# FLOATING POINT

## INTRODUCTION

The floating point subroutines for Exec III have been written. The routines are accessible by IOT's, as with the other "Common Routines". There are twelve subroutines; four perform mathematical functions, six convert between floating point form and some other form, and two move information. They will be discussed individually later.

## THE NUMBERS

Each floating point number takes up two contiguous storage registers. The 18 bits of the first register and the first 1Ø bits of the second comprise the 28-bit ones-complement mantissa, A, where -1<A<+1. The last 8 bits of the second register comprise the 8-bit ones-complement binary characteristic, B, which ranges from -127 through +127 (decimal). The address of the first register of the pair is taken to be the address of the floating point number. The value of the floating point number is taken to be "A times 2-to-the-B". Thus the range of a positive floating point number, X, is exactly 2.9387358E-39<X<1.7Ø14118E+38, where "E" means "times-ten-to-the".

A floating point number is considered to be "normalized" if both registers are +Ø or if the sign bit and bit 1 of the mantissa are different. For example, the constant "1." may be represented in any of the following ways:

ØØØØØØ-ØØØ433
ØØØØØ1-ØØØØ21
2ØØØØØ-ØØØØØ1

The preceding numbers are octal. Only the last of the three pairs is the normalized form of the floating point number 1. The routines (except flac and fdac, which simply move information) all leave their results in normalized form. Further, the time-sharing version of MIDAS eventually will be equipped with pseudo-instructions to facilitate generation of normalized floating point constants. Consequently, it is assumed that there is no source of un-normalized numbers in the system, and the routines all expect the numbers they are given to be in normalized form. They do unspecified (but guaranteed bad) things with un-normalized numbers, so caveat user.

## THE SYSTEM

For each routine, a symbol or macro is defined and entered on the system p-tape. These are the calling sequences for the routines and are each two words long, except for flip, which consists simply of a symbol and requires only one register. The first word of each calling sequence is an IOT, which causes a trap, and the remaining word contains the argument(s). The definitions are listed in the Appendix.

The time-sharing version of MIDAS will have two pseudo-instructions to aid the programmer in the assembly and storage of floating point constants. They are called "lfloat" and "rfloat" and have the value, respectively, of the first word or second word of the floating point representation of the number that

follows them. In this way they function similarly to the "flexo" pseudo-instruction. For example, the floating point representation of the number "3.Ø" is Ø11ØØØ-ØØØØØ2. Thus the expression "1float 3.Ø" in a MIDAS source-language program is equivalent to writing "Ø11ØØØ"; and similarly, "rfloat 3.Ø" is equivalent to "ØØØØØ2" or, simply, "2".

The argument of the pseudo-instruction can be any text string acceptable to the Floating Input routine (q.v.). Two macros, "cfloat" and "pfloat", using these pseudo-instructions, may be entered on the p-tape. Each generates a pair of registers containing the floating point representation of its argument. Both macros further simplify the production of floating point constants by the programmer. Cfloat might be used as follows:

```
const1,          cfloat 3.1416
```

This would generate a floating point constant whose value is 3.1416 and whose address is "const1". Pfloat is a "remote" macro which is valuable in cases where a number of floating point constants are each to be used only once in a source program. The expression "pfloat 1.414", in conjunction with the later use of the "here" macro, automatically generates a two-register floating point constant 1.414 in the area designated by the "here". Further, the expression itself has as its value (on pass 2) the address of the constant. See the Appendix for the definitions of cfloat and pfloat.

There is a two-register block in the user's core called the floating accumulator, to which all the floating point routines

refer. The value of the symbol "fac", defined on the system p-tape, is equal to the address of the first word of the floating accumulator.

There is also a register called "fov". When an exponent overflow or underflow occurs (a result is either too large or too small to be represented in the floating point format), the user's overflow flip-flop is set and a jump instruction is placed in "fov+1". The address part of the jump instruction is the address to which control would have been returned if the overflow had not occurred, and this address is, in general, that of the register following the floating point calling sequence. (For the two-return routines, fdiv and flip, it is the second return in each case.) Then control is returned to register "fov". (The p-tape macro assembles a "zero" into fov.) The user is expected to detect the overflow by placing in "fov" a jump instruction to his own routine to handle overflows. Later, if he wishes, he may return control to his main sequence by executing a "jmp fov+1". In some cases it may be preferable to test a series of floating point calculations after they are completed to see whether any one of them caused an overflow. Since the routines are transparent to the overflow flip-flop except when an overflow occurs, this may be accomplished by clearing the overflow flip-flop before the series and testing it afterwards. Consequently, it may be desirable to ignore overflows temporarily when they occur. If the user places a "nop" in "fov" before calling the floating routines, when an overflow occurs the overflow flip-flop gets set and control flows through "fov+1" back to the user's main sequence, temporarily ignoring the overflow. Division by zero is not considered an exponent overflow, and its detection is accomplished in a different manner: see description of the Floating Divide routine.

It is worth noting that, since the routines place their result in the floating accumulator at the end of the calculation, when an overflow occurs the previous contents of the floating accumulator are unchanged. At present the routines give the user no indication whether an exponent "overflow" was really an overflow or an underflow. If there is sufficient demand for such an indication, it will be provided, but currently it is felt that this feature would be too costly of space.

## THE ROUTINES

The floating point routines are transparent to all registers and statuses that they do not expressly modify. They share the region of temporary storage in lower core defined on the p-tape as "atem" with the other Common Routines. None of them ever changes the program flags, link bit, or ring mode. On overflows, they set the overflow flip-flop but are otherwise transparent to it. Only "fix" and "fix2" ever change the AC or I-O. All addresses in the calling sequences may be indirect, and the routines will follow an indirect-address chain to any level. The routines can all be called in ring mode.

### 1. The Floating Add Routine.

Call: fadd x

This routine adds the floating point number found at address x to the contents of the floating accumulator and leaves the result in the floating accumulator. It normally returns to the first line following the fadd but is subject to exponent

overflow trap (see above). (It leaves the AC, IO, and program flags unchanged, as well as the registers x and x+1.)

2. The Floating Subtract Routine.

Call: fsub x

This routine functions exactly like the fadd routine except it subtracts.

3. The Floating Multiply Routine.

Call: fmul x

This routine functions exactly like the fadd routine except it multiplies.

4. The Floating Divide Routine.

```
Call:  fdiv x
       jmp err
```

This routine divides the contents of the floating accumulator by the number found at x and leaves the result in the floating accumulator. It is subject to exponent overflow (see above). When the divisor is not equal to zero, the return is to the second line following the fdiv. When the divisor is zero, the divide does not take place and the exit is to the line immediately following the fdiv.

5. The Floating lac Routine.

Call: flac x

This routine simply replaces the contents of fac and fac+1 with the contents of x and x+1, respectively. It does not attempt to normalize the floating point number found at x before placing it in the floating accumulator.

6. The Floating dac Routine.

```
Call:  fdac x
```

This routine simply replaces the contents of x and x+1 with the contents of fac and fac+1, respectively. It does not attempt to normalize the floating point number found in the floating accumulator before storing it at x.

7. The Floating Input Routine.

```
Call:  flip
       jmp error
```

This routine is the floating point analog of dnm and converts a text string to a floating point number. Fsa contains a character pointer.

Starting at the location designated by fsa, characters are interpreted one by one, and a floating point number is developed, until a character is encountered which is incompatible with the foregoing ones or which is not a constituent of a legal floating input text string. (The only characters acceptable to flip are space, plus, minus, decimal point, "e", and digits Ø through 9.) The floating point number which has been developed is put in the floating accumulator, and a pointer to the first non-interpretable character is left in register "fsa" (which is

defined on the system p-tape). The routine is subject to exponent overflow (see above). If a legal input string is found, control is returned to the second line following the flip. If the combination makes no sense, control is returned to the line immediately following the flip, and the floating accumulator is unchanged. In any case, a pointer to the first non-interpretable character is left in fsa. Following are some legal and some illegal examples of input text strings. The arrow points to the first non-interpretable character in each case.

The character "e" means "times-ten-to-the". These examples are far from exhaustive, but the user can easily resolve any questions about the legality of a text string by referring to the diagram that has been included in the appendix.

The following examples are all legal strings. In each case the second column contains the floating point number that will be put in the floating accumulator.

| | | |
|---|---|---|
| 1.Øabc | 1. | (All letters except "e" are non-interpretable) |
| Ø.1.35- | Ø.1 | (A second decimal point doesn't make sense) |
| +1.3e5 a | $1.3 \cdot 1Ø^5$ | (Space is only interpretable in place of "plus" sign) |
| -1.6-3e5bq | -1.6 | (The second minus sign has no meaning) |
| +1.6e+7-3. | $1.6 \cdot 1Ø^7$ | (The minus sign has no meaning except immediately after the "e" or as first character) |

| | | |
|---|---|---|
| 1e-1.6 | $1 \cdot 1 \emptyset^{-1}$ | (A decimal point has no meaning in the exponent field) |
| 1e1e1 | $1 \cdot 1 \emptyset^{1}$ | (The second "e" has no meaning) |
| ø,3 | ø | (Commas and all other special characters are non-interpretable) |

The following examples are all illegal. The return will be to the first line after the flip with the floating accumulator unchanged.

```
abcdef
1.e aa
1e.aaa
.e1 aa
+-3aaa
++3aaa
..3aaa
+e+1
```

8. The Floating Output Routine

Call: flop flags, acc, dig, col

or: flop flags, acc, dig, col

This routine is the floating point analog of snm and converts the floating point number that is contained in the floating accumulator to internal code. Register "sts" (defined on the

system p-tape) contains the character pointer to the place where text is to be stored, or the text string is typed out directly. In the latter case, sts is ignored and no text is stored. The text is stored, packed three to a word, and a pointer to the first unused character position is placed in register sts. In the non-type-out case, an eom is stored after the text string, but the pointer in sts is not incremented past it. "Acc" is a number from one through eight and is the number of significant figures to which the routine is to round the eight-significant-figure result that represents its maximum precision. In the interest of neat-looking printouts, it is best not to call for the full 8-digit precision unless necessary; for, due to the nature of the algorithm, the routine will "string nines" rather than zeros whenever it can, and numbers like "1.ØØØØØØØØ" will come out as ".99999999". If 7-digit accuracy is specified, however, the routine will round the number in the eighth significant figure and will store "1.ØØØØØØØØ" instead. The fourth argument of the macro, "dig", is a number from Ø through 777 (octal) and represents the number of digits to be stored to the right of the decimal point. This is a rudimentary type of format control and is totally independent of the "acc" specification. If necessary, a round-off will be performed on the first digit that is too many places past the decimal point to be stored. If both the "acc" and "dig" specifications would cause round-offs independently, the routine determins which round-off is more significant (i.e., farthest left) and performs that one. Note that the full "dig" digits are still stored to the right of the decimal point, however, filled out by trailing zeros if necessary.

The text can be stored in either of two formats, selected by bit 13 of the IOT. The "f" format is that of ordinary decimal numbers. Examples: " .Ø1", " 3.1416", "-1ØØØ.ØØ", " 2.". The "e" format is that of "scientific notation". Examples: " 1.86273e Ø5", "-3.e 21", " 3.Ø2e-13". The "e" format consists of a space or minus sign, one digit, a decimal point, "dig" more digits, an "e", a space or minus sign, and exactly two more digits. In both formats, the first character stored is always a space or minus sign, and a decimal point is always stored.

The low 6 bits of the IOT are referred to as the "flags". As in all IOT's these 6 bits specify various options which are available to the user. On the "flop" these options are as follows:

| bit | number to add to IOT | option |
|---|---|---|
| 17 | 1 | type out result directly |
| 16 | 2 | right adjust number in field specified by "col" |
| 13 | 2Ø | place result in "e" format (otherwise result in "f" format) |

The right-adjust feature allows columnation of floating point numbers. If the number of characters in the floating point number is less than or equal to "col" then the appropriate number of spaces are generated (i.e., typed or stored) so that the floating point number will appear right-justified. If the number of characters in the floating point number is greater than "col", no spaces are generated (i.e., this feature then has no effect.)

As in all set-up IOT's, if the "type" bit (bit 17) is set and the Job Hunter register "jmode" is negative the IOT does nothing.

9. Fixed point to floating point conversion — single precision.

Call: float n

This routine takes the fixed point contents of the accumulator, multiplies it by $2^n$, and converts it to a floating point number, which it places in the floating accumulator. (The AC is unchanged.) For extreme values of n, exponent overflow trap can occur. Since the binary point of most fixed point numbers is to the right of bit 17, conversion ordinarily is made by "float Ø". If the binary point of the fixed point number is to the right of bit Ø instead, conversion is made by "float -17.".

1Ø. Fixed point to floating point conversion — double precision.

Call: float2 n

This routine takes the contents of the combined AC and I-O, which it treats as a 36-bit fixed point double precision integer, multiplies it by $2^n$, and converts it to a floating point number, which it places in the floating accumulator. For extreme values of n, exponent overflow trap can occur. Since the binary point of most fixed point double precision numbers is to the right of bit 35, conversion ordinarily is made by "float2 Ø". If the binary point of the number is to the right of bit Ø instead, conversion is made by "float2 -35.".

11. Floating point to fixed point conversion — single precision.

Call: fix n

This routine converts the floating point number in the floating accumulator to fixed point without rounding. The fixed point number is left in the ac. (The contents of the floating accumulator are unchanged.) The binary point of the fixed point number is determined by n. If n is ∅, the binary point is to the right of bit 17. If n is +17., the binary point is to the right of bit ∅. If the fixed point number is too large for an 18-bit word, an exponent overflow trap occurs, and the accumulator remains unchanged.

12. Floating point to fixed point conversion — double precision.

Call: fix2 n

This routine converts the floating point number in the floating accumulator to a 36-bit fixed point double precision number without rounding. The fixed point number is left in the combined AC and I-O. The binary point of the fixed point number is determined by n. If n is ∅, the binary point is to the right of bit 35. If n is +35, the binary point is to the right of bit ∅. If the fixed point number is too large for a 36-bit word, an exponent overflow trap occurs, and the AC and I-O) are unchanged.

# APPENDIX

## FLOATING POINT SYMBOLS AND MACROS

```
flip=IOT 126ØØ

define      flop flags, acc, dig, col
            IOT 14ØØØ flags
            col"T"1ØØØØ + acc"T"1ØØ + dig
terminate

define      fadd x
            IOT 127ØØ
            x
terminate

define      fsub x
            IOT 1274Ø
            x
terminate

define      fmul x
            IOT 13ØØØ
            x
terminate

define      fdiv x
            IOT 131ØØ
            x
terminate
```

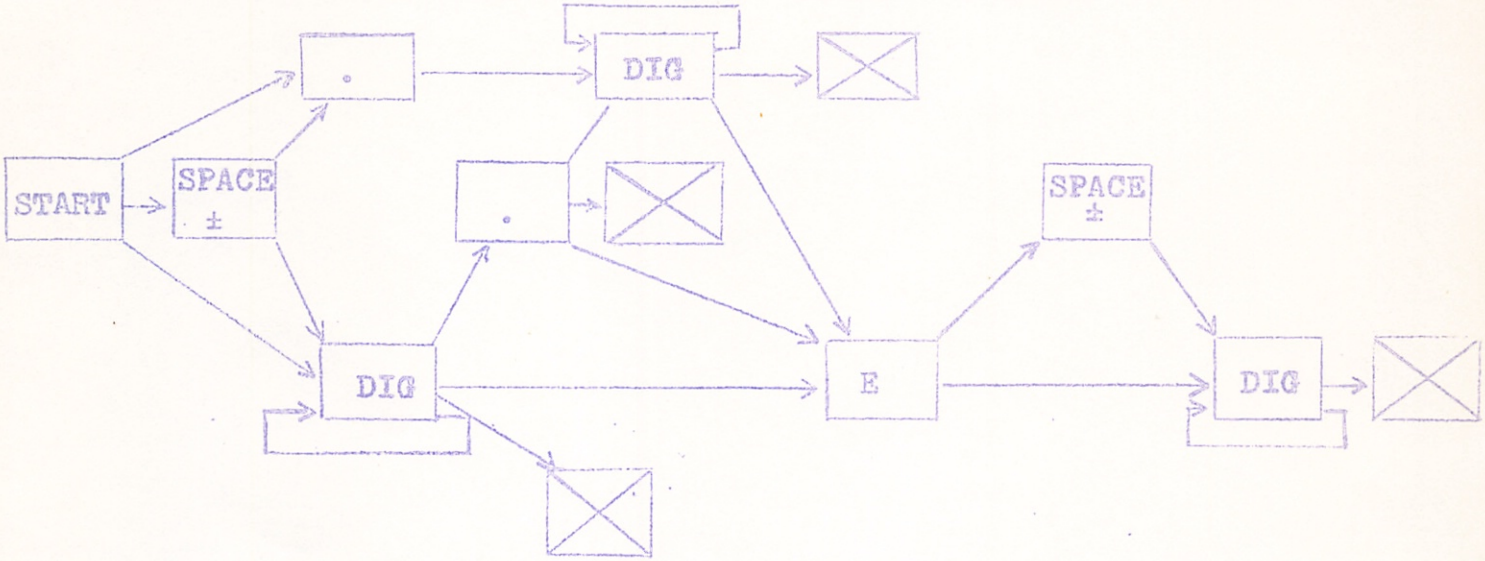
Floating Point Symbols and Macros (continued)

```
define      flac x
            IOT 132ØØ
            x
terminate

define      fdac x
            IOT 133ØØ
            x
terminate

define      float n
            IOT 134ØØ
            n
terminate

define      fix n
            IOT 136ØØ
            n
terminate

define      float2 n
            IOT 135ØØ
            n
terminate

define      fix2 n
            IOT 137ØØ
            n
terminate
```

Floating Point Symbols and Macros (continued)

```
define      cfloat a
            lfloat a
            rfloat a
terminate

define      pfloat a/b
b           remote [
b,          cfloat a
]
terminate
```

Diagram of Legal Floating Input Text Strings




A text string may be tested for legality by following its path, character by character, through the above diagram. From any given box, the possible legal choices of "next character" are indicated by the arrows leaving the box. If, at any time, the next character is not one of the allowable choices, flip looks no farther, and the string immediately becomes illegal. If, at any time, one of the boxes containing a big X is reached, the interpretation terminates and the foregoing string is taken as legal. A box with an X indicates "any character (including e.o.m.) other than those explicitly permitted by the other arrows leaving the previous box" and always is an allowable choice where shown.

The routine's action may also be predicted from the following: both at the very beginning and as soon as the first "e" is seen, a flag is cleared. Whenever a digit is seen, this flag is set. When a character is encountered which makes no sense, interpretation stops, and the flag is checked. The string is "legal" if the flag is set and "illegal" if it is clear at this point.

EXAMPLES

a. Suppose we wish to write a routine called qdr which evaluates the formula $ax^2+bx+c$, where a,b,c are stored in memory, x is in the floating accumulator and the result is to be left in the floating accumulator. Below is a subroutine to do this, called by a jda in order that it be transparent to the AC.

```
qdr,  ø
      dap qdx
      fdac qdt
      fmul a
      fadd b
      fmul qdt
      fadd c
      lac qdr
qdx,  jmp  .
qdt,  ø
      ø
```

b. Suppose we want to write a subroutine to be used in a syntax definition. (By a "jmp" pseudo-instruction.) This routine will accept a floating point number, storing its value in the output buffer. It will increment the Syntax Verifier's text pointer (fsa) past the floating point number. If there is no floating point number, it will "fail" and start Syntax Verifier's search for a "lac" pseudo-instruction.

Called by "jmp fpn" in a syntax definition.

```
fpn,    flip        /convert to floating point from
                    /internal
                    /return pointer to fsa
        stvl        /lose
        fdac i sug  /save number in output buffer
        idx sug     /index output pointer
        idx sug
        stvw        /return to syntax verifier
```

Note: It is desirable to return to stv even after no floating number was found. This is because there may be an "or" in the syntax definition, and the text string might pass under a different condition.