ORCA/M Floating Point Libraries

Macro and subroutine library support (with source) for single and double precision floating point with ProDOS ORCA/M.

By Mike Westerfield

The Byte Works Inc.

Overview

This package contains macro libraries that allow you to use floating point math from within your assembly language programs. The macros allow a variety of addressing modes, including absolute, immediate, indirect and stack addressing. in addition to the expected operations of add, subtract, multiply and divide, support is included for trig functions (sine, cosine, tangent and arctangent) as well as natural log, exponent, power, and a full range of data conversion macros. Input and output is accomplished via macros that provide for Pascal-like formatting of output. Coupled with a powerful library of floating point subroutines, you can easily put the power of both single and double precision floating point to practical use in your assembly language programs.

Serving both as a programming example and as a useful tool are two calculators, one for single precision and one for double precision.

Finally, source code for the subroutine libraries is provided so that you can move your programs to non-standard hardware configurations, or simply explore the subroutines.

Installation

In this package are three disks. As with other Byte Works programs, they are unlocked and copyable. Start by making backup copies of all of the disks. Two of the disks contain only source code, and can be ignored for now. These are the Single Precision Source and Double Precision Source. The last disk, Floating Point, contains the files we will use right away.

On the floating point disk are three types of files. The first are the macro libraries. The macros are in three files, all starting with SUP. They really don't need to be placed anywhere specific, but if you have a disk where you keep all of your other macros, move those files to the same place now. Also, be aware that some of the macros make use of macros in the standard ORCA macro library. When using MACGEN to build libraries for your programs, It will be convenient to have them all together.

Next are the subroutine libraries. These are in the two files F.FLOAT.A and G.DOUBLE.A. These should be moved to the disk (or directory) where you keep the subroutine libraries that came with the original ORCA/M package. Since the linker will only search one place for libraries, it is important that these files be put in the correct place. If you are using the ORCA/M disks in the configuration In which they were shipped, this would mean placing the library files on the /LIBRARY disk. Just as Important is the order in which the files appear. If you catalog the libraries, you will find that they all start with an alphabetic character followed by a dot. When the files are alphabetized, they are in the proper order. If they are not in the proper order, you will get link errors when trying to use the floating point libraries. Use the COMPRESS command to put them in order.

Finally, there are two files called CALCF and CALCD. These are the floating point calculators, described later.

Data Formats

These libraries include support for both single and double precision floating point numbers. In both cases, the IEEE floating point standard Is used for the format of-the numbers. Note that although the IEEE floating point standard is observed for format, the full standard is not implemented. This is because of the overhead involved. By conforming with the standard, but not fully implementing it, the subroutines have been kept smaller and faster than they would otherwise be.

Single precision numbers require four bytes of storage each. The first bit is a sign bit. The next eight bits are the displaced exponent, giving a number range from about IE-38 to 1E38. Numbers are always normalized, so the most significant bit of the mantissa is always one. Since it Is known, it is not actually coded - the remaining 23 bits of the number are bits 2 to 24 of the mantissa. This gives slightly more than seven decimal digits of accuracy.

Double precision numbers require eight bytes of storage each. Again, the first bit is the sign bit. The next eleven bits are the exponent, giving a number range from IE-308 to 1E308. An implied most significant mantissa bit gives 53 significant bits in the mantissa, resulting in nearly sixteen digit accuracy.

The Calculators

Two calculators are provided, one for single precision and one for double precision. They work the same, and will be described together here. The single precision calculator is called CALCF, and the double precision calculator is called CALCD.

When you run one of the calculators, you get a copyright message and the familiar pound sign prompt. Incidentally, you can run the calculators from ORCA/M or under ProDOS's BASIC.SYSTEM since they do not access the disk, they can even be moved to DOS and executed there. Once the prompt appears, the program is waiting for a command. The commands look a great deal like BASIC statements entered from the immediate execution mode. You can type a question mark followed by an expression to get the program to write an answer immediately. For example, try typing

```
? 1+1
```

In addition, the calculator supports twenty-six predefined variables. Each variable Is identified as an alphabetic character. All except P start out as zero - P starts out as pi, to machine accuracy. The variables can be used in expressions or assigned a value. For example, try

```
A=1.5
?A*2
```

By now you have noticed that the calculator prints its results twice - once as a decimal value, and once as a hexadecimal value. The hexadecimal value Is what the number would look like in memory.

Like BASIC, the calculators let you use parenthesis and most familiar operators and functions. Operators have their standard precedence. Below is a summary of the operators and functions provided.

Operator	Operation
+	addition
-	subtraction
*	multiplication
/	division
^	raise a number to a power
ABS	absolute value function
SIGN	sign function; returns -1, 0 or 1
SQRT	square root function
RANDOM	return a random number from 0..1
SIN	sine function
COS	cosine function
TAN	tangent function
ARCTAN	arctangent function
LN	natural log function
EXP	exponent function
INT	integer function

As you can see, there are a rich variety of operations allowed. All of them are also supported as macros In the macro library. The examples below illustrate the use of these functions and operators.

```
?ABS(-10)
?2^10
?EXP(l)
?SIN(P/2)
?P
?INT(-1.5)
?1+2*3
?(1+2)*3
```

Addressing Modes

The addressing modes provided by the floating point macros duplicate those already familiar from the macros provided with ORCA/M. The information is included here for completeness.

Like the instruction set of the CPU, macros use a variety of addressing modes to increase the power and flexibility of each macro. There are four addressing modes supported by the macros: immediate, absolute, indirect and stack.

Immediate addressing is available on all macros that require an input to perform their function. An immediate operand is coded as a pound sign followed by the value for the operand. For example,

```
    PUTF  #1.5
```

would write the number 1.500000 E00 to the screen.

Absolute addresses are coded as a number, label, or expression, using the same rules as absolute addresses on instructions. An absolute address designates the memory location to use as a source or destination by the macro.

Indirect addresses take the form of an address which points to the address of the data rather than the data itself. Indirect addressing is Indicated by enclosing the absolute address where the effective address is stored in soft brackets. Thus,

```
    MULF (Pl),(P2)
```

multiplies the number pointed to by PI by the number pointed to by P2, placing the result where P1 points. Note that, unlike the 6502, the pointers do not need to be in zero page.

Stack addressing refers to taking a source value from the "evaluation stack", or storing a result there. The evaluation stack is the stack used by ORCA high level languages to pass parameters and evaluate expressions. It is a software stack, distinct from the hardware stack in page 1. The INITSTACK macro (from the macros supplied with ORCA/M) can be used to set up this stack. A stack operand is indicated using the character.

When discussing stack operations, it is customary to refer to the values based on the "top of stack" (TOS). In the ORCA macros, the TOS is designated with an * character. The value on the top of the stack is said to be at TOS, while the number below the TOS is said to be at TOS-1. With this in mind, the following operation divides the double precision number at TOS-1 by the one at TOS, placing the result on the TOS. The original two numbers are removed from the stack in the process.

```
    DIVD *,*
```

This addressing mode is very convenient for doing reverse polish notation expressions, and is used heavily in the calculators.

Error Handling

Floating point error handling is done with the aid of three subroutines in the subroutine library provided with ORCA/M. The first of these is called SYSTRAP. it is called by the floating point subroutines whenever an error condition is detected. SYSTRAP stores an error flag in a zero page location called SYSFERR ($EB) and takes corrective action. Five types of errors are possible - the errors, the error code, and the action taken by SYSFERR are shown in the table below.

Error	Number	Action
invalid operation	1	result is argument
division by zero	2	result is infinity
overflow	4	result is infinity
underflow	8	result is 0
inexact	16	best guess given

Overflow, underflow, and division by zero are all common errors on any floating point system. overflow is when the number is too large to be represented by a floating point number. Underflow is when the number is too small.. Division by zero is, of course, illegal.

The remaining two errors are required by the IEEE floating point standard, and while this package does not fully implement that standard, It does follow it when possible. Invalid operation refers to trying to do an operation based on illegal inputs. An example is taking the square root of a negative number. Inexact refers to a result that, due to the input, cannot be computed to a reasonable precision. For example, the sine of 1E20 certainly exists, and is well defined, but' due to the limited precision of floating point numbers, an answer that is as accurate as the floating point format would normally allow cannot be given.

When the system finds a floating point error, the error number is ORed with the current value of SYSFERR. Note that the error numbers are all powers of two - this means that they can all be found in SYSFERR at the same time! Testing for the errors is relatively simple, since SYSFERR is zero if no error has occurred. if an error has occurred, the library subroutine SYSEROR can be called to print the error message. The following code fragment is one way to do this.

```
         LDY   #5             loop counter
         LDA   SYSFERR        get error codes
         LDX   #5             initial error #
LB1      LSR   A              get the error bit
         BCC   LB2            branch if no error
         PHA                  save regs
         TYA
         PHA
         TXA
         PHA
         JSR   SYSEROR        print the error
         PLA                  restore regs
         TAX
         PLA
         TAY
         PLA
LB2      INX                  inc error number
         DEY                  loop
         BNE   LBI
         LDA   #0             reset SYSFERR
         STA   SYSFERR
```

This routine will only print the error message, then continue on. If you want the program to stop when an error is found, you need to replace a library subroutine called SYSERIN. SYSERIN is called by SYSEROR whenever SYSEROR is called. If SYSERIN returns with the carry flag clear, the error is printed - otherwise the error is not printed. Below is a version of SYSERIN that will place the address of a BRK instruction on the top of the stack, then return to get the error message printed. After the error message has been printed, the program ends with a break.

```
SYSERIN  START

         PLA                  save old return address
         TAX
         PLA
         TAY
         LDA   #>BRK-1        set address of break
         PHA
         LDA   #<BRK-1
         PHA
         TYA                  restore return address
         PHA
         TXA
         PHA
         CLC                  clearing carry so error
!                              is printed
         RTS                  return to SYSEROR

BRK      BRK
         END
```

Naturally, the BRK instruction can be replaced with a more sophisticated error handler, if desired.

By the way, the default SYSERIN from the library is just a CLC followed by an RTS.

Source Code

The two source code disks contain the source code for the floating point and double precision libraries, as well as the floating point calculators. On the disk labeled Single Precision Source you will find a group of files starting with FP, as well as a file called COMMON. COMMON is used by all of the ORCA libraries, including the ones shipped with ORCA. It contains the definition of zero page areas and global constants. The files starting with FP, when assembled, produce the library file F.FLOAT.A. On the same disk are some files starting with CF. This is the source code for the floating point calculator.

In both cases, there is not enough room on the disk to assemble the files. if you will be assembling them, first split the disk up, placing the source for the libraries on one disk, and the calculator source on another. Especially when assembling the calculator, place the macros on a RAM disk if you have one. The program makes extensive use of macros, and will take a great deal of time to assemble if the macros are left on a floppy disk. The MCOPY directive at the beginning of the file CF should be changed to reflect the new location of the macros.

The disk labeled Double Precision Source has the same type of source code as the one labeled Single Precision Source, except that it is for double precision. The calculator files start with CD. The library source is in the files starting with DP, and still require the use of COMMON. Once again, there Isn't enough room on the disk to assemble either of these files, so split the disk up if you will be doing anything except looking at the source.

Macro Descriptions

The following macro descriptions are in the same format as was used in the ORCA/M Macro Library Reference Manual, which starts on page 171 of the ORCA/M manual. All macros will destroy the contents of all of the registers. Since their operation is similar, single and double precision macros are described together. Single precision macros always end with the letter (for Floating point), and double precision macros always end with D (Double precision).

ABSx Absolute Value

Forms:

```
LAB ABSD N1,N2
LAB ABSF N1,N2
```

Operands:

```
LAB    Label.
N1     Argument.
N2     Result.
```

Description:

The result is the absolute value of the argument. N2 is optional; if it Is coded, the result is placed there, if it Is not coded, the result is placed at N1 No errors are possible.

Coding Examples

```
         ABSF  NUM1,*              places the
!                                   absolute value of
!                                   NUM1 on the
!                                   software stack
         ABSD  #1,NUM1             places 1 at NUM1
```

ADDx Addition

Forms:

```
   LAB      ADDD  N1,N2,N3
   LAB      ADDF  N1,N2,N3
```

Operands:

```
   LAB    Label.
   N1     First argument.
   N2     Second argument.
   N3     Result.
```

Description:

The two arguments are added together. If N3 is coded, the result is stored there; if it is not coded, the result is stored at N1. Overflows and underflows are possible.

Coding Examples:

```
            ADDF  *,*                perform a stack
   !                                  addition, placing
   !                                  the result on the
   !                                  stack
            ADDD  NUM1,#3.14159      add a constant to
   !                                  NUM1
```

ATANx Arctangent

Forms:

```
   LAB      ATAND N1,N2
   LAB      ATANF N1,N2
```

Operands:

```
   LAB    Label.
   N1     Argument.
   N2     Result.
```

Description:

The arctangent of the argument N1 is extracted. N2 is optional. if it is coded, the result is stored there; otherwise, the result is stored at N1. The result is returned in radians, and is In the range -pi/2..pi/2.

Coding Example:

```
            ATANF (P1)                replace the number
   !                                   pointed to by P1
   !                                   with its
   !                                   arctangent
```

CHSx. Change Sign

Forms:

```
   LAB      CHSD  N1,N2
   LAB      CHSF  N1 N2
```

Operands:

```
   LAB    Label.
   N1     Argument.
   N2     Result.
```

Description:

The result is the negation of N1. If N2 is coded, the result is placed there; otherwise, the result is placed at N1.

Coding Example:

```
            CHSD  #-1.2,*             places 1.2 on the
   !                                   software stack
```

CMPx Compare

Forms:

```
LAB CMPD N1,N2
LAB CMPF N1,N2
```

Operands:

```
LAB    Label.
N1     First number to compare.
N2     Second number to compare.
```

Description:

The first number is compared to the second. If first number is less than the second number, the flag is cleared and the zero flag is set to 0, indicating a non-equal result. The branches BCC, BLT and BNE will then be taken. If N1 is equal to N2, the carry flag is set, and the zero flag Is set to 1. The branches BEQ, BGE and BCS will then be taken. If N1 is greater than N2, the carry flag is set and the zero flag is set to 0. The branches BGE, BCS and BNE will then be taken.

Note that the settings of the flags and the resulting conditional branches mimic the operation of the 6502 CMP instruction.

Unlike most two operand instructions, both operands are required.

Coding Example:

```
         CMPF  NUM1,#0
         BLT   LAB1                branch if negative
```

CNVxy — Convert Type x to Type y

Forms:

```
LAB      CNV2D N1,N2
LAB      CNV2F N1,N2
LAB      CNV4D N1,N2
LAB      CNV4F N1,N2
LAB      CNV8D N1,N2
LAB      CNV8F N1,N2
LAB      CNVD2 N1,N2
LAB      CNVD4 N1,N2
LAB      CNVD8 N1,N2
LAB      CNVDF N1,N2
LAB      CNVDS N1,N2
LAB      CNVF2 N1,N2
LAB      CNVF4 N1,N2
LAB      CNVF8 N1,N2
LAB      CNVFD N1,N2
LAB      CNVFS N1,N2
LAB      CNVSD N1,N2
LAB      CNVSF N1,N2
```

Operands:

LAB	Label.
N1	Argument.
N2	Result.

Description:

These macros are used to convert from one type to another type. By varying the characters substituted for x and y, floating point an double precision numbers can be converted to or from any type supported by this package, or any type in the standard macro library supplied with ORCA/M.

Both operands can be specified using absolute, Indirect, or stack addressing. it is also possible to use immediate addressing on the argument, although this is not recommended (it would be more efficient to simply use a constant in its original form). The argument is converted from the type specified by x to the type specified by y. If N2 is coded, the result is stored there; otherwise, the result is stored at N1. Overflow and underflow errors are possible during the conversion process.

The letters used to identify the various types are!

2	two byte signed integer
4	four byte signed integer
8	eight byte signed integer
D	double precision floating point
F	single precision floating point
S	string

Coding Example:

```
         CNVD2 DP,I2               convert the double
!                                   precision number
!                                   DP to a two byte
!                                   integer
```

C0Sx Cosine

Forms:

```
   LAB      COSD  N1,N2
   LAB      COSF  N1,N2
```

Operands:

```
   LAB    Label.
   N1     Argument.
   N2     Result.
```

Description:

The cosine of the argument is extracted. if N2 is coded, the result is placed there; otherwise, the result Is placed at N1. The argument must be specified in radians. Underflow errors are possible. If the argument exceeds 1303 for single precision floating point, or 210828714 for double precision floating point, a precise answer is not possible and an inexact error Is returned. (Inexact is a type of error under the IEEE standard, not an unpredictable error!)

Coding Example:

```
            COSD  N1                  N1 is replaced by
   !                                   its cosine
```

DIVx Division

Forms:

```
   LAB      DIVD  N1,N2,N3
   LAB      DIVF  N1,N2,N3
```

Operands:

```
   LAB    Label.
   N1     Numerator.
   N2     Denominator.
   N3     Result.
```

Description:

N1 is divided by N2. If 03 is coded, the result is placed there; otherwise, the result is placed at N1. Overflow, underflow and division by zero errors are possible.

Coding Examples:

```
            DIVF  N1 #2               divide N1 by 2
            DIVD  N1,#2,N2            do the same,
   !                                   placing the
   !                                   result at N2
```

EXPx — Exponent

Forms:

```
   LAB      EXPD  N1,N2
   LAB      EXPF  N1,N2
```

Operands:

LAB	Label.
N1	Argument.
N2	Result.

Description:

The exponent of the argument is extracted. If N2 is coded, the result is placed there, otherwise the result is placed at N1. Overflow and underflow errors are possible.

Coding Example:

```
            EXPF  #1,N1               places e at N1
```

FERR — Flag a Floating Point Error

Forms:

```
   LAB      FERR  NUM
```

Operands:

```
   LAB    Label.
   NUM    Error number.
```

Description:

This macro is used to flag a floating point error. its main use is in new floating point routines that may be added to the library. The operand must be a valid operand for a LDX instruction. Errors which are currently defined are:

1	invalid operation
2	division by zero
4	overflow
8	underflow
16	inexact

Coding Example:

```
   label    op    operand             comment
            FERR  #1                  flag invalid
   !                                   operation
```

GETx Read a Number

Forms:

```
   LAB      GETD  N1,CR
   LAB      GETF  N1,CR
```

Operands:

```
   LAB    Label.
   N1     Location to place the number.
   CR     Scan to end of line flag.
```

Description:

The GET macros are the standard way to read numbers from external devices. They receive all information from the $38 input hook, which means that the input stream can be redirected to read from disk drives or other input devices by simply placing the address of the character input routine at $38.

N1 is the location to store the number read. CR is a flag. If anything is coded in the CR field, the input routine will scan to the end of the input line after reading the number. if CR is not coded, the input routine consumes one character after the last character that is in the number. Generally, that one character is a blank or carriage return.

Coding Examples:

```
            GETF  FP                  reads a floating
   !                                   point number from
   !                                   the keyboard
            GETD  DP,CR=T             reads a double
   !                                   precision number,
   !                                   then scans to the
   !                                   end of the line
```

INTx Integer

Forms:

```
   LAB      INTD  N1,N2
   LAB      INTF  N1,N2
```

Operands:

```
   LAB    Label.
   N1     Argument.
   N2     Result.
```

Description:

The result is the largest integer that is less than or equal to the argument. If N2 is coded, the result is stored there; otherwise, the result Is stored at N1.

Note that the largest integer part of -1.5 is -2. That is the normal interpretation of the integer function, but differs from the implementation used under DOS ORCA/M. There, the function returned the integer part, ignoring the sign, so the result would have been -1.

Coding Example:

```
            INTD  NUM1,NUM2           NUM2 = INT(NUM1)
```

LNx Natural Log

Forms:

```
   LAB      LND   N1,N2
   LAB      LNF   N1,N2
```

Operands:

```
   LAB    Label.
   N1     Argument.
   N2     Result.
```

Description:

The natural log of the argument is placed at N2 if N2 is coded, and at N1 if it is not. Underflows are possible on small arguments. An invalid operation error will result if the argument is zero or negative.

Coding Example:

```
            EXPF  #1,NUM1             NUM1 = 2.718282
            LNF   NUM1,NUM2           NUM2 = 1
```

MULx Multiplication

Forms:

```
   LAB      MULD  N1,N2,N3
   LAB      MULF  N1,N2,N3
```

Operands:

```
   LAB    Label.
   N1     First argument.
   N2     Second argument.
   N3     Result.
```

Description:

The two numbers at N1 and N2 are multiplied. If N3 is coded, the result is placed there; otherwise, the result is placed at N1. Overflow and underflow errors are possible.

Coding Example:

```
            MULD  *,(P1)              multiply the top
   !                                   of stack by the
   !                                   number pointed to
   !                                   by P1, placing
   !                                   the result on the
   !                                   top of stack
```

PUTx — Write a Number

Forms:

```
   LAB      PUTD  N1,F1,F2,CR
   LAB      PUTF  N1,F1,F2,CR
```

Operands:

```
   LAB    Label.
   N1     Number to write.
   F1     Field width.
   F2     Significant digits.
   CR     Carriage return flag.
```

Description:

The number at N1 is written to the current output device. Since $36 contains the name of the current output device, the output can be redirected to any character oriented device by simply placing the name of a subroutine that writes a character to the device at $36.

F1 Is an optional field width that defaults to zero. if specified, It defines the width of a field that the number will be right justified in. If the field width is less than or equal to the number of characters needed to write the number, the number will be written out without leading blanks. If the field width is greater than the number of characters needed to write the number, enough blanks will be written out before the number to right justify it in the field.

F2 specifies the number of significant digits to write. if it is left out, or if zero is used, the number will be written in exponential format to machine precision.

CR is a flag that indicates if the number should be followed by a carriage return. If anything Is coded in this field, the number will be followed by a carriage return. If nothing is coded here, further output will be on the same line.

Coding Examples:

```
            PUTD  NUM1                write the numbers
            PUTD  NUM2,,#20,CR=T       on the same line
```

PWRx Raise a Number to a Power

Forms:

```
   LAB      PWRD  N1,N2,N3
   LAB      PWRF  N1,N2,N3
```

Operands:

LAB	Label.
N1	Argument.
N2	Power.
N3	Result.

Description:

N1 is raised to the N2 power. If W3 is coded, the result is placed there; otherwise, the result is placed at N1. N1 must be positive and nonzero, or an invalid operation error will result. Overflow and underflow errors are also possible.

Coding Example:

```
            PWRF  #2,NUM1,NUM2        NUM2 = 2^NUM1
```

RANx Random Number

Forms:

```
   LAB      RAND  N1
   LAB      RANF  N1
```

Operands:

```
   LAB    Label.
   N1     Result.
```

Description:

A pseudo random number in the range 0.0 to 1.0 (including 0.0 but not 1.0) is generated and stored at N1. Since no argument is required, this is the only floating point macro that uses only one parameter.

The random number generator should be initialized using the SEED macro from the standard library supplied with ORCA/M before this macro is used.

Coding Example:

```
            SEED                      generate a random
            RAND  NUM1                 number
```

SIGNx Sign

Forms:

```
LAB      SIGND NUM2
LAB      SIGNF N1,N2
```

Operands:

LAB	Label.
N1	Argument.
N2	Result.

Description:

If the argument is less than zero, the result is -1. If the argument is zero, the result is also zero. If the argument is positive, the result is 1. If N2 is coded, the result Is placed there; otherwise, the result is placed at N1.

Coding Example:

```
         SIGND NUM1                replace NUM1 by
!                                   its sign
```

SINx Sine

Forms:

```
   LAB      SIND  N1,N2
   LAB      SINF  N1,N2
```

Operands:

LAB	Label.
N1	Argument.
N2	Result.

Description:

The sine of the argument is extracted. if N2 is coded, the result is placed there; otherwise, the result is placed at N1. The argument must be specified in radians. Underflow errors are possible. If the argument exceeds 1303 for single precision floating point, or 210828714 for double precision floating point, a precise answer is not possible and an inexact error is returned. (Inexact is a type of error under the IEEE standard, not an unpredictable error!)

Coding Example:

```
            SINF  ANGLE,SIN           SIN = SIN(ANGLE)
```

SQRTx — Square Root

Forms:

```
LAB      SQRTD N1,N2
LAB      SQRTF N1,N2
```

Operands:

```
LAB Label.
N1  Argument.
N2  Result.
```

Description:

The result is the square root of the argument. If N2 is coded, the result is placed there; otherwise, the result is placed at N1. If the argument is negative, an invalid operation error results. An underflow error is also possible.

Coding Example:

```
         SQRTF #2,NUM1             NUM1 = 1.414214
```