gri-909

floating point manual

GRI Computer Corporation
320 NEEDHAM STREET, NEWTON, MASSACHUSETTS 02164

Price $1.75

GRI-909

FLOATING POINT MANUAL

GRI Computer Corporation, 320 Needham Street, Newton, Massachusetts 02164
ĉ October 1970 by GRI Computer Corporation

71-44-001-A
1170-500

TABLE OF CONTENTS

1 FLOATING POINT INTERPRETER

1.1 Introduction 1-1

1.2 Basic Package, $SFI 1-4

1.3 Floating Point Format 1-6

1.4 Internal Registers 1-7

2 BASIC COMMANDS

2.1 Command Categories 2-1

2.2 Command Descriptions 2-2

2.2.1 Type I Commands - Load & Store Commands 2-2

2.2.2 Type II Commands - Binary Commands 2-3

2.2.3 Type III Commands - Unary Commands 2-7

2.2.4 Type IV Commands - Index Commands 2-8

2.2.5 Type V Commands - Conditionals 2-8

2.2.6 Type VI Command - Exit 2-11

3 DATA CONVERSION

3.1 Introduction 3-1

3.2 Floating Point to Character Conversion 3-2

3.3 Character to Floating Point. 3-5

3.4 Common Tables & Routines 3-7

3.5 Character Set Table 3-8

3.6 Floating Point Powers of Ten Table 3-9

3.7 Left Shift FAC 3-10

3.8 Multiply FAC by Ten 3-11

4 EXTENDED COMMANDS

4.1 Introduction 4-1

4.2 Sine, Cosine 4-3

4.3 Arc Tangent 4-6

4.4 Natural Log 4-7

4.5 Exponential 4-8

4.6 Square Root 4-9

5 NON-INTERPRETIVE MODE USAGE

5.1 Introduction 5-1

5.2 Subroutines 5-2

5.2.1 Double Precision Fixed Point Add 5-2

5.2.2 Double Precision Fixed Point Multiply 5-3

5.2.3 Double Precision Fixed Point Divide 5-4

5.2.4 Single Precision Divide 5-5

5.2.5 Floating Point Normalize 5-7

5.2.6 Negation and Store 5-9

5.2.7 Generate Zero or Largest Number 5-10

5.2.8 Floating Arithmetic Right Shift 5-11

5.2.9 Other Notes on Non-Interpretive Usage 5-12

5.3 User Generated Extended Functions 5-12

6 OPERATING INSTRUCTIONS AND SYSTEM GENERATION

6.1 Using the Package as Supplied 6-1

6.2 User Generated Systems 6-4

6.2.1 General 6-4

6.3 Assembling $SFI 6-4

6.4 Generating a Floating Point System 6-5

APPENDICES

Appendix A Command Summary - Basic A-1

Appendix B Commented Command Equate Tape B-1

Appendix C Source Tape Organization C-1

Appendix D %FCG - Floating Point Constant Generator D-1

Appendix E FPSET - Error Trap Routine E-1

Appendix F Trace Routine F-1

CHAPTER ONE

FLOATING POINT INTERPRETER

1.1 Introduction:

The GRI Floating Point Intrepreter is a complete system that allows the user to process data in floating point arithmetic. Floating point arithmetic, through the use of multiple precision arithmetic and an exponential concept greatly extends the range of precision available to the user beyond that of fixed point arithmetic. It also, through utility routines, frees the user of the bookkeeping involved with scaling and unscaling of numbers that is necessary in a fixed point system.

The GRI-909 has an instruction set which is known as machine language. The computer reads instruction words out of its memory and hardware is activated by the interpretation of each instruction word to cause the execution of that instruction. An interpretive software system fetches instructions which we shall call commands from the computer's memory and causes various subroutines to be entered as a result of the interpretation of the command. These commands fetched by the interpreter are also called psuedo-instructions because their format deviates from the machine's instruction format. The standard machine format instruction is

WORD 1 SDA MOD DDA

WORD 2 [ADDRESS] (if a memory reference instruction)

A pseudo-instruction or command such as the ones used in the GRI Floating Point Interpreter looks like this:

WORD 1 OP CODE

WORD 2 [ADDRESS] (if a memory reference pseudo-instruction)

The interpreter actually simulates the process used by the computer's hardware to execute an instruction. The interpreter fetches the OP CODE words

and addresses, sets up arguments, flags, and performs a function on the argument(s) as specified by the OP CODE of the pseudo-instruction.

An interpretive approach to floating point arithmetic provides the user with a functionally oriented language that makes usage of floating point arithmetic much easier than if it were done through a series of subroutines called in machine language. The user references floating point numbers with a single address which is the first address of the two word floating point number. The interpreter takes care of the address bookkeeping necessary for two word argument handling. The interpreter also maintains a set of accumulators much the same as an arithmetic unit. Arguments and results are manipulated and left in these accumulators. The interpreter utilizes two such accumulators plus an index register.

There are a set of commands in the interpretive system that are not floating point arithmetic commands. These are program control commands such as conditional jumps and index register manipulators. The index is simply used to keep track of the number of times command loops are executed. These commands, although they could be effected by use of basic machine language, are also provided in the interpretive mode because they can save the user time that would be spent entering and leaving the interpretive mode, and almost always save space in terms of the coding needed.

When the user is ready to execute commands in his program, he first issues a machine language command that causes a jump to the interpreter to take place. The interpreter now assumes control and starts fetching commands which follow the jump that caused interpretive mode entry. If the user wishes to begin executing machine language instructions, he must issue an interpretive command that causes the interpreter to relinquish control. In essence, the

machine is running in two different modes; a machine language mode and a psuedo-language mode -- in this case, a floating point language.

The GRI interpretive system offers a novel error trap feature which may be invoked by the user to assist in tracking down places in the program where data values are causing error checks to occur. Errors such as dividing by 0, exceeding the capacity of the psuedo-accumulators in either the mantissa or exponent portions, etc., can all be caused by an unknown data base. All manipulations of data refer to manipulations in and out of the psuedo-accumulator called FAC. This accumulator behaves like the accumulator in an adding machine. It must be loaded to initialize it, stored to save it, and all arithmetic operations leave their results in the accumulator. Commands with two operands are called binary commands and operate on a data word in user memory and the contents of FAC, replacing the result in FAC. Commands with one operand are called unary commands and operate on FAC, leaving their results in FAC. Let us consider a simple example:

Compute R = $\sqrt{X^2 + Y^2}$

```
JU $SFI      ;enter floating mode
FLDA X       ;fetch X to FAC
FMPY X       ;X^2 in FAC
FSTA T1      ;store FAC in temporary loc
FLDA Y       ;fetch Y to FAC
FMPY Y       ;Y^2 in FAC
FADD T1      ;X^2 + Y^2 in FAC
FSQT         ; sqrt(X^2 + Y^2) in FAC
FSTA R       ;store result in R
FEXT         ;exit from floating mode
```

1.2 Basic Package, $SFI:

Floating point arithmetic capabilities are provided through an interpretive package. Associated with the package is an external↔internal format data conversion routine that can be easily tailored to the character set being processed.

The interpretive package is invoked by a normal subroutine call. The call is followed by a string of commands that are established by use of equate statements during the assembly. The last command in the sequence causes a return to the calling program. Operations are performed using a pseudo accumulator maintained locally by the interpretive package. The package also contains a 16 bit pseudo index to allow loops within the command sequence. Without this feature, it would be necessary to exit and re-enter the interpretive package and perform loop counts outside the interpreter. Although the latter procedure is, in most instances, faster in terms of time taken to do the loop, it usually involves considerably more code and, therefore, takes more space.

As an example of a typical problem programmed in the interpreter language, we evaluate the polynomial

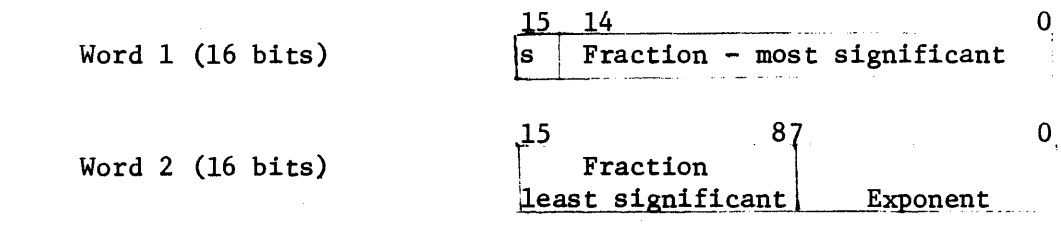
$$Y = A_0 + A_1X + A_2X^2 + A_3X^3 + A_4X^4$$

which can iteravely be expressed as $Y = (((A_4X + A_3)X + A_2)X + A_1)X + A_0$ as follows:

```
        JU      $SFI            ;ENTER INTERPRETER
        FLDX    M4              ;LOAD PSEUDO INDEX WITH -4
        FLDA    A4              ;LOAD PSEUDO-ACCUMULATOR
LOOP:   FMPY    X               ;MULTIPLY IT BY X
        FADDD   CONST           ;DEFERRED ADD A3 (THEN A2, A1, A0)
        FJIX    LOOP            ;COUNT THE LOOP
        FSTA    Y               ;STORE RESULT IN Y
        FEXT                    ;EXIT THE INTERPRETER
        .
        .
        .
   Y:   WRD     0,0             ;STORAGE SPACE FOR ANSWER
CONST:  WRD     A3-1            ;DEFERRED ADDRESS (GETS CHANGED)
   X:   WRD     X1,X2           ;TWO WORD FLOATING POINT VALUE OF X
  A4:   WRD     A41,A42         ;FLOATING A4 VALUE
  A3:   WRD     A31,A32         ;A3
        WRD     A21,A22         ;A2
        WRD     A11,A12         ;A1
        WRD     A01,A02         ;A0
  M4:   WRD     -4              ;ONE WORD INDEX COUNT VALUE
```

1.3 Floating Point Format:

Internal representation of a floating point number occupies two successive locations in memory and consists of a fixed point fraction (mantissa) with an associated exponent. The mantissa is in two's complement notation with a sign bit followed by 23 bits of significance. The binary point is assumed to be immediately to the right of the sign. The exponent, which is the power of two by which the mantissa is multiplied, has the range -200_8 to $+177_8$ (2^{-128} to 2^{+127}). This exponent is represented in "excess 200_8" notation by adding $+200_8$ to the true exponent. This requires a total of 8 bits and the range of the excess 200_8 notation is 000 to 377_8, where 200_8 represents 2^0. Thus, a floating point number looks like:



This format allows an accuracy of 6+ decimal digits and a range of $\pm 1.469368 \times 10^{-39}$ to $\pm 1.701411 \times 10^{+38}$.

To obtain correct results, all floating point operations (except FLDA, FSTA and FNOR) require the floating point numbers being operated on to be normalized; that is, bit 14 of word 1 must be the most significant bit of the fraction (mantissa). The only exception to this requirement is a floating point zero, which has no significant bits--a normalized floating point zero is two words of all zero (mantissa = 0, excess 200_8 exponent = 0).

Note: The mantissa of a normalized floating point number other than zero has an absolute value in the range $1/2 \leq |\text{mantissa}| < 1$.

Examples:

Decimal	Internal Floating Point (octal)	
	word 1	word 2
1.0	040000	000201
1.25	050000	000201
-1.0	140000	000201
-1.25	130000	000201
100.	062000	000207
-100.	116000	000207
0.5	040000	000200
0.25	040000	000177
π	062207	166602
$\pi/2$	062207	166601
$-\pi$	115570	011602

1.4 Internal Registers:

There are three pseudo-registers contained in the interpreter i) the pseudo-accumulator (FAC), ii) a temporary pseudo-accumulator (FTM), and iii) the pseudo-index register (FINDX).

i) FAC - The floating pseudo-accumulator. This consists of three locations in the interpreter and is used to contain the left-hand argument of a binary floating point command as well as the results of any floating point command. It is organized as follows:

FACHI - contains high order mantissa and sign of value in FAC

FACLO - contains low order mantissa of value in FAC

FACXP - contains excess 200_8 exponent of value in FAC

ii) FTM - temporary pseudo-accumulator. This consists of three locations analogous to FAC. They are named FTMHI, FTMLO, and FTMXP. The temporary accumulator is used to hold an additional floating point value for those commands which require two floating point values in order to operate, e.g. a type II (binary) command (see 2.2.2).

iii) FINDX - pseudo index. This consists of one location of the same name and holds the current value of the index.

Note: FACHI, FACLO and FTMHI, FTMLO are treated as full 31 bit double precision quantities for the basic arithmetic operations add, subtract, multiply, and divide.

CHAPTER TWO

BASIC COMMANDS

2.1 Command Categories:

The commands are of the following categories:

I	load & store	; the command specifies the source or destination of floating point data - the corresponding destination or source is the pseudo accumulator.
II	binary commands	; the command specifies the source of the rightmost operand - the floating accumulator contains the leftmost operand. The result will be in the accumulator.
III	unary commands	; the command merely specifies the function to be performed on the accumulator. The result will be in the accumulator.
IV	index commands	; the command specifies the source or destination of an index value - the corresponding destination or source is the pseudo index.
V	conditionals	; the command specifies an address to which control passes if the test defined by the command is true - the address must contain another floating point command. Tests may be performed on the floating accumulator, certain flags, and the index.
VI	exit	; this command causes a return to the calling program.

The load & store (Type I) and binary (Type II) commands may specify deferred (indirect and auto-indexed) addressing mode. Deferred addressing in floating point commands operates exactly as in machine language.

2.2 Command Descriptions:

2.2.1 TYPE I COMMANDS -- LOAD & STORE COMMANDS

LOAD FLOATING ACCUMULATOR (AC)

mnemonic	address	code	no. of words
FLDA	X	01	2

The contents of the location specified by X and X + 1 are treated as a floating point number and are loaded into the floating point pseudo accumulator. The floating point number in locations X and X + 1 is split into three parts i) X, which consists of the high order mantissa, goes into FACHI; ii) bits 8-15 of X + 1, which consists of the low order mantissa, goes into bits 8-15 of FACLO and bits 0-7 of FACLO are set to zero; and iii) bits 0-7 of X + 1, which consists of the excess 200_8 exponent, goes into bits 0-7 of FACXP and bits 8-15 of FACXP is set to zero.

DEFERRED LOAD FLOATING AC

mnemonic	address	code	no. of words
FLDAD	A	101	2

The contents of location A is incremented by one, replaced in A, and the result is used as the effective address X; then the contents of A are incremented and replaced a second time forming the effective address X + 1. The contents of X and X + 1 are then treated as a floating point number and loaded into FAC as explained under FLDA.

STORE FLOATING AC

mnemonic	address	code	no. of words
FSTA	X	02	2

The contents of FAC are rounded into bit 8 of FACLO, bits 0-7 of FACLO are set to zero. Then FACHI, FACLO, and FACXP are packed into a floating point number and stored in X, and X + 1. Note that this operation alters FAC so that it agrees with the value stored in X, and X + 1.

It is also possible for the rounding operation to cause exponent overflow (excess 200_8 exponent exceeds $+377_8$). This can occur only if the number being rounded is very close to the largest possible positive floating point number. The value stored in this case will be X = 077777_8, X + 1 = 177777_8, and FXFLG will be set non-zero. A successful FSTA will set FXFLG to zero.

DEFERRED STORE FLOATING AC

mnemonic	address	code	no. of words
FSTAD	A	102	2

The contents of A are incremented twice as explained under FLDAD, forming effective addresses X and X + 1 into which FAC is stored as explained under FSTA.

2.2.2 TYPE II COMMANDS - BINARY COMMANDS

All Type II commands depend on both FAC and the argument of the command to have normalized mantissas. If unnormalized numbers are used, the results are unpredictable. A FNOR instruction (see 2.2.3) is provided to normalize any quantity if it is necessary to do so. Also, if all inputs are normalized, the results in FAC will be normalized as

will the value retrieved from FAC by use of an FSTA instruction.

Type II commands can cause exponent underflow or overflow if the number created in FAC by the command has an excess 200_8 exponent outside the range 0 to $+377_8$ respectively. The occurrence of either condition is indicated by FXFLG being non-zero after the operation has been completed. It may be tested by use of the FJEV command. The successful completion of a Type II command will set FXFLG to zero.

FLOATING ADD

mnemonic	address	code	no. of words
FADD	X	03	2

The floating point number in locations X and X + 1 are added to the contents of FAC, and the result replaces FAC.

DEFERRED FLOATING ADD

mnemonic	address	code	no. of words
FADDD	A	103	2

The contents of A are incremented twice as explained under FLDAD, forming effective addresses X and X + 1, the contents of which are added to FAC, and the result replaces FAC.

FLOATING SUBTRACT

mnemonic	address	code	no. of words
FSUB	X	04	2

The floating point number in locations X and X + 1 are subtracted from the contents of FAC, and the result replaces FAC.

DEFERRED FLOATING SUBTRACT

mnemonic	address	code	no. of words
FSUBD	A	104	2

Effective address is formed from A as in FADDD.

FLOATING MULTIPLY

mnemonic	address	code	no. of words
FMPY	X	05	2

FAC is multiplied by the floating point number in X and X + 1. The result replaces FAC.

DEFERRED FLOATING MULTIPLY

mnemonic	address	code	no. of words
FMPYD	A	105	2

Effective address is formed from A as in FADDD.

FLOATING DIVIDE

mnemonic	address	code	no. of words
FDIV	X	06	2

FAC is divided by the floating point number in X and X + 1. The result replaces FAC. Divide check will occur if X, X + 1 is zero or not normalized. This causes FAC to be set to the largest possible floating point number of the sign which would be the result of the divide if it could take place, and the divide check flag (FDFLG) will be non-zero. A successful divide sets FDFLG to zero.

Note - if both FAC and X are 0, the result will be the largest possible positive floating point number in FAC with FDFLG set non-zero.

DEFERRED FLOATING DIVIDE

mnemonic	address	code	no. of words
FDIVD	A	106	2

Effective address is formed from A as in FADDD.

FLOATING ADD MAGNITUDE

mnemonic	address	code	no. of words
FADM	X	07	2

The absolute magnitude of the floating point number in X and X + 1 is added to FAC. The result replaces FAC.

DEFERRED FLOATING ADD MAGNITUDE

mnemonic	address	code	no. of words
FADMD	A	107	2

Effective address is formed from A as in FADDD.

FLOATING SUBTRACT MAGNITUDE

mnemonic	address	code	no. of words
FSBM	X	10	2

The absolute magnitude of the floating point number in X and X + 1 is subtracted from FAC. The result replaces FAC.

DEFERRED FLOATING SUBTRACT MAGNITUDE

mnemonic	address	code	no. of words
FSBMD	A	110	2

Deferred subtract magnitude. Effective address is formed from A as in FADDD.

2.2.3 TYPE III COMMANDS - UNARY COMMANDS

FLOATING ABSOLUTE VALUE

mnemonic	address	code	no. of words
FABS	none	14	1

The absolute value of the FAC replaces the FAC, i.e. $|FAC|$ replaces FAC.

FLOATING SQUARE

mnemonic	address	code	no. of words
FASQ	none	15	1

The square of FAC is returned in FAC. This instruction requires that the mantissa of FAC be normalized prior to execution as in type II instructions (see 2.2.2).

FLOATING NORMALIZE

mnemonic	address	code	no. of words
FNOR	none	16	1

The contents of FAC are normalized and replace FAC. This instruction can cause exponent overflow or underflow in which case FAC will contain the largest possible negative floating point number or all zeros respectively and FXFLG will be set non-zero. A successful normalize will set FXFLG to zero.

FLOATING NEGATIVE VALUE

mnemonic	address	code	no. of words
FNEG	none	17	1

The contents of FACHI and FACLO are twos complemented, i.e. -FAC replaces FAC.

2.2.4 TYPE IV COMMANDS - INDEX COMMANDS

LOAD INDEX

mnemonic	address	code	no. of words
FLDX	I	27	2

The pseudo-index is loaded with the 16 bit contents of location I.

STORE INDEX

mnemonic	address	code	no. of words
FSTX	I	30	2

The 16 bit pseudo-index is stored into location I.

2.2.5 TYPE V COMMANDS - CONDITIONALS

These commands allow the program to alter the path of control which the interpreter is following based on the results of certain tests. The location to which the interpreter is caused to transfer must contain a valid floating point command. If the interpreter should encounter an invalid command at any time during execution, it will come to a halt with the address of the illegal command displayed in the MB register on the front panel. This is the only halt in the program.

JUMP UNCONDITIONAL

mnemonic	address	code	no. of words
FJMP	C	20	2

Unconditional jump. The interpreter will take the next command from location C and continue from there.

JUMP IF AC POSITIVE

mnemonic	address	code	no. of words
FJAP	C	21	2

If FAC is positive or zero, the interpreter takes the next command from location C. Otherwise, the interpreter continues with the command following the FJAP command.

JUMP IF AC ZERO

mnemonic	address	code	no. of words
FJAZ	C	22	2

If FAC is 0, the interpreter will take the next command from location C. Otherwise, the interpreter continues with the command following the FJAZ command. Note: The interpreter tests only FACHI for zero. FAC may be non-zero and FACHI = 0 only if the number in FAC is not normalized. This condition cannot be created by the interpreter unless the user has introduced unnormalized numbers into his calculations (see 2.2.2).

JUMP IF AC NEGATIVE

mnemonic	address	code	no. of words
FJAN	C	23	2

If FAC is negative, the interpreter will take the next command from location C. Otherwise, the interpreter continues with the command following the FJAN command.

JUMP IF EXPONENT OVERFLOW (OR UNDERFLOW)

mnemonic	address	code	no. of words
FJEV	C	24	2

If FXFLG is non-zero, the interpreter will take the next command from location C and set FXFLG to zero. Otherwise, the interpreter will continue with the command following the FJEV command. The FJEV command is used to detect the occurrence of either exponent overflow or exponent underflow resulting from the execution of the last preceding Type II command or FSTA, FNOR, or FASQ. If desired, the type of overflow may be detected by an FJAZ command at location C, since exponent underflow returns FAC=0, and exponent overflow returns the largest number (+ or -) in FAC.

JUMP IF DIVIDE CHECK

mnemonic	address	code	no. of words
FJDC	C	25	2

If FDFLG is non-zero, the interpreter will take the next command from location C and set FDFLG to zero. Otherwise, the interpreter continues with the command following the FJDC command. The FJDC command is used to detect the occurrence of divide check during execution of the last previous FDIV or FDIVD command. If desired, one may test whether the condition occurred because the divisor was 0 or not normalized by checking the divisor with an FLDA and FJAZ instruction at location C.

JUMP IF INCREMENTED INDEX NOT ZERO

mnemonic	address	code	no. of words
FJIX	C	26	2

The pseudo-index (FINDX) is incremented by one, and if the result is non-zero, the interpreter takes the next command from location C.

If the result is 0, the interpreter continues with the command following the FJIX instruction. The pseudo-index will contain the incremented value whether or not the jump occurs.

2.2.6 TYPE VI COMMAND - EXIT

EXIT FLOATING INTERPRETER

mnemonic	address	code	no. of words
FEXT	none	0	1

This command causes the interpreter to return control to the user at the location immediately following the FEXT. None of the internal registers or flags are altered by either the FEXT or entering the package. The AO is returned in the ADD state.

CHAPTER THREE

DATA CONVERSION

3.1 Introduction:

Two conversion routines are provided; one to convert from floating point to character, the other to convert from character to floating point. Both conversion routines are core to core operations rather than being bound to a particular I/O device (that is, characters are fetched from and stored into memory). For added flexibility, all characters are referenced with an index into a character set table called @FCST, which initially contains 8-bit ASCII codes. Changing the character set for a specific I/O device can easily be accomplished by changing the character codes in @FCST.

External floating point format is expressed as a mantissa or fraction portion and a power of ten by which the mantissa is multiplied. This is written as $\underbrace{\pm n.nnnnnn}_{\text{mantissa}}\ \underbrace{E\ \pm nn}_{\text{exponent}}$, where n is a decimal digit. The number to the right of the E is the power of ten by which the mantissa is multiplied. Thus, -3.527614E+03 is $-3.527614*10^3$ or -3527.614. The floating point number +172.100123E-02 is $+172.100123*10^{-2}$ or +1.72100123.

3.2 Floating Point to Character Conversion:

NAME:	@SFC
SUBROUTINES CALLED:	$SFI, @FXC
ALTERED REGISTERS & FLAGS:	FAC, FTM, FXFLG, FDFLG
CALLING SEQUENCE:	JU @SFC WRD e1-1 WRD e2 WRD e3 return
ARGUMENTS:	e1 is the address of the location into which the first output character is to be stored. e2 is the address of the two word floating point argument. The argument need not be normalized but the magnitude must be zero or in the range $(2^{-129}, 2^{+128})$ (in decimal this is 1.469367E-39 to 1.701411E+38) e3 is the address of the error return.
FUNCTION:	Converts a signed two word floating point argument to a string of thirteen characters, stored one character per word, right justified starting in location e1. The character string is of the format $\begin{Bmatrix} + \\ * \\ - \end{Bmatrix}$ n.nnnnnn E$\begin{Bmatrix} + \\ - \end{Bmatrix}$nn where n is character representation of a decimal digit.

ERRORS: If normalization of the floating point argument caused either exponent overflow or underflow, an * is stored rather than a leading + or - sign, and when conversion is completed, control returns to e3. The * can be considered a - sign. An argument resulting in overflow converts to * 1.701411 E + 38. An argument resulting in underflow converts to *0.000000 E + 00.

NOTE: The magnitude of the three smallest normalized non-zero floating point numbers are converted to one of the character strings $\pm$1.469367E-39 or $\pm$1.469368E-39. These two character strings cannot be converted back to a floating point number. The smallest character string which can successfully be converted to a floating point number is $\pm$1.469369E-39. Therefore, if the user converts any one of these three numbers to a string of characters, he should be aware that he cannot successfully convert the string back to a floating point number.

LENGTH: 306_8 (198_{10}) locations

Description of Algorithm:

The sign of the floating point argument is stored, the argument is then normalized, and the absolute value is taken and used for conversion. If normalization caused either exponent overflow or underflow, the error return is taken when conversion is completed and an asterisk (which may be considered as a '-'), is stored rather than a leading sign.

Since the output character string is of the form $\begin{pmatrix} * \\ + \\ - \end{pmatrix}$n.nnnnnnE±nn, the floating point argument is first manipulated to make it greater than or equal to one and less than ten. (If the floating point argument is exactly zero, this portion of the algorithm is bypassed.) Making $1 \leq$ floating point argument < 10 is accomplished by first checking if it is ≥ 1. If it is not, it is multiplied by the largest possible power of ten (10^{38}) and, if necessary, it is multiplied once again by ten to force it ≥ 1. The argument is then checked for < 10. If it is not < 10, it is forced so by dividing by the largest power of ten, which is less than the argument. The powers of ten used in multiplying and dividing the argument to force its value to be between one and ten are used to form the exponent portion of the character string.

With the floating point argument (stored in FAC) now ≥ 1 and < 10, the mantissa portion of the character string can be formed. FACHI, FACLO is treated as a double precision mixed number with FACXP showing the position of the binary point. FACHI, FACLO is left shifted (with overflow bits shifted into a 3rd word) until the binary point immediately precedes bit 15 of FACHI. The overflow word is then converted to character and stored as the first digit of the mantissa, immediately followed by a decimal point. The fraction portion of the mantissa is formed by successively multiplying FACHI, FACLO by 10_{10} and storing the most significant word of the 3 word product. The exponent is then converted and stored, preceded by an E and either a + or - sign.

3.3 Character to Floating Point:

NAME: @SCF

SUBROUTINES CALLED: $SFI, @FXC

ALTERED REGISTERS & FLAGS: FAC, FTM, FXFLG, FDFLG

CALLING SEQUENCE:

```
JU @SCF
WRD e1-1
WRD e2
WRD e3
return
```

ARGUMENTS: e1 is the address of the first character in the string to be converted. The character string should be stored one character per word right justified in the format

$$\left[\begin{matrix}+\\-*\end{matrix}\right] \quad \left\{\begin{matrix}n\ [n...]\ .\ [n...]\\ [.]\ n\ [n...]\end{matrix}\right\} \quad \left[\ \left\{\begin{matrix}E\ \ [\pm]\\+\\-\end{matrix}\right\}\ n[n]\ \right] \quad [\Delta...] \quad [\ )\ ]$$

The notational conventions are:

1. n is a decimal digit
2. Δ is a space
3.) is a delimiter
4. braces [] contain optional items which may or may not be included.
5. brackets { } contain alternate items where one and only one of the items must be included.
6. ellipses ... denote permissible repetition of the preceding item.

The string is treated as follows:

1. If there is no sign, it is treated as +.

2. If the leading sign is * or -, it is treated as -.
3. If there is no decimal point, it is assumed to follow the last mantissa digit.
4. Characters are processed up to and including the first ↵, or 13_{10} characters have been processed.

e2 is the address where the two word floating point answer is stored.

e3 is the address of the error return.

FUNCTION:

Converts a string of decimal characters to a two word normalized floating point answer. The two word normalized floating point answer is returned in registers AX (MSH), AY (LSH), and is stored in location e2 (MSH) and e2+1 (LSH). The AO is returned in the ADD state.

ERRORS:

A scan error occurs if the character string is illegally formed. Location @SCF+3 is set to zero and control immediately returns to e3.

An overflow error occurs if the character string contains more than 10_{10} mantissa digits (discounting leading zeros) or if the magnitude of the number is outside the range 1.469369E-39 to 1.701411E+38. Location @SCF+3 is set to one and control immediately returns to e3.

Whenever control returns to e3, the AO is in the ADD state.

NOTES: See NOTES under @SFC.

LENGTH: 406_8 (262_{10}) locations

Description of Algorithm

The mantissa portion of the character string is converted to a double precision integer by multiplying the answer by 10_{10} and adding in the latest digit. This double precision mantissa is then converted to a normalized floating point number. A count of the number of digits to the right of the decimal point is kept and, after the exponent portion of the character string has been converted, this digit count is subtracted from it to obtain the final exponent. The magnitude of the final exponent is used as an index into the positive floating point powers of ten table (see 3.6). The floating point number obtained from the mantissa portion of the character string is then multiplied (if the final exponent was positive) or divided (if the final exponent was negative) by this power of ten to form the final floating point answer. If there was a leading minus sign or asterisk, the floating point answer is two's complemented before return.

3.4 Common Tables & Routines:

The conversion routines @SFC and @SCF reference a common routine called @FXC, which has four entry points. @FXC occupies a total of 213_8 (139_{10}) locations. Since @FXC is common to both @SFC and @SCF, it need appear only once if the conversion routines are used together. In the discussion of @FXC which follows, each of the four entry points is treated separately for the sake of clarity.

3.5 Character Set Table:

NAME: @FCST

FUNCTION: Common external character set table for floating point data conversion routines. The table is ordered as follows:

Location	Contents
@FCST	code for zero
@FCST+1	code for nine
@FCST+2	code for +
@FCST+3	code for -
@FCST+4	code for *
@FCST+5	code for .
@FCST+6	code for E
@FCST+7	code for space
@FCST+10	code for delimiter

NOTES: The standard table is in full 8-bit ASCII. The delimiter character at @FCST+10_8 is a carriage return and may be changed if desired.

The entire table may be replaced with a different character set provided that the numeric codes in the new set are sequential and the code for zero (0) is less than the code for nine. No code may occupy more than 15 bits.

LENGTH: 12_8 (10_{10}) locations

3.6 Floating Point Powers of Ten Table:

NAME: @FPT

FUNCTION: Common floating point positive powers of 10_{10} table for floating point data conversion routines. Each floating point power occupies two locations in the table. The table is organized as follows:

```
@FPT:   WRD     45473,46777     constant for 10^38
        WRD     74136,160773    constant for 10^37
         .
         .
         .
         .
        WRD     40000,201       constant for 10^0
```

NOTE: @FPT is located at @FCST+12_8

LENGTH: 116_8 (78_{10}) locations

3.7 Left Shift FAC:

NAME:	@LSHF
SUBROUTINES CALLED:	none
ALTERED REGISTERS & FLAGS:	none
CALLING SEQUENCE:	Load AX with the negative shift count JU @LSHF return
ARGUMENTS:	Register AX contains minus the number of places to left shift FACHI, FACLO
FUNCTION:	Performs double precision left shift of FACHI, FACLO. On return, the shifted result is in AX (MSH), AY (LSH). Any carry out of MSH is found in location @DIG+1.
ERRORS:	none detected
NOTES:	@LSHF=@FCST + 306_8 @DIG+1=@LSHF + 14_8
LENGTH:	23_8 (19_{10}) locations

3.8 Multiply FAC by Ten:

NAME:	@10X
SUBROUTINES CALLED:	@LSHF
ALTERED REGISTERS & FLAGS:	FAC
CALLING SEQUENCE:	JU @10X
ARGUMENTS:	n/a
FUNCTION:	Performs unsigned multiplication of FACHI, FACLO by 10_{10}. The most significant word of the three word product is returned in AY. The second and third words of the product are found in FACHI, FACLO respectively.
ERRORS:	n/a
NOTES:	@10X=@FCST+650
LENGTH:	40_8 (32_{10}) locations

CHAPTER FOUR

EXTENDED COMMANDS

4.1 Introduction:

In addition to the basic floating point interpreter, a set of mathematical functions is supplied which can be invoked by a command in the same line with the basic commands. These functions also call the floating interpreter and since the interpreter has already been entered at this point, a push-down scheme is supplied to allow recursive calls such as this. The push-down list will accomodate recursive calls up to seven levels.

It should be noted that all pseudo registers - the floating accumulator, the temporary accumulator and index - and the flags, FDFLG and FXFLG, are common to all levels of the recursion. In other words, if an extended function which calls the interpreter recursively is invoked by a command, these registers and/or flags may be altered. Information detailing such factors is supplied in the documentation accompanying the individual package.

The push-down scheme and command code structure is tailored so that the user may easily add his own functions. The procedure for doing this is described in section 5.3.

The mathematical subroutines which are supplied with the extended package are SINE, COSINE, ARC TANGENT, LOG_e, EXPONENTIAL, and SQUARE ROOT. The commands associated with these are FSIN, FCOS, FATN, FLNE, FEXP, and FSQT (codes 31, 32, 33, 34, 35, and 36) respectively. They each perform the desired function on the contents of the floating pseudo-accumulator and return the results in the same register. Errors which can result, such as attempting to take the square root or log of a negative number, are flagged by the

routines in internal locations not accessible in interpretive mode, i.e. cannot be tested with an interpreter command. An error trap routine is available which will handle these and other errors when they occur (see Appendix E).

In the writeups that follow, FAC is the floating pseudo-accumulator, FTM is the temporary floating pseudo-accumulator, FDFLG is the divide check flag, FXFLG is the exponent overflow flag, and FINDX is the pseudo-index.

4.2 Sine, Cosine:

COMMAND: FSIN (code 31), FCOS (code 32)

FUNCTION:

a. FSIN - calculates the SINE of the contents of FAC which is assumed to be a radian argument and replaces FAC with the result.

b. FCOS - calculates the COSINE of the contents of FAC which is assumed to be a radian argument and replaces FAC with the result.

ERRORS: none

ALTERED REGISTERS & FLAGS: FAC, FTM, FXFLG

METHOD:

For FCOS, the absolute value of FAC is subtracted from $\pi/2$ (=1.570796) and the SINE of the result is taken.

For FSIN, the argument (FAC) is first multiplied by $2/\pi$ to convert it into units of a quarter circle, and the result is checked for its absolute magnitude being less than one. If so, it is a first quadrant quantity and the procedure continues with the series calculation described later. If the magnitude of the result is greater than or equal to one, its sign is saved, it is forced positive, and the integer portion is shifted out - leaving a positive fraction (referred to as Y in the following). The last two bits of the integer portion and the sign are used to determine which quadrant the original argument was in and the quantity Y is altered as follows:

sign	last two bits	Y	quadrant
+	00	Y → Y	I
+	01	1-Y → Y	II
+	10	-Y → Y	III
+	11	-1 + Y → Y	IV
-	00	-Y → Y	IV
-	01	-1 + Y → Y	III
-	10	Y → Y	II
-	11	1-Y → Y	I

This new value of Y is then treated as a fraction and is normalized.

The series used to calculate the sine is basically a 5 term Chebyshev economized polynomial approximation of a 6 term McLaurin series for $\sin\left(\frac{\pi}{2}Y\right)$. The coefficients are further "adapted" to allow the series to be calculated with one less multiplication than would be the case for a standard polynomial evaluation procedure. This results in the sine being calculated as follows:

$$\sin\left(\frac{\pi}{2}Y\right) = ((Z - Y + A_2) * Z + A_3) * A_4 * Y$$

where

$$Z = (Y + A_0) * Y + A_1$$

and

$A_0 = -14.93104811$

$A_1 = -39.74079011$

$A_2 = +367.8139482$

$A_3 = +23410.00773$

$A_4 = +0.0001514440767$

Accuracy is 6 + significant decimal digits for arguments in the first

quadrant ($|FAC| \leq \frac{\pi}{2}$). Accuracy loss is about two thirds of a decimal digit for each complete rotation, i.e. if $2\pi n \leq |FAC| < 2\pi(n+1)$, the accuracy is about $6 - \frac{2}{3}n$ decimal digits.

4.3 Arc Tangent:

COMMAND: FATN (code 33)

FUNCTION: The arc tangent of the contents of FAC replace FAC. The result is in radians and lies in the range $(-\frac{\pi}{2}, +\frac{\pi}{2})$.

ERRORS: none

ALTERED REGISTERS & FLAGS: FAC, FTM, FDFLG, FXFLG

METHOD: The argument (FAC) is checked for its absolute magnitude being greater than or equal to one. If so, a flag is set and the reciprocal of the argument is taken and replaces FAC.

The arc tangent of the quantity in FAC is then approximated by

$$Z = \text{ATAN } X = X \cdot \frac{(A_0 + A_1X^2 + A_2X^4)}{(B_0 + B_1X^2 + B_2X^4)}$$

where X is the argument and

$A_0 = 0.6402481953$

$A_1 = 0.4229908144$

$A_2 = 0.0264694361$

$B_0 = 0.6402487022$

$B_1 = 0.6363779373$

$B_2 = 0.1108328778$

If the flag was set by the initial check, the value Z is checked for + or -. If Z is +, $\left(\frac{\pi}{2} - Z\right)$ replaces Z. If Z is -, $\left(-\frac{\pi}{2} + Z\right)$ replaces Z. (This is effected by subtracting Z from + or $-\frac{\pi}{2}$ depending on the sign of Z.)

If the flag was not set by the initial check, the value Z is not altered. Accuracy is 6+ significant decimal digits for all arguments.

4.4 Natural Log:

COMMAND: FLNE (code 34)

FUNCTION: The natural log of the contents of FAC replace FAC.

ERRORS: If FAC is negative, a flag (FNLNF) is set, FAC is forced positive, and the natural log taken.

ALTERED REGISTERS & FLAGS: FAC, FTM, FXFLG, FNLNF (FPLNE+4)

METHOD: The quantity in FAC is

$Z = X \cdot 2^I$ where $.5 \leq X < 1$ and I is an integer.

$$\ln Z = \ln [X \cdot 2^I]$$
$$= \ln X + I \ln 2$$

The quantity ln X is approximated by the polynomial.

$$\ln X = \ln A - Z (Y + Y^3/3 + Y^5/5 + Y^7/7)$$

which is a Taylor series evaluated at A

where $A = \frac{1}{\sqrt{2}}$

and $Y = \frac{A - X}{A + X}$

The product [I ln 2] is added to ln X, and the sum is left in FAC.

A = 0.70710678

ln A = 0.34657359

ln 2 = 0.69314718

Accuracy is 6+ significant decimal digits except for $.904 \leq Z \leq 1.110$. In the latter range, accuracy decreases as $Z \rightarrow 1$.

4.5 Exponential:

COMMAND: FEXP (code 35)

FUNCTION: The exponential of the contents of FAC replace FAC. (FAC = e^{FAC})

ERRORS: If the result is going to be out of range, i.e. if FAC 88.722, a flag (FEXOF) is set. If FAC was negative, zero is left in FAC. If it was positive, the largest positive number is left.

ALTERED REGISTERS & FLAGS: FAC, FTM, FDFLG, FXFLG, FEXOF (FPEXP+1)

METHOD:

$$e^X = 2^{X \log_2 e}$$

$$= 2^{I+F} = 2^I \cdot 2^F$$

where I is the integer portion
and F is the fractional portion of $X \log_2 e$
Multiplication by 2^I is computed by the continued fraction:

$$\cfrac{A}{B + F + \cfrac{C}{F + \cfrac{D}{F}}}$$

where

A = -34.624680982

B = -17.312340491

C = 104.0684491

D = 20.813689813

$\log_2 e = 1.442695041$

Accuracy is 6+ significant decimal digits for $|X| \leq 10$. Accuracy decreases slowly as $|X|$ becomes large until at $|X| \approx 88$, the accuracy is 5+ significant decimal digits.

4.6 Square Root:

COMMAND: FSQT (code 36)

FUNCTION: The square root of |FAC| replaces FAC.

ERRORS: If FAC is negative, it is forced positive, and FSFLG (internal to the square root routine) is set non-zero. If FAC is positive, FSFLG is set to zero.

ALTERED REGISTERS & FLAGS: FAC, FSFLG (=FPSQT + 6)

METHOD: After FAC is forced positive and FSFLG is determined, the exponent of the result is determined by dividing FACXP by two (by shifting right once) and adding 100_8 to preserve the excess 200_8 notation. If the original exponent was odd, the shifted FACXP is increased by one; otherwise, it is left alone. If the original exponent was even, FACHI and FACLO are shifted left once. Since the algorithm treats FACHI and FACLO as a 32 bit positive fraction with the binary point to the left of bit 15 of FACHI, the fact that the left shift will set the sign bit (bit 15) of FACHI does not matter.

The algorithm then proceeds to determine a fourteen bit first approximation to the square root by a method based on the fact that N^2 is the sum of the first N odd numbers. This method also leaves as a "remainder" the difference between the square of the approximation and the original

number. This remainder and the initial approximation are then used for one Newton-Raphson iteration which completes the square root using the single precision divide entry (FSDVD) of the floating point package.

Accuracy is 6+ significant decimal digits for all input arguments.

CHAPTER FIVE

NON-INTERPRETIVE MODE USAGE

5.1 Introduction:

Certain sections of the floating point interpreter are directly accessible to the user without the need to supply commands. These sections may be invoked by a JU SUBR instruction and, after the operation is completed, will return control to the instruction following the jump. In order to use these routines successfully, it is necessary to know that in addition to the pseudo-accumulator (FACHI, FACLO, and FACXP) there is a "temporary" accumulator (FTMHI, FTMLO, and FTMXP) which is used to contain the floating argument of a Type II command during the execution of the operation (see 1.3). This temporary pseudo-accumulator, referred to as FTM, is loaded in the same manner as FAC (see FLDA instruction in 2.2.1). If the user desires to access the routines described in this section, he may need to load FTM in addition to FAC for those routines that operate on both accumulators.

These sections will be described as subroutines since they are essentially used in this manner when accessed directly. When the floating interpreter resides in memory, all of these subroutines also lie in memory.

A "command equate" source tape is included in the software package. This tape defines these subroutines as well as the various pseudo-registers and locations associated with them (see chapter 6).

5.2 Subroutines:

5.2.1 Double Precision Fixed Point Add

NAME:	FDAD
CALLING SEQUENCE:	JU FDAD
INPUT:	FACHI, FACLO; FTMHI, FTMLO; AO must be in ADD state.
FUNCTION:	FACHI, FACLO and FTMHI, FTMLO are treated as signed double precision numbers and added. The result of the addition appears in FACHI, FACLO. FTMHI, FTMLO are left unchanged. If arithmetic overflow occurred (two numbers of like sign are added and the result has opposite sign), the link will be set to 1. If no arithmetic overflow occurred, the link will be zero. The AO is in the ADD state upon return.
NOTES:	It is possible to generate the maximum negative number (FACHI = 100000_8, FACLO = 000000_8), which is <u>not</u> considered a case of arithmetic overflow; and so the link will not be set.

5.2.2 Double Precision Fixed Point Multiply

NAME:	FDMPY
CALLING SEQUENCE:	JU FDMPY
INPUT:	FACHI, FACLO; FTMHI, FTMLO AX must be set to the value in FTMHI AY must be set to the value in FTMLO The AO must be in the ADD state
FUNCTION:	FACHI, FACLO and FTMHI, FTMLO are treated as signed double precision numbers and are multiplied. The high-order 30 bits of the 62 bit product are returned, right justified, in FACHI, FACLO. The value in FTMHI, FTMLO is unchanged. The AO is in the ADD state upon return.
NOTES:	The 30 bit product is inaccurate in the right-most two bits. If FACHI, FACLO and FTMHI, FTMLO are each considered as a double precision fraction with its binary point immediately to the right of the sign, i.e. between bits 14 and 15 of the high-order word, the binary point of the product will be shifted right once so that it is between bits 13 and 14 of FACHI.

5.2.3 Double Precision Fixed Point Divide

NAME:	FDDIV
CALLING SEQUENCE:	JU FDDIV
INPUT:	FACHI, FACLO; FTMHI, FTMLO AX must be set to the value in FTMHI AY must be set to the value in FTMLO The AO must be in the ADD state
FUNCTION:	FACHI, FACLO and FTMHI, FTMLO are treated as signed double precision numbers, and the former is divided by the latter. The quotient appears in FACHI, FACLO. The value in FTMHI, FTMLO has been destroyed. The quotient will be 30 bits in FACHI, FACLO with the binary point displayed one position to the right in the same way as explained in the note for FDMPY. The absolute magnitude of FTMHI, FTMLO must have bit 14 of FTMHI set for the divide to take place. If this condition is not satisfied, divide check will occur. The AO is in the ADD state upon return.
NOTES:	The rightmost three bits of the quotient are inaccurate. Divide check causes FACHI, FACLO to be set to a large double precision number of the sign which would result if the divide could take place (FACHI, FACLO = 077777, 177400 or 100000, 000400 for + and - respectively); also, FDFLG is set non-zero. A successful divide sets FDFLG to zero.

5.2.4 Single Precision Divide

NAME: FSDVD

CALLING SEQUENCE: JU FSDVD

INPUT:

AX = high order dividend
FLODV = low order dividend

} must be a positive 30 bit double precision number (see below)

AY = negative divisor

The AO must be in the ADD state.

FUNCTION:

This is an inner loop which, if used correctly, can be invoked to supply an unsigned single precision divide. The quotient is incomplete in the sense that it is right shifted and truncated upon return.

To obtain a complete single precision unsigned divide, the following procedure may be used. First, load AX and the location FLODV with a valid two word positive product (bits 14 and 15 of AX must be zero). Then load AY with the positive single precision divisor and twos complement it. The following code will then perform the divide:

```
        JU      FSDVD           ;INCOMPLETE QUOTIENT IN TRP
        RRC     AO,L1,0         ;GET LAST BIT OF QUOTIENT
        SFM     NOT LNK         ;UPDATE
        RR      AO,AX           ;REMAINDER
        NOP                     ;IN AX
        RR      TRP,L1,AY       ;TRUE QUOTIENT IN AY
```

This code may be eliminated if the remainder is to be disregarded.

Note that the incomplete quotient is in the TRP register on return from FSDVD. The AO is in the ADD state upon return.

If either the link is set or AY (the final quotient) is negative following this code, divide check has occurred. This means that the high-order portion of twice the dividend was greater than or equal to the divisor, and the quotient is incorrect.

NOTES:

No flag is set if divide check occurs.

5.2.5 Floating Point Normalize

NAME: FNORM

CALLING SEQUENCE: JU FNORM

INPUT: FACHI, FACLO, FACXP

FUNCTION: Same as FNOR command (see 2.2.3), including the setting of FAC and FXFLG should exponent overflow or underflow occur.

The advantage of the accessibility of this routine lies mainly in the saving of time. For instance, to convert a single precision integer value to floating point, the following two methods could be used. (Assume the integer is in AX, and the floating equivalent is wanted in location X.)

1)
```
RM      AX, FACHI
ZM      FACLO
MRI     217, AX
RM      AX, FACXP
JU      FNORM
JU      $SFI
FSTA    X
FEXT
 .
 .
 .
```

or,

2)
```
RM      AX, FACHI
ZM      FACLO
MRI     217, AX
RM      AX, FACXP
JU      $SFI
FNOR
FSTA    X
FEXT
```

Version 1) takes one more location in core and saves about 80 machine cycles.

NOTE: The AO may not be in the ADD state upon return.

5.2.6 Negation and Store

NAME:	FACMP, FACMA
CALLING SEQUENCE:	JU FACMP or JU FACMA
INPUT:	FACHI, FACLO or AX, AY
FUNCTION:	a) FACMP - replaces FACHI, FACLO with its two's complement. Result is also returned in AX, AY. b) FACMA - replaces FACHI, FACLO with the two's complement of the double precision number in AX, AY. Result is also returned in AX, AY.
NAME:	FTCMP, FTCMA
CALLING SEQUENCE:	JU FTCMP or JU FTCMA
INPUT:	FTMHI, FTMLO or AX, AY
FUNCTION:	a) FTCMP - replaces FTMHI, FTMLO with its two's complement. Result is also returned in AX, AY. b) FTCMA - replaces FTMHI, FTMLO with the two's complement of the double precision number in AX, AY. Result is also returned in AX, AY.
NAME:	FASAX
CALLING SEQUENCE:	JU FASAX
INPUT:	AX,AY
FUNCTION:	Stores AX into FACHI and AY into FACLO
NAME:	FTSAX
CALLING SEQUENCE:	JU FTSAX
INPUT:	AX,AY
FUNCTION:	Stores AX into FTMHI and AY into FTMLO

5.2.7 Generate Zero or Largest Number

NAME: FOFAC

CALLING SEQUENCE: JU FOFAC

INPUT: none

FUNCTION: sets FACHI, FACLO and FACXP to zero also returns AX and AY = 0

NAME: FCMAX

CALLING SEQUENCE: JU FCMAX

INPUT: FACHI

FUNCTION: FACHI, FACLO, FACXP will be set to the maximum possible floating point number of the original sign of FACHI.

1) If FACHI <0, this routine sets

FACHI = 100000
FACLO = 000400
FACXP = 000377

2) If FACHI ≥ 0, this routine sets

FACHI = 077777
FACLO = 177400
FACXP = 000377

Upon return, AX and AY will be equal to the value stored in FACHI and FACLO respectively.

5.2.8 Floating Arithmetic Right Shift

NAME:	FARSN
CALLING SEQUENCE:	JU FARSN
INPUT:	AX, AY, FARSC
FUNCTION:	This routine arithmetically right shifts the double precision number in AX, AY by the number of places indicated by -FARSC.
NOTES:	FARSC must be set to a negative count before calling FARSN.

5.2.9 Other Notes on Non-Interpretive Usage

1) The pseudo-index is kept in location FINDX and may be set by the user without using an FLDX command (see 2.2.4) by simply storing the desired value via a RM R,FINDX where R is a register containing the index value. This, as with FNORM, is a time saver.

2) The two flags, FXFLG and FDFLG, are in locations defined by their names, and can be checked (or cleared) in non-interpretive mode to save time.

3) The usage of the locations FTBLE, FARGD, FETCH, and FMASK which are on the command equate source tape is described in 5.3.

5.3 User Generated Extended Functions:

If the user desires to add functions of his own to the extended package, the procedure is quite easy as outlined below.

The extended package as delivered uses command codes $00\text{-}36_8$ inclusive and 101_8 through 110_8 inclusive. There are available codes of $37_8\text{-}77_8$ inclusive which the user may assign to his own functions.

User functions may be of two types - invoked by one word commands or invoked by two word commands where the second word is an argument address or value. If deferred mode addressing is desired as an option for the same function, it must be accomplished by user code. Setting bit 6 of the command code to attempt deferred addressing will cause the floating interpreter to take the error halt.

Suppose the command name used to invoke the function is to be FFCN

assigned to code 37_8.

Step 1) Using the Source Text Editor, add the statement FFCN = 37 to the Command Equate Tape (see operating instructions).

Step 2) The user code which accomplishes the function must have the following code just before the END statement

```
LOC  FTBLE + FFCN
WRD  ENTRY
```

where ENTRY is the location at which the user function begins execution.

Step 3) The last instruction executed by the user function must return control to FGET, usually via a JU FGET. Remember that when the user function is invoked by a command, the interpreter passes control to the user function. The JU FGET returns control to the interpreter.

Step 4) If the function the user is generating needs the floating point capability supplied by the interpreter, the user function may call the interpreter followed by a list of commands to accomplish the task subject to the following restrictions:

a) The command name corresponding to the function itself (in this case, FFCN) may not be used.

b) Commands which cause the interpreter to be called recursively may be used so long as care is taken not to exceed seven levels of recursion in total (see 4.1) (remember that the function being coded is at least at level 1 during its execution, and if it calls the interpreter, all commands in the list are at least at level 2).

c) No function invoked by a command may have in its code

a call to the interpreter whose command string contains the command name corresponding to the function itself. This is an indirect violation of restriction a) above.

Step 5) Assemble the function using the Command Equate Tape for pass 1 as explained under operating instructions.

Notes: If the function being generated is invoked by a two word command whose second word is an argument, one and only one of the following steps must occur during its execution.

a) JU FARGD

This fetches the contents of the location following the command into register AX.

b) JU FETCH

This calls FARGD and uses the contents of the location following the command as an address to fetch a floating point argument which is placed in FTM. Also, AX and AY will be set to the value in FTMHI and FTMLO respectively upon return.

c) ZM FMASK
JU FETCH

This causes deferred fetching of a floating point argument. The contents of the location following the command is used as an address of another location which is incremented twice to form the addresses of the floating argument which is loaded into FTM and AX, AY as in b).

Examples:

1) FCSX is to be the command name, 1 word, code 37_8. When invoked it is to take the COSINE of the SINE of the value in FAC. Assume Step 1 has been accomplished by adding the statement FCSX=37 to the command

equate tape. This function may be accomplished by the following code:

```
FCS:    JU $SFI              ;enter floating interpreter
        FSIN                 ;sin of FAC
        FCOS                 ;cos of FAC
        FEXT                 ;exit interpreter
        JU FGET              ;return to interpreter
        LOC FTBLE + FCSX
        WRD FCS
        END
```

When this is assembled with the Command Equate Tape resulting from step 1 and loaded with $SFI and the SINE, COSINE routine, the user may now call the routine in the floating interpretive mode as follows:

```
JU  $SFI
    .
    .
    .
FCSX
    .
    .
    .
FEXT
```

Note: The routine must be assembled with the new command equate tape.

2) FMCS is to be the command name, 2 words, assigned to code 40_8. When invoked, it is to take the SIN of the COSINE of the value in FAC and set the sign of the result to the sign of the floating point argument whose address is the second word of the command. Step 1 requires the new command to be added to the command equate tape. This function could be coded as follows:

```
FMC:    JU FETCH            ;fetch arg to AX,AY
        RMI AX, 0           ;save MSH arg (sign of arg)
        JU  $SFI            ;enter floating interpreter
        FCOS                ;cos of FAC
        FSIN                ;sin FAC
        FABS                ;abs value of FAC
        FEXT                ;exit floating interpreter
        MR FMC + 3,AX       ;get sign of arg
        JC  AX, GEZ, FGET   ;plus, exit
        JU  FACMP           ;minus, comp FAC
        JU  FGET            ;return to interpreter
        LOC FTBLE + FMCS
        WRD FMC
        END
```

When assembled with the Command Equate Tape and loaded with $SFI, and the SINE, COSINE routine, it may be invoked by another routine assembled with the Command Equate Tape via

```
JU  $SFI
    .
    .
    .
FMCS  X
    .
    .
    .
FEXT
```

3) FMCSD is to be the command which does the same thing as FMCS, only using deferred mode addressing for the argument. FMCSD must be assigned a different code - say 41_8. Both FMCSD and FMCS may be coded in the same routine as follows, assuming their equates have been added to the Command Equate Tape.

```
FMCD:  ZM FMASK
 FMC:  JU FETCH
       RMI AX, 0
       JU $SFI
       FCOS
       FSIN
       FABS
       FEXT
       MR FMC + 3, AX
       JC AX, GEZ, FGET
       JU FACMP
       JU FGET
       LOC FTBLE + FMCS
       WRD FMC
       LOC FTBLE + FMCSD
       WRD FMCD
       END
```

CHAPTER 6

OPERATING INSTRUCTIONS AND SYSTEM GENERATION

6.1 Using the Package as Supplied:

A Command Equate Tape is supplied for the package as delivered. This tape is a source tape and contains the definitions of the floating point commands as well as the entries for $SFI, the conversion routines, and the names and entry points for the various flags and routines in $SFI that can be used in non-interpretive mode or by user coded extended functions. A commented listing of this tape appears in Appendix B of this document. The tape supplied with the package is not commented.

This tape is intended for use during assembly of the user programs which reference all or part of the Floating Point System. The user need not define the names or locations appearing on this tape when he generates the source tape for the program he has written. Assuming the source tape is generated, the following procedure will assemble the program for use with $SFI:

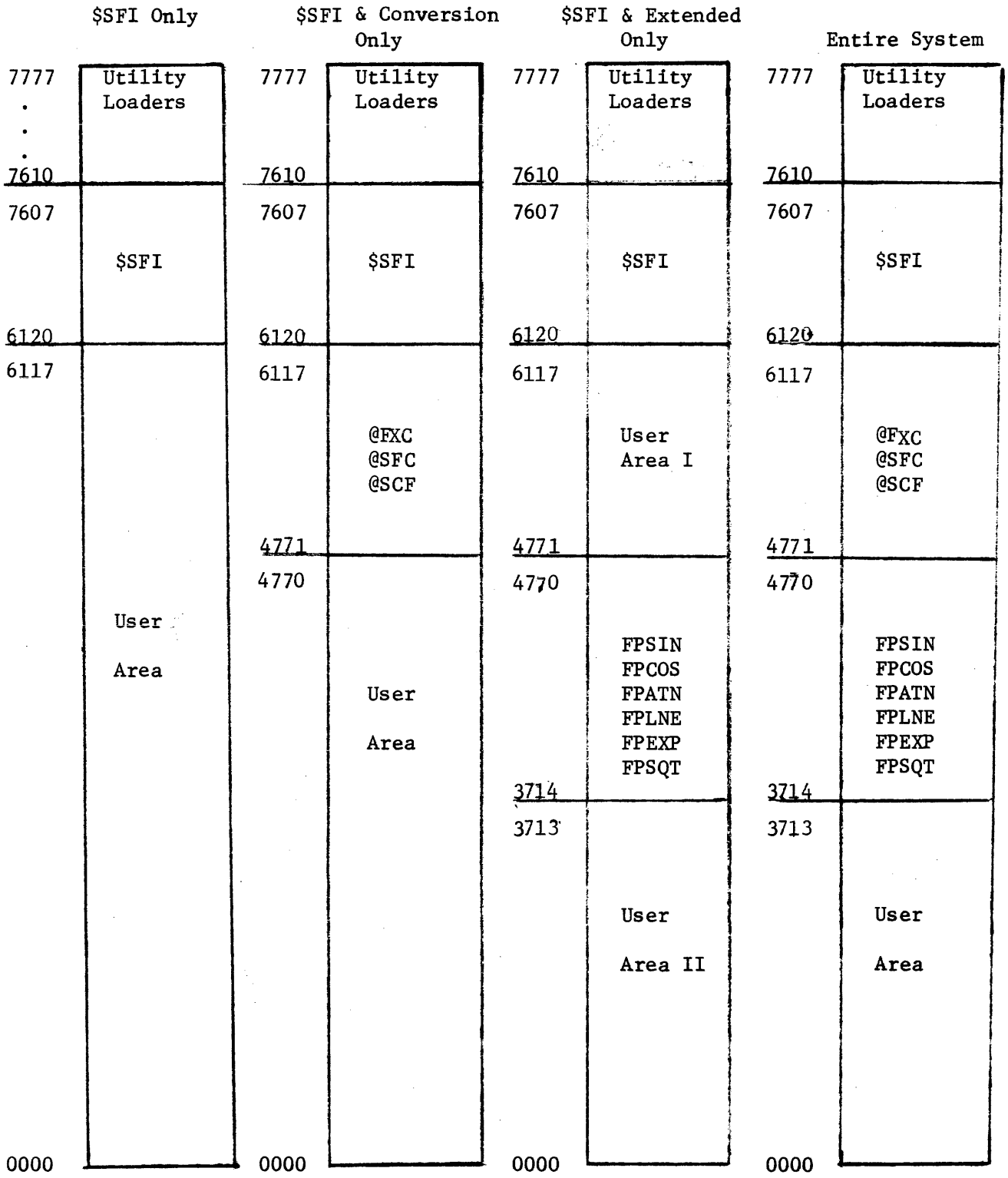
1) Load the Assembler and select pass 1.
2) Position the Command Equate Tape in the reader.
3) Press start. The assembler will come to a halt after reading the Command Equate Tape.
4) Position the source tape for the program in the reader.
5) Press continue. The assembler will now complete pass 1.
6) Run pass 2 and pass 3 as usual, using only the source tape for the program. The Command Equate Tape need not be used during these two passes.

Steps 1-6 apply to each separate assembly of a user program. When the program(s) are assembled and are to be loaded, the user must be sure that $SFI is loaded before anything else, because any extended functions being used (including user-generated functions) overlay portions of FTBLE in $SFI.

With the object tapes for $SFI, the conversion routines, and the extended routines, as delivered, it is possible to create one of the four memory maps depicted below. The user may generate his own system suiting his needs by following the procedures outlined in the next section.

POSSIBLE MEMORY MAPS

WITH OBJECTS AS DELIVERED



6.2 User Generated Systems:

6.2.1 General

The source tapes supplied for the basic package, conversion routines, and extended package are organized so that the components of any Floating Point System the user may select for his problem will assemble such that the object tape generated will load as high as possible in memory. Since any configuration of routines which could comprise a Floating Point System must contain $SFI as a component, the procedure outlined for generating the system assumes that $SFI is located at the highest possible place in memory, and all other (if any) components will be located at successively lower locations.

6.3 Assembling $SFI:

Should the user desire to have $SFI load at some other place in core, the package must be re-assembled. Since $SFI will be assembled so that it occupies the highest location possible, the user need only supply that address. This is done by creating a separate source tape which has the following two assembly instructions:

```
$$END = XXXXX
EOT
```

where XXXXX is the last address that the user wishes $SFI to occupy.

To assemble $SFI, one follows the usual assembly procedure, except that the short source tape constructed above must be read in before the $SFI source tape at each pass.

Note: Do not use the Command Equate Tape for this assembly.

After completion of this assembly, the "$SFI=" statement on the Command Equate Tape should be updated (using the Source Text Editor) so that it states the new location at which $SFI begins. This location may be read off the symbol table printed out at the beginning of pass 3 of the assembly of $SFI.

6.4 Generating a Floating Point System:

If a configuration not obtainable with the object tapes supplied is desired, the user may, by editing the source tapes supplied (using the Source Text Editor), create a new source tape consisting of the routines he needs for his purposes.

Since $SFI is the integral component of any floating point configuration, it is assumed here that either the object tape supplied for $SFI or a relocated version assembled as in 6.3 (and the associated revised Command Equate Tape) will be used.

The source tapes for the conversion and extended routines are arranged so that there is one routine per block on the tape starting with the second block. (The first block contains comments comprising an index to the rest of the tape.) This makes it easy to extract the routines desired using the Source Text Editor. The routines on the extended package source tape are completely independent. However, the conversion source tape consists of three routines - the first (physically on the tape) of which is shared by both @SFC and @SCF. This means if the user desires a conversion routine in only one direction, he must

also include the shared routine (@FXC). The routines @SCF and @SFC are independent of each other.

The procedure for extracting the desired routines from these two tapes is outlined below:

1) Load and start the Source Text Editor

2) Punch a single block at the beginning of the source tape being created which says:

$$END = $SFI

($SFI-1 will be the last location into which the resulting object tape will load.)

3) If either or both of the conversion routines are desired, position the conversion routine source tape in the reader and skip the first (index) block. Copy the second block (@FXC), which is the shared routine; then copy the block(s) containing the desired conversion routine(s). If neither conversion is required, this step may be skipped entirely.

4) If one or more of the Extended Package routines is required, position the source tape for the extended routines in the reader and skip the first (index) block. Then copy the blocks corresponding to the routines required. If no extended package routines are required, this step may be skipped entirely.

5) Punch a final block consisting of the single statement:

END

6) Load the Basic Assembler, select pass 1.

7) Position the Command Equate Tape which corresponds to the object of $SFI to be used in the reader.

8) Press start. The assembler will come to a halt after reading the Command Equate Tape.

9) Position the source tape generated in steps 1-5 in the reader.

10) Press continue. The assembler will now complete pass 1.

11) Run pass 2 and pass 3 as usual using only the source tape created in steps 1-5. The Command Equate Tape need not be used for these two passes. (Even if no assembly listing is required, pass 3 should at least be started so that the Symbol Table is obtained.

12) Check the Symbol Table entries for @FCST, @SFC and @SCF against the values appearing on the Command Equate Tape used for the assembly and, if they are not the same, create a new Command Equate Tape which has the values for @FCST, @SFC, and @SCF as they appear in the Symbol Table.

13) Assemble any user programs using this new Command Equate Tape as explained in 6.1.

Note: 1) The lowest location occupied by the Floating Point System as created above will be the value appearing for $$END on the Symbol Table printed as a result of step 11) above.

2) The object tape created by this procedure must be loaded after $SFI as explained in 6.1.

APPENDIX A

Command Summary - Basic

Definitions:

Y ~ address of floating operand

~ address of location containing address - 1 of floating operand

~ address of another floating command

~ address of index value

[D] ~ optional selection of deferred addressing

I ~ index value of source or destination at address Y

A ~ pseudo-accumulator (FAC)

X ~ pseudo-index register

F ~ floating value of source or destination at effective address formed from Y.

Code (octal)	Basic Commands	Operation	Flags	Registers
00	FEXT	exit	none	none
01 [101]	FLDA [D] Y	F → A	none	FAC,FTM
02 [102]	FSTA [D] Y	A → F	FXFLG	FAC
03 [103]	FADD [D] Y	A+F → A	FXFLG	FAC,FTM
04 [104]	FSUB [D] Y	A-F → A	FXFLG	FAC,FTM
05 [105]	FMPY [D] Y	A*F → A	FXFLG	FAC,FTM
06 [106]	FDIV [D] Y	A/F → A	FXFLG, FDFLG	FAC,FTM
07 [107]	FADM [D] Y	A+|F| → A	FXFLG	FAC,FTM
10 [110]	FSBM [D] Y	A-|F| → A	FXFLG	FAC,FTM
14	FABS	|A| → A	none	FAC
15	FASQ	A^2 → A	FXFLG	FAC,FTM
16	FNOR	normalized A → A	FXFLG	FAC
17	FNEG	-A → A	none	FAC
20	FJMP Y	jump to Y	none	none
21	FJAP Y	jump to Y if A ≥ 0	none	none
22	FJAZ Y	jump to Y if A = 0	none	none

Code (octal)	Basic Commands	Operation	Flags	Registers
23	FJAN Y	jump to Y if A<0	none	none
24	FJEV Y	jump to Y if FXFLG set$\neq 0$	FXFLG (set to zero)	none
25	FJDC Y	jump to Y if FDFLG set$\neq 0$	FDFLG (set to zero)	none
26	FJIX Y	X+1$\rightarrow$X, jump to Y if X$\neq 0$	none	FINDX
27	FLDX Y	I$\rightarrow$X	none	FINDX
30	FSTX Y	X$\rightarrow$I	none	none

Command Summary - Extended Functions

Code (octal)	Extended Command	Operation	Flags	Registers
31	FSIN	SIN (FAC)$\rightarrow$FAC	FXFLG	FAC,FTM
32	FCOS	COS (FAC)$\rightarrow$FAC	FXFLG	FAC,FTM
33	FATN	TAN^{-1} (FAC)$\rightarrow$FAC	FXFLG,FDFLG	FAC,FTM
34	FLNE	LOG_e ($\lvert$FAC$\rvert$)$\rightarrow$FAC	FXFLG(*)	FAC,FTM
35	FEXP	$e^{FAC}\rightarrow$FAC	FXFLG,FDFLG(1)	FAC,FTM
36	FSQT	$\sqrt{\lvert FAC \rvert}\rightarrow$FAC	none(+)	FAC

(*) If input argument is negative, FNLNF internal to the FPLNE routine will be set non-zero (see write-up).

(+) If input argument is negative, FSFLG internal to the FPSQT routine will be set non-zero (see write-up).

(1) If input argument is too large, FEXOF internal to the FPEXP routine will be set non-zero (see write-up).

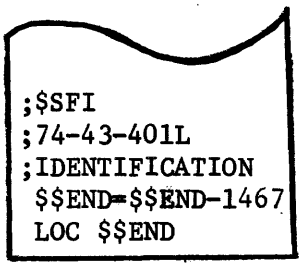
```
001                 ;$FCQ - COMMENTED
002                 ;74-43-402L
003                 ;COMMAND EQUATE TAPE
004                 $SFI=6120           ; ENTRY, INTERPRETER
005                 @FCST=5705          ; CHARACTER SET TABLE
006                 @SFC=5377           ; ENTRY, FLTNG. TO CHAR.
007                 @SCF=4771           ; ENTRY, CHAR. TO FLTNG.
008                 FGET=$SFI+2         ; ENTRY, EXTERNAL RETURN
009                 FMASK=FGET+12       ; MASK FOR DEFERRED MODE
010                 FPUNT=FMASK+13      ; ILLEGAL COMMAND ROUTINE
011                 FTBLE=FPUNT+14      ; TABLE OF ENTRIES
012                 FTCMP=FTBLE+111     ; ENTRY, FTM NEGATION
013                 FTCMA=FTCMP+4       ; ENTRY, -(AX,AY) TO FTM
014                 FTSAX=FTCMA+5       ; ENTRY, (AX,AY) TO FTM
015                 FTMHI=FTSAX+1       ; HIGH ORDER FTM
016                 FTMLO=FTMHI+2       ; LOW ORDER FTM
017                 FACMP=FTMLO+2       ; ENTRY, FAC NEGATION
018                 FACMA=FACMP+4       ; ENTRY, -(AX,AY) TO FAC
019                 FASAX=FACMA+5       ; ENTRY, (AX,AY) TO FAC
020                 FACHI=FASAX+1       ; HIGH ORDER FAC
021                 FACLO=FACHI+2       ; LOW ORDER FAC
022                 FNORM=FACLO+2       ; ENTRY, NORMALIZATION
023                 FXFLG=FNORM+3       ; EXPONENT OVERFLOW FLAG
024                 FACXP=FXFLG+50      ; FAC EXPONENT
025                 FINDX=FACXP+20      ; PSEUDO-INDEX
026                 FDAD=FINDX+5        ; ENTRY, DBLE. PREC. ADD
027                 FDMPY=FDAD+31       ; ENTRY, DBLE. PREC. MULT.
028                 FDDIV=FDMPY+143     ; ENTRY, DBLE. PREC. DIV.
029                 FDFLG=FDDIV+3       ; DIVIDE CHECK FLAG
030                 FSDVD=FDFLG+121     ; ENTRY, SNGL. PREC. DIV.
031                 FLODV=FSDVD+10      ; L. O. DIVIDEND FOR ABOVE
032                 FETCH=FLODV+27      ; ENTRY, FLTNG ARG TO FTM
033                 FSPLT=FETCH+16      ; SPLIT ARG. TO FAC.
034                 FTMXP=FSPLT+6       ; FTM EXPONENT
035                 FARGD=FTMXP+24      ; ENTRY, IMMED FETCH TO AX
036                 FPUSH=FARGD+1       ; PUSH DOWN POINTER
037                 FARSN=FPUSH+2       ; ENTRY, ARITH RGHT SHFT
038                 FARSC=FARSN+6       ; SHIFT COUNTER FOR ABOVE
039                 FCMAX=FARSC+7       ; ENTRY, MAX. NO. TO FAC
040                 F0FAC=FCMAX+21      ; ENTRY, ZERO TO FAC
041                 FPSTA=F0FAC+32      ; STORE FAC
042                 FINAC=FPSTA+34      ; PACK FAC
043                 FLIST=FINAC+373     ; PUSH DOWN LIST
044                 FEXT=0              ; EXIT COMMAND
045                 FLDA=1              ; LOAD FAC COMMAND
046                 FLDAD=101           ; LOAD FAC DEFERRED
047                 FSTA=2              ; STORE FAC
048                 FSTAD=102           ; STORE FAC DEFERRED
049                 FADD=3              ; FLOATING ADD
050                 FADDD=103           ; FLOATING ADD DEFERRED
051                 FSUB=4              ; FLOATING SUBTRACT
052                 FSUBD=104           ; FLOATING SUB. DEFERRED
053                 FMPY=5              ; FLOATING MULTIPLY
054                 FMPYD=105           ; FLOATING MULT. DEFERRED
055                 FDIV=6              ; FLOATING DIVIDE
056                 FDIVD=106           ; FLOATING DIVIDE DEFERRED
057                 FADM=7              ; FLOATING ADD MAGNITUDE
058                 FADMD=107           ; FLTNG ADD MAG DEFERRED
059                 FSBM=10             ; FLOATING SUB. MAGNITUDE
```

```
060             FSBMD=110       ; FLTNG SUB MAG DEFERRED
061             FTRN=11         ; TRACE ON
062             FTRF=12         ; TRACE OFF
063             FSET=13         ; SET ERROR TRAP
064             FABS=14         ; ABSOLUTE MAGNITUDE
065             FSQR=15         ; SQUARE
066             FNOR=16         ; NORMALIZE
067             FNEG=17         ; NEGATE
068             FJMP=20         ; UNCONDITIONAL JUMP
069             FJAP=21         ; JUMP IF FAC > OR = 0
070             FJAZ=22         ; JUMP IF FAC = 0
071             FJAN=23         ; JUMP IF FAC < 0
072             FJEV=23         ; JUMP IF FXFLG NOT 0
073             FJDC=25         ; JUMP IF FDFLG NOT 0
074             FJIX=26         ; BUMP FINDX, JMP IF NOT 0
075             FLDX=27         ; LOAD PSEUDO-INDEX
076             FSTX=30         ; STORE PSEUDO-INDEX
077             FSIN=31         ; SINE
078             FCOS=32         ; COSINE
079             FATN=33         ; ARCTANGENT
080             FLNE=34         ; NATURAL LOGARITHM
081             FEXP=35         ; EXPONENTIAL
082             FSQT=36         ; SQUARE ROOT
083                     EOT     ; ASSEMBLER HALT
```

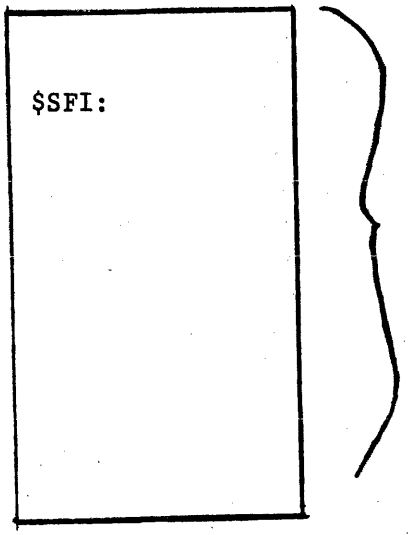
APPENDIX C

Source Tape Organization

NAME: $SFI

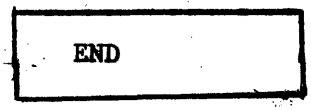


block mark



comprises several blocks

block mark



block mark

NAME; @FXV

```
;@FXV
;74-43-410L
;IDENTIFICATION
;INDEX (BLOCK NO.
 VS. ROUTINE)
;;...
```

index (block 0)

block mark

```
;@FCST
;74-43-411L
;IDENTIFICATION
 $$END=$$END-213
 LOC $$END
 @FCST=.
```

common module (block 1)

block mark

```
;@SFC
;74-43-412L
;IDENTIFICATION
 $$END=$$END-306
 LOC $$END
 @SFC=.
```

floating to character (block 2)

block mark

```
;SCF
;74-43-413L
;IDENTIFICATION
 $$END=$$END-406
 LOC $$END
 @SCF=.
```

character to floating (block 3)

block mark

```
   END
```

block mark

NAME: $FEF

```
;$FEF
;74-43-420L
;IDENTIFICATION
;INDEX (BLOCK NO.
;VS. ROUTINE)
;...
```

index (block 0)

block mark

```
;FPSIN
;74-43-421L
;IDENTIFICATION
 $$END=$$END-173
 LOC $$END
 FPSIN:
```

Sine/Cosine (block 1)

block mark

```
;FPATN
;74-43-422L
 $$END=$$END-146
 LOC $$END
 FPATN:
```

arctangent (block 2)

block mark

```
;FPLNE
;74-43-423L
;IDENTIFICATION
 $$END=$$END-164
 LOC $$END
 FPLNE:
```

natural logarithm (block 3)

block mark

```
;FPEXP
;74-43-424L
;IDENTIFICATION
 $$END=$$END-201
 LOC $$END
 FPEXP:
```

exponential (block 4)

block mark

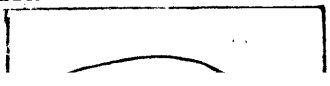
```
;FPSQT
;74-43-425L
;IDENTIFICATION
 $$END=$$END-127
 LOC $$END
 FPSQT:
```

square root (block 5)

block mark

```
   END
```

block mark



APPENDIX D

%FCG - Floating Point Constant Generator

%FCG - Floating Point Constant Generator

%FCG is a utility routine which is provided should the user wish to use floating point constants whose octal equivalences are unknown. With %FCG, the user can type in a floating point decimal number and receive the equivalent internal floating point representation.

%FCG occupies locations 0-2660 inclusive.

Operating Instructions

1. Load %FCG by means of %ALD.
2. Turn teletype on-line.
3. Set SC=0.
4. Press START.
5. %FCG responds with a carriage return, line feed.
6. Type a string of up to 13_{10} characters terminated with an equal sign (=). The character string should be in the format described in section 3.3, where the delimiter is an = rather than a carriage return. Typing a back arrow at any point causes the first previous non-back arrow to be ignored. Typing rubout at any point causes %FCG to type a carriage return, line feed, question mark (?) and returns to step 5. Typing more than 13 characters before typing an equal sign has the same effect as typing rubout.
7. When the user terminates the character string with the equal sign, %FCG responds by typing the 2 word floating point equivalent (in octal) and returns to step 5.
8. If the character string did not conform to the format specified in section 3.3, the message SCAN ERROR is typed and %FCG returns to step 5.

9. If the character string resulted in a number whose magnitude was outside the range 1.469369E-39 to 1.701411E+38 or if the character string contained more than 10_{10} mantissa digits, the message ANSWER OUT OF RANGE is typed and %FCG returns to step 5.

APPENDIX E

FPSET - Error Trap Routine

Introduction:

A series of floating point calculations on an unknown data base can generate errors, such as results which exceed the capacity of the machine or dividing by 0, etc. In order to facilitate the localization of the occurrence of such errors, FPSET is provided and serves as an error trap routine. When an error specified by the user is detected, FPSET will interrupt the operation of the interpreter and give control to a user supplied error routine. FPSET supplies the user error routine with the following information, allowing the user to pinpoint the step in his calculations at which the error occurred:

AX = recursion level at which the command at the address in AY was executed.

AY = address of command executed immediately previous to detecting the error.

TRP = error number indicating which flag in the user supplied error list was set non-zero. (TRP = position of address of error flag in user supplied table (see usage)).

The recursive capability of the interpreter somewhat complicates certain usages of FPSET and, for this reason, three modes of operation of FPSET are allowed: "On", "Off", and "Partially On". The latter mode allows FPSET to keep track of commands and recursion levels without examining any error flags. The utility of this mode is described in the examples at the end of this appendix.

Usage:

FPSET is controlled by the use of the FSET command in the sequence of floating commands being executed by the interpreter. There are three modes of operation of FPSET: 1) ON, 2) OFF, and 3) PARTIALLY ON.

1) To turn FPSET ON, the command is

```
FSET A
```

where A is the address of a table with the following format:

```
A:      WRD     ERR     ;USER ERROR ROUTINE ENTRY
        WRD     FLG1    ;ADDRESSES OF SYSTEM
        WRD     FLG2    ;FLAGS TO BE CHECKED...
         .
         .
         .
        WRD      -1     ;END OF TABLE SIGNAL
```

When this FSET command is encountered with a positive non-zero value for A, FPSET will examine the state of every flag listed in the table at address A after every command executed by the interpreter from the point of the FSET A command onward. Whenever a flag whose address is in the user list has become non-zero (indicating an error), FPSET zeros the flag, and then gives control to the user error routine at the address specified in the first word of the table at A. The information supplied to the user error routine is as stated in the introduction.

The user error routine may use $SFI, but any additional errors which might occur will not be checked by FPSET, and any FSET commands in the command sequence will be ignored. If the user wishes to call $SFI in his error routine, it is up to him to save and restore the states of the interpreter system flags and the floating accumulator (FAC) before and after such $SFI use.

2) To turn FPSET "off", the command is

FSET 0

This completely disconnects FPSET from the interpreter.

3) To turn FPSET "partially on", the command is

FSET N

where N is any negative number.

In this mode, FPSET will keep track of the current command address and recursion level but will not examine any flags. If FPSET is at some later time turned "on" and discovers a flag set non-zero, the level and the command address will be correct within certain limitations (see Notes).

This mode is useful when the user does not wish to enter his error routine for errors which occur during execution of a section of his command sequence. For example, the command sequence may contain an FJEV or similar test for conditions known to the user, and with FPSET "on", these conditions could be altered (cleared) if the corresponding flags are in the user error list at A. In this case, an FSET N (where $N < 0$), issued before entering this section, and an FSET A ($A > 0$, A=address of table), issued after completion of this section will allow FPSET to retain the necessary information should other errors occur and allow the section itself to operate properly.

User Error Routine:

Basically, the user error routine may do anything. However, the user must remember that his error routine is considered as an extension of the interpreter. At the completion of the error routine, control should be given back to the interpreter via a JU FGET or similar return.

Register AX is used as an argument upon return (via JU FGET) and can turn

FPSET "on", "off", or "partially on" according to AX$>$0, AX=0, or AX$<$0 respectively. If AX$>$0, it must be the address of an error table as described above (it need not necessarily be the same one as before).

Notes:

1) It is generally the case that an error flag is set by the command immediately preceding the detection of the flag non-zero. In the case where FPSET was not "on" at that moment, but was turned on later and found the flag non-zero, FPSET will report that it does not know which command caused the error by giving an AY value which points to the FSET "on" command or by AY = -1. The difference in meaning of the two AY values is as follows:

   a) AY = address of FSET "on" command if the flag was non-zero at the time the FSET "on" command was encountered.

   b) AY = -1 if all flags were zero when the FSET "on" was encountered, but a flag was set non-zero later at a point which indicated that the command which caused the error was at a recursion level one less than the level at which the error was detected. This situation is avoided if FPSET is partially on throughout until it is turned on.

2) If the user wishes to restart his entire program or in any other way wishes to use the interpreter without reloading it, he should make sure that FGET+1 is initialized to FARGD and FPUSH is initialized to FLIST-1.

3) A FEXT command does not affect the mode of operation of FPSET, i.e. upon re-entering $SFI, FPSET will operate as per the last FSET command encountered before the FEXT.

Operating Instructions:

1. If a user generated floating point system is being used (as per section 6.4), the FPSET program needs to be reassembled. To do this, make a short source tape with the following two instructions:

```
$$END = XXXXX

EOT
```

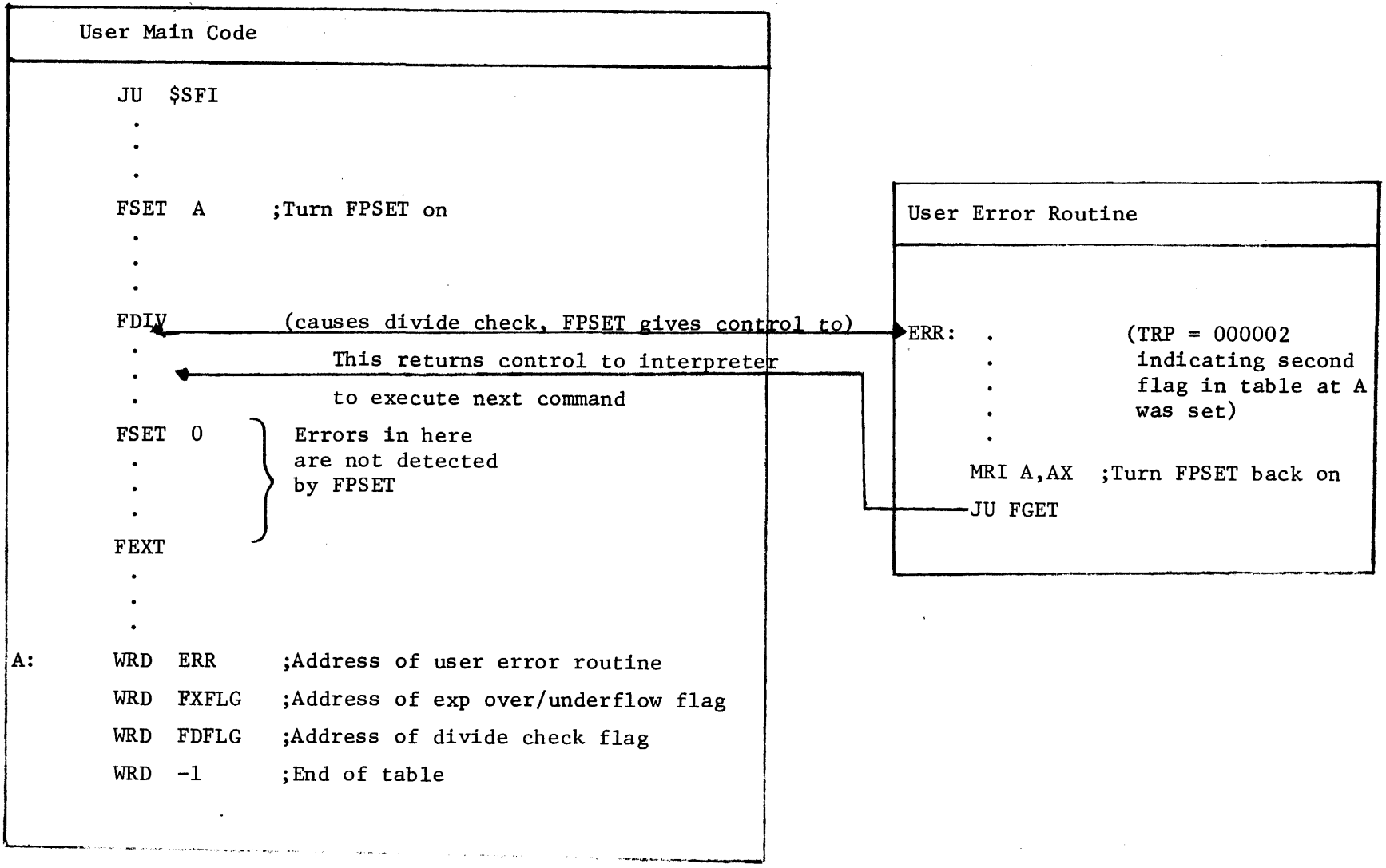
where XXXXX is the lowest location occupied by the user generated system. Then assemble FPSET as explained in section 6.1 of this document, making sure that the short source tape above is read in before the FPSET source at the beginning of each pass (steps 4 & 6 of 6.1) and the Command Equate Tape is read in at the beginning of Pass 1.

2. Load FPSET after $SFI.

3. Start user program which has FSET commands in the usual way.

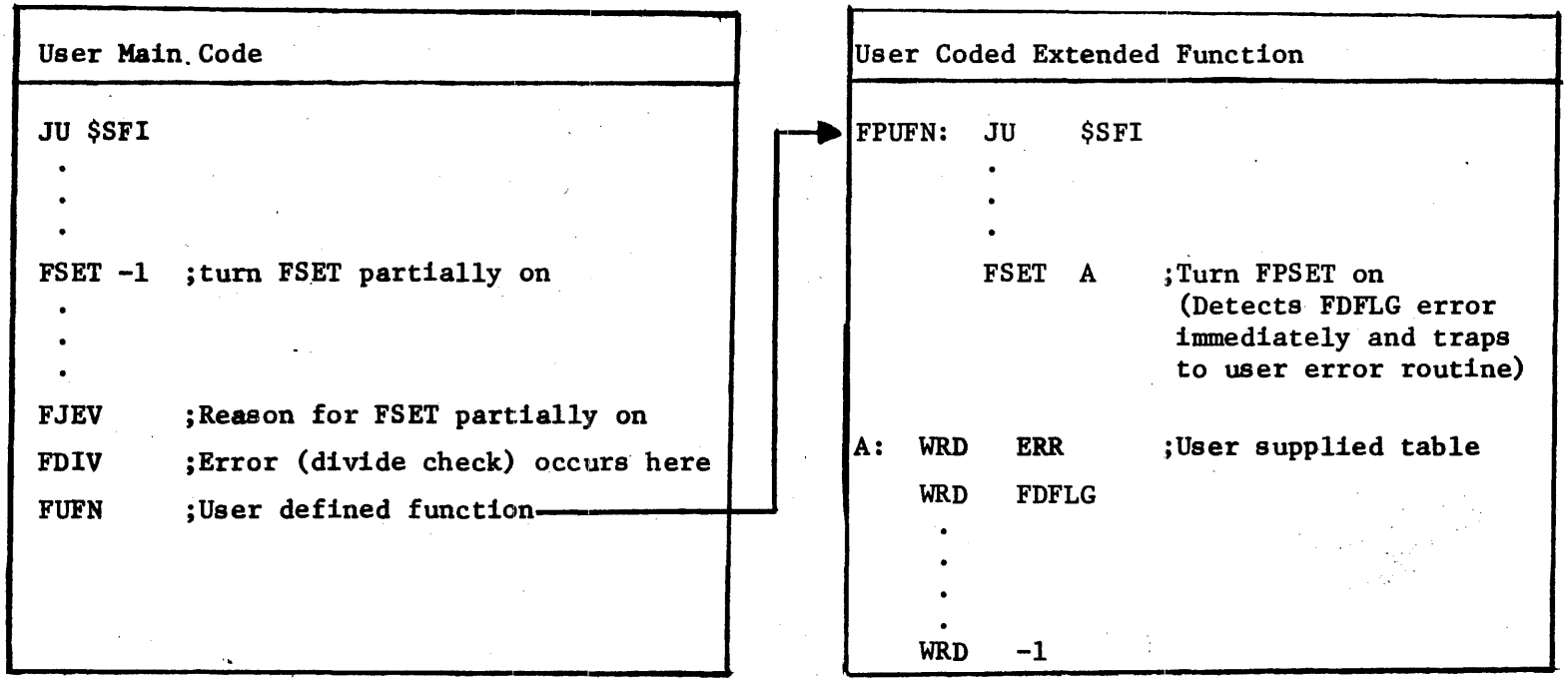
Examples

1. Typical usage of FSET "on" and "off"



Example

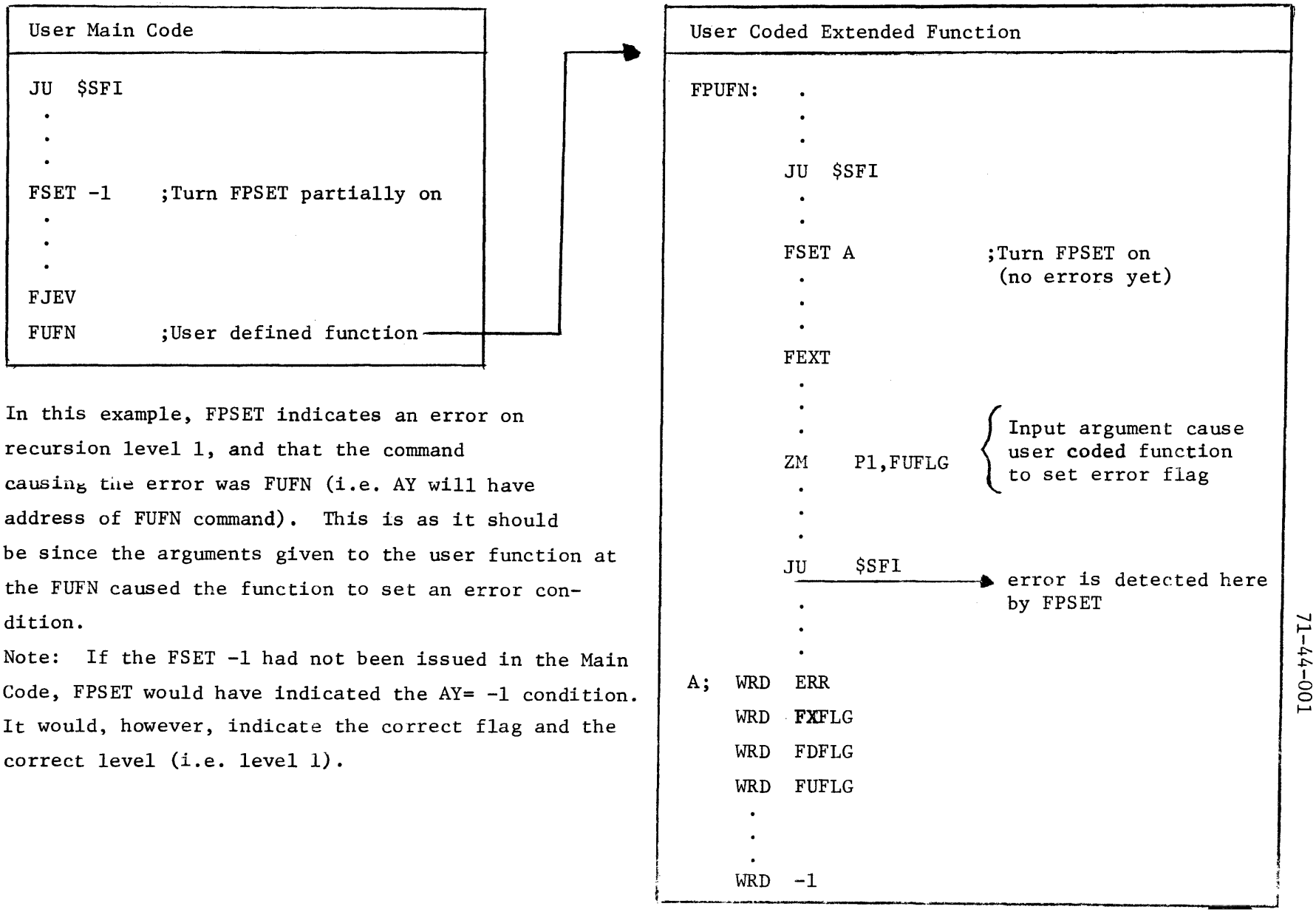
2. Use of FSET "Partially On"



In this example, because FPSET was only partially on when the error actually occurred, the error trap will indicate that the erring command was FSET. It will, however, indicate that the divide check flag was on (TRP = 000001 since FDFLG is first in table A) and that the error occurred in recursion level 2 since FPSET was turned on in the user function.

Examples

3. Another usage of FPSET "Partially On"



In this example, FPSET indicates an error on recursion level 1, and that the command causing the error was FUFN (i.e. AY will have address of FUFN command). This is as it should be since the arguments given to the user function at the FUFN caused the function to set an error condition.

Note: If the FSET -1 had not been issued in the Main Code, FPSET would have indicated the AY= -1 condition. It would, however, indicate the correct flag and the correct level (i.e. level 1).

APPENDIX F

Trace Routine

The floating point trace is a debugging aid which prints the value of pertinent variables in $SFI and the user's program before the execution of each floating point pseudo command. The variables printed are:

A. current level of $SFI

B. address of the instruction to be executed

C. code for the instruction to be executed

D. FINDX (floating point index)

E. FDFLG (divide check flag)

F. FXFLG (exponent overflow/underflow flag)

G. FAC (floating point pseudo accumulator)

H. effective address of argument, if any

I. value of argument, if any

The user specifies which of the variables are to be printed and the maximum level for which he wants the information printed. This is done through the floating point pseudo commands FTRN and FTRF.

To turn the trace on and specify which of the nine variables are to be printed, the pseudo command is:

FTRN X

where bits 0-8 of the integer X correspond to the variables A through I above. For each bit that is on (=1) the corresponding variable

will be printed before the execution of each floating point command. The FTRN command sets maximum recursion level to be traced to 7, turns the trace on and prints a heading (A-I), telling which variables are to be printed. The "trace on" causes the specified variables to be printed on one line before each instruction is executed.

The printed value of variables H (argument effective address) and I (argument) need further explanation. If the command to be executed has no argument, columns H and I will be blank. If the argument is floating point, I is printed as a floating point decimal number, otherwise it is octal. If the command is FTRN, FTRF or a JUMP command, then H is the address+1 of the command and I is the contents of H. In the case of the commands FLDX Y, or FSTX Y, H is the address Y and I is the contents of Y. For user coded extended functions, columns H and I will be blank unless the function has a floating point argument.

To turn the trace off beyond a certain level, the pseudo command is:

FTRF X

where the integer value X specifies the maximum recursion level (1-7) for which the specified variables are to be printed. If X is less than or equal to 0, the trace is disabled and no variables will be printed from then on until another FTRN X command is executed.

Notes:

1) When the trace has been turned on, certain locations in $SFI are changed. $SFI is restored to its original state only after the trace is completely disabled by an FTRF 0 command. Therefore, to restart the user program or use $SFI without reloading when the trace has been on, the user should make sure that:

a) FGET+1 is initialized to FARGD

b) FSPLT is initialized to 11 0000 06

c) FSPLT+1 is initialized to FTMHI

d) FPSTA+1 is initialized to FARGD

2) The trace program cannot run at the same time as FPSET, and so should be assembled over the FPSET program. Therefore, if the trace is on and an FSET command is encountered, the FSET command is considered illegal. That is, the interpreter halts with the address of the illegal command in the MB register on the front panel. (See section 2.2.5).

Operating Instructions:

1) If a user generated floating point system is being used (see section 6.4), the trace program (FPTRC) should be reassembled. Since FPSET and FPTRC are mutually exclusive, FPTRC is assembled over the FPSET program. To do this, make a short source tape with the following two instructions:

```
$$END = XXXXX
EOT
```

where XXXXX is the lowest location occupied by the user generated system (excluding FPSET). Then assemble FPTRC as explained in 6.1. Make sure that the short source tape above is read in before the FPTRC source at the beginning of each pass (steps 4 & 6 of 6.1). Read the Command Equate Tape in at the beginning of Pass 1.

2) Load FPTRC after $SFI.

3) Load and start user program with FTRN and FTRF commands in the usual way.

```
001                         ;EXAMPLE 1
002                         ;THIS EXAMPLE USES TWO FTRN COMMANDS.
003                         ;THE 1ST FTRN SELECTS VARIABLES A,B,C,D,F,G,H,I
004                         ;TO BE PRINTED.  THIS IS IN EFFECT UNTIL THE
005                         ;2ND FTRN IS EXECUTED.  THE SELECTED VARIABLES
006                         ;ARE THEN CHANGED TO A,B,C,E,F,G,H.  THE
007                         ;FTRF 0 COMPLETELY DISABLES THE TRACE.
008 00000 00 0100 03              JU    $SFI
    00001 006120
009 00002 00 0000 11              FTRN  757           ;PRINT A,B,C,D,F,G,H,I
    00003 000757
010 00004 00 0000 27              FLDX  W
    00005 000032
011 00006 00 0000 01              FLDA  X             ;Y=X*Y
    00007 000033
012 00010 00 0000 05              FMPY  Y
    00011 000035
013 00012 00 0000 02              FSTA  Y
    00013 000035
014 00014 00 0000 26              FJIX  .-6           ;DONE LOOP 3 TIMES?
    00015 000006
015 00016 00 0000 11              FTRN  367           ;PRINT A,B,C,E,F,G,H
    00017 000367
016 00020 00 0000 01              FLDA  X
    00021 000033
017 00022 00 0000 06              FDIV  Y
    00023 000035
018 00024 00 0000 02              FSTA  Z
    00025 000037
019 00026 00 0000 12              FTRF  0             ;TURN TRACE OFF
    00027 000000
020 00030 00 0000 00              FEXT
021 00031 02 0100 00              FOM   HLT
022 00032 177775          W:      WRD   -3            ;LOOP COUNT
023 00033 050000          X:      WRD   50000,203     ;5.0
    00034 000203
024 00035 040000          Y:      WRD   40000,202     ;2.0
    00036 000202
025 00037 000000          Z:      WRD   0,0           ;0.0
    00040 000000
026                               END
```

A	B	C	D	F	G	H	I
1	00004	00027	000000	0	+2.356227E-39	00032	177775
1	00006	00001	177775	0	+2.356227E-39	00033	+5.000000E+00
1	00010	00005	177775	0	+5.000000E+00	00035	+2.000000E+00
1	00012	00002	177775	0	+1.000000E+01	00035	+2.000000E+00
1	00014	00026	177775	0	+1.000000E+01	00015	000006
1	00006	00001	177776	0	+1.000000E+01	00033	+5.000000E+00
1	00010	00005	177776	0	+5.000000E+00	00035	+1.000000E+01
1	00012	00002	177776	0	+5.000000E+01	00035	+1.000000E+01
1	00014	00026	177776	0	+5.000000E+01	00015	000006
1	00006	00001	177777	0	+5.000000E+01	00033	+5.000000E+00
1	00010	00005	177777	0	+5.000000E+00	00035	+5.000000E+01
1	00012	00002	177777	0	+2.500000E+02	00035	+5.000000E+01
1	00014	00026	177777	0	+2.500000E+02	00015	000006
1	00016	00011	000000	0	+2.500000E+02	00017	000367

A	B	C	E	F	G	H
1	00020	00001	0	0	+2.500000E+02	00033
1	00022	00006	0	0	+5.000000E+00	00035
1	00024	00002	0	0	+2.000000E-02	00037
1	00026	00012	0	0	+2.000000E-02	00027

```
001                         ;EXAMPLE 2
002                         ;THIS EXAMPLE PRINTS ALL 9 VARIABLES
003                         ;FOR RECURSION LEVELS 1 AND 2.
004                         ;THE USER EXTENDED FUNCTION IS AT
005                         ;RECURSION LEVEL 2.
006 00000 00 0100 03              JU    $SFI
    00001 006120
007 00002 00 0000 11              FTRN  777           ;PRINT A,B,C,D,E,F,G,H,I
    00003 000777
008 00004 00 0000 12              FTRF  2             ;MAX TRACE LEVEL=2
    00005 000002
009 00006 00 0000 27              FLDX  X
    00007 000043
010 00010 00 0001 01              FLDAD X+1           ;FETCH ARG DEFERRED
    00011 000044
011 00012 00 0000 02              FSTA  Y
    00013 000046
012 00014 00 0000 37              FUFN  Y             ;USER EXTENDED FUNCTION
    00015 000046
013 00016 00 0000 26              FJIX  .-6           ;DONE?
    00017 000010
014 00020 00 0000 12              FTRF  0             ;YES, TRACE OFF
    00021 000000
015 00022 00 0000 00              FEXT
016 00023 02 0100 00              FOM   HLT
017 00024 00 0100 03 FPUFN:       JU    FARGD
    00025 007057
018 00026 11 0000 06              RM    AX,ARG+1
    00027 000035
019 00030 00 0100 03              JU    $SFI
    00031 006120
020 00032 00 0000 01              FLDA  Z             ;500.0/Y
    00033 000050
021 00034 00 0000 06 ARG:         FDIV  0
    00035 000000
022 00036 00 0001 02              FSTAD X+2
    00037 000045
023 00040 00 0000 00              FEXT
024 00041 00 0100 03              JU    FGET
    00042 006122
025 00043 177776           X:     WRD   -2            ;LOOP COUNT
026 00044 000077                  WRD   77            ;FETCH ADR, DEFERRED
027 00045 000077                  WRD   77            ;STORE ADR, DEFERRED
028 00046 000000           Y:     WRD   0,0           ;FUFN ARG.
    00047 000000
029 00050 076400           Z:     WRD   76400,211     ;500.0
    00051 000211
030                               LOC   100
031 00100 062000                  WRD   62000,206     ;50.0
    00101 000206
032 00102 050000                  WRD   50000,204     ;10.0
    00103 000204
033                         FUFN=37
034                               LOC   FTBLE+FUFN
035 06222 000024                  WRD   FPUFN
036                               END
```

A	B	C	D	E	F	G	H	I
1	00024	00012	000000	0	0	+2.356227E-39	00005	000002
1	00026	00027	000000	0	0	+2.356227E-39	00043	177776
1	00010	00101	177776	0	0	+2.356227E-39	00100	+5.000000E+01
1	00012	00002	177776	0	0	+5.000000E+01	00046	+6.000000E+00
1	00014	00037	177776	0	0	+5.000000E+01		
2	00032	00001	177776	0	0	+5.000000E+01	00050	+5.000000E+02
2	00034	00006	177776	0	0	+5.000000E+02	00046	+5.000000E+01
2	00036	00102	177776	0	0	+1.000000E+01	00100	+5.000000E+01
2	00040	00000	177776	0	0	+1.000000E+01		
1	00016	00026	177776	0	0	+1.000000E+01	00017	000010
1	00020	00101	177777	0	0	+1.000000E+01	00102	+1.000000E+01
1	00022	00002	177777	0	0	+1.000000E+01	00046	+5.000000E+01
1	00024	00037	177777	0	0	+1.000000E+01		
2	00032	00001	177777	0	0	+1.000000E+01	00050	+5.000000E+02
2	00034	00006	177777	0	0	+5.000000E+02	00046	+1.000000E+01
2	00036	00102	177777	0	0	+5.000000E+01	00102	+1.000000E+01
2	00040	00000	177777	0	0	+5.000000E+01		
1	00016	00026	177777	0	0	+5.000000E+01	00017	000010
1	00020	00012	000000	0	0	+5.000000E+01	00021	000000

```
001                     ;EXAMPLE 3
002                     ;THIS IS THE SAME AS EXAMPLE 2
003                     ;EXCEPT THE MAXIMUM RECURSION
004                     ;LEVEL PRINTED IS LEVEL 1 DUE
005                     ;TO THE FTRF 1 COMMAND.
006 00000 00 0100 03            JU    $SFI
    00001 006120
007 00002 00 0000 11            FTRN  777           ;PRINT A,B,C,D,E,F,G,H,I
    00003 000777
008 00004 00 0000 12            FTRF  1             ;MAX TRACE LEVEL=1
    00005 000001
009 00006 00 0000 27            FLDX  X
    00007 000043
010 00010 00 0001 01            FLDAD X+1           ;FETCH ARG DEFERRED
    00011 000044
011 00012 00 0000 02            FSTA  Y
    00013 000046
012 00014 00 0000 37            FUFN  Y             ;USER EXTENDED FUNCTION
    00015 000046
013 00016 00 0000 26            FJIX  .-6           ;DONE?
    00017 000010
014 00020 00 0000 12            FTRF  0             ;YES, TRACE OFF
    00021 000000
015 00022 00 0000 00            FEXT
016 00023 02 0100 00            FOM   HLT
017 00024 00 0100 03 FPUFN:     JU    FARGD
    00025 007057
018 00026 11 0000 06            RM    AX,ARG+1
    00027 000035
019 00030 00 0100 03            JU    $SFI
    00031 006120
020 00032 00 0000 01            FLDA  Z             ;500.0/Y
    00033 000050
021 00034 00 0000 06 ARG:       FDIV  0
    00035 000000
022 00036 00 0001 02            FSTAD X+2
    00037 000045
023 00040 00 0000 00            FEXT
024 00041 00 0100 03            JU    FGET
    00042 006122
025 00043 177776      X:        WRD   -2            ;LOOP COUNT
026 00044 000077                WRD   77            ;FETCH ADR, DEFERRED
027 00045 000077                WRD   77            ;STORE ADR, DEFERRED
028 00046 000000      Y:        WRD   0,0           ;FUFN ARG.
    00047 000000
029 00050 076400      Z:        WRD   76400,211     ;500.0
    00051 000211
030                             LOC   100
031 00100 062000                WRD   62000,206     ;50.0
    00101 000206
032 00102 050000                WRD   50000,204     ;10.0
    00103 000204
033                     FUFN=37
034                             LOC   FTBLE+FUFN
035 06222 000024                WRD   FPUFN
036                             END
```

A	B	C	D	E	F	G	H	I
1	00004	00012	000000	0	0	+2.356227E-39	00005	000001
1	00006	00027	000000	0	0	+2.356227E-39	00043	177776
1	00010	00101	177776	0	0	+2.356227E-39	00100	+5.000000E+01
1	00012	00002	177776	0	0	+5.000000E+01	00046	+0.000000E+00
1	00014	00037	177776	0	0	+5.000000E+01		
1	00016	00026	177776	0	0	+1.000000E+01	00017	000010
1	00020	00101	177777	0	0	+1.000000E+01	00102	+1.000000E+01
1	00022	00002	177777	0	0	+1.000000E+01	00046	+5.000000E+01
1	00024	00037	177777	0	0	+1.000000E+01		
1	00026	00026	177777	0	0	+5.000000E+01	00017	000010
1	00030	00012	000000	0	0	+5.000000E+01	00021	000002

GRI Computer Corporation

320 NEEDHAM STREET, NEWTON, MASSACHUSETTS 02164

TEL: (617) 969-0800