TX-0 COMPUTER
Massachusetts Institute of Technology
Cambridge 39, Massachusetts

M-5001-20

FLOAT - A FLOATING POINT INTERPRETIVE ROUTINE

16 May 1960

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Cambridge 39, Massachusetts

TX-0 COMPUTER

M-5001-20

FLOAT - A FLOATING POINT INTERPRETIVE ROUTINE

Contents Page

Note: Because of last-minute revision in the memo, page 18 is not included.

Paul Brady
Gordon Bell

FLOATING POINT INTERPRETIVE ROUTINE

A. Brief Description

This routine enables the programmer to perform arithmetic operations on the TX-0 computer. The computer itself is not designed for such operations; it is much better suited for problems involving work requiring a series of rapid decisions in logical operations. Its operation code is very cleverly designed to enable the coder to work not only with his numerical data but also with his instructions. In using FLOAT the programmer throws away these valuable features to an extent and slows down the computer by a factor of about 50.

FLOAT is an interpretive routine. This means that each order of the operator's program occupies one register, as at present, but the orders never enter the memory buffer register. The computer does not obey these orders at all, but rather the orders set forth by FLOAT, which in turn looks at the programmers orders to see what to do. If the programmer gives the order "fca x" (clear and add x), the interpreter will sense this and perform the actual process of putting the contents of x in the (pseudo) accumulator.

Such a method may seem at first to be terribly cumbersome and time-consuming. At present, this is in part true. FLOAT has been written for expediency and is reasonably wasteful of registers. During the summer of 1960 the program will be condensed so that it is faster and more economical. When some of the new computer orders are installed, the program will almost vanish (at least, we hope so!) and then the floating point interpreter will become a most valuable tool in solving arithmetic problems. In fact, the TX-0 will be in a position to compete very favorably with the 704 in speed of numerical computation. None of the future changes in FLOAT will at all affect programs which will be written to make use of the existing program, except that they will take progressively less and less time to run.

Programmers familiar with the 704 coding will practically be able to skim through the following memo. Those not so familiar will probably find the concept of indexing a little tricky, but once they "catch on" to the concept, we feel certain they will agree that such a device indeed provides a counting method superior to that of building counters within a program.

B. Floating Point Numbers

The general method of representing numbers in the TX-0 computer is to let each bit represent a power of 2, except for bit zero which is generally reserved for the sign bit. A binary point is always specified. If the binary point is at zero (all the way to the right) then the next bit left is 2^0, then 2^1, then 2^2, and so on. We thus represent an integer. By setting this point at 17, the first bit to the right is 2^{-1}, then 2^{-2}, and so on, specifying a fraction. In all cases, the entire number is contained in one register.

A floating point number is really two numbers; the first a fraction, and the second an exponent. Such practice is not new - we are all accustomed to seeing 389 written as $.389 \times 10^3$. Two registers are used to hold a floating point number; in the first is a fraction whose greatest magnitude is almost 1. The second register, the one immediately following in memory, contains the power of 2 (two) by which the fraction must be multiplied to make it read the correct number.

Every conceivable positive number except zero will contain a one somewhere in its binary code. The octal number 504 is written 101000100 and so has three one's. Taking advantage of this fact, it is our convention to build up normalized floating point numbers. The fractional part of a positive stored number will always have a one for its first bit, bit one, reserving bit zero for the sign bit. Zero will be represented by having all zeros in the fractional part of the number. Zero is the only normalized positive number that will not have a one for its first bit following the binary point.

You can make a number negative by complementing just the fractional part. If we are to preserve our convention of positive numbers having bit 1 a one, then all negative numbers will have their sign bit a one and bit 1 a zero. Minus zero is the only negative number which can be stored having bit one "lit."

The exponent, stored in the register following the fraction, obeys the rules commonly used for binary integers. It is not normalized in any way and is made negative by complementing the entire number. It is self-evident that to change the sign of a number you must complement only the fractional part - but you need not concern yourself with this detail in programming since there is a "change sign" order available to you.

A small but important detail to remember is that in calling for a number from register x, you are really calling for registers x and x+1. When you give the order fst x (store in x) you are really storing in x and x+1. And when you wish to add in the number following $C_{(x)}$ in memory, you must add in $C_{(x+2)}$. The notation $C_{(x)}$ means the contents of x.

Writing a floating point number is not hard and can be done with ease if you have a desk calculator. If you do not have such a device, pencil and paper works and for small numbers such as 1, .5, 6.25 and 4.75, it will be just as adequate. The principle is as follows:

$$35.602 = .35602 \times 10^{+2} \qquad .0005674 = .5674 \times 10^{-3}$$

Rule (for the above numbers): Move the decimal point until it just encloses the first significant figure (in other words, normalize the number). The power to assign to 10 is the number of places moved. If you moved left, the power is positive; right, negative.

Binary numbers obey exactly the same rules. Note these examples:

$$101101101. = .101101101, +9 \qquad .00010001 = .10001, -3$$

The writer assumes that the reader, having examined the two examples above, will be able to convert any binary number to floating point. The chief problem is to get the binary number, and this is where the desk calculator becomes indispensible. Here are several decimal numbers. Observe the method used for each.

$$268_{10} = 414_8 = 100001100_2 = .1000011, 9_{(10)} = 0.1000011, 11_{(8)} = 206000,11$$

+268, decimal, is stored in two successive registers as:

$$\left.\begin{matrix} 206000 \\ 11 \end{matrix}\right\} \text{octal.}$$

Notice that at one point in this operation a zero sign bit is arbitrarily set in from of the number. FLOAT is designed to interpret this as the sign bit and to consider everything to the right of it as the number itself.

Thus, 17 bits are available for number representation.

$$13{,}000{,}000_{(10)} = 61456500_{(8)} = 110001100101110101000000_{(2)}$$
$$= 0.110001100101110101000000,\ 24_{(10)}$$
$$= 306273,\ 30$$

Round off has occurred. The number is too large and "complicated" for exact representation - that is, 17 bits is not enough to contain it exactly. Note that the bit just following the cut-off point was a one. This "one" was carried into the number and thus the number reads 306273 instead of 306272.

If the reader works at enough of these he will begin to realize that you need not write these numbers in their binary form. To illustrate this, the above number:

$61456500_{(8)}$ = (divide by 2 to make it start with a zero)
= 30627240×2^1 = (each octal number represents 3 binary places except the first, which is 2 places)
= 306273 (round off), 3+3+3+3+3+3+3+2+1 (the 1 comes from our first division by 2) or 30_8.

This last step is a little involved, and anyone playing with these numbers for any length of time will come upon it himself, so if you find it just a little confusing right now, there is no need to worry.

Speaking of round off, this would be a good point to mention range and precision of floating point numbers. 17 bits will allow representation to about 5 or 6 significant figures with a range $2^{-131071}$ to $2^{+131071}$. $2^{+131071}$ is a large number and most of the computations will probably involve numbers much smaller.

To convert decimal numbers to octal numbers, simply divide the decimal number by the largest power of 8 which can be contained within it. Write down the integral number and then divide the remainder by the next largest power of 8, etc. For your convenience, a table of powers of 8 is included in the first appendix to this memo.

Fractions are no harder to convert. In fact, on the desk calculator the writer uses, they are easier. The principle is stated as follows:

Assume you have a fraction already stored in memory. It is not a floating point number, but is rather a number which has bit 1 as 2^{-1}, bit 2 as 2^{-2}, 3 as 2^{-3}, and so on. The number is guaranteed less than 1 and is positive for convenience sake.

Multiplying a binary number by 8 is the same as shifting it left three places. Assuming an accumulator that has these three places available for inspection, we can read off the first octal digit of the fraction after the first multiplication by 8. The second such multiplication will yield the second octal digit and the whole 17-bit number can thus be examined, 3 bits by 3 bits.

This octal fraction is meant to represent a decimal fraction. Hence, multiplying the octal fraction by 8 will move exactly the same information across the decimal point as multiplying the decimal fraction by 8. To convert a decimal fraction to octal, then, successively multiply by 8 --

```
+.07954325      x8     = 0.636346
     .636346    x8     = 5.090768
     .090768    x8     = 0.726144
     .726144    x8     = 5.809152
     .809152    x8     = 6. etc.
```

We have thus brought across the decimal point the following octal number: 05056. This could be carried out indefinitely, because this fraction does not have an exact representation. We may write this number in binary:

```
.05056 (really .05056362) = 0.000101000101110011110010  binary
                          = 0.101000101110011110010, -3 binary floating point
                          = 242717, -3                  octal floating point
```

It might be interesting to see just how precisely this number is represented in our system of 17 bits.

Summing powers of 2:

The original binary number written columnwise is as follows, taking only 17 bits from the first significant one:

Bit Position	Power of 2		
.0	-1	0	(1/2)
0	-2	0	(1/4)
0	-3	0	(1/8)
1	-4	.0625	(1/16)
0	-5	0	etc.
1	-6	.015625	
0	-7	0	
0	-8	0	
0	-9	0	
1	-10	.0009765625	
0	-11	0	
1	-12	.000244140625	
1	-13	.0001220703125	
1	-14	.00006103515625	
0	-15	0	
0	-16	0	
1	-17	.00000762939453125	
1	-18	.000003814697265625	
1	-19	.0000019073486328125	
1	-20	.00000095367431640625	
	Total:	.0795431137084964 (within the limits of my adding machine)	

Five significant figures have been successfully converted; the sixth leaves something to be desired; beyond the sixth is pure junk. A table of powers of 2 is also included in Appendix 1.

A mixed number, such as 576.046 is taken as the sum of an integer and a fraction and should afford no difficulty to the reader. Negative numbers are best handled by considering them as positive numbers and then complementing the fractional part of the fl. pt. number.

Converting the other way - from floating point to decimal - is not difficult and is done in a manner similar to the above described methods, except of course reversed. There are programs written that will print out such numbers in decimal form as a number with exponent. More will be said about this later.

C. The Nature of an Interpreter

FLOAT is a routine which must co-exist in memory with the user's program and data. It has a fixed location and is not available for conversion to other locations to suit the programmers' needs. At present, it occupies registers 30 to 1323 with its entry point at 54_8. This will be shortened as the summer progresses. PRINT II, an interpretive routine for printing out many kinds of numbers, is recommended for use with this program. It begins at 1400_8 and extends to about 2650_8. In any case, the two programs together do not occupy registers beyond 2700, leaving a good deal of memory for users' programs. There are many space-saving orders available with FLOAT that will enable the user to write very short programs to do rather complex operations.

Entry to FLOAT is gained by giving the orders

```
                    llr .
                    tra 54, where . is the present location
```

This is a defined macro instruction, "float". From then on, computer control is entirely within the interpretive routine and the operator's orders serve as directions to the interpreter rather than orders to the computer.

FLOAT defines a pseudo-accumulator (fac) and a pseudo-program counter (fpc). It defines fit=30, the starting address of the program, and flt=54, the entry point. There are eight index registers. These four flads are the only flads defined in the float definitions, but there are some 30 or so operations which are also defined. The user is safe if he avoids flads beginning with f (also p, if PRINT II is to be used).

The interpreter is naturally not as fast as the normal mode of the TX-0. The TX-0 time per instruction is 12 μsec except when an in-out order is involved. At this point we would prefer not to specify an exact time per order for the interpreter because each order takes vastly different numbers of FLOAT instructions. A safe estimate, and a conservative one, is an <u>average</u> of 50 machine order times, or .6 msec. With the new computer code we hope to reduce this to 20 or 25 orders. The arithmetic operations, unfortunately, do not take an average of 50 machine orders and if there is a large number of these in the operator's program, he may find that his problem requires more computer time than he at first thought it would.

D. Orders Available in FLOAT

This section is intended to be a manual of orders which can be used with this interpretive routine. Arithmetic orders handle floating point numbers stored in two successive locations, except where indicated: (C_x means contents of x).

a. Programmed arithmetic

fca x	clear, and add x (all of these, when referring to the accumulator, mean the pseudo-accumulator)
fad x	floating add Cx to the accumulator
fcs x	clear, and subtract Cx from accumulator
fsb x	floating subtract Cx from the accumulator
fst x	store the accumulator in x
fmp x	multiply the accumulator by C_x
fdv x	divide the accumulator by C_x*
fsz x	store zero in x. Accumulator unaffected.

b. Logical operations

fal x	add logically C_x (fractional part only) to the accumulator. A logical add is performed in the normal computer code by "add" or "pad+cry"
fsl x	store the logical word in x - that is, store only the fractional part of the accumulator in x

c. Transfer operations

ftp x	transfer on acc. plus (but not plus zero) to x
ftn x	transfer on negative acc. to x (but not minus zero)
ftz x	transfer on zero to x. Only the fractional part is examined to see if zero is present.
fnz x	transfer to x if acc. is not zero
ftr x	transfer unconditionally to x

d. Operative instructions (not addressable)

fsp	set acc. plus
fsn	set acc. negative
fch	change the sign of the accumulator
foz	operate zero - does nothing
fov	skip next instruction if no overflow is present. Always clears overflow indicator (more about this later)

* If divide by zero is attempted, the result will be the largest number possible of the same sign as the original number in the accumulator. The divide check test register will be set (dct).

fdc	skip next instruction if no divide check. Always clears divide check indicator. The divide check is lit if division by zero is attempted.
ftb n	test bit "n" of the tac. If off, skip the next instruction. It is senseless to let n be greater than 21_8.
fcv	take the fractional part of the accumulator as a fixed point number and convert it to floating point. Immediately preceeding this order must be a constant indicating the position of the binary point in the fixed point number. More about this later.
frc	do the reverse of the above order. The same holds true of the contents of the register preceeding this order.

e. The Halt orders

fhl	not addressable. Computer will stop, displaying program counter in real acc., clear live reg. On restart, will stop again with pseudo-accumulator in acc. and live reg. respectively. Restart again continues program.
fht x	Addressable. Same as above, except that live register on first stop contains location "x". When computer is restarted, control will go to x.
fhn n	Not addressable, but n can be as large as the number of registers in memory (17777_8). On first stop, "n" is displayed in live reg., all 1's in accumulator. Second stop and second restart same as fhl.
illegal order	If an illegal order is given (which is rather difficult to do), the flexo will type "fio" and "Float" will execute a fhl stop.

E. Indexing Operations with Float

1. The Index Register

To someone who has never before used an index register, this section of this memo may be a little confusing. This confusion can be minimized if the reader will read through once carefully, and then examine the examples given before re-reading the section.

Eight index registers have been set aside for use with this routine. They are located in registers 30_8 to 37_8. An index register does not contain a floating point number, but contains a number of which can be as big as the size of memory. It really can contain any number at all, but if it is to be used for indexing it must be positive and cannot exceed 17777_8.

This is the complete description of the index register. They are set up with the following commands:

flxx x,n	load index register n with the contents of x
flax x,n	store the contents of index register n in x (loadaddress)

As may have already become apparent, any macro instruction beginning with f and ending with xindicates an operation involving an index register. These macro instructions will be defined later on.

An "fio" alarm will result if any indexed order is given where n is not from 0 to 7, inclusive.

2. Indexed Operations

An indexed operation is an addressable operation whose address has been modified by the contents of an index register. All addressable operations are indexable except those which inherently depend on index registers. All of the addressable orders described under "Orders Available in FLOAT" are indexable.

fad x	means	"take the contents of x and add it to whatever is in the accumulator."
fadx x,n	means	"Take the contents of the register des-described by (x minus contents of n) and add to the accumulator."

A numerical example might help:

If index register 2 contained 6_8,	
fadx 4552,2 means	add contents of (4552-6) or 4544 to the accumulator

The reason for this type of operation may seem a little obscure at this point, but in a few moments it may become a little more clear.....

In a similar vein, fstx x,n; ftrx x,n; fhtx x,n; and fslx x,n are all legal macro-instructions.

3. The Index Decrement

The index decrement is a number, usually positive, not exceeding 17777_8 and occurring as part of the program. The exact nature of how the index decrement is stored in the program may be found by examining the list of macro-definitions occurring towards the end of this memo. The contents of the index register may be changed by giving an operation involving a decrement.

There are only three orders involving a decrement. The first is quite simple:

ftdx x,n,d	Transfer control to x if the contents of index register n exceed the value "d". If not, then ignore the order and proceed to the next one.

In the past example, with index register 2 containing 6, ftdx 6000,2,2, would transfer because index register 2 is greater in value than 2, the decrement. The command ftdx 6000,2,6, or ftdx 6000,2,4673 would be ignored. The second command is almost as simple:

ftxx x,n,d	Transfer control to x <u>always</u>. When doing so, increase the contents of index register n by the value "d".

If index register 2 contains 6_8, the command ftxx 6000,2,4 would send control of FLOAT to 6000, and in the process, index register 2 would become 6+4 = 12/8.

Remember that index register 0 is an index register, and unlike the 704, if you here call for index reg. 0 you will perform an indexed operation. The third order using a decrement is by far the most useful:

ftix x,n,d	Transfer control to x only if the contents of i.r. n <u>exceed</u> (not equal) the value "d". These are <u>the same</u> transfer restrictions as in ftdx. Ignore the order if C (i.r.n.) equals or is less than the value d. <u>If the transfer is performed</u>, subtract the value "d" from the contents of i.r.n.

An example of a program follows:

Suppose you have a series of floating point numbers stored in 4000_8 on. You wish to multiply them by a constant contained at 5000_8. There are 50_8 such numbers. This means that they occupy 120_8 registers.

```
aaa,        flxx 1,(120
aab,        fcax 4000+120,1
aac,        fmp 5000
aad,        fstx 4000+120,1
aae,        ftix aab,1,2
```

The answers will be found from 4000 on. Notice that the first number multiplied will be the one stored at 4000, the next, 4002, and so on. This operation will be performed 50_8 times, because the last time through (the fiftieth time) the i.r. 1 will contain "2" and the aae order will be ignored.

Another example: Say that you have numbers stored from 4000 on. There are only three numbers. You wish to put their maximum value in 5000: (assume they are all positive)

```
aaa,        flxx 1, (6
aab,        fsz 5000
aac,        fcs 5000
aad,        fadx 4000+6,1
aae,        ftn aah
aaf,        fca 4000+6,1
aag,        fst 5000
aah,        ftix aac,1,2
```

Finally, an example of double indexing. Assume you have 6_8 numbers from 4000 on, and you wish to convert them to binary numbers and store them from 5000 on:

```
aaa,        flxx 1,(14            14₈ is twice 6₈
            flxx 2,(6
aab,        fcax 4000+14,1
            fltofx 0              This order, to be described later,
                                  means "convert float to fix, with
                                  binary point at zero (extreme right
                                  end of the accumulator)
aac,        fslx 5000+6,2
aad,        ftix aae,1,2
aae,        ftix aab,2,1          This completes the loop. When
                                  Finished, the order following aae
                                  will be executed.
```

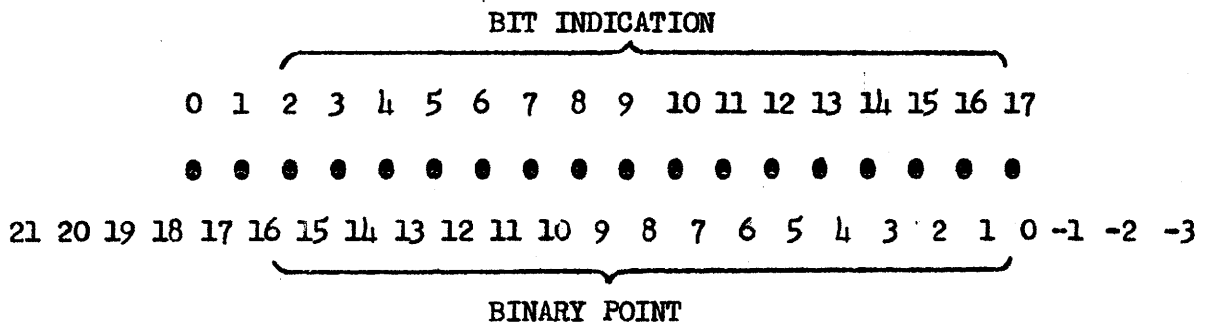
The general idea behind indexing is that if you want an operation done n times, then you load the index register with a value equal to n divided by the decrement which is to appear in the ftix instruction, that is, (n/d). This is not a rigid formula, You could load an index register with, say, 3 and the operation would be performed twice if the decrement were 2. Sometimes this is helpful in certain manipulations, but there is no point in covering the multitude of special cases which could come up.

F. Binary Point Convention

This section might well have been titled, "Unfortunate but True." There are two conflicting binary numbering conventions in operation here and they deserve at least a short explanation:

A bit is indicated by a number from 0 to 17. This is usually a decimal number, because very few octal instructions refer to specific bits. We have thoughtfully included such an order (ftb n) and the programmer should remember that if he specifies a bit number, it must be in octal.

The term "binary point" is used when referring to a normal binary number. It specifies the location of the point where to the right are bits indicating 2 to negative powers, and to the left are bits for 2 to positive powers. The convention applies only to conventional binary numbers and not to floating point numbers. The following diagram shows how these two conventions apply to a word in memory:



G. Conversion Orders

Despite attempts to show the ease of converting fixed point numbers to floating point, there are some times when one would not appreciate having to sit down with a desk calculator and make these conversions. If you do not believe this, try $.34590 \times 10^{587}$. There are features available which enable a programmer to surmount these difficulties:

1. Binary Numbers to Floating Point

If a binary number is contained in the fractional part of the accumulator (the order fca x will put any kind of number in fac) the order "fxtofl n", where n is the position of the binary point, will cause the number to be converted to a floating number. This is a macro instruction occupying two registers and uses the fcv order defined in Appendix 2. If n is not specified, it will be assumed zero. The newly formed floating point number will be in the accumulator.

The order "fltofx n" does the reverse of the above and changes the floating point number in the accumulator to binary. It is very possible that the number will be larger than the fixed accumulator can handle. If the floating point number is 568,000/and the binary point is specified at zero, then it will overflow the accumulator upon conversion. If this happens, the overflow indicator will be set and the resultant in the accumulator will be the correct answer with the overflow bits lost. If there is no overflow, the overflow indicator is <u>not affected</u>. That is, if it was set by some previous operation it will remain set regardless of the conditions of this conversion. The fltofx order uses frc (see Appendix 2) and is the only order which will set the overflow.

The overflow indicator is checked by giving the command "fov". This is non-addressable. If the overflow light was on, the next instruction in sequence will be obeyed. If not, then the next instruction will be skipped. In either case, the overflow light will be "turned off." It is advisable to give this order as a dummy instruction before using fltofx if you want to make sure that any overflow occurring will come from the fltofx order and not from a previous condition (similar to pen on the scope).

2. <u>Decimal Numbers and Floating Point</u>

Memo 5001-18 describes PRINT I, a utility tape for printing out numerical data from the TX-0. The reader is urged to read this memo (copies obtainable in the computer room). Print II, an expanded version of Print I, will be available when this memo on "Float" is printed. Print II will be exactly like Print I except that it will contain another order, pfp (print floating point).

Briefly, Print II is an interpretive routine which automatically sets up a format and prints out banks of data, flexo characters, or single numbers from the computer memory. It is entered by typing "print", a two order macro instruction defined

```
llr .
tra pit
```

Print II is available as a binary tape starting in register 1400. You must define pit = 1400 in your flexo tape using floating point orders and you must read in the proper Print program. <u>If you plan to print out floating point numbers, you must also define pfp=460000.</u>

Assuming that the reader has read the memo on Print I, he must be sure that the order pcm 4 (or less than 4) is given before the pfp order because there is a maximum of 4 columns available for printing out of floating point numbers. Also remember that each number requires 2 storage locations so that if he wants 5 numbers printed out he will have to specify that the location of the fifth number is (location of 1)+(8). Illustration:

```
pcm 4
pfp 4000
to 4004
```

The results from the on-line flexo are:

```
4000|         -.347825 5            .5000000 12        .246300-1
```

In the above example, the 5,12, and -1 will be printed out in a different color of the fractions. These three exponents are to the base 10. The accuracy is not quite so good as the number representation - thus, 1.000000, which can be represented quite exactly as a floating point number may be printed: .999986-0.

If the reader frowns upon this as being too coarse, the three programmers who have worked on PRINT II will take great delight in having the reader submit his own routine which prints these numbers.

To enter PRINT II, just type "print" on the off-line tape. It is not necessary to give an order that will leave FLOAT. Similarly, when finished giving print orders, just type "float" and control will automatically be transferred to FLOAT:

```
fca x
fst y
print
pcm 4
pfp z
to zz
float
fda q, etc.
```

There is available a subroutine which will convert decimal numbers represented in floating point fashion to real floating point numbers. Hence, the number $\left\{\begin{matrix} 48632 \\ +3 \end{matrix}\right\}$ stored as integers in the floating accumulator will be converted to a floating point number, 486.32.

The calling sequence will be

fts cvt

with the number to be converted in the accumulator. This will be performed by a separate subroutine, not a part of FLOAT and as such will have to be read in separately as with any conventional subroutine.

H. Conversion of Programs Prepared for FLOAT

1. FLOAT Itself

Location of stored parameters: (FLOAT is fixed in memory)

30 - 37 (all octal)	Index Registers
40	Program Counter (as add instruction)
41 and 42	Pseudo Accumulator
43	Overflow Indicator
44	Divide Check Indicator
54	Entry Point of Program
60 (flt+4)	Return Point from Subroutines

FLOAT is available in binary form only. The above locations will always remain the same despite revisions in the general program so that programs written for FLOAT will not become obsolete. Later on, the binary location of PRINT II may be changed, but the original version will still remain available. Those desiring to use FLOAT will find a box of English definitions available in the computer room. The contents of this tape are reproduced in Appendix 2. If one wishes to use floating operations, he should read in this tape before reading in his own tape during conversion time. If he does this, he is free to use any of the orders previously described in this memo.

Entry to FLOAT is obtained as such:

```
                    xxx
                    xxx
                    xxx (xxx refers to any normal computer
                    xxx orders such as cla, llr, add, etc.)
                    xxx
        float
                    fxx
                    fxx
                    fxx
                    fxx
                   "fxt x" leaves the floating mode and transfers
                           control to register x. If the next
                           order is to be taken, use fxt .+1.
```

The English tape, remember, defines a large number of symbols and macro definitions, all beginning with f. PRINT I English tape also defines many symbols beginning with p.

Convert tapes in the following sequence on Pass 1 and 2:

> A short title tape of your program.
> "Float Define"
> "Print I Define" if desired. Remember to define on your program tape
>
> pit = 1400
> pfp=460000
>
> Read in and convert your program.

In operation:

> Read in FLOAT, PRINT II and any other converted subroutines you wish.
> Read in your program.
> Transfer control to YOUR program, NOT to float.

2. Subroutines

It is expected that there soon will be a good number of subroutines available for floating point operations. The most common of these will be square root, sine and cosine, matrix multiply, tangent, etc. They are all entered by the command:

```
fts ---
```

Example:

```
fts srt (square root)
error return
normal return
```

Enter with the argument in the accumulator (pseudo-accumulator) and give the transfer command. The error return will occur if the argument was negative, and the normal return will occur if the argument was acceptable; the answer will be in the accumulator. Some routines may not have error returns:

```
fts sin
normal return
```

If you wish to use any of these subroutines, examine their write-ups (available in the TX-0 subroutine file) and convert them with your program. They may be located anywhere in memory except, of course, where FLOAT is stored.

The following page is intended as a sample write-up:

(This subroutine may or may not exist as such - it is merely included as a sample. Consult the subroutine file to see if there is in fact a scope plotting routine.)

SUBROUTINE: SCOPE PLOT

1. FUNCTION

Plots a point on the scope given coordinates in floating point. x coordinate in location z, y coordinate in accumulator.

1A. CALLING SEQUENCE

fts scp	y coordinate in floating accumulator
z	x coordinate location in z
OCTAL	total range, x direction, power of 2
OCTAL	total y range expressed as power of 2
OCTAL	location of x=0, y=0 expressed in TX-0 scope coordinates
pen or opr	pen for pen command, opr if no pen
error return	x exceeded specified range
error return	y exceeded specified range
normal return	

2. SPACE

Occupies 30_8 registers, uses 4 constants, and 10_8 registers of common storage specified tt, tt+1, etc.

3. TIME

Approximately .02 seconds. Varies greatly with values involved.

4. METHOD

Coordinate is divided by range and then converted to binary number using fltofx orders in FLOAT.

5. COMMENTS

This subroutine requires FLOAT in memory for its operation.

Arthur Wellesley
April 9, 1815

J. Writing Subroutines for FLOAT

Even if you do not plan to write any subroutines, you should skim through this section so you will understand what happens when a subroutine is called for.

The command "fts x" is identical to "fxt x". It was given a different coding to make it easier to distinguish between the two when examining a program. Your subroutine, then, will be entered in real computer mode.

If you plan to use FLOAT in your subroutine you should do the following:

1. Save any index registers you plan to use. This is done by "llr flt+n, slr y", where y is the location of the storage register. Before returning tothe user's program you must restore these registers.
2. Save the program counter. (fpc) This is accomplished in the same way. This must also be restored later.

You are now free to type "float". You need not worry about defining "float", as the user will already have in macro the English tape of definitions. When you are ready to return to the user's program, leave FLOAT with fxt .+1 or fxt to somewhere in your program. After restoring the necessary registers, give the order "tra flt+4". This will cause the computer to resume operation in "Float", taking as the next instruction the one succeeding the programmer's "fts" order. If you wish to skip one instruction (as in the normal return with square root), increment the program counter (fpc) by 1 before transferring to flt+4. The program counter will contain as an add instruction the location of the register which contained the fts instruction when your subroutine is entered. Your subroutine will be entered witha cleared real accumulator.

If you do not use FLOAT in your subroutine, you of course do not need to save any registers. When finished, give the command""tra flt+4" and the next order will be followed.

K. Step by Step Operation with FLOAT

The computer console has two switches, "Stop C_1" and "Stop C_2". If both these switches are turned on (along with "Suppress Chime" for the benefit of the nerves of all those present) the computer will stop twice on each instruction. The first time, you can read the memory buffer register to see what order is going to be obeyed, and pushing restart causes the order to be carried out. Pushing restart again brings the next order to memory buffer, etc.

FLOAT has such a provision. Bit 2 of tbr has been reserved for such a purpose. If bit 2 is on (raised), the computer will stop twice on each operation of FLOAT.

The first time with the instruction in the live register, and the program counter (as an "add" order) in the accumulator; the second time with the fractional accumulator in the real accumulator and the exponent in the live register.

The very first stop using this system will cause the accumulator to be displayed. From then on the sequence is "order, accumulator, order, accumulator, etc." If you wish to have the program continue in its normal manner, simply turn off bit 2 of tbr and push restart.

L. A Special Multiply Order

If you ever wish to multiply anything by an exact power of 2 (say 64) you may do so by giving the order:

fmp2to n , where n is the power of two. In the above example, n is 6 because 2^6 is 64. n may be any size, plus or minus. It is not restricted to be within 17777_8.

This is a macro instruction involving six orders, but they are not floating point orders and they will take far less time to execute than the one order, fmp x, where x contains the number in question. This order is worthwhile if speed is a consideration.

M. The Macro Instructions

One of the big advantages of the TX-0 computer is that you may debug a program using machine time and utility programs. For this reason, it would profit the programmertto understand the macro instructions ftix, fadx, etc., so he could alter them if he wished to using a utility tape.

Indexing is accomplished with the order "fir n". If "fir n" preceeds an indexable instruction (and it does not have to be immediately preceeding it) then the indexable order is modified by the contents of i.r.n.

```
fca x    is not modified while

fir 2
fca x    is exactly equivalent to fcax x,2
```

All of these orders are defined at the end of this memo.

The decrement is a do-nothing order by itself - it is defined as "fid d" and d may be plus or minus.

When FLOAT runs across such an order it ignores it completely except that if a ftx, ftd, or fti order is given two instructions later it then examines the fid order to see what the decrement was.

```
fid d
fca x
fmp y
fst z  makes no use at all of fid, but

fid d
fir n
fti x  uses the fid order to determine the decrement to be applied
```

to index register n. The order "ftix x,n,d" is in fact defined as such. Notice that the fir order here does not modify the address of x, but simply tells the fti order which index register to look at when comparing the contents of the decrement to the index. It would be ridiculous to make fti an indexable order - a few moments thought on this point should convince the reader that this is indeed true.

The programmer is urged to use the macro orders in writing a program using indexed operations. He will find them quite similar to the orders employed by the 704 computer - but he must realize that "fcax x,n" is really a two instruction operation and if he wishes to write such an order using UT-3 or another utility tape he must remember to write it

```
fir n
fca x
```

The only macro-instructions not adequately described by the above paragraphs are fxtofl, fltofx, and fmp2to n. By reading the macro-definitions at the end of this memo the reader should be able to understand how these orders are set up.

Note that ftix .-5,1,2 is not equivalent to fid 2, fir 1, fti .-5 because in parameters of a macro instruction the (.) is defined as the address of the first (fid) order. ftix .-5,1,2 is equivalent to fid 2,fir 1, fti .-7.

Appendix I

Numerical Constants

a. Powers of 2

2^n	n	2^{-n}
1	0	1.0
2	1	0.5
4	2	0.25
8	3	0.125
16	4	0.0625
32	5	0.03125
64	6	0.015625
128	7	0.0078125
256	8	0.00390625
512	9	0.001953125
1024	10	0.0009765625
2048	11	0.00048828125
4096	12	0.000244140625
8192	13	0.0001220703125
16384	14	0.00006103515625
32768	15	0.000030517578125
65536	16	0.0000152587890625
131072	17	0.00000762939453125
262144	18	0.000003814697265625

b. Powers of 8

n	8^n
0	1
1	8
2	64
3	512
4	4096
5	32,768
6	262,144
7	2,097,152
8	16,777,216
9	134,217,728
10	1,073,741,824
11	8,589,934,592
12	68,719,476,736
13	549,755,813,388
14	4,398,046,507,104
15	35,184,372,056,832
16	281,474,976,454,656
17	2,251,799,811,637,248
18	18,014,398,493,097,984
19	144,115,187,944,783,872

Appendix 2

Orders in FLOAT

The following is a summary of the definitions appearing in the "Float Definitions" Flexo tape.

```
fit=30
fpc=fit+10
fac=fpc+1
flt=fac+13

define
        float
        llr .
        tra flt
        terminate
```

fid=0	index decrement, y
ftp=020000	transfer on plus to x, but not plus zero
ftn=040000	transfer on negative to x, but not minus zero
ftz=060000	transfer on zero to x
fnz=100000	transfer on non-zero to x
fca=120000	clear, and add x
fad=140000	(floating) add x
fcs=160000	clear, and subtract x
fsb=200000	subtract x
fst=220000	store accumulator in x
fmp=240000	multiply by x
fdv=260000	divide by x
fxt=300000	leave floating mode, go to x
fts=320000	transfer to subroutine at x
fht=340000	halt, then on second restart transfer floating control to x
ftx=360000	transfer with index register n inc. by y. Requires fir,fid
flx=400000	loads index register n with C$_{(x)}$. Requires fir
fti=420000	If i.r.n. grtr. than y, goes to x,(n-y) to n. If not, ignore.
fla=440000	load address x with C (index register n). Fir required.
fir=560000	used to specify index register. n must be $0 \geq n \geq 7$
ftr=600000	unconditional transfer to x
fsz=460000	store zero in x. Accumulator unaffected.
ftd=500000	trns only if index exceeds decrement. C(index reg) not changed.
fal=520000	add logical contents of x to fraction of ps. accumulator
fsl=540000	store only the fractional accumulator in x. x+1 not changed
ftb=620000	test bit n of tac. If off, skip next instruction
fhn=640000	halt, display n in live reg., all 1's in acc, See Appendix 3
fcv=660000	convert fix to float. Used after fid
frc=700000	float to fix, use after fid

Use of an undefined instruction, such as 720000, causes fio alarm

fhl=740000	halt, then proceed. Not addressable.
fsp=740001	set plus. Not addressable.
fsn=740002	set negative. Not addressable.
fch=740003	change sign. Not addressable.
foz=740004	operate zero - does nothing
fov=740005	skip next instr. if no ovrflw. Always clears indicator.
fdc=740006	same as fov, except for divide by zero check. Not addressable.

Other macro definitions (besides float)

```
define
       falx X,N
       fir N
       fal X
       terminate
```

Similarly, these orders are defined:

```
fslx X,N       flax X,N
fcax X,N       fszx X,N
fadx X,N       flxx X,N
fcsx X,N
fsbx X,N
fstx X,N
fmpx X,N
fdvx X,N
```

```
define
       ftxx X,N,D
       fid D
       fir N
       ftx X
       terminate
```

similarly,

```
ftix X,N,D
ftdx X,N,D
```

```
define
       fxtofl N
       fid N
       fcv
       terminate

define
       fltofx N
       fid N
       frc
       terminate

define
       fmp2to N
       fxt .+1
       add fac+1
       add (N
       sto fac+1
       float
       terminate
```

Start 54

Appendix 3

Halt Orders in FLOAT

Each of these orders stops the computer twice. The first stop displays certain information in the live register and the real accumulator, the second stop displays the pseudo-accumulator. Pushing restart after the second stop continues the program.

Order	First Stop	Second Stop
fhl, halt and proceed		
live reg:	clear	exponent of accumulator
(real) accum:	program counter	fraction of accumulator
fht x, halt and transfer		
live reg:	location of next inst.	exponent ---------
(real) accum:	program counter	fraction ---------
fhn n		
live reg:	halt number, "n"	exponent ----------
(real) accum:	all 1's	fraction ----------

illegal order
flexo types "fio" and then an fht order is carried out

stop on each instruction - occurs if bit 2 of test buffer reg. is on

live reg:	instruction to be executed	exponent------
(real) accum:	program counter	fraction------