Prime Computer User Guide

For

Program Development Software

PRIME COMPUTER, INC., 145 PENNSYLVANIA AVENUE, FRAMINGHAM, MA. 01701

MAN 1879

PROGRAM DEVELOPMENT

SOFTWARE

USER GUIDE

December 1974

PRIME
COMPUTER, INC.

[145 Pennsylvania Ave., Framingham, Mass. 01701]

First Printing December 1974

THUMB INDEX

CONTENTS

ILLUSTRATIONS

TABLES

FOREWORD

This user guide describes the Prime Program development software used to generate, compile or assemble, load and debug FORTRAN or assembly language application programs. It consists of the following sections:

Section 1	Introduction
Section 2	Editors (text, box and binary)
Section 3	FORTRAN Compiler
Section 4	Prime Macro Assembler (PMA)
Section 5	Linking Loader
Section 6	Debug Aids (TAP and PSD)
Section 7	Paper Tape Utilities (MDL and PTCPY)
Section 8	Library Subroutines
Section 9	Paper Tape Program Development

Information in this guide applies both to systems with operating system and file system support and to stand-alone systems using paper tape input/output. Information in this user guide supersedes the versions previously printed in the Prime Operator's Guide.

SECTION 1

INTRODUCTION

SCOPE

This user guide contains detailed reference information on the editors, translators and utilities that are the essentail items of Prime program development software. This family of software is required to compose the source file of a FORTRAN or assembly language program, compile or assemble it, load the resulting object file and related library routines, and simulate execution and debug the result.

Related Publications

The following Prime documents should be available for reference:

	Document No.
Prime CPU Operator's Guide (control panel and paper tape device operation)	MAN 1672
Prime CPU System Reference Manual (instruction set, addressing modes, input/output programming)	MAN 1671
Macro Assembler User Guide (assembly language conventions, pseudo-operations, macro facility)	MAN 1673
FORTRAN IV Language User Guide (FORTRAN source program requirements	MAN 1674
DOS-DOS/VM User Guide (Operating system keyboard commands and description of file system)	MAN 1675
Prime Software Library User Guide (calling formats and functional description of all library subroutines)	MAN 1880

Effects of Operating System

All programs described in this manual operate in the same way whether the user is operating with file system support under DOS, DOS/VM or RTOS/VM, or operating in a stand-alone system using paper tape input/output. The only difference is in the way a program is loaded and started. (Paper tape users must mount paper tapes as required, and there are a few command restrictions owing to the absence of a file system.)

SUMMARY OF PROGRAM DEVELOPMENT SOFTWARE

The following paragraphs summarize the main functions of the program development software as used in a DOS or DOS/VM environment with file system support. For another summary emphasizing paper tape functions, see Section 9.

Editors (Section 2)

The Text Editor, ED, is the basic tool for new program development. This program permits source programs to be composed, edited, and listed at the user terminal. After entering a rough copy of the program, the programmer can locate and alter text strings, correct spelling, syntax, or spacing errors, or move lines from one place to another by simple keyboard commands.

Sections of the program can be printed for checking, or the entire program can be listed. When the program is complete and ready to be assembled or compiled, it is filed in the user's UFD. Other source files can be read in, as well, to be expanded or merged with the current version of the program.

A shorter version of the editor, EDLIN, excludes the box mode which simplifies generation of graphic or pictorial layouts. The box editing feature is not applicable to program development.

The binary editor, BINED, operates on object modules containing library subroutines. It is useful for examining the contents of library tapes or building custom libraries.

Macro Assembler (Section 3)

Source programs in the Macro assembly language are processed by the Macro Assembler program to form object program files. The assembler is invoked by the PMA external command. The assembler reads the source file and translates the symbolic codes of the source program into the binary bit patterns required by the loader. This two-pass assembler reads the source file twice - the first time to build a table of all symbolic addresses used, and the second time to translate the mnemonic expressions into an object program file. An optional listing file shows both the source symbolic code and the translated binary equivalent of each entry.

FORTRAN IV Compiler (Section 4)

Source programs in the FORTRAN IV language are processed in the same way as assembly language programs. After the FORTRAN Compiler is invoked by the FTN external command, it controls a one-pass reading of the source program file. The output object file is similar in format to the assembly language output file. An optional listing file, either a straight listing of the source statements or an expanded listing showing the assembly language breakdown of each statement, may also be created.

Linking Loader (Section 5)

Object files generated by the assembler or compiler require the Linking Loader to interpret and complete the addressing information. Indirect address links must be formed in sector zero (or another specified base sector) when address references happen to fall across sector boundaries.

Once the loader is invoked by the LOAD external command, it prints a prompt character and awaits commands from the user terminal. Through keyboard commands, the user can load main program or library files, specify addresses where loading is to start, define linkage areas for cross-sector address references, and do many special-purpose operations. The loader keeps track of instructions of the class which may be unimplemented in a particular machine, and automatically generates object code blocks to simplify loading of the appropriate segments of the VIP (Virtual Instruction Package) library.

The user can request the loader to print a memory map, which defines the memory areas occupied by the program and lists all subroutine calls and external references.

Once a program has been loaded by the Linking Loader, it is fully translated into 16-bit machine language codes and is ready to execute or be saved in SAVE file format.

Debug Aids (TAP or PSD) (Section 6)

During the early stages of program development and checkout, TAP and PSD permit the programmer to examine, alter, and list the content of memory locations in response to simple terminal keyboard commands. A "trace" function controls dynamic execution of object programs, with diagnostic printout of register contents at selected intervals (for example, whenever a specified effective address is formed).

Tape Punch and Copy Utilities (Section 7)

Memory Dump and Load (MDL): Loading of an object program and acompanying external library or subroutine programs is often a time consuming operation. MDL saves the result of a program building session by punching the entire loaded program on paper tape in the self-loading format. The program can be restored to the same memory area from which it was punched by using APL or the key-in loader. MDL uses the low- or high-speed punch.

Paper Tape Copy (PTCPY): This utility program uses the high-speed reader-punch to duplicate and verify paper tapes punched in any format (ASCII, object, or self-loading).

Library Subroutines (Section 8)

Prime supplies an extensive library of math and input/output subroutines to support the FORTRAN compiler and to supplement assembly language programs. Most of the library is in a single disk file, FTNLIB, which is automatically loaded by the loader's LIbrary command. Section 8 identifies the main library components available to the user. For full information on library subroutines and their use, refer to the Prime Software Library User Guide.

PROGRAM DEVELOPMENT EXAMPLE

In operation under DOS or DOS/VM, each of the programs described in this manual is invoked as an external command. (See DOS-DOS/VM manual for details.) For example, the following DOS keyboard dialog creates, loads and runs a simple FORTRAN program: (User input is underlined.)

```
OK;ED                          (Text editor is requested. OK; is the
GO                              DOS prompt message.)

INPUT                          (Editor is loaded and in input mode,
                                ready to accept FORTRAN program text.)

\ WRITE(1,10)                  (\ is form feed character - tabs to
                                column 6. One space then starts text
                                on column 7 as required by FORTRAN
                                compiler.)

10\ FORMAT('TEST MESSAGE')

\ CALL EXIT                    (Ensures safe return to operating system)

\ END

(CR)                           Carriage return switches to edit mode.

EDIT

TOP,PRINT 99                   User takes a look at file.

.NULL.
      WRITE(1,10)
10    FORMAT('TEST MESSAGE')
      CALL EXIT
      END
BOTTOM

FILE TEST                      User files it under name TEST.

OK;FTN TEST                    Control returns to DOS. User invokes
                                FORTRAN compiler to generate object file.

GO

NO ERRORS (FTN-1802.007)

OK; LOAD                       User invokes linking loader.
GO

$LO B←TEST                     Dollar sign is loader's prompt character.
                                B←TEST is name of binary (object) file
                                created automatically by compiler. Loading
                                starts at default value of '1000.
```

$LI LC	LI command automatically loads supporting subroutines from FORTRAN library. LC (load complete) indicates completion.
$MA	Load Map is requested.

```
*START  001000    *LOW    000200    *HIGH   006311    *PBRK   006312
*CMLOW  063753    *CMHGH  063753    *SYM    057420    *UII    000001

*BASE   000200  000212  000777  000777
*BASE   001464  001526  001525  001525
*BASE   002447  002511  002510  002510
*BASE   003335  003360  003365  003366
*BASE   004071  004111  004114  004114

LIST    000001    FSWA    001020    FSWX    001026    FSIO    001102
FSA1    001436    FSA3    001436    FSA2    001442    FSA5    001442
FSA6    001447    FSCB    001766    FSIOBF  004611    FSER    004713
FSHT    004720    AC1     005000    AC2     005001    AC3     005002
AC4     005003    AC5     005004    WRASC   005005    IOCS$   005012
IOCS$T  005111    FSAT    005123    FSAT1   005125    WATBL   005170
LUTBL   005201    PUTBL   005236    RSTBL   005273    O$AD07  005330
O$AD08  005527    I$AA01  005571    I$AP02  005603    O$AA01  005763
O$AP02  005767    PRWFIL  006063    EXIT    006066    ERRSET  006073
OPSCHK  006076    T1IN    006127    T1IB    006230    T1OB    006235
P1IN    006242    P1OU    006263    P1OB    006301    P1IB    006305
```

$SA *TEST	Program memory image is saved under filename *TEST. Low, high, starting location and other parameters are supplied automatically by loader. (See Map.)
$ QU OK;R *TEST	Control returns to DOS. User uses Run command to start test program, which
TEST MESSAGE	takes control. Printed message indicates successful operation
OK;	

This simple example shows a typical interaction of DOS, ED, FTN and LOAD. The user could also punch a self-loading tape of the memory image using MDL, or use TAP or PSD to examine or alter the memory image.

Operating with paper tape is similar but programs must be loaded from the supplied self-loading tapes. Any special instructions for paper tape operation appear in the appropriate sections of this guide. Section 9 provides additional information for the paper tape user.

SYMBOLS AND ABBREVIATIONS

Symbols and abbreviations used throughout the test are defined in Table 1-1.

Table 1-1. Symbols and Abbreviations

Symbol	Definition
Number Representations	
1000	1000 decimal
'1000	1000 octal
$1000	1000 hexadecimal
Terminal Keyboard Functions:	
.CR.	Carriage Return
.LF.	Line Feed
.NL.	Next Line (Carriage Return or Line Feed)
\	Backslash (upper case L) used as tab character
"	Delete character (cancels last typed character)
?	Kill character (deletes current line)
Miscellaneous:	
EA	Effective Address
(EA)	Content of Effective Address
[ ]	Brackets enclose optional parameters
␣	Mandatory Space Character

SECTION 2

EDITORS

PART 1 TEXT EDITOR

EDITOR FUNCTIONS

Summary

Prime's text editor programs, generically ED, apply the data handling ability of the Prime processor to the mechanics of generating source programs or text files that are compatible with the DOS and/or DOS/VM file structure. For example, the Editor program accepts characters typed at the terminal and stores them in high-speed memory. Rough text can be printed in part or in entirety and inspected for errors. With editing requests, errors can be corrected, and text can be inserted or deleted.

Completed text can be saved as named disk files or punched on paper tapes. For a discussion of files, refer to the DOS (DOS/VM) User Guide. Files created by the Editor can be transferred from disk to any storage medium or printing device available in the system.

Through the use of the appropriate Editor module, existing tapes or disk files can be read in memory and altered or appended to other text.

CONFIGURATION

For Prime software revision 6, there are four versions of the Editor available on the master disk: ED, EDLIN, BPTRED and PTRED. These versions differ in whether or not they handle BOX editing functions and whether or not they handle file functions. (See Table 2-1.) The different versions of the Editor are provided because BOX mode editing functions are isolated into a separate module that can be optionally loaded. In addition, all file operations are isolated within the Editor File System (EFS) module. The BOX mode and the file operations are independent and mutually exclusive. Thus, a degree of flexibility and space saving is achieved by providing Editors with various combinations of BOX, FILE, and standard editing modules and by allowing the user to copy the desired version of the Editor program into the user's command directory (usually CMDNCØ). Refer to Table 2-1.

Table 2-1. Available Editors and Scope of Functions

Handles Files	Does Not Handle Files	Functions
ED	BPTRED	BOX Mode
EDLIN	PTRED	No BOX Mode

As indicated by the table, (1) the Editor, ED, has the full set of functions (commands) described in this document; (2) EDLIN is used for handling files and LINE editing but does not have the BOX editing commands, thus it is smaller than ED and useful if the user does not want to do BOX editing; (3) BPTRED is an Editor for use with paper tape, and does not handle files but does provide the BOX editing commands; and (4) PTRED is an editor for paper tape and line editing only, and PTRED does not allow file handling or BOX commands.

Segregating Editor functions in four different versions allows a user to achieve a significant saving of space. For instance, use of EDLIN or PRTED vs. ED allows over three additional sectors of space to be saved, which may be desirable if the user does not need BOX editing (in the case of EDLIN) or BOX editing and file handling (in the case of PRTED).

For the convenience of users that have chosen to operate with older versions of Prime software and for users in the midst of updating their software, Table 2-2 shows the versions of the Editor that were available on previous software versions.

Table 2-2 lists the high address of the four current versions and all previous versions of the Editor (based on a line size of 145 characters).

Table 2-2. Software Revisions, Editors and their High Addresses

Version	Rev. 3A	Rev. 5	Rev. 6	Command File Name (Rev. 6)
ED	'12044	'20733	'21267	C←ED
EDLIN			'14771	C←EDLI
BPTRED			'15367	C←BPTE
PTRED		'17164	'13776	C←PTED
FILED	'12732			

Note, at Rev. 5, two versions of the Editor existed; ED and PTRED. The difference was in the Editor file system modules (EFS). Under PTRED, no temporary files were allowed and the LOAD and FILE Editor commands did not work. (However, UNLOAD, DUNLOAD, BOXIN and BOXOUT commands did.) The high address of PTRED was over '17000, making 8K stand-alone operation difficult. At Rev. 6, editor commands inconsistent with the configuration (e.g., UNLOAD for PTRED) result in the illegal command message (?).

ENTERING AND LEAVING EDITOR CONTROL

Editor Command Syntax

For details of the operating system command language, refer to the Prime Computer User Guide for Disk and Virtual Memory Operating Systems (the DOS Manual).

In this description of the Editor and its commands, strings that are to be specifically input are printed in all capital letters, variable arguments have initial capital letters, and if an argument is optional, it is enclosed within brackets [ ].

Entering Editor

The Editor may be started from DOS or DOS/VM command level by the external command:

ED [Filename]

Similarly, the line editor EDLIN may be invoked by the external command:

EDLIN [Filename]

The paper tape editor may be invoked by the external command:

PTRED

and the paper tape editor with BOX mode functions may be invoked by the external command:

BPTRED

If Filename is specified, the operating system searches the current UFD; if the file is found, the operating system loads it into the Editor's text buffer in high-speed memory and starts the Editor in EDIT mode.

Example: (In all examples in this section, user input is underlined.)

OK, ED FTNXMP

GO

EDIT

DOS (DOS/VM) prints GO, indicating that the ED program has been loaded and started; ED prints EDIT, indicating that ED is ready to accept editing requests.

If no Filename is specified, ED starts in high-speed INPUT mode. Example:

OK, ED

GO

INPUT

If the specified Filename is not found in the current UFD, DOS (DOS/VM) prints an error message. Example:

OK, ED FTNXML

GO

FTNXML NOT FOUND

ER, ED FTNXMP

GO

EDIT

EDITING IN DISK ORIENTED CONFIGURATIONS

Filenames

Filenames used by the text editors, ED and EDLIN, are formed according to the same rules as any DOS or DOS/VM names.

Leaving Editor

The FILE, QUIT, and PAUSE requests, issued in EDIT mode, return the user to DOS or DOS/VM command levels. The FILE request causes the contents of the Editor's text buffer in high-speed memory to be stored on disk under the specified Filename. Examples:

```
FILE  TEXTF1

OK,
```

Using the specified filename, DOS or DOS/VM stores the file in the current UFD and responds OK,. The specified Filename may differ from the original. The user may prefer to file edited versions under a different Filename in a temporary file until editing is complete. When the final version is checked and verified, the old versions can be deleted by the DELETE command and the final version can be renamed by the CNAME command. (Refer to the DOS User Guide for information about CNAME, and DELETE.)

The QUIT request makes it possible to return to DOS without writing over the disk copy of the file that is being edited. This is useful when an error or accident has badly garbled the contents of the editor's text buffer or the edited file was only being examined.

The PAUSE request, unlike FILE and QUIT, does not terminate editing. As long as the Editor memory image is intact, editing can be resumed by issuing the DOS (DOS/VM) START command with no arguments (e.g., typing: S).

Error Restart

If an error halts the processor, it is often possible to restart the Editor without losing the edited text. For further information, refer to "Recovery Procedures".

EDITING WITH PAPER TAPE CONFIGURATIONS

In previous revisions, paper tape handling with the Editor was awkward, especially on the ASR paper tape reader-punch. At Rev. 6 the following improvements have been implemented:

Punching:

The punch is started by the Editor on the ASR, avoiding the problem of garbage characters on the tape (notably carriage return/line feed).

Blank leader is punched as well as trailer.

All punching is done by FTNLIB routines consistent with IOCS.

Reading:

The Editor automatically starts the ASR reader for each record (DOS and DOS/VM).

Initial leader is read and ignored.

High-speed paper tape is punched with parity ON and ASR paper tape is punched with parity OFF except for the XOFF- CR.-.LF. sequence at the end of each record. Tapes punched on both devices can be read by both the high-speed and ASR-readers. (Caution: punching on the ASR punch must be done in BRIEF mode.)

EDITOR MODES

The editor has two general modes of operation: LINE (EDIT) mode and BOX mode. LINE mode allows line-by-line editing of files consisting of ASCII text (e.g., source files). BOX mode allows the editing of two-dimensional files as pictures.

In LINE mode, it is possible to operate in both an INPUT mode and an EDIT mode. INPUT mode is used for entry of new text. EDIT mode is used to locate, alter, and print lines of text and to read/write files and paper tapes. BOX mode is entered from LINE mode. BOX mode contains no concept of input as defined by INPUT mode; and therefore, the only way to enter INPUT mode from BOX mode is to first enter LINE mode.

Character Set

In either INPUT, EDIT, or BOX mode, the Editor accepts any of the characters of the 128 character ASCII subset. However, certain special characters have special effects (See Table 2-3.) Non-printing characters such as X-OFF and others that operate with the CTRL key pressed can be entered, stored in the Editor's buffer, and filed. However, when such characters are printed, the Editor converts them to an up-arrow followed by the octal equivalent of the character. For example, an X-OFF character is echoed as ↑223.

Strings

A string is a series of numerals, alphabetic characters, or punctuation. Certain punctuation characters have special significance. (See Table 2-3.)

EDITING IN LINE MODE

In the past, the semicolon was treated as a new line in both INPUT and EDIT modes. It now behaves as follows:

Meaning of Comma and Semicolon in INPUT mode: In INPUT mode, the comma has no special meaning. The semicolon, however, is treated as a new line without reading a new line. Hence, the sequence:

```
A;B;;C;D
```

causes the file to contain:

```
A
B

C
D
```

Note the null line effected by ;; without space between them.

The sequence:

```
A;B;
```

puts A and B into the file and enters EDIT mode.

Meaning of Comma and Semicolon in EDIT mode: In EDIT mode, the semicolon is synonymous with comma except that it is recognized for all commands including INSERT (I), RETYPE (R), OVERLAY (O), and APPEND (A).

Table 2-3. Special Characters for LINE Mode Editing

Character	Effect
(Shift L) \	INPUT Mode: Default "TAB" character. Inserts spaces up to next tab position. Standard tab positions are in column 6, 12 and 30. Others may be set up by TABset request. EDIT Mode: Do not use.
"	INPUT or EDIT Mode: Default "ERASE" character. Deletes preceding typed character. Each use of " deletes another proceding character: For example, ENTER " " " " " deletes the word ENTER. The erase character can be changed to another ASCII character by the ERASE or SYMBOL request.
?	INPUT or EDIT Mode: Default "KILL" character. Erases entire line to the left of the character. The kill character can be changed to another ASCII character by the KILL or SYMBOL request.
#	EDIT Mode: Default "BLANKS" character. Matches any number of spaces in LOCATE or FIND requests. (Treated as normal character in Change.) For example, L DEC 10 COMMENT could be located by L #DEC#10#COMMENT. INPUT Mode: Normal printing character.
!	EDIT Mode: Default "WILD" character. Matches any character in LOCATE or FIND requests. (Treated as normal character in CHANGE.)(In OVERLAY, changes current character to space.) Example: L D!C locates a line containing either DEC or DAC. INPUT Mode: Normal printing char.
↑or ^ (up arrow)	INPUT or EDIT Mode: Default "ESCAPE" character. ↑ddd - Enters a 3-octal-digit code for non-printing ASCII control characters, such as ↑223 for X-OFF. ↑U, ↑L - Permits alphabetic characters to be stored as capital or lower case codes. (Useful only with upper/lower case printing devices.) All characters after ↑L are stored as lower case. All characters after ↑U are stored as upper case. Does not affect symbols. ↑N,↑S,↑E,↑W - Direction characters ↑, ↓, ←, and → in BOX mode. - Except for above forms, ↑ may be used as a normal printing character by typing ↑↑.

Table 2-3. (Cont'd)

Character	Effect
*	EDIT Mode: Abbreviation for XEC EDLIN when entered in request string. (See XEC, MOVE.) INPUT Mode: Normal printing character.
,	EDIT Mode: Separates multiple request on a line. INPUT Mode: Normal printing character.
;	EDIT Mode: Default "SEMICOLON" character. Terminates INSERT, REPLACE, OVERLAY and APPEND requests in request lines, as in: N,L TEST LINE, R TL;* INPUT Mode: Acts as a new-line character.
$	EDIT Mode: Default "CPROMPT" character. Printed whenever the Editor is ready to accept a new command line. Otherwise, $ is a normal printing character.
&	INPUT Mode: Default "DPROMPT" character. Printed whenever the Editor is ready to accept a new input line. Otherwise, & is a normal printing character.

NOTE: The characters \, ", ?, #, !, ↑, ;, $, and & are default characters for TAB, ERASE, KILL, BLANKS, WILD, ESCAPE, SEMICOLON, CPROMPT, and DPROMPT. These characters can be respecified by the SYMBOL command.

Example (User input is underlined):

```
I XXX,P,,;P
 XXX,P,,
R Y,Y;P
 Y,Y
O A!B,C;P
 A B,C
```

Wherever a comma was used as a separator, a semicolon can be used as in the following command line: I XX;P;N5;*5;T;P30 which inserts and prints XX every fifth line for 25 lines, does a TOP operation and prints all 30 lines.

Two adjacent semicolons (or commas) put ED into INPUT mode. Example:

```
I XX;;ABCD
```

inserts XX and goes into INPUT mode. The ABCD is then read as an input line. The resulting text would be as follows:

```
XX
ABCD
```

An initial or terminal semicolon (or comma) also puts ED into INPUT mode. Examples:

```
,ABC
N,P,
```

In a complex request string, to visualize the result; put the INPUT command into the null spaces. Examples:

```
I XX;INPUT;ABCD
```

and

```
I XX;;ABCD
```

are equivalent. Also:

```
INPUT;ABC
```

and

```
;ABC
```

are equivalent. And:

```
N,P, INPUT (.NL.)
```

and

```
N,P,
```

are equivalent.

The INPUT request signals the editor to enter INPUT mode. The second semicolon, comma or new line closes the command. In sequences, such as

```
,,,     ,;,     ;,,     ;,;

..;     .;;     ;;.     ;;;
```

the third character is always interpreted in INPUT mode.

NOTE: The requests INPUT (PTR) and INPUT (ASR) change the device immediately, thus the remainder of the typed line is ignored by ED.

Entering INPUT Mode

INPUT mode is in effect:

1. When the editor is started by the ED (EDLIN, PTRED, or BPTRED) command without a Filename specified.

2. When a null line (two successive CR characters) or two successive semicolons or commas are entered during EDIT mode. Any new text follows the last line that was edited or printed.

The text of a new file may be typed into memory from the terminal. In INPUT mode, the terminal is used as a typewriter; everything entered at the terminal (except a few special control characters) is stored in the Editor's buffer in high-speed memory. No response is printed, so the text may be entered as fast as the user can type.

Lines

Text is entered a line at a time by the carriage return key (CR) which is stored in the Editor's buffer as a new line (NL) character ('212). Each line thus consists of a string of ASCII characters terminated by a NL character. Lines are stored in the same sequence as they are entered. Subsequent editing is done a line at a time.

Correcting Typing Errors

The erase (") and kill (?) characters may be used to correct typing errors in either INPUT or EDIT mode. Each use of the erase character deletes one of the preceding typed characters. For example, the line:

```
TEST LNN""INE 1 (CR)
```

is stored in the buffer as: TEST LINE 1.

The kill character is used when a line is so hopelessly garbled that it must be retyped. It deletes all preceding characters on the line, for example:

```
TJST LZ" " " " " " ER?TEST LINE 2 (CR)
```

After using the kill character, continue typing the desired line before entering the text with a CR. If the kill character is followed immediately by CR, it has the same effect as entering a null line, and switches the editor to EDIT mode. The characters " and ? are the default erase and kill symbols. To free them for use in text, other symbols can be assigned by the ERASE, KILL and SYMBOL commands.

Tabulation

To simplify arrangement of text in vertical columns, ED recognizes the FORM character (upper-case L) as the tabulation character. On input, this character echoes as a backslash (\), but it is interpreted as a signal to enter enough space characters to reach the next tab stop.

Standard tab stops for ED are in columns 6, 12, and 30 (mainly for convenience in coding PMA programs). However, up to eight tab stops may be set in any columns using the TABSET request. For example, in microprogramming the tab stops normally set by the user are 8, 16, 21, 24, 32, 39, 45 and 51.

Tab stops are set for the life of the local invocation of the Editor. If the user leaves the Editor via the QUIT, FILE requests, or CTL-P; then the tab stops are reset to the default values.

Entering EDIT Mode

To enter EDIT mode from INPUT mode, enter a null line (i.e., two CR's in a row or a CR following a kill (?) character, or two semicolons, etc.).

EDIT MODE

The editor is in EDIT mode:

1. After ED is started from DOS (DOS/VM) by an ED command (or other Editor commands) that specifies a Filename present in the current UFD.

2. After two successive CR characters or a kill character followed by a CR, or two successive semicolons entered in INPUT mode.

3. After a restart by the START command either at location '1000 (refer to the DOS Manual for a discussion of START).

In EDIT mode, the editor accepts editing commands from the terminal. (The commands are described later in this section.)

Pointer Location

Most editing commands in LINE mode depend on the position of a conceptual pointer that keeps track of the line to be edited. The pointer is always considered to be located at the beginning of a line; that is, between the CR that terminates the preceding line and the first character of the line to which it points. The pointer location can be altered by the BOTTOM, CHANGE, DELETE, DUNLOAD, FIND, INSERT, LOAD, LOCATE, MODIFY, NEXT, PRINT, PUNCH, TOP, UNLOAD, and XEQ commands.

To determine the present pointer location, give the PRINT or WHERE commands. ED prints the line containing the pointer and waits for further requests. After a VERIFY command, the line containing the pointer is automatically printed after every command that changes the pointer location or the line. (The Editor is initialized in the BRIEF mode, which speeds editing by suppressing verification printouts.

If the Editor is asked to print the top line (one above the first line) of the file, the bottom line (one beyond the last line) or a newly deleted line, it prints .NULL. If an attempt is made to move the pointer beyond the limits of the text image, TOP or BOTTOM is printed.

When starting in INPUT mode with an empty buffer, there is no need to pay attention to the pointer. Text can be entered at the terminal just as if at a typewriter (except that the ", the ?, and the tab (Shift L) characters cannot be entered indiscriminately).

When an existing file is specified for editing, the pointer is initialized at the top of the file. If the user switches to INPUT mode, anything that is typed is inserted at the beginning of the file.

During editing in LINE mode, the user must keep track of the pointer position. After a switch to INPUT mode, any new text is inserted following the line containing the pointer. As each new line is entered, the pointer moves along with it. After entering a block of new text in INPUT mode and returning to EDIT mode, the pointer is positioned at the last line entered.

Verify/Brief Modes

The Editor is initialized in BRIEF mode for quicker editing. Nothing is printed unless the user enters a PRINT request. After a VERIFY request, every line that is located or altered by an editing request is also printed.

Returning to INPUT from EDIT Mode

A null command (two CR's or semicolons in a row or a kill character followed by a CR) returns the Editor to INPUT mode.

LINE MODE COMMAND DESCRIPTIONS

Commands may be either spelled out fully or abbreviated. (Letters that are essential in the abbreviations are underlined.) There must be one space between an Editor command (or abbreviation) and its character arguments, if any.

If a command causes the pointer to reach the top of the file, TOP is typed and the request is terminated. If the request causes the pointer to reach the bottom of the file, BOTTOM is typed and the request is terminated.

APPEND String1

Appends String1 to the end of the current line. Trailing blank characters, if any, are eliminated before String1 is appended. The string can be terminated by either a semicolon or CR character; however, commas are treated as normal printing characters. A space must separate the request APPEND and the argument String1. Example:

```
PRINT
XXXXX

APPEND  YYY
PRINT
XXXXXYYY
```

Tab characters are used in the APPEND command to space String1 to the first tab position beyond the end of the line. Example:

```
V,TAB 7, P
AAA

A  X  X
P
AAAX  X
```

BOTTOM

Moves the pointer one beyond the last line of the file, (a dummy line).

BRIEF

Speeds editing by minimizing responses; only the PRINT request causes printing. This is the Editor's normal (default) condition. When the Editor is entered from DOS or DOS/VM command level, BRIEF mode is automatically set in effect.

CHANGE %String1%String2% [n G]

Changes the first occurrence of String1 to String2 on the current line. If the optional argument n is supplied, CHANGE searches n lines and changes String1 to String2. If G is present (Global), all occurrences of String1 on the current line are changed to String2. If both n and G are specified, then an n-line search is made and all occurrences of String1 are changed to String2.

The delimiter, %, may be any character that is not contained in String1 or String2 and is other than ", ?, ;, or comma.

The pointer moves to the start of the nth line. If n is 0, 1, or unspecified; only the current line is examined and the pointer does not move. If n is greater than the number of lines in the text buffer after the current line, the pointer is positioned at the bottom of the file, and the message BOTTOM is printed. Examples:

```
V, P
STAZLOT
C%Z% Z%
STA ZLOT
C%T%R%G
SRA ZLOR
```

DELETE [n]
TO String

DELETE [n]: Deletes n lines. The argument n may be positive or negative. The current line is always included in the count. If n is not specified or is +1, 0, or -1, only the current line is deleted. The pointer is left at the last line deleted.

Example Deleting One Line:

Contents of a Command File Before Editing:

```
.NULL.
AVAIL
AVAIL ONE
AVAIL TWO
AVAIL THREE
AVAIL FOUR
STATUS
```

Editor Request:

```
T.F AVAIL TWO
D
```

After:

```
.NULL.
AVAIL
AVAIL ONE
AVAIL THREE
AVAIL FOUR
STATUS
```

Example -- Deleting Multiple Lines:

Before:

```
3000  REM PROG TO WRITE HEADERS
3050  M$='
3060  N$='MNEMONIC'
3070  O$='OPCODE'
3080  F$='FUNCTION'
3090  I$='AVAILABLE ON'
3100  WRITE# 2,'PRIME INSTRUCTION SET'
3110  WRITE# 2,M$
3120  WRITE# 2,M$
3130  WRITE# 2,N$,O$,F$,'TYPE','OPTIONS'
3140  WRITE# 2,M$
3150  WRITE# 3,'OCTAL OP CODES'
3160  WRITE# 3,M$
3170  WRITE# 3,O$,N$,'FUNCTIONAL DESCRIPTION'
3180  WRITE# 3,M$
3190  WRITE# 4,'INSTRUCTION TIMING'
3200  WRITE# 4,M$
3210  WRITE# 4,M$
3220  WRITE# 4,'INSTRUCTION','P100','P200','P300 (750 NS)','P300 (600 N$
3230  WRITE# 4,M$
3340  RETURN
5000  REM FILE MANIPULATOR TO PRODUCE MNEMONIC LIST
```

Editor Requests:

```
F 3000,   P
3000  REM PROG TO WRITE HEADERS
D 21
```

After:

```
5000  REM FILE MANIPULATOR TO PRODUCE MNEMONIC LIST
```

Example -- Deleting When n is Negative

Before:

```
175  M$= 'ABC'
195  GOSUB 3000
199  PRINT 'HEADERS WRITTEN
200  GOSUB 3050
210  READ# 1, A$, B$, C$
```

Editor Requests:

```
F  200
D  -3
```

After:

```
175  M$= 'ABC'
210  READ#  1, A$, B$, C$
```

DELETE TO String1: deletes all lines until a line containing String1 is encountered. The current line is always deleted. The line containing String1 is not deleted. The pointer is set at the line containing String1. Example:

Before:

NUMBER	NAME	ADDRESS	CITY
0001	JANE DOE	13 13TH STREET	BOSTON, MA
0013	JOHN DOE	1 ELM STREET	CLEVELAND, O
0100	JOE SCHMOE	4 OAK STREET	NATICK, MA

...

Requests:

```
L DOE
DELETE TO  SCH
```

After:

NUMBER	NAME	ADDRESS	CITY
0100	JOE SCHMOE	4 OAK STREET	NATICK, MA

DUNLOAD

Refer to the description of UNLOAD.

ERASE %

Changes the current erase character to the character % (any ASCII printing character except those defined in SYMBOL command). When the Editor is entered from command level, the erase character is set to ". Example:

```
ERASE #;

INPUT
"FOUR SCORE AND 7# YRS###YEARS AGO
```

Resulting Text:

"FOUR SCORE AND SEVEN YEARS AGO

FILE [Filename]

Writes the contents of the Editor's text buffer into a file called Filename in the current UFD. If Filename is not specified, the file is stored under the original Filename specified in the Editor command (e.g., ED). If no Filename has yet been specified, or if two or more Filenames have been used, the Editor types the message WHAT NAME?; the user should repeat the request using a valid Filename. Example 1:

```
FILE ZILCH

OK,
```

The contents of the text buffer are written into the file ZILCH and the Editor returns to DOS (or DOS/VM) command level.

Example 2:

```
ED  OLDFIL
GO
EDIT
C  /AT/AM/300 G
FILE
OK,
```

The changed contents of the text buffer are written into the file OLDFIL that was specified at the time ED was invoked.

Example 3:

```
ED  MAIN
GO
EDIT
...
B
LOAD  PROG1
...
B
LOAD  PROG2
FILE
WHAT NAME
FILE PROG2
```

The contents of the text buffer are written into the file PROG2.

File Handling

Name syntax: File names must conform to the DOS/VM CMREAD rules. These rules are:

1. All characters must be printing characters.
2. The name cannot contain ",?, or imbedded blanks.
3. The first character must be non-numeric.
4. Lower case letters are converted to upper case.

The Editor does not create files that cannot be accessed from the user terminal under any Prime operating system.

SAM/DAM Files: Only SAM or DAM files can be accessed by the Editor. If an attempt is made to access a segment directory or UFD, the message:

Filename ILLEGAL EDIT FILE

is printed and the operation aborted.

Truncation: All file write operations (FILE, UNLOAD, DUNLOAD, and BOXOUT) overwrite the old file and truncate as opposed to deleting and rewriting. Hence, there is no danger of a complete loss of a file because of an abnormal halt after the delete but before the write.

Update Protection (DOS/VM only): When a file is edited, it is left open in read/write mode on unit 5 for the duration of the edit (until QUIT or FILE). As a result, multiple read access of a common file is precluded, but any edited file is protected from asynchronous updating from two or more people. The fact that the file is left open is functionally invisible to the original user, but causes the message: Filename IN USE for all other users.

Unique Temporary Files: To resolve any possible conflicts, the Editor uses temporary files T###XX where XX starts at 00 and proceeds, if necessary, up to 99 until a suitable, unused file name is found. For example, if two users were editing large files on the same UFD, temporary files T###00, T###01, T###02, and T###03 would be used. Which user gets which file names depends solely upon demand.

Error Messages

All error messages are precise and, when applicable, include the file name. For example, the command LOAD FILEA when FILEA was not found results in the message FILEA NOT FOUND.

FIND Stringl

Moves the pointer forward to the first line beginning with Stringl. The FIND request facilitates location of lines by statement labels. If an end of file is reached, the pointer is positioned at the last line of the file and BOTTOM is printed. A blank (ASCII space) must separate the

FIND request and the argument String (e.g., F120 causes an error; F 120 is the correct form). Example:

Before:

```
          ...* (Assume pointer is at some position, nearer the TOP
               (beginning) of file)

     DIMENSION  SET(500)
     READ (2,5,) SET(I), I = (, 200)
  50 FORMAT (6F12.8)
     TEXT = AVRG (SET, 200)
     ...
```

Requests:

```
     VERIFY

     F 50
     50 FORMAT (6F12.8)
```

After:

```
     ...
     DIMENSION SET(500)
     READ (2.5) SET(I), I = 1, 200)
     50 FORMAT (6F12.8)          ← pointer position
     TEXT = AVRG (SET, 200)
```

Text is unchanged but the pointer has been set to line that was found.

INPUT Device

Reads text from the input device specified by the argument Device. The text is entered into the Editor's buffer in high-speed memory following the current pointer position. Possible values for Device are:

(TTY) Read from terminal (default value).

(PTR) Read from high-speed paper tape reader.

(ASR) Read from ASR paper tape reader.

The close parentheses in the argument is optional. For example:

```
INPUT (ASR)
```

and

```
INPUT (ASR
```

are equivalent. With either terminal input or paper tape input, a blank line (two successive CR's, or a semicolon followed by CR) in the contents of the text input from Device will stop the input and cause ED to return to EDIT mode. Blank leader and trailer is ignored.

INPUT (PTR) and INPUT (ASR) change the device immediately, so the remainder of the line typed at the terminal is discarded. For example, in the line:

```
INPUT (PTR); T;  L SUBR BEGIN3
```

the TOP and LOCATE requests are not executed and must be issued again when control returns to the Editor after the INPUT (PTR) request is complete.

Example:

```
INPUT(PTR)
EDIT
T
P10
.NULL.
*     HSMT1  REV XX
*
*
*     SENSE SWITCH:
*     1 HALT AT END OF PASS
*     2 BYPASS RELOCATION
*     3 NO ASR-HALT ON ERRORS
*     4 BYPASS MACHINE CHECK MODE
*     5 BYPASS PAGE TEST
FILE GDMT1
```

INSERT String1

Inserts String1 as one text line without switching to INPUT mode. (String1 is inserted as the line following the line currently pointed to.) The pointer is positioned to the beginning of the inserted line. Either a semicolon or a CR may be used to terminate this command. Also, a blank (ASCII space) must be typed between the INSERT request and the argument String1; otherwise, an error message is printed.

Example:

Before:

```
...
100 IF (ALT RTN .EQ. 0) GO TO 200
200 IF (A3 .NE. 0) GO TO 210
...
```

Request:

```
F 100
INSERT     GO TO ALTRTN
```

After:

```
...
100 IF (ACT RTN ,EQ. 0 GO TO 200
    GO TO ALTRTN

200 IF (A3 .NE. 0)  GO TO 210
...
```

KILL %

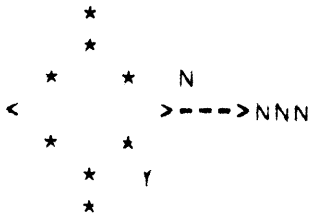
Changes the current kill character to the character % (any ASCII printing character except those defined in the symbol command). When the Editor is entered from the operating system command level, the kill character is initially set to ?. Example:

```
KILL @;

INPUT

QWERTYIOUP@ Q1?CALL?TTYIOU

Resulting Text:

Q1?CALL?TTYIOU
```

LOAD Filename

Loads Filename into the Editor's text buffer following the current line (current position of the pointer). The pointer moves to a null line following the file loaded. Examples:

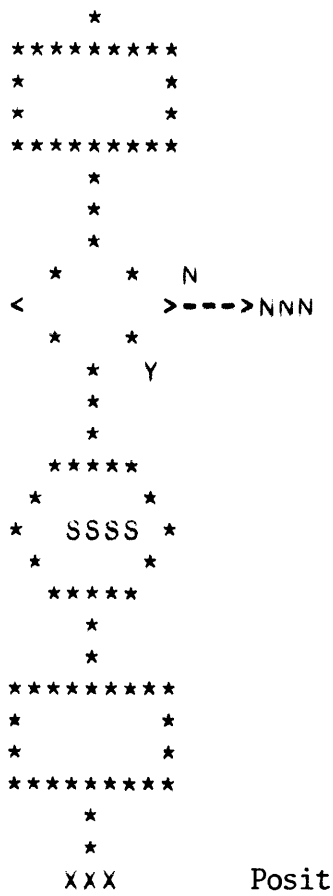
Contents of: File, BOX File, SUBR File, DECSN File, LEXIT



Editor Requests, Case 1:

```
OK, ED BOX
    GO
    EDIT
    B

    LOAD  DECSN
    EDIT
    LOAD  SUBR
    EDIT
    LOAD  BOX
    EDIT
    LOAD  LEXIT
    EDIT
```

Resulting Text:



Position of Pointer

Editor Requests, Case 2:

```
OK, ED BOX
GO
EDIT
LOAD SUBR
```

Result:

Note: This time SUBR was inserted before BOX; this is because at the initial invocation of the Editor, the pointer is positioned at the TOP of the file. When the LOAD command was given, the file specified in the LOAD command, SUBR, was inserted after the current position of the pointer, the TOP of file BOX, and before the body of the text of file BOX.

The user must pay attention to the current position of the pointer when using the LOAD command to be sure text is arranged in the desired order.

LOCATE String1

Moves the pointer forward to the first line encounted that contains the argument String1. The special meaning of the ↑, #, and ! characters is especially useful with this command (See Table 4-1 and the examples). The request LOCATE and the argument String1 must be separated by a blank (ASCII space) character. Example:

Contents of Text Line:

```
...

EVAL = F(ARG)  +  F(ARG/2)
```

The following locate requests would locate the above line (provided it was unique in the area of text searched):

```
L EVAL = F(ARG)

L E!!L

L (!!!)

L # F (ARG/

L ↑262  (ASCII 2)
```

MODE Parm

Indicates the mode change indicated by Parm. Parm may be:

PRUPPER

or

PRALL

or

PROMPT

or

NPROMPT

or

LINE

or

BOX [v h]

PRUPPER and PRALL are useful if devices with upper and lower case capabilities are present.

The effect of PRUPPER is to cause ↑L to be printed before lower case letters and ↑U before any upper case letters following lower case letters. For example, the string:

My Name

would be printed as:

M↑LY ↑UN↑LAME

The effect of PRALL is to print lower case letters as lower case letters. The string above would be printed as:

My Name

(if the device that the string is printed on has upper-lower case capabilities and software interface).

The effect of PROMPT is to cause the Editor to print a prompt character ($ in EDIT mode, & in INPUT mode) whenever the Editor is ready to accept input from the terminal.

The effect of NPROMPT is to cause the Editor to stop printing the prompt character.

The default values are PRUPPER and NPROMPT.

The use of the MODE command to specify PRUPPER or PRALL is independent of and does not conflict with its use to specify PROMPT or NPROMPT.

MODIFY %String1%String2% [n G]

MODIFY has the same format as CHANGE, but the action is to locate the string on the line, and copy String2 on top of String1. The alignment of the remainder of the line is unaltered, since any portion of String1 that is not specifically replaced is replaced with blanks.

Example: The line:

    1234567

is changed by the requests:

    MOD /34/X/

to:

    12X 567

The line:

    1234567

is changed by the request:

    MOD/34/XXX/

to:

    12XXX67

MOVE Buffer1 Buffer2

Moves one line of text to Buffer1 from Buffer2. Available buffers are:

STRA, STRB, STRC	Three string buffers
EDLIN	Buffer containing the last request typed.
INLIN	Buffer containing current line being edited.

Refer to the XECUTE Buffer request.

NEXT [n]

Moves the pointer n lines forward if n is positive, or backward if n is negative. If n is 0 or not specified, it is assumed to be 1.

OUTPUT Parm

If a 9600 baud display terminal is connected to Port 3 of the System Option Controller (SOC), the output produced by verification prints can be directed to that terminal rather than the user terminal. This is done by specifying Parm as either:

```
OUTPUT  (TTY)

or

OUTPUT  (DISPLAY)
```

However, the output of an explicit PRINT request is always directed to the user terminal.

OVERLAY String1

Overlays String1 onto the line in the text buffer, starting at the first character position. OVERLAY accepts a semicolon as a legal terminating character. A space character within String1 leaves the original character in the text unaltered. The ! character changes the original character to a space. The # character has no special meaning when used with OVERLAY. Logical tabs are treated as the appropriate number of spaces to fill to the next Tap stop. Example:

Before:

```
1234    89
```

Request:

```
TAB 8
OV X ! \  Z,A;P
```

After:

```
X2 4    8Z,A
```

PAUSE

Causes a return to operating system command level without changing the Editor state. This command provides a graceful way of escaping to the operating system and resume editing later. The Editor may be restarted at the state where the PAUSE was issued by issuing the DOS or DOS/VM START comand (simply typing: S).

PRINT [n]

Prints n lines starting with the current pointer position. n may be positive or negative. The count always includes the current line. Hence, P-1=P0=P=P1. In the case where n is less than -1, only the last line is printed. The general action of P-n is:

```
N-(n-1);P
```

The pointer is left pointing to the last line printed; it makes no difference whether the value of n is positive, negative, or zero. If the end of the file is reached BOTTOM is printed and the pointer is positioned at the last line of the file, a dummy .NULL. line. If the print request causes the pointer to reach the top of the file, TOP is printed and the pointer is stationed at the .NULL. line. A print request after a delete request or when the pointer is at the top or bottom of the file prints .NULL. indicating a dummy line.

PTABSET Tab1 ... Tab8

Provides for a setup of tabs on printing devices that have physical (mechanical) tap stops.

PUNCH (ASR) [n]
(PTP) [n]

Punches n lines on high-speed or ASR paper tape punch. Each line is followed by XOFF, CR, LF, and the tape is closed with an additional XOFF, CR, LF. Parm can be either (PTP) or (ASR); (PTP) is the default. n must be greater than or equal to Ø. When not specified, n is considered to be 1.

QUIT

Returns control to DOS (or DOS/VM) command level without writing the content of the Editor's text buffer into a disk file.

RETYPE String1

The current line is replaced by String1. The pointer does not move. A semicolon can terminate String1.

Example:

Before:

Copyright 1973

Request:

RETYPE Copyright 1974; P

Copyright 1974

The new text is the same as the system response shown above.

SYMBOL Name Char

The SYMBOL is a generalization of the KILL and ERASE requests. The allowable characters to change are:

Name	Default
KILL	?
ERASE	"
WILD	!
BLANKS	#
TAB	\
ESCAPE	↑
SEMICOLON	;
CPROMPT	$
DPROMPT	&

Char is the new character to be used for the specified name. This character cannot be a CR, comma, space, or asterisk, nor the current character used as any of the others in the above table. For example, ESCAPE cannot be set to # unless BLANKS is first set to something else.

The KILL and ERASE requests still have the syntax: COMMAND Char as before, but the validity of Char is checked as specified above.

Example:

```
SYMBOL ESCAPE @
```

Changes the escape from ↑ to @.

TABSET Tab1 ... Tab8

Sets up to eight logical tab stops (\) in the columns specified, where: Tab8 > Tab7 >... >Tab1.

TOP

Moves the pointer one above the first line of the text (a dummy line).

VERIFY

After a VERIFY request, any line that is located or altered is printed as it exists after the completion of the editing command. Verification output is printed or displayed at the device specified in the OUTPUT request.

UNLOAD and DUNLOAD

UNLOAD (and DUNLOAD) perform the inverse function of LOAD and have the forms:

```
UNLOAD  Fname n
UNLOAD  Fname TO String

DUNLOAD  Fname n
DUNLOAD  Fname TO String
```

The form: UNLOAD Fname n unloads (writes) n lines to the file Fname. If n is omitted or Ø, it is assumed to be 1. A negative n causes the preceding n-1 lines and the current line (in forward order) to be written to Fname.

The form: UNLOAD Fname TO String unloads (writes) to file Fname all lines, beginning with the current line, until a line containing String is found. The line containing String is not written to the file.

DUNLOAD is identical to UNLOAD except each line written to Fname is also deleted from the source file.

Care must be taken when specifying Fname since any existing file by that name will be deleted.

WHERE

Prints the current line number. This number is not in the file; it is a relative number updated and stored by the Editor. DELETE and INSERT etc. cause the line numbers to change.

XEQ Buffer

Executes the content of the specified line buffer as a request line. Possible buffers are STRA, STRB, STRC, EDLIN, and INLIN (See MOVE request). X EDLIN (represented by*) causes repeated execution of a command line, as in:

```
D,N-1,*
```

which deletes everything in the buffer from the current line to top; or:

```
L XYA,P,*
```

which prints every line that contains the string XYZ.

* [n]

Where n is a repeat count. When the repeat count is exhausted, the rest of the line is processed as specified by the previous Editor commands on the line. If TOP or BOTTOM is encountered in the range of a *, the entire line is terminated. If n is not specified, the processing continues until TOP or BOTTOM is encountered.

Example, in the file with 3 X's, the command:

```
L X, *3,I Y;P
```

will produce:

```
X
X
X
Y
```

Since there is only a single counter for the * repeat count, a second * on a line will always look like a terminal * with no repeat count. For example:

```
I XX;N5,*5,W,*3,T,P100
```

inserts XX every fifth line and the WHERE every 25th until BOTTOM, and never does the T, P100.

Any detected error condition terminates execution of the entire line.

EDITOR MESSAGES

Messages printed by the Editor in LINE Mode are:

Message	Significance
INPUT	Editor is in high-speed input mode waiting for text input from the keyboard.
EDIT	Editor is in Edit mode waiting for commands from the keyboard.
.NULL.	Editor has been asked to print deleted line or dummy line at top or bottom of file.
BOTTOM	An Editor command has moved pointer to bottom of file.
TOP	Editing command has moved pointer to top of buffer.
?	Unrecognized command.

EDITING IN BOX MODE

Switching between BOX and LINE mode is accomplished with the MODE request. Many requests that work in LINE mode do not work in BOX mode, and those requests that do work often have expanded, though analogous, meanings. Also, there are several new commands that work in BOX mode only.

BOX Mode

BOX mode allows character (ASCII) files to be treated as two-dimensional files. The x-coordinate, or character position, has equal status to the y-coordinate, or line position. Except in the cases of the physical bottom of the file and the physical end of the line, the two directions are equivalent.

Coordinates

Coordinates in BOX mode are always specified as vertical, horizontal, (v,h). The rationale for this is normal specification - as in a book - of page number, line and character position, in that order. All coordinates are one ordered.

The Box

The basic unit in BOX mode is the box. The box is used to define the working area in the file. Anything that works on a line in LINE mode works in a box in BOX mode. Of course, the size and the position of the box can be varied just as line position can be varied in LINE mode.

The Point

The point indicates the current line and character (v,h). In LINE mode, the point consists of line number only. In fact, the point does move through a line (as in the CHANGE command) but is always left at the beginning of the line. Thus, OVERLAY always begins in column one. In BOX mode, the point has two coordinates: line and character, and the point is always left wherever it might be. As a result, point positioning in two dimensions becomes far more important in BOX mode than in LINE mode. In addition, there are several ways of specifying the point's location: relative to the file (absolute), relative to the box (relative), or from current location (absolute or relative). The important thing to remember is that there is only one point, specifiable in several ways. The term absolute point means the point specified relative to the file and the term relative point means the point specified relative to the box. Therefore, a relative point of (1,1) is at the upper left hand corner of the box, regardless of the coordinates of the box. Similarly, an absolute point of (1Ø,1Ø) is at line number 1Ø, character position 1Ø, regardless of where the box is placed.

Direction

The point position can be specified relative to the current position by specifying direction of motion. There are four directions: north, south, east and west. They are indicated by ↑N, ↑S, ↑E, and ↑W where ↑ is the escape character (as in ↑U or ↑277). These characters are represented internally (and can be keyed in as such) as:

↑N = '234

↑S = '235

↑E = '231

↑W = '232

Directions are also represented by the four arrows ↑, ↓, →, and ← on the right side of the CRT terminal. Care should be exercised when typing in the actual characters since on many devices they actually move the screen curser whereas on others they are ignored. If entered in the INPUT mode of LINE mode, they are echoed as ↑N, ↑S, ↑E, and ↑W. The direction indicators apply to all point motions, including those implied in various commands. For example:

OVERLAY XX↑SXX↑WX↑S↑EYY specifies the string:

```
XX
 X
XX
 Y
  Y
```

Note that the default direction is ↑E, and the combined directions ↑S↑E yield southeast (combined directions work only in string specification and not in point or box coordinate specification). Note also that the direction specification causes a "step" in that direction before the next character is overlayed. In the case of an initial direction indication, as in OVERLAY ↑SXX, this pre-step is not taken. Thus, the first character of a string is always in the current position, regardless of any initial direction specifications. Thus, OVERLAY X = OVERLAY ↑DX where D stands for N, S, E, or W.

Box Dragging

Although relative versus absolute are two ways of specifying the same thing, namely point position, the consequences of using one method over the other are different. The point, by definition, is always inside the box. The point can be positioned relatively only inside the box. If the indicated directions or coordinates would take the point outside the box, the message BOX LIMIT is printed and the point is left

on the edge of the box at the coordinate (closest possible v position, closest possible h position) to the requested final location. The point can be positioned absolutely anywhere in the file and the box is repositioned, if necessary, to keep the point inside the box. Similarly, since box position is specified in absolute coordinates, the point is adjusted, if necessary, to keep it inside the box. These restrictions also apply to any command that implicitly moves the point. Some requests, like OVERLAY and MODIFY, are relative requests and do not go outside the box. Others, like FIND and LOCATE, go outside the box, but drag the box along with them. These examples are analogous to their operation in LINE mode. OVERLAY and MODIFY are restricted to the current line whereas FIND and LOCATE move across lines. Some requests, like FIND, have a corresponding request that works only inside the box. These requests are formed by adding an R to the beginning, as in RFIND, meaning relative (or restricted) FIND. When the box is dragged, it is dragged by its edge. Similarly, when the point is dragged by box repositioning; it is dragged on the edge of the box.

Point Independent Commands

A few requests are only concerned with the box size and position and not the point position. These are DISPLAY, BOXIN, and BOXOUT which operate on the entire box without altering the point location.

BOTTOM in BOX Mode

Since there is no INPUT mode or its equivalent in BOX mode, the bottom of the file defines an absolute lower limit which cannot be extended while in BOX mode. In this sense, it is equivalent to the physical line length which cannot be extended in any mode. Although the box position does take into account the top, left and right limits of the file, the bottom is an unknown entity until it is encountered. Although the box will never be positioned so that it hangs off the top, left or right of the file, it can hang over the bottom. If BOTTOM is ever encountered, the point is left, as in LINE mode, positioned to the null line just past the last line of the file. If BOTTOM is encountered by a request that does not move the point, i.e., DISPLAY, the point position remains where it was before the request and the short box is displayed.

BOX MODE COMMAND DESCRIPTIONS

The following is a description of all commands that work in BOX mode. All Strings can contain direction indicators.

BOX v h ↑D# ↑D#

Positions the upper left hand corner of the box to absolute position (v,h) or from the current position, it is moved by ↑D# where ↑D = ↑N, ↑S, ↑E, or ↑W, and # is the number of positions. If v,h, and D#'s are

specified, the final position is calculated and then the box moved. If the new position moves the box off the top, left, or right of the file, the request is rejected as an error and the box is not moved. If v and h are omitted, then the current position is used as a starting point. If only v is specified, current is used for h. If h is specified, v must also be specified. Explicit Ø's cause an error. When the box is moved, the point is also moved, if necessary, to keep it inside the box.

Special Case

If no parameters are specified, the upper left hand corner of the box is positioned to the point, if possible, and the point is not moved. Hence, the commands:

```
L string
W
BOX v h   (v and h are point position read
           from WHERE)
```

are equivalent to the commands:

```
L string
BOX
```

BOXIN Fname Parm

The inverse operation of BOXOUT causes file Fname to be loaded into the current box. Loading can be performed in two ways, according to Parm. For Parm (parentheses mandatory):

(MODIFY) - the load is carried out analogously to the MODIFY request. First the entire box is filled with blanks, then the file is loaded in. If any lines do not fit into the box, they are truncated. If there are more lines in the file than the box, they are not loaded.

(OVERLAY) - the load is carried out analogously to the OVERLAY request. Each character in the file is overlayed onto the corresponding character in the box. Any blanks in the file leave the original characters unaltered and ! characters in the file cause the corresponding position in the box to be filled with a blank. As in the (MODIFY) mode, if any lines are too long, they are truncated. If a line is too short, or the file is too short, any unspecified positions are treated as blanks.

The parameters (MODIFY) and (OVERLAY) may be abbreviated (M) or (O). If Parm is not specified, it is assumed to be (MODIFY).

BOXOUT Fname

BOXOUT writes the current box to the file Fname. The contents of Fname are exactly what is displayed by the DISPLAY command. BOXOUT does not cause the editor to quit.

BRIEF

Same as in LINE mode.

DISPLAY

DISPLAY displays the contents of the box to either the user terminal or the display tube on port 3 of the system option controller, depending upon the directions implied by the OUTPUT command. If the display is to the "DISPLAY" device on the system option controller, the screen curser is positioned to the current point position. If the display is to the user terminal, the curser position is left at the bottom of the display.

ERASE Char

Same as in LINE Mode.

FILE Fname

Same as in LINE Mode.

FIND String

FIND will find the specified string if it begins in the current horizontal position. For example, if the point is at (1Ø,5), the search will be down column 5. This command drags the box, if necessary.

KILL Char

Same as in LINE Mode.

LOCATE String

LOCATE locates the string if it begins at or after the current position. The search algorithm is to move the pointer right first, then at end of the line, move it to the far left of the file (X,1) and down until a match is found or BOTTOM encountered. This request drags the box, if necessary.

MODE Parm

Parm may be:

PRUPPER	-	same as in LINE Mode.
PRALL	-	same as in LINE Mode.
PROMPT	-	same as in LINE Mode.
NPROMPT	-	same as in LINE mode
LINE	-	go to LINE Mode. The current line number remains unaltered but the character position is reset to 1.
BOX v h	-	go to BOX Mode. v and h define the vertical and horizontal SIZE of the box. If v or h is omitted, the size in that direction is left unaltered. If h is specified, v must also be specified (but can = Ø indicating previous value). The default box size is 2Ø x 8Ø.

The use of the MODE command to specify PRUPPER, PRALL, PROMPT, or NPROMPT is independent of and does not conflict with its use to specify LINE or BOX. MODE BOX v h can be specified at any time to change the size of the box, but care should be taken since the new box position might be altered if the point should be outside the new box.

MODIFY /String1/String2/[G]

MODIFY performs the modify function within the box. The search for String1 is initiated from the current point position. The G option causes all occurrences of String1 in the box to be modified to String2. Although a repeat count is not rejected as an error, it is ignored. String1 and String2 need not have any relation in either length or directions other than that they both begin at the same position. If the box edge is reached while replacing String2, the message BOX LIMIT is printed and execution terminates and only that portion of String2 which was in the box is replaced.

MOVE Name1 Name2

As in LINE mode, Name1 is the "to" and Name2 is the "from" name. The operation is exactly the same as in LINE mode except that data cannot be moved to or from the buffer INLIN.

OUTPUT (Parm)

Same as in Line mode, but also includes DISPLAY output as well as verification output.

OVERLAY String

Same as in LINE mode except the horizontal tab (\) is treated as a normal printing character and not a tab character. If an OVERLAY reaches the edge of the box, the message BOX LIMIT is printed and execution terminates, and only that portion of string lying inside the box is overlayed.

POINT v h ↑D# ↑D#

POINT positions the point absolutely to position (v,h) or, from current position, ↑D# where ↑D = ↑N, ↑S, ↑E, or ↑W and # is the number of positions. If both v, h and ↑D# are specified, the point is first moved to (v,h), the box dragged, if necessary, and then moved by ↑D#, ↑D# and the box again dragged, if necessary. If v and h are omitted, the current position is used as a starting point. If only v is specified, current is used for h, but if h is specified, v must be specified. Explicit Ø's cause an error. If no parameters are specified, the point is not moved. Specifying the point outside the file (top, left or right) causes an error and does not move the point. If the point is moved past BOTTOM, it is positioned at BOTTOM and the box dragged accordingly.

PRINT

PRINT prints the current line in the box. The portion of the line, if any, outside the box is not printed. A repeat count other than Ø = -1 = 1 is rejected as an error as it would involve point repositioning.

PTABSET tab1...tab8

Same as in LINE Mode.

QUIT

Same as in LINE Mode.

RFIND String

RFIND is exactly the same as FIND except the search is restricted to the box. If no match is found, the point is left at (BOXB,h) where BOXB is the bottom of the box.

RLOCATE String

RLOCATE is exactly the same as LOCATE except the search is restricted to the box. If no match is found, the point is left at (BOXB, BOXR) where BOXB is the bottom of the box and BOXR is the right limit of the box.

RPOINT v h D# D#

RPOINT positions the point to RELATIVE position (v,h) or from current position ↑D# where ↑D = ↑N, ↑S, ↑E or ↑W and # is the number of positions. If both (v,h) and ↑D# are specified, the new point is calculated and, if outside the box, the message BOX LIMIT is printed and the point is left at (nearest v, hearest h) coordinates. If v and h are omitted, the current position is used as the starting point. If only v is specified, current is used for h; but if h is specified, v must be specified. Explicit Ø's cause an error. If no parameters are specified, the point is positioned to relative (1,1), the upper left hand corner of the box (RP = RP 1 1). RPOINT will never move the box.

SYMBOL Parm Char

Same as in LINE Mode.

VERIFY

VERIFY causes a verification DISPLAY whenever the contents of the box changes (including point position). Unlike DISPLAY, however, if only the point position changes, the box is not redisplayed. If the output, via the OUTPUT command, is directed to the display tube on the system option contoller, only the curser is moved to the new position. The commands that cause this kind of display are:

POINT (if new position is inside current box)

RFIND

RLOCATE

RPOINT

WHERE

WHERE prints the following information:

BOX	=	v, h AT v, h
POINT	=	v, h
RELPNT	=	v, h

the coordinates represent:

BOX	=	size AT position of upper left hand corner
POINT	=	absolute position
RELPNT	=	relative position within box

XEQ name

Same as in LINE Mode, except INLIN cannot be executed

*n

Same as in LINE Mode.

RECOVERY PROCEDURES

Abnormal Editor aborts are of the following types:

1. Deliberate user action (QUIT);
2. A specified file was (a) ILLEGAL, (b) NOT FOUND, or (c) IN USE;
3. The disk became full during a file write;
4. A file that was expected to be open was closed;
5. The disk was in WRITE-PROTECT.

The recovery procedures for each of the listed types of abort are different. Details are given in the following paragraphs. In any case, do not type the DOS (DOS/VM) command CLOSE ALL after an Editor abort.

User QUIT

A user QUIT consists of escaping to the operating system to stop such conditions as an infinite loop in a mistyped Editor command or unwanted printing at the terminal. (A QUIT is accomplished by pushing the QUIT or BREAK or INTRRPT button at the terminal, or by typing CTRL-P in DOS/VM, or by a HALT at the CPU Control Panel in DOS.)

Previously, a most common cause of forcing the Editor into an infinite loop was by typing a * as the first character of a line (e.g., typing a PMA comment line while in EDIT mode). This error is checked for and results in the message:

BAD *

However, other infinite loops are still possible

(e.g., typing:

T, L XXX,T,*

or

P, *

or typing a * as the second character of the line).

To abort an infinite loop and recover, issue the following commands:

CTL - P	escape to the operating system as appropriate
QUIT, OK, START 1000	operating system response
GO EDIT P	to verify position of current line

File ILLEGAL, NOT FOUND, or IN USE

These three types of errors result in a message and a return is made to the original user state and no action is taken as specified by Editor command line that caused the error. Thus, if the error occurs when the Editor is initially invoked with a Filename, a return is made to the operating system and the message ER! is printed at the terminal. If the error occurs at the time that a LOAD, UNLOAD, DUNLOAD, BOXOUT, BOXIN, or FILE Editor command is issued; a return is made to EDIT mode and the character ? is printed at the terminal.

The formats and the causes of the messages described here are given in the following paragraphs:

Message	Cause and Action
Filename ILLEGAL EDIT FILE	Caused by trying to Edit (1) a SAM segment directory, (2) a DAM segment directory or (3) a UFD. (Try another Filename)
Filename NOT FOUND	Caused by trying to Edit a file not in the current UFD. (Check UFD and get correct UFD or file, or type name of file correctly).
Filename IN USE	Caused by trying to access a file being used by another user. If the file, Filename, is the original edit file (open on Unit 5) this message can result if the file is currently a "bottom" file (to recover, type: BOTTOM Reposition (N-X) Retry operation

Disk or UFD Full

Under DOSVM: If a disk becomes full during a file write operation, the message:

```
DISK FULL
ER!
```

is printed at the terminal and Editor returns to the operating system. To recover, proceed as follows:

1. ATTACH, LISTF, and DELETE as necessary to create room on the disk. If the file being edited is IN USE, do not CLOSE and/or DELETE it.

2. ATTACH back to the original UFD.

3. Type the DOS (DOS/VM) START command.

The operation continues. Do not type: CLOSE ALL! Do not type: S 1000!

If the disk became full or the UFD was full on a file OPEN operation, the message:

```
DISK FULL   or   UFD FULL
?                ?
```

is typed. The Editor maintains control. To recover:

1. Escape to operating system. (Use the Editor PAUSE command.)

2. ATTACH, LISTF, and DELETE as necessary to create room on the disk. If the file is IN USE, do not CLOSE it and then DELETE it.

3. ATTACH back to the original UFD.

4. Type the DOS (DOS/VM) START command.

The operation continues. Do not type: CLOSE ALL! Do not type: S 1000!

Under DOS: At present, no adequate recovery procedure for this condition is possible.

A File Expected to be Opened Was Closed

This situation arises if the file was deliberately closed by the user, an action that constitutes an error. If the file was being read from (LOAD, BOXIN, or a "bottom" file), the error is treated as an End-of-File with no error indication to the user. If a file was being written to (UNLOAD, DUNLOAD, BOXOUT, FILE or a TOP file), the message:

```
Filename ON UNIT u NOT OPEN FOR WRITING
?
```

is typed and return is made to EDIT mode. Since closing and reopening a file also rewinds it, it is not sufficient to simply OPEN the requested file (Filename) on the specified unit (u). In general, the message indicates that the Editor has already written to the file and not closed it, but a subsequent write operation finds it closed. Two possible procedures are possible, depending upon Filename.

If Filename is an Editor temporary file (T###XX):

1. Escape to operating system (PAUSE).

2. Under DOS/VM, use CNAME Filename to a different name (not T###--). CAUTION - Under DOS, CNAME will destroy the Editor. Under DOS, in order to change the name:

   a) Type: PM

   b) SAVE the Editor memory image using the PM parameters except, use the end of memory instead of the high address (second parameter).

   c) CNAME the file.

   d) RESTORE the saved Editor image.

   d) DELETE the saved Editor image.

3. Open Filename on Unit u for reading and writing (e.g., OPEN Filename u 3).

4. Type: S 1000.

5. FILE to a new name. The original file has now been "split" into the renamed temporary file (step 2) and the new file (step 5).

6. Use the Editor to combine the two files. (Examine the last line of the renamed temporary file and the first line of the new file. These will probably be the start and end of one line).

If Filename is a named file (original or new), proceed according to what operation was specified. If this operation was:

UNLOAD, BOXOUT

1. Type PRINT to determine position.

2. Reposition to the point prior to the operation.

3. Retry the operation.

DUNLOAD

1. Type PRINT to determine position.

2. DUNLOAD the remainder of the file not written previously to a different file.

3. Use the Editor to combine the two files or leave them separate. There are no broken lines as in the case of an Editor temporary file.

FILE

1. FILE to a different file.

2. Use the Editor to combine the two files. A broken line can be present, so examine the last line of the first file and the first line of the second file.

Disk Write Protected

At present, no adequate recovery procedure for this condition is possible.

General Rules For Recovery

In recovering from any of these situations, three general rules should be kept in mind.

1. DO NOT TYPE "CLOSE ALL"!

2. If in doubt about what to do, consult this document.

3. If, at the completion of a recovery, any files are left open or any temporary files (T###XX) are left on the UFD, type "ED", enter EDIT mode, and "QUIT". This should eliminate any spurious files or open units resulting from the Editor abort.

PART 2 BINARY EDITOR

EDB is a binary editor for operation on loader-compatible object text blocks generated by the PRIME FORTRAN Compiler and Macro Assembler programs. EDB is useful for creating and updating library subroutine files on disk or paper tape. Input may be from disk or paper tape; output may be to disk or paper tape. Multiple input files may be open concurrently. EDB provides a large command set and issues explicit error messages.

LOADING AND STARTING UNDER DOS-DOS/VM

EDB is loaded and initialized by a command line beginning with 'EDB'. In general, the command line for initialization is as follows:

EDB	(PTR) Inputfile	[(PTR)] [Outputfile]

If either the input or output file is on paper tape, the appropriate filename is (PTR). Output is optional, and, as a result, an output file need not be specified. When an output filename to disk is specified, a file of that name is created in the current UFD.

When properly initialized, EDB types 'ENTER' and then loops for user command input.

USING PAPER TAPE VERSION

EDB is supplied as a self-loading, self-starting paper tape (SLT 0745.002). Once loaded, EDB types 'ENTER' and waits for user command input. The object file to be examined must be mounted on the high-speed reader. If paper tape output is desired, the high-speed punch must be turned on.

EDB FEATURES

Pointer

The user selects the next item to be processed by positioning a binary location pointer at the beginning of the desired subroutine name or entry point label. When EDB is initialized, or after a NEWINF command, the pointer is at the top of the input file. The pointer position can be changed by the FIND and TOP keyboard commands. During execution of the COPY, GENET and OMITET commands (which copy blocks from the input file to an output file), the pointer moves to the subroutine or entry point following the last item copied.

Verify/Terse/Brief Modes

In VERIFY mode, EDB prints the name of each subroutine or entry point reached by the pointer. From this printout, the user can determine the current pointer location. EDB is initialized in this mode. To speed binary editing, the user can specify TERSE mode (printing of subroutine names only) or BRIEF mode (no printing).

Special Action Blocks

Special action blocks ET, RFL, and SFL are written to the output file by the commands of the same name. These blocks are ignored (not copied) by the COPY, INSERT and OMITET commands. Thus each user can insert the special action blocks he requires.

ET (End of Tape Mark) is written by the GENET command as well as the ET command. On paper tape, ET consists of two successive characters, both '223. On disk, ET is represented by a zero word.

SFL, the Set Force Load flag block, is used in files to force loading of subroutines even if not called by a main program.

RFL, the Reset Force Load Flag block, resets the SFL condition and allows the main program to specify which subroutines within a file are to be loaded.

Messages

EDB prints the cue ENTER to show that it is ready to accept commands. Most errors in command string input cause EDB to print a question mark (?). Other messages include:

FILE NAME DOES NOT EXIST OR ALREADY OPEN

USER MUST SPECIFY INPUT FILE

YOUR INPUT FILE LOOKS LIKE SOURCE CODE

CHECKSUM ERROR- UNRECOVERABLE

BLOCK ERROR-UNRECOVERABLE

EDB COMMANDS

EDB responds to the following keyboard commands, listed in alphabetical order. Commands may be abbreviated to the underlined letters. Items enclosed in brackets are optional.

BRIEF

Inhibits printout of subroutine names and entry points as they are encountered by EDB. (See TERSE and VERIFY.)

COPY |Name / ALL|

Copies to the output file all main programs and subroutines (other than special action blocks) from the pointer to (but not including) the subroutine called 'Name' or containing Name as an entry point. If Name is not encountered or COPY ALL is specified, EDB copies to the end of the input file and types .BOTTOM. on the Teletype. Pointer moves past the last copied item.

ET

Writes an end-of-tape mark on the output file (203_8, 223_8 on paper tape; zero word on disk).

FIND Name

Moves the binary location pointer to a position on the input file corresponding to the beginning of a subroutine called 'Name' or containing Name as an entry point. If Name is not found, the pointer is moved to the end of the input file and .BOTTOM. is typed on the Teletype.

GENET [G]

Copies the subroutine to which the binary location pointer is currently positioned and follows it with an end-of-tape mark. The optional letter G specifies a global copy; all subroutines from the current position of the pointer are copied, each followed by an end-of-tape mark. When the bottom of the input file is encountered, .BOTTOM. is printed on the Teletype. The pointer moves to the next subroutine.

INSERT Name

Opens a second file, 'Name', for reading only and copies it to the output file (omitting all special action blocks). After the copy, the second input file is closed. The binary location pointer remains positioned in the original input file. An INSERT command operates only when the second input file and the output file are both on disk (however, the original input file may be paper tape).

NEWINF [Name]

Closes the current binary input file and opens a new input file for reading only. The binary location pointer is placed at the top of the new file. 'Name' must be specified to open a new file on disk.

OMITET [G]

Copies the subroutine to which the binary location pointer is currently positioned. The optional letter G specifies a global copy; all subroutines from the current position of the pointer are copied (omitting all special action blocks). When the bottom of the input file is encountered, .BOTTOM. is printed on the Teletype. The pointer moves to the next subroutine.

OPEN [Name]

Opens an output file for writing only. 'Name' must be specified to open a file on disk.

QUIT

Closes all files and exits to DOS. (When paper tape is the output file, an end-of-tape mark is punched before closing.)

RFL

Writes a reset-force-load-flag (library mode) block on the output file. This block initializes a true library file by enabling the loader to determine which subroutines within the file will be loaded. (See SFL.) This command operates only when output is to disk.

SFL

Writes a set-force-load-flag block on the output file. This block places LDR (the loader) in force-load mode; all subroutines in the files are loaded, whether or not they are called. SFL mode is in effect until the loader encounters an RFL block. A true library file should be terminated by an SFL block followed by an end-of-tape mark. This command operates only when output is to disk.

TERSE

Places the editor into 'terse' mode. Only the first name of each subroutine name block encountered by EDB is output to the Teletype. (See BRIEF, VERIFY.)

TOP

Moves the binary location pointer to the top of the input file. (Useful only when the input file is on disk.)

VERIFY

Places the editor into 'verify' mode. All subroutine names and entry points, as they are encountered by EDBIN, are printed on the Teletype. EDBIN is initialized in the 'verify' mode. (See BRIEF and TERSE.)

EXAMPLES

The following examples illustrate typical uses of EDB and show many of the commands in action.

Deleting Routines from a Library

A user named USER1 has a subroutine library under the filename LIBE that contains six subroutines:

LIBE Subroutines

ROUT1
TEST1
TEST2
ROUT2
MORE
AGAIN

The following EDB commands create another version of the library under the name LIBEV2, having the following contents:

LIBEV2 Subroutines

ROUT1
ROUT2
MORE
AGAIN

The commands are:

```
OK; A USER1

OK; EDB LIBE LIBEV2

GO

ENTER, BRIEF

ENTER, COPY TEST1

ENTER, FIND ROUT2

ENTER, COPY ALL

BOTTOM

ENTER, ET

ENTER, QUIT

OK;
```

After attaching to his UFD (i.e., USER1) the user invokes the binary editor, with LIBE specified as the input file and LIBEV2 as the output file. A BRIEF command simplifies the Teletype output. The first COPY command copies subroutine ROUT1 and the pointer stops at the beginning of TEST1. The FIND ROUT2 command moves the pointer past TEST1 and TEST2 (the two files to be omitted) to the beginning of ROUT2. A COPY ALL from that point copies the remainder of the file. An ET command is given to insert an end-of-tape block. The user then QUITs and returns to DOS.

Distributing Routines to Different Files

Assume the user has a collection of subroutines in a library file named FILIN:

FILIN Subroutines

FILE1
FILE2
FILE3

The following commands distribute these files to three different output files, named LIB1, LIB2, and LIB3, respectively:

```
OK; EDB FILIN LIB1
GO
ENTER, BRIEF
ENTER, COPY FILE2
ENTER, ET
ENTER, OPEN LIB2
ENGER, COPY FILE3
ENTER, ET
ENTER, OPEN LIB3
ENTER, COPY ALL
ENTER, ET
ENTER, QUIT
OK; LISTF

UFD=USER1
FILIN LIB1 LIB2 LIB3
OK:
```

After the first output filename (LIB1) is specified by the initial DOS command to start EDB, subsequent output filenames are set up by OPEN commands (it is not necessary to return to DOS). Each OPEN command closes the previous output file. Note that the user is careful to write an ET after each file is copied. (Remember that these files contain the object version of the specified subroutines.)

Combining Subroutines or Files Under One File Name

Assume that the same user wants to combine the separate binary files LIB1, LIB2 and LIB3 under a single filename, CLIB:

```
OK; EDB LIB1 CLIB
ENTER, BRIEF
ENTER, COPY ALL
BOTTOM
ENTER, INSERT LIB2
BOTTOM
ENTER, INSERT LIB3
BOTTOM
ENTER, ET
ENTER, QUIT
OK;
```

The first file to be inserted into CLIB is specified by the DOS command string that starts EDB. Thereafter, EDB INSERT commands specify new input files to be appended. Note that the ET marks at the end of the input files are not copied; the user explicitly adds an ET to mark the end of file CLIB.

Obtaining Subroutine and Entry Point Listings

With the aid of the VERIFY mode of operation, a FIND command can be used to print all subroutine and entry point names in a given file. Example:

```
OK; EDB FILIN
GO
ENTER; FIND XXX
FILE1
FILE2
FILE3
BOTTOM
ENTER; QUIT
OK;
```

In the FIND command, XXX is a dummy entry name that does not exist in the file.

SECTION 3

MACRO ASSEMBLER (PMA)

SOURCE PROGRAMS

Source programs must meet the requirements of the Prime Macro Assembly Language reference manual.

OPERATION UNDER DOS-DOS/VM

Loading and Starting Assembler

The Macro Assembler is loaded and started by the PMA external command to DOS:

PMA Filename [Startadd Areg]

where 'Filename' is a Macro Assembly Language source program in the current UFD, 'Startadd' is the P register starting option, and 'Areg' is an A register setting that specifies listing detail, I/O devices, and other assembly control parameters.

An alternate command format is:

PMA Filename [1/Areg]

This leaves the default starting address unaltered and modifies the A register value only.

If 'Startadd' and 'Areg' are not specified by the command string, the assembler uses the default values set up in the DOS RVEC vector at the time the assembler was SAVEd. These values are usually:

PC	'400	Normal start of assembly
A	'000777	Normal listing detail, all input and output files on disk

If in doubt, RESUME PMA and do a PM (Post Mortem) to determine the values for PC and A.

Starting Location

Starting options for the assembler are:

'400 Normal start of assembly

('401 Option is no longer used)

A Register Setting

The A register setting selects input and output devices, controls the amount of detail in the listing output, and includes other special controls. (See Figure 3-1.)

File Usage

The assembler does an automatic two-pass assembly of the specified input file, and generates object and listing outputs to the devices specified by the A register. The object file is in relocatable binary format suitable for processing and loading by Prime's Linking Loader.

When disk is used for the binary and listing files, filenames must be established. Unless it is preceded by BINARY and LISTING commands, the assembler will automatically open unit 2 to write a binary file named B←XXXX, and open unit 3 to write a listing file named L←XXXX, where XXXX is the first four letters of the input filename. The assembler closes any units that it opens. (Units opened by BINARY and LISTING commands are not closed.)

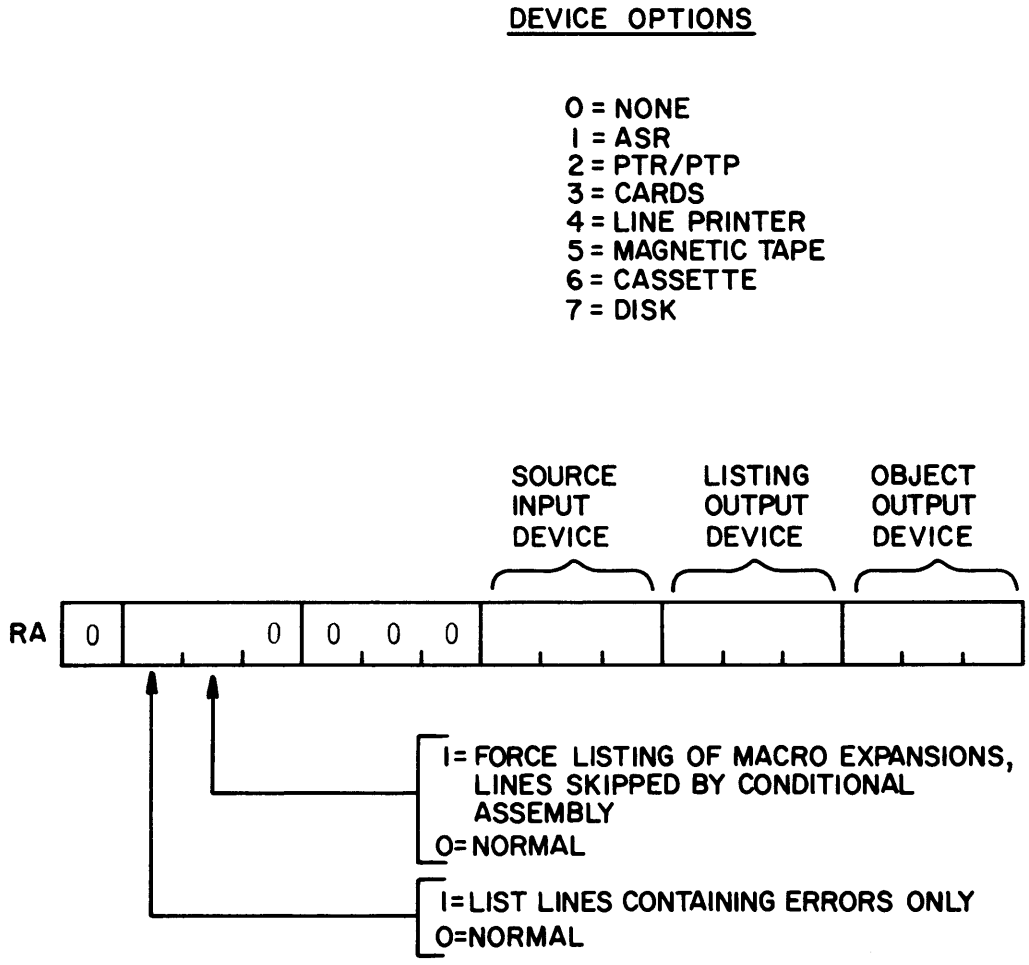


Figure 3-1. Macro Assembler A-Register Settings

USING PAPER TAPE ASSEMBLER

Procedure

The Macro Assembler is supplied as a self-loading tape (SLT 1080.013) that loads through APL or the key-in loader.

1. Turn to STOP/STEP and press MASTER CLEAR.

2. Load the P register (location 7) with the starting address:

   '00400 Normal start of assembly

3. Load the A register (location 1) with the assembly control options shown in Figure 3-1.

NOTE

It often saves time to use the LIST ERROR LINES ONLY option and to disable the object output the first time a program is assembled. Any errors can be corrected and assembly repeated. When a no-errors assembly is achieved, a full listing and object output can be specified.

4. Mount the source program tape in the selected reader. If a low-speed reader is used make the following control settings:

   ASR-33 Turn punch OFF

   ASR-35 Turn mode switch to KT position

5. Turn to RUN and press START. The Assembler reads the source tape.

6. The computer halts after reading the source tape. If the A register = 0, the MOR pseudo-operation was present. Place additional source tapes on the input device and press computer START. When the computer halts with A = '177777, pass 1 is complete (the END pseudo-operation was encountered). If there are errors, discontinue assembly and make the corrections.

7. To begin pass 2, return the source tape to the beginning and press START. The assembler will read the source tape again. If the program consists of several tape segments, the CPU will halt for each tape, as in pass 1. To resume pass 2, it is only necessary to press START.

8. During pass 2, the assembler will output an object tape, and, if specified, print a listing. While object tape is being punched, the assembler first reads a section of source tape, then punches a section of object tape, and so on, until the entire program is processed. If the low-speed punch is used to punch object tape, the CPU will halt to let the operator turn the punch on and off. The sequence is:

   a. Operator turns low speed punch OFF, presses CPU START. Assembler reads a section of source tape, then halts.

   b. Operator turns low speed punch ON, presses CPU START. Assembler punches a section of object tape, then halts.

   ------- and so on.

ACTION OF ASSEMBLER

PMA is a two-pass assembler that reads the source program twice - once to generate a symbol table and identify external references; and a second time to generate object code blocks for input to the linking loader. During the second pass, a listing output is optional.

During operation under DOS-DOS/VM when the source file is on disk, the assembler automatically returns to the beginning of the source file for the second pass. In paper tape systems, the assembler halts after the first pass and the user must rewind the source program to the beginning before starting the second pass.

ASSEMBLER MESSAGES

When the assembler reads the END statement of the input file on the second pass, it prints a message and terminates assembly (returns to DOS or, in paper tape systems, halts the CPU.) The message contains a decimal error count and version of the assembler, as in:

0001 ERRORS (PMA-1080.011)

A REGISTER (DETAILS)

Error Listing (Bit 2): If this bit is set, only the lines containing errors are listed. Otherwise, listing is controlled by pseudo-operations in the source program.

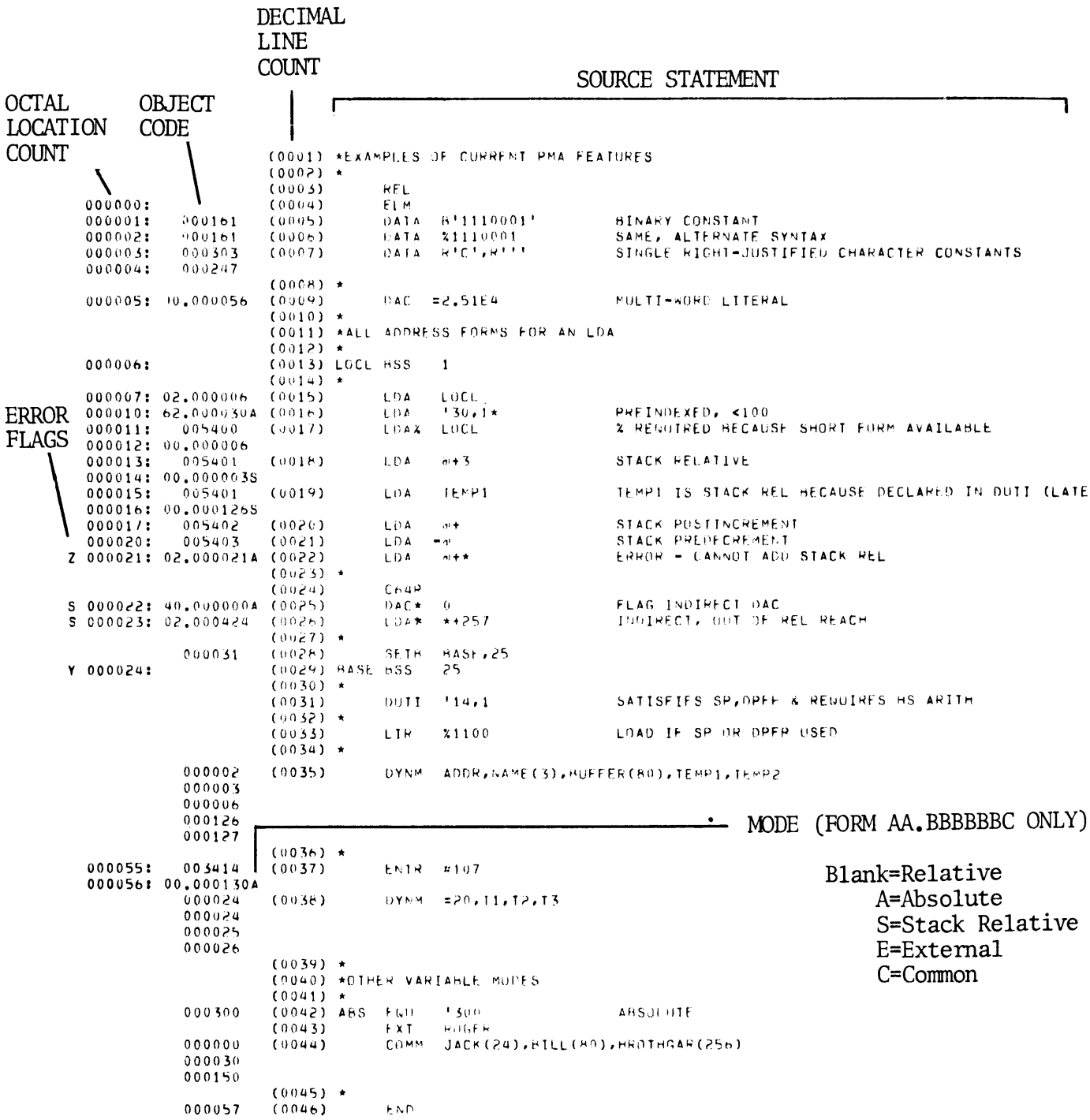
Listing Control Override (Bit 3): If this bit is set, the assembler overrides any listing control pseudo-operations in the source program and lists all statements, including lines within macro expansions and lines that would be skipped by conditional assembly. Otherwise, listing is controlled by the listing pseudo-operations in the source program.

Device Options (Bits 8-16): The last three octal digits of the A register select source input, listing input, and object output devices respectively, as shown in Figure 3-1.

LISTING FORMAT

Figure 3-2 shows a section of a typical assembly listing and defines the main features. The format is organized in columns, but when long labels or other free format features are encountered, extra space is used as required.

Each page begins with a header and a sequential page number. The first statement in a program is used as the initial page header. If column 1 of any statement contains an apostrophe ('), columns 2-72 of that statement become the header for all pages that follow until a new title is specified.




```
                                (0001) *EXAMPLES OF CURRENT PMA FEATURES
                                (0002) *
                                (0003)       REL
        000000:                 (0004)       ELM
        000001:    000161       (0005)       DATA  B'1110001'          BINARY CONSTANT
        000002:    000161       (0006)       DATA  %1110001            SAME, ALTERNATE SYNTAX
        000003:    000303       (0007)       DATA  R'C',R'!'           SINGLE RIGHT-JUSTIFIED CHARACTER CONSTANTS
        000004:    000247
                                (0008) *
        000005:  10.000056      (0009)       DAC   =2.51E4             MULTI-WORD LITERAL
                                (0010) *
                                (0011) *ALL ADDRESS FORMS FOR AN LDA
                                (0012) *
        000006:                 (0013) LOCL  BSS   1
                                (0014) *
        000007:  02.000006      (0015)       LDA   LOCL
        000010:  62.000030A     (0016)       LDA   '30,1*              PREINDEXED, <100
        000011:    005400       (0017)       LDA%  LOCL                % REQUIRED BECAUSE SHORT FORM AVAILABLE
        000012:  00.000006
        000013:    005401       (0018)       LDA   *+3                 STACK RELATIVE
        000014:  00.000003S
        000015:    005401       (0019)       LDA   TEMP1               TEMP1 IS STACK REL BECAUSE DECLARED IN DUTI (LATER)
        000016:  00.000126S
        000017:    005402       (0020)       LDA   *+                  STACK POSTINCREMENT
        000020:    005403       (0021)       LDA   -*                  STACK PREDECREMENT
      Z 000021:  02.000021A     (0022)       LDA   *++                 ERROR - CANNOT ADD STACK REL
                                (0023) *
                                (0024)       CHAR
      S 000022:  40.000000A     (0025)       DAC*  0                   FLAG INDIRECT DAC
      S 000023:  02.000424      (0026)       LDA*  **257               INDIRECT, OUT OF REL REACH
                                (0027) *
                  000031        (0028)       SETB  BASE,25
      Y 000024:                 (0029) BASE  BSS   25
                                (0030) *
                                (0031)       DUTI  '14,1               SATISFIES SP,OPFF & REQUIRES HS ARITH
                                (0032) *
                                (0033)       LIR   %1100               LOAD IF SP OR OPFR USED
                                (0034) *
                  000002        (0035)       DYNM  ADDR,NAME(3),BUFFER(80),TEMP1,TEMP2
                  000003
                  000006
                  000126
                  000127
                                (0036) *
        000055:    003414      (0037)       ENTR  #107
        000056:  00.000130A
                  000024        (0038)       DYNM  =20,I1,I2,I3
                  000024
                  000025
                  000026
                                (0039) *
                                (0040) *OTHER VARIABLE MODES
                                (0041) *
                  000300        (0042) ABS   EQU   '300                 ABSOLUTE
                                (0043)       EXT   ROGER
                  000000        (0044)       COMM  JACK(24),BILL(80),BROTHGAR(256)
                  000030
                  000150
                                (0045) *
                  000057        (0046)       END
```

Figure 3-2. Example of Assembly Listing

Columns 1 and 2 are reserved for error flags. Each flag is a single character, interpreted as shown in Table 3-1. Two flags may be combined (e.g., "FZ"). Columns 3-9 contain an octal location count. Columns 10-18 contain an octal representation of the contents of the location, in one of two formats. Non-memory-reference instructions and all data values are represented by six octal digits corresponding to a 16-bit binary value. Memory reference instructions and address constants are in the following format:

AA.BBBBBBC

The first two digits (AA) represent a six-bit binary field consisting of the indirect bit, the index bit, and (for memory reference instructions) the four op-code bits. The next six digits (BBBBBB) represent the displacement field of the instruction or a 16-bit address value. The last digit (C) indicates the mode of the address value:

Blank	Relative
A	Absolute
S	Stack-relative
E	External
C	Common

Columns 22-27 contain a decimal line sequence number and columns 29-108 contain the source statement (ASCII image) truncated if necessary because of printer limitations.

User-generated messages may be inserted into the listing output by SAY pseudo-operations in the source program itself. Such messages can be used to document the progress of a complex conditional assembly operation.

CROSS REFERENCE LISTING (CONCORDANCE)

At the end of the assembly listing appears a cross-reference listing of each symbol's name (in alphabetical order), the symbol's location or address value, and a list of all references to the symbol. (See Figure 3-3.) The location and address values are in octal unless the PCVH pseudo-operation specifies hexadecimal listing. Each reference is identified by a 4-decimal-digit line number. If listing is inhibited by the NLST pseudo-operation, the cross-reference is not listed.

Table 3-1. Assembler Error Messages

Code	Definition
C	Instruction not terminated properly.
F	Unrecognized operand type, or FAIL pseudo-op executed.
G	Improper GO TO reference, or END or ENDM within a skip area.
I	Improper indirect flag.
L	Improper label, or external label in a literal, or missing label.
M	Multiply defined.
N	END within a Macro definition or an IF area.
O	Unrecognized Operator.
P	Parentheses mismatched or nested more than 7 deep.
Q	ENDM not within a Macro definition.
R	Expression stack overflow, or improper Macro name.
S	Address out of range (LOAD mode), or improper string termination.
T	Symbol table overflow.
U	Variable undefined, or not previously defined when required to be.
V	Value is too large for field, has undefined variable, is missing, is illegal type, or END pseudo-op is within a Macro definition.
X	Improper index tag, or improper external name.
Z	Address Mode Error

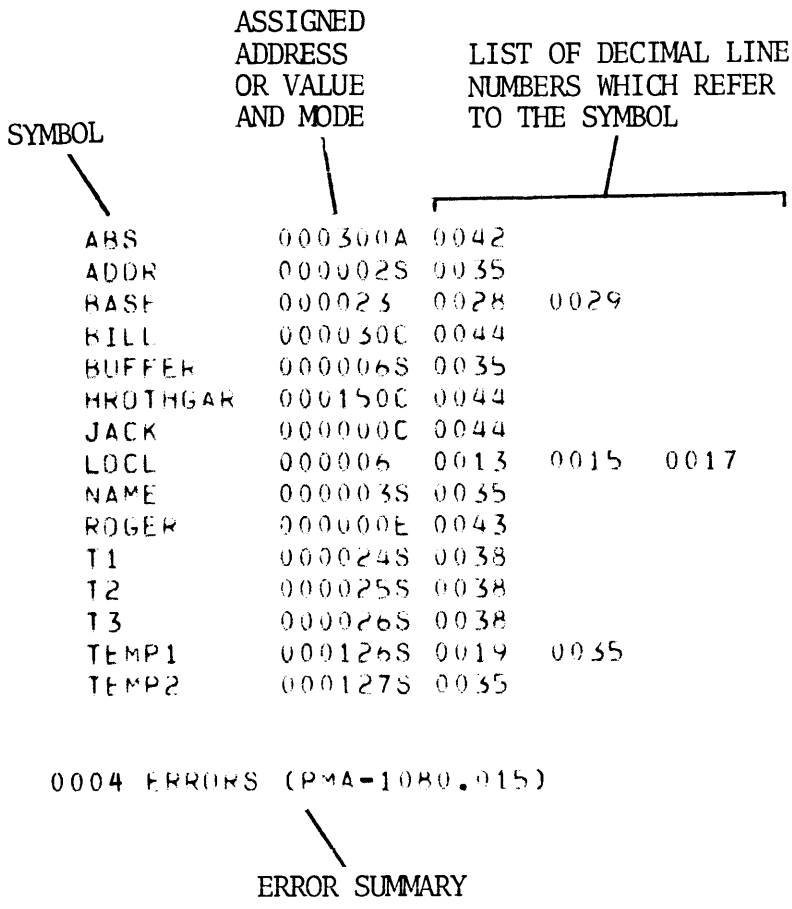


Figure 3-3. Example of Symbol Cross-Reference Listing

The information necessary for the cross-reference listing is stored in the symbol table. If, during assembly, the symbol table becomes full, cross-reference information is sacrificed in order for assembly to continue. The cross-reference listing then contains only the alphabetic symbol names and their assignment addresses.

The last line of the concordance specifies the version of the assembler and the number of lines containing error flags.

SECTION 4

FORTRAN COMPILER (FTN)

Prime's FORTRAN IV Compiler processes source programs prepared in USA Standard FORTRAN, as defined in American National Standard ANSI X3.9-1966. In addition, many powerful extensions improve the language's usefulness in writing high-level programs such as disk or real time operating systems.

The one-pass compiler is compatible with Prime's Disk Operating Systems and Real Time Operating System and is able to run in a stand-alone environment as well. The compiler produces highly optimized code and is supported by an extensive array of mathematical functions and subroutines.

Object code generated by the compiler is in a binary block format suitable for loading by Prime's Linking Loader. Library subroutines are supplied in the same format.

SOURCE PROGRAMS

Source programs must meet the requirements of the Prime FORTRAN IV Language Reference Manual.

A source program is typically prepared at a system terminal, using the Prime text editor to enter the text, make insertions and deletions, and correct errors. The resulting ASCII source file is stored on disk, punched on paper tape, or recorded on magnetic tape. Source programs may also be keypunched for input through a card reader.

OPERATION UNDER DOS-DOS/VM

Loading and Starting Compiler

The FORTRAN compiler is loaded and started by the FTN external command to DOS:

```
FTN Filename [1000 Areg]
```

where 'Filename' is a FORTRAN source program in the current UFD, '1000 is the compiler starting address, and 'Areg' is an A register setting that specifies listing detail and input-output devices.

An alternate command format is:

FTN Filename [1/Areg]

This leaves the default starting address unaltered and modifies the A register value only.

A Register Setting

If no A register setting is specified in the FTN command, the compiler uses default values set up in the DOS RVEC vector at the time the compiler was installed on the disk. Typically, the default setting is '1707 (list errors on user's terminal, no listing file, input and object output files on disk). If in doubt, resume FTN and do a PM (post mortem) to determine the A register default value. Other A register values may be set by providing parameters when FTN is started, as in:

FTN Filnam 1/41777

41777 is an A register value that specifies listing of symbolic instructions, causes errors to be listed on the user's terminal, and uses the disk for input, listing and object files. For other combinations, see Figure 4-1.

File Usage

When disk is used for the binary and listing files, filenames must be established. Unless the FTN command is preceded by BINARY and LISTING commands, the compiler will automatically open unit 2 to write a binary file named B←XXXX and open unit 3 to write a listing file named L←XXXX, where XXXX is the first four letters of the input filename. The compiler closes any units that it opens. (Units opened by BINARY and LISTING commands are not closed.)

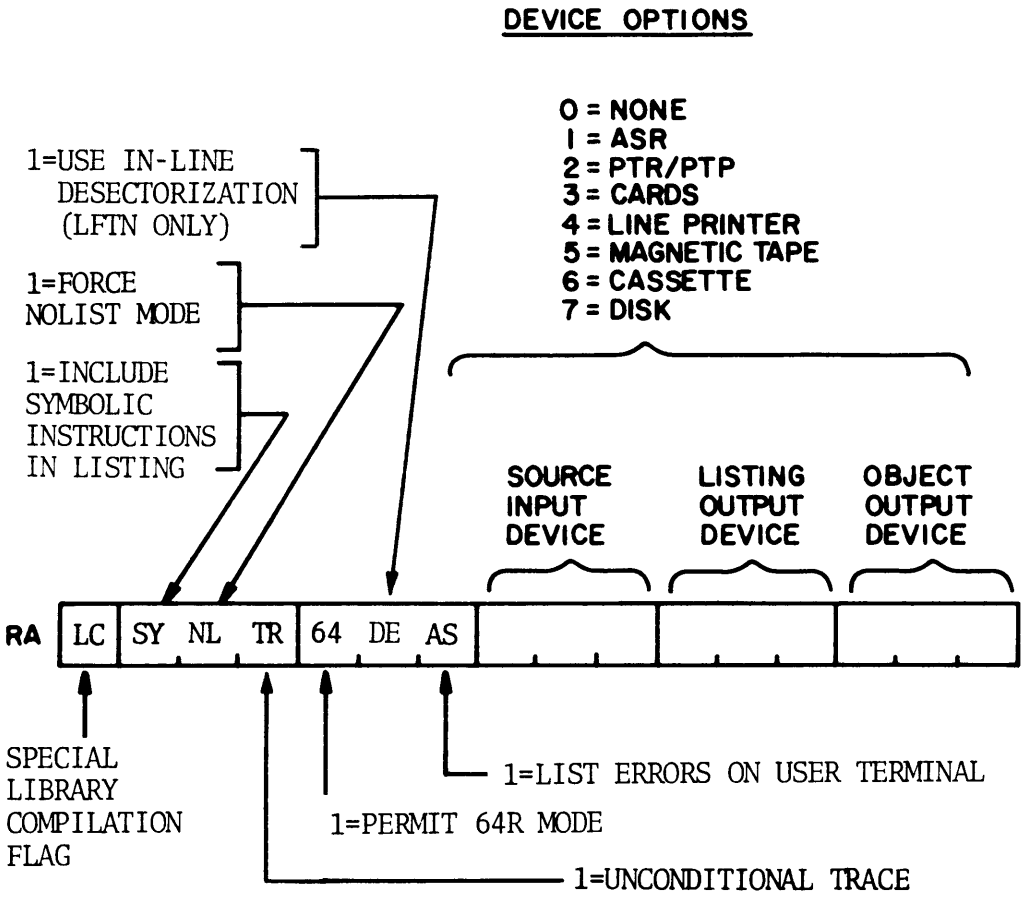


Figure 4-1. Compiler A-Register Settings

USING PAPER TAPE COMPILER

The Prime FORTRAN compiler is supplied as a self-loading tape (SLT 1082.406) for use in systems with 12K or more of memory.

Procedure

1. Use APL (or key-in loader) to load the compiler.

2. Mount the FORTRAN IV source program tape on the desired input device.

3. MASTER CLEAR the processor, then set device selection and listing option codes into the A register as shown in Figure 4-1.

4. Turn to RUN and press START to begin compilation. (Compiler starts at '1000.)

Device Selection

The input device may be the ASR, a card reader, or the high-speed paper tape reader. The listing and object file devices may be different; for example, the listing could be directed to the ASR and the object output to the high-speed tape punch. Typically, the user will generate listings only until an error-free compilation is achieved, and then punch an object tape.

ACTION OF COMPILER

The compiler does a one-pass compilation of the specified input file, and generates object and listing outputs to the devices specified by the A register. A message is printed on the user's terminal after each END statement. The object file is in relocatable binary block format, suitable for processing and loading by Prime's Linking Loader. The object output is compiled to run in 32R addressing mode unless bit 5 of the A register is set, which permits running in 64R mode.

COMPILER MESSAGES

When the compiler reads the END statement of the source program, it prints a message and the version of the compiler on the user's terminal. In DOS-DOS/VM systems, control then returns to DOS command level. In paper tape systems, the CPU halts.

The NO ERRORS message indicates that the program has been compiled without errors. If any errors are encountered, the message ERRORS is printed. If bit 7 of the A register is set, error lines are printed on the console Teletype. Otherwise, the user must print the listing file to find where the errors occurred. An end of file also terminates compilation.

A REGISTER (DETAILS)

Device Options: An essential function of the A register setting is to tell the compiler which device contains the source file, and where to output the listing and object code. The default values are:

Type of Compiler	Source	Listing	Object
Paper Tape	(Must be specified)		
DOS	Disk	None	Disk

Special Library Flag (Bit 1): When this bit is set, the compiler accepts two special arithmetic assignment statements that load or access Prime CPU accumulator registers. (Several library routines use this feature.) The first form is:

=Expression

The compiler evaluates the expression and loads the appropriate CPU accumulator according to the mode of the result:

Result Mode	CPU Accumulator
Integer	A Register
Real	Registers 4, 5, 6
Double Precision	Registers 4, 5, 2, 6
Complex	Floating accumulators AC1, AC2, AC3, AC4

Any of these accumulators can be accessed by a statement of the form:

Var=

which loads the variable 'Var' with the contents of the appropriate accumulator. (i.e., if 'Var' is integer mode, it is loaded from the A register.)

Listing Detail (Bits 2 and 3): These bits determine the content of the listing output:

Bit 02	Bit 03	Listing Detail
0	0	Source statements and error messages, if any.
1	0	Same as above plus an assembly-language-like listing of the instructions generated to implement each source statement.
0	1	No listing.
1	1	Assembly-language-like listing and errors only (no source statements).

LIST, NO LIST or FULL LIST statements in the source program will override these bit settings.

Unconditional Trace (Bit 4): If this bit is set, it has the effect of an unconditional TRACE statement. During compilation, all arithmetic statements, IF statements, and statement numbers result in object coding that will cause trace printouts at run time. If the bit is zero, such printouts occur under control of TRACE statements only.

64 Mode (Bit 5): If this bit is set, the object output is compiled so that it can be loaded to run in 64R addressing mode (i.e., does not generate code that requires multilevel indirects). (32R mode is the default value.)

In-Line Desectorization (Bit 6): This option reduces the sector zero requirements of large programs. When bit 6=1, the compiler generates double-word memory reference instructions and uses the second word as an indirect link for all references to the same item within the relative reach. Use of this option reduces sector zero usage by 70 to 80% while increasing the program size 5 to 10%. Programs compiled with this option can only be loaded in the relative addressing modes (a loader NS diagnostic is generated if an attempt is made to load in a sectored addressing mode).

The compiler named FTN in CMDNC0 on the master disk does not include the ILD option. The version LFTN includes the option but, because of its size, cannot be run in a 16K DOS environment.

List Errors (Bit 7): If this bit is set, errors detected during compilation are listed on the user's terminal. This feature is especially useful when a corrected program is being recompiled, to confirm that the errors have been corrected properly.

SOURCE PROGRAM LISTING FORMATS

Listings may be obtained at several different levels of detail. If A register bit 7 is set, errors only are listed on the user's terminal. (Figure 4-2.)

Under control of A register bits 2 and 3, source program listings can be generated with optional assembly-language-like code. (Figures 4-3 and 4-4).

ERROR MESSAGES

Coding errors and misprints are flagged on the listing by a line containing a set of asterisks (to attract attention) and a 2-character error message positioned under the source statement at the point where the error was detected. If the row of asterisks begin with "↑↑", the error is in a previous line. If the message ends in EQ, the error is in an EQUIVALENCE statement. Compiler error messages are summarized in Table 4-1.

LIBRARY ERROR MESSAGES

Some of the library routines have the capability of detecting fault conditions and delivering error messages during program execution. Error messages generated by library subroutines are summarized in Table 4-2.

TRACE PRINTOUTS

At object program run time, any trace coding inserted by the compiler causes a line to be typed consisting of a variable name, an array name,

or a statement number, followed by an equal sign, followed by the current decimal value assigned to that name. The decimal value is typed in INTEGER, FLOATING POINT, or COMPLEX format. Array names do not specify subscripts. See Figure 4-5 for sample lines of trace information as typed at object run-time.

For some logical IF statements, TRACE will evaluate expressions and print a numerical result rather than a relational value. The user can then interpret the relational condition by inspection.

```
OK, FTN FTNERR
GO
      READ(1,10),D
************       CH
      IF (D.EW.0)GO TO 50
************     CH
      ETAOIN SHRDLU
**********ID
      A+R=D/2.
********PA**
     @C=PI*/D
^^****OP****
      C=PI*/D
************OP
      DOUBLE PRECISION PI,A,R,C,D
**********EX
20    CALL EXIT
**********MS

 ** ERRORS (FTN-1082.008).

OK,
```

Figure 4-2. User Terminal Error Printout Example

```
C     DEMO FORTRAN PROGRAM
C
C
      PI=3.1415926536
1     WRITE(1,A)
5     FORMAT('DIAMETER=
      READ(1,10),D
************       CH
10    FORMAT(F20,12)
      IF (D.EW.0)GO TO 50
************     CH
      ETAOIN SHRDLU
**********ID
      A+R=D/2.
********PA**
      A=PI*R**
----****OP****
      C=PI*/D
************OP
      WRITE(1,20)A,C
20    FORMAT('AREA=',F20.12,    CIRCUMF=',F20.12)
      GO TO 1
      DOUBLE PRECISION PI,A,R,C,D
**********EX
20    CALL EXIT
**********MS
      END
$0    END
** ERRORS (FTN-1082.008)
```

Figure 4-3. Example of Brief Listing (LIST Statement or A Bits 2, 3 =0)

```
C     DEMO FORTRAN PROGRAM
C
C
      DOUBLE PRECISION PI,A,R,C,D
      PI=3.1415926536
000000     ELM
000001     JMP    000000
000002     LINK   000001
000002     FLD    ='062207
000004     CRB
000005     DFST   PI
1      WRITE(1,5)
000007     LDA    ='000001
000010     JST    F$WA
000011     DAC    _5
000012     JST    F$CB
5      FORMAT('DIAMETER= ')
000013     LINK   _5
000013     JMP    000000
000014     OCT    124247
000015     OCT    142311
000016     OCT    140715
000017     OCT    142724
000020     OCT    142722
000021     OCT    136640
000022     OCT    123651
000023     LINK   000013
      READ(1,10)D
000023     LDA    ='000001
000024     JST    F$RA
000025     DAC    _10
000026     JST    F$A6
                  000004
000160     OCT    000000
000161     LINK   ='000002
000161     OCT    000002
000102     DAC    _20
$0     END
NO ERRORS (FTN-1082.008)
```

Figure 4-4. Example of Assembly-Like Listing (FULL LIST Statement or A Bit 2 = 1 and Bit 3 = 0)

Table 4-1. Compiler Error Messages

Code	Definition
AR	Item not an array name.
BD	Code generated within a block data subprogram.
BL	Block data not first statement.
CE	Constant's exponent exceeds 8 bits (Over 255).
CH	Improper terminating character (punctuation).
CM	Comma outside parenthesis, not in a DO statement.
CN	Improper constant (data initialization).
CR	Illegal common reference.
DA	Illegal use of dummy argument.
DD	Dummy item appears in an equivalence or data list.
DM	Data and Data Name mode do not agree.
DT	Improper DO termination.
EC	Equivalence group not followed by comma or CR.
EQ	Expression to left of equals, or multiple equals.
EX	Specification statement appears after cleanup.
FA	Function has no arguments.
FD	Function name not defined by an arithmetic statement.
FS	Function/Subroutine not the first statement.
HD	Hollerith string too long in DATA statement.
HS	Hollerith data string extends past end of statement.
IC	Impossible common equivalencing.
ID	Unrecognizable statement.
IE	Impossible Equivalence grouping.
IF	Illegal IF statement type.
IN	Integer required at this position.
IO	Error in Read/Write statement syntax.
IT	Item not an integer.
MM	Mode mixing error.
MO	Data pool overflow.
MS	Multiply defined statement number.
NA	Name required.
NC	Constant must be present.
ND	Wrong number of dimensions.
NE	No END statement prior to Control statement.
NS	Subroutine name not allowed.
NT	Logical NOT, not an unary operator.
NU	Name already being used.
OP	More than one operator in a row.
PA	Operation must be within parenthesis.
PH	No path leading to this statement.
PR	Parenthesis missing in a DO statement.
PW	* preceded by an operator other than a *.
RL	More than 1 relational operator in a relational example.
RN	Reference to a specification statement's number.

Table 4-1. (Cont)

Code	Definition
RT	Return not allowed in main program.
SC	Statement number on a continuation card.
SP	Statement name misspelled.
ST	Illegal statement number format.
SU	Subscript incrementer not a constant.
TF	"TYPE" not followed by "FUNCTION" or List.
TO	Assign statement has word TO missing.
UO	Multiple + or - signs, not as unary operators.
US	Undefined statement number.
VD	Symbolic subscript not dummy in dummy array, or symbolic subscript appears on a non-dummy array.

Table 4-2. FORTRAN Library Error Messages

Code	Routine	Explanation
AT	ATAN2	ARG = ARG2 = 0
BN	F$BN	device error in REWIND
DE	F$FLEX	double precision exponent overflow
DECODE FORMAT/DATA MISMATCH (literal)		
DL	DLOG/DLOG2	ARG $\leq$ 0
DN	F$DN	device error in ENDFILE
DT	DATAN2	ARG1 = ARG2 = 0
DZ	F$FLEX	double precision divide by zero
EX	DEXP, EXP	exponential overflow
FE	F$IO	syntax error in FORMAT
FN	F$FN	device error in BACKSPACE
II	E$11	exponential overflow
LG	ALOG/ALOG10	ARG $\leq$ 0
READ FORMAT/DATA MISMATCH (literal)		
RI	C$21, F$FLEX	ARG >32767
RN	F$RN	device error in READ
SE	F$FLEX	single precision exponent overflow
SQ	SQRT	ARG < 0
SZ	F$FLEX	single precision divide by zero
WN	F$WN	device error in WRITE
XX	C$21G	ARG >32767

NOTE: The routine F$FLEX is never explicitly called. It is the handler for the hardware floating point exception interrupt.

PROGRAM TEXT		RESULTING TRACE PRINTOUT
	DO 1 J = 1,100	X= 1.000000
	DO 1 I=1,100	X= 2.000000
	TRACE X	X= 3.000000
	X=I	X= 4.000000
	TRACE X	(etc.)
	X=X/50.	
1	A(I)=SIN(X)	

ITEM TRACE

	DO 1 J = 1,100	X= 1.000000
	DO 1 I=1,100	X= 0.2000000E-01
	TRACE 2	(1)
	X=I	A= 0.1999865E-01
	X=X/50.	X= 2.000000
1	A(I)=SIN(X)	X= 0.4000000E-01
	CALL CLKOFF(I)	(1)
	WRITE (1,2) I	A= 0.3998931E-01
2	FORMAT (///7HTIME = I6)	(etc.)

AREA TRACE

C	BENCHMARK PROG II	(5)
	DIMENSION A(100)	X= 1.000000
5	CALL CLKON	X= 0.2000000E-01
	DO 1 J = 1,100	(1)
	DO 1 I=1,100	A= 0.1999865E-01
	X=I	X= 2.000000
	X=X/50.	X= 0.4000000E-01
1	A(I)=SIN(X)	(1)
	CALL CLKOFF(I)	A= 0.3998931E-01
	WRITE (1,2) I	X= 3.000000
2	FORMAT (///7HTIME = I6)	X= 0.5999999E-01
	CALL EXIT	(etc.)
	GO TO 5	
	END	

UNCONDITIONAL TRACE
(A Bit 4 set during compilation)

Figure 4-5. Example of TRACE Printouts

SECTION 5

LINKING LOADER (LOAD)

FEATURES

Prime's linking loader offers the following advanced features:

1. Operator control of the loading operation is greatly simplified. The loader accepts command lines at the user's terminal instead of requiring multiple starting options.

2. The loader is capable of loading code anywhere in 64K, above or below itself or COMMON.

3. COMMON is movable by a keyboard command.

4. An indefinite number of linkage areas can be specified; the loader automatically uses an available area which can be reached directly rather than Sector 0.

5. The user can specify the instruction execution hardware available in the CPU on which the loaded program will execute. This is coordinated with the UII object blocks in load modules so that the proper VIP library routines will load automatically.

6. Partial or full load maps can be printed.

USING LOADER UNDER DOS-DOS/VM

Several versions of the loader are provided on DOS master disks, to match the user's memory availability. The desired version is loaded and started by the external command name listed in Table 5-1. No parameters are required with the command name; all loader functions are available through user terminal keyboard commands. When loaded, the loader prints the "$" prompt character on the user terminal and awaits a command line.

USING PAPER TAPE VERSIONS

The paper tape version of the loader, LOADAP, is supplied in both self-loading and object versions. The self-loading tape is loaded by APL or the key-in loader; when loading is complete, LOADAP takes control, prints the "$" prompt character, and awaits a command line. See Section 9 for instructions on loading the object version into different memory areas than that occupied by the self-loading tape.

Table 5-1. Loader Versions and Memory Locations

Version*	Low	High	Start	Common
LOAD	60000	63777	61000	63752
LOAD40	40000	43777	41000	43752
LOAD74	74000	77777	75000	77752
LOAD20	20000	23777	21000	23752
LOADAP**	14000	17770	15000	17752

* DOS-DOS/VM
external command name
**Paper tape loader

Each loader command consists of a command name followed by a series of arguments in the same format as the Prime DOS-DOS/VM command line:

```
COMMAND   Name1   Name2    Arg1   Arg2 . . .Arg n
```

where COMMAND is the command name, each 'Name' is a DOS filename or UFD name, and each 'Arg' is an octal argument of up to six octal digits. Command names may be abbreviated to two characters. Arguments are separated by spaces. Up to three alphanumeric names and nine arguments are allowed. It is possible to skip the names and follow the command with the first numerical argument. The kill character (?) may be used to cancel a command line containing errors but the erase character (') is not accepted.

The commands are described below in alphabetical order.

ATtach [Ufd] [Password] [Ldisk] [Key]

Enables the user to attach to different UFD's. (Also see LIbrary command.) This command is converted into a CALL to the DOS subroutine ATTACH and has exactly the same effect. If the 'Ldisk' parameter is omitted, the loaded searches only device 0 for the specified UFD. If an Ldisk value of '100000 is specified, the loader searches all started devices in logical unit order. The values for 'Key' most likely to be useful during loading are:

1 Adopt named UFD as home UFD

0 Do not change home UFD

If the 'Ufd' parameter is blank, ATTACH attaches to the home UFD.

Do not use this command in paper tape systems.

COmmon Address

Moves the starting location of FORTRAN-compatible COMMON to the address specified. Space for COMMON items is allocated downward from the starting location. Default values for the start of COMMON are shown in Table 5-1.

EXecute [Areg] [Breg] [Xreg]

Enables the user to start execution of the loaded program with optional values preset into the A, B and X registers. Execution starts at the location specified by the START entry of the load map.

FOrce Filename [Loadpoint] [Linkstart] [Linkrange]

Has the same effect as a LOAD command.

HArdware Definition

Defines the instruction execution hardware of the CPU on which the loaded program will operate. Any item specified by this command is removed from the UII requirement. The 'Definition' parameter is the octal equivalent of a 16-bit word with the following bit assignments:

Bit	Hardware Available on Target CPU
1-12	(Must be zero)
13	1 = Double Precision Floating Point
14	1 = Single Precision Floating Point
15	1 = PRIME 300 Instruction Set
16	1 = High Speed Arithmetic

The default value is zero.

INitialize [Filename] [Loadpoint] [Linkstart] [Linkrange]

Initializes the loader and then performs the same actions as a LOad command. In the loader's initialized state the symbol table is empty and the following parameters are returned to their default values:

Load Map

*START	0	
*LOW	177777	
*HIGH	0	
*PBRK	1000	
*CMLOW	XX752	XX = Last Sector
*CMHIGH	XX752	Occupied by Loader
*SYM	YY000	YY = First Sector
*UII	0	Occupied by Loader

Load Parameters (if not specified

Loadpoint	'1000
Linkstart	'200
Linkrange	'600

New load parameters may be assigned by the command string.

LOad Filename [Loadpoint] [Linkstart] [Linkrange]

Loads the specified object file ('Filename') into memory starting at 'loadpoint' (if specified) or else at the current *PBRK location. The optional 'Linkstart' and 'Linkrange' parameters enable the user to define a linkage area as in a SEtbase command. When loading is complete, *PBRK points to the location following the highest location used by the object file. The other load map and load parameters are altered as required. During the first LOad command after the loader is started, all parameters have the values specified for the INitialize command.

In paper tape systems, do not specify a filename. The 'Loadpoint', 'Linkstart' and 'Linkrange' parameters are optional.

LIbrary [Filename]

Temporarily attaches to the LIB UFD, loads from the specified filename, and returns to the original UFD. FTNLIB is the default filename.

Loading of the library components begins at the *PBRK location of the load map. To begin loading at another location, ATtach to LIB and use the LOad command with a new loadpoint specified.

In paper tape systems, use the LOad command instead of LIbrary. Position the library tape at the beginning and do not spcify a filename.

MAp Option

Prints on the user's terminal part or all of a load map consisting of three sections -- the load state, linkage area information, and unsatisfied references. The 'Option' parameter selects what is to be printed:

Option	
Null	Full map
1	Load state only
2	Load state and linkage information
3	Unsatisfied references only

The eight parameters included in the load state are:

*LOW	=	the lowest location in memory loaded
*HIGH	=	the highest location in memory loaded
*START	=	the location at which execution will begin
*PBRK	=	the next location in memory to be loaded
*CMLOW	=	the lowest location in COMMON
*CMHIGH	=	the highest location in COMMON
*SYM	=	the lowest location used by the symbol table
*UII	=	the net hardware/UII package requirement (see HArdware command for meaning)

(See INitialize for default values.)

Each linkage area is described as follows:

*BASE XXXXXX YYYYYY ZZZZZZ WWWWWW

XXXXXX	=	lowest location defined for this area
YYYYYY	=	next available location if starting from XXXXXX
ZZZZZZ	=	next available location if starting from WWWWWW
WWWWWW	=	highest location defined for this area

Linkage information consists of every defined label or external reference name printed four per line in the following format:

Namexx NNNNNN (loaded)

or

Namexx NNNNNN** (not loaded)

NNNNNN is a six-digit octal address. The ** flag means the reference is unsatisfied (i.e., has not been loaded). Every map begins with a reference to the special FORTRAN array LIST which is defined as starting at location 1.

Example

Following is a load map for the FORTRAN example described in Section 4.

```
$ MAP
*START  001000    *LOW    000066    *HIGH   012106    *PBRK   012107
*CMLOW  063753    *CMHGH  063753    *SYM    057324    *UII    000000

*BASE   000200  000271  000777  000777
*BASE   002124  002166  002165  002165
*BASE   003107  003151  003150  003150
*BASE   003775  004020  004025  004026
*BASE   004531  004551  004554  004554

LIST    000001    E$61    001162    D$62    001242    S$61    001270
C$12    001307    F$RA    001320    F$RX    001326    F$WA    001460
F$WX    001466    F$IO    001542    F$A1    002076    F$A3    002076
F$A2    002102    F$A5    002102    F$A6    002107    F$CB    002426
F$IOBF  005251    F$FLEX  005353    F$ER    005521    F$HT    005526
AC1     005606    AC2     005607    AC3     005610    AC4     005611
AC5     005612    RDASC   005613    RDALN   005613    WRASC   005620
IOCS$   005625    IOCS$T  005724    F$AT    005736    F$AT1   005740
RATBL   006003    WATBL   006014    LUTBL   006025    PUTBL   006062
RSTBL   006117    I$AD07  006154    O$AD07  006316    O$AD08  006515
I$AA01  006557    I$AP02  006571    O$AA01  006751    O$AP02  006755
PRWFIL  007051    EXIT    007054    ERRSET  007061    OPSCHK  007064
T1IN    007115    T1IB    007216    T1OB    007223    P1IN    007230
P1OU    007251    P1OB    007267    P1IB    007273    UII161  007300

$
```

MOde Mode

Directs the loader to desector in one of the four CPU addressing modes:

Mode Parameter	Addressing Mode
D16S	16K Sectored
D32S	32K Sectored
D32R	32K Relative (default value)
D64R	64K Relative

The mode set by this command may be overridden by mode control pseudo-operations in the object text. If the program contains an ELM (Enter Loader's Addressing Mode) this command enables the user to select the addressing mode at load time.

QUit

Returns to the operating system with the user attached to the home UFD or the last UFD specified in an ATtach command.

(Do not use in paper tape systems.)

REcover

Enables loader to continue following a GT error message. The GT message results from an incorrect filename, an unrecognizable piece of object text, or a missing EOF or EOT. After giving this command, the user can specify the correct file in another LOad or FOrce command and continue loading.

SAve Filename [Areg] [Breg] [Xreg]

Saves the loaded memory image under the name 'Filename' in the current UFD. Also saved with the program are the low, high, start and keys parameters obtained from the loader. (There is no option to set them.) (do not use in paper tape system.)

SEtbase Linkstart Linkrange

Defines a linkage area that begins at 'Linkstart' and includes the number of locations specified by 'Linkrange'. If the range is not specified the end of the area is location '777 of the sector containing the 'Linkstart' location. Multiple linkage area are allowed. A command to create a linkage area that overlaps a previously defined area is ignored.

The default values are:

Linkstart '200

Linkrange '600

VIrtualbase Startlinks Tosector

Copies the base sector (from the 'Startlinks' location to the end) to the corresponding locations of 'Tosector'. This command is intended for use in building RTOS modules using dedicated sector zero or base sector relocation.

LOADER MESSAGES

After executing a command successfully the loader types the "$" prompt character. Under some circumstances one of the following messages may be printed. (Note that the MR message of previous loader versions is no longer issued.)

CM - COMmand error.

Illegal command syntax or nonexistent filename specified.

GT - Group Type error.

The loader has encountered an unrecognizable piece of object text. Loading is discontinued. To continue, enter the REcover command.

LC - Load Complete.

All external references are satisfied. (This does not imply satisfaction of all UII requirements.)

MI XXXXXX - Multiple Indirect.

While linking in 64R mode the loader attempted to add indirection to an already indirect instruction at location XXXXXX. The contents of XXXXXX are the proper flag, tag, and op code with an address of zero. Loading continues.

MO - Memory Overflow.

An attempt has been made to overwrite the loader or its symbol table, or the base sector is full of links. Loading is discontinued.

NS - Never Sectored

Code is being loaded in 16S or 32S mode which will not properly execute in a sectored mode. Loading is discontinued.

N6 - Never 64R mode.

Code is being loaded in 64R mode which will not execute properly. Loading is discontinued.

UII HANDLING (INTERACTION OF LOAD, PMA, AND FTN)

PMA and FTN both output an object group which informs the loader of any need for high speed arithmetic, floating point, etc., in a given module. The object group contains one data word, in the same format as the loader's HArdware command argument. The loader maintains an internal summary of UII requirements for all modules loaded.

UII library modules are headed by an object group containing two data words: the first describes the features offered by the module, and second describes the hardware required. Both words are in HArdware command format. A UII module which does not satisfy the loader's summary of requirements is skipped, not loaded. The *UII value in the load map is the total UII requirement less any requirements satisfied by a loaded UII module (as specified in a HArdware command), or the target hardware. The following example shows the loader commands required to load a program that requires floating point arithmetic and is to run on a CPU that contains the high speed integer arithmetic option. The VIP routines are in the file named UII in the LIB UFD.

```
OK, LOAD
GO
$ LO B_FTND
$ LI
LC
$ MA 1
*START  001000    *LOW    000074    *HIGH   007277    *PBRK   007300
*CMLOW  063753    *CMHGH  063753    *SYM    057331    *UII    000015

$ HA 1
$ MA 1
*START  001000    *LOW    000074    *HIGH   007277    *PBRK   007300
*CMLOW  063753    *CMHGH  063753    *SYM    057331    *UII    000014

$ LI UII
LC
$ MA 1
*START  001000    *LOW    000066    *HIGH   012106    *PBRK   012107
*CMLOW  063753    *CMHGH  063753    *SYM    057324    *UII    000000
```

REPLACING DEFAULT VALUES FOR MODE, COMMON, HARDWARE

It is not necessary to know internal memory locations to adjust the default values for the parameters of the MOde, COmmon, and HArdware commands. The following DOS sequence will suffice:

```
OK:  A CMDNCO
OK:  REST  LOAD
OK:  PM

     SA, EA, P, A, B, X, K =

     ----------------------

OK:  START

GO
$    MODE   new value
$    COMMON  new value          --       As many as
$    HARDWARE   new value                desired
$    QUIT
OK:  SAVE LOAD  (use values from previous post-mortem)
```

LIBRARY MODE

The loader maintains an internal force-load flag which specifies that an entire file (or tape) will be loaded whether or not all entry points have been referenced by a previously loaded module. The force-load flag is set when the loader is initialized and whenever an end-of-file is reached. Only an RFL code at the beginning of an object file (inserted by the binary editor) will clear the force-load flag and establish library mode. In library mode, only the components with previously specified entry points are loaded. Prime library files contain RFL codes to ensure that the user will load only the components he requires.

SECTION 6

DEBUGGING UTILITIES - OCTAL (TAP) AND SYMBOLIC (PSD)

Prime supplies two types of debug programs. TAP (Trace and Patch) is a compact, one-sector octal-mode routine to examine, dump or update programs from the user's terminal. It includes trace and breakpoint insertion features for dynamic debugging under conditions of simulated execution (in sectored addressing modes only). PSD (Prime Symbolic Debugger) is a four-sector version that adds the ability to address up to 64K of memory, and examine, dump and update memory locations in octal, hexadecimal, alphanumeric, binary or mnemonic notation. In mnemonic form, instructions are dis-assembled into an instruction mnemonic and an address value, plus symbols for indirection (*) or indexing (,1). Instructions of the extended classes (long reach, stack relative, push-pop) are identified by a % symbol followed by a class code from 0 to 3, as in LDA %2, which signifies an LDA instruction operating in extended addressing Class 2 (stack postincrement).

PART 1

TRACE AND PATCH (TAP)

TAP is an octal-mode debugging routine that permits the operator to access memory locations, process memory blocks, and trace program execution dynamically, by entering commands and octal values at the teleprinter keyboard. The main functions are summarized below:

Function	Command
Memory Words:	
Access and print or alter contents	A
List (print) contents	L
Update (alter contents)	U
Memory Blocks:	
Copy block to block	C
Print contents	D
Fill with constant	F
Search for constant under mask	S
Verify block to block	V
Not-equal search for constant under mask	N

Executable Programs:

Breakpoint set	B
Execute a subroutine	E
Jump trace (print diagnostic after JMP or HLT instructions)	J
Monitor for effective address (execute program and print diagnostic if address is formed)	M
Patch object program (insert JMP in specified location)	P
Run object program (print diagnostic if breakpoint is reached)	R
Trace object program (print diagnostic at specified intervals)	T

LOADING AND STARTING

Under DOS-DOS/VM: Enter the external command TAP. When loaded, TAP types the "$" prompt character and awaits a command string from the system terminal. To terminate long operations such as Dump, type CTRL P for a return to DOS. Restart at 'XX000, where XX is the first sector occupied by TAP.

Paper Tape Systems: TAP is provided as a self-loading, self-starting system tape that can be loaded from the high or low speed reader using APL. When properly loaded, TAP types the prompt character "$" and awaits a command string.

The relocatable object version of TAP can be loaded at the beginning of any sector except zero. Object programs to be debugged dynamically must be in the same 16K of memory.

To terminate long operations like Dump, halt the CPU and restart at 'XX000, where XX is the first sector occupied by TAP.

Relocating TAP: During program development, it may be useful to load TAP into more than one sector of memory. The following command string replicates TAP in every sector of an 8K memory from location '2000 up:

```
$C 1000 16777 2000 (CR)
```

If a program error wipes out part of the TAP program, a copy of TAP in another sector can be started without having to load again from paper tape.

Startup of Other Programs: TAP permits the operator to start other programs without having to load the P register at the control panel. For example, the command string R 70000 starts execution at location '70000.

Addressing Modes: TAP runs in the 32S addressing mode. If TAP is used to start or trace a program that executes in 16S addressing mode, TAP must be loaded in the first 16K of memory. TAP may not be used to trace programs that execute in 32R or 64R addressing modes. (See PSD.)

COMMAND DESCRIPTIONS

Each TAP command consists of a single letter function code followed by one or more octal values, separated by spaces or commas. Each command string is entered for execution by a CR. The commands are defined below in alphabetical order.

All values are right-justified octal integers. If a value is unspecified, it is considered zero (in the expression "V1,,V3" the omitted value, V2, is considered zero).

A slash (/) or question mark (?) may be used to abort a command string and return to the starting condition (signalled by the $ character).

To cancel an incorrect octal value, type an asterisk (*). If more than five digits are entered, only the last 16 bits are used.

If the wrong function code letter is entered, simply follow it with the correct character. (Only the last input letter of the command field is interpreted.)

Access memory A Startadd

Accesses word(s) in memory starting at 'Startadd'. The program types 'Startadd' and its contents, then waits for keyboard input. To change the contents, key in the new octal value, followed by CR. The program then types out the next higher address and its contents. To progress to the next higher address without changing the contents of the current location, key in a comma or CR. To backspace to the previous location without changing the contents of the current location, key in an up arrow (↑). The look/change cycle continues until the operator keys in a slash (/) or question-mark.

Breakpoint set B Location

Inserts breakpoint link in object program at 'Location'. If object program is later executed, and if control reaches 'Location', an

indirect jump through location '00777 returns control to the TAP program, which prints the register contents, then awaits further commands. Print format is given under function R. Only one breakpoint can be inserted in a program. The actual breakpoint jump is placed in the object program only at execution time, and is removed after each use. However, the breakpoint address is retained for re-use and requires user action only to change it. To remove breakpoint completely, key in B 17 (CR).

Copy memory block to memory — C From To Newblock

Copies memory block at locations 'From' through 'To' into block starting at 'Newblock'. If 'To' does not exceed 'From', only the word at location 'From' is copied. If 'Newblock' lies between 'From' and 'To', the block between 'From' and 'Newblock' is repeated cyclically until location Newblock + To - From is reached.

Dump memory to teleprinter — D From To

Dumps memory block at locations 'From' through 'To' to user terminal. The basic typing format is eight octal words per line, preceded by the octal address of the first word printed on the line. Repetitious words are suppressed as follows:

1. If the remainder of the current line is identical to word last printed, the line is terminated.

2. If one or more subsequent lines are identical to word last printed, one line is skipped.

Execute subroutine — E Subr [Areg Breg Xreg Keys]

Executes a subroutine by performing a JST to location 'Subr'. Prior to subroutine entry, the A, B, and X registers and Keys are optionally preset. The subroutine return should be via indirect jump through its entry point, incremented by 0, 1, or 2.

Fill memory block with constant — F From To Value

Fills memory block at locations 'From' through 'To' with 'Value'. If 'To' does not exceed 'From', only the first location is filled.

Jump trace object program J Startadd [Areg Breg]

Dynamically traces object program starting at location 'Startadd' with an optional preset of registers A and B. A diagnostic printout is produced prior to the interpretive execution of any JMP or JST or HLT. (See function T for format.)

List memory word L Address

Lists contents of 'Address'.

Monitor object program for effective address M Startadd Areg Breg Address

Dynamically monitors object program starting at 'Startadd', with registers A and B Preset. A diagnostic printout is produced prior to the interpretive execution of any object memory-reference instruction with an effective address equal to 'Address'. (See function T for format.)

Not-equal search N From To Nmatch [Mask]

Searches memory block between 'From' and 'To' for words not equal to 'Nmatch' under an optional 'Mask'. The masking function is a 16-bit logical AND. If no mask is specified, the entire word is tested. When a non-match is found, the address and its contents are typed out, and the search continues.

Patch object program P V1 V2

Inserts patch in object program at location V2, by replacing instruction at V2 with jump to location V1, storing the displaced instruction at V1, and entering Access function at location V1. Operator must key in desired patch, with suitable return. Either V1 and V2 must be in the same sector or V1 must be in sector zero.

Run object program R Startadd Areg Breg Xreg Keys

Runs object program by performing JMP to 'Startadd' location. Prior to program entry, registers A, B, X, and Keys are optionally loaded. Control does not return to the TAP program unless a breakpoint is encountered. If a breakpoint is encountered, the print format is:

INSTR (A) (B) (X) (KEYS)

Search memory block under mask — S From To Match [Mask]

Searches memory block at 'From' through 'To' for words equal to 'Match' under an optional 'Mask'. (If no mask is specified, the entire word is tested.) When a match is found, the address and its content are typed out, and the search continues until location 'To' has been tested.

Trace object program — T Startadd [Areg Breg]

Dynamically traces object program starting at 'Startadd' with registers A and B optionally preset. A diagnostic printout is produced prior to the interpretive execution of each object instruction. Printout is formatted as eight octal words, representing:

(P) INSTR EA (EA) (A) (B) (X) (KEYS)

For non-memory-reference instructions, the third word is 000000 and the fourth repeats the instruction word.

T Startadd Areg Breg Pval

Same as above, but printout occurs only when P='Pval'.

T Startadd Areg Breg 177777 Interval

Same as above, but printout occurs every 'Interval' instructions.

If 'Interval' is negative, its absolute value is used.
If zero, it is treated as 65536.

T Startadd Areg Breg Pval 0

Same as above, but printout occurs the first time P='Pval', and every instruction thereafter.

T, J and M Function Restrictions

a. HLT instructions always cause printout, followed by a return to TAP command mode.

b. Interrupts are executed in real time, not in interpretive mode. Tracing resumes when interrupt routine exits.

c. Tracing of input-output routines is possible, but timing should be investigated. Processing speed is reduced by a factor of 60 to 80 when no printout is involved.

d. Programs to be traced can operate only in sectored addressing modes (16S or 32S).

Update memory word — U V1 V2

Sets (V1)=V2. Prints old and new contents of V1.

Verify memory block against copy in memory — V From To Copy

Verifies memory block at 'From' through 'To' against a copy starting at 'Copy'. The program types the address and content of each location in the 'From' block which does not match corresponding word in 'Copy'.

PART 2

PRIME SYMBOLIC DEBUG (PSD)

PSD is an expanded version of TAP that provides the following enhancements:

Mnemonic, binary, ASCII, and hexadecimal input/output

Expressions in input values - with current location count (*) and relocation constant (>) symbols and + or - operators.

All four standard addressing modes (16S, 32S, 32R and 64R)

Full 64K addressability

Relocatable addressing (assembler REL)

Extended-class instructions

New commands

Several new access features, including effective address calculations

PSD includes all of the normal TAP commands except Jump, Patch and Trace, and adds several new commands. Commands which are common to both TAP and PSD operate exactly as described for TAP except for the parameters, which are entered in the current input mode. Output (if any) is printed in the current output mode. In Access mode, there are several new terminators and an address relocation offset value can be specified.

LOADING AND STARTING UNDER DOS-DOS/VM

PSD is supplied in the CMDNC0 UFD of DOS master disks in two versions. The command PSD loads and starts a version that runs in locations '60000-6377 (below DOS in a 32K memory). The command PSD20 loads and starts a version that runs in locations '20000-23777 (below DOS in a 16K memory). PSD is not relocatable as is TAP.

USING PAPER TAPE VERSION

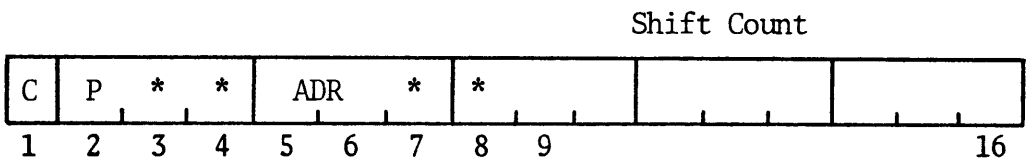
PSD is supplied as a self-loading, self-starting paper tape (SLT0790.000) that loads into locations '14000-17666.

Effective Address Search E From To Match [Mask]

Same as Search (TAP) but prints address/value when an effective address equal to 'Match' under 'Mask' is found.

Keys update K Keys

Sets the CPU status keys to the specified value. The bit assignments are:



where:

C = State of C (Carry) bit

P = Arithmetic mode; 0 - single precision, 1 double precision

* = Must be zero

ADR = Addressing Mode:

Bit 5	Bit 6	Mode
0	0	16K Sectored (Normal)
0	1	32K Sectored
1	1	32K Relative
1	0	64K Relative

Shift Count = Bits 9-16 of location 6, which may contain a normalize shift count

Mode (addressing) MO | D16S | D32S | D32R | D64R |

Selects the addressing mode in which address values are computed in symbolic input/output mode. Sets bits 5 and 6 of the keys accordingly and resets all other keys bits to zero.

Parameters P

Prints CPU/PSD parameters in octal as follows:

```
Breakpoint  Breakpoint  Areg Breg Xreg Keys  Relcon
            Contents
```

'Relcon' is the current value of the relocation constant.

Relocation Constant or RE Value / X Value

Sets a value for the PSD internal relocation constant. To remove, set to 0.

Quit Q or QU

Returns to the operating system.

X Register Setup XR Value

Sets the value of the X register, for example before executing a Run command or effective address calculation.

Breakpoint Processing

PSD has the ability to insert a Breakpoint and regain control when execution of the Breakpoint location is attempted. The Breakpoint location is defined by B and is inserted by R as defined for TAP. When PSD regains control, it prints:

```
bp  (bp)  Areg Breg Xreg Keys Relcon
```

bp is Breakpoint location
(bp) is contents of breakpoint location
Areg is A register at breakpoint
Breg is B register at breakpoint
Xreg is X register at breakpoint
Keys is Keys at breakpoint

Verify Printout

The PSD format for printout during a Verify operation is:

```
Address V1  V2
```

where 'Address' is the location in the 'From' block, V1 is the contents of that location, and V2 is the contents of the corresponding word in 'Copy'.

INPUT/OUTPUT MODES

PSD has the ability to accept input parameters and print output values in five different modes. The mode is established by ending any command with a colon followed by a single letter, as in:

```
A 1000:O
```

This accesses location '1000 and establishes the octal mode for all subsequent input/output. The following mode-changing letters are assigned:

:A	ASCII
:B	Binary
:H	Hexadecimal
:O	Octal
:S	Symbolic (i.e. mnemonic)

The effects during input and output are described below.

ASCII Input

Two characters are accepted followed by a terminator (described later). Any even number of characters will be accepted with the last two as the final value. The first character (or any odd character) may not be:

```
> = @ % , .nl.  / ? + - : * ( ) ↑ . blank
```

The second character is required and may not be:

```
/ ? , .nl.
```

ASCII Output

Two characters are printed - an @ is substituted for any non-printing character. In a Dump, up to 8 character pairs are printed per line.

Binary Input

Any sequence of 1's and 0's are accepted with the last 16 being used for the final value (if less than 16 are input, leading 0's are assumed).

Binary Output

A sequence of 16 1's and 0's is printed. In a Dump, up to 4 words are printed per line.

Hex Input

Any sequence of characters from the set 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, is accepted with the last four being used for the final value (if less than four are input, leading 0's are assumed).

Hex Output

A sequence of four hexadecimal characters is printed. Leading zeroes are supported. In a Dump, up to 8 words are printed per line.

Octal Input

Any expression is accepted (octal number or mnemonic op code).

Octal Output

A sequence of six characters (0-7) is printed with leading 0's replaced by blanks. In a Dump, up to 8 words are printed per line.

Mnemonic Input

Mnemonic input mode enables the user to enter instructions using mnemonic rather than octal op codes. The general form for mnemonic input of an instruction is:

```
Mnem [*]   [%] [>] Expr [,1]
```

where:

Mnem	is any legal instruction mnemonic (a DAC is printed if the op code is not recognized)
*	represents indirect addressing
%	indicates that the instruction is of the extended class
>	specifies that the address expression is relative to the relocation count
Expr	is an expression (forms described later)
,1	specifies indexing

The following examples show the format for most forms of a LDA instruction that addresses absolute location 1017:

```
LDA 1017       Direct addressing
LDA* 1017      Indirect
LDA 1017,1     Indexed
LDA* 1017,1    Indirect and indexed
```

Extended-class instructions (identified by the % symbol) contain, instead of an octal address, one of the following codes to represent the class code in bits 15 and 16 of the instruction word:

```
0     Long reach
1     Stack relative
2     Stack postincrement
3     Stack predecrement
```

The displacement field of the instruction is set to -255+n, where n is the class code.

A relative addressing mode (32R or 64R) must be set by the MOde command in order for extended instructions to be input properly.

The first two classes are the two-word instruction types, for which the second word is exprected to contain an address value. Indirection and indexing can be specified as usual. Examples:

```
LDA %0        Long reach
DAC 1017      Address word

LDA* %0       Same, indirect
DAC 1017

LDA %0,1      Same, indexed
DAC 1017

LDA %1        Stack relative
DAC 100       Offset from stack pointer

LDA* %1       Same, indirect
DAC 100

etc.
```

The stack postincrement and predecrement instructions are one-word types. Examples:

```
LDA %2        Postincrement
LDA* %2       Same, indirect
LDA %2,1      Same, indexed
LDA %3        Predecrement
etc.
```

Mnemonic Output

Mnemonic output is in the same form as the input but the % symbol for extended instructions is shown as part of the mnemonic field and the address value is in octal. Examples:

```
LDA% 0          Long reach
DAC 1017

LDA*% 1,1       Stack relative, indirect & indexed
DAC 1017

LDA% 2          Stack postincrement

LDA*% 3,1       Stack predecrement, indirect and indexed
```

Printouts of consecutive locations during Dump commands are formatted four per line with the octal address of the first item at the beginning of the line. Example:

```
$D 1015 1100:S
  1015 JST     1032,1 EPMJ          HLT          LDA     1017
  1021 LDA     1017,1 LDA*  1017    LDA*  1017   LDA*    1017,1
  1025 LDA*    1017,1 LDA*    30,1  LDA %    0    DAC     1017
  1031 LDA*%      0   DAC    1017   LDA %    0,1  DAC     1017
  1035 LDA*%      2,1 DAC    1017   LDA*%   0,1  DAC     1017
  1041 LDA*%      0,1 DAC    1017   LDA*%   2,1  DAC       30
  1045 LDA %      1   DAC       3   LDA %    1,1  DAC        3
  1051 LDA*%      1   DAC       3   LDA*%   3,1  DAC        3
  1055 LDA*%      1,1 DAC       3   LDA %    1    DAC      126
  1061 LDA %      2   LDA %    2,1   LDA*%   2    LDA %     2,
  1066 LDA %      3   LDA %    3,1   LDA*%   3    LDA %     3,1
  1073 LDA     1073   LDA     170   LDA %    1    HLT
  1077 LDA       77   LDA     100
```

Expressions

An expression is:

a. A signed octal number of up to six digits. If more than six digits are entered, the most recently entered six are kept. Leading zeroes may be omitted and, in the absence of an explicit indicator, + is assumed. Examples:

   +123 ; -765 ; 127102700 (value is 102700)

b. The character * whose value is the Access Mode location counter. (See "Access Mode Enhancements".)

c. An arithmetic expression which specifies the addition or subtraction of any number of expressions of type a or b. Examples:

   *+123 ; *-1 ; *+1000-2

Relocation Constant

PSD has the ability to process addresses in a relocatable mode (equivalent to assembler REL) by maintaining a relocation constant which points to the start of a module. All addresses that are preceded by > are relative to this relocation constant. For a relocation constant of 3121, both

```
$A >0   and $A 3121
```

would open location 3121.

The relocation constant is set by the RE or X command. Setting the relocation constant to 0 disables this mode.

For all output, any address which is larger than the relocation address is printed as > n, where n is the address minus the relocation address.

ACCESS MODE ENHANCEMENTS

PSD provides an Access command with the same general functions as the TAP Access command but with several extensions:

A current location count is maintained.

All input and output modes are allowed.

The relative addressing symbol (>) may be used to specify locations or update their contents.

Several new line terminator functions are added, including effective address formation and a branch/return function for convenient examination of subroutines or data tables.

Current Location Counter

In Access mode, a current location count is maintained, starting with the value of the 'Startadd' parameter of the Access command. The location count determines the next location to be accessed. For the comma or .NL. line terminators, the counter is incremented after each access. Other line terminators provide different options.

PSD replaces the value in the open location with the new value (if specified) and uses the line terminator to compute the next value of the current location counter.

PSD accepts the new value in the current input mode, which may be changed while entering the new value. Thus, :HAF enters the hex value 00AF regardless of the previous mode.

The new value may be terminated by a new line (.nl.) or an up arrow (↑). Other terminators do not cause the new value to be accepted.

Line Terminators

There are several line terminators. Only .nl. or ↑ should follow a new value. The others may be used when no change is entered.

Terminator	Next Location
.nl.	current location +1
↑	current location -1
, (comma)	current location +1
.n(.nl.)	current location +n (n is an octal number)
.-n(.nl.)	current location -n
@	effective address (if the instruction is memory reference) or current location. Saves the current address +1 as the return address.
\	return address defined by last @
=	current address. When = is entered, the effective address is calculated and printed in octal followed by the contents of that location in octal. No change takes place to open location.

Effective Address Formation

PSD processes input and output in all four major addressing modes. The mode is set by the MOde command.

When the index register is needed, the current value of the X register is used (it may be changed by using the Run or XReg commands.

When PSD prints an address, it applies the same address formation process as the hardware, using the current values of the X and S registers. For relative addresses, the Access Mode current location counter is used as the value of the P register.

SECTION 7

TAPE PUNCH AND COPY UTILITIES

PART 1

MEMORY DUMP AND LOAD (MDL)

MDL punches paper tapes of specified sections of memory in a self-loading format that can be read by the automatic program load (APL) function or an equivalent key-in loader. MDL tapes load into the same memory locations from which they are punched.

MDL first punches part of itself, a second-level bootstrap loader in 8-8 format (two tape frames per memory word image) followed by a length of leader. The memory area to be saved is then punched in a 256-word block format.

When the tape is read, the APL loader or key-in loader only needs to load the 8-8 format bootstrap portion of the tape. Regular program control is then transferred to the second level bootstrap, which interprets the block-format data and loads it into memory. An ASR reader is operated in full duplex so the printer is inactive while a self-loading tape is being read.

MDL punches the contents of all memory locations between two specified addresses. The area need not consist of solid code or data, however. Any three or more consecutive identical memory locations are compressed as follows:

Word 1	Pattern to be repeated
2	'70 (escape character)
3	'340 (repeating word flag)
4	Number of occurrences (256 max.)

USING MDL UNDER DOS-DOS/VM

Enter the MDL external command. MDL prints the cue:

SA, EA, P, K, L

and waits for a string of starting parameters to be entered from the keyboard. (See "Entering Parameters".)

USING MDL IN PAPER TAPE SYSTEMS

Versions Supplied: MDL is provided in three versions for the convenience of the user. One version is a self-loading tape of MDL that loads into the last sector of an 8K memory ('17000-'17777). Another self-loading 8K version of MDL, in combination with TAP, is provided under the name of TAPMDL. The combination is loaded into the top two sectors of an 8K memory (locations '16000-'17777). In

addition, an object tape of MDL is provided so that each user can generate a self-loading version suitable to his particular memory configuration and program development methods. Once it is loaded into the desired area of memory, this version of MDL can be used to punch a self-loading tape of itself. MDL occupies one sector ('777 locations) and runs in 64R addressing mode.

Loading Object (Relocatable) Version: Load linking loader (LOADAP) according to the procedures of Section 5, and use it as follows:

1. Mount the MDL object tape on the selected input device and prepare the device for operation.

2. Start the loader and enter the following commands:

```
$ MO D64R
$ LO Startadd
```

   where 'Startadd' is the beginning of the sector in which MDL is to load.

   The MDL tape should load and stop, and the loader should print "LC" (loading complete). Print a memory map to show the starting address for future reference.

NOTE

Use MDL to punch a self-loading tape of itself and this operation will not need to be repeated.

Loading Self-Loading Version: A self-loading tape of MDL or TAPMDL can be loaded from the high-or low-speed tape reader using the automatic program LOAD function or key-in loader. The TAPMDL tape uses the autostart feature; when the tape finishes loading, TAP automatically starts and types the cue "$".

Starting MDL: After MDL is loaded, start it as follows:

1. Set all sense switches OFF.

2. Make sure there is enough tape in tape punch to contain the memory area to be copied. Feed a few folds of leader.

NOTE

Skip the next step if the program to be punched uses any register file locations.

3. MASTER CLEAR the CPU.

4. If the TAPMDL self-loading tape was loaded, start MDL by entering the TAP command R 17000.

NOTE

Start from the panel (Step 5) if the program to be punched uses location '777. The TAP RUN command uses location '777 as a return link, so location '777 of the program may be written over before being punched.

5. If an MDL self-loading tape was loaded, turn to STOP/STEP and set the P register (Location 7) to the address specified on the MDL tape label (typically, XX000, where XX is the highest sector of memory.) Turn to RUN and press START.

6. After MDL is started, it responds by typing the cue:

   SA, EA, P, K, L:

   and waits for a string of starting parameters to be entered at the system terminal keyboard. (See "Entering Parameters".)

Aborting a Punch Cycle: To abort a punching operation, turn sense switch 02 ON momentarily. Punching will stop and the program will request a new set of parameters. Remove the unwanted tape and feed a new leader before restarting.

Entering Parameters from Panel: Parameters can be entered from the control panel rather than the terminal. Before starting MDL, set sense switch 01 and load the parameters into the following CPU registers:

Parameter:	Startloc	Endloc	Autostart	Keys	Bootloc
Location	0	1	2	3	4

Then start MDL at the location specified on the tape. The parameters have the same effect as when they are entered from the terminal keyboard.

ENTERING PARAMETERS

The parameter string consists of five octal values separated by spaces or commas and entered by the CR or LF key. Each parameter is an octal value ranging from 0 to '177777. Leading zeros can be omitted. To correct a typing error, retype the parameter without a space or comma; the last six digits are retained. For a fresh start, strike any non-octal key; the request for parameters will be repeated.

The parameters are:

Startloc Endloc Autostart Keys Bootloc (CR)

where:

Startloc — is the first memory location to be punched. It must be at or above '30.

Endloc — is the last memory location to be punched (up to '177777).

Autostart — is an autostart address. If specified, the CPU automatically begins execution at this location after reading the self-loading tape being punched. If it is zero or unspecified, the CPU halts after loading tape.

Keys — is a value to be inserted in the status bits associated with the INK and OTK instructions before the program begins execution. Bit assignments are defined in Figure 7-1. Note that bits 14, 15, and 16 have special meaning for MDL.

Bootloc — is the first location to be occupied by the second-level bootstrap loader, when it is read from the MDL tape during program load. If a value is not specified, the default value is 'XXX600, where XXX is the memory area occupied by MDL. (MDL runs in 64R addressing mode.) This parameter is required only on the first block (or tape) of a series, when bit 15 of 'Keys' is 0.

If only a starting and ending address are specified, MDL punches that memory area on the high-speed punch, automatically adds beginning-and end-of-tape records, and punches the second-level bootstrap so that it will load into the memory area occupied by MDL at the time the tape was punched. Other parameters are needed only if:

a. Autostart feature is desired

b. ASR punch is to be used

c. Second-level loader is to be relocated

d. Two or more non-contiguous memory areas are to be punched on a single tape

d. Long program is to be split into two or more separate tapes (second-level loader on first tape only)

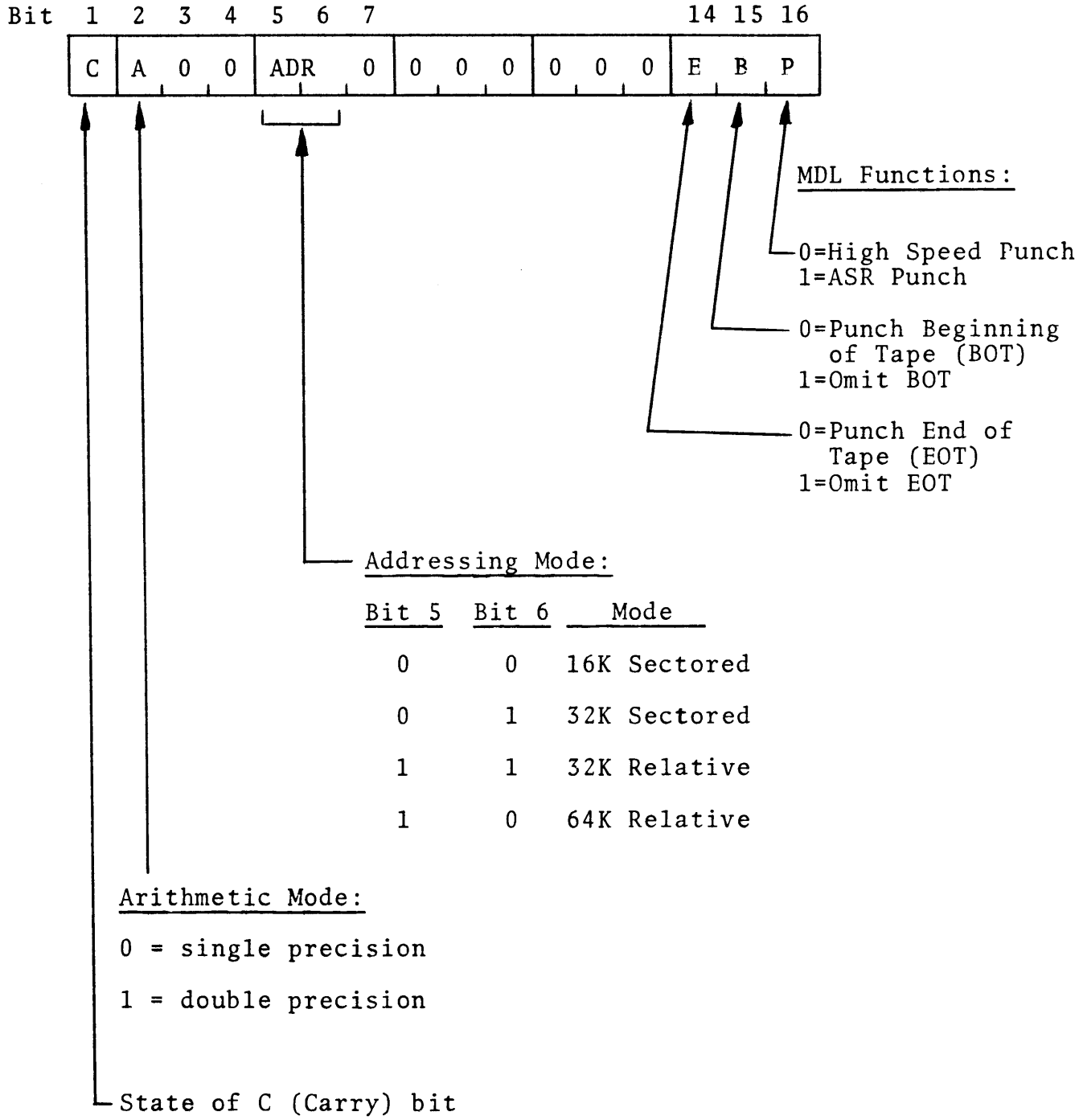


Figure 7-1. Bit Assignments of Keys Parameter.

Selecting Punch Mechanism: Bit 16 of the 'Keys' parameter determines whether the MDL tape will be punched by the ASR or by the high-speed punch. (The default value is 0 for high-speed punch.)

Punching Single Block: Set bits 14 and 15 of the 'Keys' parameter to 0. The resulting tape will be punched with BOT and EOT records. This is the default value.

Punching Multiple Blocks on Single Tape: To punch several non-contiguous blocks of memory on a single tape, use the following patterns in bits 14 and 15 of the 'Keys' parameter.

Keys Bit	14	15
First Block	1	0
Subsequent Blocks	1	1
Last Block	0	1

A 'Bootloc' should be specified when the first block is punched.

Punching Multiple Tapes: To break up a large program into several easily handled tapes, use the following 'Keys' entries:

Keys Bit	05	06	14	15
First Tape	1	0	0	0
Subsequent Tapes	1	0	0	1
Last Tape	Any		0	1

An 'Autostart' parameter and different addressing mode can be specified only for the last tape of the series.

A 'Bootloc' should be specified when the first tape is punched. Tapes prepared with this option can load only through the high-speed reader.

ADDRESS DISPLAY

When the parameter string is entered, the program takes control and begins punching the self-loading tape on the selected device. If the control panel ADDRESS/DATA switch is at DATA, the indicators will display the address of each memory location being punched.

At the end of the memory block, the punch stops and the program requests another set of parameters (or returns to TAP, if present). If this is the last block to be punched, remove the tape from the punch.

PART 2

PAPER TAPE COPY (PTCPY)

PTCPY punches and verifies frame-for-frame duplicates of eight-level paper tapes of any format (source, object, or self-loading). The only tape requirements is that there be at least 12 inches of blank trailer to identify the end of the tape. The high-speed reader/punch is required as the input/output device. PTCPY is controlled by commands entered at the system terminal keyboard. (See Table 7-1 for a summary of commands and messages.)

USING PTCPY UNDER DOS

Enter the PTCPY external command. PTCPY prints the prompt line:

L, P(N), V(N), FP or FV

and awaits a command from the keyboard. (See "Operating Procedures".)

USING PTCPY UNDER DOS/VM

1. Place the tape to be read in the high-speed reader.
2. Unassign, then assign the PTR. A small portion of the tape will be read into the DOS/VM tape buffer.
3. Reposition the tape until blank leader is under the read head.
4. Enter the PTCPY external command and use the L (load) command to read the master tape. Then proceed as under "Operating Procedures".
5. To load, verify, force punch or force verify subsequent tapes, quit from PTCPY (CTRL P) and repeat steps 1 through 3. Then enter S (start) to restart PTCPY and proceed.

USING PTCPY IN PAPER TAPE SYSTEMS

PTCPY is provided as a self-loading paper tape that occupies locations '200 through '3624 and loads into Sector 0. All of memory above PTCPY is available to hold the image of the tape to be duplicated.

Load the PTCPY tape using APL or a key-in loader. Start PTCPY at location '1000; it takes control and prints the cue:

L, P(N), V(N), FP OR FV

and awaits a command from the keyboard. (See "Operating Procedures".)

Table 7-1. PTCPY Command and Message Summary

Code	COMMANDS
L	Load a master tape into memory from the high-speed reader
Pn	Punch n copies of memory image
Vn	Verify n copies against memory image, frame-for-frame
FP	Read and simultaneously force-punch a copy of a tape too large for memory
FV	Verify a force-punched tape by counting the number of frames punched
Q	Quit to operating system
Code	**MESSAGES**
LC	Load complete
MO	Memory overflow - tape too long to fit in available memory
PC	Punch operation complete
VC ON TAPE n	Verification complete on tape n
VE ON TAPE n	Errors found during verification of tape n
n/10 BOX NEEDED	Specifies amount of tape required to complete a P command. (Enter Y to proceed.)

OPERATING PROCEDURES

Loading Master Tape:

1. Mount the tape to be duplicated in the high-speed reader.

2. Enter the L command (load). The tape will begin reading and load into memory. If loading is successful, PTCPY prints LC message. If the tape to be loaded is larger than the available memory, PTCPY prints the MO message. (See "Force Punching".)

Punching Copies:

1. Enter the letter P followed by the number of copies to be punched, as in:

   P5

   PTCPY responds by printing the amount of tape required to make the copy, in 1/10th-in box increments.

2. Check whether there is enough tape in the box to make all the copies; if not, the box should be changed. To proceed, type Y.

3. PTCPY punches the copies, with a one-inch block of fully punched tape following each copy as a separator. When punching is complete, PTCPY prints the PC message and repeats the prompt line, awaiting a new command.

Verifying a Normally Punched Tape:

1. Mount a tape generated by a Punch command in the reader. (It is not necessary to physically separate multiple copies.)

2. Enter the letter V followed by the number of copies on the tape, as in:

   V5

PTCPY checks each copy frame-for-frame against the loaded master in memory and prints the message:

VERIFY ON TAPE n

for each valid copy. If an error is detected, PTCPY prints the message:

ERRORS ON TAPE n

Force-Punching a Tape:

Use this procedure to duplicate a tape that is too long to fit into the amount of memory available:

1. Position the tape to be copied in the high-speed reader.

2. Enter the FP (force-punch) command. PTCPY will simultaneously read and punch a single copy. To verify a tape punched in this way, use the FV (force-verify) command.

Force-Verifying a Tape:

Use this procedure to verify a tape punched by the FP (force-punch) command.

1. Place the force-punched tape in the high-speed reader.

2. Enter the FV (force-verify) command. PTCPY reads and verifies the tape by counting the number of frames and comparing the total with the number of frames punched by the preceding FP command.

Example:

Following is a typical dialog while punching and verifying two copies of a master tape:

```
PTCPY
GO
L,P(N),V(N),FP OR FV
L
LC
L,P(N),V(N),FP OR FV
P2

     1/10 BOX NEEDED
Y
PC
L,P(N),V(N),FP OR FV
V2

VC ON TAPE       1
VC ON TAPE       2
L,P(N),V(N),FP OR FV
```

SECTION 8

SUBROUTINE LIBRARIES

FORTRAN/MATH LIBRARY

Prime's FORTRAN compiler is supported by an extensive library of math, input/output and function subroutines. For FORTRAN users with disk operating system support, use of the library is simplified by the loader's LIbrary command. It automatically attaches to the LIB UFD, and begins loading the library-mode object text in the FTNLIB file. In library-mode, only the subroutines with entry points that have been cited in a previously loaded module are loaded. Since the FORTRAN compiler inserts external library references automatically, the user need not know the names of the subroutines or their purpose.

Assembly language programmers also have access to any of the FORTRAN subroutines which may prove useful. However, the user must use calling sequences and data conventions as defined in the Prime Subroutine Library User Guide, which also summarizes the available routines and specifies the disk files/paper tapes on which they reside. The subroutine entry points in the FTNLIB file can be printed out by the binary editor through the following DOS command sequence:

```
OK;ATTACH LIB
OK;EDB FTNLIB
GO
ENTER,FIND ALL
```

Users with paper tape systems use the library in the same way but must mount the appropriate object paper tapes one at a time. Table 8-1 lists the tapes that correspond to the FTNLIB disk file. The input/output subroutines of IOCS occupy one tape; the math and function routines are supplied on six others of convenient size (FLIB1 through 6). In practice, the user loads the loader by APL, uses it to load the main FORTRAN program, and then mounts the library tapes one at a time and enters LOad commands until a LC (load complete) message is printed. The memory image is then ready to execute and can be saved in the form of an MDL paper tape.

Table 8-1. Library Components

Filename in LIB UFD of Master Disk	Paper Tape Name	Contains
FTNLIB	IOCS FLIB1 FLIB2 FLIB3 FLIB4 FLIB5 FLIB6	Supporting I/O math and function subroutines for FORTRAN Compiler (May be referenced in Assembly Language programs also.)
UII	UII	Virtual Instruction Package of unimplemented instruction subroutines.
MATHLB	MATHL1 MATHL2	Matrix and linear equation operations.

UII LIBRARY

The assembler and FORTRAN compiler both keep track of any requirements for hardware options beyond the standard instruction set, in the following categories:

Double precision integer arithmetic and multiply-divide

Floating point arithmetic

Prime 300 Extended Instructions

Object text always contains the appropriate in-line instructions. If the required hardware is not present in the CPU on which the program is to execute, such instructions cause a UII (Unimplemented Instruction Interrupt). This is intended to divert control to a UII library routine which simulates optional instructions by using instructions which are actually present in the CPU.

To correlate the requirements of the object module with the hardware capability of the CPU, the loader includes a UII handling feature. With the HArdware command, the user can specify the optional instructions present in the CPU on which the program is to execute. If these match the requirements of the program, the UII entry in the load map is 0, and no UII software needs to be loaded. However, if the UII word is other than 0, the user must load the UII library (also called VIP, or Virtual Instruction Package). The object text contains code blocks that specify the routines required, and the UII file contains conditional load provisions so that only the required routines are loaded - others are skipped. An example of the loading procedure appears in Section 5.

MATRIX LIBRARY

A separate library file contains matrix arithmetic and statistical routines that can be called from FORTRAN or assembly language programs. Calling sequences are described in the Prime Software Library User Guide. To load these routines, the user gives the loader command LI MATHLB (or loads the paper tapes MATHL1 and MATHL2).

SECTION 9

PAPER TAPE PROGRAM DEVELOPMENT

For a Prime CPU operating in a stand-alone programming environment (without a mass-storage device and operating system such as DOS), eight-channel punched paper tape is assumed to be the basic medium for storing and loading programs.

During program development, the facilities of the computer itself are used to help programmers and operators. For example, the Text Editor uses computer memory and data handling capacities to create and edit source program text and punch source code tapes. The FORTRAN IV Compiler and Macro Assembler convert the symbolic codes of source program tapes into object code tapes. These in turn are loaded by the Linking Loader, which resolves address references, watches for calls to external subroutines, loads the program, and requests mounting of tapes containing FORTRAN/Math/IO library or other external subroutines. Loaded programs can be checked out and altered at the Teletype keyboard, using Trace And Patch or Prime Symbolic Debug. Operational programs (or any other section of memory) can be punched on tape by Memory Dump and Load, in the self-loading format that can be read by the Automatic Program Load (APL) function or key-in loader. Tapes of any format can be duplicated by the Paper Tape Copy utility. Test and verification routines can be loaded and executed to verify operation of the central processor and its peripheral devices.

SUMMARY OF PAPER TAPE SOFTWARE

Paper Tape Software Packages

Paper tape software for a Prime CPU is supplied in several versions, depending on the peripheral devices available in the system. Appendix A identifies all currently available paper tape software packages. Tapes with the prefix "SLT" are in the MDL self-loading format. Tapes prefixed "OBJ" are in object format and must be entered into memory through the Linking Loader (LDRAP).

Editors (Section 2)

The Text Editor, PTRED, provided as a self-loading system tape, is the tool for new program development. This program permits source programs to be composed, edited, and listed at the user terminal keyboard. After entering a rough copy of the program, the programmer can locate and alter text strings, correct spelling, syntax, or spacing errors, or move lines from one place to another by simple keyboard commands.

Sections of the program can be printed for checking, or the entire program can be listed on the terminal as a reference copy. When the program is complete and ready to be assembled or compiled, a source program tape, in ASCII format, can be punched on paper tape from the low or high-speed reader. (Previously punched tapes can be read in, as well, to be expanded or merged with the current version of the program.

A longer version of the editor, BPTRED, includes the box mode which simplifies generation of graphic or pictorial layouts. The box editing facility is not applicable to program development.

The binary editor, BINED, operates on object modules containing library subroutines. It is useful for examining the contents of library tapes or building custom libraries.

Macro Assembler (Section 3)

Source programs in the Macro assembly language are processed by the Macro Assembler program to form object program tapes. The self-loading assembler is loaded (using the panel APL operation) and the source tape is placed in the high- or low-speed reader. After placing instructions to the assembler in a computer register (through the control panel), the operator starts the computer, which reads the source tape and translates the symbolic addresses used, and the second time to translate the mnemonic expressions into an object program tape. An optional listing shows both the source symbolic code and the translated binary equivalent of each entry.

FORTRAN IV Compiler (Section 4)

Source programs in the FORTRAN IV language are processed in the same way as assembly language programs. After the FORTRAN Compiler is loaded, it controls a one-pass reading of the source program tape. The output object tape is similar in format to the assembly language output tape. An optional listing, either a straight listing of the source statements or an expanded listing showing the machine language breakdown of each statement, may also be printed.

Linking Loader (Section 5)

Object tapes punched by the assembler or compiler are not in the MDL format required for APL or key-in loading. They are relocatable, and require the Linking Loader to interpret and complete the addressing information. Indirect address links must be formed in sector zero (or another specified base sector) when address references happen to fall across sector boundaries.

To load a program tape in object format, the self-loading Linking Loader is first loaded. The operator then mounts the object tape on the reader and supplies the loader with a start-of-load address (and other parameters by keyboard inputs from the user terminal). The loader then reads the object tape, resolves addresses, and loads the program starting at the specifiec base location.

When loading of main programs and subroutines is complete, the user can request the loader to print a memory map. The memory map defines the memory areas occupied by the program and lists all subroutine calls and external references.

Once a program has been loaded by the Linking Loader, it is fully translated into 16-bit machine language codes and is ready to execute or be converted to a self-loading tape by MDL.

Debug Aids (TAP or PSD) (Section 6)

During the early stages of program development and checkout, TAP and PSD permits the programmer to examine, alter, and list the content of memory locations in response to simple terminal keyboard commands. A "trace" function controls dynamic execution of object programs, with diagnostic printout of register contents at selected intervals (for example, whenever a specified effective address is formed).

Trace, Punch and Copy Utilities (Section 7)

Memory Dump and Load (MDL): Loading of an object program and accompanying external library or subroutine programs is often a time consuming operation. MDL saves the result of a program building session of punching the entire loaded program on paper tape in the self-loading format. The program can be restored to the same memory area from which it was punched by using APL or the key-in loader. MDL uses the low- or high-speed punch.

Paper Tape Copy (PTCPY): This utility program uses the high-speed reader-punch to duplicate and verify paper tapes punched in any format (ASCII, object, or self-loading).

Library Subroutines (Section 8)

Another function of the Loader is to recognize calls to subroutines of the FORTRAN/Math library or other unloaded subroutines. After completing the loading of a main program, the loader halts and requests any referenced subroutines. The operator then mounts the library tapes in the input device, and starts the computer again until a "load completed" message is printed.

GENERATING SELF-LOADING TAPES

Self-loading tapes are those that can be read automatically by the Automatic Program Load option (panel LOAD) or the key-in loader. They are punched from memory by MDL, and load only into the same locations from which they were punched. Editor output tapes in ASCII format, and the object output of the Macro Assembler and FORTRAN IV Compiler, are not suitable for self-loading.

The Macro Assembler and FORTRAN Compiler always load into lower memory (Sector 0 and above) since they are the only programs in the computer while they are being used. Self-loading versions of these programs are supplied for use with any computer configuration.

A self-loading version of the Linking Loader is supplied to aid the user in setting up his own system software. The first task of the loader is usually to load the object versions of both MDL and the Linking Loader, in order to generate self-loading tapes of both these programs for use in higher memory. To accomplish this, the 8K self-loading version is loaded into lower memory. (See Figure 9-1A.) It in turn is used to load the object version of MDL or TAPMDL in the top sector of memory (or elsewhere, to suit user requirements).

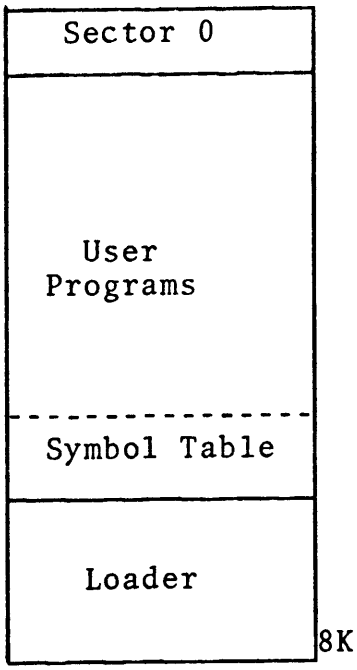
The linking Loader is then usually loaded into higher memory, just below MDL (Figure 9-1B). This location of the loader leaves all of lower memory available for program loading, and the area occupied by the loader itself can be assigned as a common area in response to COMN (Assembler) or COMMON (FORTRAN) statements. The loader symbol table can grow downward from the loader without wiping out MDL, and the loader can detect memory overflow.

MDL can then be used to punch self-loading tapes of both the loader and itself (the loader and MDL may be combined on a single tape). The user then has a package of self-loading tapes to form systems configured as shown in Figure 9-1A and B.

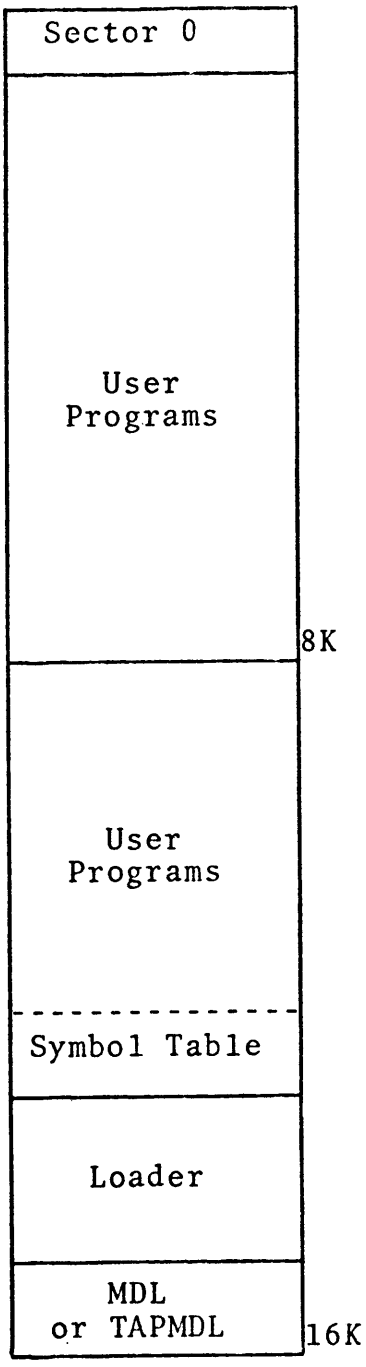
Some application programs may be so long that the space occupied by a high version of MDL cannot be spared. To handle this situation, another self-loading version of the loader can be loaded into the highest sectors of memory. The self-loading version of MDL in lower memory can be used to punch a self-loading tape of this loader.

The supplied self-loading version of TAP loads at '1000, but TAP can replicate itself anywhere in memory. It is useful to generate self-loading versions of TAP for other areas of memory, such as the area initially occupied by the loader. Incidentally, TAP should be loaded and used to clear memory before any other programs are loaded. The MDL output tapes are then as compact as possible, since blocks of identical memory locations are condensed into a few tape frames.

In situations where very large programs must be loaded, the user may need to load versions of LOADAP and MPL to run in lower memory as well.



A. SLT Version of Loader
at Top of 8K Memory



B. Typical Location of Loader
and MDL in 16K Memory

Figure 9-1. Memory Areas for Utility Programs.

Figure 9-2 shows how the two different sets of LOADAP and MDL can be used to generate two self-loading tapes that together contain the image of all the available memory. The self-loading tape of LOADAP that occupies the top of an 8K memory is used to load self-loading program components into lower memory. A load map is obtained; then a self-loading version of MDL is loaded into upper memory and used to punch a self-loading tape of the loaded components. Another version of LOADAP (generated by the user from the supplied object version) is then loaded into lower memory and used to load components in upper memory. Location PBRK from the first load map is the next available location to be loaded. When loading is complete, a lower-memory version of MDL is loaded and used to punch a self-loading tape of upper memory.

If the user programs contain address references between the upper and lower sections of memory, the user may need to prepare and load a program consisting of assembler ENT statements to resolve such linkages.

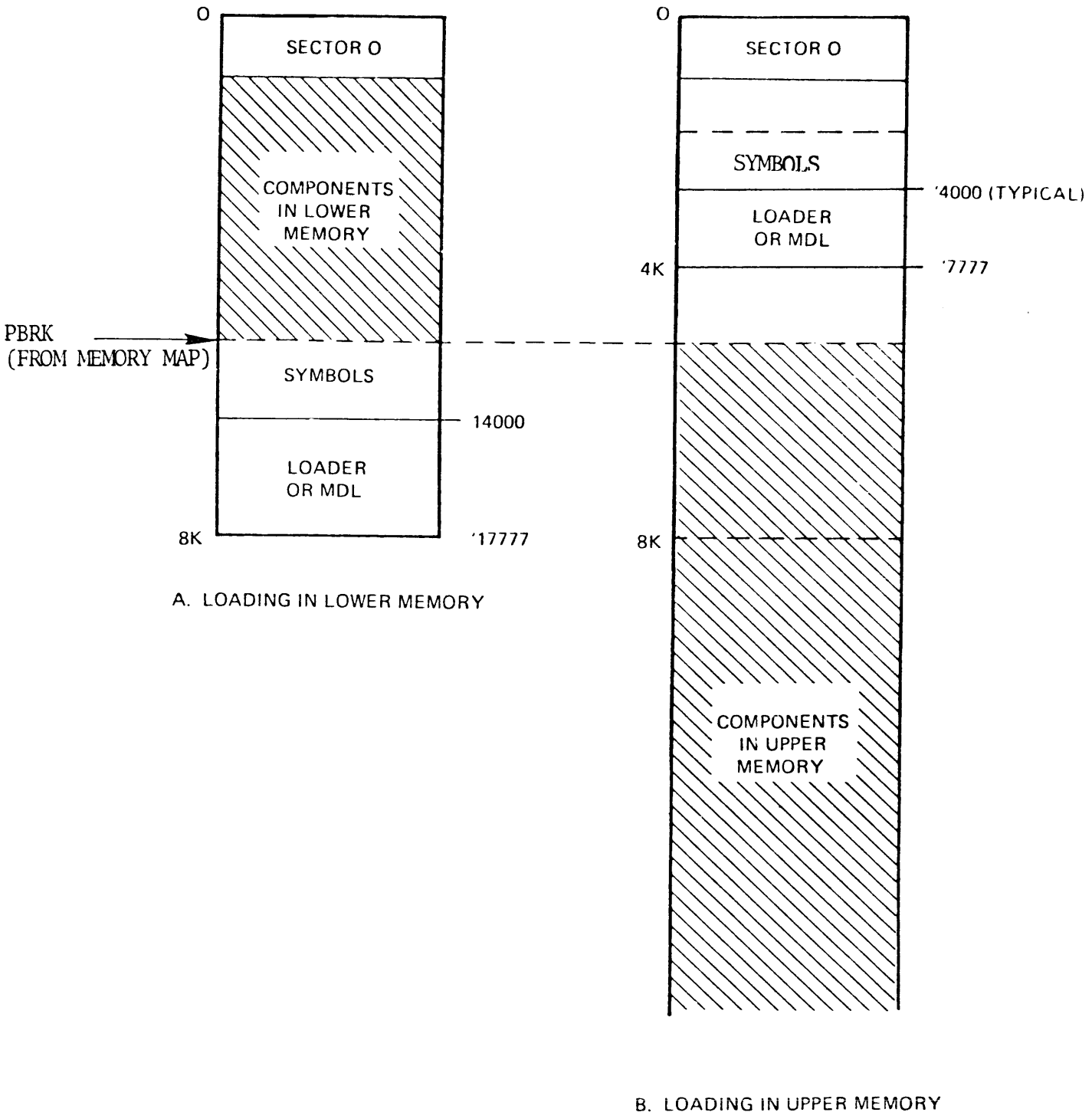


Figure 9-2. Using Loader to Build Systems in Both Lower and Upper Memory.

APPENDIX A

PAPER TAPE SOFTWARE PACKAGES

Prime paper tape software is distributed in several packages according to the hardware characteristics of the user's system:

Package	Contents	Supplied to
A	Standard program development and test software for CPU and System Option Controller.	Users without MHD and Master Disk (4000, 412X).
B	Loaders and test programs necessary to install a DOS system.	Users with MHD and Master Disk but no disk APL option.
C	Basic software for building DOS on FHD. (Packages A and B also required).	Users with FHD (410X).
D	Tapes and listings for hardware options.	As required.
U	Software updates.	As required.

Contents of packages A through D are summarized in the following tables. Package U contains updates or additions that are issued between master disk revisions. It is initially empty.

Table A-1. Package A (Rev. 6)

ITEM NO.	NAME	PART NO.	HIGH	LOW	START	TAPE	LISTING
1	LOADAP	SLT1086.002	17770	14000	*15000	X	
2	LOADAP	OBJ1086.002	N/A	N/A	N/A	X	
3	PMA	SLT1080.413	15463	100	400	X	
4	FTN	SLT1082.005	22347	100	1000	X	
5	IOCS	OBJ9100.005	N/A	N/A	N/A	X	
6	MATHL1	OBJ9305.005	N/A	N/A	N/A	X	
7	MATHL2	OBJ9605.005	N/A	N/A	N/A	X	
8	FLIB1	OBJ9001.005	N/A	N/A	N/A	X	
9	FLIB2	OBJ9002.005	N/A	N/A	N/A	X	
10	FLIB3	OBJ9003.005	N/A	N/A	N/A	X	
11	FLIB4	OBJ9004.005	N/A	N/A	N/A	X	
12	FLIB5	OBJ9005.005	N/A	N/A	N/A	X	
13	FLIB6	OBJ9006.005	N/A	N/A	N/A	X	
14	JII	OBJ1025.005	N/A	N/A	N/A	X	
15	ED	SLT0713.002	21423	66	*1002	X	
16	EDB	SLT0745.002	10720	200	*1000	X	
17	TAP	SLT0718.002	17777	17000	*17000	X	
18	PSD	SLT0790.000	14000	17775	*14000	X	
19	TAPMDL	SLT9200.204	17777	15000	*15000	X	
20	MDL	OBJ0714.005	N/A	N/A	N/A	X	
21	PTCPY	SLT0709.303	200	3524	1000	X	
22	CPJT1	SLT0717.003	5502	60	1000	X	X
23	CPJT2	SLT0712.004	5554	60	1000	X	X
24	CPJT3	SLT0728.002	4062	200	1000	X	X
25	HSMT1	SLT0723.003	1775	60	1000	X	X

* AUTOSTART

Table A-2. Package B (Rev. 6)

ITEM NO.	NAME	PART NO.	HIGH	LOW	START	TAPE	LISTING
+							
1	DOSBOOT	SLTO748.402	1576	1000	*1000	X	X
+							
2	CPJT1	SLTO717.003	5502	50	1000	X	X
3	CPJT2	SLTO712.004	5654	50	1000	X	X
4	CPJT3	SLTO728.002	4062	200	1000	X	X
5	HSMT1	SLTO723.008	1776	50	1000	X	X
+							

* AUTOSTART

Table A-3. Package C (Rev. 6)

ITEM NO.	NAME	PART NO.	HIGH	LOW	START	TAPE	LISTING
1	MAKE	SLT0720.403	11606	66	1000	X	
2	*DOS16	SLT0608.406	17776	7000	31000	X	
3	*DOS24	SLT0608.406	17776	7000	51000	X	
4	*DOS32	SLT0608.406	17776	7000	71000	X	
5	DOS16	SLT0608.406	37776	27000	*31000	X	
6	FILMEM	SLT0704.003	1136	1000	1000	X	
7	FTN	SLT1082.408	21166	100	1000	X	
8	LOAD	SLT1084.404	23752	20000	21000	X	
9	PMA	SLT1080.415	16322	100	400	X	
10	FILCPY	SLT0700.402	4010	66	1000	X	
11	UFDCPY	SLT0708.400	4140	66	1000	X	
12	DOSEXT	SLT0705.403	2360	600	1000	X	
13	FIXRAT	SLT0707.403	11471	66	1000	X	
14	MACHK	SLT0743.001	1007	1000	1000	X	
15	PRSER	SLT0701.800	4530	66	1000	X	
16	PRMPC	SLT0789.800	4341	66	1000	X	
17	CRSER	SLT0722.702	7725	66	1000	X	
18	CRMPC	SLT0788.700	7722	66	1000	X	
19	SORT	SLT0759.400	14520	66	1000	X	
20	FILVER	SLT0758.000	2574	66	1000	X	

* AUTOSTART

Table A-4. Package D (Rev. 6)

ITEM NO.	TYPE NO.	NAME	PART NO.	HIGH	LOW	START	TAPE	LISTING
+								
1	140) 240) 340)	DST1	SLT0711.002	1652	600	1000	X	X
2	322)	FLTP0	SLT0779.000	2531	60	1000	X	X
3	330) 332) 333)	NCSP	SLT0778.001	17577	60	1000	X	X
4	332)	DFLT	SLT0756.001	13000	60	1000	X	X
5	332) 333) 360)	FLTPT1	SLT0764.001	13015	60	1000	X	X
6	3XX	PAST1	SLT0776.000	2447	60	1000	X	X
7	3XX	P3000T1	SLT0721.000	7000	60	1000	X	X
8	3001) 3003)	TTYT1	SLT0725.007	5475	200	1000	X	X
9	3001) 3003)	RTCT1	SLT0729.004	1654	1000	1000	X	X
10	3003	HSRPT1	SLT0724.008	2512	200	1000	X	X
11	3005	HSRPT2	SLT0735.003	5303	60	1000	X	X
12	3005	RTCT2	SLT0734.003	5051	60	1000	X	X
13	3006	TTYT2	SLT0733.004	11313	60	1000	X	X
14	3007	BPIOT1	SLT0735.002	3732	60	1000	X	X
15	3127	LPTST1	SLT0775.000	2023	200	1000	X	X
16	3141) 3161) 3181) 3191) 3195)	DPCARD	SLT0730.002	2557	66	1000	X	X
17	4000	DRATIT	SLT0727.004	14777	200	1000	X	X
18	4001) 4002)	DISCT1	SLT0737.001	16000	66	1000	X	X
19	4020	MTJT1	SLT0715.002	2442	66	1000	X	X
		MAG7	OBJ9201.003	N/A	N/A	N/A	X	
		MAG9	OBJ9202.003	N/A	N/A	N/A	X	
20	4030	DSCTT1	SLT0772.000	11777	66	1000	X	X
21	5002) 5004)	DSCTST	SLT0703.000	1760	1000	1000	X	X
22	5002) 5004) 5052) 5054)	AMLCT1	SLT0751.000	3777	200	3000	X	X
23	5201	MSLCT1	SLT0752.002	16777	140	1000	X	X
24	5402	MACIT1	SLT0780.001	3120	200	1000	X	X
25	6000	A/DST1	SLT0734.004	5727	60	1000	X	X
26	6020	DIGINP	SLT0731.000	4351	100	1000	X	X
27	7000	GPIBT1	SLT0750.001	1674	777	1000	X	X
28	7030	IPCT1	SLT0733.000	5031	200	1000	X	X
+								

* AUTOSTART

INDEX