The Math Module

Standard Function Library Module

OS-9 contains a standard function library math module which provides common subroutines for extended mathematical and I/O conversion functions. OS-9 C, Basic09, and Fortran compilers also use this module.

OS-9 math modules provide the following functions:

- Basic floating point math
- Extended integer math
- Type conversion
- Transcendental and extended mathematical functions

Normally, the math module uses software routines located in a library file to provide the extended functions. User programs can call the library directly, using the 68000 trap instruction. You can also use these library files for non-OS-9 target systems. The following are library files that can be embedded in your applications:

Library File	Use on:
Math.l	Systems without a math co-processor
Math881.l	Systems with a math co-processor

Systems that do not use a math co-processor can use the Math.l library file. In systems that do have a math co-processer, you can replace the software-based files with files that use arithmetic processing hardware, without altering the application software. For example, use the Math881 file for the 68881/882 FPCP.

If you do not want the math module functions embedded within your application program, you can install the appropriate module as a user trap routine, and call it using the 68000 trap instruction.

Module Name	File Name	Trap#	Use on:
Math	Math	15	Systems without a math co-processor.
Math	Math881	15	Systems with a math co-processor.

Calling Standard Function Module Routines

You can use the OS-9 Load command to pre-load the standard function library module in memory for quick access when needed. You can make it part of the system's startup file. Including the trap handlers in the OS9Boot file is not recommended. The following description of standard function module linkage and calling methods is intended for assembly language programmers. Programs generated by the OS-9 compilers automatically perform all required functions without any special action on the part of the user.

Prior to calling the standard function modules, an assembly language program should use the OS-9 F$TLink system call. The TLink parameters should be the trap number and module name (refer to the table on the previous page). This installs and links the user's process to the desired module(s). Calls to individual routines are made using the trap instruction. For example, a call to the FAdd function could look like this:

```
trap    #T$Math      Trap number of module
dc.w    T$FAdd       Code of FAdd function
```

For simplicity, a macro is included in the assembler for this purpose. The following line is equivalent to the above example:

```
tcall   T$Math,T$FAdd  Trap number and code for FAdd
```

In non-OS-9 target environments, you may also call these routines directly using bsr instructions, and including the appropriate library in the code (math.l). For example:

```
bsr     _T$FAdd      Floating point addition
```

Many functions set the MPU status register N, Z, V, and C bits so the trap or bsr may be immediately followed by a conditional branch instruction for comparisons and error checking. When an error occurs, the system-wide convention is followed, where the C condition code bit is set and register d1 returns the specific error code.

In some cases a trapv instruction executes at the end of a function. This causes a trapv exception if the V (overflow) condition code is set.

Data Formats

Some functions support two integer types:

unsigned 32-bit unsigned integers
long 32-bit signed integers

Two floating point formats are also supported:

float 32-bit floating point numbers
double 64-bit double precision floating point numbers

Floating point math routines use formats based on the proposed IEEE standard for compatibility with floating point math hardware. 32-bit floating point operands are internally converted to 64-bit double precision before computation and converted back to 32 bits afterwards as required by the IEEE and C language standards. Therefore, the float type has no speed advantage over the double type. This package does not support de-normalized numbers and negative zero.

The Math Module

The math module provides single and double precision floating point arithmetic, extended integer arithmetic, and type conversion routines.

Integer Operations

T$LMul T$UMul T$LDiv T$LMod T$UDiv T$UMod

Single Precision Floating Point Operations

T$FAdd T$FInc T$FSub T$FDec T$FMul T$FDiv T$FCmp T$FNeg

Double Precision Floating Point Operations

T$DAdd T$DInc T$DSub T$DDec T$DMul T$DDiv T$DCmp T$DNeg

ASCII to Numeric Conversions

T$AtoN T$AtoL T$AtoU T$AtoF T$AtoD

Numeric to ASCII Conversions

T$LtoA T$UtoA T$FtoA T$DtoA

Numeric to Numeric Conversions

T$LtoF T$LtoD T$UtoF T$UtoD T$FtoL T$DtoL T$FtoU T$DtoU
T$FtoD T$DtoF T$FTrn T$DTrn T$FInt T$DInt T$DNrm

The math module also provides transcendental and extended mathematical functions. The calling routine controls the precision of these routines. For example, if fourteen digits of precision are required, the floating-point representation for 1E-014 should be passed to the routine.

Function Name	*Operation*
T$Sin	Sine function
T$Cos	Cosine function
T$Tan	Tangent function
T$Asn	Arc sine function
T$Acs	Arc cosine function
T$Atn	Arc tangent function
T$Log	Natural logarithm function
T$Log10	Common logarithm function
T$Sqrt	Square root function
T$Exp	Exponential function
T$Power	Power function

The following table contains the hex representations which you should pass to these routines to define the precision of the operation.

Precision Hex	*Representation*
1E-001	**3fb99999 9999999a**
1E-002	**3f847ae1 47ae147b**
1E-003	**3f50624d d2f1a9fc**
1E-004	**3f1a36e2 eb1c432d**
1E-005	**3ee4f8b5 88e368f1**
1E-006	**3eb0c6f7 a0b5ed8e**
1E-007	**3e7ad7f2 9abcaf4a**
1E-008	**3e45798e e2308c3b**
1E-009	**3e112e0b e826d696**
1E-010	**3ddb7cdf d9d7bdbd**
1E-011	**3da5fd7f e1796497**
1E-012	**3d719799 812dea12**
1E-013	**3d3c25c2 68497683**
1E-014	**3d06849b 86a12b9c**

NOTE: Using a precision greater than 14 digits may cause the routine to get trapped in an infinite loop.

T$Acs — Arc Cosine Function

ASM CALL: **TCALL T$Math,T$Acs**

INPUT: **d0:d1 = x**
d2:d3 = Precision

OUTPUT: **d0:d1 = ArcCos(x) (in radians)**

CONDITION CODES: **C Set on error**

POSSIBLE ERRORS: **E$IllArg**

FUNCTION: T$Acs returns the arc cosine() in radians. If the operand passed is illegal, an error is returned.

T$Asn — Arcsine Function

ASM CALL: **TCALL T$Math,T$Asn**

INPUT: **d0:d1 = x**
d2:d3 = Precision

OUTPUT: **d0:d1 = ArcSin(x) (in radians)**

CONDITION CODES: **C Set on error**

POSSIBLE ERRORS: **E$IllArg**

FUNCTION: T$Asn returns the arcsine() in radians. If the operand passed is illegal, an error is returned.

T$Atn — Arc Tangent Function

ASM CALL: **TCALL T$Math,T$Atn**

INPUT: **d0:d1 = x**
d2:d3 = Precision

OUTPUT: **d0:d1 = ArcTan(x) (in radians)**

CONDITION CODES: **C Set on error**

POSSIBLE ERRORS: **E$IllArg**

FUNCTION: T$Atn returns the arc tangent() in radians. If the operand passed is illegal, an error is returned.

T$AtoD — ASCII to Double-Precision Floating Point

ASM CALL: **TCALL T$Math,T$AtoD**

INPUT: **(a0) = Pointer to ASCII string**
Format: <sign><digits>.<digits><E or e><sign><digits>

OUTPUT: **(a0) = Updated pointer**
d0:d1 = Double-precision floating-point number

CONDITION CODES: **N Undefined**
Z Undefined
V Set on underflow or overflow
C Set on error

POSSIBLE ERRORS: **E$NotNum or E$FmtErr**

FUNCTION: T$AtoD performs a conversion from an ASCII string to a double-precision floating-point number. If the first character is not the sign (+ or -) or a digit, E$NotNum is returned. If the first character following the E is not the sign or a digit, E$FmtErr is returned.

If the overflow bit (V) is set, zero (on underflow) or +/- infinity (overflow) is returned.

T$AtoF — ASCII to Single-Precision Floating-Point

ASM CALL: **TCALL T$Math,T$AtoF**

INPUT: **(a0) = Pointer to ASCII string**
Format: <sign><digits>.<digits><E or e><sign><digits>

OUTPUT: **(a0) = Updated pointer**
d0:d1 = Double-precision floating-point number

CONDITION CODES: **N Undefined**
Z Undefined
V Set on underflow or overflow
C Set on error

POSSIBLE ERRORS: **E$NotNum or E$FmtErr**

FUNCTION: T$AtoF performs a conversion from an ASCII string to a single-precision floating-point number. If the first character is not the sign (+ or -) or a digit, E$NotNum is returned. If the first character following the E is not the sign or a digit, E$FmtErr is returned.

If the overflow bit (V) is set, zero (on underflow) or +/- infinity (overflow) is returned.

T$AtoL ASCII to Long Conversion

ASM CALL: **TCALL T$Math,T$AtoL**

INPUT: **(a0) = Pointer to ASCII string (format: <sign><digits>)**

OUTPUT: **(a0) = Updated pointer**
d0.l = Signed long

CONDITION CODES: **N Undefined**
Z Undefined
V Set on overflow
C Set on error

POSSIBLE ERRORS: **E$NotNum**

FUNCTION: T$AtoL performs a conversion from an ASCII string to a signed long integer. If the first character is not a sign (+ or -) or a digit, an error is returned.

T$AtoN — ASCII to Numeric Conversion

ASM CALL: **TCALL T$Math,T$AtoN**

INPUT: **(a0)** = **Pointer to ASCII string**

OUTPUT: **(a0)** = **Updated pointer**
d0 = **Number if returned as long (signed or unsigned)**
d0:d1 = **Number if returned in floating point format**

CONDITION CODES: **See explanation below.**

POSSIBLE ERRORS: **TrapV**

FUNCTION: T$AtoN can return results of various types depending on the format of the input string and the magnitude of the converted value. The type of the result is passed back to the calling program using the V and N condition code bits.

V=0 and N=1 indicate a signed integer is returned in d0.l

V=0 and N=0 indicate an unsigned integer is returned in d0.l

V=1 indicates a double-precision number is returned in d0:d1

If any of the following conditions are met, the number is returned as a double-precision floating-point value:

- The number is positive and overflows an unsigned long.
- The number is negative and overflows a signed long.
- The number contains a decimal point and/or an E exponent.

If none of the above conditions are met, the result is returned as an unsigned long (if positive) or a signed long (if negative).

T$AtoU ASCII to Unsigned Conversion

ASM CALL: **TCALL T$Math,T$AtoU**

INPUT: **(a0) = Pointer to ASCII string (format: <digits >)**

OUTPUT: **(a0) = Updated pointer**
d0.l = Unsigned long

CONDITION CODES:
N Undefined
Z Undefined
V Set on overflow
C Set on error

POSSIBLE ERRORS: **E$NotNum**

FUNCTION: T$AtoU performs a conversion from an ASCII string to an unsigned long integer. If the first character is not a digit, an error is returned.

T$Cos — Cosine Function

ASM CALL: **TCALL T$Math,T$Cos**

INPUT: **d0:d1 = x (in radians)**
d2:d3 = Precision

OUTPUT: **d0:d1 = Cos(x)**

CONDITION CODES: **C Always clear**

POSSIBLE ERRORS: **None**

FUNCTION: T$Cos returns the cosine() of an angle. The angle must be specified in radians. No errors are possible, and all condition codes are undefined.

T$DAdd — Double Precision Addition

ASM CALL: **TCALL T$Math,T$DAdd**

INPUT: **d0:d1 = Addend**
d2:d3 = Augend

OUTPUT: **d0:d1 = Result (d0:d1 + d2:d3)**

CONDITION CODES: **N Set if result is negative**
Z Set if result is zero
V Set on underflow or overflow
C Always cleared

POSSIBLE ERRORS: **TrapV**

FUNCTION: T$DAdd adds two double-precision floating point numbers. Overflow and underflow are indicated by setting the V bit. In either case, a trapv exception is generated. If an underflow caused the exception, zero is returned. If it was an overflow, infinity (with the proper sign) is returned.

T$DCmp Double Precision Compare

ASM CALL: **TCALL T$Math,T$DCmp**

INPUT: **d0:d1 = First operand**
d2:d3 = Second operand

OUTPUT: **d0.l through d3.l remain unchanged**

CONDITION CODES: **N Set if second operand is larger than the first**
Z Set if operands are equal
V Always cleared
C Always cleared

POSSIBLE ERRORS: **None**

FUNCTION: Two double-precision floating point numbers are compared by T$DCmp. The operands passed to this function are not destroyed.

T$DDec Double Precision Decrement

ASM CALL: **TCALL T$Math,T$DDec**

INPUT: **d0:d1 = Operand**

OUTPUT: **d0:d1 = Result (d0:d1 - 1.0)**

CONDITION CODES: **N Set if result is negative**
Z Set if result is zero
V Set on underflow
C Always cleared

POSSIBLE ERRORS: **TrapV**

FUNCTION: This function subtracts 1.0 from the double-precision floating point operand. Underflow is indicated by setting the V bit. If an underflow occurs, a trapv exception is generated and zero is returned.

T$DDiv — Double Precision Divide

ASM CALL: **TCALL T$Math,T$DDiv**

INPUT: **d0:d1 = Dividend**
d2:d3 = Divisor

OUTPUT: **d0:d1 = Result (d0:d1 / d2:d3)**

CONDITION CODES:
N Set if result is negative
Z Set if result is zero
V Set on underflow, overflow, or divide by zero
C Set on divide by zero

POSSIBLE ERRORS: **TrapV**

FUNCTION: T$DDiv performs division on two double-precision floating point numbers. Overflow, underflow, and divide-by-zero are indicated by setting the V bit. In any case, a trapv exception is generated. If an underflow caused the exception, zero is returned. If it was an overflow or divide-by-zero, infinity (with the proper sign) is returned.

T$DInc — Double Precision Increment

ASM CALL: **TCALL T$Math,T$DInc**

INPUT: **d0:d1 = Operand**

OUTPUT: **d0:d1 = Result (d0:d1 + 1.0)**

CONDITION CODES:
N Set if result is negative
Z Set if result is zero
V Set on overflow
C Always cleared

POSSIBLE ERRORS: **TrapV**

FUNCTION: T$DInc adds 1.0 to the double-precision floating point operand. Overflow is indicated by setting the V bit. If an overflow occurs, a trapv exception is generated and infinity (with the proper sign) is returned.

T$DInt — Round Double-Precision Floating-Point Number

ASM CALL: **TCALL T$Math,T$DInt**

INPUT: **d0:d1 = Double-precision floating-point number**

OUTPUT: **d0:d1 = Rounded double-precision floating-point number**

CONDITION CODES: **All condition codes are undefined.**

POSSIBLE ERRORS: **None**

FUNCTION: Floating point numbers consist of two parts: integer and fraction. The purpose of T$DInt is to round the floating point number passed to it, leaving only an integer. If the fraction is exactly 0.5, the integer is rounded to an even number.

EXAMPLES:
23.45 rounds to **23.00**
23.50 rounds to **24.00**
23.73 rounds to **24.00**
24.50 rounds to **24.00** (rounds to even number)

T$DMul — Double Precision Multiplication

ASM CALL: **TCALL T$Math,T$DMul**

INPUT: **d0:d1 = Multiplicand**
d2:d3 = Multiplier

OUTPUT: **d0:d1 = Result (d0:d1 * d2:d3)**

CONDITION CODES:
N Set if result is negative
Z Set if result is zero
V Set on underflow or overflow
C Always cleared

POSSIBLE ERRORS: **TrapV**

FUNCTION: T$DMul multiplies two double-precision floating point numbers. Overflow and underflow are indicated by setting the V bit. In either case, a trapv exception is generated. If an underflow caused the exception, zero is returned. If it was an overflow, infinity (with the proper sign) is returned.

T$DNeg — Double Precision Negate

ASM CALL: **TCALL T$Math,T$DNeg**

INPUT: **d0:d1 = Operand**

OUTPUT: **d0:d1 = Result (d0:d1 * -1.0)**

CONDITION CODES:
N Set if result is negative
Z Set if result is zero
V Always cleared
C Always cleared

POSSIBLE ERRORS: **None**

FUNCTION: T$DNeg negates a double-precision floating point operand. To eliminate the overhead of calling this routine, it is simple to change the sign bit of the floating-point number. However, you should check for a zero number because this package does not support negative zero.

This example is written as a subroutine and expects the floating-point number to be in d0:d1.

```
Negate  tst.l d0     test for zero
        beq.s Neg10  branch if it is zero
        bchg  #31,d0 change sign bit
Neg10   rts          return
```

T$DNrm — 64-bit Unsigned to Double-Precision Conversion

ASM CALL: **TCALL T$Math,T$DNrm**

INPUT: **d0:d1 = 64-bit Unsigned Integer**
d2.l = Exponent

OUTPUT: **d0:d1 = Double-precision floating-point number**

CONDITION CODES:
N Undefined
Z Undefined
V Set on underflow or overflow
C Undefined

POSSIBLE ERRORS: **None**

FUNCTION: Double-precision floating point numbers maintain 52 bits of mantissa. T$DNrm converts a 64-bit binary number to double-precision format. The extra 12 bits are rounded. If an underflow or overflow occurs, the V bit is set, but a trapv exception is not generated.

T$DSub — Double-Precision Subtraction

ASM CALL: **TCALL T$Math,T$DSub**

INPUT: **d0:d1 = Minuend**
d2:d3 = Subtrahend

OUTPUT: **d0:d1 = Result (d0:d1 - d2:d3)**

CONDITION CODES:
N Set if result is negative
Z Set if result is zero
V Set on underflow or overflow
C Always cleared

POSSIBLE ERRORS: **TrapV**

FUNCTION: T$DSub performs subtraction on two double-precision floating point numbers. Overflow and underflow are indicated by setting the V bit. In either case, a trapv exception is generated. If an underflow caused the exception, zero is returned. If it was an overflow, infinity (with the proper sign) is returned.

T$DtoA Double-Precision Floating-Point to ASCII

ASM CALL: **TCALL T$Math,T$DtoA**

INPUT: **d0:d1 = Double-precision floating-point number**
d2.l = Low-Word: digits desired in result
High-Word: digits desired after decimal-point
(a0) = Pointer to conversion buffer

OUTPUT: **(a0) = ASCII digit string**
d0.l = Two's complement exponent

CONDITION CODES: **N Set if the number is negative**
Z Undefined
V Undefined
C Undefined

POSSIBLE ERRORS: **None**

FUNCTION: The double-precision float passed to T$DtoA is converted to an ASCII string. The conversion terminates as soon as the number of digits requested are converted, or when the specified digit after the decimal point is reached; whichever comes first. A null is appended to the end of the string. Therefore, the buffer should be one byte larger than the expected number of digits.

The converted string only contains the mantissa digits. The N bit indicates the sign of the number, and the exponent returns in register d0.

T$DtoF — Double to Single Floating-Point Conversion

ASM CALL: **TCALL T$Math,T$DtoF**

INPUT: **d0:d1 = Double-precision floating-point number**

OUTPUT: **d0.l = Single-precision floating-point number**

CONDITION CODES:
N Undefined
Z Undefined
V Set on underflow or overflow
C Undefined

POSSIBLE ERRORS: **TrapV**

FUNCTION: T$DtoF converts floating-point numbers in double-precision format to single-precision format. No errors are possible and all condition codes are undefined. If an overflow or underflow occurs, the V bit is set and a trapv exception is generated.

T$DotL — Double -Precision to Signed Long Integer

ASM CALL: **TCALL T$Math,T$DtoL**

INPUT: **d0:d1 = Double-precision floating-point number**

OUTPUT: **d0.l = Signed Long Integer**

CONDITION CODES:
N Undefined
Z Undefined
V Set on overflow
C Undefined

POSSIBLE ERRORS: **TrapV**

FUNCTION: The integer portion of the floating point number is converted to a signed long integer. The fraction is truncated. If an overflow occurs, the V bit is set and a trapv exception is generated.

T$DtoU — Double-Precision to Unsigned Long Integer

ASM CALL: **TCALL T$Math,T$DtoU**

INPUT: **d0:d1 = Double-precision floating-point number**

OUTPUT: **d0.l = Unsigned Long Integer**

CONDITION CODES:
N Undefined
Z Undefined
V Set on overflow
C Undefined

POSSIBLE ERRORS: **TrapV**

FUNCTION: The integer portion of the floating point number converts to an unsigned long integer. The fraction is truncated. If an overflow occurs, the V bit is set and a trapv exception is generated.

T$DTrn — Truncate Double-Precision Floating-Point Number

ASM CALL: **TCALL T$Math,T$DTrn**

INPUT: **d0:d1 = Double-precision floating-point number**

OUTPUT: **d0:d1 = Normalized integer portion of the floating point number**
d2:d3 = Normalized fractional portion of the floating point number

CONDITION CODES: **All condition codes are undefined.**

POSSIBLE ERRORS: **None**

FUNCTION: Floating point numbers consist of two parts: integer and fraction. The purpose of T$DTrn is to separate the two parts. For example, if the number passed is 283.75, this function returns 283.00 in d0:d1 and 0.75 in d2:d3.

T$Exp — Exponential Function

ASM CALL: **TCALL T$Math,T$Exp**

INPUT: **d0:d1 = x**
d2:d3 = Precision

OUTPUT: **d0:d1 = Exp(x)**

CONDITION CODES: **C Always clear**

POSSIBLE ERRORS: **None**

FUNCTION: T$Exp performs the exponential function on the argument passed. That is, it raises e to the x power (where e = 2.718282 and x is the argument passed).

T$FAdd — Single Precision Addition

ASM CALL: **TCALL T$Math,T$FAdd**

INPUT: **d0.l = Addend**
d1.l = Augend

OUTPUT: **d0.l = Result (d0 + d1)**

CONDITION CODES: **N Set if result is negative**
Z Set if result is zero
V Set on underflow or overflow
C Always cleared

POSSIBLE ERRORS: **TrapV**

FUNCTION: T$FAdd adds two single-precision floating point numbers. Overflow and underflow are indicated by setting the V bit. In either case, a trapv exception is generated. If an underflow caused the exception, zero is returned. If it was an overflow, infinity (with the proper sign) is returned.

T$FCmp — Single Precision Compare

ASM CALL: **TCALL T$Math,T$FCmp**

INPUT: **d0.l = First operand**
d1.l = Second operand

OUTPUT: **d0.l and d1.l remain unchanged**

CONDITION CODES:
N Set if second operand is larger than the first
Z Set if operands are equal
V Always cleared
C Always cleared

POSSIBLE ERRORS: **None**

FUNCTION: Two single-precision floating point numbers are compared by T$FCmp. The operands passed to T$FCmp are not destroyed.

T$FDec — Single Precision Decrement

ASM CALL: **TCALL T$Math,T$FDec**

INPUT: **d0.l = Operand**

OUTPUT: **d0.l = Result (d0 - 1.0)**

CONDITION CODES:
N Set if result is negative
Z Set if result is zero
V Set on underflow
C Always cleared

POSSIBLE ERRORS: **TrapV**

FUNCTION: T$FDec subtracts 1.0 from the single-precision floating point operand. Underflow is indicated by setting the V bit. If an underflow occurs, a trapv exception is generated and zero is returned.

T$FDiv — Single Precision Divide

ASM CALL: **TCALL T$Math,T$FDiv**

INPUT: **d0.l = Dividend**
d1.l = Divisor

OUTPUT: **d0.l = Result (d0 / d1)**

CONDITION CODES:
N Set if result is negative
Z Set if result is zero
V Set on underflow, overflow or divide by zero
C Set on divide by zero

POSSIBLE ERRORS: **TrapV**

FUNCTION: T$FDiv performs division on two single-precision floating point numbers. Overflow, underflow, and divide-by-zero are indicated by setting the V bit. In any case, a trapv exception is generated. If an underflow caused the exception, zero is returned. If it was an overflow or divide-by-zero, infinity (with the proper sign) is returned.

T$FInc — Single Precision Increment

ASM CALL: **TCALL T$Math,T$FInc**

INPUT: **d0.l = Operand**

OUTPUT: **d0.l = Result (d0 + 1.0)**

CONDITION CODES:
N Set if result is negative
Z Set if result is zero
V Set on overflow
C Always cleared

POSSIBLE ERRORS: **TrapV**

FUNCTION: T$FInc adds 1.0 to the single-precision floating point operand. Overflow is indicated by setting the V bit. If an overflow occurs, a trapv exception is generated and infinity (with the proper sign) is returned.

T$FInt — Round Single-Precision Floating-Point Number

ASM CALL: **TCALL T$Math,T$FInt**

INPUT: **d0.l = Single-precision floating-point number**

OUTPUT: **d0.l = Rounded single-precision floating-point number**

CONDITION CODES: **All condition codes are undefined.**

POSSIBLE ERRORS: **None**

FUNCTION: Floating point numbers consist of two parts: integer and fraction. The purpose of T$FInt is to round the floating point number passed to it, leaving only an integer. If the fraction is exactly 0.5, the integer is rounded to an even number.

EXAMPLES: **23.45** rounds to **23.00**
23.50 rounds to **24.00**
23.73 rounds to **24.00**
24.50 rounds to **24.00** (rounds to even number)

T$FMul — Single Precision Multiplication

ASM CALL: **TCALL T$Math,T$FMul**

INPUT: **d0.l = Multiplicand**
d1.l = Multiplier

OUTPUT: **d0.l = Result (d0 * d1)**

CONDITION CODES:
N Set if result is negative
Z Set if result is zero
V Set on underflow or overflow
C Always cleared

POSSIBLE ERRORS: **TrapV**

FUNCTION: T$FMul multiplies two single-precision floating point numbers. Overflow and underflow are indicated by setting the V bit. In either case, a trapv exception is generated. If an underflow caused the exception, zero is returned. If it was an overflow, infinity (with the proper sign) is returned.

T$FNeg — Single Precision Negate

ASM CALL: **TCALL T$Math,T$FNeg**

INPUT: **d0.l = Operand**

OUTPUT: **d0.l = Result (d0 * -1.0)**

CONDITION CODES:
N Set if result is negative
Z Set if result is zero
V Always cleared
C Always cleared

POSSIBLE ERRORS: **None**

FUNCTION: T$FNeg negates a single-precision floating point operand. To eliminate the overhead of calling this routine, it is simple to change the sign bit of the floating-point number. Be sure to check for a zero number, because this package does not support negative zero.

This example is written as a subroutine and expects the floating-point number to be in d0.

```
Negate  tst.l d0     test for zero
        beq.s Neg10  branch if it is zero
        bchg  #31,d0 change sign bit
Neg10   rts          return
```

T$FSub Single Precision Subtraction

ASM CALL: **TCALL T$Math,T$FSub**

INPUT: **d0.l = Minuend**
d1.l = Subtrahend

OUTPUT: **d0.l = Result (d0 - d1)**

CONDITION CODES:
N Set if result is negative
Z Set if result is zero
V Set on underflow or overflow
C Always cleared

POSSIBLE ERRORS: **TrapV**

FUNCTION: T$FSub performs subtraction on two single-precision floating point numbers. Overflow and underflow are indicated by setting the V bit. In either case, a trapv exception is generated. If an underflow caused the exception, zero is returned. If it was an overflow, infinity (with the proper sign) is returned.

T$FtoA — Single-Precision Floating-Point to ASCII

ASM CALL: **TCALL T$Math,T$FtoA**

INPUT: **d0.l = Single-precision floating-point number**
d2.l = Low-Word: digits desired in result
High-Word: digits desired after decimal-point
(a0) = Pointer to conversion buffer

OUTPUT: **(a0) = ASCII digit string**
d0.l = Two's complement exponent

CONDITION CODES: **N Set if the number is negative**
Z Undefined
V Undefined
C Undefined

POSSIBLE ERRORS: **None**

FUNCTION: The single-precision float passed to T$FtoA is converted to an ASCII string. The conversion terminates as soon as the number of digits requested are converted or when the specified digit after the decimal point is reached; whichever comes first. A null is appended to the end of the string. Therefore, the buffer should be one byte larger than the expected number of digits.

The converted string only contains the mantissa digits. The N bit indicates the sign of the number, and the exponent is returned in register d0.

T$FtoD — Single to Double Floating-Point Conversion

ASM CALL: **TCALL T$Math,T$FtoD**

INPUT: **d0.l = Single-precision floating-point number**

OUTPUT: **d0:d1 = Double-precision floating-point number**

CONDITION CODES: **All condition codes are undefined.**

POSSIBLE ERRORS: **None**

FUNCTION: T$FtoD converts floating-point numbers in single-precision format to double-precision format. No errors are possible and all condition codes are undefined.

T$FtoL — Single-Precision to Signed Long Integer

ASM CALL: **TCALL T$Math,T$FtoL**

INPUT: **d0.l = Single-precision floating-point number**

OUTPUT: **d0.l = Signed Long Integer**

CONDITION CODES:
N Undefined
Z Undefined
V Set on overflow
C Undefined

POSSIBLE ERRORS: **TrapV**

FUNCTION: T$FtoL converts the integer portion of the floating point number to a signed long integer. The fraction is truncated. If an overflow occurs, the V bit is set and a trapv exception is generated.

T$FtoU — Single Precision to Unsigned Long Integer

ASM CALL: **TCALL T$Math,T$FtoU**

INPUT: **d0.l = Single-precision floating-point number**

OUTPUT: **d0.l = Unsigned Long Integer**

CONDITION CODES:
N Undefined
Z Undefined
V Set on overflow
C Undefined

POSSIBLE ERRORS: **TrapV**

FUNCTION: T$FtoU converts the integer portion of the floating point number to an unsigned long integer. The fraction is truncated. If an overflow occurs, the V bit is set and a trapv exception is generated.

T$FTrn — Truncate Single-Precision Floating-Point Number

ASM CALL: **TCALL T$Math,T$FTrn**

INPUT: **d0.l = Single-precision floating-point number**

OUTPUT: **d0.l = Truncated single-precision floating-point number**

CONDITION CODES: **All condition codes are undefined.**

POSSIBLE ERRORS: **None**

FUNCTION: Floating point numbers consist of two parts: integer and fraction. The purpose of T$FTrn is to truncate the fractional part. For example, if the number passed is 283.75, this function returns 283.00.

T$LDiv — Long (Signed) Divide

ASM CALL: **TCALL T$Math,T$LDiv**

INPUT: **d0.l = Dividend**
d1.l = Divisor

OUTPUT: **d0.l = Result (d0 / d1)**

CONDITION CODES:
N Set if result is negative
Z Set if result is zero
V Set on divide by zero
C Always cleared

POSSIBLE ERRORS: **None**

FUNCTION: T$LDiv performs 32-bit integer division. A division by zero error is indicated by setting the overflow bit. If a division by zero is attempted, infinity (with the proper sign) is returned.

Positive Infinity = $7FFFFFFF
Negative Infinity = $80000000

T$LMod — Long (Signed) Modulus

ASM CALL: **TCALL T$Math,T$LMod**

INPUT: **d0.l = Dividend**
d1.l = Divisor

OUTPUT: **d0.l = Result (Mod(d0/d1))**

CONDITION CODES:
N Set if result is negative
Z Set if result is zero
V Set on divide by zero
C Always cleared

POSSIBLE ERRORS: **None**

FUNCTION: T$LMod returns the remainder (modulo) of the integer division. If an overflow occurs, the V bit is set and zero is returned.

T$LMul — Long (Signed) Multiply

ASM CALL: **TCALL T$Math,T$LMul**

INPUT: **d0.l = Multiplicand**
d1.l = Multiplier

OUTPUT: **d0.l = Result (d0 * d1)**

CONDITION CODES: **N Set if result is negative**
Z Set if result is zero
V Set on overflow
C Always cleared

POSSIBLE ERRORS: **None**

FUNCTION: T$LMul performs a 32-bit signed integer multiplication. If an overflow occurs, the V bit is set and the lower 32 bits of the result is returned. If an overflow occurs, the sign of the result is still correct.

T$Log — Natural Logarithm Function

ASM CALL: **TCALL T$Math,T$Log**

INPUT: **d0:d1 = x**
d2:d3 = Precision

OUTPUT: **d0:d1 = Log(x)**

CONDITION CODES: **C Set on error**

POSSIBLE ERRORS: **E$IllArg**

FUNCTION: T$Log returns the natural logarithm of the argument passed. If an illegal argument is passed, an error is returned.

T$Log10 Common Logarithm Function

ASM CALL: **TCALL T$Math,T$Log10**

INPUT: **d0:d1 = x**
d2:d3 = Precision

OUTPUT: **d0:d1 = Log10(x)**

CONDITION CODES: **C Set on error**

POSSIBLE ERRORS: **E$IllArg**

FUNCTION: T$Log10 returns the common logarithm of the argument passed. If an illegal argument is passed, an error is returned.

T$LtoA Signed Integer to ASCII Conversion

ASM CALL: **TCALL T$Math,T$LtoA**

INPUT: **d0.l = Signed long integer**
(a0) = Pointer to conversion buffer

OUTPUT: **(a0) = ASCII digit string**

CONDITION CODES: **N Set if the number is negative**
Z Undefined
V Undefined
C Undefined

POSSIBLE ERRORS: **None**

FUNCTION: The signed long passed to T$LtoA is converted to an ASCII string of ten (10) digits. If the number is smaller than ten digits, it is right justified and padded with leading zeros. A null is appended to the end of the string making the minimum size of the buffer eleven (11) characters.

NOTE: The N bit indicates the sign and is not included in the ASCII string.

T$LtoD — Signed Long to Double-Precision Floating-Point

ASM CALL: **TCALL T$Math,T$LtoD**

INPUT: **d0.l = Signed long integer**

OUTPUT: **d0:d1 = Double-precision floating-point number**

CONDITION CODES: **All condition codes are undefined.**

POSSIBLE ERRORS: **None**

FUNCTION: T$LtoD converts the signed integer to a double-precision float. No errors are possible and all condition codes are undefined.

T$LtoF — Signed Long to Single-Precision Floating-Point

ASM CALL: **TCALL T$Math,T$LtoF**

INPUT: **d0.l = Signed long integer**

OUTPUT: **d0.l = Single-precision floating-point number**

CONDITION CODES: **All condition codes are undefined.**

POSSIBLE ERRORS: **None**

FUNCTION: T$LtoF converts the signed integer to a single-precision float. No errors are possible and all condition codes are undefined.

T$Power — Power Function

ASM CALL: **TCALL T$Math,T$Power**

INPUT: **d0:d1 = x**
d2:d3 = y
d4:d5 = Precision

OUTPUT: **d0:d1 = x^y**

CONDITION CODES: **C Set on error**

POSSIBLE ERRORS: **E$IllArg**

FUNCTION: T$Power performs the power function on the arguments passed. That is, it raises x to the y power. If an illegal argument is passed, an error is returned.

T$Sin — Tangent Function

ASM CALL: **TCALL T$Math,T$Sin**

INPUT: **d0:d1 = x (in radians)**
d2:d3 = Precision

OUTPUT: **d0:d1 = Sin(x)**

CONDITION CODES: **C Always clear**

POSSIBLE ERRORS: **None**

FUNCTION: T$Sin returns the sine() of an angle. The angle must be specified in radians. No errors are possible, and all condition codes are undefined.

T$Sqrt — Square Root Function

ASM CALL: **TCALL T$Math,T$Sqrt**

INPUT: **d0:d1 = x**
d2:d3 = Precision

OUTPUT: **d0:d1 = Sqrt(x)**

CONDITION CODES: **C Set on error**

POSSIBLE ERRORS: **E$IllArg**

FUNCTION: T$Sqrt returns the square root of the argument passed. If an illegal argument is passed an error is returned.

T$Tan — Tangent Function

ASM CALL: **TCALL T$Math,T$Tan**

INPUT: **d0:d1 = x (in radians)**
d2:d3 = Precision

OUTPUT: **d0:d1 = Tan(x)**

CONDITION CODES: **C Always clear**

POSSIBLE ERRORS: **None**

FUNCTION: T$Tan returns the tangent() of an angle. The angle must be specified in radians. No errors are possible, and all condition codes are undefined.

T$UDiv — Unsigned Divide

ASM CALL: **TCALL T$Math,T$UDiv**

INPUT: **d0.l = Dividend**
d1.l = Divisor

OUTPUT: **d0.l = Result (d0 / d1)**

CONDITION CODES: **N Undefined**
Z Set if result is zero
V Set on divide by zero
C Always cleared

POSSIBLE ERRORS: **None**

FUNCTION: T$UDiv performs 32-bit unsigned integer division. The overflow bit is set when a division by zero error occurs. If a division by zero is attempted, infinity ($FFFFFFFF) is returned.

T$UMod — Unsigned Modulus

ASM CALL: **TCALL T$Math,T$UMod**

INPUT: **d0.l = Dividend**
d1.l = Divisor

OUTPUT: **d0.l = Result (Mod(d0/d1))**

CONDITION CODES: **N Undefined**
Z Set if result is zero
V Set on divide by zero
C Always cleared

POSSIBLE ERRORS: **None**

FUNCTION: T$UMod returns the remainder (modulo) of the integer division. If an overflow occurs, the V bit is set and zero is returned.

T$UMul — Unsigned Multiply

ASM CALL: **TCALL T$Math,T$UMul**

INPUT: **d0.l = Multiplicand**
d1.l = Multiplier

OUTPUT: **d0.l = Result (d0 * d1)**

CONDITION CODES: **N Undefined**
Z Set if result is zero
V Set on overflow
C Always cleared

POSSIBLE ERRORS: **None**

FUNCTION: T$UMul performs a 32-bit unsigned integer multiplication. If an overflow occurs, the V bit is set and the lower 32 bits of the result is returned.

T$UtoA — Unsigned Integer to ASCII Conversion

ASM CALL: **TCALL T$Math,T$UtoA**

INPUT: **d0.l = Unsigned long integer**
(a0) = Pointer to conversion buffer

OUTPUT: **(a0) = ASCII digit string**

CONDITION CODES: **All condition codes are undefined.**

POSSIBLE ERRORS: **None**

FUNCTION: The unsigned long passed to T$UtoA is converted to an ASCII string of ten digits. If the number is smaller than ten digits, it is right justified and padded with leading zeros. A null is appended to the end of the string, making the minimum size of the buffer eleven characters.

T$UtoD Unsigned Long to Double-Precision Floating-Point

ASM CALL: **TCALL T$Math,T$UtoD**

INPUT: **d0.l = Unsigned long integer**

OUTPUT: **d0:d1 = Double-precision floating-point number**

CONDITION CODES: **All condition codes are undefined.**

POSSIBLE ERRORS: **None**

FUNCTION: T$UtoD converts the unsigned integer to a double-precision float. No errors are possible and all condition codes are undefined.

T$UtoF Unsigned Long to Single-Precision Floating-Point

ASM CALL: **TCALL T$Math,T$UtoF**

INPUT: **d0.l = Unsigned long integer**

OUTPUT: **d0.l = Single-precision floating-point number**

CONDITION CODES: **All condition codes are undefined.**

POSSIBLE ERRORS: **None**

FUNCTION: T$UtoF converts the unsigned integer to a single-precision float. No errors are possible and all condition codes are undefined.

End of Chapter 6

NOTES