MAN1879

PROGRAM DEVELOPMENT SOFTWARE
User Guide

Revision A
February, 1977

PR1ME
Computer, Inc.

145 Pennsylvania Ave.
Framingham, Mass. 01701

CONTENTS

ILLUSTRATIONS

TABLES

FOREWORD

This user guide describes the Rev. 11 Prime Program Development Software used to generate, compile or assemble, load and debug FORTRAN, COBOL or assembly language programs. It consists of the following sections:

Section 1	Introduction
Section 2	Editor
Section 3	Binary Editor
Section 4	Prime Macro Assembler (PMA)
Section 5	FORTRAN Compiler
Section 6	COBOL Compiler
Section 7	Linking Loader
Section 8	Debug Aids (TAP and PSD)
Section 9	Segmentation Utility for Prime 400 and 500

Information in this guide applies to systems with operating system and file system support.

SECTION 1

INTRODUCTION

SCOPE

This user guide contains detailed reference information on using the editors, translators and utilities that are the essential 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:

Title	Document No.
FORTRAN User Guide	MAN 1674
COBOL Reference Guide	MAN 2797
PRIME MACRO ASSEMBLER User Guide	MAN 1673
PRIMOS Interactive User Guide	MAN 26Ø2

SUMMARY OF PROGRAM DEVELOPMENT SOFTWARE

The following paragraphs summarize the main functions of the program development software as used in a PRIMOS environment with file system support.

Editor (Section 2)

The Editor is the means by which a programmer creates a program or edits a program file for the purpose of making changes.

Binary Editor (EDB) (Section 3)

The Binary Editor examines loader-compatible object text blocks generated by Prime's FORTRAN compiler and Macro Assembler. It is useful for creating library subroutine files.

Macro Assembler (PMA) (Section 4)

Source programs in the Macro assembly language are processed by the Macro Assembler program to form object program files. The assembler reads the source file and translates the symbolic codes of the source program into the object code 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 5)

Source programs in the FORTRAN IV language are processed in the same way as assembly language programs. The FORTRAN Compiler 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.

COBOL Compiler (Section 6)

Source programs in the COBOL language are processed by the COBOL compiler, which creates the COBOL object file and a listing file.

The object file is loaded into memory and executed in the same manner as FORTRAN.

Linking Loader (Section 7)

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 base 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 PRIMOS SAVE file format.

Debug Aids (TAP and PSD) (Section 8)

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).

SEGMENTATION UTILITY (SEG) (Section 9)

SEG is a PRIMOS IV or IV utility module for loading and running segmented programs and making modifications to segmented run files.

FILE USAGE IN PROGRAM DEVELOPMENT

File types encountered during program development under PRIMOS are illustrated in Figure 1-1. A typical collection of program development files within a user's UFD is shown in Figure 1-2.

Source Files

Source files are ASCII data files created by the text editor or entered into the system from unit record devices or magnetic tape. They are the text of the program in the appropriate source language - PMA, FORTRAN, or COBOL, for example.

Object Files

Object files are the result of a translation process by one of the PRIMOS language translators (PMA, FTN, COBOL). They are in a compatible binary format suitable for processing by LOAD (to run in any addressing mode except 64V) or SEG (to run in 64V mode). All library files (such as those contained in FTNLIB and UII) are in object format. Object files can be examined and altered by the binary editor, EDB. Prime's translators all use the convention of naming object files by prefixing the first four characters of the source filename by B←. (However, the user can assign other names by using the PRIMOS BINARY command before starting translation.)

Listing Files

Listing files are optionally produced by the translators. They contain a listing of the source program with error lines flagged and with other optional details such as the assembly-language translation of the source statements. Prime's translators follow the convention of naming listing files by prefixing the first four characters of the source filename by L←. (However, the user can assign other names by using the PRIMOS LISTING command before starting translation.)

Map Files

The loader and segmentation utility both have the option of generating load maps and writing them to files specified by the user. Maps can also be printed at the user's terminal, but in the case of large maps, this is time consuming. Large maps can be written to a file, then spooled for printing on the system's high-speed line printer.

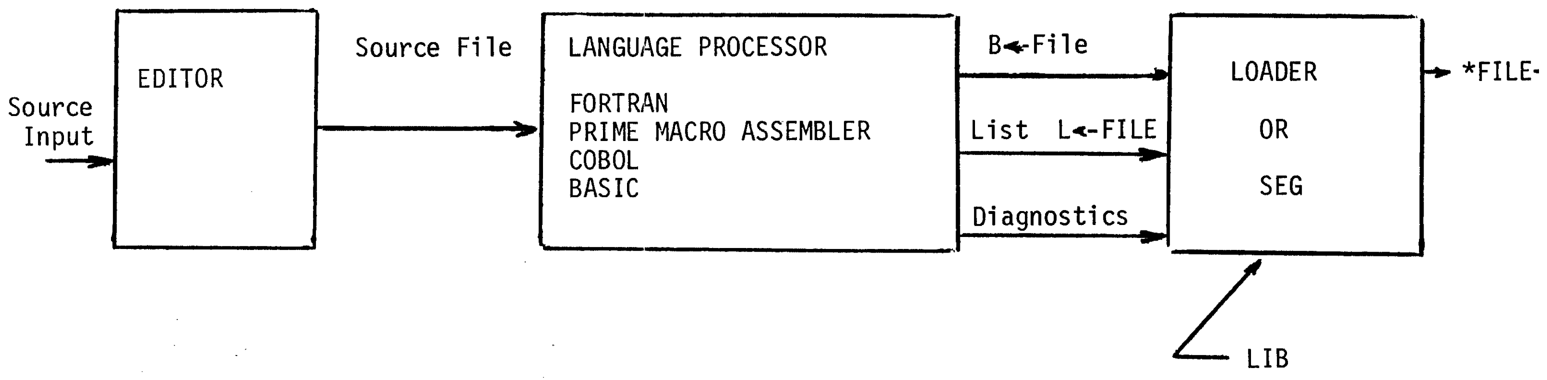


Figure 1-1. Typical Program Development Files

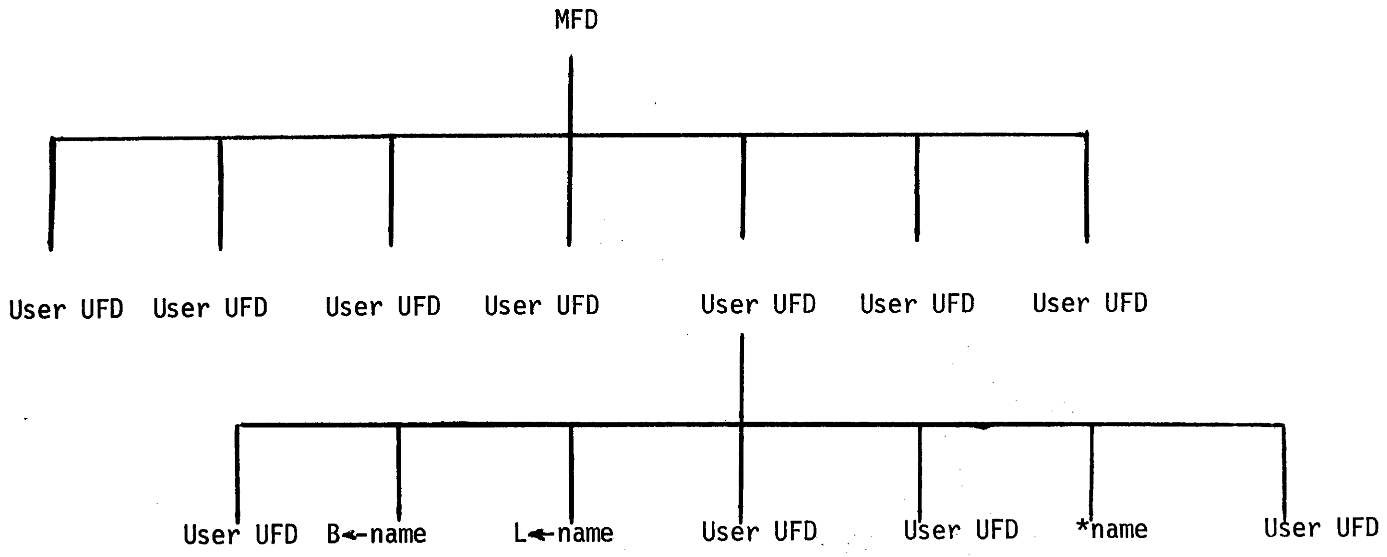


Figure 1-2. Typical Program Development Operation

Run Files

Run files are the end result of the translation and loading process. Run files created by LOAD are in memory-image format with a header block that specifies the starting address, register contents, keys parameters, and other run-time parameters. Run files created by SEG are in segmented run-file format.

Prime's convention is to name run files by prefixing the first four letters of the source filename with an asterisk (*).

Other Files

Files used as input during program execution, or files generated as a result of execution, may be in any format compatible with PRIMOS file systems. These files are entirely under control of the user.

SYMBOLS AND ABBREVIATIONS

Symbols and abbreviations used throughout the text 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)	Contents of Effective Address

COMMAND FORMATS

All software described in this guide communicates with the user through a series of commands entered at the user terminal. This guide uses the following conventions to define command syntax:

```
    COMMAND Filename Param1  [Param2]  | LITERAL |
                                       | Param3  |
```

The command name is shown in capital letters. The underlined letters are mandatory. The remaining letters are optional.

Following the command name are parameters and/or literals, separated from the command name and each other by at least one space. Generally, parameters have the first letter capitalized. (These are occasional exceptions such as the use of n for an integer value.) Optional parameters are enclosed in brackets and may be omitted. For each parameter, the user must substitute an alpha or numeric value required by the specific command definition.

Literals are shown in capital letters with the permissible abbreviation underlined.

Parameters or literals stacked between vertical bars are alternatives, of which one must be chosen.

When Filename appears as a parameter, the user must specify a filename existing (or to be created) within the current UFD.

SECTION 2

EDITOR

INTRODUCTION

This section defines the use of Prime's Editor for developing software. It briefly describes the Editor modes and commands for reference purposes.

The Editor provides two modes of operation: the INPUT mode and the EDIT mode.

INPUT MODE

The INPUT mode is used when typing information into a file (e.g., when creating a program). The word 'INPUT' is displayed at the user's terminal to indicate that the Editor is in the INPUT mode.

The INPUT mode also provides a line terminator capability, a character or line erase capability, and tabulation.

The RETURN key terminates the current line and prepares the Editor to receive a new line.

Corrections can be made on the current line while in INPUT mode. When it is desired to erase one or more characters, the erase symbol is typed (default is "). For each " typed, a previous character is erased.

If the entire current line is to be deleted, the line erase symbol is typed (default is ?). Note that these symbols may be changed using a symbol assign command (SYMBOL) while in the EDIT mode.

Tabulation is achieved with a backslash (\) character. Each backslash represents the first, second, third, etc. tab setting. Default values are at columns 6, 15, and 30. Up to eight tab settings can be made.

EDIT MODE

The EDIT mode is used when the contents of one or more lines are to be changed. More than 45 commands are available and described in the EDITOR COMMAND SUMMARY.

In Edit mode, the Editor keeps track of the current line by maintaining an internal line pointer. Commands such as TOP, BOTTOM, FIND and LOCATE, move the pointer. Use WHERE to find out what line number is current. Use POINT to move to another line number. Specify MODE NUMBER to display the line number.

SOFTWARE DEVELOPMENT TIPS

The following tips should aid the programmer in quickly adapting to Prime's software development techniques. A summary of the Editor commands appears at the end of this section.

1. TO MOVE LINES OF CODE TO ANOTHER LOCATION

A programmer can move any number of lines from one location in a program to another. The DUNLOAD command deletes the lines as it unloads and creates a file. A LOAD command loads the new file at the desired point.

A programmer may copy any number of lines of code with the UNLOAD command rather than DUNLOAD and use the LOAD command to load the copy at the desired point.

2. TO SAVE TIME USING TAB SETTINGS

When writing source code, much time can be saved by making use of the TABSET command. When entering the source program, each backslash character represents one tab setting. If the TABSET command was not used to define the tab settings, the default values of columns 6, 15 and 3Ø are used.

3. TO OVERLAY COMMENTS AFTER CODE IS WRITTEN

A useful technique of adding comments to an existing source program is with the OVERLAY command using tabs.

4. TO FIND A LINE BY STATEMENT NUMBER

Use the FIND command to locate a statement number in a FORTRAN program or a symbol in a PMA program.

5. TO MODIFY A LINE WITHOUT CHANGING CHARACTER POSITIONS

The MODIFY command is used when a line modification is required, but the relative column alignment must remain the same.

EDITOR COMMAND SUMMARY

The following is an alphabetic list of each Editor command and its function. Acceptable command abbreviations are underlined.

Command	Function
APPEND String	Appends String to the end of the current line.
BOTTOM	Moves the pointer beyond the last line of the file.
BRIEF	Speeds editing by minimizing responses to Editor commands.
CHANGE/String1/String2/	Changes the text of String1 to String2.
DELETE [n]	Deletes n lines, including the current line.
DELETE TO String	Deletes all lines up to but not including line containing String.
DUNLOAD Filename n	Deletes n lines from current file and writes them into Filename.
ERASE Character	Changes current erase character to Character.
FILE [Filename]	Writes the contents of the current file into Filename.
FIND String	Moves the pointer to the first line beginning with String.
GMODIFY	Allows user to enter a string of subcommands which modify characters within a line.
INPUT |(ASR)| |(PTR)| |(TTY)|	Reads text from the specified input device: ASR (Teletype paper tape reader), PTR (high-speed paper tape reader) or TTY (terminal).
INSERT String	Inserts String after current line without switching to INPUT mode.

KILL Character	Changes current kill Character to Character.
LOAD Filename	Loads Filename into text following the current line.
LOCATE String	Moves pointer forward to the first line containing String.
MODE COLUMN	Displays column numbers.
MODE NCOLUMN	Turns off the column display.
MODE NUMBER	Displays line numbers with a PRINT or VERIFY command.
MODE NNUMBER	Turns off line number display.
MODE PRALL	Prints lower case characters if device has that capability or precedes lower case characters with an ^L and precedes upper case characters with an ^U if the device is upper case only.
MODE PRUPPER	Prints all characters as upper case.
MODE PROMPT	Prints prompt character.
MODE NPROMPT	Stops printing prompt character.
MODIFY/String1/String2/	Superimposes String2 on top of String1.
MOVE Buffer1 Buffer2	Moves one line of text from Buffer2 into Buffer1. Buffer names are STRA, STRB, STRC, INLIN and EDLIN.
NEXT n	Moves the pointer n lines forward.
OVERLAY String	Superimposes String on current line. Use tabs to start in middle of line. Use ! to delete existing characters.
PAUSE	Returns to operating system without changing the Editor state.
POINT Line	Relocates the pointer to Line.

PRINT [n]	Prints the current line or n lines beginning with the current line.
PSYMBOL	Prints a list of current symbol characters and their function.
PTABSET Tab1...Tab8	Provides for a setup of tabs on devices that have physical tab stops.
PUNCH | (ASR) | [n] | (PTP) |	Punches n lines on high- or low-speed paper-tape punch.
QUIT	Returns control to PRIMOS without filing text.
RETYPE String	The current line is replaced by String.
SYMBOL Name Character	Changes a symbol Name to Character. Current default values are:

Name	Default Characters
KILL	?
ERASE	"
WILD	!
BLANK	#
TAB	\
ESCAPE	^
SYMBOL	;
CPROMPT	$
DPROMPT	&

TABSET Tab1...Tab8	Sets up to eight logical tabstops to be invoked by the tab symbol (\).
TOP	Moves the pointer one line before the first line of text.
UNLOAD Filename n	Copies n lines into Filename.
VERIFY	Displays each line after completion of a command.
WHERE	Prints the current line number.

XEQ Buffer	Executes the contents of Buffer. See MOVE.
*[n]	Repeat symbol. Causes preceding command to be repeated n times as in: `F /;D;N;*10` which deletes the next ten lines beginning with / . If n is omitted, the command repeats until the bottom of file is reached.

SECTION 3

BINARY EDITOR (EDB)

INTRODUCTION

EDB is a binary editor for operation on loader-compatible object text blocks generated by the Prime language translators. 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.

USING EDB

Loading and Starting Under Primos

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

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

If either the input or output file is on paper tape, the appropriate entry is (PTR). An output file

need not be specified. When Outputfile is specified, a file of that name is created in the current UFD.

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

Positioning 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 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.

Printing 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 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.

An end-of-tape mark is written by the GENET command as well as the ET command. On paper tape, the end-of-tape mark consists of two succesive characters, both '223. On disk, the end-of-tape mark 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.

Error Messages

EDB prints 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 COMMAND SUMMARY

EDB responds to the following 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 terminal. The pointer moves past the last copied item.

ET

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

```
FIND | Name |
     | ALL  |
```

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 terminal. In the VERIFY mode, the FIND ALL command can be used to print all subroutines and entry names in the input file.

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 terminal. 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 [Filename]

Closes the current binary input file and opens a new input file, Filename, for reading only. The binary location pointer is placed at the top of the 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 terminal. The pointer moves to the next subroutine.

OPEN [Filename]

Opens an output file, Filename, for writing only.

QUIT

Closes all files and exits to PRIMOS. 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 terminal.

(see BRIEF, VERIFY).

TOP

Moves the binary location pointer to the top of the input file which is useful only when the input file is on disk.

VERIFY

Places EDB into VERIFY mode. All subroutine names and entry points, as they are encountered by EDB, are printed on the terminal. EDB 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: ROUT1, TEST1, TEST2, ROUT2, MORE, and AGAIN.

The following EDB commands create another version of the library under the name LIBEV2, having the following contents: ROUT1, ROUT2, MORE, and 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 the UFD, USER1, the user invokes EDB, with LIBE specified as the input file and LIBEV2 as the output file. A BRIEF command simplifies the terminal output. The first COPY command copies subroutines ROUT1 and TEST1, and the pointer stops at the beginning of TEST2. The FIND command skips all of TEST2 by moving the pointer 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 PRIMOS.

Distributing Routines to Different Files

Assume the user has a collection of subroutines in a library file named FILIN containing FILE1, FILE2, and 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
ENTER, 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 PRIMOS command to start EDB, subsequent output filenames are set up by OPEN commands (it is not necessary to return to PRIMOS). Each OPEN command closes the previous output file. The user must be careful to issue an ET command 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 object 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 PRIMOS command string that starts EDB. Thereafter, EDB INSERT commands specify new input files to be appended. End-of-tape marks at the end of the input files are not copied. The user issues an ET command 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.

Adding a Subroutine to a Library

Any subroutine can be added to any library. The following example illustrates this technique:

```
     OK, A LIB
     OK, EDB FTNLIB TEMP
     GO
     ENTER I SUB
     ENTER COPY ALL
enter, QUIT
     OK, CNAME FTNLIB ELSE
     OK, CNAME TEMP FTNLIB
     OK, A TED
     OK, LOAD
     $ LO B<-PGM
     $ LIB
     LC
     $
```

SECTION 4

PRIME MACRO ASSEMBLER (PMA)

INTRODUCTION

This section describes the necessary procedure for assembling source programs for PMA up to and including Rev. 11.

SOURCE PROGRAMS

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

OPERATION UNDER PRIMOS

Loading and Starting Assembler

The Macro Assembler is loaded and started by the PMA external commanc to PRIMOS:

```
        PMA Filename [1/A-register]
```

where Filename is a Prime Macro Assembly Language source program in the current UFD, A-register is an A-Register setting that specifies listing detail, I/O devices, and other assembly control parameters. (See Figure 4-1.)

If A-register is not specified by the command string, the assembler uses the default values set up in the RVEC vector at the time the assembler was SAVEd. This value is usually:

A-register	'000777	Normal listing detail, all input and output files on disk

If in doubt, issue the following command sequence:

```
    OK, A CMDNC0
    OK, REST PMA
    OK, PM
```

PM will print the correct RVEC vector and the value from A-register may be determined from this information. Bit assignments are described in detail at the end of this section.

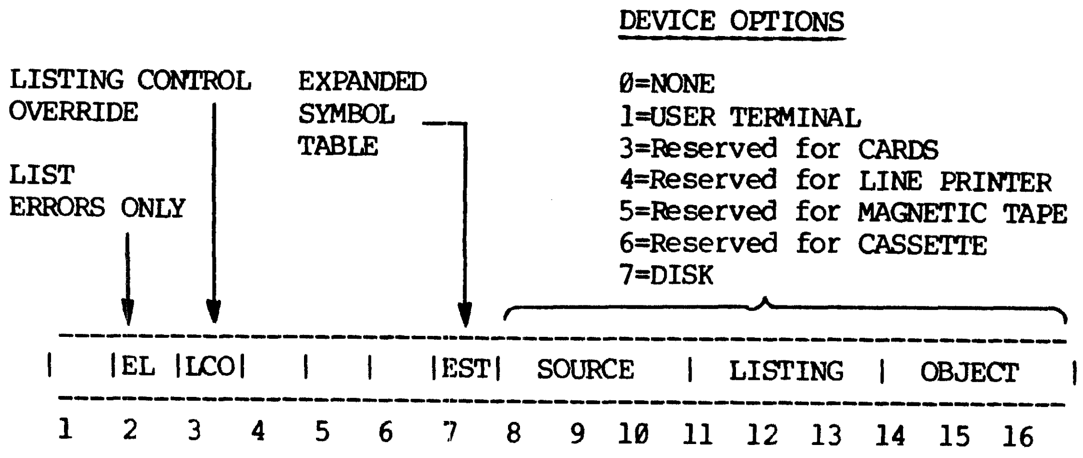


Figure 4-1. Assembler A-Register Settings

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.

File Usage

Three files may be involved during a assembly:

File Type	PRIMOS File Unit
Source	1
Listing	2
Object	3

PMA will automatically open files for the listing and object output, provided disk is specified as the destination for those files. The names are formed by prefixing the first four letters of the source filename with B<- for the object output file (binary file) and with L<- for the listing output. If the user prefers other names, he can used the PRIMOS BINARY and LISTING commands to open files on units 2 and 3 before invoking PMA.

Opening File Unit 2 before the PMA command allows the listing output of more than one source file to be concatenated, the file to be written in other than the current UFD, and the filename to be other than L<-XXXX. File Unit 2 is opened by using the PRIMOS LISTING command.

All file units opened by PMA are closed before PMA returns control to PRIMOS. However, files opened by the user are not closed or truncated by PMA.

ASSEMBLER MESSAGES

When the assembler reads the END statement of the input file on the second pass, it prints a message, terminates assembly, and returns control to PRIMOS command level. The message contains a decimal error count and version of the assembler, as in:

```
        0001 ERRORS (PMA-1080.019)
```

LISTING FORMAT

Figure 4-2 shows a section of a typical assembly listing and illustrates the main features.

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 source statement contains an apostrophe ('), columns 10-72 of that statement become the header for all pages that follow, until a new title is specified.

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.

```
            UCOMP, MSORTS, MLG, 4 JULY 1974
                              (0001) *     UCOMP, MSORTS, MLG, 4 JULY 1974
                              (0002) *     UNSIGNED INTEGER COMPARISON
                              (0003) *     PRIME COMPUTER INC., SRCQQQQ.000
                              (0004) *     COPYRIGHT 1974, PRIME COMPUTER INC., FRAMINGHAM, MASS.
                              (0005) *

                              (0020) *     FROM PMA PROGRAM
                              (0021) *
                              (0022) *     CALL    UCOMP
                              (0023) *     DAC     FWORD1         ADDRESS OF FIRST WORD
                              (0024) *     DAC     FWORD2         ADDRESS OF SECOND WORD
                              (0025) *     OCT     0
                              (0026) *
                              (0027) *
                              (0028) *
                              (0029) *
                              (0030) *
                              (0031) *     FROM FORTRAN PROGRAM
                              (0032) *
                              (0033) *     CALL UCOMP(FWORD1,FWORD2)
                              (0034) *        WHERE THE PARAMETERS ARE DEFINED AS ABOVE.
                              (0035) *
                              (0036) *
                              (0037) *
                              (0038) *
                              (0039) *
                              (0040) *     UCOMP IS USED AS A FORTRAN INTEGER FUNCTION.  IT SHOULD BE USED
                              (0041) *     IN ARITHMETIC "IF" STATEMENTS AS:
                              (0042) *
                              (0043) *             IF(UCOMP(ARG1,ARG2)) -,0,+
                              (0044) *
                              (0045) *     IN PLACE OF:
                              (0046) *
                              (0047) *             IF(ARG1-ARG2) -,0,+
                              (0048) *

                  000000      (0056)       SUBR    UCOMP          UNSIGNED INTEGER COMPARISON
                              (0057) *
                              (0058) *
                              (0059)       C64R
                              (0060)       REL
                              (0061) *
                              (0062) *
000000:      00.000000A (0063) UCOMP  DAC     **             ENTER
000001:      35.000000  (0064)        LDX     UCOMP          POINT TO PARAMETER LIST
000002:      22.000000A (0065)        LDA     0,1            FIRST WORD ADDRESS
000003:      42.000001A (0066)        LDA*    1              FIRST WORD
000004:        140024   (0067)        CHS                    PREPARE FOR COMPARISON
000005:        140104   (0068)        XCA                    SAVE IN B-REG
000006:      22.000001A (0069)        LDA     1,1            SECOND WORD ADDRESS
000007:      42.000001A (0070)        LDA*    1              SECOND WORD
000010:        140024   (0071)        CHS                    PREPARE FOR COMPARISON
000011:      11.000002A (0072)        CAS     2              COMPARE WITH FIRST WORD IN B-REG
000012:      01.000020  (0073)        JMP     NRETN          WORD1 < WORD2 (-)
000013:      01.000016  (0074)        JMP     ZRETN          WORD1 = WORD2 (0)
                        (0075) *                             WORD1 > WORD2 (+)
                        (0076) *
                        (0077) *      POSITIVE RETURN (WORD1 - WORD2 = +)
                        (0078) *
000014:        140417   (0079)        LT                     RETURN VALUE UCOMP=+1
000015:      21.000003A (0080)        JMP     3,1            RETURN
                        (0081) *
                        (0082) *      ZERO RETURN (WORD1 - WORD2 = 0)
                        (0083) *
000016:        140040   (0084) ZRETN  CRA                    RETURN VALUE UCOMP=0
000017:      21.000003A (0085)        JMP     3,1            RETURN
                        (0086) *
                        (0087) *      NEGATIVE RETURN (WORD1 - WORD2 = -)
                        (0088) *
000020:      02.000022  (0089) NRETN  LDA     MONE           RETURN VALUE UCOMP=-1
000021:      21.000003A (0090)        JMP     3,1            RETURN
                        (0091) *
                        (0092) *      DATA
                        (0093) *
000022:        177777   (0094) MONE   DEC     -1             =-1
                        (0095) *
                        (0096)        FIN
                        (0097) *
             000023     (0098)        END
```

Figure 4-2. Example of Assembly Listing

Column Allocation for Assembly Listing

Column	Description
1-2	Error codes
3-8	six digits of 16-bits each, containing the octal address or displacement
9	Contains address mode

:	procedure
>	link frame (Seg mode only)
.	common

11-24 Are dependent on the instruction types (i.e., memory ref. & non memory ref.):

non-memory ref- 17-22 contain octal representation of the instruction or data

1-word memory ref- 15-24 contain: 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 of a 16-bit address value. The last digit (C) indicates the mode of the address value.

Blank	Relative
A	Absolute
S	Stack relative/stack base relative (SB) - seg mode
E	External
C	Common
P	Procedure-base relative (PB) - seg mode
L	Link-base relative (LB) - seg mode
X	Temp-base relative (XB) - seg mode

Two-word memory reference (columns 11-24) as follows:

```
AAAAAA.BBBBBBC - A is opcode
                 B is address
                 C is address qualifier
```

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 4-3). The location and address values are in octal unless the PCVH pseudo-operation specifies hexadecimal listing. Each reference is identified by a four-digit line number. If listing is inhibited by the NLST pseudo-operation, the cross-reference is not listed.

```
MONE ØØØØ22 ØØ89 ØØ94

NRETN ØØØØ2Ø ØØ73 ØØ89

UCOMP ØØØØØØ ØØ63 ØØ64

ZRETN ØØØØ16 ØØ74 ØØ84

ØØØØ ERRORS (PMA-1Ø8Ø.Ø19)
```

Figure 4-3. Example of Cross-Reference Listing

ERROR DIAGNOSTICS

ERR	DESCRIPTION
C	INST IMPROPERLY TERMINATED
F	BAD TERMINATOR ON ARGUMENT # EXPRESSION (MACRO CALL)
	ILLEGAL OPERATOR ON STACK PUSH/POP
	FAIL PSEUDO-OP
G	GOTO ERROR WITHIN MACRO
	END/ENDM PSEUDO-OP WITHIN 'GOTO SKIP AREA
I	GENERIC, I/O, OR SHIFT HAS TAG MODIFIER
	TAG MODIFIER FIELD NOT PERMITTED ON 32I MODE FIELD INSTR
	SHORT INSTRUCTION SPECIFIER (#), CAN'T MAKE SHORT
	64V MODE, LDX CLASS INSTR, BAD TAG MODIFIER FIELD
	64V MODE, TAG MODIFIER NOT PERMITTED ON BRANCH INSTR
	SEG MODE, COMMON OR EXTERNAL REF, BAD INDIRECT OR INDEX
	AP/IP, INDEX SPECIFIER INVALID
	TAG MODIFIER NOT PERMITTED ON 32I BRANCH
L	IMPROPER LABEL (CONSTANT/TERMINATOR IN LABEL FIELD)
	EXTERNAL VARIABLE PRESENT IN LITERAL
	BAD ARGUMENT IN EQU, SET, OR XSET
M	MULTIPLY DEFINED LABEL
N	'END' WITHIN MACRO OR IF
O	UNRECOGNIZED OPCODE OR 32I-ONLY OPCODE IN NON-32I MODE
	64V MODE MEMORY REF, NOT IN 64V MODE
	S/R MODE MEMORY REFM NOT IN S/R MODE
P	MISMATCHED PARENTHESIS

Q	AP, NOT IN 64V/32I MODE
	IP, NOT IN 64V/32I MODE
	ENDM PSEUDO-OP NOT IN MACRO
R	STACK OVERFLOW
	MULTIPLY DEFINED MACRO OR MACRO NAME FIELD EMPTY
S	'LOAD' MODE, INSTRUCTION WOULD REQUIRE DESECTORIZATION
	INDIRECT DAC IN C64R MODE
T	32I MODE TAG MODIFIER SYNTAX ERROR
U	UNDEFINED VARIABLE IN ADDRESS FIELD / EXPRESSION
	UNDEFINED VARIABLE IN ORG/SETB
V	BIT FIELD IN BIT INST OUT OF RANGE
	UNRECOGNIZED OPERATOR IN EXPRESSION
	FIELD ADDRESS INST, FAR OUT OF RANGE
	I/O INST, FUNCTION CODE / DEVICE ADDR OUT OF RANGE
	SHIFT INST, SHIFT COUNT OUT OF RANGE
	FIELD ADDRESS INST, NO COMMA FOLLOWING FAR SPEC
	32I MODE REGISTER GENERIC, NO COMMA AFTER REGISTER #
	32I MODE FPR REGISTER GENERIC, NO COMMA AFTER REGISTER #
	32I MODE BIT TEST INSTR, NO COMMA AFTER REGISTER #
	32I MODE BIT TEST INSTR, NO COMMA AFTER BIT #
	32I MODE GEN REGISTER MEMORY REF, BAD DELIMITER
	32I MODE SHIFT INSTR, BAD DELIMITER
	BAD SHIFT COUNT IN 32I MODE SHIFT INSTR
	BAD TAG MODIFIER IN 32I MODE SHIFT
	BAD DELIMITER AFTER REGISTER # IN 32I MODE PIO INSTR

OPEN PARENTHESIS MISSING ON DFTB ARGUMENT

CLOSE PARENTHESIS MISSING ON DFTB ARGUMENT

LABEL MISSING ON IFTF, IFTT, IFVT, IFVF

NAME NOT FOUND IN IFTF, IFTT, IFVT, IFVF

ABS/REL ILLEGAL IN SEG MODE

SEG/SEGR AFTER CODE HAS BEEN GENERATED

PROC/LINK FOUND OUTSIDE OF SEG MODE

FIELD OUT OF RANGE ON DDM PSEUDO-OP

BAD ARGUMENT FOLLOWING 'EXT'

'END' WITHIN MACRO

SYNTAX ERROR IN 'DYNM' PSEUDO-OP

BAD ARGUMENT ON SUBROUTINE (SUBR) STATEMENT

VFD PSEUDO-OP, 16 BITS NOT DEFINED

UNTERMINATED CHARACTER STRING

EXPRESSION OVERFLOW ON FLOATING PT NORMALIZE

EXPRESSION OVERFLOW ON FLOATING PT RE-NORMALIZE

SCALED BINARY LOSS OF SIGNIFICANCE

FLOATING POINT NUMBER OUT OF RANGE

BCI REPEAT COUNT ERROR

BCI COUNT VARIABLE TYPE ERROR

CALL CONTAINS CONSTANT OR TERMINATOR IN ADDR FIELD

COMMON (COMN) PSEUDO-OP HAS BAD ADDRESS FIELD

DEC, DATA, DBP, HEX, OR OCT REPEAT COUNT ERROR

DEC/OCT PSEUDO OP HAS BAD OPERATOR

RLIT FOUND AFTER CODE HAS BEEN GENERATED

NO LABEL ON DFTB

X	32I MODE GENERAL REGISTER SPECIFICATION ERROR
Y	PHASE ERROR
Z	ILLEGAL ABSOLUTE REFERENCE IN SEG MODE
	SEG MODE, ABSOLUTE REF NOT PERMITTED UNLESS Ø-7
	AP/IP, ABSOLUTE REF INVALID
	MORE THAN 1 EXTERNAL NAME IN AN EXPRESSION
	INCORRECT EXPRESSION MODE FOR GIVEN INSTRUCTION
	EXPRESSION MODE ERROR
	>1 OPERATOR NON-ABS/REL OR RIGHT-HAND OP NOT ABS/REL
	EXTERNAL NAME NOT PERMITTED

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 pseudo-operations in the source program.

Expanded Symbol Table Area (Bit 7)

When bit 7 is set, the assembler uses the entire 64K virtual space for symbol and macro storage during assembly.

NOTE

Bit 7 should be set only for a PRIMOS III or IV system.

Device Options (Bits 8-16)

The last three octal digits of the A Register select source, listing input, and object output devices respectively, as shown in Figure 4-1.

SECTION 5

FORTRAN COMPILER (FTN)

INTRODUCTION

This section describes the run and compile procedures for Prime's Rev. 11 FORTRAN compiler.

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.

The one-pass compiler operates in PRIMOS II, III, or IV environments. The compiler produces highly optimized code and is supported by an extensive library of mathematical functions and subroutines.

Object code generated by the compiler is in a format suitable for loading by Prime's Linking Loader or segmentation utility. Library subroutines are in the same format. The FORTRAN compiler also generates object code in segmented (64V) mode suitable for processing by SEG on a Prime 400.

SOURCE PROGRAMS

Source programs must meet the requirements of the Prime FORTRAN IV Language Reference Manual (MAN 1674).

A source program is typically prepared at a user terminal, using the Prime text editor. It must be accessible in the users UFD under the assigned filename.

OPERATION UNDER PRIMOS

The FORTRAN compiler is invoked by the FTN command to PRIMOS:

```
    FTN Filename [1/A-register] [1/B-register]
```

The FTN command loads the compiler and starts compilation of an object program by reading an ASCII source file, Filename, in the current UFD.

A- and B-Register Options

The A-register and B-register parameters control compiler functions such as input and output device selection, listing detail, trace enable, concordance enable, and others. The functions and default values are summarized in Tables 5-1 and 5-2 and described in detail at the end of this section. Some common options are:

A-Register	Option
1777	Lists errors on terminal and generates listing file
40777	Generates listing file that includes symbolic listing

B-Register	Option
10	List errors on terminal and create cross-reference listing
400	Generate Prime 400 64V-mode code

File Usage

Three files may be involved during a compilation:

File Type	PRIMOS File Unit
Source	1
Listing	2
Object	3

FTN will automatically open files for the listing and object output, provided disk is specified as the destination for those files. The names are formed by prefixing the first four letters of the source filename with B<- for the object output file (binary file) and with L<- for the listing outut. If the user prefers other names, he can used the PRIMOS BINARY and LISTING commands to open files on units 2 and 3 before invoking FTN.

Opening File Unit 2 before the FTN command allows the listing output of more than one source file to be concatenated, the file to be written in other than the current UFD, and the filename to be other than L<-XXXX. File Unit 2 is opened by using the PRIMOS LISTING command.

All file units opened by FTN are closed before FTN returns control to PRIMOS. However, files opened by the user are not closed or truncated by FTN.

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,

Table 5-1. Typical A-Register Default Values

Bit	Option	Default
1	Special Library Compilation Flag	0
2	Symbolic Instructions	0
3	Error Listing Only	0
4	Global Trace	0
5	64R Mode	0
6	In-Line Desectorization	0
7	Print Errors at User Terminal	1
8-10	Select Source Device	7
11-13	Select Listing Device	0
	0	
	0	
14-16	Select Output Device	7

Table 5-2. Typical B-Register Default Values

Bit	Option	Default
8	Prime 400 Segment Addressing Mode	Ø
1Ø	Long Integers	Ø
12	Partial Concordance	Ø
13	Generate Concordance	Ø
15	Suppress Floating Point	Ø
16	Flag Undeclared Variables	Ø

The object output is compiled to run in 32R addressing mode unless bit 5 of the A Register is set, or bit 8 of the B Register is set.

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 PRIMOS systems, control returns to command level after the last END statement. An end of file also terminates compilation.

The 0000 ERRORS message indicates that the program has been compiled without errors. If any errors are encountered, the number of ERRORS is printed. If bit 7 of the A Register is set, error lines and error messages are printed on the user terminal. Otherwise, the user must print the listing file to find where the errors occurred.

LISTINGS

Listing Options

Listings may be obtained at several different levels of detail.

To create a List File: an L<-File is created when the listing device on the A Register (bits 11, 12 and 13) specifies the Disk.

Bits 2 and 3 of the A Register and the NOLIST, LIST, or FULL LIST statements in a program determine the detail level of the listing.

To print a listing atthe user terminal rather than disk file or other device, set A-register bits 14-15-16 to octal 1.

Figure 5-1 is an example of a full listing at the user terminal.

Compiler Error Messages

Coding errors and misprints are flagged on the listing by a line containing a set of asterisks (to attract attention) and an error message containing the source context at the point the error was detected. Error messages are self-explanatory text comments.

The following example contains one error. The error is denoted by four asterisks followed by the line number and the context when error was detected. (The expression in a computed GO TO statement must yield an integer result).

Example:

```
    SLIST POO
     GO
     310     X=48
             B=I*5
             C=5-I
             I=3
     20      GO TO (100,310,320),X
     320     A=B + C
             I=1
             GO TO 20
     100     Y=A*X
             WRITE (1,110)Y
     110     FORMAT (I5)
             CALL EXIT
             END

     OK, FTN POO
     GO
     (0006)  320    A=B + C
     **** LINE 0005  [ 310,320),X ]  DATA MODE ERROR
     0001 ERRORS (FTN=1082.L13)
```

```
        FTN POO 1/40717
GO
 310     X=48
(0001)   310     X=48
(0002)           B=I*5
(0003)           C=5-I
(0004)           I=3
(0005)   20      GO TO (100,310,320),I
(0006)   320     A=B + C
(0007)           I-1
(0008)           GO TO 20
(0009)   100     Y=A*X
(0010)           WRXTE (1,110)X
**** LINE 0010    [ WRXT ]    UNRECOGNIZED STMT
(0011)   110     FRMAST (I5)
**** LINE 0011    [ FRMA ]    UNRECOGNIZED STMT
(0012)           FULL LIST
(0013)           CALL EXIT
(0014)           END
         000041:    JST   EXIT
         000042:    LINK  A
         000042:    OCT   000000
         000043:    OCT   000000
         000044:    LINK  B
         000044:    OCT   000000
         000045:    OCT   000000
         000046:    LINK  C
         000046:    OCT   000000
         000047:    OCT   000000
         000050:    LINK  I
         000050:    OCT   000000
         000051:    LINK  X
         000051:    OCT   000000
         000052:    OCT   000000
         000053:    LINK  =3
         000053:    OCT   000003
         000054:    LINK  =5
         000054:    OCT   000005
         000055:    LINK  =24576
         000055:    OCT   060000
         000056:    OCT   000206
         000041:    DAC   _100
**** LINE 0011  [ END ]   _110 - UNDEFINED STMT NO.
**** LIN000022: [ DAC ]   _2010 - UNDEFINED STMT NO.
        000001:   DAC     _310
        000030:   DAC     _320
0003 ERRORS (FTN-1082.L13) 0003 ERRORS (FTN-1082.L13)

OK,
```

Figure 5-1. FULL LIST Example

LIBRARY ERROR MESSAGES

During program execution, certain library subroutines may detect error conditions and invoke printing of an error message through the PRIMOS error message facility. All error codes are self-explanatory text messages and include the name of the subroutine from which they originate.

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. See Figure 5-2 for sample lines of trace information as typed at object run-time.

```
    FTN PRIME 1/11707
GO
0000 ERRORS [<.MAIN.>FTN-REV13.1]

OK, LOAD
GO
$ LO B_PRIME
$ LI
LC
$ SA *PRIME
$ EX
FOLLOWING IS A LIST OF PRIME NUMBERS FROM 2 TO 50
                    2
                    3
                    5
                    7
 K=      3
 (2)
                   11
 (4)
 K=      3
 (2)
                   13
— — — — — — — — —

— — — — — — — — —
 K=      6
 (4)
 K=      6
 (2)
 (2)
                   47
 (4)
 K=      7
 (2)
 (2)
 (4)
 THIS IS THE END OF THE LIST

 ****ST

 OK,
```

Figure 5-2. TRACE Example

A REGISTER DETAILS

The A Register provides a variety of FORTRAN options, as defined in Figure 5-3.

When new values are required, use the following FORTRAN compiler command with option:

```
    FTN Filename [1/Areg]
```

where Filename is a FORTRAN source program in the current UFD, and Areg is an A Register setting that specifies listing detail and input/output devices.

Input and Output Device Options (Bits 8-16)

The normal input and output device is disk. However, other devices can be specified using the A-Register bits 8 through 16, when the system is configured to include other devices such as the mag tape and line printer.

Listing Detail Options (Bits 2, 3)

The listing detail options are selected using bits 2 and 3 as follows:

	A Register Bit 2	Bit 3
LIST (source statements W/LINE NOS. and error messages)	Ø	Ø
NO LIST(error messages only)	Ø	1
FULL LIST (Assembly Language type listing plus source statements and error messages)	1	Ø

Bits 2 and 3 have no effect unless bits 12 through 13 are used to specify the output device for the listing file.

<u>DEVICE OPTIONS</u>

- 0 = NONE
- 1 = USER TERMINAL
- 2 = PTR/PTP
- 3 = Reserved for CARD READER/PUNCH
- 4 = Reserved for LINE PRINTER
- 5 = Reserved for MAGNETIC TAPE
- 6 = Not used
- 7 = DISK (FILE SYSTEM)

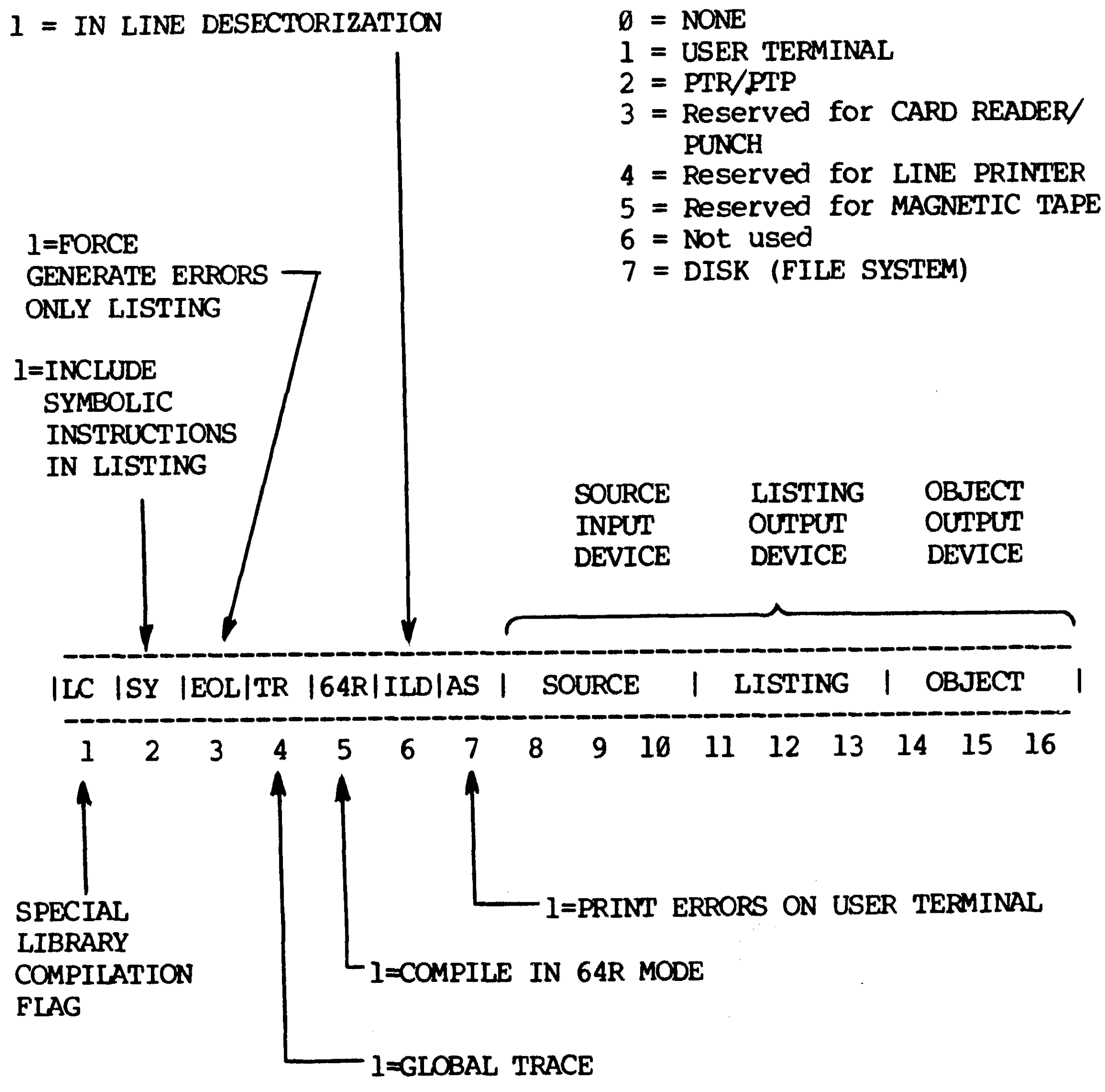


Figure 5-3. Compiler A Register Settings

Printing Errors at a User Terminal (Bit 7)

The normal system (default) allows each statement containing an error to be printed at the user terminal. This feature is especially useful when a corrected program is being recompiled, to confirm that the errors have been corrected properly.

Library Mode Flag (Bit 1)

When the Library Mode Flag is set, certain statements and program formats that would normally be flagged as errors are permitted. It also causes reinterpretation of some statements. This bit is used on the compilation of some Prime-supplied software and is not recommended for general use.

Global Trace (Bit 4)

When this option is selected, a trace printout is generated at all assignment statements and at every statement number in the program unit. The global trace option affects only the program unit being compiled. It has no effect on other program units in the same executable program.

Utilizing 128K Bytes of User Space (Bit 5)

64K bytes of user space is available to each FORTRAN user under the 32R mode (default). This means that if the main program, subprograms, , all local storage, the library routines, and the common blocks all require a sum total of less than 64K bytes, the 32R mode default is sufficient. However, a larger user area can be utilized when required by setting bit 5 of the A Register (64R mode) when compiling the main programs and all subprograms. The MODE command in the LOADER utility must also be used to change load mode to 64R. This assures the user 128K bytes of user space.

Generally, it can be determined if the 64R mode must be selected by looking at the storage areas. Each area requiring space such as the common blocks can be examined. If the common blocks require more than 64K bytes, then the 64R mode decision is obvious. For example, if it is on the boundary and a load is attempted resulting in an overflow, it is likely that the addresses for the common are overlapping the program area.

Reducing Sector Zero Requirements of a Large Program (Bit 16)

In-Line Desectorization (bit 6) when set, reduces the sector zero requirements of large programs. 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. Programs compiled with this option can be loaded only in the relative addressing modes (a loader NS diagnostic is generated if an attempt is made to load in a sectored addressing mode).

B REGISTER DETAILS

Additional options are available through the octal value of the B Register (see Figure 5-4 and Table 5-2). These include: 64V mode, 32-bit integer, full concordance, suppresing floating-point skip instructions, flagging undeclared variables, etc.

```
FTN Filename 1/Areg  Breg
        or
FTN Filename 1/Areg  2/Breg
        or
FTN Filename 2/Breg (Default Areg is used)
```

Utilizing Segmented Addressing Space (Bit 8)

When large programs require more than 128K bytes of user space, any Prime 400 (or higher) system allows a FORTRAN program to run by providing a user area up to two megabytes long. This is called the 64V mode and is selected by setting bit 8 in the B Register (see Figure 5-2).

When bit 8 is set, software features allow FORTRAN programs of up to two megabytes long (15 segments of 128K bytes) to be executed under PRIMOS IV.

Each common block can be up to 128K bytes long. The local sum of storage (local variable, arrays, indirect pointers) of any program unit (Main program or subprogram) can be up to 128K bytes.

NOTE

The size restriction on COMMON blocks (128 bytes) and total program size (15 segments) are limitations of REV 11 software. These size restrictions will be eased on later revisions.

The LOAD utility and load modes are dictated by the options selected at compile time, as shown in the following table:

UTILITY	COMPILER OPTION	LOAD OPTION
LOAD	32R (default)	32R
	64R	64R,32R
SEG	64V	64V

Any PRIMOS system can use either the 32R or 64R addressing mode. Only a Prime 400 (and up) can have 64V addressing mode.

Long Integer (Bit 10)

The normal INTEGER data type in PRIME FORTRAN is a 16-bit word. A 32-bit INTEGER data type is available through use of the INTEGER* 4 type statement.

The long integer default bit is used to ease conversions of Fortran programs to PRIME computers. When this bit is set all variables, arrays, and functions explicitly or implicitly as INTEGER will be 32-bit INTEGER. Additionally, all integer constants will be treated as 32-bit integers. Only those names appearing in INTEGER*2 type statements will be 16-bit integers.

The 32-bit integer has a greater range than the 16-bit integer (2, 147, 483 vs 32,767). The 32-bit integer hs the same storage requirement as the REAL data type.

WARNING:

FORTRAN requires that the type of actual argument in a function reference or CALL statement must agree with the corresponding dummy argument in the referenced subprogram. Note that a subprogram expecting a long integer must NOT be called with a short integer (and vice versa). Most Prime-supplied subroutines expect short integer arguments. Care should be taken when calling these routines (e.g., SEARCH) in a program compiled with the LONG INTEGER default option.

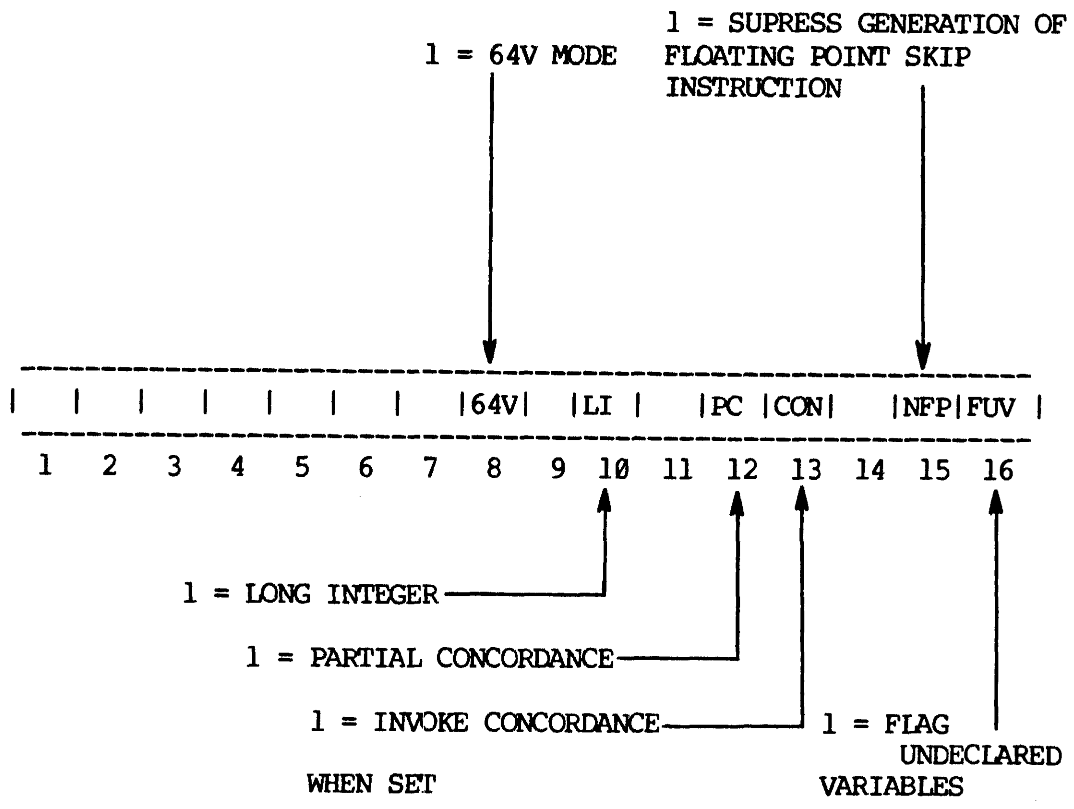
Example:

```
CALL SEARCH (INTS(1) 'FILENM',INTS(1)
```

INTS is a built-in function that connects its arguments to a short integer. If the INTS are omitted, the integer constants are compiled as long integers, providing B Register bit 10 was set during compilation.

Suppressing Floating Point Skip Instructions (Bit 15)

The compiler will generate instructions from the floating point skip set when testing the result of a floating-point operation. This may be suppressed by setting bit 15 in the compiler's B Register setting when compiling for machines that do not have the floating- point option (programs will still execute on such machines even if the bit was not set; the UII time will just be higher).



NOTE

The default B-Register is Ø.

Figure 5-4. Compiler B-Register Setting

Flag Undeclared Variables (Bit 16)

If bit 16 in the compiler's B Register is set for a compilation, the compiler will generate an error message when a variable is used in the program, but not included in a specification statement. The message will be generated once per undeclared variable.

Concordance (Bits 12, 13)

A concordance (symbol cross-reference) can be added to the listing under control of B-register bits 12 and 13:

Bit 12	Bit 13	Selected Option
Ø	Ø	No concordance
Ø	1	Full concordance
1	1	Partial concordance

The partial concordance does not include symbols that are referenced only in specification statements.

SECTION 6

PRIME COBOL COMPILER

COMPILATION

When it is desired to compile a COBOL program, the user simply types in:

```
      COBOL Filename
```

Filename is a COBOL source file in the current UFD. The result of this command will be the creation of two new files:

```
      L<-XXXX  (COBOL listing file)
      B<-XXXX  (COBOL object file)
```

L<-XXXX is a listing file containing the COBOL source with line numbers followed by an error list for the compilation.

B<-XXXX is a binary file suitable for loading by the system and subsequent execution.

XXXX are the first four characters of Filename.

LOADING

Loading of Prime COBOL programs for execution utilizes the output from the COBOL compiler and the Prime Linking Loader.

The commands are:

Indexed and Relative(MIDAS)	Sequential(Non-MIDAS)
HILOAD	LOAD
MODE D64R	LO B<-XXXX
LO B<-XXXX	LIB COBLIB
AU 2Ø	LIB FTNLIB
LIB COBKID	SAVE *XXXX
LIB FTNLIB	QUIT
SAVE *XXXX	
QUIT	

Either of these command sequences invokes the Prime Linking Loader, loads the binary file, ties in the necessary files from the UFD LIB, and saves the memory image. The Loader creates an executable (saved memory image) file. If the program is large, set MODE to D64R after invoking the Loader and use the AU command following the loading of the binary file. Refer to the Loader utility in Section 7.

EXECUTION

Execution of a COBOL program is initiated with the command:

```
R *XXXX
```

where *XXXX is the name of the file containing the saved memory image from the loading process. Upon execution of this command, the user will be asked questions regarding run-time file assignments.

The program will ask:

```
ENTER FILENAME AND UNIT
```

The proper response is to give the name of the file, as stated in the VALUE OF FILE-ID clause in the FILE DESCRIPTION, followed by a file system treename or a magnetic tape descriptor.

For example, suppose that in the COBOL program the following statements existed:

```
FD  TEST-FILE
    LABEL RECORDS ARE STANDARD
    VALUE OF FILE-ID IS FILE1
```

Then the proper response would be:

```
ENTER FILENAME AND STANDARD
>FILE=PETERS>T1

>FILE1=$MT1,S,T1,000001
```

The first statement will go to a UFD named PETERS and use a file named T1 as input to TEST-FILE in the program.

The second statement requires magnetic tape unit number one to be assigned and the tape mounted that contains TAPE-ID of T1 and a volume serial number of 000001.

A COBOL utility program, CM$L, will do all the pre-screening of the files and display the character >, as a prompt character waiting for more input. When no files remain to be entered, the single character / will conclude CM$L. Execution of the program will start, using the files that were previouslv entered.

CM$L will display the following error messages:

```
FILENAME TOO LONG (no equal sign found)
INVALID TREE SYNTAX (see allowable format)
NO FILE NAME ENTERED (equal sign with no file name)
INVALID TAPE UNIT (format did not contain MTx)
```

NO TAPE NAME ENTERED (standard label specified)
INVALID STANDARD/NON LABEL (not S or N)
TAPE NAME GREATER THAN 17
TAPE NUMBER GREATER THAN 6

DISK FILE ASSIGNMENT

The procedure for assigning a disk file follows the rules of tree names. For additional information about treenames, refer to the FUTIL section of the PRIMOS File System Users Guide.

Format:

```
           |Ufdname [Password] [Logical-disk-number(octal)] > Filename|
  FILE-ID= |* > Filename                                              |
           |Filename                                                  |
           |Volumename > Ufdname [Password] > Filename                |
```

MAGNETIC TAPE FILE ASSIGNMENTS

The format of magnetic tape file assignments is:

```
  FILE-ID=$MTn,|N           |,Tape-number
               |S,Tape-id   |
```

Where $MTn	is a 9 track drive number.
N	specifies no label information.
S	specifies the tape contains standard labels and is pre-numbered.
Tape-id	is a 1 to 17-character field. If the tape is being read it is compared to the ID on the tape. If the tape is being created it is written into the label of the tape. The label variable must be specified as S.
Tape-Number	is a 6-character field which is checked at open time when reading a tape, and is not needed when creating a tape.

SECTION 7

LINKING LOADER (LOAD)

INTRODUCTION

This section defines Prime's linking loader and provides user instructions and diagnostic information up to and including Rev. 11.

FEATURES

Prime's linking loader offers the following features:

1. The loader is capable of loading code anywhere in 64K, above or below itself or COMMON (except on top of itself!).

2. The location of COMMON is movable by a keyboard command.

3. An indefinite number of base areas can be specified; the loader automatically uses the first available area which can be reached, in preference to the sector Ø linkage area.

4. 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 UII library routines will load automatically.

5. Partial or full load maps can be displayed on the user terminal or written to a disk file.

Overview

The function of the Linking Loader is to combine into an executable program, a number of program units or subroutines that have been independently compiled. Some of the subroutines may have been held in a library; the linking loader provides the facility for automatic incorporation of any library subroutines that have been referenced in the main program, as well as resolving the cross-references between them.

Desectorization

The loader performs a function during loading called desectorization. The need for this function arises because one-word memory reference instructions cannot directly reference all of memory. The loader compensates for this by generating a pointer to the operand in a base area and then modifies the instruction to reference through the pointer.

The pointer default base area is from memory location `200 to `777. For many programs, this area will be sufficient. However, for larger programs this area might be inadequate. The loader has a number of commands to enlarge the default base area and create local base areas.

The base area below location '1000 can be used to desectorize any instruction, no matter what its location. Local base areas (above location '1000) can be used only to desectorize instructions in a window around the local base area. The window extends approximately '400 locations above and below the base area. (See Figure 7-1).

The loader uses local base areas when possible in preference to base area below location '1000. The location in base areas used by the loader are not available for any other use during program loading or execution.

USING LOADER UNDER PRIMOS

All loader functions are available through user terminal keyboard commands. When the the LOAD command is typed, the linking loader is in command; the loader prints the "$" prompt character on the user terminal and awaits a command line.

Example:

```
        LOAD
        $
```

The $ prompt character means that the loader is in command mode until a QUIT command is received. Each prompt character may be followed by a loader command, according to the command definitions.

When a system error is encountered, on disk access, the loader will print the system error and return to its command request symbol ($).

```
|--------------|    Initial location of * PBRK
| Base Area    |
|--------------|
|              |
|              |
|              |
| File Length  |
|              |
|              |
|              |
|--------------| Location of * PBRK at end of load
|              |
| Base Area    |
|--------------| Location of * PBRK for start of next load
```

Figure 7-1. FO and LO - Base Area Orientation

NOTE

The loader also accepts commands from a command file.

COMMAND DEFINITIONS

Each loader command consists of a command name followed by a series of arguments in the same format as the PRIMOS command line:

```
    COMMAND  Name1  Name2  Arg1  Arg2 . . .Argn
```

where COMMAND is the command name, each Name is a text string which may be a PRIMOS filename or UFD name, and each Arg is an octal argument (numeric only) of up to six octal digits. Command names may be abbreviated to two characters. Arguments are separated by spaces. Up to three alphanumeric fields (non-numeric first) and nine (numeric only) arguments are allowed. In some cases, it is possible to omit arguments. The kill (?) and erase (") are supported.

SUMMARY AND INTRODUCTION TO COMMANDS

The commands are summarized below and described in detail in alphabetical order following the summary.

Command	Function
ATTACH	Attach to different Ufd
AUTOMATIC XXXXXX	Automatic generation of Setbase areas
*	Comment line
COMMON	Relocate common address
EXECUTE	Direct program execution
HARDWARE	Hardware definition
INITIALIZE	Reinitialization
LIBRARY	Loads library files

LOAD	Loads object files
MAP	Load state map
MODE	Select addressing mode
SAVE	Saves loaded memory image
SETBASE	Defines a new linkage area
VIRTUAL BASE	Controls the deletion of symbols

AT̲TACH [Ufd] [Password] [Ldisk] [Key]

Attaches to different UFD's. This command is converted into a CALL to the PRIMOS subroutine ATTACH and has exactly the same effect.

Ufd:	Any User File Directory. However, the user attached is to the home Ufd when no Ufd name is specified.
Password:	The user gets owner status if he gives the owner password and nonowner status if he gives a non-owner password. The password parameter is necessary only when the UFD is password-protected.
Ldisk:	If the Ldisk parameter is omitted, the loader searches only device Ø for the specified UFD. If an Ldisk value of '1ØØØØØ is specified, the loader searches all started devices in logical unit order.
Key:	The values for Key most likely to be useful during loading are:

Ø Do not change home UFD.

1 Adopt named UFD as home UFD.

2 Attach to a subUFD in the current UFD; do not set as home.

3 Attach to subUFD in the current UFD; set as home.

AU̲TOMATIC XXXXXX

Causes the loader to insert a base area of length XXXXXX whenever the loader detects the end of a routine and more than 3ØØ (octal) locations have been loaded since the last base area was inserted.

The value of XXXXXX may be changed between load files. This automatic feature may be turned off with an AU Ø command.

AUtomatic helps to reduce the number of loads which run out of Sector Ø link space, by instructing the loader to insert linkage areas automatically.

* Comments

Comments may be included in a command file when an asterisk preceeds the comment. This line is not processed by the loader.

Example:

```
* COMMAND.FILE.TO.LOAD.THE.LOADER
FILMEM
* INVOKE.OLD.LOADER.
LOAD
$LO B<-LOAD 174000
$SA HILOAD
$* NOW.USE.NEW.TO.CREATE.NEW.LOAD
$EX
$* NOW.WE.ARE.IN.HILOAD
$LO B<- LOAD 60000
$SA LOAD
$QU
```

NOTE

The <- (left arrow) is used in place of the left arrow which was not available on the printer terminal.

COMMON Address

Moves the top or starting location of FORTRAN-compatible COMMON to the address specified. Space for COMMON items is allocated downward from but not including the starting location.

The top of COMMON is the last location used for COMMON by the loader.

The default COMMON load address is the last location in the last loader sector. This means, for example, that the top of COMMON for LOAD is 63777 (for HILOAD, it is 177777).

NOTE

To specify a COMMON load point, (top or starting location) give the location desired + 1. For example, CO 40000 specifies 37777 as the top location in COMMON. This is for compatibility with previous releases of the LOADER.

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.

FORCELOAD Filename [Loadpoint] [Base-start] [Base-range] (Format 1)

or

FORCELOAD Filename * Prebase Filesize Postbase (Format 2)

Same as LOAD but Filename is loaded unconditionally. However, all subroutine references must be satisfied.

HARDWARE Definition

Defines the instruction execution hardware of the CPU on which the removed from the UII requirement. The Definition parameter is one of the following octal values:

CPU	Definition
P400	17
P300/FP	17
P300	3
P200/HSA	1
P100/HSA	1
P200	0
P100	0

FP: with optional floating - point. HSA: with optional high - speed arithmetic.

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, for comparison with the user's hardware definition (if any).
Example:

```
HA 3
```

The 3 selects a Prime 300 without floating-point.

In the event that a program requires hardware which is not present in the user's system, these outstanding hardware requirements may be satisified by the command, LI UII, which should be the last LOAD command before the program is saved. The appropriate routines will be selected from this library to satisfy the additional hardware requirements of the program.

The user may determine whether it is necessary to load the UII package by examining the value for *UII in a Load Map.

INITIALIZE [Filename][Other Options]

Initializes the loader and then optionally performs the same actions as a LOAD command. In the loader's initialized state, the load state parameters (Table 7-1) return to their default p values. If no Filename is provided, the loader repeats its prompt character ($).

Other options: Refer to the loadpoint, linkstart and linkrange options available under the LOAD command.

LIBRARY [Filename][Loadpoint]

Temporarily attaches to the LIBRARY UFD, loads from the specified filename, and returns to the original UFD. Loadpoint is an optional starting address. If no Filename is provided FTNLIB is loaded by default.

```
LOAD Filename [Loadpoint][Base-start][Base-range]          (Format 1)
LOAD Filename * Prebase Filesize Postbase                  (Format 2)
```

<u>Format 1</u>

Loads the specified object file (Filename) into memory starting at at loadpoint or, if loadpoint is omitted, at the current *PBRK location. Base-start and Base-range define a base area as in a SETBASE command. When loading is complete, *PBRK points to the location following the highest location used by the object file.

<u>Default Load Parameters</u>

If the load parameters are not assigned by the command string, the following default values apply:

Loadpoint	*PBRK (initially '1000)
Base-start	'200
Base-range	'600

NOTES

1. If all of the symbols in the load module have been previously defined, the loader skips the module. A load module is defined to terminate with an "END" statement. To force load a module which contains only previously defined symbols, use FO Fname. The FO command will ensure loading of the first module in an object file.

2. Since the compiler (e.g., FORTRAN, COBOL) converts the program to binary format, a new name (e.g., B<-MUX) is created by the compiler. This binary version must be specified in the LOAD command.

Example:

A FORTRAN program called MUX when compiled would be converted to binary format and assigned a name of B<-MUX. The programmer loads this program as follows:

```
OK, LOAD
$ LO B<-MUX
$ LI
LC
$
```

Format 2

Loads the specified object file (Filename) and defines a base area before and/or after the loaded file (see Figure 7-1). The current *PBRK is used as the first location for this operation.

Prebase is the length of the base area that is to precede the file to be loaded (X may be zero).

Filesize is the length of the file to be loaded (in octal).

Postbase is the length of the base area to follow the loaded file.

If Postbase is zero, there will be no terminal setbase area. The loader remembers the location of the end of the second setbase area. Before loading the next object file (if any), PBRK is left unmodified so that the programmer may easily verify that the specified length of the object file is correct. The next module, however, will be loaded after the terminal setbase area (if any).

The loader's remembered PBRK may, of course, be overridden by a regular LO command for the next object file.

MAP [Filename] [Option]

Lists a load map. Filename is the name of the map file to be opened, and Option is an octal value that selects one of four map options. The loader will close the map file if it opened it. In any case, the map file is truncated.

Option Number	
None	Load state, base area, and symbol storage map
1	Load state only
2	Load state and base area
3	Unsatisfied references only

MAP Option 1 - Load State Map

The load state map identifies the lowest and the highest storage memory location, the location at which the program execution will begin, the next location available for loading, the high and low common area, the lowest location used by the symbol table, and the net hardware UII package requirement.

These eight parameters are printed in the load state map with a corresponding storage address. (See Table 7-1.)

Table 7-1. Load State Definition

Parameter	Definition	Default
*LOW	The lowest location in memory loaded	177777
*HIGH	The highest location in memory loaded	0
*START	The location at which execution will begin	0
*PBRK	The next location in memory to be loaded	1000
*CMLOW	The lowest location in COMMON	XX777
*CMHIGH	The highest location in COMMON	XX777
*SYM	The lowest location used by the symbol table	YY000
*UII	The net hardware/UII package requirement (see HARDWARE command for meaning)	0

NOTE:

XX = Last Sector in loader
Occupied by Loader

YY = First Sector
Occupied by Loader

Example:

```
    OK, LOAD
    GO
    $ LO B<-FTND
    $ LI
    LC
    $MA 1


*START 001000   *LOW     000074   *HIGH    007277   *PBRK    007300
*CMLOW 063753   *CMHGH   0063753  *SYM     057331   *UII     000015

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

MAP Option 2 - Load State Map and Base Area Map

The base area map includes the lowest and the highest locations and the next available locations. Each line contains four addresses as follows:

```
*BASE    XXXXXX   YYYYYY   ZZZZZZ   WWWWWW

    XXXXXX = Lowest location defined for this area
    YYYYYY = Next available location if starting up
             from XXXXXX
    ZZZZZZ = Next available location if starting down
             from WWWWWW
    WWWWWW = Highest location defined for this area
```

The base area map includes a load state map

MAP Option Number Omitted - Full Map

A full map contains all components of a load map including a full symbol storage listing.

The symbol storage listing consists of every defined label or external reference name printed four per line in the following format:

```
        Namexx  NNNNNN

                or

        Namexx  NNNNNN**
                             was referenced)
```

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:

This example illustrates how the loaded memory image can be stored as a file (RUNFIL) in the UFD, and a map saved to a file MAP1.

Load State Map 1

```
OK, LOAD
GO
$ LO B<-SIMP
$ LI
LC
$ MA 1
*START  001000  *LOW    000200  *HIGH   006512  *PBRK   006513
*CMLOW  063777  *CMHGH  063777  *SYM    057401  *UII    000001
```

Load State and Linkage Area Map 2

```
$ MA 2
*START  001000  *LOW    000200  *HIGH   006512  *PBRK   006513
*CMLOW  063777  *CMHGH  063777  *SYM    057401  *UII    000001

*BASE   000200  000220  000777  000777
*BASE   001527  001571  001570  001570
*BASE   002515  002557  002556  002556
*BASE   003404  003427  003434  003435
```

Unsatisfied References Only MAP 3

```
$MA 3 (No unsatisfied references, therefore no printout)
```

Load State, Linkage Area and Instruction Storage MAP 4

```
$ MA
*START  001000  *LOW    000200  *HIGH   006512  *PBRK   006513
START   063777  *CMHGH  063777  *SYM    057401  *UII    000001

*BASE   000200  000220  000777  000777
*BASE   001527  001571  001570  001570
*BASE   002515  002557  002556  002556
*BASE   003404  003427  003434  003435

LIST    000001  F$WA    001020  F$WX    001026  F$IO    001102
F$A1    001501  F$A3    001501  F$A2    001505  F$A5    001505
F$A6    001512  F$CB    002034  F$IOBF  004660  F$ER    004762
F$HT    004767  AC1     005047  AC2     005050  AC3     005051
AC4     005052  AC5     005053  WRASC   005054  IOCS$   005061
IOCS$T  005160  F$AT    005172  F$AT1   005174  WATBL   005237
LUTBL   005256  PUTBL   005313  RSTBL   005350  O$AD07  005405
```

Figure 7-2. Storage MAP Example

```
OK, LOAD

GO

$ LO B<-SIMP

$ LI

LC

$MA MAP 1

$SA RUNFIL

$ EX

TEST MESSAGE
```

Filename RUNFIL is now stored in the current UFD and filename XX contains the MAP.

MODE parameter

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

Parameter	Addressing Mode
D16S	16K Sectored
D32S	32K Sectored
D32R	32K Relative (default value)
D64R	64K Relative
D64V	64K Virtual (executable on P-4ØØ only)

NOTE

The mode command is used when an addressing mode other than 32k relative is required.

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 command level with the user attached to the home UFD or the last UFD specified in an ATTACH command. If the loader has opened a MAP file, it is closed and truncated at this time.

SAVE Filename [Aregister][Bregister][Xregister]

Saves the loaded memory image from *LOW to *HIGH, including all initialized COMMON areas, 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).

```
OK, slist xx
GO
*START  001000  *LOW    000200  *HIGH   006603  *PBRK   006604
*CMLOW  063777  *CMHGH  063777  *SYM    057374  *UII    000001
```

```
SETBASE|Base-start|Base-range
       |    *     |
```

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

The user may wish to increase the size of the sector zero base area by the command: SE 100 at the start of his load session. The beginning of the sector zero base area should not be made lower than '100.

The default values are:

Base-start	'200
Base-range	'600

Base-start can be set at the current location by effectively defining Base-start as "*". The command SE * will cause the creation of a setbase area of the specified length to be inserted at the current location. Thus, if PBRK (Base-start) is 1765, the command SE * 20 will create a setbase area of length 20 at 1765 and the PBRK will be set at 2005 after the command has been executed.

VIRTUALBASE Base-start To-sector

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

XPUNGE Dsymbols Dbase

Delete COMMON symbols, other defined symbols and base areas. Dsymbols controls the deletion of symbols as follows:

Dsymbols	
Ø	causes all symbols except undefined symbols to be deleted.
1	causes all symbols except undefined symbols and symbols for COMMON areas to be deleted.

Dbase controls the deletion of base areas;

Dbase	
Ø	deletes all defined base areas from the symbol table.
1	deletes all defined base areas except sector zero base areas.
2	retains all defined base areas

NOTE

When a symbol is defined at any time as in COMMON, the entry in the symbol table will appear as a COMMON symbol.

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.)

Message	Meaning
LC ied.	Load complete. All external references are satisif (This does not imply satisfaction of all UII requirements.)

Error Messages

Message	Meaning
CM	COMmand error. Illegal command format or non-existent filename specified.
GT	Group Type error. The loader has encountered an unrecognizable piece of object text. Loading is discontinued.
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. The source module is not an object file (output of FTN, PMA, etc.) or is a P400 object file.
MO	Memory Overflow Errors As users' programs become larger MO (memory overflow) errors become more common. This section contains a description of the several causes of these errors and suggested solutions to these causes. When an MO error occurs, the user should do a 'MA 2' and examine the map for the following possible situations: a. The address of the bottom of the symbol table (*SYM) is at or close to PBRK. This indicates that there is not enough room below the loader for the whole program. HILOAD will probably solve the problem -- assuming the user is not already using HILOAD.

b. The sector zero base area is full - the next free location is '1000. The size of the sector zero base may be increased by a SETB'100 command at the beginning of the load - if locations 100 to 200 are free - or an AU command may be used to insert base areas throughout the load.

c. *CMLOW is near *PBRK. COMMON should be moved to higher memory using the COmmon command. If COMMON must be moved above 100000, it may be necessary to recompile or reassemble the load program in 64R mode and the program load must begin with a MO D64R command.

d. None of the above. The user's program requires initialized common. Common is usually defaulted to overwrite the space used by the Loader. Those locations between the bottom of the symbol table and the top of the Loader cannot be initialized as this would destroy the loader. The solution is to use a CO(mmon) command to move COMMON out of the way of the loader. Possibly the user will want to use HILOAD to permit COMMON to use the locations normally used by the Loader.

OR Out of Reach. An attempt has been made to reference a common area that is out of reach of the load mode.

Begin the load with an MO D64R command, or move COMMON to '100000 or lower with the CO command.

NS Never Sectored. Code is being loaded in 16S or 32S mode, which will not properly execute in a sectored mode. Loading is discontinued.

Don't include the MO D16S or MO D32S command in the load session, or check the PMA source module to see if it includes one these commands.

N6 Never 64R mode. Code is being loaded in 64R mode, which will not execute properly. Loading is discontinued.

Recompile or reassemble the source files in 64R mode, or remove a MO D64R command from the load session, or look for a PMA module which has set the load mode to 64R.

SECTION 8

DEBUGGING UTILITIES

OCTAL (TAP) AND SYMBOLIC (PSD)

INTRODUCTION TO PRIME DEBUGGING UTILITIES

Prime supplies two types of debugging programs: TAP (Trace and Patch) and PSD (Prime Symbolic Debugger). Both are used to examine or alter locations in memory-resident binary run files.

TAP is a compact, one-sector, octal-mode routine that examines, dumps, or updates programs from the user terminal. It includes trace and breakpoint insertion features for dynamic debugging under conditions of simulated execution (in sectored addressing modes only.)

PSD is a 4-,5-, or 6-sector (addressable up to 64K) version that performs the same functions as TAP except for EXECUTE and PATCH. In addition, it examines, dumps, or updates memory locations in octal, hexadecimal, alpha-numeric, binary, or symbolic notation. In symbolic form, instructions are disassembled 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 of Ø to 3, as in LDA% 2, which signifies an LDA instruction operating in extended addressing class 2 (stack postincrement). Furthermore there is an option to interpret memory maps and use symbol table names in address expressions.

The selected debugging program resides in memory along with the binary run file to be examined. Care must be taken not to write over part of the run file when TAP or PSD is invoked.

Command Summary

Table 8-1 summarizes all TAP and PSD commands according to their functions. The commands are defined in full detail at the end of this section in alphabetical order.

TABLE 8-1. TAP AND PSD COMMAND SUMMARY

Function	Command
Memory Words:	
Access and print or alter contents	ACCESS
Inserts breakpoint link in the object program	BREAKPOINT
List (print) contents	LIST
Update (alter) contents	UPDATE
Memory Blocks:	
Copy block to block	COPY
Print contents of block (or, in PSD, write to optional file)	DUMP
Fill block with constant	FILL
Search block for contant under mask	SEARCH
Verify block to block	VERIFY
Not-equal search for constant under mask	NOT EQUAL
Executable Programs:	
Breakpoint set	BREAKPOINT
Execute a subroutine	EXECUTE (TAP only)
Jump trace (print diagnostic after JMP, JST, or HLT instructions)	JUMPTRACE
Monitor for effective address (execute program and print diagnostic if location is referenced.)	MONITOR
Patch object program (insert JMP in specified location)	PATCH (TAP Only)
Run object program (print diagnostic if breakpoint is reached)	RUN

TABLE 8-1 (Cont.)

Function	Command
Executable Programs (Cont.)	
Trace object program (print diagnostic at specified intervals)	TRACE
Update memory word	UPDATE
Advanced Features (PSD Only)	
Load symbols from map file and enter symbolic address mode	LS (Load Symbols)
Address mode selection	MODE
Open file for memory dump or symbol tables	OPEN
CPU/PSD Parameter Printout	PARAMETERS
Relocation constant alteration	RELOCATE or X
Quit to PRIMOS	QUIT
Search for effective address under mask	EFFECTIVE
Enable/disable symbolic input/output	SYMBOL
Update CPU status keys	KEYS
Compare contents of one memory block with another	VERIFY

USING TRACE AND PATCH (TAP)

Before starting TAP, the binary program to be debugged must be made memory-resident, through a PRIMOS LOAD or RESTORE command.

Starting TAP

Enter the PRIMOS command TAP. When ready, TAP prints the $ prompt character and waits for command strings from the user terminal. See COMMAND DESCRIPTIONS for further requirements.

Terminating Long Operations

To terminate long operations such as DUMP, type CTRL P for a return to PRIMOS.

Restarting

Restart at 'XXØØØ, where XX is the sector occupied by TAP (to determine this value, RESTORE TAP and do a PM to print the starting location.)

Multiple Copies of 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 '2ØØØ up:

```
    $C 1ØØØ 16777 2ØØØ (CR)
```

USING PRIME SYMBOLIC DEBUG (PSD)

Before starting PSD the binary program to be debugged must be made memory-resident, through a PRIMOS LOAD or RESTORE command.

Starting PSD

PSD is supplied in the CMDNCO UFD of PRIMOSE master disks in three versions. The command PSD starts a version that runs below PRIMOS in a 32K memory. The command PSD20 loads and starts a version that runs below PRIMOS in a 16K memory. The command HPSD loads and starts a version that resides in the upper 32K. PSD is not relocatable.

Enter the appropriate PRIMOS command (PSD, PSD20 or HPSD). When ready, the symbolic debugger prints the $ prompt character and waits for command strings from the user terminal. See COMMAND DESCRIPTIONS for further requirements.

Terminating Long Operations

To terminate long operations such as DUMP, type CTRL P for a return to PRIMOS.

Restarting

Restart at XX000 where XX is the sector occupied by PSD, (to determine this value, RESTORE the version of PSD to be used and do a PM to print the starting location.)

PSD Input/Output Formats

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

```
    A 1ØØØ:O
```

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

:A	ASCII
:B	Binary
:D	Decimal
:H	Hexadecimal
:O	Octal
:S	Symbolic

The effects during input and output are described below.

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

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

The second character is required and must not be:

/ ? , .nl.

ASCII constants may be input in any mode. Use the form 'cc (single quote followed by two characters).

ASCII Output: Two characters are printed - an @ is substituted for any non-printing character. In a DUMP, up to eight character pairs are printed per line.

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

Binary Output: A sequence of sixteen 1's and 0's is printed. In a DUMP, up to four words are printed per line.

Decimal Input: 0 - 9, up to five characters.

Decimal Output: In a DUMP, the addresses are printed in octal; their content in decimal.

Hex Input: Any sequence of characters from the set , 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 suppressed. In a DUMP, up to eight words are printed per line.

Octal Input: Any expression is accepted (octal number of 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 eight words are printed per line.

Symbolic Format - General Features: Symbolic format enables the user to reference instructions using mnemonics rather than octal op codes. In addition, if a load map is properly converted and loaded, the user may use load map symbols in expressions. (See LS command for details.)

The general form for symbolic instruction representation is:

    Mnem[*] [%][<]Expr[,1]

where:

Mnem is any legal instruction mnemonic

*	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

,1 Specifies indexing

<u>Expressions</u>: 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 count.

c. A symbol from a memory map properly converted, loaded, and enabled by the LS command.

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

   Examples:

   *+123 ; *-1 ; *+1000-2 ; ZILCH + 77

Symbolic Input: The following examples show the format for most of the 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, information which specifies 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 for extended instructions to be input properly.

The first two classes are the two-word instruction types, for which the second word is expected 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      Post Increment
LDA*    %2      Same, indirect
LDA     %2,1    Same, indexed
LDA     %3      Predecrement
etc.
```

If a load map has been converted and the LS command given, symbol table labels may be used in place of octal address values, as in:

```
LDA     START

LDA     %0
DAC     START

LDA     START+5
```

Symbolic Output: Symbolic 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.

```
LDA% Ø          Long reach
DAC 1Ø17

LDA*% 1,1       Stack relative, indirect and indexed
DAC 1Ø17

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
   1073 LDA      1073    LDA      170     LDA%        1    HLT
   1077 LDA      77      LDA      100
```

If a load map has been converted and the LS command given, address values are printed using symbol table labels wherever possible, as in:

```
TEMP        LDA START

TEMP+10     JMP START+37

TEMP+30     JMP* START-1
```

Offset values are in octal.

COMMAND DESCRIPTIONS

Following are detailed descriptions of all TAP and PSD commands, in alphabetical order.

Once started from PRIMOS command level, both TAP and PSD print a $ prompt character and wait for keyboard input.

Each TAP or PSD command consists of a one or two function code followed by one or more parameters, separated by spaces or commas. Each command string is entered for execution by a CR (carriage return) terminator.

The ACCESS command differs from the others in that it remains in control and allows the user to examine and/or alter more than one location without returning to command mode (signalled by the prompt character). The next location to be accessed is selected by the termination used. (See ACCESS for details).

For TAP (and for PSD in octal input/output format) all values are right-justified octal integers. If a value is unspecified, it is assumed to be a zero. For example, if the parameters Value-1,,Value-3 are given, the omitted item, Value-2, is assumed to be zero.

A slash (/) or question mark (?) may be used to abort a command string and return to command mode.

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

In TAP, 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 Start-address

Accesses word(s) in memory starting at Start-address. For both PSD and TAP, the program types Start-address and its contents, then waits for keyboard input, in the following form:

TAP: [Value] Terminator

PSD: [Format] [Value] Terminator

Format is an optional format symbol to select one of the PSD input/output formats:

Format Symbol	Input/Output Mode
:A	ASCII
:B	Binary
:D	Decimal
:H	Hexadecimal
:O	Octal (default value)
:S	Symbolic

The new format takes effect immediately. For example, :HAF enters the hex value AF, regardless of the previous mode.

Value, if specified, replaces the contents of the accessed location. In TAP, Value must be in octal; for PSD, value must be in the current input/output formats.

See PSD INPUT/OUTPUT FORMATS for information on the different formats.

Terminator is one or more characters selected from the following list:

TERMINATOR		FUNCTION
TAP	PSD	
CR or ,	CR or ,	Alters contents of current location (if a value is given), moves to current location +1 and prints its contents.
↑	↑	Alters contents of current location (if a value is given), moves to current location -1 and prints its contents.
/ or ?	/ or ?	Exits from access mode
NOT USED	.n (CR)	Moves to current location +n and prints its contents (n is octal)
NOT USED	.-n (CR)	Moves to current location -n and prints its contents (n is octal)

TERMINATOR		FUNCTION
TAP	PSD	
NOT USED	@	For memory reference instructions of the form INST* location only: saves a return address (current location + 1), moves to the effective address location, and prints its contents. Subsequent accesses (terminated by CR, comma, , . or . -) are relative to the effective address. A returns to the return address.
NOT USED	(	Same as @, but saves current location as return address.
NOT USED	\	Returns to the return address saved by the last @.
NOT USED	)	Returns to the return address saved by the last (.
NOT USED	=	For memory reference instructions only: calculates and prints the effective address and its contents. No change is made to the current location or its contents.

Effective Address Formation (PSD Only)

PSD processes input and output in all four Prime-300 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.

Current Location Counter (PSD only)

In Access mode, a current location count is maintained, starting with the value of the Start-address parameter of the ACCESS command. The location count determines the next location to be accessed.

During each access operation, PSD replaces the value [illegible] the open location with the new value (if specified) and uses the line terminator to compute the next value of the current location counter. For the comma or CR line terminators, the counter is incremented after each access. Other line terminators provide different options.

<u>Relocation Constant (PSD Only)</u>

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 > Ø and $A 3121 would open location 3121.

The relocation constant is set by the RELOCATE command. Setting the relocation constant to Ø 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.

<u>B</u>REAKPOINT Location

Inserts a breakpoint link in the restored program at Location. If restored program is later executed, and if control reaches Location, TAP or PSD prints CPU status, then awaits further commands.

CPU status is printed in the following format:

    Start-address (A-register) (B-register) (X-register) (Keys).

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 reuse and requires user action only to change it. To remove the breakpoint completely, key in B 17 (CR).

<u>C</u>OPY Blockstart Blockend Newblock

This command copies memory block at locations Blockstart through Blockend into a new block starting at Newblock. If Blockend does not exceed Blockstart, only the word at location Blockstart is copied. If Newblock lies between Blockstart and Blockend, the block between Newblock + Blockend - Blockstart is reached.

Example:

```
    $C 555Ø4   55577   613Ø2
```

<u>D</u>UMP Blockstart Blockend

Dumps the memory block at locations Blockstart through Blockend to user terminal or (for PSD only) to an external file. 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.

Example:

```
$D  555Ø4  55577
```

In order to DUMP to a file in PSD, a file must be OPENED for writing on unit 2. After the dump, the unit should be closed.

Example:

```
$O  DMPFIL  1  2
$D  1ØØØ  2ØØØ
$O  Ø 1 4
```

EFFECTIVE Blockstart Blockend Address [Mask]

Searches for an effective Address in the block from Blockstart to Blockend under an optional Mask. If no Mask is specified, the entire word is tested. When a match is found, the effective address and its contents are printed at the user terminal. The search continues until location Blockend has been tested.

EXECUTE Subr [A-register] [B-register] [X-register] [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. (See KEYS command for Keys parameter format.) The subroutine return should be via indirect jump through its entry point, incremented by 0, 1, or 2.

FILL Blockstart Blockend Constant

Fills the memory block at locations Blockstart through Blockend with an octal constant. If Blockend does not exceed Blockstart, only the first location is filled.

JUMPTRACE Startadd [A-register] [B-register]

Dynamically traces the object program, starting at location Startadd, with an optional preset of the A and B registers. A diagnostic printout is produced prior to the interpretive execution of any JMP or JST or HLT (see function T for format).

KEYS Value (PSD Only)

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

```
    -------------------------------------------------------------
  | C | P   *   * |  ADR   * | *         |          |          |
    -------------------------------------------------------------
    1   2   3   4   5   6   7   8   9                         16
```

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	16S
0	1	32S
1	1	32R
1	0	64R
1	0	64V (Prime-400 only)

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

LIST Address

List the content of Address.

LS (LOAD SYMBOLS) (PSD Only)

Enables true symbolic address references during ACCESS input and output or DUMP output. In order to use symbols from a load map, three steps are required. First, load the program and specify the load map to be sent to a file. Second, convert the load map file so that PSD can read it, by running CNVTMA. Third, restore the user program, invoke PSD, and request PSD to load the converted file. These steps are given in the following example:

1. Create load map and send it to a file.

```
OK,LISTING LMAP
OK,LOAD
$LOAD B  PROG
$LIB
LC
$LIB UII
LC
$SAVE*PROG
$MAP $F
$QUIT
OK,CLOSE 2
```

2. Convert loader load map to PSD load map.

```
OK,CNVTMA LMAP PMAP
```

3. Restore program and use LS command to enter full symbolic format:

```
OK,RESTOR *PROG
OK,PSD
$O PMAP 1 1          (open PMAP on unit 1 for reading)
$LS                  (load symbols
$O 0 1 4
```

The LS command puts PSD into symbolic mode. All addresses are typed as symbol and offset number or simply symbol if the address matches the value of the symbol exactly.

Once the load map is prepared in this manner, the user can enable or disable symbol interpretation with the SYMBOL Command.

```
MODE   [D16S]                         (PSD Only)
       [D32S]
       [D32R]
       [D64R]
       [D64V]
```

Selects the addressing mode in which address values are computed in symbolic input/output format, sets bits 4, 5 and 6 of the CPU keys accordingly, and resets all other keys bits to zero.

MONITOR Startadd A-register B-register Address

Dynamically monitors the 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 equal to Address. (See TRACE function for printout format.)

NOT-EQUAL Blockstart Blockend Nmatch [Mask]

Searches memory block between Blockstart and Blockend 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.

OPEN Filename Funit Key (PSD Only)

Opens a file to be used as a DUMP output file or symbol table input file. The parameters are the same as the PRIMOS OPEN Command:

Filename	A PRIMOS filename existing (or to be created) in the current UFD.
Funit	PRIMOS file unit (1-16)
Key	Action Key; octal values are: 1. Open for reading 2. Open for writing 3. Open for reading and writing

PATCH Patchloc Branchloc

Simplifies insertion of a patch in the executable program. The instruction at Branchloc is replaced by a jump to Patchloc, the displaced instruction is stored at Patchloc, and the ACCESS function is entered with the current location set to Patchloc. The user can then enter the desired patch, including a suitable return. Patchloc must either be in the same sector as Branchloc or in sector Ø.

PRINT (PSD Only)

Prints CPU/PSD parameters in octal as follows:

```
Breakpoint  Breakpoint  A-register  B-register  X-register  Keys Relcon
            Contents
```

Relcon is the current value of the access mode relocation constant.

QUIT (PSD Only)

Returns to the operating system.

RELOCATE Value (PSD Only)

Sets a new Value for the access-mode relocation counter. (Can also be entered as X Value.)

<u>R</u>UN Startadd [A-register] [B-register] [X-register] [Keys]

Runs the executable program starting at 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.

<u>S</u>EARCH Blockstart Blockend Matchword [Mask]

Searches memory block from Blockstart to Blockend for words equal to Matchword under an optional Mask. (If Mask is not specified, the entire word is tested.) When a match is found, the address and its contents are typed out, and the search continues until location Blockend has been tested.

<u>T</u>RACE Startadd [A-register] [B-register] [| Pval [Ø] |]
| 177777 Interval |

Dynamically traces executable 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-register) (B-register) (X-register) (Keys)

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

When Pval is specified, the printout occurs only when P=Pval. If Pval is followed by Ø, printout occurs the first time P=Pval, and every instruction thereafter.

When 177777 Interval is specified, printout occurs every Interval instructions. If interval is negative, its absolute value is used; if zero, it is treated as 65536.

<u>T, J AND M Function Restrictions</u>

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

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

c. Tracing of input/output routines is possible, but timing should be investigated. Processing speed is reduced by a factor of 6Ø to 8Ø percent when no printout is involved.

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

UPDATE Location Contents

Sets Contents into Location and prints old and new contents of Location.

VERIFY Blockstart Blockend Copy

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

The format of a VERIFY printout is:

```
    Location Block-contents  Copy-contents
```

XREGISTER Value (PSD Only)

Sets X Register to contain Value - for example, before executing a RUN command or doing an effective address calculation.

SECTION 9

SEG (SEGMENTATION UTILITY)

INTRODUCTION

This section describes the use of Rev. 11 SEG. Seg is the Primos IV utility module for loading, modifying, and running segmented programs. A segment is a 64K word block of a user's virtual address space. Segment '4000 is the segment that SEG and other external commands occupy when invoked. SEG creates a run file of up to fifteen segments, starting at segment '4001.

PRIMOS at Rev. 11 assigns memory segments to a user as they are accessed. They are kept until logout. Because Rev. 11 PRIMOS IV makes only 64 segments available for all users, extra segments should not be invoked, unless the user is actually executing or examining a segmented program. Most of the functions of SEG use only one segment; only those options which restore a run file use extra segments, i.e., RESTORE, RESUME, LOAD, and EXECUTE.

SEG permits the user to perform many of the operations on segmented run files which are normally done to Prime 300 run files at command level. Since the run files are different from Prime 300 run files, the operations themselves have some different effects than the Prime 300 operations.

SEGMENTED RUN FILES AND THE SEGMENT LOADER

A segmented run file consists of segment subfiles in a segment directory. For this reason, you should not delete a SEG run file with a PRIMOS IV DELETE command. Instead, use FUTIL's TREDEL.

Segment subfile 0 of the run file is used for startup information, for the load map, and for a map of the memory image subfiles. The memory image subfiles begin in segment subfile 1. Each memory image subfile contains 2048 words. Each 2048-word block of memory in the 15-segment user space is assigned a position in the segment directory. Thirty-two memory subfile are reserved for each memory segment; however, only those required for the run file are actually stored on disk.

SEG has a virtual loader (loads to a file rather than memory) which requires the name of the run file before anything is loaded. The run file may be new or may be a previously used SEG run file and may be in any UFD. An old unsegmented SAVE file may not be used.

As the symbol table is always available, SEG's loader may be used to add modules to an existing run file. Similarly, a partial load may be saved with the SEG SAVE command and the load completed later.

Loading Object Files

The object file of the main program must have been created by PMA using the pseudo-op 'SEG', or by FTN with the 64V code option (B-register bit 8 set) because the entry point will be treated as an ECB (Entry Control Block). The load mode is 64V. The following commands found in the Prime 300 loader are not supported:

- AUTOMATIC - Automatically insert base areas
- COMMON - Set the top of common
- HARDWARE - Define the current hardware
- MODE - Set the Load Mode
- SETB - Define a base area
- VIRTUALBASE - Establish a virtual sector zero base area

To support reentrant procedures, code and data are loaded in separate segments. Data consists of all COMMON blocks and link frames. The loader assigns code and data segments. The first segment ('4001) is used for code. Usually segment '4002 will be used for data. The loader loads data and code into appropriate segments and opens new segments as required.

The Stack

The loader assigns a stack when SAVE is invoked. The stack is usually assigned as the next free location in the first procedure segment with '6000 free words. If no such segment exists, a new data segment will be assigned with the first location in the stack set to 4. The user may force the location of the stack and/or may change its size.

The Load Map

The following is a sample of the format of the SEG load map for a multi-segment run file: (All segment numbers and word numbers are in octal).

```
*START  004002  000001  *STACK  004001  001243  *SYM    016632

SEG. #    TYPE     LOW      HIGH     TOP
004001    PROC##   001000   001242   001242
004002    DATA     000000   000252   000252


ROUTINE        ECB           PROCEDURE       ST. SIZE   LINK FR.
####     4002  000001     4001  001000     000012     177400
TNCU     4002  000063     4001  001027     000020     177463
TNOUA    4002  000107     4001  001041     000020     177507
TONL     4002  000127     4001  001176     000012     177527
T1OU     4002  000151     4001  001204     000016     177551
EXIT     4002  000173     4001  001224     000012     177573
T1IB     4002  000213     4001  001227     000012     177613
T1OB     4002  000233     4001  001235     000012     177613

           4002  000025     NAMED     4002  000051
```

Symbols are divided into two classes: those that relate to ECBs and those that do not. Those that relate to ECBs are listed separately with additional information about the procedures to which they apply. This part of the symbol table is sorted on procedure address. The remainder of the symbols (e.g., FORTRAN COMMON block names) are listed with their addresses.

Addresses are given in two-word form. The high order word is the segment number in octal, the low order word is the location. *START should be the address of the ECB for the main program. If this routine has no name, it is called #### in the map. The address supplied for *STACK is the first available location of the stack. The CMHGH, CMLOW, and PBRK entries of LOAD - style load maps do not appear. Instead, LOW, HIGH and TOP are listed separately for each segment. TOP is the last assigned location in the segment rather than the first tree location. HIGH and TOP will frequently be the same.

The type of each segment is shown. The segment chosen for the stack is identified by the characters ## following the type entry. The base area map includes segment number as well as word addresses.

USING SEG

SEG is a command under CMDNCO which can be invoked as SEG [Filename]. SEG Filename is equivalent to R Filename for a Prime 300 run file. The run file is loaded into segmented memory and execution started. If the user enters SEG only, its other functions may be invoked.

All SEG functions are listed under SEG COMMAND SUMMARY and detailed in the SEG COMMAND, LOAD SUBCOMMAND, and SAVE SUBCOMMAND paragraphs.

SEG displays a # on the terminal as a prompt character. The SAVE and LOAD commands display a $ as a prompt character to solicit subcommands.

SEG COMMAND SUMMARY

Command	Subcommand	Description
HELP		List SEG commands
LOAD		Invoke the LOAD subcommand processor
	ATTACH	Attach to another UFD
	EXECUTE	Execute the loaded program
	FORCE	Force load an object file
	INITIALIZE	Reinitialize the loader and start over
	LIBRARY	Library load
	LOAD	Load an object file
	MAP	Produce a load map

Command	Subcommand	Description
	QUIT	Return to PRIMOS command level
	RETURN	Return to SEG command level
	SAVE	Save the run file
	XPUNGE	Expunge symbols from the Symbol Table
MAP		Generate a load map
PARAMS		Display SAVE parameters
PSD		Starts a simple debug utility for run file examination and patching
QUIT		Return to PRIMOS command level
RESTORE		Restore, but don't execute
RESUME		Restore (if necessary) and execute
SAVE		Invoke the SAVE subcommand processor
	A	Change the initial A Register setting
	B	Change the initial B Register setting
	END	Return to SEG command level
	KEYS	Change the initial Key setting
	NEW	Create a new run file
	START	Change the Entry Control Block (ECB) start address
	WRITE	Write the whole run file to disk
	X	Change the initial X Register setting
TIME		Report date and time of last file save

SEG COMMANDS

HELP

Causes SEG to print a brief list of its commands.

LOAD [*] [Filename]

Invokes the LOAD subcommand processor that permits the user to load segmented programs. In addition, it can be used to complete an incomplete load from a previous load session, or to add routines to a completed load. Carefully used with suitable patching of the original ECB, it can be used to replace a routine in a existing run file.

The optional arguments determine the file to which binary code will be written during a load operation. The choices are:

LOAD Filename	Select Filename in the current UFD. If Filename exists, it is initialized (previously loaded material is deleted).
LOAD * Filename	Selects Filename in the current UFD but does not initialize it. Loading continues using the current state of Filename's load map.
LOAD *	Continues loading to the current load file (specified by the original SEG command) using the current state of its stored load map.
LOAD	SEG responds with: SAVE FILE TREE NAME. Enter the tree name of a file in the current UFD or one of its sub-UFD's. The file is initialized if it already exists.

```
MAP  [ |Runfile| ]  [Mapfile]  [Option]
       |   *   |
```

Generates a load map at the user terminal or in a specified file.

Runfile specifies the run file for which a map is to be generated. If the asterick is specified, the current named run file is mapped.

Mapfile specifies an ASCII file in which the map will be written. If Mapfile is omitted, the map is displayed at the user terminal.

The Option parameter determines the extent of the map:

Option	Map extent
Omitted	Full map
1	Extent map only
2	Extent map plus base areas
3	Undefined symbols only

PARAMS [Filename]

Displays six parameters from the last SAVE operation of Filename or, by default, of the current named run file as:

START(2), STACK (2), A, B, X, KEYS

START(2) is a two-word starting address (address of the ECB of the main program).

STACK(2) is a two-word address of the stack.

A, B, X are the contents of the specified registers

KEYS are the CPU keys.

PSD

Activates a simplified debugging utility similar to PSD to examine or patch the current run file. The following subcommands are provided:

S	Segment	Select a Segment number
L	Location	Look at a location
U	Value	Update the location to a new octal value
D	Start End	Dump locations from Start to End.

All input and output values are octal.

QUIT

Closes all files opened by SEG and returns to PRIMOS command level.

RESTORE [Filename]

Restores a run file, Filename, or by default, the current named run file, for examination or patching with PSD. A patched file may be executed later without SEG attempting to restore the file, so long as no new file name has been entered since the RESTORE Command.

RESUME [Filename]

Executes a run file Filename or by default, the current named run file. Also performs a restore if a RESTORE command was not previously executed. RESUME may follow any combination of SEG commands.

SAVE [Filename]

Invokes the SAVE subcommand processor to operate on run file Filename, or by default on the current named run file. The $ prompt is used to request SAVE subcommands that can assign a stack, change a save vector, patch a run file, and write from memory all or parts of a run file. These subcommands are detailed in the SAVE SUBCOMMANDS paragraph.

TIME [Filename]

Displays on the terminal the time and date the run file, Filename, was last saved or, by default, when the current named run file was last saved.

LOAD SUBCOMMANDS

When the user enters a LOAD command, the prompt $ is used to request LOAD subcommands. These subcommands are:

ATTACH [Ufd] [Password] [Ldisk] [Key]

Attaches to different UFD's. This subcommand is converted into a CALL to the PRIMOS subroutine ATTACH and has the same values.

Ufd	Any user file directory. The default is the home UFD.
Password	Necessary only when UFD is password-protected. An owner password provides owner status; a non-owner password provides non-owner status.
Ldisk	The logical disk on which the MFD is to be searched for the Ufd. '100000 specifies a search of all started devices in logical unit order. '177777 specifies search the MFD of the Ldisk on which the current UFD resides. If Ldisk is omitted, logical unit 0 is searched.
Key	Describes how to attach to the Ufd: 0 = Attach to Ufd in MFD on Ldisk. 1 = Attach to Ufd in MFD on Ldisk and set home UFD to current UFD after attaching. 2 = Attach to sub-Ufd in current UFD. 3 = Attach to sub-Ufd in current UFD and set home UFD to current UFD after attaching.

EXECUTE

Runs a loaded program and exits to PRIMOS command level. The SAVE command must be invoked first, because EXECUTE does not invoke the SAVE command.

FORCE Filename Address

Loads a file regardless of any factors which would usually cause the loader to bypass the module.

Filename is the name of the object file to be loaded, and Address is the address at which loading is to start.

INITIALIZE [| Filename |]
              | ! |

Causes the loader to reinitialize itself. This subcommand may be used to abort a bad load (starting over with either the same file or a new one) or begin a new load after a SA(ve).

Filename	Open run file Filename
!	Ask for a new tree name
Null	Initialize current run file

LIBRARY [Filename] [Address]

Loads the named file from the UFD LIB and reattaches the user to the home UFD. If no file is specified, the library file PRIMOS 4 FORTRAN Subroutine Library (See MAN 1880) is assumed.

Filename is the name of the object file to be loaded, and Address is the address at which the module is to be loaded (Ø implies the current load point).

LOAD Filename [Address]

Loads the named file. Filename is the name of the object file to be loaded, and Address is the address at which loading is to start. If Address is 0 or omitted, loading starts at the current load point. If the Address is specified, the user is responsible to determine whether the module will fit in the segment.

MAP [| Mapfile |] [Option]
       | * |

Writes a map of the current run file to Mapfile (if specified) or to the currently open map file (if * is specified). The map file is left open, so that any number of maps can be written to the file if * is specified each time. If, prior to invoking SEG, the user OPENs a file on unit 13, that file will be used as the map file when * is specified.

If Mapfile or * is not specified, the map is listed at the user terminal.

The option parameter determines the contents of the map:

Option	Contents of Map
1	Extent map only
2	Extent plus base areas
3	Undefined symbols only
Omitted	Full map

Note that only one map file may be open per Load session, and that QUIT, EXECUTE and RETURN truncate any open map file.

QUIT

Exits from the loader and return to PRIMOS command level. If the user intends to run the loaded module, the SAVE command must be invoked first because QUIT does not SAVE the module.

RETURN

Exits from the loader and returns to SEG command level. If the user intends to run the loaded module, the SAVE command must be invoked first because RETURN does not SAVE the module.

SAVE

Finishes writing all buffers to the run file and sets the stack into the first available segment, if the user has not specified the stack with the ST command.

XP [Dsymbols Dbase]

Allows the user to remove some or all of the defined symbols from the symbol table. Undefined symbols may not be removed.

Dsymbols controls the deletion of symbols and Dbase controls the deletion of base areas:

```
Dsymbols = 0 - Delete only entry points, leaving common areas
         = 1 - Delete all symbols - including entry points

Dbase    = 0 - Retain all base area information
         = 1 - Retain only sector zero information
         = 2 - Delete all base area information
```

SAVE SUBCOMMANDS

SAVE Subcommands are invoked when SA is typed at the user terminal.

SAVE functions include: assign a Stack enabling the user to change the Save Vector, patch the run file, and write out from memory all or part of the run file. When the user is in the SAVE subprocessor, the prompt $ is used. The SAVE subcommands are:

A Value

Changes the contents of the A Register. Value is the new value, in octal.

B Value

Changes the contents of the B Register. Value is the new value, in octal.

END

Returns to SEG Command Level.

KEYS Value

Sets the CPU status keys to the octal Value. The bit assignments are:

```
 -------------------------------------------------------------
| C | P    * |   ADR      * | *        |          |           |
 -------------------------------------------------------------
  1   2   3   4   5   6   7   8   9                          16
```

where:

C = State of C (Carry) bit

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

* = Must be zero

ADR = Addressing Mode:

Bit 4	Bit 5	Bit 6	Mode
Ø	Ø	Ø	16S
Ø	Ø	1	32S
Ø	1	1	32R
Ø	1	Ø	64R
1	1	Ø	64V

NEW Filename

Creates a new run file with the name Filename. If necessary, NEW copies the current run file and writes out a new run file under the name Filename. It may be used to save a patched version of a run file already in existence.

START Segment Address

Sets a new ECB address for the start of execution. Segment is the new segment and Address is the ECB location.

WRITE

Copies the entire run file to disk without changing any previously declared segment ranges.

WRITE ensures that all patches on the RUN file have made it to the disk. This is useful when there are many of them and no segment ranges have been changed.

XREGISTER Value

Sets the CPU X register. Value is the new octal value.

INDEX