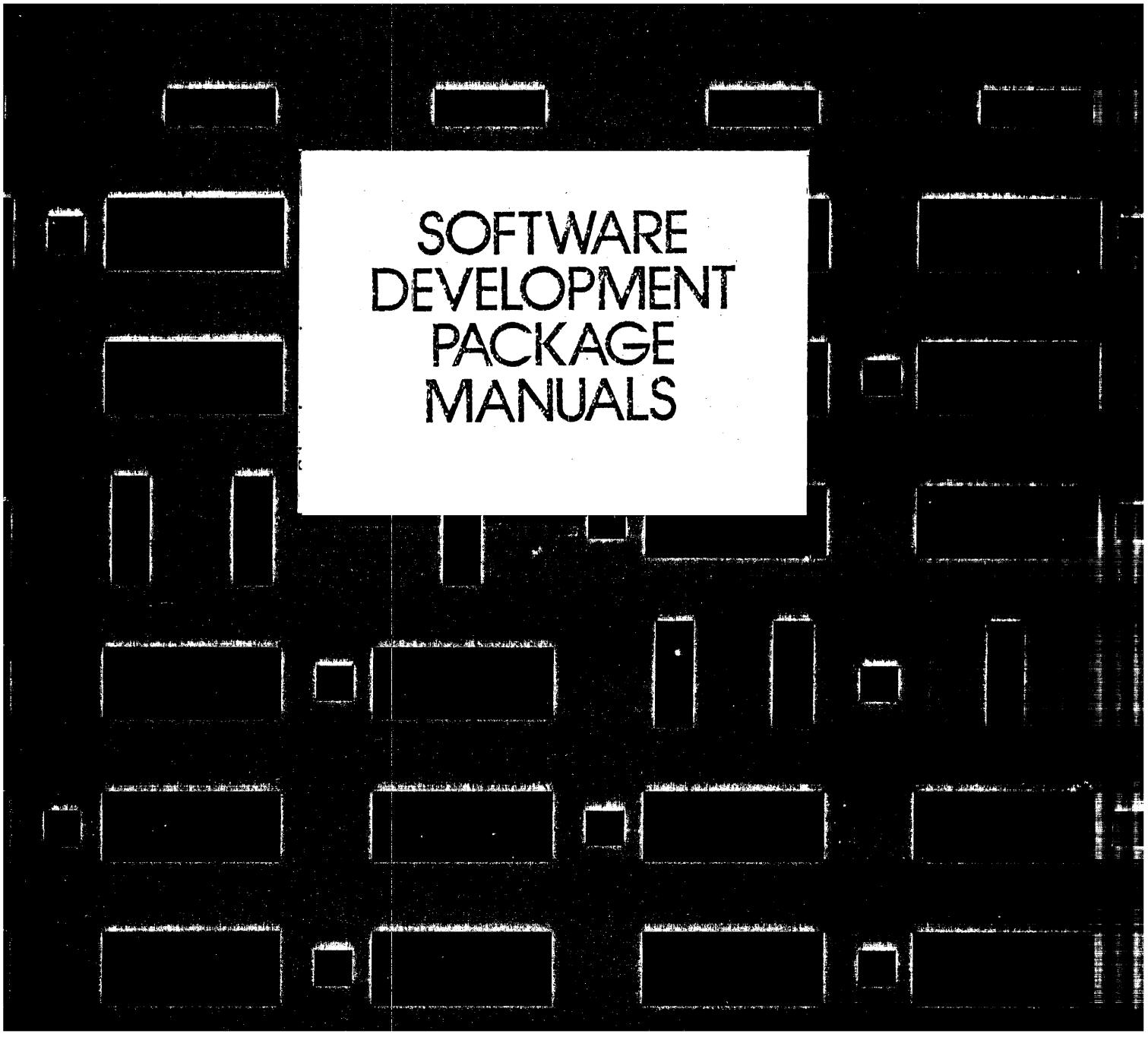
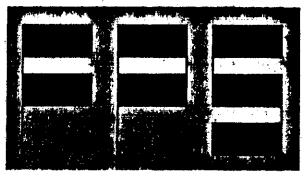
FLOATING POINT
SYSTEMS, INC.

AP-120B

ARRAY TRANSFORM PROCESSOR





FLOATING POINT
SYSTEMS, INC.

SOFTWARE DEVELOPMENT PACKAGE MANUALS

P.O. BOX 23489 PORTLAND, OR 97223 11000 S.W. 11TH STREET, BEAVERTON, OR 97005 (503) 641-3151 TLX: 360470 FLOATPOINT PTL

SOFTWARE DEVELOPMENT PACKAGE MANUALS

FPS-7292

This document contains four (4) manuals:

1. HOW TO PROGRAM THE AP-120B

   Manual -

   FPS-7303

   March 1976

2. AP-120B APAL - Array Processor Assembly Language

   Manual -

   FPS-7275-01

   Revised February 26, 1976

3. AP-120B APLINK - Array Processor Linking Loader

   Manual -

   FPS-7276-01

   Revised Feburary 26, 1976

4. AP-120B DEBUG - Array Processor Debugger

   Manual -

   FPS-7277-01

   Revised February 26, 1976

TABLE OF CONTENTS

Documentation Update

Abstract: These pages reflect program changes in Software Update #1 and replace corresponding pages in the Software Development Package Manuals. Changed or newly added material is identified by a bar along the outside margin of the page.

Manuals Affected:

APAL #7275-01 February 26, 1976

Pages: 4-1
4-1a
4-2

APLINK #7276-01 February 26, 1976

Pages: i
3-1
3-1a
3-4
3-5
3-5a
3-6
3-7
A-1

APDEBUG #7277-01 February 26, 1976

Pages: A-1

HOW TO
PROGRAM THE AP-120B
FPS-7303

FPS-7303

PUBLISHED IN THE UNITED STATES OF AMERICA
MARCH 1976

TABLE OF CONTENTS

SECTION 1
MEET THE AP.....AGAIN

1.1 INTRODUCTION

The purpose of this manual is to illustrate the way to use the AP most efficiently, i.e., to write good loops. It assumes that the reader has already read the Processor Handbook (especially Section 3) and the Software Development Package Manual (APAL, Sections 2 and 3), and has at least a passing acquaintance with the AP instruction set.

The first section presents a short review of the basic elements of the Array Processor from the programmer's point of view. The second section covers methods and techniques of writing loops. The third section consists of a set of common pitfalls to avoid.

The review in this section of the basic AP instructions is not meant to be all-inclusive. It is intended to briefly cover the most-often-used things. Further details can be found in the other manuals.

This manual assumes the use of the AP's 333ns interleaved memory.

1.2 BASIC OVERVIEW

1.2.1 Arithmetic. Both the Floating Adder and Floating Multiplier need explicit instructions (e.g., FADD and FMUL, respectively) to push their respective answers out of the pipelines. Given these "pushers", the Floating Adder result (FA) will be available 2 cycles after the original instruction, and the Floating Multiplier result (FM) will be available 3 cycles after the original instruction:

```
0.  FADD DPX,DPY  "add                0.  FMUL DPX,DPY  "multiply
1.  FADD          "push               1.  FMUL          "push
2.  DPX(1)<FA     "store answer       2.  FMUL          "push
                                      3.  DPY(1)<FM     "store answer
```

The empty FADD and FMUL "pushers" can also be real Adder or Multiplier operations, thus producing new answers each cycle.

If the "pushers" do not directly follow the original instructions, FA will come out 1 cycle after the first FADD pusher, and FM will come out 1 cycle after the second FMUL pusher. Both FA and FM will remain available for succeeding cycles until a new FA or FM is pushed out.

The arguments for Adder and Multiplier instructions consist of one from column A and one from column B, (in that order):

COLUMN A (A1 or M1)	COLUMN B (A2 or M2)
FM	FA
TM	MD
DPX	DPX
DPY	DPY

The Adder has additional arguments of ZERO and NC (no change), which can be used in either or both columns.

1.2.2 Main Data Memory. Reading from memory requires one of the following instructions: SETMA, INCMA, DECMA, or LDMA. In practice, it is usually done by the SETMA instruction. The result, MD, comes out three cycles later and is also available for succeeding cycles until a new MD comes out. No "pushers" are needed. Writing into memory requires one of the above instructions plus MI<source, where source is FA, FM, or DB. This goes on the same line as SETMA, and gets done in that cycle. Memory can be referenced every 2 cycles, for either a read or write.

1.2.3 Table Memory. Table memory is usually referenced by the SETTMA or LDTMA instruction. Two cycles later, TM is available and remains so until 2 cycles after the next instruction affecting TMA. Such instructions can occur in every cycle, producing a new TM every cycle.

1.2.4 Data Pad. DPX and DPY each contain 32 registers, 8 of which are accessible from any given DPA. That is, one can reference DPX from DPX(DPA-4) to DPX(DPA+3), and similarly for DPY.

The Data Pad Bus is usually used to store data from memory or from one Data Pad register into another, or to utilize a value, e.g., in conjunction with a load operation:

```
DPX(1)<DB; DB=DPY(-2)          (This can be shortened to
                                DPX(1)<DPY(-2).)

DPX<D3; DB=MD                  (Or DPX<MD)

LDDPA; DB=3                    (This sets DPA=3)
```

Storing into Data Pad from FA or FM does not use the Data Pad Bus. This is important, as it leaves DB free for other uses.

1.2.5 S-Pad. S-Pad registers are usually used as address pointers or counters, and thus to pass parameters to a program. An S-Pad operation must accompany a SETMA (or SETDPA, SETTMA, etc.) instruction. An S-Pad operation must also precede a conditional branch (BGT, BNE, etc.) by one cycle. That is, conditional branches are based on the S-Pad Function (SPFN) of the S-Pad operation in the previous cycle.

The fastest way to get an integer into S-Pad is to use the LDSPI instruction:

```
LDSPI COUNT; DB=5
```

This puts 5 into an S-Pad register called COUNT. The value is assumed to be octal unless a decimal point is added.
DB=15. (note point) is equivalent to DB=17 (octal), or to DB=ØFX (hex). Hexadecimal numbers must start with a numeric digit and end with "X".

Although the Floating Adder operation FSUB A1,A2 will do A1-A2, the S-Pad operation SUB subtracts in the opposite direction, i.e., SUB PIECE,TOTAL will do:
(contents of S-Pad TOTAL) minus (contents of S-Pad PIECE).

1.3 REFERENCING MEMORY

In order to read something out of memory, or write into it, the location in memory where this will occur must be provided. The SETMA instruction gets this necessary information from the S-Pad Function (SPFN) of the same cycle. Therefore, one needs to construct an S-Pad operation which will result in a pointer to the appropriate memory location. Generally this takes the form of adding increments to pointers. For example, if there was a 4-element vector in memory locations 100, 102, 104, 106, one would need an S-Pad register (say, APTR) containing the base address (100), and another S-Pad register (AINC) containing the increment between elements (2). Then, if one wanted to read the element in location 102, the appropriate instruction would be ADD AINC,APTR; SETMA. Now APTR would contain 102. If one wrote another ADD AINC,APTR; SETMA the contents of memory location 104 would be read.

Consider the following instruction: MOV APTR,APTR. This doesn't seem to accomplish much, but in the light of the above discussion, it can be seen that its SPFN could be useful for a SETMA. This is how one would get the first element of a vector.

All of the above is correspondingly true for writing into memory.

1.4 S-PAD MNEMONICS

S-Pad names such as APTR, AINC, N are really only temporary names for the 16 S-Pad registers. A statement such as DEC N will not mean anything to the assembler unless the program has equated the mnemonic "N" with a specific S-Pad register, such as S-Pad 0. This is done by the following assembler pseudo-op: N $EQU 0. All S-Pad names used in a program must be declared in this manner before using them in an instruction. Thus, programs generally begin with lists like:

```
APTR $EQU 0
AINC $EQU 1
BPTR $EQU 2
BINC $EQU 3
N    $EQU 4
      .
      .
      .
      .
```

These S-Pad numbers should not be confused with the contents of the S-Pads. ADD BINC,BPTR would not add 3 to 2 (using the above list), but would add the contents of S-Pad 3 to the contents of S-pad 2.

There can be more than one name for an S-Pad register. If you had 2 different vectors, A and B, and wished to use the mnemonics AINC and BINC for their increments, you could use the same S-Pad register if the increment for both is the same in all cases, by declaring:

```
AINC $EQU 1
BINC $EQU 1
```

1.5 OTHER PSEUDO-OPS

Besides the $EQU pseudo-op, the typical program includes $TITLE and $ENTRY pseudo-ops at the very beginning, and an $END at the very end. A basic program with one loop would have the following form:

```
                          $TITLE name
                          $ENTRY name
          S-Pad mnemonic    $EQU  Ø
                  .           .    1
                  .           .    2
                  .           .    .
                  .           .    .
                  .           .    .
name:  (code)
         "
         "      ("intro" to loop and any initializations
         "       and pointer adjustments)
         "

loop:  (code)
         "
         "
         "

                          $END
```

See the software manual for explanations of these pseudo-ops.

SECTION 2
LOOPS

2.1 A POOR LOOP

The loop is where the potential of the AP comes into full bloom. For example, one way (lengthy but workable) to write a dot product program is as follows:

Given: Vectors A and B in Main Data memory, with elements of each vector in equally spaced locations in memory (e.g. even-numbered locations).

Produce: $c = \sum_{m=1}^{N} A_m \cdot B_m$

Parameters passed in S-Pad:

S-Pad Name	Contains:
APTR	base address of vector A
BPTR	base address of vector B
XINC	increment (number of locations from one element to the next) (same for both vectors)
N	number of elements in each vector
CPTR	address of answer

```
DOTPROD:SUB XINC,APTR (*see below)
        SUB XINC,BPTR (*see below)
        FADD ZERO,ZERO          "initialize FA=0
        FADD
LOOP:   ADD XINC,APTR; SETMA    "get mth element of vector A
        NOP                     "                  from memory
        NOP
        DPX<MD                  "MD=Am, store into DPX
        ADD XINC,BPTR; SETMA    "get mth element of vector B
        NOP
        NOP
        FMUL DPX,MD             "MD=Bm, do Am.Bm
        FMUL
        FMUL
        FADD FM,FA              "add product to sum of products
        FADD
        DEC N                   "decrement counter
        BGT LOOP                "branch back if not done yet
                                "(i.e. if N>0)
DONE:   MOV CPTR,CPTR; SETMA;   MI<FA
                                "otherwise, store answer
```

*This is so that the first time through the loop, ADD XINC,APTR and ADD XINC,BPTR will not move the pointer to the second element, passing up the first altogether.

To begin with, this program can certainly be shortened by combining instructions and overlapping memory fetches. Thus:

```
DOTPROD:  FADD ZERO,ZERO; SUB XINC,APTR
          FADD; SUB XINC,BPTR
LOOP:     ADD XINC,APTR; SETMA     "get Am
          NOP
          ADD XINC,BPTR; SETMA     "get Bm
          DPX<MD                   "store Am in DPX
          NOP
          FMUL DPX,MD              "do Am.Bm
          FMUL
          FMUL
          FADD FM,FA; DEC N        "add prod to sum of products
                                   "  and decrement counter
          FADD; BGT LOOP           "test if done.  If not, branch
                                   "                    to LOOP
DONE:     MOV CPTR,CPTR; SETMA; MI<FA
                                   "if so, store answer
```

Note the extra FMUL's and FADD's, described as "pushers." These push the answers through the pipelines, so that FM and FA will contain what they are intended to contain. This is pointed out because the beginning AP programmer is likely to forget to put "pushers" in his code.

Now the loop of the evolving dot product program is 10 cycles long. This means that each new pair of elements costs 10 more cycles. Although better than the initial example, which had a 14-cycle loop, this can actually be cut down to a mere 4 cycles!

2.2 DETERMINING LENGTH OF LOOP

One might suppose that the length of a program loop depends on what one is trying to do. This is true, but not in the way one would think. The AP programmer decides ahead of time how many cycles his loop should contain, and then fits everything into that framework. How does he pick the magic number? Most commonly, loops are memory-limited. Recall that one can reference memory (to read or to store) every 2 cycles. If one has 2 memory references to do (e.g., "get A" and "get B"), then the loop will be at least 4 cycles long (2 per memory reference). And, unless one has more than 4 different FMUL's, 4 different FADD's, or 4 different S-Pad operations to do, the loop should be, at most, 4 cycles. A lot can be done in 4 cycles when one can do a Floating Multiplier operation, a Floating Adder operation, an S-Pad operation, a branch, a memory reference, a Data Pad Bus transfer, etc., in each cycle.

2.3 WRITING A REAL MEMORY-LIMITED LOOP

Before continuing with the transformation of the dot product program, another example will be utilized.

Given: Vectors A and B in Main Data memory, length=N elements
Produce: Vector C (in memory), where

$$C_m = A_m^2 + B_m \quad \text{for } m=1 \text{ to } N$$

Parameters:

S-Pad Name	Contains
APTR	base address of A
BPTR	base address of B
CPTR	base address of C
XINC	increment (same for all vectors in this example)
N	number of elements

Note that there should be 3 memory references in the loop: "get A", "get B", and "store C". (Unlike the dot product which accumulated a running sum in the Adder, this program needs to store an answer after each set of computations. For the dot product, storing was not a repeated process, and hence not included in its loop). Three memory references, one every other cycle, means the loop would be 6 cycles long. It would start like this:

```
1)  ---(nothing here, but count a cycle)
2)  ADD XINC,APTR; SETMA     "get A
3)  ---
4)  ADD XINC,BPTR; SETMA     "get B
5)  DPX<MD                   "store A in DPX
6)  FMUL DPX,MD              "do A*A
```

(The reason for starting on the second line will be explained later.)

Now it has run out of cycles, but there is still more to do, so it starts back up at the first cycle, which is where the end will branch to, when it gets around to testing if it's done.

LOOP:	1) ---	FMUL	"B is available here, but "not needed yet
	2) ADD XINC,APTR; SETMA	FMUL	
	3) ---	FADD FM,MD	"add B to A^2
	4) ADD XINC,BPTR; SETMA	FADD	
	5) DPX<MD	DEC N	"answer is available here "but can't reference mem. "yet to store it
	6) FMUL DPX,MD	ADD XINC,CPTR; SETMA; MI<FA; BGT LOOP	"store answer and "test if done

This is the entire loop. In its proper form, taking out lines and adding semicolons, it looks like this:

```
LOOP: FMUL
      ADD XINC,APTR; SETMA; FMUL
      FADD FM,MD
      ADD XINC,BPTR; SETMA; FADD
      DPX<MD; DEC N
      FMUL DPX,MD; ADD XINC,CPTR; SETMA; MI<FA; BGT LOOP
```

2.4 WRITING INTROS

Notice, however, that if the program goes right into this loop, after initial overhead such as SUB XINC,APTR
SUB XINC,BPTR
SUB XINC,CPTR
it picks up the first element of A and B as it's supposed to, but it also stores something into C before it's ready to, and decrements the counter too early. It goes through both columns at the same time. What is desired, however, is that computations in the second column continue from the first column. The only way it can do this is to continue from what the first column did in the previous time through the loop. And the first time, there was no previous time. Hence the need for additional microcode before getting into the loop.

Exactly what needs to go before the loop? In order for the second column of the loop to be doing what it's supposed to when the

program gets to it, the first column must precede it. Essentially, one rewrites the first column as an "intro" to the loop. Thus:

```
PROGRAM:                          MOV APTR,APTR; SETMA  "get first element
                                                        "of A
                                  SUB XINC,CPTR         "to offset ADD in loop

                                  MOV BPTR,BPTR; SETMA  "get first element
                                                        "of B
                                  DPX<MD                "store A(1) in DPX

                                  FMUL DPX,MD           "do A(1)^2

LOOP:                             FMUL

    ADD XINC,APTR; SETMA;         FMUL                  "get  A(m+1)

                                  FADD FM,MD            "do A(m)^2+B(m)

    ADD XINC,BPTR; SETMA;         FADD                  "get B(m+1)

    DPX<MD;                       DEC N                 "store A(m+1)

    FMUL DPX,MD;                  ADD XINC,CPTR; SETMA; MI<FA; BGT LOOP
                                                        "do A(m+1)^2, store
                                                        "C(m), test if done

DONE:  RETURN
```

To clear up a loose end regarding the structure of memory-limited loops, one might notice that since the branch must be in the last cycle, the DEC N instruction must be in the second-to-last cycle. DEC is an S-Pad operation and cannot be in the same cycle as another S-Pad operation, such as ADD XINC,XPTR. A memory-limited loop has SETMA's (requiring S-Pad operations) on every other line. Since the DEC N operation will go on an odd-numbered line of the loop, the SETMA's must go on even-numbered lines. This is why the first thing to do, ADD XINC,APTR; SETMA (see section 2.3), was put on line 2.

2.5 A DOT PRODUCT PROGRAM

It is now possible to write the 4-cycle dot product. Using the technique outlined above, the loop should be constructed as follows:

```
1)  ---
2)  ADD XINC,APTR; SETMA   "get A
3)  ---
4)  ADD XINC,BPTR; SETMA   "get B
```

then

```
1)  ---                          | DPX<MD          "store A
2)  ADD XINC,APTR; SETMA         | ---
3)  ---                          | FMUL DPX,MD     "do A·B
4)  ADD XINC,BPTR; SETMA         | FMUL
```

then

```
1)  ---                          | DPX<MD          | FMUL
2)  ADD XINC,APTR; SETMA         | ---             | FADD FM,FA    "add A·B to sum of
                                                                   "products
3)  ---                          | FMUL DPX,MD     | FADD; DEC N   "decrement counter
4)  ADD XINC,BPTR; SETMA         | FMUL            | BGT LOOP      "test if done
```

The intro to this 3-column loop will consist of the first column alone, then the first and second column together. Other overhead, such as initializing FA to 0, can be mixed in with the intro.

To generalize, an N-column loop would require an intro consisting of column 1 followed by columns 1 and 2 together, followed by columns 1, 2, and 3 together.... followed by columns 1, 2,..,N-1 together.

```
$TITLE DOTPROD
$ENTRY DOTPROD

APTR $EQU 0
BPTR $EQU 1
CPTR $EQU 2
XINC $EQU 3
N    $EQU 4

DOTPROD:  MOV APTR,APTR; SETMA; FADD ZERO,ZERO    "get A(1) and
                                                  "initialize FA=0
          MOV BPTR,BPTR; SETMA; FADD              "get B(1)

          DPX<MD                                  "store A(1)

          ADD XINC,APTR; SETMA                    "get A(2)

          FMUL DPX,MD                             "do A(1)*B(1)

          ADD XINC,BPTR; SETMA; FMUL              "get B(2)

LOOP:     DPX<MD; FMUL                            "store A(m+1)

          ADD XINC,APTR; SETMA; FADD FM,FA        "get A(m+2), add
                                                  "A(m)B(m) to sum
          FMUL DPX,MD; FADD; DEC N                "do A(m+1)B(m+1)
                                                  "decrement counter
          ADD XINC,BPTR; SETMA; FMUL; BGT LOOP    "get B(m+2), test if
                                                  "done
DONE:     MOV CPTR,CPTR; SETMA; MI<FA; RETURN     "if so, store answer

.END
```

Now each new pair of elements will only cost 4 more cycles, because every 4 cycles a new pair are being fetched; every 4 cycles another product is added to the sum. The longer overhead is no disadvantage as it is only done once, and even if the program was called with N containing 1, making the streamlined loop unnecessary, it takes no longer than the unstreamlined program.

Note that there are 2 SETMA's in a row at the beginning and again at the end of the program. This will not cause any problems except to make memory spin, which is the memory's way of putting in the NOP's the programmer leaves out. The timing is still the same, and this way there are 2 less locations of Program Source used up.

It might be mentioned that if one were getting Vectors A and B out of Data Pad instead of memory, the dot product could be written with a 1-cycle loop! This will be demonstrated later.

2.6 NOTATION

A few words about notation are in order. The "---" used when writing loops in column form simply denotes a blank spot, indicating a cycle goes by while awaiting the results of a memory fetch or while looking for a more propitious spot to use the results of the Adder or Multiplier, etc. Normally, something else will eventually go on the same line, in a different column.

Example: This takes vector A, multiplies it by a constant in DPX, and stores it in vector B.

```
1)  ---                              FMUL DPX,MD
2)  ADD XINC,APTR; SETMA             FMUL
3)  ---                              FMUL; DEC N
4)  ---                              ADD XINC,BPTR; SETMA; MI<FM; BGT LOOP
```

Since the length of the loop was already decided by the number of SETMA's, these blank spots cause no harm to the speed. It is the number of cycles in the loop, not the number of columns, which determines speed. Extra columns simply mean longer intros, which the program only goes through once anyway unless it's part of a nested loop.

In loops with several Adder or Multiplier operations, it often happens that one such instruction will be a "pusher" for another in another column.

```
1)  (code)                           FMUL DPY,MD (code)
2)     " FMUL DPX,DPY                FMUL          "
3)     " FMUL                        FMUL          "
3)     " FMUL                        DPY<FM        "
5)     " DPX(1)<FM                                 "
6)     "                                           "
```

In column 2, lines 2 and 3 are illegal, as those lines already contain FMUL's (which will do the pushing for column 2 as well as column 1). However it may be advantageous to the programmer to note to himself somehow that FMUL's do belong there, in case things in the first column get moved around for some reason. This is the purpose of such notation as (fmul) or (fadd).

Thus:

```
1)  (code)                          FMUL DPY,MD (code)
2)    " FMUL DPX,DPY                (fmul)        "
3)    " FMUL                        (fmul)        "
4)    " FMUL                        DPY<FM        "
5)    " DPX(1)<FM                                 "
6)    "                                           "
```

Now, if pieces of the first column were moved down a couple of lines for some reason,

```
1)  (code)                          FMUL DPY,MD (code) DPX(1)<FM
2)    "                             (fmul)        "
3)    "                             (fmul)        "
4)    " FMUL DPX,DPY                DPY<FM        "
5)    " FMUL                                      "
6)    " FMUL                                      "
```

the programmer would be reminded to put real FMUL's back on those lines.

When writing loops with a small number of cycles, these reminders can also help one keep track of the columns, as in:

```
---                        | ---  DPY<MD          FMUL      FADD FM,FA; DEC N
ADD XINC,APTR; SETMA       | ---  FMUL DPY,MD     (fmul)    FADD; BGT LOOP
```

This gets a vector from memory, squares each element and adds the squares together (sort of a dot product between vector A and itself). The seemingly empty columns, which disappear when the loop is written in proper form (see below), are necessary in order to write the intro properly. If one left out the second column, for example, his intro would start with:

```
MOV APTR,APTR; SETMA
DPY<MD
ADD XINC,APTR; SETMA; FMUL  DPY,MD
        .
        .
        .
```

Clearly, the first MD will not be the first element fetched. By the time it gets down to FADD FM,FA in the loop, something which doesn't belong will be added in.

This is what the intro and loop should look like:

```
      MOV APTR,APTR; SETMA
      FADD ZERO,ZERO                          "initialize FA=0
      ADD XINC,APTR; SETMA; FADD
      DPY<MD
      ADD XINC,APTR; SETMA; FMUL DPY,MD
      DPY<MD; FMUL
      ADD XINC,APTR; SETMA; FMUL DPY,MD

LOOP: DPY<MD; FMUL; FADD FM,FA; DEC N
      ADD XINC,APTR; SETMA; FMUL DPY,MD; FADD; BGT LOOP

      (answer)<FA
```

2.7 DROPPING OUT ONE EARLY

```
1)  ---                          (code)                    (code)
2)  ADD XINC,APTR; SETMA           "                         "
3)  (code)                         "                         "
4)  ADD XINC,BPTR; SETMA           "                         "
5)  (code)                         "                         "
6)     "                       ADD XINC,CPTR; SETMA          "
7)     "                       (code)                      "  DEC N
8)     "                         "                       ADD XINC,DPTR; SETMA;
                                                             MI<DPX; BGT LOOP
```

Here, there are 2 memory reads in the first column, one read in the second column, and a store in the last column. When writing the intro, the pointers should be taken care of as follows:

```
MOV APTR,APTR; SETMA
SUB XINC,DPTR; (code)
MOV BPTR,BPTR; SETMA
(code)
  .
  .
  .
ADD XINC,APTR; SETMA; (code)
(code)                   "
ADD XINC,BPTR; SETMA;    "
(code)                   "
  "                MOV CPTR,CPTR; SETMA
  "                  (code)
  "                    "
```

If the memory reference in the second column of the loop was a store instead of a read, the problem would become more complicated. By the time the counter went down to zero and the last result was stored at DPTR, an extra C would have been stored, possibly over a valuable piece of data, such as the beginning of vector D. Or if instead of ADD XINC,CPTR; SETMA; MI<DPY in the second column, we had DPY<FA (where FA is cumulative, as in the dot product) and later stored DPY into CPTR after getting out of the loop, an extra FA would have been computed and DPY would contain an incorrect answer. In this case, it would be wise to drop out of the loop one time early. One would put an extra DEC N somewhere in the intro, so that the loop would be done N-1 times. Then, after the loop, write just the last column (not including DEC and the branch, of course), which is all that remains to be done from the loop anyway.

Example: This does a dot product of vectors A and B, and also outputs the square of each updated sum into vector D.

```
---                          FMUL DPX,MD      ---
ADD XINC,APTR; SETMA         (fmul)           FMUL DPY,DPY
---                          FMUL             (fmul)
ADD XINC,BPTR; SETMA         FADD FM,FA       FMUL
DPX<MD                       FADD             DEC N
---                          DPY<FA           ADD XINC,DPTR; SETMA; MI<FM;
                                                BGT LOOP
```

When it is going through the loop for the last time and storing the very last thing in D (column 3), it is also simultaneously doing extra executions of columns 1 and 2. Normally, that doesn't matter, but in this case, something extra is being added to the cumulative sum of the dot product (column 2), which was completed the previous time through the loop. By dropping out of the loop before its last time around, this error is avoided:

```
MOV APTR,APTR; SETMA
DEC N                        "to cause dropping out early
MOV BPTR,BPTR; SETMA
DPX<MD
SUB XINC,DPTR                "to nullify the first ADD XINC,DPTR
                             FMUL DPX,MD
ADD XINC,APTR; SETMA;        FMUL
                             FMUL
ADD XINC,BPTR; SETMA;        FADD FM,FA
DPX<MD;                      FADD
                             DPY<FA
LOOP:                        FMUL DPX,MD
  ADD XINC,APTR; SETMA                        FMUL DPY,DPY
                             FMUL
  ADD XINC,BPTR; SETMA;      FADD FM,FA;       FMUL
  DPX<MD;                    FADD;             DEC N
                             DPY<FA;           ADD XINC,DPTR; SETMA; MI<FM;
                                                 BGT LOOP
OUT:                                           FMUL DPY,DPY
                             MOV CPTR,CPTR;    SETMA; MI<DPY; FMUL
                                               FMUL
                                               ADD XINC,DPTR; SETMA; MI<FM;
                                                 RETURN
```

Notice that the (fmul) in column 2 became a real FMUL in the intro.

OUT starts just the last column. The next line stores the completed dot product.

One might wish to come out one early even if one doesn't strictly need to, if the loop is long and there are only a couple of lines in the last column:

```
1)  (code)        (code)
2)     "   SETMA      "
3)     "              "
4)     "              "   SETMA; MI<DPX
5)     "
6)     "   SETMA
7)     "
8)     "   SETMA
9)     "
10)    "   SETMA
11)    "              DEC N
12)    "   SETMA      BGT LOOP
```

In this case, coming out of the loop one time early and adding on the last 4 lines afterward would save going through 8 cycles for nothing.

2.8 INTERACTION BETWEEN COLUMNS

In order to fit things into complicated loops without creating op-code conflicts, the AP programmer takes advantage of results (e.g. MD, FA) which are the same for one or more cycles after it is first available. Sometimes he will purposely delay the pushing of an answer through a pipeline by leaving out "pushers". But he must be careful of the way the columns interact with each other within the loop.

```
1)  FMUL DPX,DPY        .
2)       .              FMUL DPY(3),DPX(2)
3)       .              FMUL
4)  FMUL                  .
5)  FMUL                DPY(1)<FM
6)  DPX(1)<FM             .
```

The FMUL's in column 2 will act as "pushers" for the FMUL DPX,DPY in column 1, whose answer will come out on line 4 instead of line 6 as desired and will disappear forever when replaced by a new FM on line 5. Notice the FMUL on line 4 in column 1 acts as a pusher for column 2, which was planned for.

Another example:

1)	(code)	DPX<MD	(code)
2)	ADD XINC,APTR; SETMA	(code)	DPX<FA
3)	(code)	FADD FM,DPX	(code)
4)	"	(code)	"

The DPX of column 2 line 3 will not be the same as what was stored into it in column 2, line 1. It will be FA from column 3, line 2.

2.9 CHANGING DPA

Because one can access things in Data Pad much faster than things in memory, it makes sense to store things from memory into Data Pad if they will be used again. For example, if one is going to use an N-element vector for several different computations, one could store it in DPX(0), DPX(1),....,DPX(N-1). Because the Data Pad indices can only be accessed from -4 to +3 with a static DPA, it becomes useful to leave the index alone and change DPA.

Storing vector A in DPX is basically the repeated operation of DPX<MD; INCDPA. If DPA is initially set to zero, then the first element will be stored into DPX(0). INCDPA will increase DPA for the next instruction. Thus:

```
DPX<MD; INCDPA  "refers to DPX(0)
DPX<MD          "refers to DPX(1)
```

The ways to set DPA to zero:

```
CLR# (S-Pad name); SETDPA   "uses up S-Pad field
                or
DB=ZERO; LDDPA              "uses up Adder field
```

This loop will read a vector from memory into Data Pad X:

---	---	DPX<MD; INCDPA; DEC N
ADD XINC,APTR; SETMA	---	BGT LOOP

With intro:

```
      MOV APTR,APTR; SETMA
      CLR# APTR; SETDPA
      ADD XINC,APTR; SETMA
LOOP: DPX<MD; INCDPA; DEC N
      ADD XINC,APTR; SETMA; BGT LOOP
```

2.10 NON-MEMORY-LIMITED LOOPS

A non-memory-limited loop is a loop in which 2 times the number of memory references is less than the number of same-op-code-field operations required. For example, if there are 5 Floating Adder operations to be done (FADD, FSUB, FSUBR, etc.) but only 2 memory references (a fetch and a store), the 5 Adder operations cannot fit into 4 cycles.

Incidentally, "pushers" don't count in figuring out how many cycles are needed. In a 5-cycle loop with 5 different Adder operations, the Adder instructions become each other's pushers.)

Recall that in memory-limited loops, the first instruction in column 1 usually starts on line 2, to avoid S-Pad conflicts on the next-to-last line. (See last paragraph of Section 2.4). This is not necessary in non-memory-limited loops.

The following loop will test whether each element of a vector in DPY is within the range between a maximum limit and minimum limit. If so, the element is added to a cumulative sum. The maximum limit is conveniently located in MD, and the minimum limit in FM, by the grace of whatever program uses this loop. Neither FM nor MD change during this loop's execution.

```
FSUB DPY,MD       BFGT BIGGER       | (fadd)
FSUB FM,DPY       BFGT SMALL        | DPX<FA; DEC N
(fadd) INCDPA     FADD DPY(-1),DPX  | BGT LOOP
```

Note that the BFGT instruction tests FA of the previous cycle.

2.11 A 1-CYCLE LOOP

For the 1-cycle dot product, it is assumed that the vectors are already in Data Pad, starting at DPX(0) and DPY(0) (where DPA=0). Obviously, vectors longer than 32 elements cannot be handled this way (or can only be handled in segments of 32 or less).

This is what the loop really looks like:

```
FMUL DPX,DPY; INCDPA | (fmul)  (fmul)  FADD FM,FA; DEC N | (fadd) BGT LOOP
```

The FMUL and FADD instructions become their own "pushers".

```
$TITLE DOTPROD
$ENTRY DOTPROD

N $EQU 0          "number of elements in each vector
CPTR $EQU 1       "where to store answer

DOTPROD:  CLR# N; SETDPA              "DPA=0
          FMUL DPX,DPY;               "do A(1)*B(1)
            INCDPA;                   "DPA to 1
            DEC N                     "set drop out early
          FMUL DPX,DPY;               "do A(2)*B(2)
            INCDPA                    "DPA to 2
          FMUL DPX,DPY;               "do A(3)*B(3)
            INCDPA;                   "DPA to 3
            FADD ZERO,ZERO            "init. FA=0
          FMUL DPX,DPY;               "do A(4)*B(4)
            INCDPA;                   "DPA to 4
            FADD FM,ZERO;             "A(1)B(1) in Adder
            DEC N                     "decrement counter
LOOP:     FMUL DPX,DPY;               "do A(m)*B(m)
            INCDPA;                   "DPA to DPA+1
            FADD FM,FA;               "add A(m-3)B(m-3) to sum
            DEC N;                    "decrement counter
            BGT LOOP                  "test if done

OUT:      DPX<FA; FADD                "store cumulative FA
          FADD DPX,FA                 "add it to other cumulative FA
          FADD
          MOV CPTR,CPTR; SETMA; MI<FA;  "store answer
            RETURN
$END
```

This particular sort of loop has a problem with the Floating Adder, in that a cumulative FA needs at least 2 cycles to accumulate each new addition. Hence, the 1-cycle loop is actually operating with 2 mutually exclusive cumulative FA's, interwoven with each other:

```
FADD FM,FA
FADD FM,FA
FADD FM,FA
FADD FM,FA
FADD FM,FA
```

At the end of all this, they (the two strings of sums) need to be added to each other. (See OUT, the label after LOOP).

This also illustrates the practice of dropping out of the loop one time early. If it didn't drop out early, the last (unneeded) FADD FM,FA of the loop would push out one of the 2 cumulative FA's. By the next cycle it would be gone forever. By dropping out early, DPX<FA can be done before it's too late.

This line of reasoning can eventually lead one to the idea that the last column of the loop (see beginning of Section 2.11) is unnecessary, since there is no way for the Adder result to come out in time for the next FADD FM,FA. The FADD FM,FA of each of the two strings of cumulative FA's will push out the other string. So the loop need only be of the form:

FMUL DPX,DPY; INCDPA	(fmul)	(fmul) DEC N	FADD FM,FA; BGT LOOP

This is one column less than before, which means that there will be one column's worth (in this case, one line) less to put in the intro. It will also not be necessary to come out of the loop one time early, as there is no extra FADD FM,FA to push away something needed. It is still necessary to add the 2 cumulative FA's together at the end.

```
$TITLE DOTPROD
$ENTRY DOTPROD

N    $EQU 0
CPTR $EQU 1

DOTPROD:  CLR# N; SETDPA                  "DPA=0
          FMUL DPX,DPY;                   "A(1)*B(1)
            INCDPA;                       "  DPA to 1
            FADD ZERO,ZERO                "  initialize cum. FA=0
          FMUL DPX,DPY;                   "A(2)*B(2)
            INCDPA;                       "  DPA to 2
            FADD ZERO,ZERO                "  initialize other cum. FA=0
          FMUL DPX,DPY;                   "A(3)*B(3)
            INCDPA;                       "  DPA to 3
            DEC N
LOOP:     FMUL DPX,DPY;                   "A(m)*B(m)
            INCDPA;                       "  DPA to DPA+1
            DEC N;                        "  decrement counter
            FADD FM,FA;                   "  add A(m-3)B(m-3) to cum. FA
            BGT LOOP                      "  test if done
OUT:      DPX<FA; FADD                    "store first cumulative FA
          FADD DPX,FA                     "add it to other cumulative FA
          FADD
          MOV CPTR,CPTR; SETMA; MI<FA;    "store answer
            RETURN
$END
```

SECTION 3
CAVEAT PROGRAMMER (Let the Programmer Beware)

3.1 CALLING ANOTHER SUB-ROUTINE

The JSR instruction allows one program to utilize another program, for example the divide sub-routine (DIV). In order to do this, one must declare DIV external ($EXT DIV) so that the assembler and linker will know what to do with the otherwise undefined symbol. One must also save everything he will need when program execution gets back to his main program. Depending upon what was used in the called sub-routine, some things may remain untouched. Commonly one should not count on being able to leave things in the Adder or Multiplier. Parts of Data Pad may also be changed, or DPA may change. S-Pad will probably not remain inviolate. (Remember, it's the S-Pad register number, not name, which is important.) These things need to be checked before doing a JSR.

3.2 OTHER THINGS TO WATCH OUT FOR

The rest of this section consists of various short examples, cautions, and reminders.

DPX<MD; DPY<DPX(1)
Illegal. Data Bus is assigned twice. (The above is really DPX<DB; DB=MD; DPY<DB; DB=DPX(1).)

DPX<MD; DPY<MD is legal. (DB=MD; DPX<DB; DPY<DB)

DPX<FA; DPY<FM
Legal. FA and FM don't use the Data Bus.

DPX<FA; DPY<FM; DPX(1)<MD
Illegal. Data Pad X is being written into twice (different indices). Within each cycle, there should be no more than one of each of the following:

- write into DPX
- write into DPY
- read from DPX
- read from DPY

The exception is when reading out of a Data Pad more than once but using the same index:

- FADD DPX,FA; FMUL DPX,FA; DPY(1)<DPX is legal.
- FADD DPY,DPY; FMUL DPY,DPY is legal.
- FADD DPX,FA; FMUL DPX(1),FA is not legal.

FADD DPX,DPY; DPY<MD
The old value of DPY, before MD replaces it, is used in the sum.

```
DB=4; LDSPI XINC; LDDPA; DPX(2)<FM
```
Both DPA and the contents of XINC will become 4, but the previous DPA is used in referencing DPX(2).

```
SUB# XINC,APTR; BGT OUT
```
Illegal. The # uses the condition field (branch).

THE $END

AP-120B APAL

Array Processor Assembly Language Manual

7275-01

FPS-7275-01

Revision 01, February 26, 1976

TABLE OF CONTENTS

SECTION 1
INTRODUCTION

APAL (Array Processor Assembly Language) is a cross-assembler written in Fortran IV which provides a two-pass assembly of symbolic coding for the AP-120B.

APAL is a conventional assembly language, and as such, should pose no difficulties to programmers familiar with using assembly language on other computers.

On typical 16-bit mini-computers, APAL requires approximately 24K of available memory to operate as supplied.

INTENTIONALLY BLANK

SECTION 2
BASIC SYNTAX

2.1 CHARACTER SET

APAL recognizes the following characters:

Alphabetic	A thru Z
Numeric	Ø thru 9
Special	+ - * / . $ space tab = <
	( ) ; , : " # & !

2.2 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 an alphabetic character, the subsequent characters may be either letters of numbers.

Examples: LOOP
A6
STARTHERE

Symbols are given a value in any of three ways:

1. Being defined by a $EQU pseudo-op.
2. Being used as a label.
3. Being declared $EXTernal.

2.3 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 a "!" 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 is in Appendix C. For example, to fetch PI from Table Memory, and add it to a number in DPX(2)

```
LDTMA; DB=!PI     "fetch PI from TM
NOP               "Wait (or do something else)
FADD TM, DPX(2)   "Add PI to DPX(2)
```

2.4 NUMBERIC CONSTANTS

Numbers may be written in four radices: octal, decimal, binary, or hex. In each radix, a number may be either signed or unsigned. Unsigned numbers may range from Ø to 65535. Signed numbers may range from -32768 to +32767. The radix of a number is established by a radix identifying character which is written immediately after the number. Octal numbers are denoted by a "K" immediately following the number. Decimal is denoted by a ".", binary by a "B" and Hex by an "X". The first digit of a hex number must be a numeric character. The default radix, if a radix identifier is not used, is octal.

```
Examples:  Octal integers:      177777
                               -4Ø727K
                               -10
           Decimal integers:    32767.
                               -1ØØØ.
                               +1Ø.
           Binary integers:     1Ø1Ø11Ø1ØB
                               -1Ø1B
           Hex integers:        ØABCDX
                                123FX
                                ØCX
```

2.5 EXPRESSIONS

Expressions may be used whenever a numeric value is required. Expressions are made up of operands and operators.

2.5.1 Operands. Operands may be symbol names, numeric constants, or the location counter, denoted by ".".

```
Examples:  TBLADR
           598X
           .
```

2.5.2 Operators. Operators denote operations of addition ("+"), subtraction ("-"), multiplication ("*"), or division ("/") upon a pair of operands.

Some sample expressions:

```
TBLADR + 37
. + 9.
LOOP + 6 * A
```

Expressions are evaluated from left to right, modulo 2^{16}.

SECTION 3
SOURCE PROGRAM STATEMENTS

APAL source statements may be divided into three categories:

1. Comment statements
2. Instruction statements
3. Pseudo-op statements

Comment statements allow program documentation, instruction statements contain the actual symbolic machine code, and pseudo-op's provide directives to APAL during the assembly process.

APAL statements have a basically free format: spaces and tabs may be used as desired to improve legibility.

3.1 COMMENT STATEMENTS

Everything on a line after 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. Carriage return terminates a comment.

3.2 INSTRUCTION STATEMENTS

An APAL assembly language instruction statement has the general format of:

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

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

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 refering 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, if the current location is 76 the instruction statement:

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

Assigns the value of 76 to the symbol "LOOP".

3.2.2 Op-code field(s). The Op-code field follows the label field in an instruction statement and contains one or more AP-120B op-code mnemonic. Individual op-codes in an instruction are separated from each other by a semicolon ";". The last op-code in an instruction is not terminated by a semicolon. This tells the assembler when it has reached the end of a complete AP-120B instruction statement. For example, both of the following instruction statements are equivalent:

```
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. Instruction statements are terminated by an op-code field which is not followed by a semicolon: not by the end of a line.

Op-codes may be written in any order preferred within any given instruction statement. The assembler will flag with an error message any conflicts between op-codes.

Some op-codes require operands as arguments. The operand(s) are separated from the op-code by a space or tab, and from each other by a comma. Some example op-codes:

```
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 will give an appropriate error message.

A listing of the various AP-120B op-codes is contained in Appendix A.

3.2.3 Comment Field. The remainder of any line following a quote mark (") is treated as a comment by the assembler and ignored. The comment field is terminated by a carriage return. Thus, in the previous example, we could write:

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

As before, an instruction statement is ended by the absence of a semicolon following the last op-code in that instruction.

3.3 PSEUDO OPERATIONS STATEMENTS

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

3.3.1 $EQU. This operator equates a symbol to 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 characters " = " may be used in place of "$EQU":

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

When so used, the "=" must be both preceeded and followed by a least one space or tab character.

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 earlier in the program.

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

Caution: $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.

3.3.3 $END. $END causes APAL to terminate the assembly.

3.3.4 $VAL. This operator defines 64 bits worth of data to fill one program word. The data is specified as four 16-bit integers, 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
```

3.3.5 $FP. This operator fills the right-most 38-bits of a program word with a specified floating-point number. The left-hand 26-bits of the word are cleared.

```
Examples:  $FP  6.0023E23
           $FP  2
           $FP  E-17
PI:        $FP  3.141592654  "PI IS HERE
```

Note: a floating-point number (say 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
```

3.3.6 $TITLE. This pseudo-op names a program. The name need not be unique from other symbols in the program. The $TITLE pseudo-op must occur first in the program, before any other pseudo-ops or instruction statements.

```
Examples:  $TITLE FFT
           $TITLE DIVIDE
```

3.3.7 $ENTRY. This pseudo-op declares a symbol to be global; i.e., a symbol which is defined in this program and may be referenced by other separately assembled programs. The identified symbols must be defined in this program either by an "$EQU" pseudo-op or by being used as a label. $ENTRY pseudo-ops must occur before any instruction statements in the program.

If the symbol is to be an entry point for Host Computer Fortran Calls, then following the symbol name must be the number of S-Pad parameters expected in the CALL. This may be a number from $0\text{-}17_8$, 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
```

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

3.4 ORDER OF PROGRAM STATEMENTS

There is definite ordering of statement types with a program. The $TITLE Pseudo-op comes first. Next, if present, any $ENTRY and $EXT pseudo-ops. The the program body, i.e. the code, then occurs. Finally comes the $END pseudo-op. Statement order:

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

*Need not be present.

```
        $TITLE DØTPR
        $ENTRY DØTPR, 6
"VECTØR DØT PRØDUCT
"DØES C(0) = SUM ( A(MI) * B(MJ) )    FØR M = 0 TØ N-1

"                    --- STATISTICS ---
"AUTHØR: A.E. CHARLESWØRTH, JULY 75
"REVISIØN 1.3  FEB 76  CHANGED $ENTRY
"SIZE: 9. LØCATIØNS
"SPEED: 2 MEMØRY REFERENCES PER PØINT
"SCRATCH:  SP: 0,4,5;  DPX: 0 (RELATIVE TØ DPA)

"S - PAD PARAMETERS:
        A $EQU 0          "BASE ADDRESS ØF A
        I $EQU 1          "INCREMENT FØR A
        B $EQU 2          "BASE ADDRESS ØF B
        J $EQU 3          "INCREMENT FØR B
        C $EQU 4          "ADDRESS ØF C
        N $EQU 5                          "VECTØR LENGTH
DØTPR:  MØV A,A; SETMA          "FETCH A(0)
        MØV B,B; SETMA          "FETCH B(0)
        DPX<MD;                 "SAVE A(0)
          INC N                 "  KEEP CØUNT RIGHT
        ADD I,A; SETMA;         "FETCH A(1)
          FADD ZERØ,ZERØ        "  CLEAR SUM
LØØP:   FMUL DPX,MD;            "DØ A(M)*B(M)
          FADD;                 "  PUSH ADDER
          DEC N                 "  SEE IF DØNE
        BEQ DØNE;               "BRANCH IF DØNE
          FMUL;                 "  PUSH MULTIPLIER
          ADD J,B; SETMA        "  FETCH B(M+1)
        DPX<MD;                 "SAVE A(M+1)
          FMUL                  "  PUSH MULTIPLIER
        FADD FM,FA;             "ADD (A(M)*B(M)) TØ SUM
        ADD I,A; SETMA;         "  FETCH A(M+2)
          BR LØØP               "  BRANCH BACK
DØNE:   MI<FA; MØV C,C; SETMA;  "STØRE ANSWER IN C(0)
          RETURN                "  RETURN
       $END
```

SECTION 4
OPERATING PROCEDURES

4.1 USING APAL

APAL assembles a file of source text containing an AP-120B program into a relocatable object file. Optionally an assembly listing is produced.

APAL first requests whether another assembly is to be done:

```
DONE? 1=YES, Ø=NO:
```

A response of "1" will cause APAL to exit to the system monitor. A "Ø" will signal APAL that another assembly is to be done.

APAL then requests the names of the three files to be used for source, object, 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 output 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.

Finally, APAL outputs:

```
LISTING? 1=YES, Ø=NO:
```

A response of "1" will yield a full assembly listing, symbol table, and any error messages. A "Ø" will suppress the assembly and symbol table listings and put out only any error messages into the listing file.

Finally, if a listing was requested, APAL outputs:

```
LISTING RADIX? 1=HEX, Ø=OCTAL:
```

A response of "1" will cause the assembly listing to be done in Hexadecimal (base 16). A"Ø" will make the assembly listing in Octal (base 8).

In each of the above cases, if the sought after file cannot be found or is otherwise unavailable, APAL types "??", and waits

for another user response.

An example dialogue is given below. The user desires to assemble an AP-120B program in file "FFT.AP" and put the object output into file "FFT.RB". The listing will be put out on the line printer. Of course, the precise details of how files and devices are named will depend on the particular operating system being used. The messages printed by the computer are underlined for clarity; a "↓" means carriage return:

```
RUN APAL↓
DONE? 1=YES, Ø=NO: 1↓
SOURCE FILE = FFT.AP↓
OBJECT FILE = FFT.RB↓
LISTING FILE = LP:↓
LISTING? 1=YES, Ø=NO: 1↓
LISTING RADIX? 1=HEX, Ø=OCTAL: Ø↓
DONE? 1=YES, Ø=NO: Ø↓
```

4.2 LISTING FORMAT

Upon commencement of the assembly, APAL outputs:

```
APAL
###
PASS 1
```

"###" is the version number of the assembler being used. Any errors detected during pass one are output next. At the start of pass two APAL outputs:

```
PASS 2
```

The assembly listing follows. The listing contains the following information for each program statement:

Columns	Contents
1-6	The location counter, if relevant.
7-8	Blank
9-14	The assembled data, if relevant.
15-16	Blank
17-132	The 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

```
****   ###   ERRORS   ****
```

Where "###" is the number of errors detected. Finally, APAL outputs:

```
SYMBOL     NAME
```

Followed by the symbol table:

Columns	Contents
1-6	Symbol Name
7-8	Blank
9-14	Symbol value
15	Blank
16-18	Symbol type:

Blank - local symbol
EXT - external symbol
ENT - entry symbol

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

(INTENTIONALLY BLANK)

```
>>R APAL
 SØURCE FILE=/CDR
 ØBJECT FILE=TEMP
 LISTING AND ERRØR FILE=/TTØ
 LISTING? 1=YES, 0=NØ: 1
 LISTING RADIX: 1=HEX, 0=ØCTAL: 0


APAL
V2.1
PASS 1

PASS 2
                        $TITLE DØTPR
                        $ENTRY DØTPR, 6
                    "VECTØR DØT PRØDUCT
                    "DØES C(0) = SUM ( A(MI) * B(MJ) )   FØR M = 0 TØ N-1

                    "                    --- STATISTICS ---
                    "AUTHØR: A.E. CHARLESWØRTH, JULY 75
                    "REVISIØN 1.3  FEB 76  CHANGED $ENTRY
                    "SIZE: 9. LØCATIØNS
                    "SPEED: 2 MEMØRY REFERENCES PER PØINT
                    "SCRATCH:  SP: 0,4,5;  DPX: 0 (RELATIVE TØ DPA)

                    "S - PAD PARAMETERS:
          000000            A $EQU 0          "BASE ADDRESS ØF A

          000001            I $EQU 1          "INCREMENT FØR A

          000002            B $EQU 2          "BASE ADDRESS ØF B

          000003            J $EQU 3          "INCREMENT FØR B

          000004            C $EQU 4          "ADDRESS ØF C

          000005            N $EQU 5                          "VECTØR LENGTH

000000    040000  DØTPR:    MØV A,A; SETMA          "FETCH A(0)
          000000
          000000
          000060

000001    040210            MØV B,B; SETMA          "FETCH B(0)
          000000
          000000
          000060

000002    001124            DPX<MD;                 "SAVE A(0)
          000000              INC N                 "  KEEP CØUNT RIGHT
          045004
          000000

000003    020101            ADD I,A; SETMA;         "FETCH A(1)
          155000              FADD ZERØ,ZERØ        "  CLEAR SUM
          000000
          000060
```

```
000004  001225  LØØP:   FMUL DPX.                "DØ A(M)*B(M)
        100000            FADD;                 "  PUSH ADDER
        000400            DEC N                 "  SEE IF DØNE
        013400

000005  020310            BEQ DØNE;             "BRANCH IF DØNE
        000623            FMUL;                 "  PUSH MULTIPLIER
        000000            ADD J,B; SETMA        "  FETCH B(M+1)
        010060

000006  000000            DPX<MD;               "SAVE A(M+1)
        000000            FMUL                  "  PUSH MULTIPLIER
        045004
        010000

000007  020101            FADD FM,FA;           "ADD (A(M)*B(M)) TØ SU
        111115            ADD I,A; SETMA;       "  FETCH A(M+2)
        000000            BR LØØP               "  BRANCH BACK
        000060

000010  040420  DØNE:     MI<FA; MØV C,C; SETMA; "STØRE ANSWER IN C(0)
        000340            RETURN                "  RETURN
        000000
        000160

                        $END


****      0 ERRØRS    ****


SYMBØL  VALUE

A       000000
I       000001
B       000002
J       000003
C       000004
N       000005
DØTPR   000000 ENT
LØØP    000004
DØNE    000010
```

INTENTIONALLY BLANK

SECTION 5
ERROR MESSAGES

APAL error messages are printed in the listing following the offending statement. There are five basic error classes, which are listed below along with the action taken by the assembler:

O - Out of range: the offending 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 useage.

Error diagnostics issued by APAL consist of two lines. The first line consists of the error number. The second line contains the error class and error message. Following are the assembler error messages, along with an explanation as to the possible causes and/or cures.

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. O S-PAD ADDRESS OUT OF RANGE
   An S-Pad address was outside the legal range of 0-15.

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.

7. M MISSING BRANCH ADDRESS
No target address was given for a branch op-code.

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 WRONG FADD ARGUMENT
A floating adder op-code had an invalid A1 or A2 operand.

11. M WRONG FMUL AGREEMENT
A FMUL op-code had an invalid M1 or M2 operand.

12. M MISSING FADD OR FMUL ARGUMENT
An operand was missing following a FADD or FMUL op-code.

13. C VALUE FIELD CONFLICT
Only one op-code which uses a 16-bit VALUE field operand may be used per instruction statement.

14. M MISSING DATA PAD INDEX
A Data Pad Index was missing from an op-code where it was needed.

15. M UNDEFINED OP-CODE
An op-code name was not a legal AP-120B instruction.

16. M $EXT 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.

18. M MISSING ARITHMETIC OPERATOR
An arithmetic operator (+ - * /) was missing from an expression.

19. O INTEGER OVERFLOW
An integer constant was too large to fit in 16 bits.

20. B UNRECOGNIZED STATEMENT
A statement line was neither a comment, instruction, or pseudo-op statement.

21. M IMPROPER $LOC OR $EQU VALUE
The value of a $LOC or $EQU pseudo-op was either an undefined symbol or an improper expression.

22. M $EXT 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

25. B MISSING PARENTHESES
The right parenthesis following a Data Pad Index was missing.

26. M BAD DATA PAD INDEX EXPR
A Data Pad Index expression could not be resolved into a numeric value.

27. B COMMA MISSING
Only a comma may be used to separate pseudo-op arguments.

28. B SYMBOL MISSING IN $EXT PSEUDO-OP
No symbol names were found as arguments for an $EXT pseudo-op.

29. B MISSING SEP AFTER D.P. INDEX
An illegal character was found following a Data Pad Index.

30. B MULTIPLE PSEUDO-OPS
Only one pseudo-op statement may appear on a line.

31. M BAD FLOATING POINT NUMBER
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.

33. W $ENTRY SYMBOL NOT LOCAL
An $ENTRY Symbol must not be $EXTernal also.

34. W UNREFERENCED $EXT SYMBOL
A declared external symbol was never used in the program.

35. W UNDEFINED $ENTRY SYMBOL
An $Entry symbol was not defined.

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 ADDRESS
An S-Pad op-code was missing its S-Pad Register Address.

38. M MISSING PROGRAM SOURCE ADDRESS
An op-code requiring a program address, such as a JMP or JSR, was missing its address.

39. 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 1) no write into Data Pad X is done, or 2) the indexes are the same for the writes into both DPX and DPY. Examples:

```
Legal:   JSR SQRT;      "Uses the value field
         DPY(2)<FM      "A store into DPY

Legal:   JSR SQRT;      "Uses the value field
         DPX(2)<FA;     "Both Data Pad write indexes
         DPY(2)<FM      " are the same

Illegal: JSR SQRT:      "Uses the value field.
         DPX(-1)<FA;    "the two Data Pad write
         DPY(2)<FM      "indexes are different
```

APPENDIX A
SPECIAL CHARACTER USAGE

Character	Function
+	integer addition operator
-	integer 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
(	preceeds a Data Pad index expression
)	terminates a Data Pad index expression
<	used in DPX, DPY, and MI op-codes
;	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

(INTENTIONALLY BLANK)

APPENDIX B
AP-120B SYMBOLIC OP-CODES

The various AP-120B 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 two symbols are used throughout this appendix:

<>	Indicated optional operands or mnemonics. The item enclosed in the brackets(e.g., <#>) may or may not be coded, depending upon whether or not the associated option is desired.
———	Indicates a specific substitution is required. Substitute the desired address, name, number or mnemonic for the abbreviation underlined.

The following list of abbreviations are used to facilitate the op-code descriptions. They are explained in the section of the op-code group where they first occur:

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 Mulitplier argument #1	B.7
m2	Floating Multiplier argument #2	B.7
dbe	Data Pad Bus enable	B.8
adr	address or value	B.8

B.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 to spd
AND<sh><#> <&>sps,spd	AND sps to spd
OR <sh><#> <&>sps,spd	OR sps to spd
EQV<sh><#> <&>sps,spd	EQuiValence sps to spd

The result

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		Function
LDSPNL	spd	LoaD Spd from PaNeL bus
LDSPE	spd	LoaD SPd from data pad bus Exponent
LDSPI	spd	LoaD SPd from data pad bus Integer (low 16-bits)
		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 Ø 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.

Op-code Examples:
```
MOV 3,6
SUBL 1,15
ADDL#  &PTR, BASE
DEC CTR
CLR 9.
LDSPI 6
```

B.2 MEMORY ADDRESS OP-CODE GROUP

Purpose: to initiate Main Data Memory cycles

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

B.3 TABLE MEMORY ADDRESS OP-CODE GROUP

Purpose: to initiate Table Memory fetches

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

B.4 DATA PAD ADDRESS OP-CODE GROUP

Purpose: to 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

B.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
```

B.6 FLOATING ADDER OP-CODE GROUP

Purpose: Floating-point adds

Double Operand

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		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:

a1 Floating adder argument #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

a2 Floating adder argument #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:

idx Data Pad index: A name, expression, or number which lies in a range of -4 to +3.

Op-code 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.

B.7 FLOATING POINT MULTIPLY OP-CODE GROUP

Purpose: Floating Point multiplies

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

MULTIPLIER OPERANDS:

m1 Multiplier-operand #1

Choices	Meaning
FM	Floating Multiplier output
DPX<(idx)>	Data Pad X
DPY<(idx)>	Data Pad Y
TM	Table Memory

m2 Multiplier-operand #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
```

B.8 DATA PAD X OP-CODE GROUP

Purpose: Storing into Data Pad X

Op-code	Function
DPX<(idx)> <FA	Store Floating Adder output into Data Pad X
DPX<(idx)> <FM	Store Floating Multiplier output into Data Pad X
DPX<(idx)> <DB	Store Data Pad Bus into Data Pad X
DPX<(idx)> <dbe	Store dbe into Data Pad X

ABBREVIATIONS:

dbe Data Pad Bus enable: has the same effect as an explicit Data Pad Bus op-code. (see B.11)

Choices	Meaning
ZERO	Floating zero
adr	adr
DPX<(idx)>	Data Pad X
DPY<(idx)>	Data Pad Y
MD	Memory Data
SPFN	S-Pad Function
TM	Table Memory data

Note: only one choice of Data Pad Bus enable may be made per instruction statement.

adr An address or numeric value. Any 16-bit integer expression is legal. A Floating Multiplier, Memory Input, Memory Address, Table Memory Address, or Data Pad Address op-code may not be used in an instruction statement where an "adr" is used.

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

B.9 DATA PAD Y OP-CODE GROUP

Purpose: Storing into Data Pad Y

Op-Code	Function
DPY<(idx)><FA	Store Floating Adder output into Data Pad Y
DPY<(idx)><FM	Store Floating Multiplier output into Data Pad Y
DPY<(idx)><DB	Store Data Pad Bus into Data Pad Y
DPY<(idx)><dbe	Store dbe into Data Pad Y

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

B.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 reg.
MI<FM	Move Floating Multiplier output to the Mem. Input reg.
MI<DB	Move Data Pad Bus to the Memory Input Register
MI<dbe	Move dbe to the Memory Input Register

Note: to effect a memory write, an op-code from the memory address group, or an "LDMA" op-code must also 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
```

B.11 DATA PAD BUS OP-CODE GROUP

Purpose: to 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

Note: 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
```

B.12 SPECIAL OPERATION OP-CODE GROUP

Note: if an op-code from this group is chosen, an S-Pad Group op-code may not be used in the same instruction statement.

B.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
BFLØ disp	Branch if FLag Ø 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

Note: 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
```

B.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
           JMPA  300
```

B.12.3 SETEXIT

Purpose: to 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

B.12.4 P.S.

Purpose: read/write of 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 Prog. Source Left half at address on Panel bus
RPSFP		Read Prog. Source Floating-point number at address on Panel bus

Note: 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 Prog. Src Left half pointed at by Panel bus
LPSRP		Load Prog. Src Right half pointed at by Panel bus

Note: these op-codes load from the Data Pad Bus

Example: RPSF PI

B.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
RPSØ<A>	adr	Read Program Source quarter Ø 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
RPSØT		Read Program Source quarter Ø 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
WPSØ<A>	adr	Write Program Source quarter Ø 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
WPSØT		Write Program Source quarter Ø 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

B.12.6 HOSTPANEL

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

Op-code	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

B.12.7 Miscellaneous

SPMDA	SPin until a Main Data memory cycle Available

B.13 I/O OP-CODE GROUP

Note: if an op-code is used from this group, a Floating Adder op-code may not be used in the same instruction statement.

B.13.1 Load Reg, Read Reg

Purpose: reading/writing of 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

Note: 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

Note: the above read onto the Panel bus

B.13.2 INOUT

Purpose: Program control/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, then OuTput data, then set DA to spfn

Note: 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, then INput data, then set DA to spfn

Note: the above enable data onto the Input Bus from the I/O device specified by the Device Address Register (DA). To be used the data must be enabled onto the Data Pad Bus, and from there to a register or memory. An example:

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

B.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

B.13.4 FLAG

Purpose: set/reset of program flags

Op-codes	Function
SFLØ	Set Flag Ø
SFL1	Set Flag 1
SFL2	Set Flag 2
SFL3	Set Flag 3
CFLØ	Clear Flag Ø
CFL1	Clear Flag 1
CFL2	Clear Flag 2
CFL3	Clear Flag 3

B.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

INTENTIONALLY BLANK

APPENDIX C

TABLE MEMØRY SYMBØLS:

1. TABLE MEMØRY CØNSTANTS:

SYMBØL	CØNSTANT REPRESENTED	VALUE IN TABLE MEMØRY	TABLE MEMØRY ADDRESS (ØCTAL)
!ZERØ	ZERØ	0.0	4371
!ØNE	ØNE	1.0	4001
!TWØ	TWØ	2.0	4002
!THREE	THREE	3.0	4441
!FØUR	FØUR	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	ØNE THIRD	0.333333333	4430
!FØRTH	ØNE FØURTH	0.25	4431
!FIFTH	ØNE FIFTH	0.2	4432
!SIXTH	ØNE SIXTH	0.166666667	4433
!SVNTH	ØNE SEVENTH	0.142857143	4434
!EGHTH	ØNE EIGHTH	0.125	4435
!NINTH	ØNE NINETH	0.111111111	4436
!TENTH	ØNE TENTH	0.1	4437
!SXNTH	ØNE 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	1.0/SQRT(5)	0.447213596	4453
!ISQ10	1.0/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
!LØG2E	LØG2(E)	1.442695041	4317
!LØG2	LØG10(2)	0.301029996	4411
!LØGE	LØG10(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
!TWØPI	2*PI	6.283185308	4415
!INVPI	1.0/PI	0.318309886	4412
!PI2	PI/2	1.570796327	4312
!PI4	PI/4	0.785398164	4373
!PI180	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

2. ELEMENTARY FUNCTIØN TABLES:

SYMBØL	ELEMENTARY FUNCTIØN	TABLE MEMØRY ADDRESS (ØCTAL)
!DIV	DIVIDE	4000
!SQRT	SQUARE RØØT	4202
!SNCS	SIN/CØS	4306
!LØG	LØGARITHM	4333
!EXP	EXPØNENTIAL	4317
!ATAN	ARC TANGENT	4365

3. SIZE ØF INSTALLED FFT CØSINE TABLE

SYMBØL	SIZE (TYPICAL)
!FFTSZ	2048

EXAMPLE OUTPUT

from

APAL

```
APAL
V2.1
PASS 1

PASS 2
                $TITLE SINCØS
$ENTRY SIN
$ENTRY CØS
"SINE, CØSINE FUNCTIØN
"
"                --- ABSTRACT ---
"CØMPUTES THE FUNCTIØN SIN(X) ØR CØS(X), WHERE X IS IN
"       DPXX(DPA)
"
"                --- STATISTICS ---
"LANGUAGE:       AP-120B ASSEMBLER
"EQUIPMENT:      AP-120B
"STØRAGE:        PS - 31 LØCATIØNS
"                MD - NØT AFFECTED
"                TM - 9
"                DPX - 2
"                DPY - 2
"                SP - NØT USED
"SPEED:          SIN - 4.42 US. AVERAGE  (4.00 - 4.83)
"                CØS - 4.75 US. AVERAGE (4.33 - 5.17)
"AUTHØR:         A.E. CHARLESWØRTH
"DATE:           NØV. 1975
"REVISIØN:       2.2   FEB 76  CHANGED $ENTRY
"
"                --- USEAGE ---
"SAMPLE CALL:    JSR SIN, JSR CØS
"ARGUEMENT:      X IS IN DPX(DPA)
"ANSWER:         SIN(X) ØR CØS(X) IS LEFT IN DPX(DPA)
"SCRATCH:        DPX(0-2), DPY(0-3)
"
"                --- ERRØR CØNDITIØNS ---
"NØNE
"                --- ALGØRITHM ---
"1.      IF CØS(X) IS DESIRED, CØS(X) = SIN(X+PI/2)
"2.      X IS MULTIPLIED BY 2/PI, AND SPLIT INTØ (I + F
"                WHERE I IS AN INTEGER AND F A PØSITIVE
"                BETWEEN 0 AND .9999999999...
"        IF I IS ØDD, F<= (1.0 - F)
"4.      SIN(F) = A + BF**3 + CF**5 + DF**7 + EF**9
"                THE PØLYNØMIAL IS FRØM
"                HART & CHANEY #3341 (PRECISIØN 8.27=5*
"                        RANGE 0 TØ PI/2
"5.      IF I/2 IS ØDD, SIN(F) IS NEGATED
"
"NØTE:   THE PØLYNØMIAL IS FACTØRED AS:
"        (AF + F**3(B + CF**2)) + F**7(D + EF**2)
"
"
"THE TABLE IN TABLE MEMØRY IS AS FØLLØWS:
"SINTBL:0.63661 97724            2/PI
"        0.79689 67894 6 E -1    C
"        0.999999925             FRACTIØN MASK (1000,37
"        1.0                     ØDD BIT MASK
"        0.15707 96318 44 E 1    A (PI/2)
"        0.15148 5129 E-3        E
"        -.64596 37105 99        B
"        -.46737 6661 E-2        D
"        2.0                     NEXT ØDD BIT MASK
"
```

```
                    "TABLE LOCATION     TM:
004306                      SINTBL $EQU !SNCS

                    "DATA PAD X:
000000                      X $EQU 0

000001                      TEMPX $EQU 1

                    "DATA PAD Y:
000000                      C $EQU 0

000000                      AF $EQU 0

000001                      F $EQU 1

000001                      F2 $EQU 1

000001                      F3 $EQU 1

                    "COME HERE FOR COSINE(X).................
000000  000003      COS:    LDTMA; DB=SINTBL+4          "FETCH PI/2 (A)
        103000
        002000
        004312

000001  000003              DB=SINTBL; LDTMA            "FETCH 2/PI
        103000
        002000
        004306

000002  000001              FADD TM,DPX(X)              "D0 X*2/PI
        142000
        000400
        000000

000003  000001              FADD; INCTMA                "FETCH C
        100000
        000000
        000001

000004  000000              FMUL TM,FA;                 "D0 X*2/PI
        000124                INCTMA;                   "  FETCH FRACTION MASK
        000000                BR COMMON                 "  G0 D0 SIN(X+PI/2)
        016001

                    "COME HERE FOR SINE(X)...................
000005  000003      SIN:    LDTMA; DB=SINTBL            "FETCH 2/PI
        103000
        002000
        004306

000006  000000              INCTMA                      "FETCH C
        000000
        000000
        000001

000007  000000              FMUL TM,DPX(X);             "D0 X*2/PI
        000000                INCTMA                    "  FETCH FRACTION MASK
        000400
        016401

000010  000000  COMMON:  FMUL; INCTMA;                  "FETCH 0DD BIT MASK
        000000              DPY(C)<TM                   "  SAVE C
        017000
        110001
```

```
000011  000000          FMUL; D:  (EMPX)<TM;     "SAVE FRACTIØ. MASK
        000000            INCTMA                 "  FETCH A
        047005
        010001

                "ØH, THE GLØRIES ØF A FLØATING PØINT AND...
                "       WE CAN GET THE FRACTIØNAL PART ØF A NUMBER,
                "       ØR
                "       WE CAN TEST TØ SEE IF THE INTEGER PART ØF
                "       A NUMBER IS ØDD...
000012  000002          FAND FM,DPX(TEMPX);      "GET F = FRAC(2X/PI)
        112000            DPX(TEMPX)<TM          "  SAVE ØDD BIT MASK (
        047505
        000000

000013  000002          FAND FM,DPX(TEMPX);      "SEE IF I IS ØDD
        112000            DPX(X)<FM              "  SAVE 2X/PI
        140504
        000000

                "HERE WE PLUNGE AHEAD AND START THE PØLYNØMIAL,
                "       HØPING THAT WE AREN'T IN THE
                "       SECØND ØR FØURTH QUÄDRENTS
000014  000001  ØDD:    FMUL DPY(C),FA;          "DØ C*F
        021000            DPY(F)<FA;             "  SAVE F
        020540            FSUB DPX(TEMPX),FA     "  DØ 1.0-F
        134000

000015  000001          FMUL DPY(F),DPY(F);      "DØ F**2
        155000            FADD ZERØ,ZERØ         "  PUSH 1-F ØUT
        000050
        015000

                "IF I IS ØDD, THE FAND WILL PRØDUCE 1.0 AS A RESULT,
                "       WHICH WE NØW TEST FØR...
                "THIS IS NØT AN INFINITE LØØP
                "       IF I IS ØDD, AND WE CØME THIS WAY AGAIN,
                "       THE SECØND TIME WE WILL BE TESTING THE RESULT
                "       ØF ADDING 0.0 + 0.0, WHICH WILL
                "       ALWAYS BE ZERØ
000016  000000          FMUL TM,DPY(F);          "DØ A*F
        000456            BFNE ØDD               "  BRANCH BACK IF I IS
        000050
        017000

                "NØW WE ARE INTØ THE PØLYNØMIAL FØR GØØD.........
000017  000000          FMUL FM, DPY(F)          "DØ CF * F
        000000
        000050
        011000

000020  000000          FMUL FM,DPY(F);          "DØ F**3
        000000            DPY(F2)<FM;            "  SAVE F**2
        030050            INCTMA                 "  FETCH E
        131001

000021  000000          FMUL DPY(F2),DPY(F2);    "DØ F**4
        000000            DPY(AF)<FM;            "  SAVE AF
        030050            INCTMA                 "  FETCH B
        115001
```

```
000022  000000          FMUL TM,DPY(F2);          "DØ E * F**2
        000000            DPX(TEMPX)<FM;          "  SAVE CF2
        140055            INCTMA                  "  FETCH D
        017001

000023  000001          FADD TM,DPX(TEMPX);       "DØ B+CF2
        142000            FMUL;
        030500            DPY(F3)<FM              "  SAVE F**3
        130000

000024  000001          FMUL FM,DPY(F3);          "DØ F**7
        100000            DPX(TEMPX)<TM;          "  SAVE D
        047055            FADD
        011000

000025  000001          FMUL DPY(F3),FA;          "DØ F**3 * (B+CF2)
        112000            FADD FM,DPX(TEMPX);     "  DØ D + EF2
        000550            INCTMA                  "  FETCH NEXT ØDD BIT
        014001

000026  000001          FMUL; FADD                "WAIT
        100000
        000000
        010000

                "HERE WE SEE IF MØD(I/2,2) IS 1 ØR 0.........
000027  000002          FMUL FM,FA;               "DØ F**7 * (D+EF2)
        142000            FAND TM,DPX(X)          "  SEE IF I/2 IS ØDD
        000400
        010000

000030  000001          FADD FM,DPY(AF); FMUL     "DØ AF + (BF3+CF5)
        113000
        000040
        010000

000031  000001          FMUL; FADD                "WAIT
        100000
        000000
        010000

000032  000001          FADD FM,FA;               "DØ (AF+BF3+CF5)+(DF7+
        111463            BFNE NEG                "  SEE IF WE NEED TØ N
        000000
        000000

000033  000001  DØNE:   FADD                      "WAIT
        100000
        000000
        000000

000034  000000          DPX(X)<FA; RETURN         "STØRE ANSWER AND RETU
        000340
        100004
        000000

                "CØME HERE IF WE NEED TØ NEGATE THE ANSWER....
000035  000001  NEG:    FADD                      "WAIT
        100000
        000000
        000000
```

```
000036  000001          FSUB ZERØ,FA;          "0.0 - ANSWER
        051115            BR DØNE              "  GØ FINISH
        000000
        000000

                        $END


****      0 ERRØRS    ****


SYMBØL   VALUE

SINTBL   004306
X        000000
TEMPX    000001
C        000000
AF       000000
F        000001
F2       000001
F3       000001
CØS      000000 ENT
SIN      000005 ENT
CØMMØN   000010
ØDD      000014
DØNE     000033
NEG      000035
```

(INTENTIONALLY BLANK)

AP-120B APLINK

Array Processor Linking Loader Manual

7276-02

FPS-7276-01

Revision 02, March 30, 1976

TABLE OF CONTENTS

SECTION 1
INTRODUCTION

1.1 INTRODUCTION

APLINK links separate object modules produced by APAL together into a single load module for execution by the AP-120B 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 be performing the following tasks:

1. Relocating each object module and assigning absolute addresses.

2. Linking the modules together by correllating global entry symbols defined in one module with external symbols referenced in another module.

3. Selectively loading modules from program library.

4. Optionally producing a load map showing the layout of the load module.

APLINK is written in Fortran IV and requires roughly 10K of available memory in which to operate.

INTENTIONALLY BLANK

SECTION 3
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
###
*
```

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

To load his 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, the user must 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, the user should correct the error and restart the loading operation from step 1.
3. Obtain the memory limits of the loaded program and/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 "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 3.12.

The three 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.
↓	Carriage Return

——— Indicates characters output by the program.

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

3.1 LOAD, "L"

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

```
L↓
filename↓
```

where "filename" is the name of the file containing the desired program or library. Example:

```
* L↓
FFT. RB↓
Loads a program from file
        FFT. RB.
```

3.2 SYMBOLS, "S"

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

```
S↓
filename↓
```

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    U
  .               .
  .               .
  .               .
```

where:

aaaaaa — Highest program address so far loaded. Normally, the next program will be 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.

An example command:

```
*S↓
LP:↓
```

Dumps the loader symbol table onto the line printer.

3.3 UNDEFINED, "U"

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

```
U↓
filename↓
```

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. An example command:

```
*U ↓
TP:↓
```

Prints the names of any undefined symbols on the teletype.

3.4 NEXT BASE, "B"

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

```
B↓
loc ↓
```

where "loc" is the location specified. An example:

```
*B ↓
200 ↓
```

Sets the next location loaded to location 200.

3.5 RESET, "R"

To reset APLINK, enter:

R↓

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.

3.6 FORCE, "F"

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

F↓
name↓

where "name" is the name of the symbol to be forced. This command enters "name" into the symbol table as an external symbol. This will cause the loading of a library program which has "name" as an entry symbol. An example:

*F↓
DOTPRD↓

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

3.7 MEMORY, "M"

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

M↓

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.

3.8 END, "E"

To end a load module and output the completed load module for use with APDEBUG, enter:

E↓
filename↓

where "filename" is the name of the file to receive the loader output. The output is a "core image" which can be loaded by APDEBUG 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:

```
###   UNDEFINED SYMBOLS
```

where "###" is the number still undefined. A value of Ø was used in linking these undefined symbols.

```
*E↓
 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 mode, a reset ("R") command must be given to clear the linker.

3.9 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↓
filename↓
```

where "filename" is the name of the file to receive the loader output. This output is a short host assembly language subroutine, which is the linkage between host computer Fortran "CALL's" and the AP-120B executive. The AP-120B 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.

3.10 NUMBER RADIX, "N"

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

```
N↓
radix↓
```

where the 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.

3.11 EXIT, "X"

To exit to the operating system, enter:

X↓

Note: X does not cause any output. An "E" or "A" must be used to output a load module.

3.12 AN EXAMPLE LOADING SESSION

```
>>RUN APLINK
APLINK
REV 1
*L
BENCH:RB
APLIB
*S
/TTØ
HIGH=000116


SYMBOL TABLE'

SYMBOL VALUE
BENCH 000000
DIV   000063


*E
TEMP
HIGH=000116
*X
STOP

>>
```

The user runs APLINK. He then loads his program, in file BENCH:RB. Since Bench uses the scalar divide subroutine, he links in the library of AP-120B subroutines, APLIB. APLINK extracts from the library any subroutines needed by BENCH, in this case DIV. The user prints out the loader map, and then ends with an "End" command, putting the linked-up code into file TEMP.

To debug the program, the user would run APSIM, where he could execute the code put into TEMP.

Another example loading session.

```
RUN APLINK
APLINK
REV 1
*F
POLAR

*L
APLIB
*S
/TTØ
HIGH=000166

SYMBOL TABLE'

SYMBOL VALUE
POLAR  000000
SQRT   000133
ATN2   000023
ATAN   000035
DIV    000077

*A
TEMP
HIGH=000166
*X
STOP

>>
```

The user runs APLINK. He wants to install subroutine POLAR into his computer operating system, so that he can CALL POLAR from Fortran. The "F" (force) command sets up APLINK so that it will load in POLAR from the library APLIB. The loader map shows that POLAR used subroutines SQRT (square root), ATN2 and ATAN (arc-tangent), and DIV (divide). The "A" command stores the linked-up code as host assembly language in file TEMP.

The host assembly code in TEMP is assembled by the host computer assembler and the resulting relocatable binary saved where it can be linked with Fortran programs.

INTENTIONALLY BLANK

SECTION 4
ERROR MESSAGES

Any deviation from the prescribed command syntax will cause 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 outputted and the command is ignored.

The specific error messages outputted 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 will cause immediate termination of the "L" (load) command following the error message. If the user attempts to execute another "L" command, the program will output the message:

RESET!!!

and ignore 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 past the upper limit of Program Source Memory. The load module is too large to fit in program 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 offending symbol is output to the console:

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 put out host assembly code (an "A" command) from a load module which either 1) did not have any entry points (defined entry global symbols), or 2) the first entry point loaded did not have an S-Pad parameter count.

APPENDIX A
SUMMARY OF APLINK COMMANDS

These abbreviations are used:

Symbol	Meaning
↓	Carriage Return
filename	Name of a file, as appropriate for the host operating system being used.
loc	A location, in octal or hex as appropriate
name	A symbol name, 6 characters or less.

Command	Effect
L↓ filename↓	Load the program in file FILENAME, link with previously loaded programs.
S↓ filename↓	Output the loader symbol table to file FILENAME.
U↓ filename↓	Output any undefined symbols to file FILENAME.
B↓ loc↓	Set APLINK to load the next program at location LOC.
R↓	Reset the loader.
F↓ name↓	Force the loading of a program defining symbol. Name from any subsequent program libraries loaded.
M↓	Output the highest program memory location used.
E↓ filename↓	End the loading session. Store the resulting load module into file FILENAME.
A↓ filename↓	End the loading session. Output host computer assembly code for use with APEX into file FILENAME.
N↓ number↓	Set the Radix for numeric user console I/O to either 8, 1Ø, or 16.
X↓	Exit to the operating system.

INTENTIONALLY BLANK

APPENDIX B
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 is that relocatable library files may be edited with an ordinary text editor. This makes unneccessary the need for a special-purpose Librarian or 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 (Ø)
4. External Blocks (5)
5. End Block (1)

An object module contains at least a Title Block and a Start/End Block. The presence of one or more of the other block types will depend 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 below, in numeric order by block type numbers

B.1 CODE BLOCK (Ø)

LINE	CONTENTS:			
Ø	Ø	count	address	Ø***
1	Bits 0-15	Bits 16-31	Bits 32-47	Bits 48-63
2	"	"	"	"
⋮	⋮	⋮	⋮	⋮
count	"	"	"	"

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

B.2 END BLOCK (1)

LINE	CONTENTS			
0	1	0	0	0***

B.3 TITLE BLOCK (3)

LINE	CONTENTS			
0		1	0	0***TITLE
1		0		

B.4 ENTRY SYMBOL BLOCK (4)

LINE	CONTENTS			
0	4	count	0	0***
1	name	value		# S-Pad parameters
2	"	"		"
:	:	:		:
count	"	"		"

B.5 EXTERNAL SYMBOL BLOCK (5)

LINE	CONTENTS			
0	5	count	0	0***
1	name	link		
2	"	"		
3	"	"		
:	:	:		
count	"	"		

B.6 LIBRARY START BLOCK (6)

LINE	CONTENTS			
0	6	0	0	0***

B.7 LIBRARY END BLOCK (7)

LINE	CONTENTS			
0	7	0	0	0***

An example relocatable object module, from the Dot Product program.

```
     3.      1.      0.      0.***TITLE
DØTPR        0.
     4.      1.      0.      0.***
DØTPR        0.      6.
     0.      9.      0.      0.***
 16384.      0.      0.     48.
 16520.      0.      0.     48.
   596.      0.  18948.      0.
  8257.  55808.      0.     48.
   661.  32768.    256.   5888.
  8392.    403.      0.   4144.
     0.      0.  18948.   4096.
  8257.  37453.      0.     48.
 16656.    224.      0.    112.
     1.      0.      0.      0.***
```

Title block

Entry block

Code block

End block

The Title block contains the title of the program, DOTPR

The Entry block has the name of the entry point, DOTPR; its relative address, Ø; and the number of expected S-Pad parameters, 6.

The Code block contains the 9 AP-120B program words in DOTPR, each as four 16-bit quarters of a 64-bit program word.

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

Example output from APLINK produced by an "E" (End) command from the Dot Product program. APDBUG would load this output into either the simulated AP-120B (APSIM),or the actual hardware, for debugging.

```
    9                    number of AP-120B program words
16384
    0
    0                    word #1
   48
16520
    0
    0                    word #2
   48
  596
    0                    word #3
18948
    0
 8257
-9728
    0                    word #4
   48
  661
-32768
  256                    word #5
 5888
 8392
  403
    0                    word #6
 4144
    0
    0
18948                    word #7
 4096
 8257
-28083
    0                    word #8
   48
16656
  224
    0                    word #9
  112
```

Example host assembly code produced by APLINK from the Dot Product program.

```
        .TITL DØTPR
        .ENT DØTPR
        .EXTD .APEX
        -1
DØTPR : JSR@ .APEX
            6.
            9.
            0.
        16384.
            0.
            0.
           48.
        16520.
            0.
            0.
           48.
          596.
            0.
        18948.
            0.
         8257.
        -9728.
            0.
           48.
          661.
       -32768.
          256.
         5888.
         8392.
          403.
            0.
         4144.
            0.
            0.
        18948.
         4096.
         8257.
       -28083.
            0.
           48.
        16656.
          224.
            0.
          112.
        .END
```

This is the appropriate host computer assembly code for use with Data General Corporation Fortran IV on Nova or Eclipse computers.

A CALL DOTPR in Fortran ends up at location "DOTPR" in the Nova, which does a subroutine jump to APEX, the AP-120B executive.

If this is the first CALL of DOTPR, then APEX will load the AP-120B program words for DOTPR from Nova memory into AP-120B Program Memory.

The "6" following the JSR@ .APEX is the number of arguements expected in the Fortran CALL. "9" is the number of AP-120B program words in DOTPR. "0" is the re;ative starting address of DOTPR.

Following these three parameters is the nine AP-120B program words in DOTPR, 16-bits at a time.

For each particular host computer, the exact form of the host computer assembly language is different, but the content is the same.

INTENTIONALLY BLANK

AP-120B APDEBUG
DEBUGGER MANUAL
7277-02

FPS-7277-01

REVISION 02, MARCH 31, 1976

TABLE OF CONTENTS

APPENDIX A
AP-120B SUMMARY OF DEBUG COMMANDS

Abbreviations used below:

Symbol	Meaning
↓	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-120B internal memory
reg	The name of an AP-120B internal register

Debug types a "*" when ready for further action.
A "?" is typed when a command is not understood.

A.1 Program Execution Commands

B↓ mem↓ loc↓	Breakpoint. Delete the last breakpoint and set a new breakpoint at location LOC of memory MEM. MEM must be PS, MD, or TM.
D↓	Delete. Delete the current breakpoint
L↓	List. List the current breakpoint
Q↓ count	Set the continue counter to (COUNT).
S↓ val↓	Step. If (VAL) is not zero place the AP-120B in step mode.
I↓ val↓	Initialize. If VAL is not zero, reset the AP-120B before program execution is resumed next
R↓ loc↓	Run. Begin program execution at Program Source location LOC
P↓	Proceed. Begin instruction execution at the Program Source location pointed to by the AP-120B (PSA) (Program Source Address) Register.
X↓	Exit to the operating system

A.2 Register Examination/Modification Commands

E↓
reg↓
: Examine register. Print out the contents of AP-120B register REG

E↓
mem ↓
loc ↓
: Examine memory. Pring out the contents of AP-120B memory MEM, location LOC

. ↓
: Re-examine the currently open register or memory location (the last location examined)

+ ↓
: Examine the next higher sequestial memory location of the memory that is currently open

- ↓
: Examine the next lower sequential memory location of the memory that is currently open

F ↓
val ↓
: 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≠0: floating-point

V ↓
val ↓
: 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≠0: Decode into the 24 instruction word field values.

C ↓
val ↓
: 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↓ three integers in the current radix
VAL↓ thich represent the exponent, high
VAL↓ mantissa, and low mantissa

F≠Ø: FPN↓ a floating point number legal to Fortran

64-bit wide registers:
V=0 VAL↓ four integers in the current radix
VAL↓ which are the four quarters of an AP-120B
VAL↓ program word
VAL↓

V≠0: FIELD↓ FIELD is the name of the instruction
VAL ↓ field to be changes, VAL is the new integer value.

N↓ VAL↓	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↓ VAL↓	Offset. Sets the base address to which Program Source Memory addresses are relative (for user I/O).
Z	Zero. Zero out all AP-120B memories and registers.

A.3 Memory Load/Dump Commands

Y↓ MEM↓ LOC↓ filename↓	Yank. Load memory MEM starting at location LOC from an external data FILENAME. MEM can be PS, MD, OR TM.
W↓ MEM↓ START↓ STOP↓ file name↓	Write. Dump memory MEM starting at location (START) and ending at location (STOP) to external data FILENAME. MEM can be PS, MD, or TM.

A.4. Accessable Functional Units

AP-120B Functional Units that may be examined or changed using DEBUG:

Memories:		Contents:
PS	Program Source Memory	64-bit instruction word
MD	Main Data Memory	38-bit floating-point
TM	Table Memory	" "
DPX	Data Pad X	" "
DPY	Data Pad Y	" "
IODEV	I/O Devices	" "
SP	S-Pad Registers	16-bit integer
SRS	Subroutine Return Stack*	" "

Registers		Contents
MA	Memory Address	16-bit integer
TMA	Table Memory Address	"
DPA	Data Pad Address	"
PSA	Program Source Address	"
SPD	S-Pad Destination Address	"
STAT	AP-120B Internal Status Register	"
DA	I/O Divice Address	"
SWCH	Panel Switch Register	"
LGTS	Panel Lights Registerr	"
MDR	Memory Read Data Buffer*	38-bit floating point
TMR	Table Memory Data Buffer*	"
MI	Memory Input Register*	"
DPBS	Data Pad Bus*	"
INBS	Input Bus*	"
PNBS	Panel Bus*	16-bit integer
SPFN	S-Pad Function	"
FLAG	Program flags*	"
SRA	Subroutine Return Stack Address*	"
A1	Floating Adder Input Reg. #1*	38-bit floating point
A2	Floating Adder Input Reg. #2*	"
FA	Floating Adder Output*	"
M1	Floating Multiplier Input Reg. #1*	"
M2	Floating Multiplier Input Reg. #2*	"
FM	Floating Multiplier Output*	"

*Accesable only when using the AP-120B Simulator

A.5 Program Word Fields

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

FLOATING POINT SYSTEMS, INC.
P.O. BOX 23489 PORTLAND, OR 97223 11000 S.W. 11TH STREET, BEAVERTON, OR 97005 (503) 641-3151 TLX: 360470 FLOATPOINT PTL