MOTOROLA MC6839 FLOATING POINT ROM

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

SECTION 1
INTRODUCTION

1.1 EARLY APPROACH TO MATHEMATICAL OPERATION

Since the earliest days of computers, it has been obvious that no computer was capable of doing all the desired mathematical operations in binary integer arithmetic. Some early implementations perform floating point operations as a long string of BCD characters. Although the accuracy of this approach and the ease of implementation make it a popular alternative, the speed is too slow for most applications. Even though the BCD approach is still used today in most BASIC systems and in most systems doing dollars and cents calculations, most scientific calculations use a binary floating point representation.

As binary floating point became widely used during the 1960s, each computer manufacturer created his own floating point representation. There was (and is) a wide variation in formats and accuracy which almost guarantees that a program executed on one computer will get different results if executed on another computer. The mini-computer manufacturers improved the representations somewhat; but each manufacturer still had a different format and different ways to represent and handle exceptions and errors.

Meanwhile, research has been completed which formulates an optimal binary floating point representation. Unfortunately, the existing manufacturers had far too much money invested in software and hardware to incur the costs of conversion to a new standard. Powerful microprocessors, on the other hand, were in their infancy and the floating point "experts" recognized the opportunity to standardize a floating point format for microprocessors. The IEEE appointed a committee to address the standard. This manual describes an implementation of a proposed IEEE standard, for the MC6809 Microprocessor (MPU), which is in a ROM. The information discussed in this manual is not a "restatement of the proposed IEEE Standard;" instead, it addresses those areas that are required by the proposed IEEE standard and those optional areas that are implemented for the MC6809. Specific details of the proposed IEEE standard can be found in *IEEE Proposed Standard for Binary Floating Point Arithmetic Draft 8.0* (referred to as the proposed IEEE standard in this manual) which is available from the IEEE.*

Much of the information, and many of the suggestions, for the proposed IEEE standard originated in a series of papers, published by Jerome Coonen at the University of California at Berkeley, the most recent of which is entitled "Specifications for a Proposed Standard for Floating Point Arithmetic" and appeared in the January 1980 issue of *Computer* magazine.

1.2 PROGRAMS-IN-ROM

From its inception, the MC6809 MPU was designed to support a concept of "ROMable" software by using an improved instruction set and addressing modes. One way, and to some extent the only way, to reduce the escalating cost of software was to supply "software on silicon." Since the original cost of developing the software can be amortized over a very large number of parts, the actual cost of the ROM part is low.

Shortly after completing the MC6809 MPU, Motorola selected floating point to become the first Motorola Standard Product ROM (SPR). Floating point was selected because it is standard software that can be used in many diverse systems. Furthermore, implementation of the proposed IEEE standard is sufficiently complex that many potential customers would not wish to develop the necessary expertise to write their own software; however, they would enjoy the advantages of its many benefits.

Hardware implementations of floating point are always much faster (and more expensive) than software implementations. However, the MC6839 Floating Point ROM substitutes increased functionality for speed. In addition, the MC6839 Floating Point ROM supports all precisions, modes, and operations required or suggested by the proposed IEEE standard.

1.3 MC6839 FLOATING POINT (FP) ROM

1.3.1 General

The MC6839 FP ROM provides floating point capability for the MC6809 and MC6809E MCUs. It implements the entire proposed IEEE standard providing a relatively simple, economical, and reliable solution to a wide variety of numerical applications. The MC6839 FP ROM provides three different formats, namely: single precision, double precision, and extended. Both the single and double precision formats provide results which are bit-for-bit reproducible across all proposed IEEE standard implementations. The extended format provides the extra precision needed for the intermediate results of long calculations, particularly the implementation of transcendental functions and interest calculations. All applications benefit from the extensive error checking and well-defined responses to exceptions, which are strengths of the proposed IEEE standard.

The MC6839 FP ROM takes full advantage of the advanced architectural features of the MC6809/MC6809E MPU. It is position independent and re-entrant, facilitating its use in real-time, multi-tasking systems.

*This proposed standard was published in the April 1981 issue of *Computer* magazine.

A brief summary of the MC6839 FP ROM is shown below:

- Totally Position Independent
  - Operates in any Contiguous 8K Block of Memory
- Re-Entrant
  - No Use of Absolute RAM
  - All Memory References are made Relative to the Stack Pointer
- Flexible User Interface
  - Operands are Passed to the FP Package by One of Two Methods
    1) Machine Registers are Used as Pointers to the Operands
    2) The Operands are Pushed onto the Hardware Stack
  - The Latter Method Facilitates the Use of the MC6839 FP ROM in High-Level Language Implementations
- Easy to Use Two/Three Address Architecture
  - The User Specifies Addresses of Operands and Result and Need Not be Concerned with any Internal Registers or Intermediate Results
- A Complete Implementation of the Proposed IEEE Standard
  - Includes All Precisions, Modes, and Operations Required or Suggested by the Standard
  - Includes the Following Operations:

    Add
    Subtract
    Multiply
    Divide
    Remainder
    Square Root
    Integer Part
    Absolute Value
    Negate
    Predicate Compares
    Condition Code Compares
    Convert Integer—Floating Point
    Convert Binary Floating Point—Decimal String

1.3.2 Pin Assignment

The MC6839 FP ROM is housed in one 24-pin 8K-by-8 mask programmable ROM: the MCM68364. It uses a single 5 V power supply and is available with access times of 250 or 350 ns. For electrical characteristics, refer to the Advance Information Sheet for the MC68A39 (1.5 MHz) and MC68B39 (2.0 MHz).

Figure 1-1 shows a pin assignment diagram of the MC6839 FP ROM and Figure 1-2 contains a block diagram of the device.

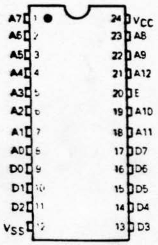


Figure 1-1. Pin Assignment Diagram

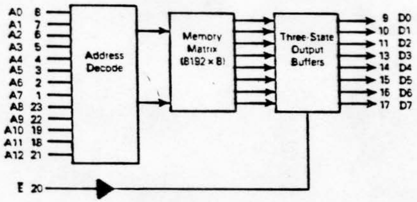


Figure 1-2. MC6839 FP ROM Block Diagram

SECTION 2
STANDARD FLOATING POINT FORMATS

2.1 INTRODUCTION

The MC6839 Floating Point ROM (also referred to as the floating point package in this manual) supports three precisions of floating point numbers: single, double, and extended. It supports normalized numbers plus four special types of numbers for each precision: zeros, infinities, NANs, and denormalized numbers. The following paragraphs describe how the numbers are represented in user memory for each precision. Also described are the formats used to represent binary integers and BCD strings.

2.2 NORMALIZED NUMBERS

Normalized numbers are floating point numbers that are not one of the special types. The bulk of the numbers in any real program will be normalized numbers. Three different formats are used with normalized numbers, namely: single precision format, double precision format, and extended format.

2.2.1 Single Precision Format

All single precision numbers are represented in a four byte string as shown below:

1	← 8 →	← 23 Bits →
s	Exponent	Significand

In single precision formats shown above, the exponent is biased by + 127. That is, an exponent of: 0 is 127, 2 is 129, and −2 is 125.* A normalized floating point number always has a 1 to the left of the binary point; this bit is not explicit in the memory formats. This saves one bit in memory which allows more precision with the same number of bits. In this specification, the fraction is referred to as a significand in order to indicate that it has an implied 1.0 added to the fraction. Hence, significands lie in the range

*A biased exponent makes floating point compare easier to implement since the exponent and significand can be considered as a long integer. Also, an unsigned integer compare can be used to calculate the condition codes rather than a floating subtract.

1.0<significand<2.0. S is the sign of the significand. The significand is stored in sign magnitude rather than twos complement form. The equation for the single precision format representation is:

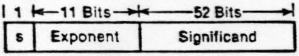
X	$= (-1)^2 \times 2^{(\text{exponent } -127)} \times (1.\text{significand})$.
s	= sign bit = bit string length of 1.
exponent	= biased exponent — bit string length of 8.
significand	= bit string length of 23 encoding the significant bits of the number that-low the binary point, yielding a 24 bit significand digit field for the number that always begins "1.___".

Examples:

$+1.0 = 1.0 \times 2^0 =$ \$3F, 80, 00, 00
$+3.0 = 1.5 \times 2^1 =$ \$40, 40, 00, 00
$-1.0 = -1.0 \times 2^0 =$ \$BF, 80, 00, 00
$+7.0 = 1.75 \times 2^2 =$ 140, E0, 00, 00
$+0.5 = 1.0 \times 2^{-1} =$ \$3F, 00, 00, 00

2.2.2 Double Precision Format

All double precision numbers are represented in an eight byte string as shown below:



In the double precision format shown above, the exponent is biased by +1023. Interpretation of the format is similar to the single precision format except the bias is +1023 instead of +127. The equation for the double precision format representation is:

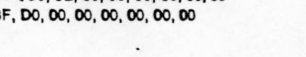
$X = (-1)^S \times s^{(\text{exponent } -1023)} \times (1.\text{significand})$

Examples:

$7.0 = 1.75 \times 2^2 =$ \$40, 1C, 00, 00, 00, 00, 00, 00
$16.0 = 1.0 \times 2^4 =$ \$40, 30, 00, 00, 00, 00, 00, 00
$30.0 = 1.875 \times 2^4 =$ \$40, 3E, 00, 00, 00, 00, 00, 00
$-30.0 = -1.875 \times 2^4 =$ \$C0, 3E, 00, 00, 00, 00, 00, 00
$0.25 = 1.0 \times 2^{-2} =$ \$3F, D0, 00, 00, 00, 00, 00, 00

2.2.3 Extended Format

Single and double precision formats should be used to represent the majority of floating point numbers in the user's system (e.g., storage of arrays). The extended format should only be used for intermediate calculations such as occur in the evaluation of a complex expression. In fact, extended format may never be required by most users; however, since it is required internally, it is optionally provided. Extended numbers are represented in a 10 byte string as shown below:



A notable difference between this format and the single and double precision formats is that the 1.0 is explicitly present in the significand and the exponent contains no bias and is in twos complement form. The equation for double extended is:

$$X = (-1)^{S} \times 2^{(\text{exponent})} \times \text{significand}$$

Where the significand contains the explicit 1.0.

Examples:

$30.0 = 1.875 \times 2^{4} = \$00, 04, F0, 00, 00, 00, 00, 00, 00, 00$

$0.5 = 1.0 \times 2^{-1} = \$7F, FF, 80, 00, 00, 00, 00, 00, 00, 00$

$-1.0 = -1.0 \times 2^{0} = \$80, 00, 80, 00, 00, 00, 00, 00, 00, 00$

$384.0 \times 1.5 \times 2^{8} = \$00\ 08, C0, 00, 00, 00, 00, 00, 00, 00$

2.3 SPECIAL VALUES (SINGLE AND DOUBLE MEMORY FORMAT)

No derivable floating point format can represent the infinite number of possible real numbers, so it is very useful if some special numbers are recognized by a floating point package. These numbers are: +0, −0, + infinity, − infinity, very small (almost zero) numbers, and in some cases unnormalized numbers. It is also convenient to have a special format to indicate that the contents of memory do not contain a valid floating point number. This "not a number" might occur if a variable is defined in a high level language (HLL) and is used before it is initialized with a value. The most positive and negative exponents of each format are reserved to represent these special values. How these special format numbers enter into calculations will be covered in the detailed description of each operation (Appendix C).

2.3.1 Zero

Zero is represented by a number with both a zero exponent and a zero significand. The sign is significant and differentiates between plus or minus zero.

s	0	0

2.3.2 Infinity

The infinities are represented by a number with the maximum exponent and a zero significand. The sign differentiates plus or minus infinity.

s	1111 1111	0

2.3.3 Small Numbers (Denormalized)

When a number is so small that its exponent is the smallest allowable normal biased value (1), and it is impossible to normalize the number without further decrementing the exponent, then the number becomes denormalized. The format for denormalized numbers has a zero exponent and a nonzero significand. Note that in this form the implicit bit is no longer one but is zero. The interpretation for denormalized numbers is:

Single: $X = (-1)^s \times 2^{-126} \times (0.\ \text{significand})$

Double: $X = (-1)^s \times 2^{-126} \times (0.\ \text{significand})$

Note that the exponent is always interpreted as 2^{-126} for single and 2^{-1022} for double instead of 2^{-127} and 2^{-1023} as might be expected. This is necessary since the only way to insure that the implicit bit becomes zero is to right shift the significand (divide by 2) and increment the exponent (multiply by 2). Thus the exponent ends up with the interpretation of 2^{-126} or 2^{-1022}.

The format for denormalized numbers is:

s	0	Non-Zero

Note that zero may be considered a special case of denormalized numbers where the number is so small that the significand has been reduced to zero. The concept of denormalized numbers has perhaps been the most controversial aspect of the proposed IEEE standard. However, the concept of allowing a number to "gently underflow" to zero seems intuitive and straightforward to most inexperienced users who do not have a built-in bias for some existing floating point representation.

Examples:

Single: $1.0 \times 2^{-128} = 0.25 \times 2^{-126} =$ 00, 20, 00, 00

Double: $1.0 \times 2^{-1025} = 0.125 \times 2^{-1022} =$ 00, 02, 00, 00, 00, 00, 00, 00

2.3.4 Not a Number (NAN)

The format for NANs has the largest allowable exponent, a nonzero significand, and an undefined sign. As an implementation feature (not IEEE required), the nonzero fraction and undefined sign are further defined as shown below:

d	1111.....1111	t	Operation Address	00...000

d: 0 = This NAN has never entered into an operation with another NAN.

1 = This NAN has entered into an operation with other NANs.

t :0 = This NAN will not necessarily cause an invalid operation trap when operated upon.

1 = This NAN will cause an invalid operation trap when operated upon (trapping NAN).

Operation address:

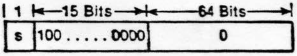
The 16 bits immediately to the right of the t bit contain the address of the instruction immediately following the call to the floating point package of the operation that caused the NAN to be created. If d (double NAN) is also set, the address is arbitrarily one of the addresses in the two or more participating NANs.

2.4 SPECIAL VALUES (EXTENDED FORMAT)

The special values discussed below are implemented using the extended format which was discussed earlier in this section. As explained before, numbers are represented in this format as a 10 byte string.

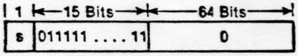
2.4.1 Zero

Zero is represented by a number with the smallest unbiased exponent and a zero significand:



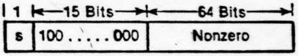
2.4.2 Infinity

Infinity has the maximum unbiased exponent and a zero significand:



2.4.3 Denormalized Numbers

Denormalized numbers have the smallest unbiased exponent and a nonzero significand:



The exponent of denormalized extended and internal numbers is -16384, and has the value:

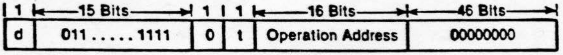
$(-1)^{s} \times 2^{-16383} \times 0.f$

Example:

$1.0 \times 2^{-16387} \times 0.125 \times 2^{-16384} \times$ 40, 00, 08, 00, 00, 00, 00, 00, 00, 00

2.4.4 NANs

These have the largest unbiased exponent and a nonzero significand. The operation addresses, "t" and "d", are implementation features and were defined in an earlier paragraph of this section.



The operation address always appears in the 16 bits immediately to the right of the t bit.

2.4.5 Unnormalized Numbers

Unnormalized numbers occur only in extended or internal format. Unnormalized numbers have an exponent which is greater than the minimum established for the extended format (i.e., they are not denormalized or normal zero); however, the explicit leading significand bit is a zero. If the significand is zero, this is an unnormalized zero. Even though unnormalized and denormalized numbers are handled similarly in most cases, they should not be confused. Denormalized numbers are numbers that are very small (have minimum exponent) and hence have lost some bits of the significance. Unnormalized numbers are not necessarily small (the exponent may be large or small) but the significand has lost some bits of significance, hence, the explicit bit and possibly some of the bits to the right of the explicit bit are zero.

s	> 100 . . . 000	0. Significand

Unnormalized numbers cannot be represented (thus, cannot represent a result) for single precision and double precision formats. Unnormalized numbers can only be created when denormalized numbers, in single precision or double precision formats, are converted to extended (or internal) formats.

Example:

0.0625×2^2 (unnormalized) = 00, 02, 08, 00, 00, 00, 00, 00, 00, 00

2.5 BCD STRINGS

A BCD string is the input to the BCD-to-FP operation and the output of the FP-to-BCD operation. All BCD strings are represented by a 26 byte string with the following format:

0	1 5	6	24	25 (Byte #)
se	4 Digit BCD Exponent	sf	19 Digit BCD Fraction	p

se = sign of the exponent. 00_{16} = plus, $0F_{16}$ = minus. (1 byte)
sf = sign of the fraction. 00_{16} = plus, $0F_{16}$ = minus. (1 byte)
p = number of fraction digits to the right of the decimal point. (1 byte)

All BCD digits are unpacked and right justified in each byte:

7	0
0000	0-9

The byte ordering of the fraction and exponent is consistent with all Motorola processors in that the most significant BCD digit is in the lowest memory address.

Examples:

$2.0 = 2.0 \times 10^0$ ($p = 0$)

```
      00                  [se = +]
      00, 00, 00, 00      [exponent = 0]
      00                  [sf = +]
      00, 00, 00, 00, 00  [fraction = 2]
      00, 00, 00, 00, 00
      00, 00, 00, 00, 00
      00, 00, 00, 02
      00                  [p = 0]
```

or $2.0 = 20{,}000 \times 10^{-4}$ ($p = 0$)

```
      0F                  [se = -]
      00, 00, 00, 04      [exponent = 4]
      00                  [sf = +]
      00, 00, 00, 00, 00  [fraction = 200000]
      00, 00, 00, 00, 00
      00, 00, 00, 00, 02
      00, 00, 00, 00
      00                  [p = 0]
```

(The above might be the output of an FP to BCD operation with k = 5.)

or $2.0 = 2.0 \times 10^0$ ($p = 10$)

```
      00                  [se = +]
      00, 00, 00, 00      [exponent = 0]
      00                  [sf = +]
      00, 00, 00, 00, 00  [fraction = 20000000000]
      00, 00, 00, 02, 00
      00, 00, 00, 00, 00
      00, 00, 00, 00
      0A                  [p = 10]
```

2.6 BINARY INTEGERS

Two sizes of binary integers are supported: short and double. Short integers are 16 bits long and double integers are 32 bits long. The byte ordering is consistent with all Motorola processors in that the most significant bits are in the lowest address.

SECTION 3
SUPPORTED OPERATIONS

3.1 INTRODUCTION

The supported operations are divided into two groups: those required by the proposed IEEE standard, and those implemented to support real data types for Motorola Pascal. A larger number of operations are required by the proposed standard to insure portability of floating point algorithms.

3.2 REQUIRED OPERATIONS

The operations required to support the proposed IEEE standard are shown in Table 3-1. The mnemonic column in Table 3-1 illustrates the suggested mnemonics although, at present, no Motorola assembler supports them. The opcodes are used when calling the MC6839 to differentiate the various functions. The method for calling is described in Section 6.

All routines shown in Table 3-1, except FMOV and the compares (FCMP, FTCMP, FPCMP, and FTPCMP), accept arguments of the same precision and generate a result containing the same precision.

Table 3-1. Required Operations to Support IEEE Standard

Opcode	Mnemonic	Operation
00	FADD	arg1 + arg2—result
02	FSUB	arg1 − arg2—result
04	FMUL	arg1 × arg2—result
06	FDIV	arg1/arg2—result
08	FREM	remainder (arg1/arg2)—result
8A	FCMP	arg1 − arg2, set condition codes
CC	FTCMP	arg1 − arg2, set condition codes, trap on unordered
8E	FPCMP	arg1 − arg2, affirm or disaffirm a predicate
D0	FTPCMP	arg1 − arg2, affirm or disaffirm a predicate, trap on unordered
9A	FMOV	move (or convert) arg2—result
12	FSQRT	square root arg2—result
14	FINT	integer part of arg2—result
16	FFIXS	floating arg2—short integer result
18	FFIXD	floating arg2—double integer result
24	FFLTS	short integer arg2—floating result
26	FFLTD	double integer arg2—floating result
1C	BINDEC	binary floating—decimal BCD string
22	DECBIN	decimal BCD string—binary floating

3.3 EXTRA OPERATIONS

In order to support Motorola Pascal, two other operations are supplied. They include:

Opcode	Mnemonic	Operation
1E	FAB	Absolute Value of arg2 — Result
20	FNEG	— arg2 — Result

3.4 ARCHITECTURE

All floating point operations are of the "two address" or "three address" variety; all the user need supply are the addresses of both the operand(s) and the result. The package looks for operands at the specified location(s) and delivers the result to the specified destination. For example,

```
  arg1          arg2          Result
<Source>  +  <Source>  –  <Destination>
```

The only permanent state information is contained in floating point control block (FPCB) which defines the modes of the package. This control block is much like the control blocks frequently used to define I/O or operating system operations. The FPCB is discussed in detail in Section 5.

SECTION 4
MODES OF OPERATION

4.1 INTRODUCTION

In addition to supporting a wide range of precisions and operations, the MC6839 Floating Point ROM supports all modes required or suggested by the proposed IEEE standard. These include: rounding modes, infinity closure modes, and exception handling modes. The various modes are selected by bits in the floating point control block (FPCB) that resides in user memory. Thus, a unique set of modes is available for user calculations. The selection bits in the FPCB are defined in Section 5 of this manual. Details of algorithms used for rounding and exception checking are discussed in Appendix C.

For most users, the default modes specified in the proposed IEEE standard will be sufficient. The strength of the proposed IEEE standard is that it provides experienced numerical analysts with the necessary tools (modes) to generate special complex programs while, at the same time, making it easier for the average engineering user to get the best results possible by selecting the defaults.

4.2 ROUNDING MODES

For the following examples, assume z is the infinitely precise result of an arithmetic operation. Further, assume z1 and z2 are the nearest numbers that bracket z and can be exactly represented in the selected precision. That is: $z1<z<z2$ (barely). Then the following criteria are used to select the delivered result.

Round to Nearest (RN) — The nearer of z1 or z2 is selected. In the case of a tie, either of z1 or z2 with a zero, least significant bit is chosen (round to even). This is the default mode.

Round Toward Zero (RZ) — The smaller in magnitude of z1 and z2 is selected (truncation).

Round Toward Plus Infinity (RP) — z2 is selected.

Round Toward Minus Infinity (RM) — z1 is selected.

4.2.1 Rounding Precision

Normally a result is rounded to the precision of its destination. However, when the destination is Extended Format, the user may specify that the result significand be rounded to the precision of the basic format of his choice, although the exponent range remains extended. This allows programs written for an implementation with only the smaller basic formats to be moved to a full implementation and still generate the same results.

4.2.2 No Double Rounding

The MC6839 Floating Point ROM is implemented such that no result will undergo more than one rounding error.

4.3 INFINITY CLOSURE MODES

The way in which infinity is handled in a floating point package may limit the number of applications in which the package can be used. To solve this problem, the proposed IEEE standard requires two types of infinity closures. A bit in the control byte of the floating point control block (FPCB) will select the type of closure that is in effect at any time.

4.3.1 Affine Closure

In affine closure: minus infinity < (every finite number) < plus infinity. Thus infinity takes part in the real number system in the same manner as any other signed quantity. The sign of zero also takes on meaning in affine mode such that:

$+n/+0 =$ plus infinity $> +n/-0 =$ minus infinity where $n =$ floating point number.

In all other operations, $+0$ and -0 participate identically.

4.3.2 Projective Closure

In projective closure: infinity = minus infinity = plus infinity, and all comparisons between infinity and a real number involving order relations other than equal (=) or not equal (≠) are invalid operations. In projective closure, the real number system can be thought of as a circle with zero at the top and infinity at the bottom. Thus, infinity + infinity and infinity − infinity are invalid operations. Projective closure is the default closure.

4.4 EXCEPTION MODES

Existing floating point implementations vary in the way they handle exceptions, so the proposed IEEE standard carefully prescribes how exceptions must be handled and what constitutes an exception. Seven types of exceptions will be recognized by the MC6839 Floating Point ROM; however, only the first five are required by the proposed IEEE standard. These include:

1. Invalid Operation
2. Underflow
3. Overflow
4. Division by Zero
5. Inexact Result
6. Integer Overflow on FINT
7. Comparison of Unordered Values

For each exception the caller will have the option of specifying whether: (1) the routine should trap to a user supplied trap routine on exception or (2) deliver a default specified by the proposed standard and proceed with execution. In either case, a status bit is set in the FPCB status byte and remains set until cleared by the caller's program. The selection of whether to trap or continue is made by setting bits in the enable byte of the FPCB. For more details on the FPCB, see Section 5. For a detailed description of each exception, refer to Appendix D.

If a trap is taken, the floating point package supplies a pointer in the U register that points to the current stack frame (refer to Appendix D). This stack frame contains the following diagnostic information:

1. Which Event Caused the Trap (Overflow, etc.)
2. Its Location in the Caller's Program
3. The Opcode
4. The Input Operands
5. The Default Result in Internal Format

In the event more than one exception occurs, only one trap will be invoked according to the following precedence:

1. Invalid Operation
2. Overflow
3. Underflow
4. Division by Zero
5. Unordered
6. Integer Overflow
7. Inexact Result

The user supplied trap routine (if any) will usually accomplish one of the four items listed below.

1. Change the result on the internal stack to the desired result. This result can then be returned to the caller by the floating point package during its stack cleanup.
2. Correct the result directly in the memory space of the caller. In this case the floating point package does not overwrite the result during its stack cleanup.
3. Do nothing to the result and allow the floating point package to deliver the default value to the result.
4. Abort execution.

All user supplied trap routines must return to the floating point package (using an RTS instruction) for cleanup unless they abort. If the C-bit in the condition code register is set on return, then the result (possibly corrected by the trap) is returned to the destination; otherwise, no result is returned to the destination (with the assumption that the user supplied trap handler already returned a value to the destination).

SECTION 5
FLOATING POINT CONTROL BLOCK

5.1 INTRODUCTION

The floating point control block (FPCB) is a user defined block that contains information needed to select the operating mode for a particular call to the floating point (FP) package. The FPCB must be defined in user RAM. The FPCB is also used to pass status back to the caller or to invoke the trap routine. The caller of the floating point package must pass the address of the FPCB on each call (see Section 6, User Interface, for calling sequence details). The general form of the FPCB is:

Control Byte	0
Enable Byte	1
Status Byte	2
Secondary Status Byte	3
Address of Trap Routine	4
	5

The following paragrpahs discuss the use of the various bytes in the FPCB.

5.2 CONTROL BYTE

The control byte configures the floating point package for the caller's operation and is written by the user.

7	6	5	4	3	2	1	0
Precision			x	NRM	Round Mode		A/P

Bit 0	Closure Bit 0 = Projective Closure 1 = Affine Closure
Bits 1-2	Rounding Mode 00 = Round to Nearest (RN) 01 = Round Toward Zero (RZ) 10 = Round Toward Plus Infinity (RP) 11 = Round Toward Minus Infinity (RM)

Bit 3	Normalize Bit 1 = Normalize denormalized numbers while in internal format before using. Precludes the creation of unnormalized numbers. 0 = Do not normalize denormalized numbers (warning mode).
Bit 4	Undefined, Reserved
Bits 5-7	Precision Mode 000 = Single 001 = Double 010 = Extended With No Forced Rounding of Result 011 = Extended — Force Round Result to Single 100 = Extended — Force Round Result to Double 101 = Undefined, Reserved 110 = Undefined, Reserved 111 = Undefined, Reserved

For move and compare operations, bits 5-7 are "don't cares" since the source and destination precisions are specified by an extra argument passed to the routine. See Section 6, User Interface, for more details of the moves and compare.

Note that if the control byte is set to zero by the user, all defaults in the proposed IEEE standard will be selected.

5.3 STATUS BYTE

The bits in the status byte are set by the MC6839 if any errors have occurred. Note that each bit of the status byte is a "sticky" bit and must be manually reset (written) by the user. The floating point package writes bits into the status byte but never clears existing bits. This is done so that a long calculation can be completed and the status need only be checked once at the end.

7	6	5	4	3	2	1	0
x	INX	IOV	UNOR	DZ	UNF	OVF	IOP

Bit 0	Invalid Operation (also see Secondary Status)
Bit 1	Overflow
Bit 2	Underflow
Bit 3	Division by Zero
Bit 4	Unordered
Bit 5	Integer Overflow
Bit 6	Inexact Result
Bit 7	Undefined, Reserved

5.4 TRAP ENABLE BYTE

If any bit of the trap enable byte is set, it enables the floating point package to trap if that error occurs. The bit position definitions are the same as for the status byte. Note that if a trapping compare is executed and the result is unordered, then the unordered trap will be taken regardless of the state of the UNOR bit in the trap enable byte.

7	6	5	4	3	2	1	0
x	INX	IOV	UNOR	DZ	UNF	OVF	IOP

Bit 0	Invalid Operation
Bit 1	Overflow
Bit 2	Underflow
Bit 3	Division by Zero
Bit 4	Unordered
Bit 5	Integer Overflow
Bit 6	Inexact Result
Bit 7	Undefined, Reserved

5.5 TRAP VECTOR

If a trap occurs, the floating point package will initiate a jump indirectly through the trap address in the FPCB. An index in the A accumulator then indicates the trap type. Trap types are as follows:

0 = Invalid Operation
1 = Overflow
2 = Underflow
3 = Divide by Zero
4 = Unnormalized
5 = Integer Overflow
6 = Inexact Result

If more than one enabled trap occurs, the MC6839 Floating Point ROM returns the index of the highest priority enabled error. Index 0, which is an invalid operation, is the highest priority, whereas, index 6 (inexact result) is the lowest.

5.6 SECONDARY STATUS

The floating point package writes a status into this byte whenever a new IOP occurs. As is the case with the status byte, it is up to the caller to reset the "IOP type" field.

7	6	5	4	3	2	1	0
x	x	x	Invalid Operation Type				

Bits 0-4 represent the invalid operation type field. These four bits are encoded as shown below.

0 = No IOP error

1 = Square Root of: a Negative Number, Infinity in Projective Mode, or a Not Normalized Number

2 = (+ Infinity) + (− Infinity) in Affine mode

3 = Tried to Convert a NAN to Binary Integer

4 = In Division: 0/0, Infinity/Infinity, or the Divisor is not Normalized and the Dividend is Not Zero and is Finite

5 = One of the Input Arguments was a Trapping NAN

6 = Unordered Values Compared via Predicate Other Than = or ≠

7 = k Out of Range for BINDEC or
p Out of Range or DECBIN

8 = Projective Closure Use of +/− Infinity in Add or Subtract

9 = 0 × Infinity

10 = In Remainder arg2 is Zero, or Not Normalized in arg1 is Infinite

11 = Unused, Reserved

12 = Unused, Reserved

13 = Unused, Reserved

14 = Unused, Reserved

15 = Tried to MOV a Single Denormalized Number to a Double Destination

16 = Tried to Return an Unnormalized Number to Single or Double (also called Invalid Result in the Proposed IEEE Standard).

17-31 = Unused, Reserved

SECTION 6
USER INTERFACE

6.1 INTRODUCTION

There are two types of calls to the floating point package: register calls and stack calls. For register calls, the user loads the machine register with pointers (addresses) to the operand(s) and to the result; the call to the floating point package is then performed. For stack calls, the operand(s) is pushed onto the stack and the call to the floating point package is performed. The result then replaces the operands on the stack after completion. The operand(s) must be pushed least significant bytes first; this is consistent with the other Motorola architectures in that the most significant byte resides in the lowest address. The two types of calls look like:

General form of a register call:

```
    load registers
    LBSR     FPREG     register call
    FCB      opcode
```

General form of a stack call:

```
    push arguments
    LBSR     FPSTAK    stack call
    FCB      opcode
    pull result
```

6.2 OPERATION OPCODES AND ENTRY POINTS

The suggested mnemonics and the opcode values for the various operations available in this floating point package are shown below (in opcode order).

Mnemonic	Opcode Value	Operation Description
FADD	00	Add
FSUB	02	Subtract
FMUL	04	Multiply
FDIV	06	Divide
FREM	08	Remainder
FSQRT	12	Square Root
FINT	14	Integer Part
FFIXS	16	Float – Short Integer
FFIXD	18	Float – Double Integer
BINDEC	1C	Binary Float – BCD String
FAB	1E	Absolute Value
FNEG	20	Negate
DECBIN	22	BCD String – Binary Float
FFLTS	24	Short Integer – Float
FFLTD	26	Double Integer – Float
FCMP	8A	Compare
FPCMP	8E	Predicate Compare
FMOV	9A	Move (or Convert) arg1 – Result
FTCMP	CC	Trapping Compare
FTPCMP	D0	Trapping Predicate Compare

The two entry points to the MC6839 are referred to as FPREG (register call) and FPSTAK (stack call). Their addresses are:

FPREG = ROM starting address + $3D
FPSTAK = ROM starting address + $3F

The first $3C locations of the ROM contain a fixed size ROM header. The entry points for the floating package are located in a branch table immediately following this header. Therefore, the addresses of the entry points will remain constant for future versions of the ROM.

6.3 STACK REQUIREMENTS

When the MC6839 Floating Point ROM is called by the user, local storage is reserved on the hardware stack by the floating point package. The input arguments are then moved from user memory to the local storage area, and are expanded into a convenient internal format. The operations use these "internal" numbers to arrive at an "internal" result. The "internal" result is then converted to the memory format of the result and returned (as the result) to the user. For this reason, the user must insure that adequate memory exists on the hardware stack before calling the MC6839 Floating Point ROM. The maximum stack sizes that any particular operation will ever require are:

register calls	170 bytes
stack calls	200 bytes

6.4 CALLING SEQUENCE

6.4.1 Register Call

In this calling method the addresses of the arguments and the floating point control block (FPCB) are passed in the register:

U = address of argument 1
Y = address or argument 2
X = address of result
D = address of FPCB

If an argument is not used in a particular operation, it need not be included. In monadic operations, Y contains the address of the single argument. The result may be the same address as either of the arguments. All registers will be restored on exit.

Example of a position independent call to the add routine:

```
LEAU    arg1, PCR
LEAY    arg2, PCR
LEAX    FPCBPTR, PCR      pointer to FPCB*
TFR     X, D
LEAX    result, PCR
LBSR    fpreg
FCB     FADD
```

Example of a position independent monadic call to the square root routine:

```
LEAY    arg2, PCR
LEAX    FPCBPTR, PCR
TFR     X, D
LEAX    result, PCR
LBSR    FPREG
FCB     FSQRT
```

For some operations the arguments have slightly different meanings. See Appendix A for details. All subroutines in the floating point package are re-entrant and position independent. However, the caller must use caution to insure that his call does not violate the rules of re-entrancy or position independence. For example, each calling task should have its own FPCB to remain re-entrant. Also, if in the previous examples load immediates had been used, rather than load effective address program counter relative, the calling program could not have been position independent.

6.4.2 Stack Call

In this mode the actual argument(s), not their addresses, and the address of the FPCB are assumed to be on the top of the hardware stack and they will be removed and replaced by the result on exit. If two arguments are on the stack, then argument 2 should be above (lower address) argument 1. The address of the FPCB is on the top of the stack above the argument(s).

Example of a stack call to the add routine:

```
push argument 1
push argument 2
push FPCBPTR          Pointer to FPCB
LBSR FPSTAK
FCB FADD
pull result
```

For monadic operations, arg2 contains the single input argument and there is no arg1. On return, the FPCB pointer and any other parameters are lost from the top of the stack. The only object left on the stack after an operation is the result. For some operations, the arguments have slightly different meaning. See Appendix A for details.

*Two instructions are required here if the caller wishes his call to remain position independent (there is no LEAD instruction).

APPENDIX A
OPERATION DESCRIPTIONS

A.1 INTRODUCTION

This appendix contains detailed information covering specific operations and their required calling sequences. The operations are arranged in alphabetical order with a summary listing on the last page of this appendix. Detailed descriptions of the algorithms are provided in Appendix B.

A.2 NOTATION

In describing each specific operation, symbols are used to indicate the operation. Table A-1 lists these symbols and their meaning. Abbreviations which are used for the source form, various registers, bits, bytes, etc. are listed in Table A-2.

Table A-1. Specific Operation Notation

Symbol	Meaning
—	Is Transferred As (Stored As)
●	Boolean Exclusive OR
+	Arithmetic Plus
−	Arithmetic Minus
×	Arithmetic Multiply

Table A-2. Abbreviations

Abbreviation	Meaning
arg	Argument.
BCS	Branch if Carry Set
BGE	Branch if Greater Than or Equal to Zero
*BINDEC	Binary Floating Point to Decimal String
*DECBIN	Decimal String to Binary Floating Point
*FAB	Absolute Value of an Argument.
*FADD	Add
FCB	Form Constant Byte (assembler directive)
*FCMP	Compare. Compares two arguments and sets condition codes
*FDIV	Divide. Divides one argument by another
*FFIXD	Fix. Double converts an argument from a floating point number into a 32-bit binary integer.
*FFIXS	Fix. Single converts an argument from a floating point number into a 16-bit binary integer
*FFLTD	Float. Double converts a 32-bit binary integer into a floating point result
*FFLTS	Float. Single converts a 16-bit binary integer into a floating point result
*FINT	Integer Part. Floating point argument is converted to its floating point integer part
*FMOV	Move. Moves an argument to the result (with any implied conversions)
*FMUL	Multiply. Multiplies two arguments and stores the result.
*FNEG	Negate. Change the sign of an argument
FPCB	Floating Point Control Block
*FPCMP	Predicate Compare. Compares two arguments and affirms or disaffirms a predicate
FPREG	Register Call Entry Point
FPSTAK	Stack Call Entry Point
*FSQRT	Square Root. Stores the square root of an argument
*FSUB	Subtract
*FTCMP	Trapping Compare. Compares two arguments and sets condition codes. Traps on unordered
*FTPCMP	Trapping Predicate Compare. Compares two arguments and affirms or disaffirms predicate. Traps on unordered
LBSR	Branch to Subroutine

*These abbreviations represent the specific operations which are described in this appendix. See Appendix H for more definitions

ABSOLUTE VALUE

Mnemonic: FAB

Operation: |arg2|—result

Description: Return absolute value of arg2 as the result.

Opcode: $1E

Precisions: All. The result will have the same precision as arg2. The precision is specified in bits 5-7 of the FPCB control byte.

Register Calling Sequence:

```
load X with address of result
load Y with address of arg2
load D with address of FPCB
LBSR FPREG (FPREG = ROM start + $3D)
FCB FAB
```

The result is automatically returned to user memory. This calling sequence is the same for all monadic calls.

Stack Calling Sequence:

```
push arg2
push address of FPCB
LBSR FPSTAK (FPSTAK = ROM start + $3F)
FCB FAB
pull result
```

Only the result is left on the stack after return from the subroutine. This calling sequence is the same for all monadic calls.

ADD

Mnemonic: FADD

Operation: arg1 + arg2 – result

Description: Add arg2 to arg1 and store the result.

Opcode: $00

Precisions: All. Both arg1 and arg2 must be of the same precision. The result will also be the same precision. The precision is specified in bits 5-7 of the FPCB control byte.

Register Calling Sequence:

```
load X with the address of the result
load Y with the address of arg2
load U with the address of arg1
load D with the address of the FPCB
LBSR FPREG (FPREG = ROM start + $3D)
FCB FADD
```

The result is automatically returned to user memory. This calling sequence is the same for all dyadic calls.

Stack Calling Sequence:

```
push arg1
push arg2
push address of FPCB
LBSR FPSTAK (FPSTAK = ROM start + $3F)
FCB FADD
pull result
```

Only the result is left on the stack after the call. This calling sequence is the same for all dyadic calls.

BINARY FLOATING TO DECIMAL STRING

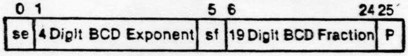
Mnemonic: BINDEC

Operation: arg2 – BCD string with k significant digits

Description: Convert a floating point argument in arg2 to an unpacked BCD string in the result. A parameter k is also passed to the routine to indicate the number of significant digits desired in the result ($1 \leq k \leq 9$ for single; $1 \leq k \leq 17$ for double).

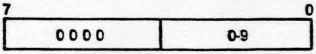
Opcode: $1C

Precisions: Single and double results are delivered to the accuracy required by the proposed IEEE standard. Extended results, however, are not necessarily more accurate than double and may take considerably more time to compute. The precision of arg2 is specified in bits 5-7 of the FPCB control byte. The output BCD string is a standard 26 byte BCD string of the form:



se = sign of the exponent. 00 = plus, $0F = minus.
sf = sign of the fraction. 00 = plus, $0F = minus.
p = number of fraction digits to the right of the decimal point (one byte).

All BCD digits are unpacked and right justified in each byte:



Since some special floating point values have no obvious BCD equivalent, the sign of the exponent (se) is used to indicate these special cases:

se = 00 = regular positive number.
= 0F = regular negative number.
= 0C = NAN. The four digit BCD exponent contains the unpacked hex address that was in the NAN.
= 0B = minus infinity. All remaining bytes of the BCD string are zero.
= 0A = plus infinity. All remaining bytes of the BCD string are zero.

Even though these special numbers can be created as output, they are not legal inputs to DECBIN.

Register Calling Sequence:

load X with address of result
load Y with address of arg2
load U with k
load D with address of FPCB
LBSR FPREQ (FPREG = ROM start + $3D)
FCB BINDEC

The resultant BCD string is automatically returned to the user.

Stack Calling Sequence:

push arg2
push k
push address of FPCB
LBSR FPSTAK (FPSTAK = ROM start + $3F)
FCB BINDEC
pull BCD string

COMPARE

Mnemonic: FCMP, FTCMP, FPCMP, FTPCMP

Operation: arg1 – arg2 (return condition code register or affirm/disaffirm a predicate)

Description: Compare arg1 with arg2. Both arg1 and arg2 may be of different precisions. Two basic types of compares are provided. One returns condition codes in the condition code register to the user to indicate the result of the comparison. The other is given a predicate (e.g., is arg1 equal to arg2?) and either affirms or disaffirms the predicate.

1) Condition code compares:

FCMP
Compare arg1 with arg2 and set the condition codes. Do not trap on unordered unless the trap on unordered bit (UNOR) is set in the trap enable byte of the FPCB.

FTCMP
Compare arg1 with arg2 and set the condition codes. Trap if the unordered conditions occur regardless of the state of the UNOR bit in the trap enable byte of the FPCB.

The intermediate result of any comparison can yield one of five possible results: arg1 is > arg2, arg1 is < arg2, arg1 = arg2, arg1 ≠ arg2, or arg1 cannot be compared to arg2 (unordered). The unordered condition occurs when a comparison is made between a NAN and anything else or when infinity is compared to anything except itself in projective closure. This intermediate result is then used to set the condition codes as follows:

Result	N	Z	V	C
>	0	0	0	0
<	1	0	0	0
=	0	1	0	0
unordered	0	0	0	1

The remaining condition code register bits (E, F, H, and I) are unaffected by compare.

This allows the following signed branches to be taken immediately following the return from FCMP or FTCMP.

Condition	Branch	Test for Branch
>	(L)BGT	[not (N • V)] and (not Z) = 1
≥	(L)BGE	not (N • V) = 1
<	(L)BLT	not (N • V) = 0
≤	(L)BLE	not (N • V) and (not Z) = 0
=	(L)BEQ	Z = 1
≠	(L)BNE	Z = 0
unordered	(L)BCS	C = 1

If FCMP and the unordered trap is disabled, a BCS should immediately follow the call and precede any of the other branches:

```
      LBSR      fpxxx
      FCB       FCMP
      BCS       unordered
      BGE       label
```

Note that this implementation of compare conditions (as defined by the proposed IEEE standard) does not support the dichotomy principle normally associated with integer compare. For example, BGE is not necessarily the inverse of BLT (the result may be unordered too). Compiler writers must take care not to switch the condition of a branch during code generation.

2) Predicate Compares

FPCMP
Compare arg1 with arg2. Either affirm or disaffirm an input predicate. Do not trap on unordered unless the UNOR bit is set in the trap enable byte of the FPCB. For register calls the Z-bit in the condition code register is set to 1 for affirm (true) and set to 0 for disaffirm (false). For stack calls a byte of zeros is pushed on top of the stack for true and a byte of ones ($FF) is pushed for false.

FTPCMP
Compare arg1 with arg2. Either affirm or disaffirm an input predicate. Trap if the unordered condition occurs regardless of the state of the UNOR bit in the trap enable byte of the FPCB. For register calls the Z-bit in the condition code register is set to 1 for affirm (true) and set to 0 for disaffirm (false). For stack calls a byte of zeros is pushed on top of the stack for true and a byte of ones ($FF) is pushed for false.

Unordered conditions occur when a comparison is made between a NAN and anything else or when infinity is compared to anything except itself in projective closure.

The predicate to be affirmed or disaffirmed is passed to the compare in the parameter word:

15	11	Predicates		8	7	6 4	3	2 0
000 ≠	>	=	<	u	0	arg1	0	arg2

The predicates are >, ×, <, and unordered, or a reasonable combination of these (e.g., > =). The intermediate result of a predicate compare is either >, =, <, or unordered. The table below gives the predicate affirmed or disaffirmed for each possible intermediate result.

Intermediate Result	Predicates Affirmed
less than	< ≤ ,
equal	= ≤ ≥
greater than	> ≥ ,
unordered	unordered

Intermediate Result	Predicates Disaffirmed
less than	= ≥ > unordered
equal	< > unordered ≠
greater than	= ≤ < unordered
unordered	< ≤ = ≥ > < >

The result returned for affirmed is a zero byte and for disaffirmed it is a −1 or $FF byte for a stack call. For a register call, Z = 1 if the predicate is affirmed.

Opcodes: FCMP = $8A
FTCMP = $CC
FPCMP = $8E
FTPCMP = $D0

Precisions: Since the compares allow arg1 and arg2 to be of different precisions, a parameter word must be passed on each call to any compare. The format of the parameter word is:

15	11	Predicates			8	7	6 4	3	2 0
000 ≠	>	=	<	u		0	arg1	0	arg2

Where arg1 or arg2 is defined:

000	Single
001	Double
010	Extended
011	Unused (defaults to extended)
100	Unused (defaults to extended)
101-111	Undefined

Since the parameter word specifies both arguments of the compare, bits 5-7 of the control byte of the FPCB do not affect the compare instructions.

Register Calling Sequence:

load X with the parameter word
load Y with address of arg2
load U with address of arg1
load D with address of FPCB
LBSR FPREG (FPREG = ROM start + $3D)
FCB <Opcode>

The result is returned in the condition code register. It is either a setting of the condition code register (condition code call) or the Z bit is set to 1 for affirm and Z = 0 for disaffirm (predicate calls).

Stack Calling Sequence:

push arg1
push arg2
push parameter word
push address of the FPCB
LBSR FPREG (FPREG = ROM start + $3F)
FCB <Opcode>
pull result

If the compare is a condition code compare, no result is delivered on the stack — only the condition codes are returned in the condition code register. If the compare is a predicate compare, a 1-byte result is returned on top of the stack. The result = 0 for affirmed and −1 ($FF) for disaffirmed.

DECIMAL STRING TO BINARY FLOATING POINT

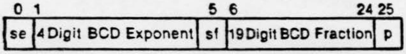
Source Form: DECBIN

Operation: BCD string—floating point result

Description: Convert a standard BCD string into a binary floating point result. The value "p" in the standard decimal string indicates the number of digits of the fraction that are to the right of the decimal point.

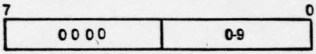
Opcode: $22

Precisions: The precision of the result is defined by bits 5-7 of the FPCB control byte. The input BCD string is a standard 26 byte BCD string of the form:



se = sign of the exponent. 00 = plus, $0F = minus (one byte).
sf = sign of the fraction. 00 = plus, $0F = minus (one byte).
p = number of fraction digits to the right of the decimal point (one byte).

All BCD digits are unpacked and right justified in each byte:



The byte ordering of the fraction and exponent is consistent with all Motorola processors in that the most significant BCD digit is in the lowest memory address.

Register Calling Sequence:
load X with the address of result
load U with the address of the BCD input string
load D with the address of the FPCB
LBSR FPREG (FPREG = ROM start + $3D)
FCB DECBIN
The result is automatically returned to the user.

Stack Calling Sequence:
push the BCD string
push address of the FPCB
LBSR FPSTAK (FPSTAK = ROM start + $3F)
FCB DECBIN
pull floating point result

DIVIDE

Mnemonic: FDIV

Operation: arg1/arg2—result

Description: Divide arg1 by arg2 and store the result.

Opcode: $06

Precisions: Both arg1 and arg2 must be of the same precision. The result will also be the same precision. The precision is specified in bits 5-7 of the FPCB control byte.

Register Calling Sequence:

```
load X with the address of the result
load Y with the address of arg2
load U with the address of arg1
load D with the address of the FPCB
LBSR FPREG (FPREG = ROM start + $3D)
FCB FDIV
```

The result is automatically returned to user memory. The calling sequence is the same for all dyadic calls.

Stack Calling Sequence:

```
push arg1
push arg2
push address of FPCB
LBSR FPSTAK (FPSTAK = ROM start + $3F)
FCB FDIV
pull result
```

Only the result is left on the stack after the call. This calling sequence is the same for all dyadic calls.

FIX

Mnemonic: FFIXS, FFIXD

Operation: arg2—binary integer result

Description: Converts arg2 from a floating point number into a 16- or 32-bit binary integer. If arg2 is infinity, then the integer returned is the largest or smallest twos complement integer.

Opcode:
FFIXS = $16 (16-bit integer)
FFIXD = $18 (32-bit integer)

Precisions: Same as absolute value except that the result will be a 16- or 32-bit integer as specified by the opcode.

Register Calling Sequence:

```
load X with the address of the result
load Y with the address of arg2
load D with the address of the FPCB
LBSR FPREG (FPREG = ROM start + $3D)
FCB FFIXS or FFIXD
```

The result is automatically returned to user memory. The calling sequence is the same for all monadic calls.

Stack Calling Sequence:

```
push arg2
push address of FPCB
LBSR FPSTAK (FPSTAK = ROM start + $3F)
FCB FFIXS or FFIXD
pull result
```

Only the result is left on the stack after return from the subroutine. This calling sequence is the same for all monadic calls.

FLOAT

Mnemonic: FFLTS, FFLTD

Operation: Binary integery arg2—floating point result

Description: Converts a 16- or 32-bit integer into a floating point result.

Opcode: FFLTS = $24 (16-bit binary integer)
FFLTD = $26 (32-bit binary integer)

Precisions: All. The size of the binary integer is specified in the opcode. The precision is specified in bits 5-7 of the FPCB control byte.

Register Calling Sequence:

load X with the address of the result
load Y with the address of arg2
load D with the address of the FPCB
LBSR FPREG (FPRET = ROM start + $3D)
FCB FFLTS or FFLTD

The result is automatically returned to user memory. The calling sequence is the same for all monadic calls.

Stack Calling Sequence:

push arg2
push address of FPCB
LBSR FPSTAK (FPSTAK = ROM start + $3F)
FCB FFLTS or FFLTD
pull result

Only the result is left on the stack after return from the subroutine. This calling sequence is the same for all monadic calls.

INTEGER PART

Mnemonic: FINT

Operation: Integer part (arg2)—floating point result

Description: The floating point argument in arg2 is converted to its floating point integer part. This differs from FIX which returns a binary integer. Integer part returns a floating point number. For example, the integer part of 3.14159 is 3.00000 if the rounding mode is round to nearest.

Opcode: $14

Precisions: All. The result will have the same precision as arg2. The precision is specified in bits 5-7 of the FPCB control byte.

Register Calling Sequence:

```
load X with the address of the result
load Y with the address of arg2
load D with the address of the FPCB
LBSR FPREG (FPREG = ROM start + $3D)
FCB FINT
```

The result is automatically returned to user memory. The calling sequence is the same for all monadic calls.

Stack Calling Sequence:

```
push arg2
push address of FPCB
LBSR FPSTAK (FPSTAK = ROM start + $3F)
FCB FINT
pull result
```

Only the result is left on the stack after return from the subroutine. This calling sequence is the same for all monadic calls.

MOVE

Mnemonic: FMOV

Operation: arg2—result

Description: For register calls, the move instruction moves arg2 to the result. Since moves allow mixed precisions, they can be used to convert a number from one precision to another during the move. For stack calls, the move is essentially a "convert precision of stack top" operation.

Opcode: $9A

Precisions: The move allows arg2 and the result to be of different precisions. In order to specify the two precisions, a parameter word must be passed on each call to move. The form of the parameter word is:

```
15                  8   6                  4   2                   0
|     00000000      | 0 |       arg2       | 0 |       result       |
```

Where arg2 (source) or result (destination) is defined:

000	Single
001	Double
010	Extended
011	Extended round to single
100	Extended round to double
101-111	Illegal

Since the parameter word specifies both arguments of the move, bits 5-7 of the FPCB control byte do not affect the move operation.

Register Calling Sequence:

load X with the address of the result
load Y with the address or arg2
load U with the address of precision parameter word
load D with the address of the FPCB
LBSR FPREG (FPREG = ROM start + $3D)
FCB FMOV

The result is automatically returned to user memory in the precision specified in the parameter word.

Stack Calling Sequence:

```
push arg2
push precision parameter word
push address of FPCB
LBSR FPSTAK (FPSTAK=ROM start+$3F)
FCB FMOV
pull result
```

Only the result is left on the stack after the operation. The result has the precision (and size) specified in the precision parameter word.

MULTIPLY

Mnemonic: FMUL

Operation: arg1 × arg2—result

Description: Multiply arg1 and arg2 and store the result.

Opcode: $04

Precisions: arg1 and arg2 must be of the same precision. The result will also be the same precision. The precision is specified in bits 5-7 of the FPCB control byte.

Register Calling Sequence:

```
load X with the address of the result
load Y with the address of arg2
load U with the address or arg1
load D with the address of the FPCB
LBSR FPREG (FPREG = ROM start + $3D)
FCB FMUL
```

The result is automatically returned to user memory. The calling sequence is the same for all dyadic calls.

Stack Calling Sequence:

```
push arg1
push arg2
push address of FPCB
LBSR FPSTAK (FPSTAK = ROM start + $3F)
FCB FMUL
pull result
```

Only the result is left on the stack after the call. This calling sequence is the same for all dyadic calls.

NEGATE

Mnemonic: FNEG

Operation: arg2—result

Description: Negate arg2 by changing the sign and store as the result.

Opcode: $20

Precisions: All. The result will also be the same precision as arg2. The precision is specified in bits 5-7 of the FPCB control byte.

Register Calling Sequence:

```
load X with the address of the result
load Y with the address of arg2
load D with the address of the FPCB
LBSR FPREG (FPREG = ROM start + $3D)
FCB FNEG
```

The result is automatically returned to user memory. The calling sequence is the same for all monadic calls.

Stack Calling Sequence:

```
push arg2
push address of FPCB
LBSR FPSTAK (FPSTAK = ROM start + $3F)
FCB FNEG
pull result
```

Only the result is left on the stack after the call. This calling seqüence is the same for all monadic calls.

REMAINDER

Mnemonic: FREM

Operation: arg1 – (arg2 × n) – result [where n = integer part of (arg1/arg2) in round nearest]

Description: Finds the remainder of arg1/arg2 and stores it as the result. Note, as defined by the proposed IEEE standard, this is not the same as "modulo." For example, the remainder of 8/3 is – 1 not 2. This can be seen by substituting 8 and 3 in the equation in the operation description:

n – integer part of 8/3 = 3 (round nearest)
remainder = 8 – (3 × 3) = – 1

This form of remainder was chosen for a number of reasons. First, it is the remainder most useful for scaling the inputs to trigonometric subroutines. Secondly, all other remainder type functions may be easily derived from this one. For example, the "modulo" function is found by taking:

Z = remainder (arg2/arg1);
if Z < 0 then Z = Z + arg1.

Opcode: $08

Precisions: arg1 and arg2 must be of the same precision. The result will also be the same precision. The precision is specified in bits 5-7 of the FPCB control byte.

Register Calling Sequence:

```
load X with the address of the result
load Y with the address of arg2
load D with the address of the FPCB
LBSR FPREG (FPREG = ROM start + $3D)
FCB FREM
```

The result is automatically returned to user memory. The calling sequence is the same for all dyadic calls.

Stack Calling Sequence

```
push arg2
push address of FPCB
LBSR FPSTAK (FPSTAK = ROM start + $3F)
FCB FREM
pull result
```

Only the result is left on the stack after the call. This calling sequence is the same for all dyadic calls.

SUBTRACT

Mnemonic: FSUB

Operation: arg1 – arg2 – result

Description: Subtract arg2 from arg1 and store the result.

Opcode: $0F

Precisions: All. Both arg1 and arg2 must be of the same precision. The result will also be the same precision. The precision is specified in bits 5-7 of the FPCB control byte.

Register Calling Sequence:

```
load X with the address of the result
load Y with the address of arg2
load U with the address of arg1
load D with the address of the FPCB
LBSR FPREG (FPREG = ROM start + $3D)
FCB FSUB
```

The result is automatically returned to user memory. The calling sequence is the same for all dyadic calls.

Stack Calling Sequence:

```
push arg1
push arg2
push address of FPCB
LBSR FPSTAK (FPSTAK = ROM start + $3F)
FCB FSUB
pull result
```

Only the result is left on the stack after the call. This calling sequence is the same for all dyadic calls.

SQUARE ROOT

Mnemonic: FSQRT

Operation: Square root of arg2—result

Description: Returns the square root of arg2 as the result.

Opcode: $12

Precisions: All. The result will also be the same precision as arg2. The precision is specified in bits 5-7 of the FPCB control byte.

Register Calling Sequence:

```
load X with the address of the result
load Y with the address of arg2
load D with the address of the FPCB
LBSR FPREG (FPREG = ROM start + $3D)
FCB FSQRT
```

The result is automatically returned to user memory. The calling sequence is the same for all monadic calls.

Stack Calling Sequence:

```
push arg1
push address of FPCB
LBSR FPSTAK (FPSTAK = ROM start + $3F)
FCB FSQRT
pull result
```

Only the result is left on the stack after return from the subroutine. This calling sequence is the same for all monadic calls.

Table A-3. MC6839 Calling Sequence and Opcode Summary Table

Function	Opcode	Register Calling Sequence	Stack Calling Sequence[1]
FADD FSUB FMUL FDIV	$00 $02 $04 $06	U— Addr. of Argument #1 Y— Addr of Argument #2 D— Addr of FPCB X— Addr of Result LBSR FPREG FCB <opcode>	Push Argument #1 Push Argument #2 Push Addr of FPCB LBSR FPSTAK FCB <opcode> Pull Result
FREM FSQRT FINT FFIXS FFIXD FAB FNEG FFLTS FFLTD	$08 $12 $14 $16 $18 $1E $20 $24 $26	Y— Addr of Argument D— Addr. of FPCB X— Addr of Result LBSR FPREG FCB <opcode>	Push Argument Push Addr of FPCB LBSR FPSTAK FCB <opcode> Pull Result
FCMP FTCMP FPCMP FTPCMP	$8A $CC $8E $D0	U— Addr of Argument #1 Y— Addr of Argument #2 D— Addr of FPCB X— Parameter Word LBSR FPREG FCB <opcode> NOTE Result returned in the CC register For predicate compares the Z-Bit is set if predicate is affirmed cleared if disaffirmed	Push Argument #1 Push Argument #2 Push Parameter Word Push Addr of FPCB LBSR FPSTAK FCB <opcode> Pull Result (if predicate compare) NOTE Result returned in the CC register for regular compares For predicate compares a one byte result is returned on the top of the stack The result is zero if affirmed and − 1($FF) if disaffirmed
FMOV	$94	U— Precision Parameter Word Y— Addr of Argument D— Addr of FPCB X— Addr of Result LBSR FPREG FCB FMOV	Push Argument Push Precision Parameter Word Push Addr of FPCB LBSR FPSTAK FCB FMOV Pull Result
BINDEC	$1C	U— k (# of digits in result) Y— Addr of Argument D— Addr of FPCB X— Addr of Decimal Result LBSR FPREG FCB BINDEC	Push Argument Push k Push Addr of FPCB LBSR FPSTAK FCB BINDEC Pull BCD String
DECBIN	$22	U— Addr of BCD Input String D— Addr of FPCB X— Addr of Binary Result LBSR FPREG FCB DECBIN	Push Addr of BCD Input String Push Addr of FPCB LBSR FPSTAK FCB DECBIN Pull Binary Result

[1]All arguments are pushed on the stack least-significant bytes first so that the high-order byte is always pushed last and resides in the lowest address

Entry points to the MC6839 are defined as follows

FPREG = ROM start + $3D
FPSTAK = ROM start + $3F

APPENDIX B
APPLICATION EXAMPLE OF THE QUADRATIC EQUATION

This appendix provides an application example using the MC6839 Floating Point ROM. The program shown below is one that finds the roots to quadratic equations using the classic formula:

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Note that the program uses a standard set of macro instructions to set up the parameters in the correct calling sequences. Perhaps the easiest way to program MC6839 Floating Point ROM is through the use of these macro instructions. Errors are reduced because, once the macro instructions are shown to be correct, their internal details can be ignored allowing the programmer to concentrate only on the problem at hand.

```
 NAM QUAD
*
* HERE IS A SIMPLE EXAMPLE INVOLVING THE QUADRATIC EQUATION THAT
* SHOULD SERVE TO ILLUSTRATE THE USE OF THE MC6839 IN AN ACTUAL
* APPLICATION.
*
* LINKING LOADER DEFINITIONS
*
 XDEF  QUAD
*
 XREF  FPREG
*
* RMBS FOR THE OPERANDS, BINARY TO DECIMAL CONVERSION BUFFERS,
* AND THE FPCB.
*
ACOEFF  RMB  26      COEFFICIENT A IN  AX^2 + BX + C
BCOEFF  RMB  26      COEFFICIENT B
CCOEFF  RMB  26      COEFFICIENT C
*
REG1  RMB  4        REGISTER 1
REG2  RMB  4        REGISTER 2
REG3  RMB  4        REGISTER 3
*
FPCB  RMB  4         FLOATING POINT CONTROL BLOCK
*
TWO  FCB  $40,00,00,00        FLOATING PT. CONSTANT  TWO
FOUR  FCB  $40,$80,00,00                             FOUR
*
*
* HERE ARE THE EQUATES AND MACRO DEFINITIONS TO ACCOMPANY THE
* QUADRATIC EQUATION EXAMPLE OF AN MC6839 APPLICATION.
*
ADD  EQU  00                 OPCODE VALUES
SUB  EQU  02
MUL  EQU  04
DIV  EQU  06
SQRT  EQU  $12
ABS  EQU  $1E
NEG  EQU  $20
BNDC  EQU  $1C
DCBN  EQU  $22
*
*
* MACRO DEFINITIONS
*
* HERE ARE THE CALLING SEQUENCE MACROS
*
MCALL  MACR
*
*  MCALL SETS UP A MONADIC REGISTER CALL.
*
*  USAGE: MCALL  <INPUT OPERAND>,<OPERATION>,<RESULT>
*
```

```
 LEAY  \0,PCR                    POINTER TO THE INPUT ARGUMENT
 LEAX  FPCB,PCR                  POINTER TO THE FLOATING POINT CONTROL BLOCK
 TFR  X,D
 LEAX  \2,PCR                    POINTER TO THE RESULT
 LBSR  FPREG                     CALL TO THE MC6839
 FCB  \1                         OPCODE
*
 ENDM
*
*
DCALL  MACR
*
*  DCALL SETS UP A DYADIC REGISTER CALL
*
*  USAGE: DCALL  <ARGUMENT #1>,<OPERATION>,<ARGUMENT #2>,<RESULT>
*
 LEAU  \0,PCR                    POINTER TO ARGUMENT #1
 LEAY  \2,PCR                    POINTER TO ARGUMENT #2
 LEAX  FPCB,PCR                  POINTER TO THE FLOATING POINT CONTROL BLOCK
 TFR  X,D
 LEAX  \3,PCR                    POINTER TO THE RESULT
 LBSR  FPREG                     CALL TO THE MC6839
 FCB  \1                         OPCODE
*
 ENDM
*
*
DECBIN  MACR
*
* DECBIN SETS UP A REGISTER CALL TO THE DECIMAL TO BINARY CONVERSION FUNCTION.
*
* USAGE: DECBIN  <BCD STRING>,<BINARY RESULT>
*
 LEAU  \0,PCR                    POINTER TO THE BCD INPUT STRING
 LEAX  FPCB,PCR                  POINTER TO THE FLOATING POINT CONTROL BLOCK
 TFR  X,D
 LEAX  \1,PCR                    POINTER TO THE RESULT
 LBSR  FPREG                     CALL TO THE MC6839
 FCB  DCBN                       OPCODE
*
 ENDM
*
*
BINDEC  MACR
*
* BINDEC SETS UP A REGISTER CALL TO THE BINARY TO DECIMAL CONVERSION FUNCTION.
*
* USAGE: BINDEC  <BINARY INPUT>,<BCD RESULT>,<# OF SIGNIFICANT DIGITS RESULT>
*
 LDU  \2                         # OF SIGNIFICANT DIGITS IN THE RESULT
 LEAY  \0,PCR                    POINTER TO THE BINARY INPUT
 LEAX  FPCB,PCR                  POINTER TO THE FLOATING POINT CONTROL BLOCK
 TFR  X,D
 LEAX  \1,PCR                    POINTER TO THE BCD RESULT
```

```
  LBSR  FPREG                             CALL TO THE MC6839
  FCB  BNDC                               OPCODE
*
  ENDM
*
*
QUAD  EQU  *
*
  LDS  #$6FFF                                   INITIALIZE THE STACK POINTER
*
  LEAX  FPCB,PCR
  LDB  #4
  WHILE  B,GT,#0                                INITIALIZE STACK FRAME TO
    DECB                                        SINGLE, ROUND NEAREST.
    CLR  B,X
*
  ENDWH
*
* CONVERT THE INPUT OPERANDS FROM BCD STRINGS TO THE INTERNAL
* SINGLE BINARY FORM.
*
  DECBIN  ACOEFF,ACOEFF
  DECBIN  BCOEFF,BCOEFF
  DECBIN  CCOEFF,CCOEFF
*
.* NOW START THE ACTUAL CALCULATIONS FOR THE QUADRATIC EQUATION
*
  DCALL  BCOEFF,MUL,BCOEFF,REG1                 CALCULATE  B^2
  DCALL  ACOEFF,MUL,CCOEFF,REG2                 CALCULATE  AC
  DCALL  REG2,MUL,FOUR,REG2                     CALCULATE  4AC
  DCALL  REG1,SUB,REG2,REG1                     CALCULATE  B^2 - 4AC
*
*  CHECK RESULT OF B^2 - 4AC TO SEE IF ROOTS ARE REAL OR IMAGINARY
*
  LDA  REG1,PCR
  IFCC  GE                                      SIGN IS POSITIVE; ROOTS REAL
    MCALL  REG1,SQRT,REG1                       CALCULATE  SQRT( B^2 - 4AC )
    DCALL  ACOEFF,MUL,TWO,REG2                  CALCULATE  2A
    MCALL  BCOEFF,NEG,BCOEFF                    NEGATE  B
*
    DCALL  BCOEFF,ADD,REG1,REG3                 CALCULATE  -B + SQRT( B^2 - 4AC )
    DCALL  REG3,DIV,REG2,REG3                   CALCULATE  (-B + SQRT( B^2 - 4AC ))/2A
    BINDEC  REG3,ACOEFF,#5                      CONVERT RESULT TO DECIMAL
*
    DCALL  BCOEFF,SUB,REG1,REG3                 CALCULATE  -B - SQRT( B^2 - 4AC )
    DCALL  REG3,DIV,REG2,REG3                   CALCULATE  (-B + SQRT( B^2 - 4AC ))/2A
    BINDEC  REG3,BCOEFF,#5                      CONVERT RESULT TO DECIMAL
*
    LDA  #$FF                                   SENTINAL SIGNALING THAT ROOTS ARE REAL
    STA  CCOEFF,PCR
*
  ELSE                                          SIGN IS NEGATIVE; ROOTS IMAGINARY
    MCALL  REG1,ABS,REG1                        MAKE SIGN POSITIVE
    MCALL  REG1,SQRT,REG1                       CALCULATE  SQRT( B^2 - 4AC )
    DCALL  ACOEFF,MUL,TWO,REG2             CALCULATE  2A
    DCALL  REG1,DIV,REG2,REG1              CALCULATE  ( SQRT( B^2 - 4AC ))/2A
*
    DCALL  BCOEFF,DIV,REG2,REG2            CALCULATE  -B/2A
    MCALL  REG2,NEG,REG2
*
    BINDEC  REG1,BCOEFF,#5                 CONVERT  -B/2A TO DECIMAL
    BINDEC  REG2,ACOEFF,#5                 CONVERT  ( SQRT( B^2 - 4AC ))/2A
*
    CLR  CCOEFF,PCR                        SENTINAL SIGNALING IMAGINARY ROOTS
*
  ENDIF
*
*
  NOP
  NOP
```

APPENDIX C
DETAILED DESCRIPTION OF OPERATIONS

C.1 INTRODUCTION

This appendix contains detailed algorithmic information for each operation. Some implementation is also given to help explain how the MC6839 Floating Point ROM operates.

C.1.1 Argument Type Matrix

In order to speed up execution of the operations, tables are used to define special actions required for most operations. That is, most operations require special handling of values such as +0, −0, infinities, etc. For monadic operations, the type of arg2 is used to index into a one dimensional table. The index is determined by the type of argument.

		Index
	Normalized	0
	Zero	2
arg2	Infinity	4
	NAN	6
	Not Normalized	8

For dyadic operations, the type of both arguments determine the index.

		arg2				
		Normalized	Zero	Infinity	NAN	Not Normalized
	Normalized	00	02	04	06	08
	Zero	10	12	14	16	18
arg1	Infinity	20	22	24	26	28
	NAN	30	32	34	36	38
	Not Normalized	40	42	44	46	48

The index is used by each operation to jump indirectly through a table of values that specifies the offset from the start of the ROM to the routine to be executed.

C.1.2 Reading The Matrix Table

Argument type matrix tables are used in the discussion of each operation which follows. An entry in the table that contains "arg1" or "arg2" means that the operation will return that argument as the result and that no other processing is necessary. A letter in the matrix indicates that the operations specified in the paragraph with that letter will be executed to calculate the result. An example table and explanation is given below:

		Z
	Normalized	a
	Zero	arg2
arg2	Infinity	c
	NAN	arg2
	Not Normalized	b

In this example, if the input argument (arg2) is normalized, infinity, or not normalized, then refer to a, b, or c (of that particular paragraph) respectively. If the input argument is a NAN or zero, then return that NAN or zero (arg2) as the result.

In the following operation, description "Z" is used to represent the floating point result of an operation and "I" is used to represent an integer result.

If a trapping NAN is one of the operands (arg1 and arg2) and the invalid operation trap is enabled, then an invalid operation (=5) trap will be taken before the operation begins and, hence, the matrix table will not be used. Trapping NANs can be used by the user to create new or special data types or to provide special handling.

If "NAN" appears in the table as the result, it implies that a new NAN is created. The MC6839 Floating Point ROM will return the address of the instruction immediately following the operation that caused the NAN to be generated. Since the NAN is a new NAN, the "d" (double NAN), and "t" (trapping NAN) bits will be set to zero. See Section 2 for NAN details.

The final step for most arithmetic operations where the operands are well behaved includes checking for underflow, invalid operation, rounding, and overflow. In the operation descriptions, the following functions and procedures are used in algorithms without detailed explanations. For clarification, calls to these procedures and functions are always in upper case in the description of the operations. The functions and procedures used are:

CKINVALID	Check for invalid result.
OVERFLOW	Function. Returns true if overflow occurred.
UNDERFLOW	Function. Returns true if underflow occurred.
OVFL_NO_TRAP	Handles overflow when traps are disabled.
SUB_BIAS	Handles overflow when traps are enabled by subtracting a bias.
UNFL_NO_TRAP	Handles underflow when traps are disabled.
ADD_BIAS	Handles underflow when traps are enabled by adding a bias.
ROUND	Does correct rounding.

Detailed descriptions of the algorithms used for these functions and procedures are in Appendix D.

C.2 ADD (FADD), SUBTRACT (FSUB)

Z = arg1 + arg2; Z = arg1 + (– arg^2)
Opcode = \$00 (FADD)
Opcode = \$02 (FSUB)

		arg2				
		Normalized	Zero	Infinity	NAN	Not Normalized
	Normalized	b	b	arg2	arg2	b
	Zero	b	a	arv2	arg2	b
arg1	Infinity	arg1	arg1	c	arg2	arg1
	NAN	arg1	arg1	arg1	m	arg1
	Not Normalized	b	b	arg2	arg2	b

a.

		arg2	
		–0	+1
	–0	–0	d
arg1	+0	d	+0

d = +0 in rounding modes RN, RZ, RP
–0 in rounding mode RM

b.

```
1) Align binary points of arg1 and arg2 by unnormalizing the operand with the smaller
   exponent until the exponents are equal. Note if both operands are unnormalized.
2) Add the operands in internal form.
3) If arithmetic overflow occurs, right shift fraction one bit and increment exponent.
4) If all bits of the unrounded result are zero, then
             sign (Z) = + in rounding modes RN, RZ, RP
             sign (Z) = − in rounding mode RM
             If either arg1 or arg2 was normalized after step 1, then exponent
             (Z) = most negative value (i.e., true zero).
   Else ("Not all bits are zero")
             If, after step1, both operands were unnormalized,
             then go to step 5.
             else
             Normalize the result, if necessary, by shifting left while decrementing
             the exponent until n = 1.
             Zero or s may be shifted into r from the right.
5) If UNDERFLOW then
       if trap enabled then
            ADD_BIAS
            ROUND
       else
            UNFL_NO_TRAP
       endif
   else
       ROUND
       CKINVALID
     If OVERFLOW then
                  if trap enabled then
                       SUB_BIAS
                  else
                       OVERFL_NO_TRAP
                  endif
     endif
   endif
```

c. If affine mode:

		arg2	
		+ Infinity	− Infinity
arg1	+ infinity	+ infinity	c1
	− Infinity	c1	− Infinity

c1. Signal invalid operation = 2. Z = NAN

If projective closure mode, return NAN and signal invalid operation = 8.

m. Return arg2 but set the "d" bit in the NAN to indicate that this is a "double" NAN.

C.3 MULTIPLY (FMUL)

Z = arg1 × arg2
Opcode = $04

The sign of Z is the "exclusive OR" of the signs of arg1 and arg2.

		arg2				
		Normalized	Zero	Infinity	NAN	Not Normalized
	Normalized	a	0*	inf*	arg2	a
	Zero	0*	0*	b	arg2	0*
arg1	Infinity	inf*	b	inf*	arg2	inf
	NAN	arg1	arg1	arg1	m	arg1
	Not Normalized	a	0*	inf*	arg2	a

*Sign determined by arg1 "exclusive OR" arg2

a.

```
1) Generate sign and exponent. Multiply the significands in internal form.
2) If arithmetic overflow occurs, then right shift the significand one bit and increment the exponent.
3) If UNDERFLOW then
        if trap enabled then
            ADD_BIAS
            ROUND
        else
            UNFL_NO_TRAP
        endif
   else
        ROUND
        CKINVALID
        Iv OVERFLOW then
                if trap enabled then
                    SUB_BIAS
                else
                    OVERFL_NO_TRAP
                endif
            endif
        endif
```

b. Signal invalid operation = 9. Z = NAN.

m. Return arg2 but set the "d" bit in the NAN to indicate that this is a "double" NAN.

C.4 DIVIDE (FDIV)

Z = arg1/arg2 with sign of Z equal to the "exclusive-OR" of the signs of arg1 and arg2.
Opcode = $06

		arg2				
		Normalized	Zero	Infinity	NAN	Not Normalized
arg1	Normalized	c	a	0*	arg2	b
	Zero	0*	b	0*	arg2	0*
	Infinity	inf	inf*	b	arg2	inf*
	NAN	arg1	arg1	arg1	m	arg1
	Not Normalized	c	a	0*	arg2	b

*With correct sign

a. Signal Division by zero.
Z = infinity with correct sign

b. Z = NAN. Signal invalid operation = 4.

c.

```
1) Generate sign and exponent. Divide the significands in internal format.
2) If n = 0, then left shift significand one bit and decrement exponent. S need not par-
   ticipate in the left shift. A zero or s may be shifted into r from the right.
3) If UNDERFLOW then
          if trap enabled then
               ADD_BIAS
               ROUND
          else
               UNFL_NO_TRAP
          endif
   else
          ROUND
          CKINVALID
        if OVERFLOW then
                    if trap enabled then
                         SUB_BIAS
                    else
                         OVFL_NO_TRAP
                    endif
          endif
   endif
```

m. Return arg2 but set the "d" bit in the NAN to indicate that this is a "double" NAN.

C.5 REMAINDER (FREM)

$Z = arg1 - arg2 \times n$
Where n = integer part of arg1/arg2 in round nearest.
Opcode = $08

		arg2				
		Normalized	Zero	Infinity	NAN	Not Normalized
	Normalized	b	a	arg1	arg2	a
	Zero	arg1	a	arg1	arg2	a
arg1	Infinity	a	a	a	arg2	a
	NAN	arg1	arg1	arg1	m	arg1
	Not Normalized	b	a	arg1	arg2	a

a. Signal invalid operation = 10. Set Z to NAN.

b. Create number of integer bits in quotient "n" as:

```
        n = exp1 - exp2 - 1
  Generate "n" quotient bits, leaving raw remainder "r."
  If r > arg2/2
  then
      remainder = r - arg2
  else
      remainder = r.
  Normalize remainder
  If UNDERFLOW then
      If trap enabled then
          ADD_BIAS
          ROUND
      else
          UNFL_NO_TRAP
      endif
  else
      ROUND
      CKINVALID
    If OVERFLOW then
          If trap enabled then
              SUB_BIAS
          else
              OVFL_NO_TRAP
          endif
    endif
endif
```

m. Return arg2 but set the "d" bit in the NAN to indicate that this is a "double" NAN.

C.6 SQUARE ROOT (FSQRT)

Z = SQRT (arg2)
Opcode = $12

		Z
	Normalized	a
	Zero	arg2
arg2	Infinity	c
	NAN	arg2
	Not Normalized	b

a.

1) For a positive normalized number: compute Z = SQRT (arg2) to the number of bits required to produce a correctly rounded result. To round correctly in all cases, calculate two more bits of Z than the precision of the destination. ROUND as in Appendix C.
2) For negative normalized numbers: signal invalid operation = 1; Z = NAN.

b. Signal invalid operation = 1; Z = NAN.

c.

1) For projective mode signal invalid operation – 1; Z = NAN.
2) In affine mode, for plus infinity, set Z = arg2. For minus infinity, signal invalid operation = 1; Z = NAN.

C.7 INTEGER PART (FINT)

Z = Integer part of arg2
Opcode = $14

		Z
arg2	Normalized	a
	Zero	arg2
	Infinity	arg2
	NAN	arg2
	Not Normalized	a

a.

1) If arg2 has no fraction bits in its signficand, then set Z to arg2. This occurs if the exponent is so large that no fraction bits exist, for example, in single precision a number with an unbiased exponent greater than or equal to 23.
2) If arg2 has fraction bits, right shift the arg2 significand, while incrementing the exponent, until no bits (zero or nonzero) of the fractional part of arg2 lie within the rounding precision in effect. When this occurs, the unbiased exponent will be:

single	23
double	52
extended	63

3) ROUND as specified in Appendix D.
4) If all significand bits are zero, then Z = 0 with the sign of Z, otherwise, normalize Z. Zero or s is shifted into g from the right since s = 0 after rounding.

C.8 ABSOLUTE VALUE (FAB)

Z = |arg2|
Opcode = $1E

		Z
arg2	Normalized	a
	Zero	a
	Infinity	a
	NAN	arg2
	Not Normalized	a

a. Z = arg2 with zero (plus) sign.

C.9 NEGATE (FNEG)

Z = −arg2
Opcode = $20

		Z
	Normalized	−arg2
	Zero	−arg2
arg2	Infinity	−arg2
	NAN	arg2
	Not Normalized	−arg2

C.10 COMPARE (FCMP, FTCMP, FPCMP, FTPCMP)

CC = arg1 − arg2

Compare arg1 to arg2 and set condition codes accordingly or generate a true/false value for a predicate.

The four versions of compare are:

FCMP (Opcode — $8A) — Compare arg1 with arg2 and set the condition codes. Do not trap on unordered unless the trap on the unordered bit (UNOR) is set in the enable byte of the FPCB.

FTCMP (Opcode — $CC) — Compare arg1 with arg2 and set the condition codes. Trap if the unordered condition occurs regardless of the state of the UNOR bit in the enable byte of the FPCB.

FPCMP (Opcode — $8E) — Compare arg1 with arg2. Either affirm or disaffirm an input predicate. Do not trap on unordered unless the UNOR bit is set in the enable byte of the FPCB. For register calls, the Z-bit in the condition code register is set to 1 for affirm or true, and set to 0 for disaffirm or false. For stack calls, a byte of zeros is pushed on the top of the stack for true and a byte of ones ($FF) is pushed for false. Predicate compares are used often by HLLs when evaluating the conditional expression in control statements like IF.

FTPCMP (Opcode — $D0) — Compare arg1 with arg2. Either affirm or disaffirm an input predicate. Store the true or false indication in the result. Trap if the unordered condition occurs regardless of the state of the UNOR bit in the enable byte of the FPCB.

Since a compare allows different precision arguments, the X-register, on the call, contains a parameter specifying the precision of arg1 and arg2 and the predicate, if this is a predicate call. The format of the X-register is:

15		11	Predicates		8	7	6 4	3	2 0
000	≠	>	=	<	u	0	arg1	0	arg2

Where arg1 or arg2 is defined:

000	Single
001	Double
010	Extended
011	Unused (defaults to extended)
100	Unused (defaults to extended)
101-111	Undefined

The predicates are >, =, ≠, <, and unordered or the combinations: ≥ or ≤.

Once the arguments have been expanded into internal format, the following comparisons are made with the internal values.

		arg2				
		Normalized	Zero	Infinity	NAN	Not Normalized
	Normalized	a	a	b	f	d
	Zero	a	g	b	f	d
arg1	Infinity	c	c	e	f	c
	NAN	f	f	f	f	f
	Not Normalized	d	d	b	f	d

STEP 1.

a.

		arg2	
		+	−
arg1	+	*	>
	−	<*	

*Compare magnitudes of arg1 and arg2. Go to step 2.

b. If affine mode, then
If arg2 = + infinity then < else >. Go to step 2.
If projective mode, signal unordered. Go to step 2.

c. If affine mode, then
If arg1 = + infinity, then > else <. Go to step 2.
If projective mode, then signal unordered. Go to step 2.

d. Normalize one or both of the input arguments.
Set unnormal zeros to true zeros.
If both arguments are zero, then go to "g" else go to "a."

e. In projective mode set to equal. In affine mode:

		arg2	
		+ Inf.	− Inf.
arg1	+ Inf.	=	<
	− Inf.	<	=

go to step 2.

f. Set unordered. Go to step 2.

g. Set to equal. Go to step 2.

STEP 2.

1) If condition codes are to be returned (FCMP or FTCMP), then set the returned condition code bits in the following patterns:

	N	Z	V	C
>	0	0	0	0
<	1	0	0	0
=	0	1	0	0
Unordered	0	0	0	1

This allows the following *signed* branches to be taken immediately following the return from FCMP or FTCMP.

Condition	Branch	Test for Branch
>	(L)BGT	(not [N ⊕ V]) and (not Z) = 1
≥	(L)BGE	not (N ⊕ V) = 1
<	(L)BLT	not (N ⊕ V) = 0
≤	(L)BLE	(not [N ⊕ V]) and (not Z) = 0
=	(L)BEQ	Z = 1
≠	(L)BNE	Z = 0
Unordered	(L)BCS	C = 1

If CMP and the unordered trap is disabled, a BCS should immediately follow the call and precede any of the other branches:

```
CMP     arg1, arg2
BCS     unordered
BGE     label
```

If unordered occurred, then set the UNOR bit in the status byte of the FPCB so that the trap will be taken during post processing.

2) If a predicate is to be returned, then:

less than affirms:	< ≠
equal affirms:	= ≤ ≥
greater than affirms:	> ≥ ≠
unordered affirms:	unordered ≠
less than disaffirms:	= ≥ > unordered
equal disaffirms:	< > ≠ unordered
greater than disaffirms:	= ≤ < unordered
unordered disaffirms:	< ≤ = ≥ >

The result returned for affirmed is a zero byte and for disaffirmed it is a minus 1 or $FF byte for a stack call. For a register call, Z = 1 iff affirm.

If unordered occurred, then set the UNOR bit in the status byte of the FPCB so that the trap will be taken during post-processing. Additionally, if the predicate is ≠, then set the result to true to give the user a test for a NAN; i.e., If A≠A returns true, then "A" is a NAN.

```
If unordered and TPCMP or PCMP then
    If ≠ then
        set result true
    else if not (= or unordered) then
            signal unordered and invalid operation = 6
            so that a trap will be taken during
            post-processing
    endif
endif
```

C.11 FLOATING TO BINARY INTEGER (FFIXS, FFIXD)

I = INTEGER (arg2)
  I = 16 bit signed integer for FFIXS (Opcode = $16)
  I = 32 bit signed integer for FFIXD (Opcode = $18)

The resultant integer is stored on the *internal* stack in the first 2(4) bytes of the fraction for the result with the lower address containing the most significant byte.

		I
arg2	Normalized	d
	Zero	0
	Infinity	a
	NAN	b
	Not Normalized	d

STEP 1.

a. **Set V bit in returned condition code register and integer overflow bit in status. Set I as shown below:**

short positive	32767
short negative	−32768
long positive	2,147,483,647
long negative	−2,147,483,648

b. **Signal invalid operation = 3; return I = address of the instruction following the call to the floating point package.**

c. **If arg2 is not an integer, then call FINT to convert it to an integer. Convert arg2 to a binary integer and return it to the destination. If the integer exceeds the size of the destination, then go to "a" above.**

STEP 2.

Set the Z and N bits in the returned condition codes (V will already be set if overflow occurred) according to the resultant integer.

C.12 BINARY INTEGER TO FLOATING (FFLTS, FFLTD)

Z = FLOAT (arg2)
    arg2 = 16 bit signed integer for FFLTS (Opcode = $24)
    arg2 = 32 bit signed integer for FFLTD (Opcode = $26)

The integer is stored on the internal stack in the first 2(4) bytes of the fraction for arg2.

a. Convert arg2 to floating representation. If arg2 cannot be represented exactly, then ROUND as described in Appendix D.

C.13 BINARY FLOATING TO DECIMAL FLOATING STRING (BINDEC)

Opcode = $1C

Required Functions and Tables.

For both BINDEC and DECBIN, several functions and tables are required. BINSTR and STRBIN are required as well as a function to find the log_base_10(X). BINSTR converts a binary floating integer to a signed unpacked BCD string. STRBIN converts a signed decimal unpacked BCD string to a binary floating integer. Fortunately, the log_base_10(X) can be derived from:

log_base_2(X) x log_base_10(2).

Also, the log_base_10(X) need only be calculated to the nearest integer. Fortunately, this can be accomplished by noting that log_base_2(X) is approximately equal to the unbiased exponent of X. A table of the powers of 10 (in internal format) will be needed. This table need not contain all powers of 10 as some can be derived from the others. Negative powers shall be obtained by dividing by the corresponding positive powers instead of multiplying. The following 31 values (to full internal accuracy) will be required in the table:

10^0
10^1
•
•
•
•
10^{26}
10^{27}
10^{54}
10^{108}
10^{216}

Argument Requirements

For register calls, the U register contains the constant "k" that specifies the number of significant digits desired.

For stack calls, the input stack looks like:

```
       |−n−|1|−2−|Bytes
 | // | arg2 | k | FPCB |
                    S
```

Where n = 4, 8, or 10 bytes
S = hardware stack pointer

The return string is a standard BCD string as defined in Section 2 (BCD Strings).

Conversion Process

Given binary floating point number arg2 and an integer k (passed in arg1) with $1 \le k \le 9$ for single precision and $1 \le k \le 17$ for double precision, we can compute the signed decimal strings I and E such that I has k significant digits and interpreting I and E as the integers they represent:

$$arg2 = I \times 10^{(E+1-k)} = sd\ ddddddd \times 10^{E}$$

where s is the sign of arg2 and the ds are the k decimal digits of I.

The size of I and E are defined by the output string generated by BINSTR for the supported precisions.

STEP 1.

		String
arg2	Normalized	c
	Zero	b
	Infinity	a
	NAN	d
	Not Normalized	c

a. For + infinity, deliver a nondecimal string with se = $0A and the remaining bytes equal to zero.

   For − infinity, deliver a nondecimal string with se = $0B and the remaining bytes equal to zero.

b. I = string of "+0" or "−0"; E = String of "0." Go to step 2.

c.

1) Remember sign of arg2. Let p = absolute_value (arg2). Remember whether arg2 is normalized.

1a) If arg2 is unnormal zero, then go to b.

2) If p is not denormalized, compute q = long_base_10(p); otherwise let q = log_base_10 (smallest normalized number).

3) Remember the current rounding mode. Compute:

   $v = \mathrm{FINT}(q) + 1 - k$

   with rounding mode RZ.

4) Compute $w = \mathrm{FINT}(p/10^v)$ using powers of 10 from the tables with rounding mode RN. Restore original rounding mode.

5) Adjust w for special cases:

   i. If $w \geq (10^k) + 1$, then increment v and go to 4.

   ii. If $w = 10^k$, then increment v, divide w by 10 (exactly) and go to 6.

   iii. If $w \leq 10^{(k-1)} - 1$ and arg2 was *normalized* in step 1, then decrement v a n d

   go to 4.

6) I = BINSTR (w with sign of arg2); E = BINSTR (v).

d. Deliver a nondecimal string with se = $0C followed, in the exponent field, by the unpacked hex address where the NAN was created.

STEP 2.

Return a BCD string as defined in Section 2 (paragraph 2.5) with p = 0.

C.14 DECIMAL FLOATING STRING TO BINARY FLOATING (DECBIN)

Opcode = $22

Required Funtions and Tables

For both BINDEC and DECBIN, several functions and tables are required. BINSTR and STRBIN are required. BINSTR converts a binary floating integer to a signed unpacked BCD string. STRBIN converts a signed unpacked BCD string to a binary floating integer. A table of the powers of 10 (in internal format) will be needed. This table need not contain all powers of 10 as some can be derived from the others. Negative powers shall be obtained by dividing by the corresponding positive powers instead of multiplying. Those required in the table are:

10^0
10^1
•
•
•
•
10^{26}
10^{27}
10^{54}
10^{108}
10^{216}

Argument Requirements

For stack calls the input stack looks like:

|BCD String	FPCB

S

For the format of the BCD string see Section 2 (BCD Strings Paragraph). The total size of the BCD string is 26 bytes.

The result for stack calls is on top of the stack.

For register calls:
X = result
D = FPCB
U = pointer to input BCD string

The input argument is a standard BCD string as defined in Section 2 (BCD Strings paragraph) where "p" is set to the number of fraction digits to the right of the decimal point.

Conversion Process

The number to be converted (arg2) can be thought of as a number of the form:

$$arg2 = sddddd.DDDDDDDDD \times 10^{E}$$

On entry, arg2 contains a pointer to a string as defined in paragraph 2.4. Let I = sddddd.DDDDDDDDD. arg2 contains an integer p that indicates how many digits of I are to the right of the decimal point such that:

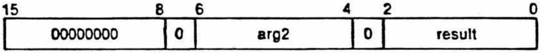
$$arg2 = I \times (10^{-P}) \times (10^{E})$$

```
1) Compute U = STRBIN (I) and w = binary_integer_of (E)

2) Compute result: $Z = u \times 10^{(w - p)}$

3) If UNDERFLOW then
        If trap enabled then
            Z = NAN
        else
            UNFL_NO_TRAP
        endif
   endif
   If OVERFLOW then
        If trap enabled then
            Z = NAN
        else
            OVFL_NO_TRAP
        endif
   endif
```

C.15 MOVE (MOV)

Move arg2→result
Opcode = $9A

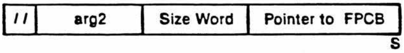
Since move allows for arguments of different precisions, it requires that the precisions be specified by the calling program in a size word parameter. The U-register is used to hold the size word in register calls. In stack calls the size word is pushed onto the top of the stack above arg2 but before the pointer to the FPCB. The format of the size word is:



Where arg2 (source) or result (destination) is defined:

000	Single
001	Double
010	Extended
011	Extended round to single
100	Extended round to double
101-111	Illegal

For stack calls the calling stack looks like:



Move allows the input argument and the result to be of different precisions. For stack calls the MOV operations transform the stack top from the source precision to the destination precision.

For moves where the precision of arg2 equals the precision of the result, the arguments will not be moved to the stack since no conversion is really needed; otherwise, the source argument (arg2) will be moved onto the internal stack. This is necessary since if the MOV were a conversion from a shorter to a longer precision value at the same address, the result might overwrite parts of the source before they have been read.

		Result (Destination)
	Normalized	a
	Zero	arg2
arg2	Infinity	arg2
(Source)	NAN	arg2
	Not Normalized	b

a.

```
If the destination is shorter than the source, then Z = arg2.
If UNDERFLOW then
     If trap disabled then
          UNFL_NO_TRAP
     else
          deliver to the trap handler the result in internal
          format but rounded to the precision of the destination.
     endif
endif
ROUND to the precision of the destination
CKINVALID
If OVERFLOW then
     If trap disabled then OVFL_NO_TRAP
     else
          deliver to the trap handler the result in internal
          format but rounded to the precision of the destination.
     endif
endif
If the destination field is wider than the source, then the
move is exact.
```

b.

```
If (single to double) then Invalid operation = 15
Else If (source = ext) and (dest < > ext) and (not denormalized)
          Invalid operation = 16
     else
          go to a .
     endif
endif
```

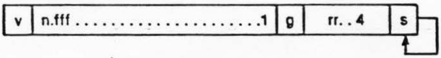
APPENDIX D
ROUNDING AND EXCEPTION CHECKING ROUTINES

D.1 INTRODUCTION

The following routines and functions are used after the arithmetic operations to round and to detect error conditions.

D.2 ROUNDING

Rounding is accomplished using the v, 1, g, 4, and s bits as defined in the internal formats. In general, the *significand* of the number to be rounded looks like:



where:

v = overflow bit
n = 1 (explicit 1.0)
f = fraction
l = 1s bit of fraction
g = guard bit
r = round bits
s = sticky bit

The s bit is the logical "OR" of all the bits to the right of the r bits. Thus during the calculation stage of an arithmetic operation, any nonzero bits which are generated that are to the right of the r bits show as a 1 in the s bit. If the precision mode specified in the control byte of the FPCB is 6 or 7, then rounding should be to the precision specified. The following algorithm is used to round the result to 1 of the four rounding modes:

```
begin
If g = s = r = 0 then result is exact
```

```
  else (not all zero)

        set inexact result flag bit

        case rounding mode of
        RM: if sign = 1 then add 1 to l
        RP: if sign = 0 then add 1 to l
        RN: if g = 1 and r = s = 0 then
              if 1 - 1 then add 1 to g
        else add 1 to g
  (RZ falls through)
  end case
  if v = 1 then
        shift right
        increment exponent
  endif
  set g = r = s = 0
endif
end
```

where:
- RM = round to minus
- RP = round to plus
- RN = round to nearest
- RZ = round to zero

D.3 EXCEPTION HANDLING

D.3.1 Invalid Operation

Invalid operation encompasses problems arising in a variety of arithmetic operations; it is the blanket covering those errors which do not occur frequently enough or are not important enough to merit their own fault condition. The result to be delivered by an invalid operation is a NAN.

D.3.2 Underflow

In a general sense, underflow is the condition that exists when an arithmetic operation creates a result that is too small to be represented in the normal memory format for the destination. If the trap is enabled when underflow occurs, the user can determine what he wants to do. The actual result of the operation will not be lost since the internal formats are capable of representing the underflowed number. If no trap is enabled, the floating point package will automatically denormalize the result as discussed previously (gentle underflow). In the case of trap enabled, but the trap wishes to return the result, the delivered exponent will be the result of *adding* a bias adjust for each precision as shown below. This bias adjusts the exponent so that it will contain a number in the middle of the exponent range.

Bias adjust for overflow/underflow	
Single	192
Double	1536
Extended	24576

D.3.3 Overflow

In a general sense, overflow is the condition that exists whan an arithmetic operation creates a result that is too large and cannot be represented in the normal memory format for the destination.

Overflow is handled much more harshly and quickly than underflow and with a corresponding loss of information. The number system chosen is slightly biased towards underflow for this reason. If a trap is to be taken on overflow, then a bias is *subtracted* from the exponent to wrap it around into the range of valid exponents. The bias for each precision is given above. If no trap is to be taken, then a suitable result is returned.

D.3.4 Division by Zero

This exception occurs when a normal zero divisor occurs in a division. If the divisor is normal zero and the dividend is finite and nonzero, the default result is infinity with the correct sign. If the division by *zero* trap is enabled, then it is taken and the default result is returned.

D.3.5 Inexact Result

If the rounded result of an operation is not exact or if it overflows to infinity, then the inexact exception shall be signaled unless the result would be an invalid result. If the trap is enabled, it is taken; otherwise, the rounded result or the infinity that resulted from overflow shall be the default value returned.

D.3.6 Integer Overflow

This occurs when a large floating point number is converted to an integer that cannot be represented in the destination. If the trap is enabled, it will be taken and the caller can "fix" the result. If no trap is enabled, the largest positive or negative integer is returned.

D.3.7 Unordered

Unordered occurs when a comparison is made between a NAN and anything else or when infinity is compared to anything except itself in projective closure.

D.3.8 Error Trap Handling

When an error trap occurs, the post processing code passes control to the location specified in the FPCB vector with the U register pointing to the stack frame. The trap routine may then modify the result on the stack frame or it may choose to create a new result and store the result directly in memory. If the result on the stack frame is modified, the routine must remember that this number is in internal format. On return from the error trap routine, if the C-bit is set, the result in the stack frame will be moved to memory. If the C-bit is cleared, no result will be delivered to the destination.

On entry to the trap routine, the U-register will contain the pointer to the current stack frame. The temporary status stored in the stack frame should be used to determine the status of the last operation. If more than one bit is set in the status register, the floating point package will determine which trap should have precedence as discussed in Section 4 (Exception Modes pargraph). In the case where the highest precedence exception does not have its trap enabled, then the next highest precedence will be checked, etc., until the highest precedence enabled trap, if any, is found.

D.4 ALGORITHMS FOR EXCEPTION PROCESSING

The following are the algorithms implemented in the MC6839 to check for the process exceptions.

D.4.1 Check for Invalid (CKINVALID)

```
        procedure CKINVALID
        begin
            if not infinity or true zero then
                if destination precision is single or double then
                    if result is denormalized
                        fix exponent for denorm result
                    else
                      if not normalized then
                          iop = 16
                      endif
                  endif
              endif
          endif
      end
```

D.4.2 Test for Overflow (OVERFLOW)

```
function OVERFLOW
begin
      (*test for overflow*)
      if the rounded result is finite and its exponent is too large for the destination
      then OVERFLOW: = true; set overflow flag
      else OVERFLOW: = false
end
```

D.4.3 Overflow With Traps Disabled (OVFL NO TRAP)

```
procedure OVFL__NO__TRAP
begin
    set inexact result flag;
    If rounding mode is round to  – infinity
    then
        clear overflow flag
        If result is positive
        then
            If result is normalized
            then
                deliver largest positive
                number to destination
            else
                deliver significand and largest
                exponent to destination
            endif (*result is normalized*)
            else (*result is negative*)
                deliver — infinity to destination
            endif; (*result is positive*)
        endif; (*rounding mode is to  – infinity*)
        If rounding mode is round to  + infinity
        then
            clear overflow flag
            If result is negative
            then
            If result is normalized
            then
                deliver largest negative number to destination
            else
                deliver significand and largest
                exponent to destination
            endif; (*result is normalized)
        else (*result is positive*)
            deliver + infinity to destination
        endif; (*result is negative*)
    endif; (*rounding mode is to  + infinity*)

        If rounding mode is to nearest or to zero
        then
            deliver properly signed infinity to destination
        endif; (*round to nearest or to zero*)
end
```

D.4.4 Subtract Bias on Overflow (SUB BIAS)

```
procedure SUB__BIAS
begin
    subtract bias (from table shown in paragraph D.3.2) from exponent
end
```

D.4.5 Test for Underflow (UNDERFLOW)

```
function UNDERFLOW
begin
    if exponent not = $8000 (true zero)
        if exponent is too small for destination format
        then UNDERFLOW: = true; set underflow flag
        else UNDERFLOW: = false
        endif
        else UNDERFLOW: = false
    endif
end
```

D.4.6 Add Bias on Underflow (ADD BIAS)

```
Procedure ADD__BIAS
begin
    add bias (from table shown in paragraph D.3.2) to exponent
end
```

D.4.7 Underflow With Traps Disabled (UNFL NO TRAP)

```
Procedure UNFL__NO__TRAP
begin
    denormalized unrounded result;
    ROUND denormalized result;
    If fraction=0 then set to true zero
    deliver denormalized and rounded (only once)
        result to destination;
    If rounding mode is either round
    to -Infinity or to +Infinity
    then
        clear underflow flag;
    endif; (*round to infinity*)
end
```

APPENDIX E
PROGRAM DETAILS AND STACK FRAME DESCRIPTION

E.1 PRE-PROCESSING/POST-PROCESSING

All operations undergo a pre-processing step where the calling arguments are moved from their present locations to an internal stack frame and a post-processing step where the results are returned from the stack frame. In general, the operation of any function looks like:

```
save caller registers on the stack
determine function opcode
if register call then
     initialize stack frame for register call
     move argument(s) into internal stack frame
     if no input arg is a trapping NAN, then do function
     check for traps
     if (no traps) or (trap handler wants result returned)
          then move result to user
     cleanup stack
else (stack call)
     adjust stack if necessary*
     initialize stack frame for stack call
     move argument(s) to internal stack frame
     If no input arg is a trapping NAN, then do function
     check for traps
     if (no traps) or (trap handler wants result returned)
          then move result to stack top
     cleanup stack
     adjust stack if necessary*
endif
restore caller registers
return
```

E.1.1 Stack Frame

Upon execution, the floating point package immediately reserves an area on the active hardware stack for its execution time local variables. Once this "stack frame"is initialized, it is used by all the modules of the program. The stack frame area is released on exit from the call.

*For stack calls, adjusting the stack before or after processing may be necessary if the total size of the input arguments is not equal to the size of the output argument.

The user may need to know the details of the stack frame if he plans to write trap routines to manipulate results in the internal format. Appendix F contains information on the internal format for floating point numbers on the stack frame.

Figures E-1 and E-2 are examples of the stack frame configuration. Figure E-1 is for a register call and Figure E-2 is for a stack call. Notice that from mnemonic "TYPE 1" down to the bottom of the stack, the two stack frames are identical. This allows the actual operation routines to be identical regardless of the type of call. During execution of the operation, the U-register always points to the bottom of the stack frame.

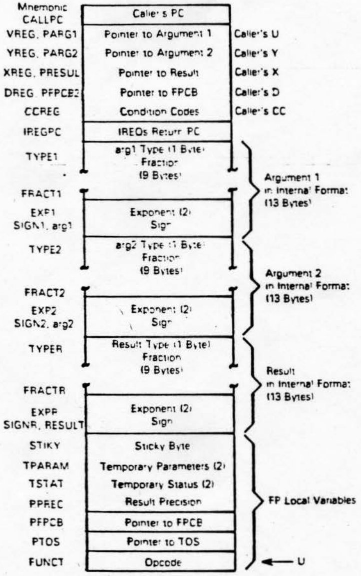


Figure E-1. Register Call Stack Frame

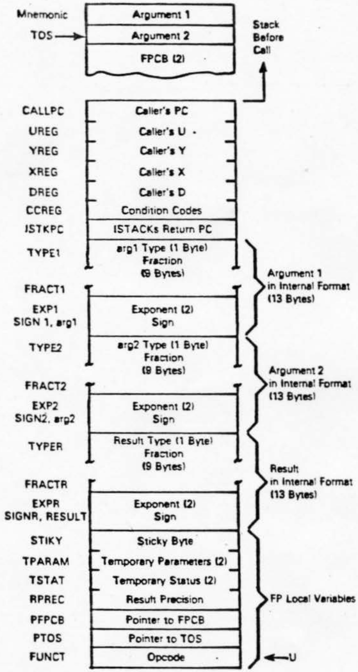


Figure E-2. Stack Call Stack Frame

Special handling of the stack frame occurs for BCD string conversions. The actual BCD strings will not be moved onto the stack frame. A pointer to the strings will be stored in the stack frame instead. The operations will access the strings directly in the user's memory or stack.

Special handling also occurs for MOV with equal precision arguments. In this case the stack frame will not normally be created since it would slow down rather than speed up the operation. However, a stack frame will be created for a MOV with different precision arguments. This enables the trap handler to do intelligent processing.

All operations have a result except for nonpredicate compares which only return with the appropriate bits set in the condition code register. Predicate compares only return a 1-byte "yes" or "no" as the result.

For operations that convert from an integer to floating point, the integer will be stored in the fraction of argument 2. For operations that convert from floating point to an integer, the resulting integer is stored in the fraction of the result.

Note that space for argument 1 is reserved on the stack frame even if the call is monadic. This insures consistent use of subroutines to manipulate arguments on the stack.

If bit 3 (NRM) of the control byte in the FPCB is set, then all denormalized (not unnormalized) numbers will be normalized during the move onto the stack frame.

E.1.2 FP (Floating Point) Variables

E.1.2.1 POINTER TO FPCB (PFPCB). This word contains the address of the start of the FPCB to be used by this call.

E.1.2.2 TOS (TOP OF STACK) POINTER (PTOS). For stack calls, this word points to the top floating point argument on the stack when the floating point package was initially called. This may not be the address just above the caller return PC, since it might have been necessary to reserve some empty stack space when the result of a function uses more bytes on the stack than the input arguments. Note that the pointer to the FPCB, as passed by the user, is always at PTOS-2.

E.1.2.3 TEMPORARY PARAMETERS (TPARAM). This temporary two byte location is used by DECBIN and BINDEC to store parameters. It is also used by calls to MOV or the compares to store the parameter word and may be used by other operations as a scratch location.

E.1.2.4 TEMPORARY STATUS (TSTAT). This temporary two byte status is used by the floating point package to generate status bytes of this operation. The first byte (lower address) has a format identical to the status byte in the FPCB. At the completion of the operation this temporary status is logically "ORed" into the existing status in the caller FPCB. The second byte contains a temporary byte that has the same format as the secondary status byte in the FPCB. At the completion of the operation, if an invalid operation occurred, this byte will be written into the secondary status.

E.1.2.5 RESULT PRECISION (RPREC). The index stored at this location defines the precision of the result.

Index	Precision
0	Single
2	Double
4	Extended
6	Extended Rounded Single
8	Extended Rounded Double

For compares, this location contains the index of arg2 instead of the result.

E.1.2.6 OPCODE (FUNCT). This byte contains the opcode picked up from the user's calling sequence. This is used by various subroutines. It also allows an error trap to determine what operation caused an error. Some bits in the opcode have special meaning:

bit 7 = 1 = Mixed size arguments (MOV, CMP)
bit 6 = 1 = Trap on unordered compare
bit 5 = 0 = Function number

E.1.2.7 STICKY BYTE (STIKY). This byte is used during arithmetic operations to "OR" all the least significant bits of an operation. The sticky byte is then used during rounding. Some sticky bits are also picked up by "ROUND" from the low order bits of the internal fraction.

E.1.2.8 ARGUMENT TYPE (TYPEx). A byte is reserved for each argument to indicate its type. The routine that initializes the stack frame initializes the values for TYPE 1 and TYPE 2. The values are:

0 = Normal, in range, normalized value
2 = Normal zero
4 = Infinity
6 = Not a number
8 = Not normalized

Note that an unnormalized zero will have an index = 8, ≠ 2.

This byte occupies the highest address of the fraction for each argument.

The type of the result may not be valid at the time of a trap.

E.1.2.9 SIGNX, EXPX, FRACTX. The bytes describe the fields in internal format numbers. Appendix F provides details of internal format numbers.

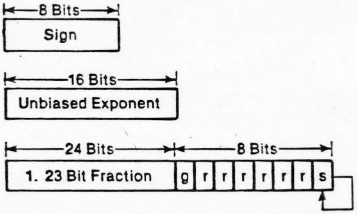
APPENDIX F
INTERNAL FORMATS

F.1 INTRODUCTION

The memory formats are chosen to provide the greatest amount of precision in the least amount of memory, whereas, the internal formats are selected to permit the easiest and fastest implementations of the desired operations. A caller to a floating point subroutine passes arguments in memory formats and receives the result in memory format; however, internally the floating point package converts to the internal formats, does the operation, and then converts the result back to memory format.

F.2 SINGLE INTERNAL FORMAT

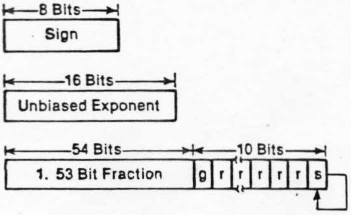
This format consists of 7 bytes:



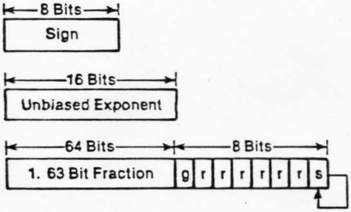
where:	
sign	= positive or negative byte containing the sign of the fraction. Only the most significant bit is defined: b7 = 0 = plus; b7 = 1 = minus
unbiased exponent	= twos complement exponent
g	= guard bit
r	= rounding bits
s	= sticky bit

The g, r, and s bits are used for rounding as described in Appendix D.

F.3 DOUBLE INTERNAL FORMAT



F.4 EXTENDED INTERNAL FORMAT



Note that single, double, and extended internal formats differ only in the number and location of the g, r, and s bits.

F.5 ZERO

Zero is represented by a number with the smallest unbiased exponent and a zero significand:

s	100. . . . 0000	0

F.6 INIFINITY

Infinity has the maximum unbiased exponent and a zero significand:

s	011111. . . . 11	0

F.7 NANS

NANs have the largest unbiased exponent and a nonzero significand. The operation addresses, "t" and "d," are implementation features and are defined in Section 2 (Not a Number paragraph).

d	011 1111	0	t	Operation Address	00000000

The operation address always appears in the 16 bits immediately to the right of the t bit.

F.8 INTERNAL UNNORMALIZED NUMBERS

Unnormalized numbers occur only in extended or internal format. Unnormalized numbers have an exponent greater than the internal formats minimum (i.e., they are not denormalized or normal zero) and the explicit leading bit is a zero. If the significand is zero, this is an unnormalized zero. Even though unnormalized and denormalized numbers are handled similarly in most cases, they should not be confused. Denormalized numbers are numbers that are very small (have minimum exponent) and hence have lost some bits of significance. Unnormalized numbers are not necessarily small (the exponent may be large or small) but the significand has lost some bits of significance, hence, the explicit bit and possibly some of the bits to the right of the explicit bit are zero.

s	> 100. . . 000	0. Significand

Note that unnormalized numbers cannot be represented (hence cannot exist) for single and double formats. Unnormalized numbers come into existence when denormalized numbers, in single or double formats, are represented in extended or internal formats.

APPENDIX G
BASIC LEVELS OF PRECISION

G.1 SINGLE PRECISION SPECIFICATION

Length in Bits	32
Fields:	
s = sign	1
e = exponent	8
f = significand	(1) + 23
Storage Format:	$\boxed{s}\boxed{e}\boxed{\quad f \quad}$
Interpretation of Sign:	
positive	0
negative	1
Normalized Numbers:	
interpretation of e	unsigned integer
bias of e	127
range of e	$0<e<225$
interpretation of f	1.f
relation to represent real numbers	$(-1)^2 \times 2^{(e-127)} \times 1.f$
Signed Zeros:	
e =	0
f =	0
Reserved Operands:	
Denormalized Numbers:	
e =	0
bias of e	126
interpretation of f	0.f
range of f	nonzero
relation to represent real numbers	$(-1)^s \times 2^{-126} \times 0.f$
Signed Infinities:	
e =	255
f =	0
NANs:	
3 =	255
f =	nonzero
interpretation of f	don't care

Ranges:	
maximum positive normalized	3.4×10^{38}
minimum positive normalized	1.2×10^{-38}
minimum positive denormalized	$1.4 \times ^{-45}$

G.2 DOUBLE PRECISION SPECIFICATION

Length In Bits	64
Fields:	
s = sign	1
e = exponent	11
f = significand	(1) + 52
Storage Format:	$\boxed{s \mid e \mid f}$
Interpretation of Sign:	
positive	0
negative	1
Normalized Numbers	
interpretation of e	unsigned integer
bias of e	1023
range of e	$0 < e < 2047$
interpretation of f	1.f
relation to represent real numbers	$(-1)^s \times 2^{(e-1023)} \times 1.f$
Signed Zeros:	
e =	0
f =	0
Reserved Operands:	
Denormalized Numers:	
e	= 0
bias of e	1022
interpretation of f	0.f
range of f	nonzero
relation to represent real numbers	$(-1)^s \times 2^{-1022} \times 0.f$
Signed Infinities	
e =	2047
f =	0
NANs:	
e =	2047
f =	nonzero
interpretation of f	don't care

Ranges:

maximum positive normalized 18×10^{307}
minimum positive normalized 2.2×10^{-308}
minimum positive denormalized 4.9×10^{-324}

G.3 EXTENDED PRECISION SPECIFICATION

Length in Bits	80
Fields:	
s = sign	1
e = exponent	15
j = integer part	1
f = significand	63
Storage Format:	| s | e | j.f |
Interpretation of sign:	
positive	0
negative	1
Normalized Numbers:	
interpretation of e	twos complement integer
bias of e	0
range of e	$-16384 \leq e < 16383$
interpretation of significand	j.f
relation to represent real numbers	$(-1)^s \times 2^e X j.f$
Signed Zeros:	
e =	-16384 (\$4000)
significand =	0
Reserved Operands:	
Denormalized Numbers:	
e =	-16384
bias of e	0
interpretation of significand	0.f
range of f	nonzero
relation to represent real numbers	$(-1)^s \times 2^{-16384} \times 0.f$
Signed Infinities:	
e =	16383 (\$3FFF)
significand =	0
NANs:	
e =	16383 (\$3FFF)
significand =	nonzero
interpretation of significand	don't care

Ranges:

maximum positive normalized	6×10^{4931}
minimum positive normalized	8×10^{-4933}
minimum positive denormalized	9×10^{-4952}

Table G-1. MC6839 Floating Point ROM Memory and Internal Data Format

Type / Precision	Normalized	Zero	Infinity	NAN	Denormalized	Unnormalized
Single Memory	1 8 23 S | Exp | Signif Bias = 127 = $7F 0 < Exp < $FF	1 8 S | 0 | 0	1 8 S | $FF | 0	1 8 1 16 6 d | $FF | t | Addr | d - Double NAN t - Trapping NAN	1 8 23 S | 0 | ≠0	Not Possible
Single Internal	8 S | **** 16 Unbiased Exp 24 8 1 Fract | S | r | S	8 S | **** 16 $8000 32 0	8 S | **** 16 $7FFF 32	8 d | **** 16 $7FFF 1 1 16 14 0 | t | Addr |	8 S | **** 16 $FF82 32 0 F (≠0)	8 S | **** 16 Unbiased Exp 32 0 F
Double Memory	1 11 32 S | Exp | Signif Bias = 1023 = $3FF 0 < Exp < $7FF	1 11 32 S | 0 | 0	1 11 32 S | $7FF | 0	1 11 1 16 32 d | $7FF | t | Addr |	1 11 32 S | 0 | ≠0	Not Possible
Double Internal	8 S | **** 16 Unbiased Exp 53 19 1 Fraction | S | r | S	8 S | **** 16 $8000 72 0	8 S | **** 16 $7FFF 72 0	8 d | **** 16 $7FFF 1 1 16 34 0 | t | Addr |	8 S | **** 16 $FC02 72 0 F (≠0)	8 S | **** 16 Unbiased Exp 72 0 F
Extended Memory	1 15 64 S | Exp | 1 Signif Bias = 0 $4000 < Exp < $3FFF − 16384 < Exp < 16383	1 15 64 S | $4000 | 0	1 15 64 S | $3FFF | 0	1 15 1 1 26 96 d | $3FFF | 0 | t | Addr |	1 15 64 S | $4000 | 0 F (≠0)	1 15 64 S | Exp | 0 F
Extended Internal	8 S | **** 16 Unbiased Exp 64 8 1 Fraction | S | r | S	8 S | **** 16 $8000 72 0	8 S | **** 16 $7FFF 72 0	8 d | **** 16 $7FFF 1 1 16 34 0 | t | Addr |	8 S | **** 16 $C000 72 0 F (≠0)	8 S | **** 16 Unbiased Exp 72 0 F

APPENDIX H
DEFINITIONS AND ABBREVIATIONS

This appendix defines several terms and abbreviations used in this manual which are peculiar to the MC6839 Floating Point ROM. Many of these definitions are also found in the *IEEE Proposed Standard for Binary Floating Point Arithmetic Draft 8.0.*

User — The user of a floating point system is considered to be any person, hardware, or program having access to and controlling the operations of the programming environment.

Binary Floating Point Number — A bit string characterized by three components: a sign, a signed exponent, and a significand. Its numerical value, if any, is the signed product of its significand and two raised to the power of its exponent. A bit string is not always distinguished from a number it may represent.

Exponent — That component of a binary floating point number which signifies the power to which two is raised in determining the value of the represented number. Occasionally, the exponent is called signed or unbiased exponent.

FP — An abbreviation for "floating point."

FPCB — An abbreviation for "floating point control block."

Biased Exponent — The sum of the exponent and a constant (bias) chosen to make the range of the biased exponent non-negative.

Significand — That component of a binary floating point number which consists of an explicit or implicit leading bit to the left of its binary point and a fraction field to the right of the binary point.

Fraction — The field of the significand that lies to the right of its implied binary point.

Normal Zero — The exponent is the minimum established for format and the significand is zero. Normal zero may have either a positive or negative sign. Only the extended format has any unnormalized zeros.

Denormalized — The exponent is the minimum established for the format, the explicit or implicit leading bit is a zero, and the number is not normal zero. To denormalize a binary floating point number means to shift its significand right while incrementing its exponent until it is a denormalized number.

Unnormalized — The exponent is greater than the minimum established for the extended format and the explicit leading bit is zero. If the significand is zero, this is an unnormalized zero.

Normalize — If the number is nonzero, shift its significand left while decrementing its exponent until the leading significand bit becomes one; the exponent is regarded as if its range were unlimited. If the significand is zero, the number becomes normal zero. Normalizing a number does not change its sign.

Double Rounding — Double rounding occurs if any single operation causes a result to be rounded more than once.

NAN — Not a number.

Sticky Bit — A status bit that, once set by the system as the result of some operation, remains set until explicitly cleared by the user. This feature relieves the user of the constraint of having to examine this bit in any particular time window.

Hardware Stack — The stack defined on the MC6809 by the S (or SP) register. This stack is also used by the hardware during subroutine calls and interrupts.

Floating Point Package — A package of subroutines that supports the basic capabilities required to do calculations with real numbers.

Internal Format — A format resembling extended format that is used by the MC6839 during calculations. It does not exist before the package is called nor does it exist after the package returns and it is only an intermediate format.