be removed by placing a sentinel file after the preceding file.
CLOSE FAILED	CREF	CLOSE on output file failed.
CO	F4	Syntax error in COMMON statement.
COMMAND LINE OVERFLOW	CCL	The command line specified with the @ construction is more than 512 characters in length.
COMMAND TOO LONG	CCL	The length of a text argument in a MUNG command is too long.
COMPLIER MALFUNCTION	FORT	The meaning of this message has been extended to cover various unlikely Monitor errors.
CONTRADICTORY SWITCHES	CCL	Either two CCL processor switches were specified in the same command line or the file extension and the processor switch do not agree.
?CORE	BUILD	A CORE command specified more memory than is physically available, or the BOOT command was issued on an 8K system with a 2-page system handler active. Two page system handlers require at least 12K of core to be present on the OS/8 system.
D	SABR	A device handler has returned a fatal condition.
DA	BRTS	Attempt to read past end of data list.

Message	Program	Explanation
DA	F4	Bad syntax in DATA statement.
DE	BCOMP	Error in DEF statement.
	BRTS	Device driver error. Caused by hardware I/O failure.
	F4	This type of statement illegal as end of DO loop.
	PAL8	Device error. An error was detected when trying to read or write a device. Fatal error—assembly cannot continue.
DELETES PERFORMED ONLY ON INPUT DEVICE GROUP 1 CAN'T HANDLE MULTIPLE DEVICE DELETES	FOTP	More than one input device was specified with the /D option when no output specification (device or filename) was included.
?DEV IS NOT FILE STRUCTURED	RESORC	The input device specified is not a file-structured device; e.g., PTR.
DEV LPT BAD	CREF	The default output device, LPT, cannot be used as it is not available on this system.
DEV NOT IMPLEMENTED	BATCH	BATCH cannot accept input from the specified input device because its handler is not permanently resident (SYS: or co-resident with SYS:). Control returns to the Command Decoder.
DEVICE DOES NOT HAVE A DIRECTORY	DIRECT	The input device is a non-directory device; e.g., PTR, DIRECT can only read directories from file-structured devices.
DEVICE FULL	PIP10	DECsystem-10 ran out of space on the output file during a transfer.
DEVICE #n NOT A DIRECTORY DEVICE	PIP	Message occurs when: 1. Trying to list the directory of a non-directory device.

Message	Program	Explanation
		2. The input designated in a /Y transfer is not on a directory device. n gives the number of the device in the input list.
DF	PAL8	Device full. Fatal error—assembly cannot continue.
	F4	Bad DEFINE FILE statement.
D.F. TOO BIG	FRTS	Product of number of records times number of blocks per record exceeds number of blocks in file.
DH	F4	Hollerith field error in DATA statement.
DI	BCOMP	Error in DIM statement syntax or string dimension greater than 72, or array dimensioned twice.
DIRECTORY ERROR	PIP	An error has occurred while reading or writing the directory during a /S option.
DIVIDE BY 0	FRTS	Attempt to divide by zero. The resulting quotient is set to zero and execution continues.
DIVZ	FORT Library	Division by zero; very large number is returned.
DL	F4	Data list and variable list are not same length.
DN	F4	DO-end missing or incorrectly nested. This message is not printed during pass 3. It is followed by the statement number of the erroneous statement rather than the ISN.
DO	BRTS	No more room for drivers. Too many different devices used in file commands.
	F4	Syntax error in DO or implied DO.

Message	Program	Explanation
name DOES NOT EXIST	CCL Command Decoder MCPIP	The device with the name given is not present on the OS/8 system.
DP	F4	DO loop parameter not integer or real.
?DSK	BUILD	The device specified in a DSK command is not a file-structured device.
DV	BRTS	Attempt to divide by 0. Result is set to zero (NF).
	FLAP RALF	An attempt was made in an expression evaluation to divide by zero.
E	SABR	There is no END statement.
EF	BRTS	Logical end of file. Usually caused when I/O device runs out of medium.
EG	FLAP RALF	The preceding line contains extra code which could not be used by the assembler.
EM	BRTS	Attempt to exponentiate a negative number to a power.
EN	BRTS	Enter error in opening file. Device is read only or there is already one variable file open on that device or file not found.
END OF TAPE	EPIC	EPIC was expecting a block of tape and found end of tape instead. Press CONT to retry.
END OF TAPE ENTER NEXT	EPIC	When EPIC is reading a file that is segmented across a number of paper tapes and encounters the end of a segment, it outputs this message and halts with AC=7777 to allow the user to enter the next segment of papertape. Press CONT to continue reading.

Message	Program	Explanation
ENTER ERROR	MCPIP	Error occurred while trying to enter an output file. This message usually means that the cassette or magnetic tape has no sentinel file.
ENTER FAILED	CREF	Entering an output file was unsuccessful—possibly output was specified to a read-only device.
EOF ERROR	FRTS	End of file encountered on input.
EQUALS OPTION BAD	DIRECT	The =n option is not in the range 0-7.
ERROR CLOSING FILE	DIRECT	System error.
ERROR DELETING FILE	PIP PIP10	An attempt was made to delete a file that does not exist.
ERROR IN COMMAND	CCL	A command not entered directly from the console terminal is not a legal CCL command. This error occurs when the argument of a UA, UB, or UC command was not a legal command.
ERROR ON INPUT DEVICE SKIPPING (filename)	FOTP	The file specified is not transferred, but any previous or subsequent files are transferred and indicated in the new directory.
ERROR ON OUTPUT DEVICE	BITMAP	Error occurred while writing on output device; i.e., output error on DECtape write.
ERROR ON OUTPUT DEVICE SKIPPING (fiilename)	FOTP	The file specified is not transferred, but any previous or subsequent files are transferred and indicated in the new directory.
ERROR READING INPUT DIRECTORY	DIRECT FOTP	An error occurred while reading the directory.
ERROR WHILE WRITING OUTPUT FILE	LIBSET	Fatal output error occurred.
ERROR WRITING FILE	DIRECT	An error occurred while writing the output file.

Message	Program	Explanation
ERROR WRITING OUTPUT DIRECTORY	FOTP	Self-explanatory.
ES	RALF	External symbol error.
EX	F4	Syntax error in EXTERNAL statement.
EXCESSIVE SUBSCRIPTS	FORT	Self-explanatory.
FB	FORT Library	Argument to EXP too large; very large number is returned.
	BRTS	FILE busy. Attempt to use a file already in use.
FC	BRTS	OS/8 error while closing variable file. Device is read-only on file already closed.
FE	BRTS	Fetch error in opening file. Device not found, or device handler too big for available space.
FETCH ERROR	MCPIP	Error occurred while trying to fetch an OS/8 device handler.
FI	BRTS	Attempt to close or use unopened file.
FILE ERROR	FRTS	Any of: a. A file specified as an existing file was not found. b. A file specified as a nonexistent file would not fit on the designated device. c. More than 1 nonexistent file was specified on a single device. d. File specification contained "*" as name or extension.
FILE NOT FOUND	PIP10	The requested file was not found on the specified device.
FILE OVERFLOW	FRTS	Attempt to write outside file boundaries.
FIX	FORT Library	Attempt to fix a number >2047; 2047 is returned.

Message	Program	Explanation
FL	FLAP RALF	An error has occurred in the FPP or software floating conversion routines.
FLPW	FORT Library	Negative number raised to floating point power; absolute value taken.
FM	BRTS	Attempt to fix minus number. Usually caused by negative subscripts or file numbers.
FMT1	FORT Library	Invalid format statement.
FMT2	FORT Library	Illegal character in I format.
FMT3	FORT Library	Illegal character in F or E format.
FN	BCOMP	Error in file number of file name designation.
	BRTS	Illegal file number. Only 0, 1, 2, 3, 4 are legal.
FO	BRTS	Attempt to fix number greater than 4095. Usually caused by negative subscripts of file numbers.
FORMAT ERROR	FRTS	Illegal syntax in FORMAT statement.
FP	BCOMP	Incorrect FOR loop parameters or FOR loop syntax.
FP	FLAP RALF	A syntax error was encountered in a floating point or extended precision constant.
FPP ERROR	FRTS	Hardware error on FPP start-up.
FR	BCOMB	Error in function arguments or function not defined.
FULL *	Editor	The specified output device has become full. The file is closed; the user must specify a new output file.
GR	BRTS	RETURN without a GOSUB.

Message	Program	Explanation
GS	BRTS	Too many nested GOSUBS. The limit is 10.
GT	F4	Syntax error in GO TO statement.
GV	F4	Assigned or computed GO TO variable must be integer or real.
HANDLER FAIL	CREF	This is a fatal error on output and can occur if either the system device or the selected output device is WRITE-LOCKed.
?HANDLERS	BUILD	More than 15 handlers, including SYS and DSK were active when a BOOT command was issued.
HO	F4	Hollerith field error.
I	SABR	An illegal syntax has been used.
IA	BRTS	Illegal argument in UDEF function call.
IC	FLAP RALF	The symbol or expression in a conditional is improperly used, or left angle bracket is missing. The conditional pseudo-op is ignored.
IC	PAL8	Illegal character. The character is ignored and the assembly continued.
ID	PAL8	Illegal redefinition of a symbol.
IE	F4	Error reading input file. Control returns to the Keyboard Monitor.
	PAL8	Illegal equals—an attempt was made to equate a variable to an expression containing an undefined term. The variable remains undefined.

Message	Program	Explanation
IE	RALF	An entry point has not been defined, or is absolute, or also is defined as a common section, or external.
IF	BCOMP	THEN or GOTO missing from IF statement, or bad relational operator.
	BRTS	Illegal DEV:filename specification.
	F4	Logical IF statement cannot be used with DO, DATA, INTEGER, etc.
II	PAL8	Illegal indirect—an off-page reference was made; a link could not be generated because the indirect bit was already set.
IL	FLAP	A literal was used in an instruction which cannot accept one.
ILLEGAL*	DIRECT FOTP	An asterisk (*) was included in the output file specification or an illegal * was included in the input file name.
ILLEGAL * OR ?	CCL MCPIP	An * or ? was used in a CCL command that does not accept the wild card construction.
ILLEGAL?	DIRECT FOTP	A question mark (?) was included in the output file specification.
ILLEGAL ARG.	Keyboard Monitor	The SAVE command was not expressed correctly; illegal syntax used.
ILLEGAL ARITHMETIC EXPRESSION	FORT	Self-explanatory.
ILLEGAL BINARY INPUT, FILE #n	PIP	Self-explanatory; n is the number of the file in the input list.
ILLEGAL CONSTANT	FORT	Self-explanatory.
ILLEGAL CONTINUATION	FORT	Self-explanatory.

Message	Program	Explanation
ILLEGAL EQUIVALENCING	FORT	Self-explanatory.
#ILLEGAL INPUT	BATCH	A file specification designated TTY or PTR as an input device when the initial dialogue indicated that an operator is not available. The current job is aborted, and BATCH scans the input file for the next $JOB command record.
ILLEGAL OR EXCESSIVE DO NESTING	FORT	Self-explanatory.
ILLEGAL ORIGIN	Loader	A RALF routine tried to store data outside the bounds of its overlay.
ILLEGAL SPOOL DEVICE	BATCH	The device specified as a spooling output device must be file-structured. Control returns to the Command Decoder.
ILLEGAL STATEMENT	FORT	Self-explanatory.
ILLEGAL STATEMENT NUMBER	FORT	Self-explanatory.
ILLEGAL SYNTAX	CCL Command Decoder MCPIP	The command line was formatted incorrectly.
ILLEGAL VARIABLE	FORT	Self-explanatory.
IN	BRTS	Inquire failure in opening file. Device not found.
INCOMPATIBLE!	ABSLDR	The versions of ABSLDR and the Keyboard Monitor being used are incompatible.
?INPUT ERROR	RESORC	An input error occurred during a RESORC operation.
INPUT ERROR	CREF MCPIP	An input error occurred while reading the file.
	FRTS	Illegal character received as input.
	LIBSET	Parity error on input.

Message	Program	Explanation
INPUT ERROR, FILE # n	PIP	An input error occurred while reading file number n in the input file list.
INPUT ERROR READING INDIRECT FILE	CCL	CCL cannot read the file specified with the @ construction.
#INPUT FAILURE	BATCH	Either a hardware problem prevented BATCH from reading the next record of the input file, or BATCH read the last record of the input file without encountering a $END command record.
INSUFFICIENT CORE FOR BATCH RUN	BATCH	OS/8 BATCH requires 12K of core to run. Control returns to the OS/8 Monitor.
IO	BCOMP	I/O error.
	BRTS	TTY input buffer overflow. Causes input buffer to be cleared and output another ? (NF).
	FLAP RALF	Input/output error (fatal error).
	FORT	A device handler has signalled an I/O FORT error.
IOER	FORT Library	One of the following has occurred: 1. Device independent input or output attempted without /I or /O options, or user attempted to specify a device requiring a two-page handler for device-independent I/O without using the /H option. 2. Bad arguments to IOPEN or OOPEN, or 3. Transmission error while doing I/O.
I/O ERR	BUILD	An error occurred while reading from an input device during a LOAD command.

Message	Program	Explanation
I/O ERROR	FRTS	Error reading or writing a file, tried to read from an output device, or tried to write on an output device.
	PIP10	I/O device error; e.g., parity, write lock, out of paper.
	EPIC	If EPIC encounters an error while reading or writing a mass storage device, or a paper tape read fails three consecutive times, it outputs this error message, deletes the output file if one exists, and returns to the Command Decoder.
I/O ERROR, FILE #n	ABSLDR BITMAP	An I/O error has occurred in input file number n.
IO ERROR IN (file name) —CONTINUING	PIP	An error has occurred during a /S transfer.
I/O ERROR ON SYS:	CCL	An error occurred while doing I/O to the system device. The system must be restarted at 7600 or 7605 (see Restarting OS/8 in the Getting On Line with OS/8 section of Chapter 1). Do not press CONT, as that will surely cause further errors.
I/O ERROR TRYING TO RECALL	CCL	An I/O error occurred while CCL was trying to remember an argument.
IP	PAL8	Illegal pseudo-op—a pseudo-op was used in the wrong context or with incorrect syntax.
IR	FLAP	Invalid reference in a PDP-8 instruction.
IX	FLAP RALF	An index register was specified for an instruction which cannot accept one.

Message	Program	Explanation
IZ	PAL8	Illegal page zero reference—The pseudo-op was found in an instruction which did not refer to page zero. The Z is ignored.
L	SABR	/L or /G option was indicated, but the LOADER.SV file does not exist on the system device.
LD	PAL8	The /L or /G options have been specified and ABSLDR is not present on the system.
LG	PAL8	Link Generated—only printed if the /E switch was specified to PAL8.
LI	F4	Argument of logical IF is not type Logical.
LIBRARY DIRECTORY OVERFLOW	LIBSET	Too many subroutines were specified.
LINE TOO LONG IN FILE #n	PIP	In ASCII mode, a line has been found greater than 140 characters.
LM	BRTS	Attempt to take log of negative number or 0.
LOADER I/O ERROR	Loader	Fatal error message indicating that an error was detected by OS/8 while trying to perform a USR function.
LS	BCOMP	Missing equal sign in LET statement.
LT	BCOMP	Statement too long (greater than 80 characters).
	F4	Input line too long, too many continuations.
	FLAP RALF	The line is longer than 128 characters. The first 127 characters are assembled and listed.
L/T ERROR	EPIC	EPIC was expecting leader/trailer and found non-leader trailer while attempting to

Message	Program	Explanation
		read a block. The program prints this error message and halts with AC=7777 to allow the user to reposition the tape then press the CONT key.
M	SABR	A symbol is multiply-defined. Listings of programs with multiple definitions have unmarked errors.
#MANUAL HELP NEEDED	BATCH	BATCH is attempting to operate an I/O device, such as PTR or TTY, that will require operator intervention.
MD	BCOMP	Line number defined more than once. YY equals the line number before line in error.
	FLAP RALF	The tag on the line has been previously encountered at another location or has been used in a context requiring an absolute expression.
ME	BCOMP	Missing END statement.
MIXED INPUT	Loader	The L option was specified on a line that contained some file other than a library file. The library file (if any) is accepted. Any other input file specification is ignored.
MIXED MODE EXPRESSION	FORT	Self-explanatory.
MK	F4	Misspelled keyword.
ML	F4	Multiply-defined line number.
MM	F4	Mismatched parenthesis.
MO	BCOMP F4	Operand expected, not found.
MONITOR ERROR 2 AT xxxx (DIRECTORY I/O ERROR)	Keyboard Monitor	Attempt made to output to a WRITE-LOCKed device usually DECtape; or an error has occurred reading/writing a directory.

Message	Program	Explanation
MONITOR ERROR 5 AT xxxx (I/O ERROR ON SYS=)	Keyboard Monitor	An error occurred while doing I/O to the system device. This error is normally the result of not WRITE-ENABLEing the system device.
MONITOR ERROR 6 AT xxxx (DIRECTORY OVERFLOW)	Keyboard Monitor	A directory overflow has occurred (no room for tentative file entry in directory).
#MONITOR OVERLAYED	BATCH	The Command Decoder attempted to call the BATCH monitor to accept and transmit a file specification, but found that a user program had overlayed part or all of the BATCH monitor. Control returns to the monitor level, and BATCH executes the next Keyboard Monitor command.
MORE CORE REQUIRED	FRTS	The space required for the program, the I/O device handlers (I/O buffers) and the resident Monitor exceeds the available core.
MP	BCOMP	Missing parenthesis or error in expression within parentheses.
MT	BCOMP F4	Operand of mixed type or operator does not match operands.
MULT SECT	Loader	Any combination of entry point, COMMON section (with the exception of multiple COMMONs) or program section of the same name causes this error, except as shown in the Table 8-6.
?NAEM	BUILD	A device or filename was not designated in a command that requires one to be present.

Message	Program	Explanation
NE	FLAP RALF	Number error. A number out of range was specified or an 8 or 9 occurred in octal radix.
NEED:n1FOUND:n2	EPIC	EPIC read block n2 of the file when it was expecting block n1 of the file. EPIC halts with AC+7777 to allow the user to reposition the paper tape.
NEED:name1 FOUND name2	EPIC	EPIC read a block of tape for the file NAME2 when it was expecting a block of the file NAME1.
NF	BCOMP	NEXT statement without corresponding FOR statement.
NM	BCOMP	Line number missing after GOTO, GOSUB, or THEN.
NO!!	Keyboard Monitor	The user attempted to start (with .ST) a program which cannot be started.
NO CCL!	Keyboard Monitor	CCL.SV is not present on the system device or an I/O error occurred on the file. Refer to the Getting occurred while trying to read On Line section of Chapter 1 for instructions on loading programs onto the system device.
NO DEFINE FILE	FRTS	Direct access I/O attempted without a DEFINE FILE statement.
NO END STATEMENT	FORT	The input to the compiler has been exhausted.
NO FILES OF THE FORM xxxx	FOTP	No files of the form (xxxx) specified were found on the current input device group.
NO /I	BITMAP	Cannot produce a bitmap of an image file.
NO /I!	ABSLDR	Use of /I is prohibited at this point.

Message	Program	Explanation
NO INPUT	ABSLDR BITMAP	No input or binary file was found on the designated device.
NO INPUT FILE	MCPIP	No input file was specified when one was required.
NO MAIN	LOADER	No RALF module contained section #MAIN.
NO NUMERIC SWITCH	FRTS	The referenced FORTRAN I/O unit was not specified to the run-time system.
NO OUTPUT FILE	MCPIP	No output file was specified when one was required.
?NO ROOM	BUILD	Too many device handlers were present on the system when a LOAD or BUILD command was typed. The UNLOAD command must be used to remove a handler before another can be loaded.
NO ROOM FOR OUTPUT	FORT	The file FORTRN.TM cannot fit on the system device.
NO ROOM FOR OUTPUT FILE	DIRECT PIP	Either room on device or room in directory is lacking.
NO ROOM IN (file name) —CONTINUING	PIP	Occurs during use of the /S option. The output device cannot contain all of the files on the input device.
NO ROOM, SKIPPING (filename)	FOTP	No space is available on the output device to perform the transfer. Predeletion may already have occurred.
NO SUCH DEVICE	PIP10	Device name used is not legal in this OS/8 system.
% NON SYSTEM DEVICE	RESORC	The input device specified in a RESORC command line is not an OS/8 system device.
NOT A LOADER IMAGE	FRTS	The first input file specified to the run-time system was not a loader image file.
% NOT A SYSTEM HEAD	RESORC	The filename specified is not a system-head file.

Message	Program	Explanation
name NOT AVAILABLE	Keyboard Monitor	The device with the name given is not listed in any system table, or it is not available for use at the moment, or the user tried to obtain input from an output-only device.
NOT ENOUGH CORE	CCL	The number specified in a CORE command is larger than the number of 4K core banks on the system.
name NOT FOUND	BUILD CCL Command Decoder Keyboard Monitor	The device or file name designated in the command was not found.
file NOT FOUND	MCPIP	The file specified cannot be found.
NOT OS8 FILE	PIP10	The output device specified with a /L or /F option was not an OS/8 device or file.
NOT PDP-10 FILE	PDP10	The output device specified with a /Z option was not a DECsystem-10 tape, or the input device specified with a /L or /F option was not a DECsystem-10 tape.
? NUMBER TOO BIG	CAMP	The "nnnn" specified in a BACKSPACE or SKIP command is greater than 4095.
OE	BRTS	Driver error while overlaying. Caused by SYS device hardware error.
OF	BCOMP	Output file error.
	F4	Error writing output file. Control returns to the Keyboard Monitor.
OP	F4	Illegal operator.
OS/8 ENTER ERROR	Loader	Fatal error message indicating that an error was detected by OS/8 while trying to perform a USR function.

Message	Program	Explanation
OT	F4	Type/operator use illegal (e.g., A.AND.B where A and/or B not typed Logical).
OUT DEV FULL	CREF	The output device is full (directory devices only).
OUT-IN	MCPIP	Both the input and the output devices were specified as the same cassette or magnetic tape drive.
?OUTPUT DEVICE FULL	RESORC	The output device specified does not have enough room to copy the RESORC file.
OUTPUT DEVICE FULL	MCPIP	Either room on device or room in the directory is lacking.
?OUTPUT DEVICE IS READ ONLY	RESORC	The output device specified is a read-only device; e.g., PTR.
?OUTPUT ERROR	RESORC	An error occurred while attempting to output during a RESORC operation.
OUTPUT ERROR	MCPIP	Output error—possibly a WRITE-LOCKed device, parity error, or attempt to output to a read-only device.
OUTPUT FILE OPEN ERROR	PIP10	The output file could not be opened. Check output directory to ensure that enough space exists on the device.
OV	BRTS	Numeric or input overflow.
OVER CORE	Loader	The loader image requires more than 32K of core memory.
OVER IMAG	Loader	Output file overflows in the loader image file.
OVER SYMB	Loader	Symbol table overflow. More than 253 (decimal) symbols in one FORTRAN job.
OVERFLOW	FRTS	Result of a computation exceeds upper bound for that class of variable. The result

Message	Program	Explanation
		is set equal to zero and execution continues. This error is detected only if an FPP is present.
OVERLAY ERROR	FRTS	Error while reading overlay.
OVFL	FORT Library	Floating point overflow; very large number is returned.
PA	BRTS	Illegal argument in POS function.
PARENS TOO DEEP	FRTS	Parens nested too deeply in FORMAT statement.
PARITY ERROR	EPIC	EPIC failed to read a block correctly; e.g., the reader dropped some bits. EPIC halts with AC=7777 to allow the user to reposition the tape so that it can try the read again.
PD	BCOMP	Pushdown stack overflow. Result of either too complex a statement (or statements) or too many nested FOR-NEXT loops.
	F4	Compiler stack overflow; statement too big and/or too many nested loops.
PE	PAL8	Current non-zero page exceeded—an attempt was made to:
PH	F4	Bad program header line.
	PAL8	Phase error—a conditional assembly bracket is still in effect at the end of the input stream—this is caused by nonmatching < and > characters in the source.
PIP10 CANNOT BE CHAINED	PIP10	Self-explanatory.
?PLAT	BUILD	The =n in a SYS command is too large for the device specified; e.g., RF08=5.

Message	Program	Explanation
PO	FLAP	Page overflow. Literals and instructions have been overlapped.
PREMATURE END OF FILE, FILE #n	PIP	Message occurs in Binary Mode (/B) only. A physical end-of-file has been found before the final leader/trailer.
PTR:name IS TOO BIG FOR dev:	EPIC	The paper tape file name will not fit on the specified output device DEV:. EPIC aborts the command and returns to the Command Decoder. EPIC makes the check for size before writing on the output device.
QL	F4	Nesting error in EQUIVALENCE statement.
QS	BCOMP	String literal too long or does not end in quote.
	F4	Syntax error in EQUIVALENCE statement.
RD	F4	Attempt to redefine the dimensions of a variable.
	PAL8	Redefinition — a permanent symbol has been defined with =. The new and old definitions do not match. The redefinition is allowed.
RE	BRTS	Attempt to read past end of file (NF).
	RALF	Relocatability error. A relocatable expression has been used in context requiring an absolute expression.
RECORD SIZE TOO BIG	MCPIP	The output record size specified is greater than 1000 or an output record size is 0.
RT	F4	Attempt to redefine the type of variable.
RW	F4	Syntax error on READ/WRITE statement.

Message	Program	Explanation
S	SABR	Either the symbol table has overflowed, common storage has been exhausted, more than 64 different user-defined symbols occurred in a core page, or more than 64 external symbols have been declared. Could also indicate a system error such as overflowed output file.
SABR.SV NOT FOUND	FORT	The SABR assembler is not present on the system device.
SAVE ERROR	Keyboard Monitor	An I/O error has occurred while saving the program. The program remains intact in core.
SC	BRTS	String too long (greater than 72 characters) after concatenating.
SE	PAL8	Symbol table extended—too many symbols have been defined for the amount of memory available. Fatal error—assembly cannot continue.
SF	F4	Bad arithmetic statement function.
SL	BRTS	String too long or undefined.
SN	F4	Illegal subroutine name in CALL.
SORRY—NO INTERRUPTIONS	PIP	1. ↑C (CTRL/C) is typed while compressing a file onto itself; the transfer continues. 2. A /Y transfer is done with system device as the output, or if the transfer has both input and output on the same device.
#SPOOL TO FILE BTCHA1	BATCH	Where the "A" may be any character of the alphabet and the "1" may be any decimal digit. This message indicates that BATCH has intercepted

Message	Program	Explanation
		a non-file structured output file and rerouted it to the spool device. This is not, generally, an error condition.
SQRT	FORT Library	Attempt to take square root of negative number; absolute value used.
SR	BRTS	Attempt to read string from numeric file.
SS	BCOMP	Subscript or function argument error.
	F4	Error in subscript expression; i.e., wrong number, syntax.
ST	BCOMP	Symbol table overflow due to too many variables, line number, or literals. Combine lines using backslash (/) to condense program.
ST	BRTS	String truncation on input. Stores maximum length allowed (NF).
ST	F4	Compiler symbol table full, program too big. Causes an immediate return to the Keyboard Monitor.
	FLAP RALF	User symbol table overflow (fatal error).
SU	BRTS	Subscript out of DIM statement range.
SUBR. OR FUNCT. STMT. NOT FIRST	FORT	Self-explanatory.
% SUPERSEDED	CCL	The file specified in a MAKE command already exists. This is a warning message indicating that the file is being replaced.
SW	BRTS	Attempt to write string into numeric file.
SWITCH NOT ALLOWED HERE	CCL	Either a CCL option was specified on the left side of

Message	Program	Explanation
		the < or was used when not allowed.
SY	BCOMP	System incomplete. System files such as BASIC.SV, BCOMP.SV, and BRTS.SV missing.
	F4	System error; i.e., PASS20. SV or PASS2.SV missing, or no room for output file. Causes an immediate return to the Keyboard Monitor.
SYM OVERFLOW	CREF	More than 896 (decimal) symbols and literals were encountered.
SYMBOL TABLE EXCEEDED	FORT	Self-explanatory.
?SYNTAX	BUILD	An illegal character was typed in a BUILD command line. The line must be retyped.
?SYNTAX ERROR	CAMP	An illegal character was typed in a CAMP command or a command was formatted incorrectly. The command must be retyped.
SYNTAX ERROR	FORT PIP 10	Invalid command line.
?SYS	BUILD	This message appears when one of the following conditions exists: a. A permanent name in a SYS command was not a system handler or coresident with one. b. A BOOT command was issued when two system handlers were active. c. A BOOT command was issued with a active handler which must be coresident with a SYS handler did not have the system handler active.

Message	Program	Explanation
#SYS ERROR	BATCH	A hardware problem prevented BATCH from performing an I/O operation.
SYS ERROR	BUILD	An I/O error occurred with a system handler. The computer halts. Press CONT to retry or restart the BUILD procedure from the beginning. Do not assume that a valid OS/8 system remains in core.
SYS NOT FOUND	BUILD	No active handler with the name SYS was present when a BOOTSTRAP command was issued.
SYSTEM DEVICE ERROR	FRTS	I/O failure on the system device.
SYSTEM ERR	Keyboard Monitor	An error occurred while doing I/O to the system device. The system should be restarted at 7600 or 7605. Do not press CONTinue as this is sure to cause further errors.
SYSTEM ERROR	Loader	Fatal error message indicating that an error was detected by OS/8 while trying to perform a USR function.
SYSTEM ERROR— CLOSING FILE	FOTP	Self-explanatory.
TB	BCOMP	Program too big. Condense or CHAIN.
TD	BCOMP	Too much data in program.
	F4	Bad syntax in type declaration statement.
THERE IS NO HOPE— THERE IS NO TTY HANDLER IN YOUR SYSTEM!	DIRECT	A command was issued to print a directory on the terminal when no TTY handler is present on the OS/8 system. Use BUILD to insert a TTY handler in the system.

Message	Program	Explanation
TOO FEW ARGS	Keyboard Monitor	An important argument has been omitted from a command.
TOO MANY FILES	CCL	Too many files were included in a CCL command.
TOO MANY FILES	Command Decoder	More than three output files or nine input files were specified. Some programs may restrict the user to fewer files.
TOO MANY FILES	MCPIP	More than 1 output device was specified or more than 1 input device was specified.
TOO MANY HANDLERS	FRTS	Too many I/O device handlers are resident in memory, or files have been defined on too many devices.
TOO MANY LEVELS	Loader	The 0 option was specified more than 7 times.
TOO MANY OVERLAYS	Loader	More than 16 overlays were defined in the current level.
TOO MANY RALF FILES	Loader	More than 128 input files were specified.
TS	BCOMP	Too many total characters in the string literals.
?TTY DOES NOT EXIST	RESORC	An output device was not specified in the RESORC command line and the TTY handler does not exist on the OS/8 system. See the BUILD section of Chapter 2 for instructions on inserting TTY handlers.
U	SABR	No symbol table is being produced, but there is at least one undefined symbol in the program.
UD	BCOMP	Error in UDEF statement.
UF	BCOMP	FOR loop without corresponding NEXT statement.
UNDF	SABR	Undefined symbol; printed in the symbol table listing.

Message	Program	Explanation
UNIT ERROR	FRTS	I/O unit not assigned, or incapable of executing the requested operation.
UO	PAL8	Undefined origin—an undefined symbol has occurred in an origin statement.
US	BCOMP F4	Undefined statement number.
	FLAP RALF PAL8	Undefined symbol in an expression.
USE PIP FOR NON-FILE STRUCTURED DEVICE	FOTP	An input device specified is not a file-structured device; e.g., PTR.
USER ERROR	FRTS	Illegal subroutine call, or call to undefined subroutine. Execution continues only if the E option was requested.
USER ERROR 0 AT xxxx	Keyboard Monitor	An input error was detected while loading the program. xxxx refers to the Monitor location where the error was generated.
USER ERROR 1 AT xxxx	FORT Library	The user tried to reference an entry point of a program which was not loaded, or he failed to define a subscripted variable in a DIMENSION statement. xxxx has no meaning.
USR n dev:name	EPIC	The USR encountered an error while attempting to perform a fetch, lookup, enter or close on the file name on device dev. n=1 is a fetch, n=2 is lookup, n=3 is enter, n=4 is close. EPIC aborts the command and returns to the Command Decoder.
UU	BCOMP	Incorrect or missing array designator in USE statement.

Message	Program	Explanation
VE	F4	Version error. One of the compiler programs is absent from SYS: or is present in the wrong version.
VR	BRTS	Attempt to read variable length file.
WE	BRTS	Attempt to write past end of file (NF).
WRONG OS/8 MONITOR	BATCH	OS/8 BATCH requires an OS/8 Monitor no older than version 3.
XC	BCOMP	Extra characters after the logical end of line.
XS	RALF	External symbol table overflow. Control returns to the OS/8 Keyboard Monitor.
ZE	PAL8	Page 0 exceeded—same as PE except with reference to page 0.
ZERO SYS?	PIP	If any attempt is made to zero the system device directory, this message occurs. Responding with Y causes the directory to be zeroed; any other character aborts the operation.

appendix F

OS/8 file name extensions

This appendix lists the file name extensions used in OS/8.

Extension	Meaning
.BA	BASIC source file (default extension for a BASIC input file).
.BI	Batch input file.
.BK	Backup ASCII file (default extension for a TECO output file).
.BN	Absolute binary file (default extension for ABSLDR, BUILD, and BITMAP input files; also used as default extension for PAL8 binary output file).
.DA	Data file.
.DC	Documentation file.
.DI	Directory listing.
.FT	FORTRAN language source file (default extension for FORT input files).
.HL	Help file (default extension for HELP input files).
.LD	F4 load mode (default assumed by run-time system, F4 loader).
.LS	Assembly listing output file (default extension for PAL8 and SABR).
.MA	Macro source file.
.MP	File containing a loading map (used by the Linking Loader).
.PA	PAL8 source file.
.RA	RALF assembly language file.

Extension	Meaning
.RB	Relocatable binary source file.
.RL	Relocatable binary file (default extension for a Linking Loader input file; also used as the default extension for an 8K SABR output file).
.SB	8K SABR source file.
.SV	Core image file or SAVE file; appended to a file name by the R, RUN, SAVE, and GET Keyboard Monitor commands.
.SY	System head.
.TE	TECO macro file (default extension for a MUNG input file).
.TM	Temporary file generated by FORTRAN or SABR for system use (default extension for CREF input files and PAL8 output files).
.TX	Text files.

appendix g

OS/8 device handlers

The device handlers supplied with the OS/8 system have certain operating characteristics which the user should understand. Most of these are extremely simple and require no action by the user. Some device handlers perform additional operations for the user when I/O is being performed on a given device. This appendix gives a brief description of the OS/8 device handlers. See the *OS/8 Software Support Manual* (DEC-S8-OSSMB-A-D) for more detailed information concerning device handlers.

HIGH-SPEED READER/PUNCH

The device handler for the high-speed paper tape reader, before reading a tape, prints an uparrow (↑) and waits for the user to type any single character at the keyboard. This gives the user time to check the reader to ensure that the tape is loaded correctly, and it facilitates reading multiple tapes, e.g., a PAL8 source tape must be loaded three times for the three passes of the assembler. Characters are read from the paper tape and packed into an input buffer. The end of the paper tape or a full input buffer causes the buffer to be made available to the user program. Typing CTRL/C while the tape is moving causes a return to the Keyboard Monitor.

The handler for the high-speed paper tape punch unpacks characters from the output buffer and punches them on paper tape. Typing CTRL/C causes a return to the Keyboard Monitor. The punch must be manually turned on before an attempt is made to output to that device.

LOW-SPEED READER/PUNCH

In addition to the handler for the high-speed reader/punch, a similar handler is available for the ASR-33 Teletype low-speed reader/punch. This handler allows users not having high-speed I/O to read and punch binary format tapes. (The standard TTY handler cannot be used for binary format tapes, as the binary

format can appear as control characters to the handler.) The operation of this handler is exactly the same as that for the high-speed reader/punch except that the uparrow is not printed.

TTY HANDLERS

There are two TTY (console terminal) handlers available: a 1-page handler and a 2-page handler. Both handlers perform I/O transfers between the terminal keyboard and an input buffer, or between an output buffer and the terminal.

The 1-page handler echoes all terminal input and performs a line feed operation after any typed carriage return. A CTRL/O typed while output is being printed terminates printing of the current output buffer. A CTRL/C typed at any time during input or output causes a return to the Keyboard Monitor. Typing CTRL/Z as input terminates input and gives an end-of-file indication to the calling program. The TTY handler should *not* be used to read binary tapes from the low-speed reader.

The 2-page TTY handler may be used only to read or write ASCII files; results are unpredictable with non-ASCII files. In addition to the features included in the 1-page handler, this handler includes the use of the RUBOUT key to delete the previous character and echo it either as a backslash (\) or as the character rubbed out, the use of CTRL/U to delete the current line, and the use of the TAB key to output the correct number of spaces to bring the text to the start of the next tab stop.

The 2-page TTY handler also includes approximately 30 free locations so that the user may conditionalize certain nonstandard features. See the *OS/8 Software Support Manual* for a complete list of these features.

LINE PRINTERS

The OS/8 line printer handler is a 1-page handler for the LP08, LS8E, and LV8 line printers. This handler performs a form feed operation before beginning an output task. The characters are unpacked from the output buffer and printed. A form feed is also produced following the completion of an output task. Typing CTRL/C while the line printer is in operation causes a return to the Keyboard Monitor. A CTRL/Z found in the output buffer causes printing to terminate and a form feed to be produced. Tabs and line overflow are handled; nulls are ignored.

Relative location 0 of this handler specifies the width of the line printer. This location may be patched using the ALTER command in BUILD. The location is set to the one's complement of the width desired. Initially, this location is set to 7573 (octal) which corresponds to a 132-column printer. For example, to indicate an 80-column printer, location 0 should be set to 7657 (octal).

VR12 SCOPE

The VR12 scope handler for OS/8 (running on a PDP-12) displays characters on the VR12 scope on both channels. When the scope is full, the handler stops reading characters from the buffer and displays what is known as a scope page. The screen is considered full whenever the end of the buffer is reached, a CTRL/Z is encountered in text, or when the number of lines displayed become equal to the maximum number specified by the user. The user can advance to the next scope page by typing any character, other than CTRL/C.

When CTRL/C is typed, control returns to the Keyboard Monitor. Control does not return to the calling program until a character is typed at a point when the handler is displaying the last scope page of a particular buffer load.

To use the VR12 handler, the user sets the number of lines desired in a single scope page via the switch register (right switches). The switch register is set to the negative of the number of lines to be displayed in a scope page. When text reaches the right margin of the scope face, it is continued on the next physical line of the scope.

A line feed or form feed character causes succeeding text to continue on the next physical line. Carriage return characters have no effect on the display.

CARD READER

The device handler for the card reader reads cards in alphanumeric format from either a punched card reader or an optical mark card reader. Card format can have up to 80 characters per card; trailing blanks are deleted from each card. Blank cards cause a carriage return/line feed to be entered into the data stream. Typing CTRL/C while cards are being read terminates reading and returns control to the Keyboard Monitor. Typing CTRL/Z terminates further reading and performs as though an end-of-file card was read. (An end-of-file card contains a ← character in

column 1 (0-8-5 punch) with the remaining columns blank. Either CTRL/Z or the end-of-file card is necessary to terminate reading.) It is not possible to RUN or GET a program from the card reader as these commands assume a directory device.

DECTAPES

Any DECtape other than the system device (if the system is a DECtape system) can be interrupted with a CTRL/C, returning control to the Keyboard Monitor. DECtape unit 0 on a DECtape system must never be WRITE LOCKed while operating OS/8.

MAGNETIC TAPE

The handler for magnetic tape reads and writes either 7- or 9-channel magnetic tape with odd parity at 800 bpi. This handler is non-file structured but may be altered by the user to read and write files. CTRL/C returns control to the Keyboard Monitor but its use is not recommended since it leaves the tape without an end-of-file indicator.

CASSETTES

The cassette handler performs character I/O transfer between the cassettes and the buffer. It treats cassettes as non-file structured devices. Data appears on cassette in 192-byte records. Typing CTRL/C returns control to the Keyboard Monitor.

BATCH HANDLER

The OS/8 batch handler is used from a BATCH job to read from the BATCH stream. This is a 1-page handler for read-only, non-file structured devices. If this handler is used when BATCH is not running, it generates a fatal error. The BATCH handler reads characters from the BATCH stream, ignoring line feeds, and creating a line feed after a carriage return. When the handler encounters a line beginning with a dollar sign, it pads the buffer with CTRL/Z and nulls, and takes the end-of-file return.

DSK AND SYS

The DSK and SYS device handlers work automatically without any user intervention.

appendix h

obtaining os/8 program version numbers

When the user receives new OS/8 software or when he wishes to report problems with the software, he must know the version number of the OS/8 program in question. Most OS/8 system programs have version numbers that can be obtained by typing a command to the OS/8 Command Decoder * or to the called program. Some system programs print the version number at the beginning of the output listing. The following table shows how to obtain version numbers for most OS/8 system programs.

Program	How to Obtain Version Number
ABSLDR	Internal only.
BASIC	Printed in program heading.
BATCH	Type /V in BATCH command string.
BITMAP	Printed at top of output listing.
BOOT	Type VE to the / printed by BOOT.
BUILD	Type VE to the $ printed by BUILD.
CAMP	Type VE to the # printed by CAMP.
CCL	Type VER to the Keyboard Monitor.
Command Decoder	Internal only.
CREF	Printed at end of CREF output listing.
DIRECT	Type /W to the * printed by DIRECT.
EDIT	Type # to the # printed by EDIT.
EPIC	Internal only.
F4 Compiler	Printed in heading of output listing.
F4 Loader (LOAD)	Printed in heading of loading map.
FLAP	Printed in heading of output listing.
POTP	Type /W to the * printed by FOTP.
FRTS	Type /V to the * printed by FRTS (to be implemented later).

Program	How to Obtain Version Number
Keyboard Monitor	Type VER to the Keyboard Monitor.
MCPIP	Type /V to the * printed by MCPIP.
ODT	Internal only.
PAL8	Printed in heading of output listing.
PIP	Type /V to the * printed by PIP.
PIP10	Printed in heading of directory listing.
RALF	Printed at heading of output listing.
RESORC	Type /V to the * printed by RESORC.
SRCCOM	Printed in heading of output listing.
TECO	Type CTRL/V to the * printed by TECO.

index

C

E

F

J

K

L

M

Q

R

S

T

DIGITAL EQUIPMENT CORPORATION, Maynard, Massachusetts, Telephone: (617) 897-5111 • ARIZONA, Phoenix • CALIFORNIA, Sunnyvale, Santa Ana, Los Angeles, Oakland, San Diego and San Francisco (Mountain View) • COLORADO, Denver • CONNECTICUT, Meriden • DISTRICT OF COLUMBIA, Washington (Riverdale, Md.) • FLORIDA, Orlando • GEORGIA, Atlanta • ILLINOIS, Chicago • INDIANA, Indianapolis • LOUISIANA, New Orleans • MASSACHUSETTS, Cambridge and Waltham • MICHIGAN, Ann Arbor and Detroit (Southfield) • MINNESOTA, Minneapolis • MISSOURI, St. Louis • NEW JERSEY, Englewood, Metuchen, Parsippany and Princeton • NEW MEXICO, Albuquerque • NEW YORK, Centereach (L.I.), Manhattan, Syracuse and Rochester • NORTH CAROLINA, Durham/Chapel Hill • OHIO, Cleveland and Dayton • OKLAHOMA, Tulsa • TEXAS, Dallas and Houston • UTAH, Salt Lake City • WASHINGTON, Seattle • WISCONSIN, Milwaukee • ARGENTINA, Buenos Aires • AUSTRALIA, Adelaide, Brisbane, Melbourne, Perth and Sidney • AUSTRIA, Vienna • BELGIUM, Brussels • BRAZIL, Rio de Janeiro, São Paulo and Porto Alegre • CANADA, Calgary, Alberta; Vancouver, British Columbia; Ottawa and Toronto, Ontario; and Montreal, Quebec • CHILE, Santiago • DENMARK, Copenhagen • FINLAND, Helsinki • FRANCE Grenoble and Paris • GERMANY, Cologne, Hannover, Frankfurt, Munich and Stuttgart • INDIA, Bombay • ITALY, Milan • JAPAN, Tokyo • MEXICO, Mexico City • NETHERLANDS, The Hague • NEW ZEALAND, Auckland • NORWAY, Oslo • PHILIPPINES, Manila • PUERTO RICO, Miramar • SPAIN, Barcelona and Madrid • SWEDEN, Stockholm • SWITZERLAND, Geneva and Zurich • UNITED KINGDOM, Birmingham, Edinburgh, London, Manchester and Reading • VENEZUELA, Caracas

digital

1802 • 00574 • 3384
JH • 09 • 30

Printed in U.S.A.