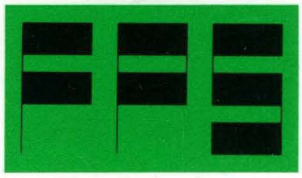


FLOATING POINT
SYSTEMS, INC.

Program Development Software Manual

860-7292-002

by FPS Technical Publications Staff

Program Development Software Manual

860-7292-002

Publication No. 860-7292-002
September, 1978

NOTICE

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

Floating Point Systems, Inc. assumes no responsibility for any errors which may appear in this publication.

Printed in USA

CONTENTS

ILLUSTRATIONS

TABLES

CHAPTER 1

APAL

1.1 INTRODUCTION

Array Processor Assembly Language (APAL) is an unconventional assembly language and poses certain difficulties to programmers. In particular, the capability of the AP to process several instructions simultaneously needs attention.

APAL code is compiled on the host system for execution on the array processor. APAL requires about 24K of memory on typical 16-bit mini-computers.

1.2 BASIC SYNTAX

This section presents the grammar of APAL.

1.2.1 CHARACTER SET

APAL recognizes the following characters:

alphabetic	A through Z
numeric	0 through 9

APAL recognizes also the special characters in Table 1-1.

SPECIAL	FUNCTION
+	integer addition operator; unary addition operator
-	integer subtraction operator; unary subtraction operator
*	integer multiplication operator
/	integer division operator
.	decimal point; current location
$	first character of pseudo-op names
space	symbol terminator
tab	symbol terminator
=	$EQU pseudo-op; DB = op-code; arithmetic identity
(	preceeds a data pad index expression
)	terminates a data pad index expression
<	used with DPX, DPY, and MI op-codes; arithmetic less than
;	op-code terminator
,	operand separator
:	label terminator
"	comment start indicator (carriage return terminates)
#	s-pad no-load indicator
&	s-pad bit-reverse indicator
!	first character of predefined symbols
%	logical OR operator
'	logical complement
>	arithmetic greater than
?	no system function
~	no system function

0493

Table 1-1 Special Characters

1.2.2 FILE NAMES

File names may contain 30 characters, including special characters and numbers. On systems where programmed file assignment is not allowed or is very difficult, the user must enter the number of the logical unit of a file he assigned prior to calling APAL.

A special symbol (which is different for each host system) exists for referencing the user terminal (for example: TT: for the PDP11).

Examples:

```
RUNNER
RUNNER.OBJ
P38
CHANNEL
```

1.2.3 SYMBOL NAMES

Symbol names may be of any length; however, only the first six characters of a name are significant. The first character of a name must be alphabetic. The subsequent characters may be either alphabetic or numeric.

Examples:

```
LOOP
A6
STARTHERE
```

A symbol can be created and given a value by:

- defining it with the $EQU pseudo-op
- using as a label
- declaring it an external with the $EXT pseudo-op

1.2.4 TABLE MEMORY SYMBOLS

A symbol with a value preset to the address of each of the constants in table memory ROM is predefined in APAL. These symbols all start with the character ! to avoid conflict with any user-defined symbol. They may be used in expressions in the same manner as ordinary symbols.

A complete list of these symbols can be found in section 6.14. For example, the following fetches PI from table memory and adds it to a number in DPX(2).

```
LDTMA; DB=!PI          "Fetch PI from TM
NOP                    "Wait
FADD TM, DPX(2)        "Add PI to DPX(2)
```

1.2.5 INTEGERS

Integers can be written in four radixes: octal, binary, decimal, or hexadecimal. In each radix, an integer can be either signed or unsigned. The radix of a number is established by a radix identifying character which is written immediately after the number. Octal integers are denoted by a K, decimal by a ., hexadecimal by an X, and binary by a T. The first digit of a hexadecimal integer must be a decimal digit. The default radix, if a radix identifier is not user, is octal unless otherwise specified by a $RADIX pseudo-op.

Integers are stores as 16-bit two's complement numbers. Integers larger than 16 bits are truncated to 16 bits. Negative integers larger than 16 bits are truncated before they are negated.

Examples:

```
octal integers:          177777
                         -40727K
                         -10

decimal integers:        32767.
                         -1000.
                         +10.

hexadecimal integers:    0ABCDX
                         123FX
                         0CX

binary integers:         101101T
                         -1101T
```

1.2.6 EXPRESSIONS

Expressions are symbolic representations of numbers. They are made of operands and operators.

1.2.6.1 Operands

Operands are symbol names, numbers or the location counter (denoted by .).

Examples:

```
TBLADR
598X
.
33K
```

1.2.6.2 Operators

Operators are of two types: unary and binary.

Unary Operators

'	logical complement
+	positive reminder (+3K,+10.)
-	gives negative of a number (-15X,-777)

Binary Operators

Standard Arithmetic operators

+	addition
-	subtraction
*	multiplication
/	division

standard arithmetic relations, which return a value of one if the relation is true, and zero if the relation is false. For example, B $EQU 6<10 sets B to 1

<	less than
=	equals
>	greater than

Some expressions are:

```
TBLADR+3F
. + 9.
LOOP + 6 * A
(34 - 10X) * 2
```

Expressions are evaluated left to right in 16-bit two's complement arithmetic according to FORTRAN precedence standards and parenthesis may be used liberally.

NOTE

Only the low order 16 bits are used if an expression results in a value larger than 65535.

1.3 SOURCE PROGRAM STATEMENTS

APAL source statements may be divided into three categories:

- comment statements
- instruction statements
- pseudo-op statements

Comment statements allow program documentation. Instruction statements make up the actual symbolic machine code. Pseudo-ops provide directives to APAL during the assembly process.

APAL statements are free format; spaces and tabs may be used as desired to improve legibility.

1.3.1 COMMENT STATEMENTS

Everything on a line following a quote mark (") is treated as a comment by APAL. A line which contains only comments, or a line that is completely blank, is a comment statement and is ignored during the assembly process. A carriage return terminates a comment.

1.3.2 INSTRUCTION STATEMENTS

An APAL assembly language instruction statement has the format:

```
label:   Op-code fields          "Comments
```

The label and comments are optional. The assembler processes the op-code fields and generates one 64-bit instruction word for each instruction statement.

1.3.2.1 Label Field

A label is a user-defined symbol which is assigned the value of the current location counter and entered into the user symbol table. A label is a symbolic means of referring to a specific location within a program. If present, a label always occurs first in an instruction statement and must be terminated by a colon. For example, assume the following instruction statement is entered.

```
LOOP:  FADD DPX, DPY             "LOOP HERE
```

If the current location is 76, value of 76 is assigned to symbol LOOP.

1.3.2.2 Op-code Field (Operation Code Field)

The op-code field follows the label field in an instruction statement and contains one or more AP op-code mnemonics. Individual op-codes in an instruction are separated by a semi-colon. For example, the following two groups of op-codes are equivalent. The absence of a semi-colon following an op-code terminates the instruction.

```
LOOP:   FADD DPX, DPY; FMUL TM, MD; BFGT DONE
```

or

```
LOOP:   FADD DPX, DPY;
        FMUL TM, MD;
        BFGT DONE
```

Each is one instruction statement, which assembles into one 64-bit instruction word. Thus, one instruction statement may be continued over as many lines as desired to achieve a readable program document. The absence of semi-colon after the last op-code signals the assembler that the instruction is ended.

Op-codes may be written in any order within an instruction. The assembler flags any conflicting op-codes with an error message.

Some op-codes require operands as arguments. The operand is separated from the op-code by a space or tab and from another operand by a comma. Some example op-codes are:

```
no operands:        HALT; RETURN
one operand:        FABS MD; BFGT LOOP
two operands:       FADD DPX, DPY; FMUL TM, MD
```

If an operand is missing or improper, the assembler generates an appropriate error message.

A list of all AP op-codes is contained in Section 1.6.

1.3.2.3 Comment Field

The remainder of any line following a quote mark (") is treated as a comment by the assembler and is ignored. The comment field is terminated by a carriage return. Thus, the previous example can be written:

```
LOOP:    FADD DPS, DPY;      "DO AN ADD
         FMUL TM, MD;        "AND A MULTIPLY
         BFGT DONE           "AND A BRANCH
                             "ALL IN ONE INSTRUCTION
```

An instruction is ended by the absence of a semi-colon following the last op-code.

1.3.3 PSEUDO-OPERATION STATEMENTS

Pseudo-operations are directives to the assembler which control certain aspects of the assembly translation process. Each pseudo-op must appear on a separate line in the source text. All pseudo-op names start with a "$". As with instruction statements, pseudo-op statements may be labeled and have comments.

1.3.3.1 $EQU

This operator equates a symbol with an expression. If user defined symbols are used in the expression, they must have been previously defined in the program.

Examples:

```
A $EQU 321
LOOP $EQU LOC + 3
HERE $EQU . - 3
MASK $EQU 132*3+6
```

Alternatively, the character "=" may be used in place of "$EQU":

Examples:

```
A = 6
X = A*3
```

1.3.3.2 $LOC

$LOC sets the current location counter to the value of an expression. If symbols are used in the expression they must have been previously defined in the program.

Examples:

```
$LOC    300
$LOC    . + 6    "LEAVE NEXT SIX UNUSED
$LOC    LOOP + 10
```

NOTE

$LOC should not be set to an absolute address, as in the first example, if the assembly output is to be linked relocatably with other programs.

1.3.3.3 $END

$END causes APAL to terminate the assembly.

1.3.3.4 $VAL

This operator defines 64 bits of data to fill one program word. The data must consist of four 16-bit integers or integer expressions which represent the four 16-bit quarters of a program word. The four expressions are separated by commas.

Examples:

```
$VAL    -377, 104763, 10., LOOP + 6
$VAL    0, 0, 2000, 33
```

1.3.3.5 $FP

This operator fills the right-most 38 bits of a program word with a specified floating point number. The left-most 26 bits of the word are cleared.

Examples:

```
$FP   6.0023E23
$FP   2
$FP   E-17
PI:     $FP   3.141592653  "PI
```

A floating point number (for example, a constant for an algorithm) can be read out of program source memory and onto the data pad bus using a RPSF op-code. As an example, to load the contents of location PI into data pad X:

```
RPSF PI; DPX<DB      "GET PI INTO DPX
```

1.3.3.6 $TITLE

This pseudo-op names a program. The name need not be unique among the other symbols in the program. The $TITLE pseudo-op must occur as the first statement in a program.

Examples:

```
$TITLE FFT
$TITLE DIVIDE
```

1.3.3.7 $ENTRY

This pseudo-op declares a symbol to be global, that is, a symbol which is defined in this program and may be referenced by other separately assembled programs. The identified symbols must be defined in this program either by an $EQU pseudo-op or by their use as a label. $ENTRY pseudo-ops must occur before any instruction statements in the program.

If a symbol is to be an entry point for host computer FORTRAN calls, it must be the first entry symbol defined. Following the symbol name must be the number of s-pad parameters expected in the call. This may be a number from $0-17_8$, and is separated from the symbol name by a comma.

Examples:

```
$ENTRY A
$ENTRY B,6    "Expect 6 s-pad parameters
$ENTRY C,0    "Expect 0 s-pad parameters
```

1.3.3.8 $EXT

This pseudo-op declares global symbols which are referenced by this program, but are defined by another separately assembled program. $EXT pseudo-ops must occur in the program before any instruction statements. Symbol names are separated by commas.

Examples:

```
$EXT FLOAT, SCALE, FFT
$EXT DIVIDE
```

1.3.3.9 $INSERT

This pseudo-op causes source to be read from the designated file. The line number is reset. When end-of-file is encountered source is again read from the file originally specified in the APAL call. The line count is set to its original value when the end of the $INSERT file is reached. Also, when the $INSERT file is reached during Pass 1 of assembly the line containing the $INSERT is written to the terminal. When its end is reached, the message "END $INSERT" is written in the listing. (This happens during Pass 2, also.)

Example:

```
    $INSERT   FILEA
```

1.3.3.10 $RADIX

$RADIX changes the default number radix to the value of the expression. It is entered and evaluated in base 10. It must be either 8, 10, or 16.

Example:

```
    $RADIX 10
    $RADIX 16
```

1.3.3.11 $IF ...$ENDIF

This allows conditional assembly. If the expression which follows $IF evaluates to zero, any subsequent source lines up to $ENDIF are not assembled. They do, however, appear on the listing.

Examples:

```
    $IF   PROG
    PROG $EQU 0
    $ENDIF
```

1.3.3.12 $PAGE

$PAGE begins a new page on the listing.

1.3.3.13 $BOX...$ENDBOX

All source lines found between $BOX and $ENDBOX statements are considered comments and are surrounded by a box of asterisks when a listing is produced. They may be used to improve readability of the listing.

1.3.3.14 $LIST and $NOLIST

Source occurring after a $NOLIST does not appear on the listing file. The presence of a $LIST terminates this condition. Source occurring after a $LIST does appear on the listing file. If no listing was specified in the call to APAL, neither of these has any effect.

1.3.3.15 $LIB...$ENDLIB

These two pseudo-ops cause loader library blocks and end library blocks to be written to the object file. APLINK treats an object module preceded by a library block as a library, and loads only those routines that satisfy unsatisfied externals.

1.3.3.16 Dummy FMUL and FADD Pushers

When programming pipelines as desribed in Part One of the Programmer's Reference Manual, it is convenient for readability to include in the code all the FMULs and FADDs that are used as pushers in any of the columns of the handwritten pipelines. These are coded without parentheses. Any FMUL or FADD without arguments does not conflict with other arithmetic arguments of like type and is completely ignored unless it is the only op-code of its type.

Example:

```
FADD DPX1, DPY1; FMUL FM,FA; FADD
```

In this example the last FADD is ignored.

1.3.4 ORDER OF PROGRAM STATEMENTS

There is a definite ordering of statement types within a program which must be followed. $TITLE pseudo-op must appear first. Any $ENTRY and $EXT pseudo-ops must follow. Then the program body, that is, the code, occurs. Finally the $END pseudo-op occurs. Statement order is as follows:

```
$TITLE              pseudo-op
$ENTRY              pseudo-op(s)
$EXT                pseudo-op(s)
"program, etc"
  .
  .
  .
$END                pseudo-op
```

1.3.5 SAMPLE APAL PROGRAM

Figure 1-1 illustrates a sample APAL program.

```
          $TITLE PROG1
          $ENTRY PROG1,3
          $EXT DIV

"PROG1 DIVIDES TWO SCALARS IN MAIN DATA
"AND RETURNS THE ANSWER TO MAIN DATA
"MEMORY.  C=A/B.
"
"S-PAD PARAMETER DEFINITIONS:
"

          A $EQU  0       "ADDRESS OF A IN MAIN DATA MEMORY
          B $EQU  1       "ADDRESS OF B IN MAIN DATA MEMORY
          C $EQU  2       "ADDRESS OF C IN MAIN DATA MEMORY
"
PROG1:    MOV A,A: SETMA            "FETCH A
          MOV B,B: SETMA            "FETCH B
          NOP                       "WAIT
          DPY< MD                   "STORE A IN DPY
          DPX< MD:                  "STORE B IN DPX
              JSR DIV               "AND DIVIDE A BY B
          MOV C,C; SETMA; MI<DPX;       "STORE ANSWER IN C
            RETURN                      "AND RETURN
          $END
```

Figure 1-1 Sample APAL Program

1.4 OPERATING PROCEDURES

This section describes the operation of APAL.

1.4.1 USING APAL

APAL assembles a file of source code into a relocatable object file. Optionally an assembly listing is produced.

APAL first requests the names of the three files to be used for source, oject, and listing and errors respectively. The program requests the name of the source file by outputting to the user console:

```
SOURCE FILE=
```

The user responds by entering the desired program file name. APAL then requests the name of the file to receive the relocatable object module by outputting:

```
OBJECT FILE=
```

The user responds by entering the desired object file name. APAL then requests the name of the file to receive the assembly listing by outputting:

```
LISTING AND ERROR FILE=
```

The user replies by entering the name of the desired listing file. If APAL cannot find or assign the requested file it outputs the message "FILE NOT FOUND OR UNAVAILABLE" and repeats its request.

APAL then outputs:

```
LISTING? (Y/N)
```

A response of Y(cr) yields a full assembly listing, symbol table, and any error messages. An N(cr) suppresses the assembly and symbol table listings and writes any error messages to the listing file.

Finally, if a listing was requested, APAL outputs:

```
LISTING RADIX?  (8,10,16)
```

A response of "8" causes the assembly listing to be done in octal; a "10" specifies decimal; and a "16", hexadecimal.

APAL responds to invalid input with ??? and repeats the request.

The following is an example of a dialogue with APAL. The user desires to assemble an AP program on file FFT.AP and write the object output into file FFT.RB. The listing is placed on file FFT.LS. Of course, the precise details of how files and devices are named depends on the particular operating system being used. The messages printed by the computer are underlined for clarity; the (cr) indicates a carriage return.

```
APAL
SOURCE FILE =
FFT.AP(cr)
OBJECT FILE =
FFT.RB(cr)
LISTING FILE =
FFT.LS
LISTING?
Y(cr)
LISTING RADIX?
8(cr)
```

1.4.2 EXECUTION

If a fatal error occurs, the message "RUN ABORTED" is displayed at the terminal and control is returned to the operating system.

1.4.3 LISTING FILE FORMAT

APAL is a two-pass assembler. When APAL is called, it outputs:

```
APAL
(version)
```

(version) is the version number of the assembler being used. Any errors detected during pass 1 are output next. The assembly listing (if requested) follows and is interspersed with pass 2 error messages. The listing contains the following information for each program statement:

first column	second column	third column	fourth column
source code line number	program source address (location counter)	assembled program	source statement

For program instruction statements, the assembled data is presented as four numbers representing bits 0-15, 16-31, 32-47 and 48-63 of each program source word.

At the end of pass two, APAL outputs

(num) ERROR(S) FOR (title)

(num) is the number of errors detected and (title) is specified by the $TITLE pseudo-op in the last routine assembled. Finally, APAL outputs:

SYMBOL NAME

Followed by the symbol table:

first column	second column	third column
symbol name	symbol value	symbol type blank - local symbol EXT - external symbol ENT - entry symbol

In all of the above occurrences where a number (location, data value, etc.) is printed on the listing, the radix is either octal, decimal or hexadecimal, as specified by the user during the initial dialogue.

1.4.4 SAMPLE APAL LISTING

Figure 1-2 contains a sample APAL listing.

```
APAL REV 2.1  PROG1                                    03/13/78    11:49
  PAGE 0001

PASS1
PASS2
00001                              $TITLE PROG1
00002                              $ENTRY PROG1,3
00003                              $EXT DIV
00004
00005                    "PROG1 DIVIDES TWO SCALARS IN MAIN DATA
00006                    "AND RETURNS THE ANSWER TO MAIN DATA
00007                    "MEMORY.  C=A/B.
00008                    "
00009                    "S-PAD PARAMETER DEFINITIONS:
00010                    "
00011          000000         A $EQU 0   "ADDRESS OF A IN MAIN DATA MEMORY

00012          000001         B $EQU 1   "ADDRESS OF B IN MAIN DATA MEMORY

00013          000002         C $EQU 2   "ADDRESS OF C IN MAIN DATA MEMORY

00014                    "
00015  000000  040000   "PROG1:  MOV A,A; SETMA          "FETCH A
                000000
                000000
                000060

00016  000001  040104            MOV B,B; SETMA          "FETCH B
                000000
                000000
                000060

00017  000002  000000            NOP                     "WAIT
                000000
                000000
                000000

00018  000003  000000            DPY<MD                  "STORE A IN DPY
                000000
                015000
                1000000
```

Figure 1-2 Sample APAL Listing

```
00019   000004   011014          DPX<MD;                 "STORE B IN DPX
00020            000000              JSR DIV             "AND DIVIDE A BY B
                 045004
                 177777

00021   000005   040210          MOV C,C; SETMA; MI<DPX "STORE ANSWER IN C
                 000000
                 003400
                 003600

00022   000006   000000          RETURN                  "AND RETURN
                 000340
                 000000
                 000000

00023                            $END
0000 ERROR (S) FOR PROG1

      SYMBOL     VALUE

   DIV          000000 EXT
   A            000000
   B            000001
   C            000002
   PROG1        000000 ENT
```

Figure 1-2 Sample APAL Listing (cont.)

1.5 ERROR MESSAGES

APAL error messages are printed in the listing following the illegal statement.

There are five basic error classes, which are listed in the following along with the action taken by the assembler:

O - Out of range: an illegal numeric value was truncated to the proper range

C - Conflicting definitions: the first definition was used

M - Missing (or improper) argument: a value of zero was used

B - Bad syntax: the bad op-code field or pseudo-op was ignored

W - Warning of improper usage

The actual diagnostic takes the following form:

*** c msg nn ON LINE nnnnn

where c is the error class, msg is the error message, nn is the error number, and nnnnn is the number of the erroneous line. The assembler error messages, along with an explanation as to the possible causes and/or cures, are given in Table 1-2.

Table 1-2 Error Messages

Error No.	Category	Message	Explanation
1	W	LINE BUFFER OVERFLOW	An instruction statement was too long (600 characters maximum) for the listing buffer.
2	C	MULTIPLY DEFINED SYMBOL	A symbol may be defined only once in a program.
3	C	CONFLICTING OP-CODES	Two op-codes were used in an instruction statement which used the same instruction word bit fields.
4	W	S-PAD ADDRESS TRUNCATED	An s-pad address was outside the legal range of 0-15 and was truncated to 4 bits.
5	O	BRANCH ADDRESS OUT OF RANGE	A branch address was more than 16 locations lower or 15 locations higher than the current location.
6	C	CONFLICTING BRANCH ADDRESSES	Only one branch address may be used in any given instruction statement.
8	C	CONFLICTING DATA PAD INDEXES	Only one value may be given to each data pad index (XR, XW, YR, YW) per instruction statement.
9	M	BAD OR MISSING EXPRESSION	The assembler could not process an expression.
10	M	BAD OR MISSING FADD ARGUMENT	A floating adder op-code had an invalid A1 or A2 operand.
11	M	A WRONG FMUL ARGUMENT	A FMUL op-code had an invalid M1 or M2 operand.
13	C	VALUE FIELD CONFLICT	Only one op-code which uses a 16-bit VALUE field operand may be used per instruction statement.
15	B	UNDEFINED OP-CODE	An op-code name was not a legal AP instruction.
16	M	EXTERNAL SYMBOL IN EXPRESSION	An external symbol may not be used to form an expression.
17	M	UNDEFINED USER SYMBOL	A user symbol was referenced which was not defined.
20	B	UNRECOGNIZED STATEMENT	A statement line was neither a comment, instruction, or pseudo-op statement.

0494

Table 1-2 Error Messages (cont.)

Error No.	Category	Message	Explanation
22	M	EXTERNAL SYMBOL NOT ALLOWED	An external symbol may not be used as an argument for this op-code.
23	W	MISSING $END	A program must terminate with a $END pseudo-op.
24	O	DATA PAD INDEX OUT OF RANGE	A data pad Index must be between -4 and +3 inclusive.
31	M	BAD FLOATING POINT CONSTANT	A floating-point number was unacceptable to the assembler.
32	W	ILLEGAL PSEUDO-OP POSITION	If used, A $TITLE pseudo-op must appear first in a program, followed by any $EXT or $ENTRY pseudo-ops.
34	W	UNREFERENCED $EXT SYMBOL	A declared external symbol was never used in the program. The symbol appears in the symbol table labeled EXT with the value of 177777 (octal).
36	C	DATA PAD BUS CONFLICT	Only one data source may be enabled onto the data pad bus per instruction statement.
37	M	MISSING S-PAD ARGUMENT	An s-pad op-code was missing its s-pad register address.
39	C	XW/YW CONFLICT	If the value field is used in an instruction, an op-code which writes into data pad Y (such as DPY(2)<FM) may be used also only if • no write into data pad X is done, or • the indexes are the same for the writes into both DPX and DPY. examples: legal: JSR SQRT; DPY(2)<FM — uses the value field and a store into DPY. legal: JSR SQRT; DPX(2)<FA DPY(2)<FM — Uses the value field and both data pad write indexes are the same. Illegal: JSR SQRT; DPX(-1)<FA; DPY(2)<FM — Uses the value field and the two data pad write indexes are different.

0495

Table 1-2 Error Messages (cont.)

Error No.	Category	Message	Explanation
43		READ ERROR	There was a file I/O error.
44	O	SYMBOL TABLE OVERFLOW	Too many user symbols.
45	B	BAD OR MISSING SYMBOL STRING	A symbol was missing or illegal.
46	O	EXPRESSION STACK OVERFLOW	Too many parenthesis in an expression.
47	B	BAD $ENTRY	Incorrect $entry statement or the $entry symbol was also found in a $EXT.
48	B	BAD $VAL	Incorrect $VAL statement.
49	W	BAD $TITLE	Incorrect cyntax in a $TITLE statement.
50	W	EXTRANEOUS BROUHAHA	Extraneous characters were found with an opcode.
51	W	BAD OR MISSING DELIMITER DPX-(3)	Incorrect punctuation.
52	M	BAD OR MISSING DATA PAD (BUS) ARG	A data pad argument is missing or incorrect.
53	B	UNRECOGNIZED PSEUDO-OP	An illegal pseudo-op was encountered.
54		FILE NOT FOUND OR NOT AVAILABLE	The specified file was not found or is not available
55	B	NESTED PSUEDO-OP NOT ALLOWED	The specified pseudo"op cannot be nested.
56	W	$ENDBOX WITHOUT $BOX	A $BOX must occur before an $ENDBOX.
57	W	DIVISION BY ZERO	Result is 65,535.
59	W	MULTIPLE LABELS	Attempt to put more than one label on a line.

0496

1.6 AP SYMBOLIC CODES

The various AP op-codes may be divided into 13 groups. One op-code from each group may be used in any given instruction statement, unless otherwise stated.

The following conventions are used throughout this chapter:

{ } Indicates optional operands or mnemonics. The item enclosed in the brackets (for example, {#}) may or may not be entered, depending upon whether or not the associated option is desired.

Under the headings "Function" and "Meaning," upper case characters are used to indicate the origin of the mnemonic code names.

The list of abbreviations contained in Table 1-3 are used to facilitate the op-code descriptions. They are explained later when the op-code group first appears.

Table 1-3 Op-code Abbreviations

Abbreviation	Meaning	Section in which described
sh	s-pad shift	B-1
#	s-pad no-load	B-1
sps	s-pad source register	B-1
spd	s-pad destination register	B-1
&	bit reverse	B-1
disp	branch displacement	B-5
a1	floating adder argument #1	B-6
a2	floating adder argument #2	B-6
idx	data pad index	B-6
m1	floating multiplier argument #1	B-7
m2	floating multiplier argument #2	B-7
dbe	data pad bus enable	B-8
adr	address or value	B-8

0497

1.6.1 S-PAD OP-CODE GROUP

Purpose: s-pad integer arithmetic

Double Operand Op-codes	Function
ADD{sh}{#}{&}sps,spd	ADD sps to spd
SUB{sh}{#}{&}sps,spd	SUBtract sps from spd
MOV{sh}{#}{&}sps,spd	MOVe sps tp spd
AND{sh}{#}{&}sps,spd	AND sps tp spd
OR{sh}{#}{&}sps,spd	OR sps to spd
EQV{sh}{#}{&}sps,spd	EQuiValence sps to spd

Single Operand Op-codes	Function
CLR{sh}{#} spd	Clear spd
INC{sh}{#} spd	INCrement spd
DEC{sh}{#} spd	DECrement spd
COM{sh}{#} spd	COMplement spd

The result of the above op-codes is SPFN (s-pad function).

Miscellaneous Op-Codes	Function
LDSPNL	LoaD Spd from PaNeL bus
LDSPE	LoaD SPd from data pad bus Exponent
LDSPI	LoaD SPd from data pad bus Integer (low 16-bit)
LDSPT	LoaD SPd from data pad bus Table look-up bits
WRTEXP	enable WRiTe of EXPonent only into DPX, DPY or MI
WRTHMN	enable WRiTe of High MaNtissa only into DPX, DPY or MI
WRTLMN	enable WRiTe of Low MaNtissa only into DPX, DPY or MI

Abbreviations:

Name	Meaning

sh — s-pad shift:

Choices	Meaning
(omitted)	no shift
L	shift SPFN left once
R	shift SPFN right once
RR	shift SPFN right twice

\# — s-pad no-load: if present, do not load SPFN into spd (s-pad destination register). If specified, a branch group op-code may not be used in the same instruction statement.

sps — s-pad source register: a name, number, or expression specifying a register number between 0 and 17_8.

spd — s-pad destination register: a name, number, or expression specifying a register number between 0 and 17_8.
SPFN is loaded into the s-pad destination register unless s-pad no-load (#) is specified.

& — Bit reverse: if present, bit reverse the contents of sps before using. The bit reverse is done as specified by bits 13-15 of the internal status register.

Examples:

```
MOV 3,6
SUBL 1,15
ADDL# &PTR, BASE
DEC CTR
CLR 9.
LDSPI 6
```

1.6.2 MEMORY ADDRESS OP-CODE GROUP

Purpose: initiate main data memory cycles

Op-codes	Function
INCMA	INCrement Memory Address
DECMA	DECrement Memory Address
SETMA	SET Memory Address from SPFN

1.6.3 TABLE MEMORY ADDRESS OP-CODE GROUP

Purpose: initiate table memory fetches

Op-codes	Function
INCTMA	INCrement Table Memory Address
DECTMA	DECrement Table Memory Address
SETTMA	SET Table Memory Address from SPFN

1.6.4 DATA PAD ADDRESS OP-CODE GROUP

Purpose: change the DPA (data pad address) register

Op-codes	Function
INCDPA	INCrement data pad Address
DECDPA	DECrement data pad Address
SETDPA	SET data pad Address from SPFN

1.6.5 BRANCH OP-CODE GROUP

Purpose: conditional branches

Op-code		Function
BR	disp	Branch unconditionally
BINTRQ	disp	Branch on INTerrupt ReQuest flag non-zero
BION	disp	Branch if I/O data ready flag Non-zero
BIOZ	disp	Branch if I/O data ready flag Zero
BFPE	disp	Branch on Floating Point Error
BFEQ	disp	Branch on Floating adder EQual to zero
BFNE	disp	Branch on Floating adder Not Equal to zero
BFGE	disp	Branch on Floating adder Greater or Equal to zero
BFGT	disp	Branch on Floating adder Greater than zero
BEQ	disp	Branch on s-pad function EQual to zero
BNE	disp	Branch on s-pad function Not Equal to zero
BGE	disp	Branch on s-pad function Greater or Equal to zero
BGT	disp	Branch on s-pad function Greater than zero
RETURN		RETURN from subroutine

Abbreviation:

disp	branch displacement: The branch target address, an address between 16 locations behind and 15 locations ahead of the current location.

Examples:

```
BR LOOP
BGT .+3
BFNE A-4
```

1.6.6 FLOATING ADDER OP-CODE GROUP

Purpose: floating-point adds

Double Operand Op-codes

Op-codes		Function
FADD	a1,a2	Floating ADD (a1+a2)
FSUB	a1,a2	Floating SUBtract (a1-a2)
FSUBR	a1,a2	Floating SUBtract Reverse (a2-a1)
FAND	a1,a2	Floating AND (a1 and a2)
FOR	a1,a2	Floating OR (a1 or a2)
FEQV	a1,a2	Floating EQuiValence (a1 eqv a2)

Single Operand Op-codes

Op-codes		Function
FIX	a2	FIX a2 to an integer
FIXT	a2	FIX a2 to an integer (Truncated)
FSCALE	a2	Floating SCALE of a2
FSCLT	a2	Floating SCALE of a2, (Truncated)
FSM2C	a2	Format conversion, Signed Magnitude to 2's complement
F2CSM	a2	Format conversion, 2's complement to signed magnitude
FABS	a2	Floating ABSolute value

Adder Operands:

operand	Meaning
a1	floating adder argument no. 1:

Choices	Meaning
NC	No Change (use previous a1)
FM	Floating Multiplier output
DPX {(idx)}	Data Pad X
DPY {(idx)}	Data Pad Y
TM	Table Memory data
ZERO	floating-point ZERO

Operand	Meaning
a2	adder argument no. 2:

Choices	Meaning
NC	No Change (use previous a2)
FA	Floating Adder output
DPX {(idx)}	Data Pad X
DPY {(idx)}	Data Pad Y
TM	Table Memory data
ZERO	floating ZERO
MDPX {(idx)}	use Mantissa from data pad X, and exponent from SPFN
EDPX {(idx)}	use Exponent data pad X, and mantissa from SPFN

Abbreviation:

Name	Meaning
idx	data pad index: a name, expression, or number which lies in a range of -4 to +3.

Examples:

```
FADD TM,MD
FSUB DPX(3), DPY(-4)
FEQV DPX, DPY(C)
FAND ZERO, MDPX(2)
FSUBR NC,FA
FADD
```

NOTE

Up to four unique data pad indices may be specified in one instruction statement. In particular, only one indexing each may be used for reading from data pad X and Y, regardless of how many op-codes use the data read from data pad.

1.6.7 FLOATING POINT MULTIPLY OP-CODE GROUP

Purpose: floating point multiplies

Op-code	Function
FMUL m1,m2	Floating MULtiply m1 times m2

Multiplier Operands:

Operand	Meaning
m1	Multiplier-operand no. 1

Choices	Meaning
FM	Floating Multiplier output
DPX{(idx)}	data pad X
DPY{(idx)}	data pad Y
TM	Table Memory

Operand	Meaning
m2	Multiplier-operand no. 2

Choices	Meaning
FA	Floating Adder output
DPX{(idx)}	data pad X
DPY{(idx)}	data pad Y
MD	Memory Data

Examples:

```
FMUL TM, MD
FMUL DPX (AR),DPY (BI)
FMUL
```

1.6.8 DATA PAD X OP-CODE GROUP

Purpose: storing into data pad X

Op-code	Function
DPX{(idx)}<opt	Store opt into Data Pad X. One of the following must be used for opt.

Opt	Meaning
FA	Floating Adder Output
FM	Floating Multiplier outut
DB	Data pad Bus
dbe	Data Pad bus Enable This has the same effect as an explicit data pad bus op-code. One choice of data pad bus enable may be made per instruction statement.

Choices	Meaning
ZERO	floating ZERO
adr	An address or numeric value. any 16-bit integer expression is legal. A floating multiplier, memory input, memory address or data pad address op-code can not be used in an instruction statement where an "adr" is used.
DPX{(idx)}	Data Pad X
DPY{(idx)}	Data Pad Y
MD	Memory Data
DPFN	s-pad Function
TM	Table Memory data

Examples:

```
DPX(3)<FM
DPX(-2)<SPFN
DPX MD
DPX(1)<DPY (-2)
DPX(-2)< -123
```

1.6.9 DATA PAD Y OP-CODE GROUP

Purpose: storing into data pad Y

Op-code	Function
DPY{(idx)}<opt	Store opt into data pad Y. The possibilities for opt are the same as those described in Section 1.6.8.

Examples:

```
DPY(-2)<FA
DPY<MD
DPY(2)<TM
DPY(1)<39
```

1.6.10 MEMORY INPUT OP-CODE GROUP

Purpose: writing into main data memory

Op-codes	Function
MI<FA	Move Floating Adder output to the Memory Input register
MI<FM	Move Floating Multiplier output to the Memory Input register
MI<DB	Move data pad Bus to the Memory Input Register
MI<dbe	Move dbe to the Memory Input Register

To effect a memory write, an op-code from the memory address group or an LDMA op-code must be included in the instruction statement to supply the memory address.

Examples:

```
MI<FA; INCMA
MI<DPX(3); DECMA
MI<MD; SETMA; ADD 3,6
```

1.6.11 DATA PAD BUS OP-CODE GROUP

Purpose: explicitly enable data onto the data pad bus

Op-codes	Function
DB=ZERO	enable ZERO onto the data pad bus
DB=adr	enable adr onto the data pad bus
DB=DPX{(idx)}	enable Data Pad X onto the data pad bus
DB=DPY{(idx)}	enable Data Pad Y onto the data pad bus
DB=MD	enable Memory Data onto the data pad bus
DB=SPFN	enable S-Pad Function onto the data pad bus
DB=TM	enable Table Memory data onto the data pad bus

As mentioned earlier, only one data source may be enabled onto the data pad bus per instruction statement.

Examples:

```
DB = 37
DB = DPX(-2)
DB = MD
DB = SPFN
```

1.6.12 SPECIAL OPERATION OP-CODE GROUP

If an op-code from this group is chosen, an s-pad group op-code can not be used in the same instruction statement.

1.6.12.1 Special Tests

Purpose: additional conditional branches

Op-codes	Function
BFLT disp	Branch on Floating adder Less Than zero
BLT disp	Branch on s-pad function Less Than zero
BNC disp	Branch on Non-zero Carry bit
BZC disp	Branch on Zero Carry bit
BDBN disp	Branch if Data pad Bus Negative
BDBZ disp	Branch if Data pad Bus Zero
BIFN disp	Branch if Inverse FFT flag Non zero
BIFZ disp	Branch if Inverse FFT flag Zero
BFL0 disp	Branch if FLag 0 is 1
BFL1 disp	Branch if FLag 1 is 1
BFL2 disp	Branch if FLag 2 is 1
BFL3 disp	Branch if FLag 3 is 1

If one of these tests is used along with a test from the branch group, the conditions are OR'd. In this case, only one of the branch op-codes need have the target address as an operand.

Examples:

```
BNC ODD
BFEQ LOOP; BFLT LOOP "LESS THAN OR EQUAL TO
```

1.6.12.2 SETPSA

Purpose: jumps and subroutine jumps

Op-codes	Function
JMP{A} adr	JuMP to location adr
JMPT	JuMP to location whose address is in TMA
JMPP	JuMP to location whose address is on the Panel bus
JSR{A} adr	JumP to SubRoutine at location adr
JSRT	JumP to SubRoutine at address in Tma
JSRP	JumP to SubRoutine at address on Panel bus

Examples:

```
JMP LOOP + 3
JSR FFT
JMPS 300
```

1.6.12.3 SETEXIT

Purpose: alter a subroutine return

Op-codes	Function
SETEX{A} adr	SET subroutine EXit to adr
SETEXT	SET subroutine EXit to contents of Tma
SETEXP	SET subroutine EXit to contents of Panel bus

Example:

```
SETEX BAD
```

1.6.12.4 Program Source

Purpose: read/write program source memory

Op-codes	Function
RPSL{A} adr	Read Program Source Left half of location adr
RPSF{A} adr	Read Program Source Floating-point number from location adr
RPSLT	Read Program Source Left half at address in Tma
RPSFT	Read program Source Floating point number at address in Tma
RPSLP	Read Program Source Left half at address on Panel bus
RPSFP	Read Program Source Floating-point number at address on Panel bus

These op-codes read onto the data pad bus

Op-codes	Function
LPSL{A} adr	Load Program Source Left half of location adr
LPSR{A} adr	Load Program Source Right half of location adr
LPSLT	Load Program Source Left half pointed at by Tma
LPSRT	Load Program Source Right half pointed at by Tma
LPSLP	Load Program Source Left half pointed at by Panel bus
LPSRP	Load Program Source Right half pointed at by Panel bus

These op-codes load from the data pad bus.

Example:

```
    RPSF PI
```

1.6.12.5 PS Odd and Even

Purpose: reading the host panel switches into program source memory, writing program source to the panel lites.

Op-codes	Function
RPS0{A} adr	Read Program Source quarter 0 from location adr
RPS1{A} adr	Read Program Source quarter 1 from location adr
RPS2{A} adr	Read Program Source quarter 2 from location adr
RPS3{A} adr	Read Program Source quarter 3 from location adr
RPS0T	Read Program Source quarter 0 from address in Tma
RPS1T	Read Program Source quarter 1 from address in Tma
RPS2T	Read Program Source quarter 2 from address in Tma
RPS3T	Read Program Source quarter 3 from address in Tma
WRS0{A} adr	Write Program Source quarter 0 into location adr
WPS1{A} adr	Write Program Source quarter 1 into location adr
WPS2{A} adr	Write Program Source quarter 2 into location adr
WPS3{A} adr	Write Program Source quarter 3 into location adr
WPS0T	Write Program Source quarter 0 into address in Tma
WPS1T	Write Program Source quarter 1 into address in Tma
WPS2T	Write Program Source quarter 2 into address in Tma
WPS3T	Write Program Source quarter 3 into address in Tma

1.6.12.6 Hostpanel

Purpose: reading the host panel switches, writing to the host panel lites

Op-codes	Function
PNLLIT	PaNel bus to LITes
DBELIT	Data pad Bus Exponent to LITes
DBHLIT	Data pad Bus High mantissa to LITes
DBLLIT	Data pad Bus Low mantissa to LITes
SWDB	SWitches to Data pad Bus
SWDBE	SWitches to Data pad Bus Exponent
SWDBH	SWitches to Data pad Bus High mantissa
SWDBL	SWitches to Data pad Bus Low mantissa

1.6.12.7 Miscellaneous

Op-codes	Function
SPNDAV	Spin until MD available

1.6.13 I/O OP-CODE GROUP

If an op-code is used from this group, a Floating Adder op-code can not be used in the same instruction statement.

1.6.13.1 Load REG, Read REG

Purpose: reading/writing various internal registers

Op-codes	Function
LDSPD	LoaD s-pad Destination address register
LDMA	LoaD Memory Address register
LDTMA	LoaD Table Memory Address register
LDDPA	LoaD Data Pad Address register
LDSP	LoaD s-pad register pointed at by spd
LDAPS	LoaD AP Status register
LDDA	LoaD I/O Device Address

The above op-codes load from the data pad bus.

Op-codes	Function
RPSA	Read Program Source Address
RSPD	Read s-pad Destination register
RMA	Read Memory Address register
RTMA	Read Table Memory Address register
RDPA	Read Data Pad Address register
RSPFN	Read s-pad Function
RAPS	Read AP Status
RDA	Read I/O Device Address

These op-codes read onto the panel bus.

1.6.13.2 INOUT

Purpose: program contol/input output of data

Op-codes	Function
OUT	OUTput data
SPNOUT	SPiN until device ready, then OUTput data
OUTDA	OUTput data, then set DA to spfn
SPOTDA	SPin until device ready, OuTput data, then set DA to spfn

The above write to the I/O device specified by the device address register (DA) whatever data is enabled onto the data pad bus.

Op-codes	Function
IN	INput data
SPININ	SPIN until device ready, then INput data
INDA	INput data, then set DA to spfn
SPINDA	SPin until device ready, the INput data, then set DA to spfn

These instructions put data onto the input bus from the I/O device specified by the device address register (DA). To be used the data must be put onto the data pad bus, and from there moved to a register or memory.

Example:

```
IN; DPX(2)<INBS    "READ I/O DATA INTO DPX
```

1.6.13.3 SENSE

Purpose: sensing an I/O device condition

Op-codes	Function
SNSA	SeNSe condition A
SPINA	SPIN on condition A
SNSADA	SeNSe condition A, then set DA to spfn
SPNADA	SPIN on condition A, then set DA to spfn
SNSB	SeNSe condition B
SPINB	SPIN on condition B
SNSBDA	SeNSe condition B, then set DA to spfn
SPNBDA	SPIN on condition B, then set DA to spfn

1.6.13.4 FLAG

Purpose: set/reset of program flags

Op-codes	Function
SFL0	Set Flag 0
SFL1	Set Flag 1
SFL2	Set Flag 2
SFL3	Set Flag 3
CFL0	Clear Flag 0
CFL1	Clear Flag 1
CFL2	Clear Flag 2
CFL3	Clear Flag 3

1.6.13.5 CONTROL

Purpose: miscellaneous control functions

Op-code	Functions
HALT	HALT processor
IORST	I/O ReSeT
INTEN	INTerrupt ENable
INTA	INTerrupt Acknowledge
REFR	memory REFResh synch
WRTEX	enable WRiTe of Exponent only into DPX, DPY or MI
WRTMN	enable WRiTe of MaNtissa only into DPX, DPY or MI
SPMDAV	SPin until a Main Data memory cycle AVailable

1.6.13.6 Miscellaneous

Purpose: miscellaneous control functions

Op-codes	Functions
REXIT	read subroutine exit into panel bus

1.6.14 TABLE MEMORY

This section lists the constants and functions available in table memory.

1.6.14.1 Constants

SYMBOL	CONSTANT REPRESENTED	VALUE IN TABLE MEMORY	2K TABLE MEMORY ROM ADDRESS (OCTAL)
!ZERO	ZERO	0.0	4371
!ONE	ONE	1.0	4001
!TWO	TWO	2.0	4002
!THREE	THREE	3.0	4441
!FOUR	FOUR	4.0	4442
!FIVE	FIVE	5.0	4443
!SIX	SIX	6.0	4444
!SEVEN	SEVEN	7.0	4445
!EIGHT	EIGHT	8.0	4446
!NINE	NINE	9.0	4447
!TEN	TEN	10.0	4450
!SIXTN	SIXTEEN	16.0	4451
!HALF	HALF	0.5	4427
!THIRD	ONE THIRD	.333333333	4430
!FOURTH		ONE FOURTH	0.254431
!FIFTH	ONE FIFTH	0.2	4432
!SIXTH	ONE SIXTH	0.166666667	4433
!SVNTH	ONE SEVENTH	0.142857143	4434
!EGHTH	ONE EIGHTH	0.125	4435
!NINTH	ONE NINTH	0.111111111	4436
!TENTH	ONE TENTH	0.1	4437
!SXNTH	ONE SIXTEENTH	0.0625	4440
!SQRT2	SQRT(2)	1.414213562	4203
!SQRT3	SQRT(3)	1.732050808	4422
!SQRT5	SQRT(5)	2.236067977	4423
!SQT10	SQRT(10)	3.162277660	4424
!ISQT2	1.0/SQRT(2)	0.707106781	4206
!ISQT3	1.0/SQRT(3)	0.577350269	4452
!ISQT5	I.0/SQRT(5)	0.447213596	4453
!ISQ10	I.0/SQRT(10)	0.316227766	4454
!CBT2	CBRT(2)	1.259921050	4417
!CBT3	CBRT(3)	1.442249570	4420
!QDRT2	(2.0)**1/4	1.189207115	4421
!LOG2E	LOG2(E)	1.442695041	4317
!LOG2	LOG10(2)	0.301029996	4411
!LOGE	LOG10(E)	0.434294482	4337
!LN2	LN(2)	0.693147181	4336
!LN3	LN(3)	1.098612289	4407
!LN10	LN(10)	2.302585093	4410
!E	E	2.718281828	4403
!INVE	1.0/E	0.367879441	4404
!ESQ	E**2	7.389056096	4405
!PI	PI	3.141592654	4402
!TWOPI	2*PI	6.283185308	4415
!INVPI	I.0/PI	0.318309886	4412
!P12	P1/2	1.570796327	4312

!PI4	PI/4	0.785398164	4373
!PII80	PI/180	0.017453293	4413
!PISQ	PI**2	9.869604404	4414
!SQTPI	SQRT(PI)	1.772453851	4416
!LNPI	LN(PI)	1.144729886	4406
!GAMMA	GAMMA	0.577215663	4425
!PHI	PHI	1.618033989	4426

1.6.14.2 Elementary Function Tables

SYMBOL	ELEMENTARY FUNCTION	TABLE MEMORY ADDRESS (OCTAL)
!DIV	DIVIDE	4000
!SQRT	SQUARE ROOT	4202
!SNCS	SIN/COS/4306	
!LOG	LOGARITHM	4333
!EXP	EXPONENTIAL	4317
!ATAN	ARC TANGENT	4365

1.6.14.3 Size of Installed FFT Cosine Table

SYMBOL	SIZE (TYPICAL)
!FFTSZ	2048 = 4000 (octal)

CHAPTER 2

APLINK

2.1 INTRODUCTION

APLINK links together separate object modules produced by APAL into a single load module for execution by the AP hardware or the simulator.

The user can separately code and assemble a main line program and the associated subroutines and later link them together for execution. APLINK serves this purpose by performing the following tasks:

- relocating each object module and assigning absolute addresses
- linking the modules together by correlating global entry symbols defined in one module with external symbols referenced in another module
- selectively loading modules from program library
- optionally producing a load map showing the layout of the load module

APLINK is written in FORTRAN IV and requires approximately 10K of memory.

2.2 OPERATING PROCEDURE

Program modules are linked interactively via a dialogue between the user and APLINK. The user enters a series of commands which direct the linking process.

When execution begins, APLINK outputs:

```
APLINK
version
*
```

The version is the version number of APLINK. The asterisk (*) indicates that the program is ready to accept commands. After each user command, an * is output when that command has been executed and APAL is ready for a new command. An illegal command causes a "?" to be output.

To load relocatable programs and prepare them for execution, the user would normally follow the procedure outlined below.

1. Using the L (load) command, load the file or files containing the desired main program, required subroutines, and library subprograms, if any. If a fatal error occurs during this step, reinitialize using the R command and repeat this step.

2. Using the U (undefined) command, check to see if any global symbols are still undefined. If nothing is output from this command, continue to step 3. If any symbols are output, it usually means that there was an error in one or more of the programs loaded or that the loading sequence was wrong. In these cases, correct the error and restart the loading operation from step 1.

3. Obtain the memory limits of the loaded program or a loader map by using the M (memory) or S (symbols) command.

4. Complete and output the load module by using the E (end) or by A command. Note the values of HIGH and START as well as the possible presence of any remaining undefined symbols.

5. Return to the operating system with an X (exit) command.

The individual APLINK commands are described in the following sub-sections, and a complete example loading session is given in section 2.2.12.

The following abbreviations are used in the following sub-sections:

Abbreviations	Meaning
(filename)	A user-specified input or output file. The (filename) follows whatever naming conventions exist for the particular host computer operating systems.
(cr)	carriage return
*	This is the terminal prompt indicating that the terminal is ready for input.
------	Indicates characters output by the program.

The examples given are illustrative only, as file and I/O device names vary from system to system.

2.2.1 LOAD L

To load a program module, or a program library, enter:

```
L           (cr)
(filename)  (cr)
```

where (filename) is the name of the file containing the desired program or library.

Example:

```
*
L       (cr)
FFT.RB  (cr)
```

This example loads the program contained on file FFT.RB.

In loading routines, the first entry point defined becomes the name of the host source output. An entry point may be defined by doing a force (F) without having loaded an object module previously.

2.2.2 SYMBOLS S

To output the global (external and entry) symbols enter:

```
S          (cr)
(filename) (cr)
```

where (filename) is the name of the file (or I/O device) to receive the symbol listing. The output of the loader map is as follows.

```
HIGH = aaaaaa
SYMBOL TABLE
SYMBOL              VALUE

ssssss              nnnnnn
  .                   .
  .                   .
  .                   .
```

where:

aaaaaa	highest program address so far loaded. Normally, the next program is loaded starting at location HIGH+1.
ssssss	symbol name
nnnnnn	symbol value; if undefined, the last location loaded which referenced this symbol.
U	if present indicates the symbol is as yet undefined.

Example:

```
*
S    (cr)
TP: (cr)
```

This example lists the loader symbol table at the terminal.

2.2.3 UNDEFINED U

To output to the console any presently undefined global symbols, enter:

```
U          (cr)
(filename) (cr)
```

where (filename) is the file to receive the list of undefined symbols. The list format is:

```
ssssss          nnnnnn
```

where ssssss is the symbol name and nnnnnn is the location of the last program instruction which referenced the symbol.

Example:

```
*
T    (cr)
TP:  (cr)
```

This example lists the names of any undefined symbols at the terminal.

2.2.4 NEXT BASE B

To specify a base address at which to load the next program, enter:

```
B       (cr)
(loc)   (cr)
```

where (loc) is the location specified.

Example:

```
*
B    (cr)
200  (cr)
```

This example sets the next location loaded to location 200.

2.2.5 RESET R

To reset APLINK, enter:

```
R(cr)
```

This reinitializes the program to its initial state. The symbol table is cleared, any previously loaded programs are disregarded, and the next location is set to zero. This command must be given following a fatal error.

2.2.6 FORCE F

To force loading of a program module from a library, enter:

```
F      (cr)
(name) (cr)
```

where (name) is the name of the symbol to be forced. This command enters (name) into the symbol table as an external symbol. This causes the loading of a library program which has (name) as an entry symbol.

Example:

```
*
F(cr)
DOTPRD(cr)
```

forces the loading of any program defining symbol DOTPRD from any subsequently loaded library file.

2.2.7 MEMORY M

To get the address of the highest program source memory location so far loaded, enter:

```
M (cr)
```

The information is printed as follows:

```
HIGH = aaaaaa
```

where aaaaaa is the highest address so far loaded, and bbbbbb, if present, is the load module starting address.

2.2.8 END E

To end a load module and output the completed load module for use with APDBUG or APSIM, enter:

```
E           (cr)
(filename)  (cr)
```

where (filename) is the name of the file to receive the loader output. The output is a core image which can be loaded by APDBUG and executed by either the simulator APSIM or the hardware.

APLINK outputs the following information to the user console:

```
HIGH = aaaaaa
```

where aaaaaa is the highest program address loaded. If any symbols were still undefined, APLINK outputs:

```
(num) UNDEFINED SYMBOLS
```

where (num) is the number still undefined. A value of 0 was used in linking these undefined symbols.

Example:

```
*
# (cr)
SAVE
```

stores the completed load module into file SAVE.

The E (or A) command causes links between global symbols in the completed load module to be frozen. The load module can be output again (with another E or A) but no further links can be added (with an L).

To work on another load module a reset (R) command must be given to clear the linker.

2.2.9 END WITH ASSEMBLY CODE A

To end a load module and output the completed load module as host computer assembly code (for use with APEX), enter:

```
A          (cr)
(filename) (cr)
```

where (filename) is the name of the file to receive the loader output. This output is a short host assembly (or possibly FORTRAN) language subroutine, which is the linkage between host computer FORTRAN calls and the AP executive. The AP code from the load module follows the host subroutine as assembly language data statements.

Information concerning the highest address loaded into, and any undefined symbols, are output to the user console as described above for the E command.

2.2.10 NUMBER RADIX N

To set the radix for numeric input/output to and from the user console, enter:

```
N       (cr)
(radix) (cr)
```

where (radix) is either 8 (for octal), 10 (for decimal), or 16 (for hexadecimal). The default radix for user I/O is set to either of these choices at installation.

2.2.11 EXIT X

To exit to the operating system, enter:

```
X(cr)
```

Notice that the X command does not cause any output. The E or A command must be used to output a load module.

2.2.12 AN EXAMPLE LOADING SESSION

```
OK, APLINK                          Call LINKER
GO
 APLINK
 REV 2
 *
L
PROG1.OBJ                           Load PROG1.OBJ
 *
U                                   Output any undefined symbols
TTY                                 to the terminal
 DIV      000004  U
 *                                  DIV subroutine is undefined
L                                   Load DIV
APLIB                               from subroutine library
 LOAD COMPLETE
 *
S                                   Output global External and Entry symbols
TTY
 HIGH=000042


 SYMBOL TABLE

 SYMBOL VALUE
 PROG1  000000
 DIV    000007
 *
E
PROG1.SOURCE
 PROG1      HIGH=000042             Create PROG1.APSIM to run on the simulator
 *
A
PROG1.APSIM                         Create PROG1.SOURCE (host FORTRAN or assembler)
 PROG1      HIGH=000042             to run on host system
 *
X                                   End (Return to the operating system)
```

2.3 ERROR MESSAGES

Any deviation from the prescribed command syntax causes APLINK to output a "?" to the user console. The illegal command is ignored, and APLINK outputs a "*" to indicate its readiness to accept a new command.

If a specified filename cannot be found or is otherwise unavailable for use, the message:

```
FILE NOT FOUND!!!
```

is output and the command is ignored.

The specific error messages output by APLINK are the result of loading errors detected during execution of an L (load) command. There are two classes of loading errors:

F - fatal. Reinitialization of the loader (the R command) is required before loading can continue.

W - warning. An advisory message indicating a non-error.

Any fatal error detected during loading causes immediate termination of the L (load) command following the error message. If the user attempts to execute another L command, the program outputs the message:

```
RESET!!!
```

and ignores the command. After the user reinitializes the loader (R command), he must reload any programs loaded up to that point.

Following are the error messages, along with notes of explanation for each:

F SYMBOL TABLE OVERFLOW

The loader symbol table is full. The only recourse is to recompile APLINK with a longer symbol table size.

F PROGRAM MEMORY OVERFLOW nnnnnn

An attempt was made to load the upper limit of program source memory. The load module is too large to fit in program source memory. nnnnnn is the memory location involved.

F OVERWRITE nnnnnn

An attempt was made to overwrite a previously loaded program memory location. The loader does not permit any given program memory location to be loaded more than once. nnnnnn is the program memory location involved.

F ILLEGAL BLOCK TYPE nnnnnn

An illegal relocatable object code block type was encountered. The file specified does not contain legal object code. nnnnnn is the illegal block type, as read from the block header in question.

W MULTIPLE ENTRY

An $ENTRY symbol having the same name as one already defined was encountered during a load. The name and value of the symbol is output to the terminal as follows:

```
ssssss          nnnnnn
```

where ssssss is the symbol name and nnnnnn the symbol value. The loader proceeds by ignoring the latest definition.

W MISSING OR IMPROPER ENTRY

The user attempted to produce host assembly code (an A command) from a load module and one of the following was true:

- The load module did not have any entry points (defined entry global symbols).
- The first entry point loaded did not have an s-pad parameter count.

2.4 SUMMARY OF APLINK COMMANDS

This section contains a summary of APLINK commands. The following abbreviations are used in this section.

Abbreviation	Meaning
(cr)	carriage return
(filename)	name of a file, as appropriate for the host operating system being used
(loc)	a location, octal or hex decimal as appropriate
(name)	a symbol name, six characters or less

Command	Effect
L(cr) (filename)(cr)	load the program in file (filename), link with previously loaded programs.
S(cr) (filename)(cr)	output the loader symbol table to file (filename)
U(cr) (filename)(cr)	output any undefined symbols to file (filename)
B(cr) (loc)(cr)	set APLINK to load the next program at location (loc).
R(cr)	reset the loader.
F(cr) (name)(cr)	force the loading of a program defining symbol; name from any subsequent program libraries loaded.
M(cr)	output the highest program memory location used.
E(cr) (filename)(cr)	end the loading session; store the resulting load module into file (filename).
A(cr) (filename)(cr)	end the loading session; output host computer assembly code for use with APEX into file (filename).
N(cr) (number)(cr)	set the radix for numeric user consol I/O to either 8, 10, or 16.
(X)(cr)	exit to the operating system.

2.5 RELOCATABLE OBJECT CODE BLOCK TYPES

Unlike most relocatable binary, the relocatable object code produced by APAL consists of numbers written as decimal integer characters. Those were output (and readable) by FORTRAN formatted I/O statements.

An advantage of this type of code is that relocatable library files can be edited with an ordinary text editor. This makes unnecessary the need for a special-purpose library of library file editor.

The relocatable object code is divided into a series of blocks. The order in which blocks appear, if each type is present, is as follows (the block type number is in parenthesis):

1. title blocks (3)
2. entry blocks (4)
3. code blocks (0)
4. external blocks (5)
5. end block (1)

An object module contains at least a title block and an end block. The presence of one or more of the other block types depends upon the particular program.

The first line of each block is a block header, which contains four seven-digit numbers:

1. block type numbers
2. number of items in the block
3. initial address, if relevant
4. unused

In addition, the block header is flagged with "***" to aid in identification of blocks.

Each block type is described on the following page, in numeric order by block type numbers.

2.5.1 CODE BLOCK (0)

Line	Contents:		
0	0	count	address 0***
1	Bits 0-15	Bits 16-31/Bits 48-63	
2	"	"	"
.	.	.	.
.	.	.	.
.	.	.	.
count	"	"	"

Each code line contains a 64-bit program source word.

2.5.2 END BLOCK (1)

Line		Contents		
0	1	0	0	0***

2.5.3 TITLE BLOCK (3)

Line	Contents		
0	1	0	0***TITLE
1	0		

2.5.4 ENTRY SYMBOL BLOCK (4)

Line		Contents		
0	4	count	0	0***
1	name	value		#S-Pad parameters
2	"	"		"
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.
count	"	"		"

2.5.5 EXTERNAL SYMBOL BLOCK (5)

Line		Contents		
0	5	count	0	0***
1	name	link		
2	"	"		
3	"	"		
.	.	.		
.	.	.		
.	.	.		
count	"	"		

2.5.6 LIBRARY START BLOCK(6)

Line		Contents		
0	6	0	0	0***

2.5.7 LIBRARY END BLOCK (7)

Line		Contents		
0	7	0	0	0***

2.5.8 EXAMPLE RELOCATABLE OBJECT PROGRAM

The object module is an ASCII file and therefore can be modified with the text editor.

```
     3.      1.      0.      0.***TITLE      1.  Title
PROG1        0.
     4.      1.      0.      0.***           2.  Entry Block
PROG1        0.      3.
     0.      7.      0.      0.***
 16384.      0.      0.     48.
 16452.      0.      0.     48.
     0.      0.      0.      0.
     0.      0.   6656.  32768.              3.  Code Block
  4620.      0.  18948.  65535.
 16520.      0.   1792.    240.
     0.    224.      0.      0.
     5.      1.      0.      0.***           4.  External Block
DIV          4.
     1.      0.      0.      0.***           5.  End Block
```

*** indicates a new block

1. The title block contains the title of the program, PROG1.

2. The entry block has the name of the entry point, PROG1; its relative adress, 0; and the number of expected s-pad parameters, 3.

3. The code block contains the seven AP program words in PROG1, each as four 16-bit quarters of a 64-bit program word.

4. The external block contains the external symbol DIV.

5. The end block tells APLINK that it has reached the end of the program.

2.5.9 EXAMPLE OF E OUTPUT

Example output from APLINK produced by an E (end) command from a program, PROG1, follows. APDBUG can load this output into either the simulated AP (APSIM), or the actual hardware for debugging.

The first line says that the program contains 35 program words.

```
35.
16384.  00000.  00000.  00048.
16452.  00000.  00000.  00048.
00000.  00000.  00000.  00000.
00000.  00000.  06656.  32768.
04620.  00000.  18948.  00003.
16520.  00000.  01792.  00240.
00000.  00224.  00000.  00000.
00956.  21504.  01280.  02050.
00001.  32768.  06176.  32768.
00951.  34304.  33796.  02048.
01016.  00286.  01792.  00000.
09148.  00000.  00000.  00003.
00574.  52224.  00256.  00000.
00001.  32768.  00032.  07680.
00628.  00000.  00000.  07171.
00888.  00000.  01792.  04096.
04112.  00018.  12288.  36864.
00761.  54866.  49188.  00000.
00761.  22016.  49188.  00000.
00637.  17408.  00256.  05376.
```

```
00632.  00000.  27653.  36864.
00637.  24064.  00320.  04096.
00001.  37376.  00256.  04352.
00001.  32853.  00032.  06144.
00831.  39936.  17412.  01023.
00064.  00000.  05888.  32768.
00952.  00000.  01024.  08192.
25535.  35925.  03360.  05632.
00000.  00000.  00000.  04096.
00000.  37376.  00000.  04096.
00001.  32768.  00000.  00000.
00000.  00000.  00000.  04096.
00000.  00000.  00000.  04096.
00000.  00000.  00000.  04096.
00000.  00224.  49156.  00000.
```

2.5.10 EXAMPLE OF APLINK SOURCE OUTPUT

```
      SUBROUTINE PROG1 (I
     X    1,I 2,I
     X    3)
      INTEGER CODE(  141)
      INTEGER I 1,J 1
      INTEGER I 2,J 2
      INTEGER I 3,J 3
      INTEGER SLIST(16)
      COMMON /SPARY/SLIST
      EQUIVALENCE (J 1,SLIST( 1))
      EQUIVALENCE (J 2,SLIST( 2))
      EQUIVALENCE (J 3,SLIST( 3))
      DATA CODE(1) /  35/
      DATA CODE(    2),CODE(    3),CODE(    4),CODE(    5)/
     X  :040000,:000000,:000000,:000060/
      DATA CODE(    6),CODE(    7),CODE(    8),CODE(    9)/
     X  :040104,:000000,:000000,:000060/
      DATA CODE(   10),CODE(   11),CODE(   12),CODE(   13)/
     X  :000000,:000000,:000000,:000000/
      DATA CODE(   14),CODE(   15),CODE(   16),CODE(   17)/
     X  :000000,:000000,:015000,:100000/
      DATA CODE(   18),CODE(   19),CODE(   20),CODE(   21)/
     X  :011014,:000000,:045004,:000003/
      DATA CODE(   22),CODE(   23),CODE(   24),CODE(   25)/
     X  :040210,:000000,:003400,:000360/
      DATA CODE(   26),CODE(   27),CODE(   28),CODE(   29)/
     X  :000000,:000340,:000000,:000000/
      DATA CODE(   30),CODE(   31),CODE(   32),CODE(   33)/
     X  :001674,:052000,:002400,:004002/
      DATA CODE(   34),CODE(   35),CODE(   36),CODE(   37)/
     X  :000001,:100000,:014040,:100000/
      DATA CODE(   38),CODE(   39),CODE(   40),CODE(   41)/
     X  :001667,:103000,:102004,:004000/
      DATA CODE(   42),CODE(   43),CODE(   44),CODE(   45)/
     X  :001770,:000436,:003400,:000000/
      DATA CODE(   46),CODE(   47),CODE(   48),CODE(   49)/
     X  :021674,:000000,:000000,:000003/
      DATA CODE(   50),CODE(   51),CODE(   52),CODE(   53)/
     X  :001076,:146000,:000400,:000000/
      DATA CODE(   54),CODE(   55),CODE(   56),CODE(   57)/
     X  :000001,:100000,:000040,:017000/
      DATA CODE(   58),CODE(   59),CODE(   60),CODE(   61)/
     X  :001164,:000000,:000000,:016003/
      DATA CODE(   62),CODE(   63),CODE(   64),CODE(   65)/
     X  :001570,:000000,:003400,:010000/
      DATA CODE(   66),CODE(   67),CODE(   68),CODE(   69)/
     X  :010020,:000022,:030000,:110000/
      DATA CODE(   70),CODE(   71),CODE(   72),CODE(   73)/
     X  :001371,:153122,:140044,:000000/
      DATA CODE(   74),CODE(   75),CODE(   76),CODE(   77)/
```

```
X  :001371,:053000,:140044,:000000/
     DATA CODE(   78),CODE(   79),CODE(   80),CODE(   81)/
X  :001175,:042000,:000400,:012400/
     DATA CODE(   82),CODE(   83),CODE(   84),CODE(   85)/
X  :001170,:000000,:066005,:110000/
     DATA CODE(   86),CODE(   87),CODE(   88),CODE(   89)/
X  :001175,:057000,:000500,:010000/
     DATA CODE(   90),CODE(   91),CODE(   92),CODE(   93)/
X  :000001,:111000,:000400,:010400/
     DATA CODE(   94),CODE(   95),CODE(   96),CODE(   97)/
X  :000001,:100125,:000040,:014000/
     DATA CODE(   98),CODE(   99),CODE(  100),CODE(  101)/
X  :001477,:116000,:042004,:001777/
     DATA CODE(  102),CODE(  103),CODE(  104),CODE(  105)/
X  :000100,:000000,:013400,:100000/
     DATA CODE(  106),CODE(  107),CODE(  108),CODE(  109)/
X  :001670,:000000,:002000,:020000/
     DATA CODE(  110),CODE(  111),CODE(  112),CODE(  113)/
X  :061677,:106125,:006440,:013000/
     DATA CODE(  114),CODE(  115),CODE(  116),CODE(  117)/
X  :000000,:000000,:000000,:010000/
     DATA CODE(  118),CODE(  119),CODE(  120),CODE(  121)/
X  :000000,:111000,:000000,:010000/
     DATA CODE(  122),CODE(  123),CODE(  124),CODE(  125)/
X  :000001,:100000,:000000,:000000/
     DATA CODE(  126),CODE(  127),CODE(  128),CODE(  129)/
X  :000000,:000000,:000000,:010000/
     DATA CODE(  130),CODE(  131),CODE(  132),CODE(  133)/
X  :000000,:000000,:000000,:010000/
     DATA CODE(  134),CODE(  135),CODE(  136),CODE(  137)/
X  :000000,:000000,:000000,:010000/
     DATA CODE(  138),CODE(  139),CODE(  140),CODE(  141)/
X  :000000,:000340,:140004,:000000/
     J 1=I 1
     J 2=I 2
     J 3=I 3
     CALL APEX(CODE,    0,SLIST,    3)
     RETURN
     END
```

The APLINK source consists of four basic parts: the SUBROUTINE start, SLIST array, CODE array, and the APEX call.

The subroutine start contains this routine's arguments, the number of which corresponds to the s-pad parameter with the first $ENTRY start in the APAL code. The subroutine name is the $ENTRY symbol so that when the user calls PROG1, control is passed through this host source routine which then causes the execution of the previously loaded APAL code of the same name. At run time the arguments are transferred to the SLIST array. These are usually addresses of data already in the AP via APPUT calls in other routines. The code array contains the load module created by the user, in this case - PROG1 and the DIV routines. The first element of the array is the number of AP program source words; the following values correspond to the actual micro-code.

The APEX calls cause the micro-code to be loaded into AP program source memory unless it still resides there from previous calls. The arguments values are placed in their respective s-pad registers (16 is maximum), and control is finally passed to the routine entry point.

CHAPTER 3

APSIM AND APDBUG

3.1 INTRODUCTION

APSIM and APDBUG provide an interactive facility for checking out AP programs. The user can run portions of his AP program, stop and examine the results, make program patches, and then continue with program execution. The process of interactive debugging greatly facilitates preparation of correctly operating AP programs.

APSIM and APDBUG have commands to:

- examine or change memory locations and registers inside the AP
- type out memory contents and integers, floating point numbers or program word fields
- set, clear and examine breakpoints
- run programs, or execute them one step at a time

APSIM and APDBUG are actually independent debugging systems. However, the virtually identical input commands justify combining their descriptions.

APSIM is useful for initial program development and has the advantage that it allows debugging off-line from the AP hardware. It allows access to more internal AP registers than with the hardware. Simulation is limited, however, to program execution inside the AP. Input/output interaction with the host computer is not simulated. Depending on the speed of the host computer, the simulator runs about one million times slower than real time or about six instruction cycles per second.

APDBUG is useful for debugging AP programs which require long execution times and/or real-time interaction with the host computer.

This chapter describes Release 2.1 of APSIM-APDBUG. As usual (cr) means carriage return or end of line, as appropriate to the particular host computer system. In the examples listed, the responses output by the computer are underlined.

3.2 OPERATING PROCEDURE

Debugging is the process of detecting, locating, and removing mistakes from a program. When the programmer wishes to debug an AP program, he loads the program into APDBUG (or APSIM). He may then control program execution, causing the program to breakpoint at selected program locations so that the contents of registers or memory locations can be examined. Contents may be examined as program words, integers, or floating-point numbers.

APDBUG (or APSIM) types an asterisk (*) when ready for user input. A question mark (?) is typed when an error is detected.

3.2.1 MONITORING REGISTERS AND MEMORY LOCATIONS

Registers and memory locations in the AP may be opened, examined and modified using one of the following commands:

Command	Function
E	open and examine locations in the AP
+	examine the next higher location in an AP memory
-	examine the next lower location in an AP memory
C	change the open location
.	re-examine the currently open location
Z	zero out all AP registers and memories
O	set program source memory offset

A register in the AP is opened with an E (exam), + (next), or - (last) command. APDBUG (or APSIM) gets the value of the desired location in the AP and outputs the value at the terminal. If desired, the contents of the location may be changed with a C (change) command. A . (re-exam) lists the contents of the open register.

3.2.2 OPEN AND EXAMINE (E)

To open and examine a register in the AP enter:

```
E      (cr)
(name) (cr)
```

where (name) is the name of the desired register.

To open and examine a memory location in the AP enter:

```
E          (cr)
(name)     (cr)
(location) (cr)
```

where (name) is the memory name and (location) is the desired memory location.

A list of the examinable registers and memories and their description is given in Section 3.4.5. For the purposes of APDBUG, all functional units of the AP which have addresses are considered memories. This includes the three obvious memories (main data memory, table memory and program memory) plus data pad X, data pad Y and s-pad.

Examples:

- Examine main data memory location 23.

```
*
E  (cr)
MD (cr)
23 (cr)
-234.0000000
*
```

  MD location 23 contains -234.0.

- Examine the memory address register.

```
*
E  (cr)
MA (cr)
1376
*
```

  MA contains 1376.

3.2.3 EXAMINE NEXT, LAST AND RE-EXAMINE (+, -, .)

To open and examine the next higher sequential memory location above a currently open memory location enter:

```
+ (cr)
```

To open and examine the next lower sequential memory location below a currently open memory location enter:

```
- (cr)
```

To re-examine the currently open memory location enter:

```
. (cr)
```

Examples:

- Examine main data memory locations 23 and 24.

```
*
E  (cr)
MD (cr)
23 (cr)
-234.0000000
*
```

MD location 23 contains -234.0; now examine MD location 24.

```
*
+ (cr)
MD   000024
789.0000000
*
```

MD location contains 789.0.

- Examine s-pad registers 7 and 6.

```
*
E  (cr)
SP (cr)
7  (cr)
000027
*
```

S-pad register 7 contains 27. Now examine register 6.

```
*
- (cr)
SP   000006
-136
*
```

S-pad register 6 contains -136.

3.2.4 CHANGE (C)

To change the contents of a currently open register or memory location to a specified value enter:

```
C       (cr)
(value) (cr)
```

where (value) is an integer or floating-point number (depending upon what register or memory is open). (See Section 3.2.2.)

To change a register or memory location the user must first open it by doing an E, + or - command.

Examples:

- Examine main data memory location 20 and then change its value to -97.5.

```
*
E      (cr)
MD     (cr)
20     (cr)
76.00000000
*
C      (cr)
-97.5  (cr)
*
```

Main data memory location 20 contained 76.0. The user changed it to contain -97.5.

- Now change main data memory location 21 to -63.4.

```
*
+      (cr)
MD    000020
- 3.000000000
*
C      (cr)
-63.4 (cr)
*
```

MD location 21 contained -3.0 and was changed to contain -63.4.

- Examine s-pad register 3 and change its value to 17.

```
*
E  (cr)
SP (cr)
3  (cr)
56
*
C  (cr)
17 (cr)
*
```

S-pad register 3 contained 56 and was changed to contain 17.

3.2.5 SET PROGRAM SOURCE OFFSET (O)

To set the program source memory addressing offset enter:

```
O         (cr)
(value)   (cr)
```

where (value) is an integer in the current radix specifying the offset to be used when accessing program memory. The default setting is 0.

The offset is used when debugging a load module containing several separately assembled programs. For example, assume that programs A, B, and C have been loaded together with APLINK and the following load map obtained from APLINK with the S command.

```
SYMBOL     VALUE

  A        000000
  B        000153
  C        000247
```

This means that program A was loaded at PS location 0, B at location 153, and C at location 247.

To examine locations 3 and 4 of program B, type:

```
*
E     (cr)
PS    (cr)
156   (cr)
000000  000000  000000  000000
*
+     (cr)
PS     000157
000000  000000  000000  000000
*
-
```

This may become confusing because locations 156 and 157, printed by APDBUG (or APSIM) don't agree with the APAL listings which always start at zero. The offset simplifies matters by adjusting the base address for all PS related I/O. Thus, for convenience-sake, the offset should be set to the base address of the program currently being examined.

```
*
0     (cr)
153   (cr)
*
E     (cr)
PS    (cr)
3     (cr)
000000  000000  000000  000000
*
+     (cr)
PS     000004
000000  000000  000000  000000
*
```

The offset applies to examining or changing PS and PSA and also to breakpoints and running programs. It should be remembered, when setting the offset, that it is not relative to itself, but is an absolute address. Thus, the offset can always be reset to the default value of zero by typing the following:

```
0  (cr)
0  (cr)
```

3.2.6 CHANGING INPUT/OUTPUT FORMATS

The input and output format used when examining and changing registers and memory locations may be selected using the following commands:

Command	Function
N	sets the radix for integers
F	sets the format for input/output of 38-bit wide registers and memory words
V	sets the format for input/output of 64-bit wide program memory words

APDBUG selects the proper format for input/output depending on the word size of the particular register or memory location that is open and the setting of the previous three commands as follows.

- 16-bit words — MA, TMA, DPA, s-pad, etc. These locations are examined or changed as integers in the radix as selected by N.
- 38-bit words — DPX, DPY, main data memory, table memory, etc. These locations are examined or changed as either floating-point numbers or as three integers, depending upon the F command.
- 64-bit words — program memory. These locations are examined or changed as either op-code fields or as four 16-bit integers depending upon the V command.

The listing of accessible AP register and memories in Section 3.4.5 specifies the width of each as one of the following:

16-bit (integer word)
38-bit (floating point word)
64-bit (program word)

NOTE

Integer output is always unsigned on the range 0-177777 (octal), or 0-65536 (decimal), or 0-FFFF (hexadecimal). Thus, negative two's complement numbers are typed out as their 16-bit unsigned equivalents. For example (in octal) -1 would be output as 177777, and -2 as 177776, and so forth.

3.2.7 SET RADIX (N)

To set the radix for all integers input/output to APDBUG enter:

N (cr)
(radix) (cr)

where (radix) is either 8, 10 or 16 for octal, decimal or hexadecimal radixes respectively. (radix) is always entered as a decimal value.

The contents of 16-bit wide registers (s-pad, MA, PSA, etc.) are examined and changed using the integer radix as set by the N command. In addition, memory addresses are also entered using the current radix.

On listings, octal numbers may be recognized as having six digits, decimal numbers as having five digits and hexadecimal numbers as having four digits.

The default radix is either octal or hexadecimal depending upon the conventions of the host computer.

Examples:

- Examine s-pad register 10 (decimal) in all three radixes (starting in decimal).

```
*
E    (cr)
SP   (cr)
10   (cr)
 32768
*
N    (cr)
8    (cr)
*
.    (cr)
SP     000012
100000
*
N    (cr)
16   (cr)
*
.    (cr)
SP       000A
  8000
*
```

The value of s-pad register 10 is 32768 (decimal) or 100000 (octal) or 8000 (hexadecimal).

3.2.8 SET/RESET FLOATING POINT I/O (F)

To select floating-point input/output of 38-bit registers and memory words enter:

```
F (cr)
1 (cr)
```

To select integer input/output (in the current integer radix) of 38-bit wide registers and memory locations enter:

```
F (cr)
0 (cr)
```

38-bit wide registers are split into three pieces: 10-bit exponent, 12-bit high mantissa (bits 0-11) and 16-bit low exponent (bits 12-27) for integer I/O.

Data pad, main data memory, table memory and data pad bus are among the registers and memories whose I/O is governed by the F command.

Both examining and changing of 38-bit registers are effected by F. The default setting of the F command is one (for floating-point I/O).

- Examine command output formats.

  F=1: (floating-point number)

  F=0: (exponent) (high mantissa) (low mantissa)

- Change command input formats.

```
F=1: C                          (cr)
     (floating point number)    (cr)

F=0: C                (cr)
     (exponent)       (cr)
     (high mantissa)  (cr)
     (low mantissa)   (cr)
```

legal floating point numbers are of the form

±XX.YYE±ZZ

where:

XX is the integer part
YY is the fraction part
ZZ is the exponent

Any of the three parts may be omitted, except in the case when an exponent is used. In this case, either an integer part or a fraction part must also be included. The signs may be omitted if + is used. The decimal point may be omitted if not needed. No spaces are allowed inside floating-point numbers. The following are all legal floating-point inputs.

```
-2.3E6
  .7E-3
 -2
 3.65
  .7
```

Examples:

- Examine data pad register six in both floating point and integer. (Assume the integer radix is 16.)

```
*
E   (cr)
DPX (cr)
6   (cr)
-1.000000000
*
F   (cr)
0   (cr)
*
.   (cr)
DPX      0006
  0200  0400  0000
*
```

DPX register six contains -1.0. Its exponent is 200 (hexadecimal) which has an exponent value of zero (0). The fraction part is 4000000 (hexadecimal) which is a fraction of -1.0.

- Now change the exponent to 204 and the fraction to 2000000 and set F to 1.

```
*
C    (cr)
204  (cr)
200  (cr)
0    (cr)
*
F    (cr)
1    (cr)
*
.    (cr)
DPX      0006
8.000000000
*
```

DPX register now contains 8.0 which is 0.5*2**4.

3.2.9 SET/RESET PROGRAM WORD FIELD I/O (V)

To select input/output of 64-bit wide programs words by op-code fields enter:

```
V (cr)
1 (cr)
```

To select input/output of program words as four 16-bit numbers enter:

```
V (cr)
0 (cr)
```

The four 16-bit integers represent bits 0-15, 16-31, 32-47 and 48-63 of a program word.

Both examining and changing of program words are effected by V.

The default for the V command is 0 for integer I/O of program words.

- Examine command output formats.

```
V=1:  (24 op code field values)
V=0:  (bits 0-15) (bits 16-31) (bits 32-47) (bits 48-63)
```

- Change command input formats.

```
V=1:  C (cr)
      (desired op-code field to change)  (cr)
      (new value)  (cr)

V=0:  C (cr)
      (bits 0-15)   (cr)
      (bits 16-31)  (cr)
      (bits 32-47)  (cr)
      (bits 48-63)  (cr)
```

The program word op-code fields are listed in Section 3.4.6. When V=1, on examine, the 64-bits of a program word are divided into 24 fields, whose values are printed out. On change, the user enters the name of the field he wants to change along with the new value (hence the mnemonic V) for that field. The legal values for each field are listed in the AP Processor Handbook.

Examples:

- Program location 20 contains the instruction

  ```
  LDSPI 14; DB=200
  ```

  which sets s-pad register 14 to 200. This example changes the instruction so that s-pad 14 is set to 300 instead.

```
*
E       (cr)
PS      (cr)
20      (cr)
001660 000000 000000 000200
*
C       (cr)
1660    (cr)
0       (cr)
0       (cr)
300     (cr)
*
```

Note that to change the value field (which is the fourth quarter) (bits 48-63) of the program word, all four quarters had to be entered.

- Now change the instruction so that s-pad register 11 (instead of 14) is loaded with 300.

```
*
V     (cr)
1     (cr)
*
C     (cr)
SPD   (cr)
11    (cr)
*
```

The SPD (s-pad destination) field (program word bits 10-13) was changed to 11.

- Examine program memory location 20 in both formats.

```
*
. (cr)
PS     000020
001644 000000 000000 000300
*
V (cr)
1 (cr)
*
. (cr)
PS     000020
```

B	00	SOP	00	SH	00	SPS	16	SPD	11	FADD	00
A1	00	A2	00	COND	00	DISP	00	DPX	00	DPY	00
DPBS	00	XR	00	YR	00	XW	00	YW	00	FM	00
M1	01	M2	02	MI	00	MA	00	DPA	00	TMA	00

SPS=16 indicates the LDSPI instruction, SPD=11 indicates the s-pad destination register is 11. The YW, FM, M1, M2, MA, DPA and TMA fields are meaningless since the value of 300 occupied these fields.

- Program location 30 contains the instruction.

```
FADD FM, MD; FMUL TM, MD
```

Change the second argument for the FADD (A2) from MD to FA.

```
*
E  (cr)
PS (cr)
30 (cr)
000001 114000 000000 160000
*
V  (cr)
1  (cr)
*
C  (cr)
A2 (cr)
1  (cr)
*
```

3.3 MEMORY LOADING AND DUMPING

Blocks of AP memory locations may be loaded and dumped to and from files with the following commands:

Y	yank (load) into a memory from a file
W	write out the contents of a memory to a file
Z	zero all the memories (and registers) (APSIM only)

The list of memories on which the above commands can operate is different for APDBUG and for APSIM. In APSIM, only main data memory (MD), table memory (TM), and program source memory (PS) may be yanked into or written from. In APDBUG, the list of memories is extended to include s-pad (SP) and data pad X and Y (DPX, DPY).

A further difference lies in the area of I/O data formats. In APSIM, yanking and writing to/from 38-bit memories are always performed in the floating-point format (F=1). Program source memory I/O is always in integer mode (V=0). In APDBUG, I/O to/from 38-bit memories is governed by the F command. Hence, it is either in floating-point or integer mode, as set by the user. Program source memory input is always in integer mode, whereas output can be in either integer or op-code field format, as governed by the current setting of the V command.

The user should be aware that the procedure for entering filenames varies greatly according to the respective system. In some systems the notion of user files is nonexistent. In these cases, a logical unit number referring to an I/O device, which was opened previously by JCL control statements must be entered in place of a filename. Other systems allow access to disk files, line printers and terminals by symbolic names. Thus, what must be entered for a filename depends on the convention of each respective system. The examples given in the following are only meant to be representative and may not be legal on a given system.

3.3.1 YANK FROM A FILE (Y)

To load a memory from a file enter:

```
Y                    (cr)
(memory name)        (cr)
(starting location)  (cr)
(filename)           (cr)
```

(memory name) is an AP memory. The beginning memory address is loaded at (starting location). The name of the file from which the data is to be read is called (filename). (filename) must, of course, be in the proper form as determined by the particular host operating system.

Yank is used typically to load programs into program memory and data into main data memory.

Examples:

- Load a program into PS location 0. The program is assumed to be in a file named PROG1 which was created using the E command output from APLINK.

```
*
Y       (cr)
PS      (cr)
0       (cr)
PROG1   (cr)
*
```

- Load data into MD starting at location 20 from a file called DATA. Section 3.3.4 explains how to create data files.

```
*
Y      (cr)
MD     (cr)
20     (cr)
DATA   (cr)
*      (cr)
```

3.3.2 WRITE TO A FILE (W)

To write the contents of a memory into a file enter:

```
W                    (cr)
(memory name)        (cr)
(starting location)  (cr)
(ending location)    (cr)
(filename)           (cr)
```

(memory name) is an AP memory, (starting location) is the initial address to be written, (ending address) is the last address to be written and (filename) is the name of the file into which the data is to be written.

Examples:

- Write main data memory locations 20 through 40 into a file called DUMP.

```
*
W      (cr)
MD     (cr)
20     (cr)
40     (cr)
DUMP   (cr)
*
```

- Write data pad X, locations 3 through 6, to the line printer first, in floating-point format and second, in integer format. (In this example, the line printer is called LP:.) Note that a data pad may be dumped only from APDBUG.

```
*
F     (cr)
1     (cr)
*
W     (cr)
DPX   (cr)
3     (cr)
6     (cr)
LP:   (cr)
*
F     (cr)
0     (cr)
*
W     (cr)
DPY   (cr)
3     (cr)
6     (cr)
LP:   (cr)
*
```

If the user mistypes a W command, he has several options to abort the command. If the wrong memory name or starting address was typed, the command may be canceled by entering an ending address (which is lower than the starting address). In APDBUG, an unwanted dump already underway (for example, when a location 1000 was typed whereas location 100 was wanted) can be aborted by a USER BREAK. How this is accomplished varies from system to system. Typically, on single-user mini-computer systems, it is accomplished by raising the most significant bit of the host switch register.

3.3.3 ZERO THE AP (Z)

The Z command is legal only in APSIM. It zeros out all the registers and memories in APSIM. It should be the first command given to APSIM. It is accomplished by entering:

```
Z  (cr)
```

3.3.4 PREPARING DATA FILES FOR YANKING

Data files may be prepared by the user for loading into MD and TM by using APSIM (typically prepared by using the host system editor). The files may be prepared for loading into MD, SP, DPX, and DPY by using APDBUG. The format of the data file is as follows:

```
data count
data item #1
data item #2
... ... ...
data item #N
```

All entries must be left justified, one entry per line.

The data count is the number of memory locations to be filled and written as a real number (with a decimal point). Thus, if there were three data items, the count would be 3.

The format of data items depends on which debugger is used. In APSIM, only floating point numbers may be loaded. These must appear one per line in the data file. In APDBUG, the format is determined by the F command setting for 38-bit memories. For integer formats, the radix is determined by the N (radix) command. When floating point numbers are used they appear one per line. Also, integers must appear one per line in the file. Thus, for 38-bit memories written in integer format (F=0), three integers (exponent, high mantissa, low mantissa) written on three separate lines must be included for each memory location.

Examples:

- A four element floating point data file is entered.

```
4.
1.2
 .3
-6E7
2.3E-5
```

- This example illustrates three element integer data for a 38-bit wide memory. Three integers are loaded into the low mantissa. (This applies to APDBUG only.)

```
3.
0
0
1
0
0
2
0
0
3
```

3.3.5 EXECUTING PROGRAMS

AP program execution may be controlled with the following commands:

B set breakpoint

D delete breakpoint

L list breakpoint

Q set breakpoint counter

S set step mode

I initialize the AP

R run an AP program

P proceed (continue) with an AP program

T print elapsed execution time

M set memory speed

X exit to the host operating system

The typical strategy when debugging a program is to set a breakpoint at a strategic location in the program. Run the program. When it stops at the breakpoint, examine various data locations to see what has been changed correctly or incorrectly. This strategy typically results in alternately running a program and examining the results.

3.3.5.1 Set Breakpoint (B)

To set a breakpoint enter:

```
B               (cr)
(memory name)   (cr)
(location)      (cr)
```

(memory name) is the memory on which to break execution (must be MD, TM or PS) and (location) is the memory address on which to stop. The AP allows breakpointing on two conditions:

- read or write of a given Main Data memory or Table memory location
- execution of a given program instruction

Contrary to typical debuggers, the program halts after executing the breakpointed instruction. Only one breakpoint can be set at a time. Setting a new breakpoint clears any previously set breakpoint. Users of APDBUG should consult of the AP Processor Handbook for possible interaction between the breakpoint and the program.

Examples:

- Set a breakpoint so that the program stops after executing the instruction at location 20.

```
*
B   (cr)
PS  (cr)
20  (cr)
*
```

- Set a breakpoint so that the program stops after reading or writing Main Data location 100.

```
*
B    (cr)
MD   (cr)
100  (cr)
*
```

3.3.5.2 Delete Breakpoint (D)

To delete a breakpoint enter:

```
D  (cr)
```

This command clears any previously set breakpoints.

3.3.5.3 List Breakpoint (L)

To list at the terminal which breakpoint is currently set enter:

```
L  (cr)
```

APDBUG types the memory name in which the breakpoint is set, followed by the location of the breakpoint. If no breakpoint is set, only an asterisk (*) is typed.

Example:

- If a breakpoint is set at PS location 20, entering the L command results in the following:

```
*
L  (cr)
PS    000020
*
```

3.3.5.4 Set Breakpoint Counter (Q)

To set the breakpoint counter enter:

```
Q          (cr)
(count)    (cr)
```

(count) is the desired counter setting. The breakpoint counter is the number of times a breakpoint must be encountered before the AP program halts. It is also used by the step command. (Refer to Section 3.3.5.5.) For example, it is useful when a bug occurs every ten times around a loop. The count is reset to one every time a new breakpoint is set or the step flag is set or reset.

Example:

- Set a breakpoint at location 20 so that the program program halts only after encountering the breakpoint 10 times.

```
*
B     (cr)
PS    (cr)
20    (cr)
*
Q     (cr)
10    (cr)
*
```

3.3.5.5 Set/Reset Step Mode (S)

To set step mode enter:

```
S  (cr)
1  (cr)
```

To clear step mode enter:

```
S  (cr)
0  (cr)
```

In step mode, the program executes only a single instruction after being started and then halts. This is useful when sequencing step-by-step through a piece of code while watching for a data location to be destroyed or for the program to go awry. Step mode also uses the breakpoint counter which, in step mode, counts instructions to execute before stopping.

Examples:

- Set step mode so that when next started, the program executes one instruction and then stops.

```
*
S  (cr)
1  (cr)
*
```

- Set step mode so that when next started, the program executes 100 instructions and then stops.

```
*
S    (cr)
1    (cr)
*
Q    (cr)
100  (cr)
*
```

3.3.5.6 Initialize the AP (I)

To initialize (reset) the AP enter:

```
I  (cr)
```

In APSIM, the initialize command clears the memory, timing and arithmetic pipelines. In APDBUG, an interface reset is done to the AP. This is necessary to stop a program which is trapped in an infinite loop.

3.3.5.7 Run An AP Program (R)

To run an AP program enter:

```
R            (cr)
(location)   (cr)
```

(location) is the program location where execution starts. APDBUG starts the program at the specified location and then waits until the program encounters a breakpoint. If the program loops uncontrollably in APSIM, the user typically has no recourse. In APDBUG, control can be regained by causing a USER BREAK. (See Section 3.3.2.)

When the AP program finally halts, APSIM responds by printing out the current program address (PSA), the total elapsed program execution time after the last R command, and the contents of the currently open register or memory location. APDBUG merely responds with an asterisk (*).

Examples:

- Under APSIM, examine MA, set a breakpoint at program location 16, and then start program execution at location 10.

```
*
E   (cr)
MA  (cr)
123
*
B   (cr)
PS  (cr)
16  (cr)
*
R   (cr)
10  (cr)
PSA=000017          1.17 us.
MA
123
*
```

The program has executed four 1.17 us and has stopped with location 17 as the next instruction to be executed. MA has not changed. Note that the listing of the last examined location is useful for monitoring registers to see when they change.

- Under APDBUG, set a breakpoint at program location 16, then start execution at location 10.

```
*
B   (cr)
PS  (cr)
16  (cr)
*
R   (cr)
10  (cr)
*
```

APDBUG signals program return by outputting the *.

3.3.5.8 Proceed With AP Program Execution (P)

To proceed with AP program execution enter:

```
P  (cr)
```

This command is used to resume AP program execution after encountering breakpoint or when using step mode. The program continues execution at the address currently in the Program Source Address register (PSA). When the program next encounters a breakpoint, output is the same as that which follows a return from a run.

Examples:

- Set a breakpoint at location 16, run at location 10, examine S-Pad 3, and then continue execution.

```
*
B   (cr)
PS  (cr)
16  (cr)
*
R   (cr)
10  (cr)
PSA=000007          1.17 us.
*
E   (cr)
SP  (cr)
3   (cr)
000123
*
P   (cr)
```

- Examine MA. Then watch it change as the program is stepped starting at location 10.

```
*
S    (cr)
1    (cr)
*
E    (cr)
MA   (cr)
000103
*
R    (cr)
10   (cr)
PSA=000011          0.17 us.
MA
000104
*
P    (cr)
PSA=000012          0.33 us.
MA
000105
*
P    (cr)
PSA=000013          0.50 us.
MA
000106
*
```

3.3.5.9 Execution Time (T)

To print elapsed AP program execution time up to the last run (R) command (APSIM only) enter:

```
T  (cr)
```

3.3.5.10 Set Memory Speed (M)

To set Main Data Memory speed (APSIM only) enter:

```
M        (cr)
(speed)  (cr)
```

where (speed) is 1 for fast memory timing and 2 for standard memory timing. The default is 2 for standard memory timing.

3.3.5.11 Exit To The Host System (X)

To complete a debugging session and exit to the host operating system enter:

```
X  (cr)
```

3.4 SUMMARY OF APDBUG COMMANDS

This section summarizes the APDBUG commands.

3.4.1 ABBREVIATIONS

Abbreviations used in the following appendixes:

symbol	meaning
(cr)	carriage return
(loc)	an integer location number
(count)	an integer count
(val)	an integer value
(fpn)	a floating point number in form acceptable to FORTRAN
(mem)	the name of an AP internal memory
(reg)	the name of an AP internal register

APDBUG outputs an asterisk (*) when ready for further action. A question mark (?) is output when a command is not understood.

3.4.2 PROGRAM EXECUTION COMMANDS

B	(cr)	Breakpoint. Delete the last breakpoint
(mem)	(cr)	and set a new breakpoint at location (loc)
(loc)	(cr)	of memory (mem). (mem) must be PS, MD, or TM.
D	(cr)	Delete. Delete the current breakpoint.
L	(cr)	List. List the current breakpoint.
Q	(cr)	Set the continue counter to (count).
(count)	(cr)	
S	(cr)	Step. If (val) is not zero, place the
(val)	(cr)	AP in step mode.
I	(cr)	Initialize. Reset the AP before program execution is resumed next.
R	(cr)	Run. Begin program execution at
(loc)	(cr)	Program Source location (loc).
P	(cr)	Proceed. Begin instruction execution at the Program Source location pointed to by the AP PSA (Program Source Address) register.
T	(cr)	Print out elapsed execution time (APSIM only).
X	(cr)	Exit to the operating system.
M	(cr)	Set memory speed. val is 1 for one cycle (fast) memory and 2 for the two cycle
(val)	(cr)	(standard) memory (APSIM only).

3.4.3 REGISTER EXAMINATION/MODIFICATION COMMANDS

E (reg)	(cr) (cr)	Examine register. Print out the contents of AP register (reg).
E (mem) (loc)	(cr) (cr) (cr)	Examine memory. Print out the contents of AP memory (mem), location (loc).
.	(cr)	Re-examine the currently open register or memory location (the last location examined).
+	(cr)	Examine the next higher sequential memory location of the memory that is currently open.
-	(cr)	Examine the next lower sequential memory location of the memory that is currently open.
F (val)	(cr) (cr)	Floating point flag, affects the input/output of 38-bit wide registers and memory locations. (val)=0: 3 integers (exponent, high mantissa, low mantissa): (val)=1: floating point.
V (val)	(cr) (cr)	Program source field value flag, affects input/output of program source memory location. (val)=0: 4 integers (the four 16-bit quarters of PS) (val)=1: Decode into the 24 instruction word field values.

C (cr)
(val) (cr)

Change. Change the contents of the currently open register or memory location to (val). The format of (val) depends on the width of the current open locations as follows:

16-bit wide registers: an integer of the current radix.

38-bit wide registers:

F=0; (val) (cr) three integers in the current
(val) (cr) radix which represent
(val) (cr) the exponent, high mantissa, and low mantissa

F=1: (fpn) (cr) a floating point number legal to FORTRAN

64-bit wide registers:

V=0: (val) (cr) four integers in the current
(val) (cr) radix which are the four
(val) (cr) quarters of an AP program word
(val) (cr)

V=1: (field)(cr) (field) is the name of the instruction field to be changes
(val) (cr) (val) is the new integer value

N (cr)
(val) (cr)

Number radix. Set the radix for integer user I/O to (val), which must be 8 (for octal), 10 (for decimal), or 16 (for hexadecimal).

O (cr)
(val) (cr)

Offset. Sets the base address to which program source memory addresses are relative (for user I/O).

Z (cr)

Zero. Zero out all AP memories and registers. (APSIM only)

3.4.4 MEMORY LOAD/DUMP COMMANDS

Y (cr) (memory) (cr) (loc) (cr) (filename) (cr)	Yank:	Load memory starting at (loc). from an external file.
W (cr) (mem) (cr) (start) (cr) (stop) (cr) (filename) (cr)	Write:	Dump memory starting at location (start) and ending at location (stop) to external data (filename). (mem) can be PS, MD or TM.

3.4.5 ACCESSIBLE FUNCTIONAL UNITS

This section lists the AP functional units that can be examined or changed with APDBUG.

3.4.5.1 Memories

			Accessible from		Can 'Y'or'W'	
Mnemonic	Name	Width	APSIM	APDBUG	APSIM	APDBUG
PS	Program Source memory	64	yes	yes	yes	yes
MD	Main Data memory	38	yes	yes	yes	yes
TM	Table Memory	38	yes	(read only)	yes	yes
DPX	Data Pad X	38	yes	yes	no	yes
DPY	Data Pad Y	38	yes	yes	no	yes
IO	I/O devices	38	yes	no	no	no
SP	S-Pad	16	yes	yes	no	no
SRS	Subroutine Return Stack	16	yes	no	no	no
			readable and writeable from terminal		readable and writeable from files using "Y" and "W"	

3.4.5.2 Registers

Mnemonic	Name	Width	Accessible from: APSIM	APDBUG
MA	Main Data Address	16	yes	yes
TMA	Table Memory Address	16	yes	yes
DPA	Data Pad Address	6	yes	yes
PSA	Program Source Address	12	yes	yes
SPD	S-Pad Destination Addr.	4	yes	yes
STAT	AP Status Register	16	yes	yes
DA	I/O Device Address	6	yes	yes
SPFN	S-Pad Function	16	yes	yes
SWR	Switch register	16	no	yes
FN	Function register	16	no	yes
LITE	Lites register	16	no	yes
APMA	AP DMA Memory Address	16	no	yes
HMA	Host DMA Memory Address	16	no	yes
WC	DMA Word Count	16	no	yes
CTL	DMA Control register	16	no	yes
FMTH	Formatter high	16	no	yes
FMTL	Formatter low	16	no	yes
IFRS	Interface reset	16	no	yes
IFSTAT	Interface status	16	no	yes (when present)
MDR	Main Data Read Buffer	38	yes	no
TMR	Table Memory Read Buff.	38	yes	no
MI	Main Data Input Buff.	38	yes	no
DPBS	Data Pad Bus	38	yes	no
INBS	I/O Input Bus	38	yes	no
PNBS	Panel Bus	16	yes	no
FLAG	Program flags	4	yes	no
SRA	Subroutine Stack Addr.	4	yes	no
A1	Floating Adder #1 input	38	yes	no
A2	Floating Adder #2 input	38	yes	no
M1	Multiplier #1 input	38	yes	no
M2	Multiplier #2 input	38	yes	no
FA	Floating Adder output	38	yes	no
FM	Floating Multiplier out.	38	yes	no

3.4.6 PROGRAM WORD FIELDS

The following are fields within an instruction word that may be examined or changed by name.

Name	Program Word Bits
D	0
SOP	1-3
SH	4-5
SPS	6-9
SPD	10-13
FADD	14-16
A1	17-19
A2	20-22
COND	23-26
DISP	27-31
DPX	32-33
DPY	34-35
DPBS	36-38
XR	39-41
YR	42-44
XW	45-47
YW	48-50
FM	51
M1	52-53
M2	54-55
MI	56-57
MA	58-59
DPA	60-61
TMA	62-63
SOP1	6-9
SPEC	6-9
STST	10-13
HPNL	10-13
SPSA	10-13
PSEV	10-13
PSOD	10-13
PS	10-13
SEXT	10-13
FAD1	17-19
IO	17-19
LREG	20-22
RREG	20-22
IOUT	20-22
SNSE	20-22
FLAG	20-22
CONT	20-22

3.5 AN EXAMPLE DEBUGGING SESSION

In this example an AP program called PROG1 is being debugged. PROG1 uses the FPS supplied divide routine to divide two numbers in Main Data memory.

3.5.1 APAL SOURCE PROGRAM

```
        $TITLE PROG1
        $ENTRY PROG1,3
        $EXT DIV

"PROG1 DOES A SCALAR DIVIDE OF TWO NUMBERS IN MAIN DATA
"MEMORY AND RETURNS THE ANSWER BACK INTO MAIN DATA MEMORY
"C <= A / B
"
"S-PAD PARAMETER DEFINITIONS:
"
        A $EQU 0        "ADDRESS OF 'A' IN MAIN DATA MEMORY
        B $EQU 1        "ADDRESS OF 'B' IN MAIN DATA MEMORY
        C $EQU 2        "ADDRESS OF 'C' IN MAIN DATA MEMORY
"
PROG1: MOV A,A; SETMA  "FETCH A
        MOV B,B; SETMA  "FETCH B
        NOP             "WAIT
        DPY<MD          "SAVE A IN DPY
        DPX<MD;         "SAVE B IN DPX
           JSR DIV      "   AND DIVIDE A/B
        MOV C,C; SETMA; "STORE ANSWER IN C
           MI<DPY; RETURN "AND RETURN
        $END
```

The input parameters are the addresses of scalars A, B, and C. A and B are fetched from Main Data memory, A is divided by B, and the result stored into C. The $ENTRY declaration tells APAL that the routine's name is PROG1. The ",3" tells APAL that the routine is FORTRAN callable through APEX and has three S-Pad parameters (the addresses of A, B, and C). The $EXT tells APAL that the routine uses the DIV routine.

3.5.2 ASSEMBLING THE PROGRAM USING APAL

```
RUN APAL    (cr)
 SOURCE FILE=
PROG1.AP    (cr)
 OBJECT FILE=
PROG1.OBJ    (cr)
 LISTING AND ERROR FILE=
PROG1.LST    (cr)
 LISTING? 1=YES, 0=NO:
1    (cr)
 LISTING RADIX: 1=HEX, 0=OCTAL:
0    (cr)
     0 ERRORS: PROG1  APAL-REV 2
```

File PROG1.AP is assembled with the object code placed on file PROG1.OBJ, and the listing (reproduced below) placed on file PROG1.LST.

```
 APAL, REV 2
 PASS 1
 PASS 2
                          $TITLE PROG1
                          $ENTRY PROG1,3
                          $EXT DIV

              "PROG1 DOES A SCALAR DIVIDE OF TWO NUMBERS IN MAIN DATA
              "MEMORY AND RETURNS THE ANSWER BACK INTO MAIN DATA MEMORY
              "C <=A / B
              "
              "S-PAD PARAMETER DEFINITIONS:
              "
       000000       A $EQU 0        "ADDRESS OF 'A' IN MAIN DATA MEMORY
       000001       B $EQU 1        "ADDRESS OF 'B' IN MAIN DATA MEMORY
       000002       C $EQU 2        "ADDRESS OF 'C' IN MAIN DATA MEMORY
       000000       A $EQU 0        "ADDRESS OF 'A' IN MAIN DATA MEMORY
       000001       B $EQU 1        "ADDRESS OF 'B' IN MAIN DATA MEMORY
       000002       C $EQU 2        "ADDRESS OF 'C' IN MAIN DATA MEMORY

                    "
 000000  040000  PROG1: MOV A,A; SETMA  "FETCH A
         000000
         000000
         000060

 000001  040104          MOV B,B; SETMA  "FETCH B
         000000
         000000
         000060

 000002  000000          NOP             "WAIT
```

```
         000000
         000000
         000000

000003   000000          DPY<MD          "SAVE A IN DPY
         000000
         015000
         100000

000004   011014          DPX<MD;         "SAVE B IN DPX
         000000             JSR DIV      "   AND DIVIDE A/B
         045004
         177777

000005   040210          MOV C,C; SETMA; "STORE ANSWER IN C
         000340             MI<DPY; RETURN "AND RETURN
         004040
         000360

                         $END


****      0 ERRORS    ****


SYMBOL   VALUE
DIV      000004 EXT
A        000000
B        000001
C        000002
PROG1    000000 ENT
```

3.5.3 LINKING THE PROGRAM USING APLINK

```
RUN APLINK (cr)

APLINK

REV 2

* L (cr) PROG1.OBJ (cr)

* L (cr) APLIB (cr)

LOAD COMPLETE

* S (cr) TTY (cr)

HIGH=000041


SYMBOL TABLE

SYMBOL VALUE

PROG1 000000

DIV 000006

* E (cr) PROG1.ABS (cr)

PROG1 HIGH=000041

* X (cr)
```

First PROG1.OBJ is loaded. Then the FPS supplied object library (called APLIB) is loaded in. APLINK picks out DIV from the library. When the loading is done, a symbol table (load map) is printed at the terminal. It shows that PROG1 is loaded at location 0, and DIV at location 7. The high location loaded was 41 (octal), which means that the total size of the load module is 34 (decimal) program words. The E command is used to output the load module for APSIM (or APDBUG). It is placed on file PROG1.OBJ.

3.5.4 DEBUGGING THE PROGRAM USING APSIM

```
RUN APSIM    (cr)
 APSIM
 REV 2.0
 *
Z    (cr)
 *
Y    (cr)
PS    (cr)
0    (cr)
PROG1.ABS    (cr)
 *
```

Zero the simulator and yank in the load module.

```
E    (cr)
SP    (cr)
0    (cr)
 000000
 *
C    (cr)
2    (cr)
 *
+    (cr)
 SP     000001
 000000
 *
C    (cr)
3    (cr)
 *
+    (cr)
 SP     000002
 000000
 *
C    (cr)
4    (cr)
 *
```

Set up S-Pads 0, 1, and 2 with the addresses of A, B, and C. These are chosen here to be 2, 3, and 4, respectively.

```
E   (cr)
MD  (cr)
2   (cr)
  0.0000000000
 *
C   (cr)
10.0   (cr)
 *
+   (cr)
 MD    000003
  0.0000000000
 *
C   (cr)
2.0   (cr)
 *
```

Set A (in MD 2) to 10.0, and B (in MD 3) to 2.0. The result should be 5.0.

```
E   (cr)
PS  (cr)
5   (cr)
 040210  000340  004040  000360
 *
B   (cr)
PS  (cr)
6   (cr)
 *
```

Examine PS location 5. It appears to be the last instruction of PROG1, so set the breakpoint there.

```
R   (cr)
0   (cr)
 PSA=000000        5.00 US.
 PS    000005
 040210  000340  004040  000360
 *
```

Run the program. It took 5.0 us to reach the breakpoint. PSA is set to 0 since the program stopped on a RETURN, which returns to location 0 since the Subroutine Return Stack was zeroed by the Z command. PS location 5 is listed because it is the last location examined.

```
E   (cr)
MD  (cr)
4   (cr)
   20.00000000
 *
```

However, the answer is wrong. 10.0, not 20.0 should be the value of C (MD location 4).

```
E   (cr)
DPY   (cr)
0   (cr)
   20.00000000
 *
```

DPY (where it appeared that DIV returned the answer) also contains 20.0, not 10.0. Now examine DPX, since this might be the data pad in which DIV returns the answer.

```
E   (cr)
DPX  (cr)
0   (cr)
   5.000000000
 *
```

Yes, the answer is in DPX instead of DPY. A quick check of the AP Math Library Manual confirms that DIV returns the result in DPX, and that DPY is used as a scratch location. Now correct the program so that in PS location 5, MI<DPX appears instead of MI<DPY.

```
E   (cr)
PS  (cr)
5   (cr)
 040210  000340  004040  000360
 *
V   (cr)
1   (cr)
 *
 PS     000005
    D     00    SOP   04    SH    00    SPS   02    SPD   02    FADD  00
    A1    00    A2    00    COND  07    DISP  00    DPX   00    DPY   00
    DPBS  04    XR    00    YR    04    XW    00    YW    00    FM    00
    M1    00    M2    00    MI    03    MA    03    DPA   00    TMA   00
 *
C   (cr)
DPBS   (cr)
3   (cr)
 *
C   (cr)
XR  (cr)
4   (cr)
 *
 PS     000005
    D     00    SOP   04    SH    00    SPS   02    SPD   02    FADD  00
    A1    00    A2    00    COND  07    DISP  00    DPX   00    DPY   00
    DPBS  03    XR    04    YR    04    XW    00    YW    00    FM    00
    M1    00    M2    00    MI    03    MA    03    DPA   00    TMA   00
 *
```

Examine PS location 5 and switch the PS output mode (V) to 1 for PS opcode field mode. It shows that now the MI field (memory input) is set to 3 for data pad bus, and that the DPBS field is set to 4 for DPY. Change the DPBS field to 3 for DPX, and the DPX read address (XR) to 4, which biased by 4 gives the proper DPX address of 0. Determine that the proper corrections were made, reinitialize APSIM, and run again.

```
I   (cr)
 *
R   (cr)
0   (cr)
 PSA=000000          5.00 US.
 PS     000005
    D     00    SOP  04    SH   00    SPS  02    SPD  02    FADD 00
    A1    00    A2   00    COND 07    DISP 00    DPX  00    DPY  00
    DPBS  03    XR   04    YR   04    XW   00    YW   00    FM   00
    M1    00    M2   00    MI   03    MA   03    DPA  00    TMA  00
 *
E   (cr)
MD  (cr)
4   (cr)
   5.000000000
 *
```

The correct answer is now contained in C.

```
-   (cr)
 MD     000003
   2.000000000
 *
C   (cr)
-3.4   (cr)
 *
-   (cr)
 MD     000002
   10.00000000
 *
C   (cr)
12.5   (cr)
 *
```

Now try a different case where B = -3.4, and A = 12.5. The expected answer is approximately -3.68.

```
I    (cr)
 *
R    (cr)
0    (cr)
 PSA=000000          5.00 US.
 MD     000002
    12.50000000
 *
E    (cr)
MD   (cr)
4    (cr)
   -3.676470578
 *
```

Reinitialize and rerun, to make sure the correct answer is obtained.

X (cr)

Now exit APSIM, edit the change into the source, and reassemble, link, and simulate to make sure the program is correct.

Notice to the Reader

- Help us improve the quality and usefulness of this manual.

- Your comments and answers to the following READERS COMMENT form would be appreciated.

To mail: fold the form in three parts so that Floating Point Systems' mailing address is visible; seal.

Thank you

First Class
Permit No. A-737
Portland,
Oregon

BUSINESS REPLY

No postage stamp necessary if mailed in the United States

Postage will be paid by:

FLOATING POINT SYSTEMS, INC.

P. O. Box 23489

Portland, Oregon 97223

Attention: Technical Publications

READERS COMMENT FORM

Document Title _____

Your comments and answers will help us improve the quality and usefulness of our publications. If your answers require qualification or additional explanation, please comment in the space provided below.

How did you use this manual?

( ) AS AN INTRODUCTION TO THE SUBJECT
( ) AS AN AID FOR ADVANCED TRAINING
( ) TO LEARN OF OPERATING PROCEDURES
( ) TO INSTRUCT A CLASS
( ) AS A STUDENT IN A CLASS
( ) AS A REFERENCE MANUAL
( ) OTHER _____

Did you find this material . . .

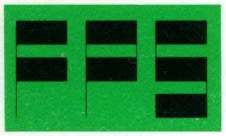
	YES	NO
• USEFUL?	( )	( )
• COMPLETE?	( )	( )
• ACCURATE?	( )	( )
• WELL ORGANIZED?	( )	( )
• WELL ILLUSTRATED?	( )	( )
• WELL INDEXED?	( )	( )
• EASY TO READ?	( )	( )
• EASY TO UNDERSTAND?	( )	( )

Please indicate below whether your comment pertains to an addition, deletion, change or error; and, where applicable, please refer to specific page numbers.

Page	Description of error or deficiency

From:

Name _____ Title _____
Firm _____ Department _____
Address _____ City, State _____
Telephone _____ Date _____



FLOATING POINT
SYSTEMS, INC.

CALL TOLL FREE 800-547-1445
P.O. Box 23489, Portland, OR 97223
(503) 641-3151, TLX: 360470 FLOATPOINT PTL