A Report on the Accuracy of PR1ME Computers' Floating Point Software and Hardware

- and -

The SWT Math Library User's Guide

Technical Report GIT-ICS-83/09

Eugene H. Spafford

School of Information and Computer Science
Georgia Institute of Technology
Atlanta, Georgia 30332

April 24, 1983

TABLE OF CONTENTS

The Hardware

The SWT Math Library

Testing

Appendix I

Appendix II

Appendix III

Appendix IV

ADDENDUM

Introduction

Users of Prime computers have been aware for some time of a number of shortcomings in the floating point arithmetic firmware. In addition, there have been a number of inaccuracies found in the standard math libraries which have gone uncorrected for years ({1}, {2}). Unlike other major computer firms, Prime has not published any documents dealing with the algorithms or error analysis of their math routines.

In the winter of 1982 I undertook the coding of a new math library to support the Georgia Tech SWT Pascal compiler, and the Georgia Tech C compiler. The results of tests on that library and the standard Prime libraries have revealed a number of interesting facts. Additionally, further experimentation with the floating point mechanisms has revealed some bugs in the way arithmetic is performed, in some cases.

First, this guide describes the architecture of the floating point mechanism, including some error analysis and description of quirks in the hardware. This includes a description of incompatibilities between the 400/550 cpu and the 750/850 cpu floating point register structure. Next is a description of the SWT Math library. Last is a discussion of some preliminary error analysis of the SWT library and the Prime standard library functions. The appendices contain information on auxiliary programs supplied with the library which will aid users in writing their own routines, and checking existing routines and floating point firmware.

Acknowledgements

I would like to thank Roy Mongiovi for his help in debugging some of the SWT Math routines, and Peter Wan for his help in preparing this guide. I would also like to thank Ann Vitale, Ron Kurtzer, and especially Emory Stevens of the Atlanta Prime office for their co-operation and aid in the testing of these routines.

Research contributing to the development of this report was conducted while the author was receiving a National Science Foundation Graduate Fellowship, support which is gratefully acknowledged.

The Hardware

Internal Representation of Floating Point Values

There are two basic forms of floating point representation on the Prime: single precision and double precision. Both forms are stored in memory and the registers in about the same manner. It should be noted, however, that the storage format **in memory** and the storage format **in the registers** are different from each other. Also, the representation of values is different on 750/850 models than on the others.

Note that both forms of floating point values are available in three of the four Prime addressing modes: R, V and I. For purposes of this discussion, assume that all references are being made to the V mode instructions and registers unless noted otherwise. Also note that when I refer to the 400/550 machines, this also includes the 550-II.

The reader might be interested in perusing {12} through {15} for information about the proposed IEEE 754 standard on floating point representation. These articles also contain information about internal representation and accuracy of results. As a matter of interest, Prime Computer, Inc. had two voting representatives on the committee.

Storage Formats

A floating point value consists of three parts: a sign, a normalized mantissa, and an exponent. The mantissa is a two's complement value with an implied leading binary point (radix point). A normalized mantissa always represents a value in the interval [0.5, 1) unless it represents zero. The sign bit is set to indicate a negative value, reset to indicate a positive value. The sign bit is always in the most significant bit position (bit one). Following the sign bit is the mantissa.

A single precision value consists of the sign bit, 23 mantissa bits, and 8 exponent bits. The sign bit is bit one, the mantissa is bits 2 to 24, and the exponent is bits 25 to 32. The exponent is stored in excess-128 representation. That is, the value stored in the 8 bits of the exponent, if viewed as a two's complement value, is always 128 greater than the value it represents. Thus,

0 0 0 0 0 0 0 0	represents -128
1 1 1 1 1 1 1 1	represents 127
1 0 0 0 0 0 0 0	represents 0

```
1 0 0 0 0 1 0 0     represents    4

0 1 1 1 1 1 0 0     represents   -4
```

This implies that the largest possible exponent is +127, and the smallest possible exponent is -128. The exponent is taken to the base 2. (You may wish to refer to a reference such as {3} or {4} for more information about value representations.)

A double precision value consists of the sign bit, a 47 bit mantissa, and 16 exponent bits. The sign bit is bit one, the mantissa is bits 2 to 48, and the exponent is bits 49 to 64. The exponent is stored as a 128-biased value. This is similar to excess-128 except that the most significant bit of the exponent is taken as a sign bit. Thus,

```
0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0    represents      0

0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0    represents      4

0 0 0 0 0 0 0 0 0 1 1 1 1 1 0 0    represents     -4

1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0    represents -32896

0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1    represents  32639
```

As you can see from the examples, the range of the exponent is larger in the negative direction than in the positive. This means that it is possible to have values in the register whose multiplicative inverses cannot be represented.

Normalization

Every arithmetic operation on a floating point value causes the mantissa to be normalized. **On the Primes** normalization means that the mantissa is shifted towards the sign bit until the bit next to the sign bit is different from the sign bit. The exponent is decreased by the same amount as the number of places shifted. Normalization does **not** always mean shifting until a "1" is present in the second bit.

Let us examine an example. Suppose we have just completed a single precision add, and the result is either 5 1/2 or -5 1/2 as follows:

```
0 00010110000000000000000 10000110     5.5
1 11101010000000000000000 10000110    -5.5
```

Neither of these values is normalized. The mantissa is shifted left until its first bit is different from the sign bit. Note that it takes exactly 3 such shifts for each value:

```
          0 10110000000000000000 10000011     5.5
          1 01010000000000000000 10000011    -5.5
```

Both of these values are now normalized. The value of each is unchanged. There is no assumed first bit as on some machines (such as certain PDP machines).

Normalization helps maintain accuracy of results between computations. Additionally, comparisons between floating point numbers is made much easier -- a zero can always be recognized by examining the first word of the value only, and comparison between two floating point numbers can sometimes be done by a simple compare of the exponents and mantissa sign. It also helps to ensure that only one of the two values needs to be adjusted prior to some arithmetic operations (such as add).

A special case is when the sign bit is one (a negative value) and every bit of the mantissa is zero. This is not equal to zero, but rather is equal to -0.5 (assuming the exponent represents zero, of course).

It should be noted that load and store operations do not cause the register contents to be normalized. There is also no "normalize" instruction which will allow the user to normalize the bit pattern in the register.

Floating skip operations (eg, FSGT, FSZE) and comparison operations (eg, FCS and DFCS) will not work correctly unless the values involved are normalized.

Representation in the Registers

The single precision floating point register has more range than can be accommodated in the memory format. The single precision floating point register overlaps the double precision register and uses the extra bits available in the double floating register as guard bits. The register is organized as follows on 400/550 cpus:

```
S MMMMMMMMMMMMMMMMMMMMMMM GGGGGGGG HHHHHHHH EEEEEEEE 0000000000000000
1      2..24                25..32   33..40   41..48      49..64
```

Where:

- S is sign of the mantissa
- M is the mantissa (2's complement)
- G is mantissa extension (guard bits)
- H is exponent extension (guard bits)
- E is exponent (128-biased)
- 0 extra bits -- must be zero

On 750 and 850 cpus (with hardware floating point) the organization is:

```
S MMMMMMMMMMMMMMMMMMMMMMM GGGGGGGGGGGGGGGGGGGGGGGG HHHHHHHH EEEEEEEE
1      2..24                    25..48              49..56   57..64
```

Where:
- S is sign of the mantissa
- M is the mantissa (2's complement)
- G is mantissa extension (guard bits)
- H is exponent extension (guard bits)
- E is exponent (excess 128)
- 0 extra bits -- must be zero

The guard bits are always zeroed whenever a floating load operation is done (FLD). The high-order guard bit may be used to round the least significant bit of the regular mantissa just before storage by using the FRN instruction. This increases accuracy somewhat at the cost of increased execution time. See the section on "Firmware Accuracy" for more details.

Double precision floating point values are similar in nature to single precision and are organized as follows on 400/550 machines:

```
S MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM EEEEEEEEEEEEEEEE MMMMMMMMMMMMMMMM
1             2..32                    33..48           49..64
```

Where:
- S is sign of the mantissa
- M is the mantissa (2's complement)
- E is exponent (excess 128, two's complement)

On 750 and 850 machines, the double precision register is organized as:

```
S MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM EEEEEEEEEEEEEEEE
1                     2..48                                49..64
```

Where:
- S is sign of the mantissa
- M is the mantissa (2's complement)
- E is exponent (128 biased)

Access Methods

Besides the standard load and store instructions, it is possible to access portions of the floating point registers with integer operations. These accesses are done either through the use of P300 address traps, or through the LDLR/STLR instructions.

If short memory references are made to locations 4, 5, and 6, the instructions actually are accessing the first two words of the mantissa and the exponent, respectively. In single precision references, the reference to the exponent fetches both the exponent and exponent guard bits. In double precision, the reference to location 6 fetches the complete exponent. Thus, the PMA sequence:

```
LDA   ='40000
STA#  4
CRA
STA#  5
LDA   =128
STA#  6
```

results in the value 0.5 being in the single precision floating register (note that this sequence also loads all the guard bits correctly on a 400/550).

It is also possible to access the floating point register via the LDLR and STLR instructions. In V mode, the first two words (bits 1 to 32) of the mantissa can be loaded into the L register by loading from register file location '12. The third word of the mantissa and the exponent can be obtained by loading from location '13. **The organization of the register file on 750/850 machines and 400/550 machines means that the L register contents after a "LDLR '13" will be different on these machines.** On 400/550 machines, the A register will contain the exponent and the B register will contain the third word of the mantissa. On a 750/850 these will be reversed. The program in Appendix I can be used to discover which case is present on your machine. When dealing with the two floating accumulators in I mode addressing, a "LDLR '11" will have the same problem.

Additionally, the floating accumulator shares the same register file location as the second field address and length registers (in the V mode register file). In the I mode registers, the first floating accumulator shares the same location as the first field address register, and the second floating accumulator shares the same location as the second field address register. Thus, various character manipulation instructions including decimal (character) arithmetic instructions may change the floating accumulators as a side effect.

Ranges

The effective range for single precision floating point values is approximately 1.701412 * (10 ** 38) to -1.701412 * (10 **38). The smallest, non-zero magnitude that can be represented is approximately 1.469368 * (10 ** -39). **This is the range for single precision storage in memory.** The guard bits in the register give extended range to values held in the register.

Effective range for double precision floating point values is approximately 2.079833 * (10 ** 9825) to -2.079833 * (10 ** 9825). The smallest, non-zero magnitude that can be represented is approximately 1.03808 * (10 ** -9903).

Available Operations

The following lists describe the instructions available on Prime 50 series machines to manipulate floating point values in 64V mode. This material has been extracted from the paper 64V Mode Instruction Summary and Addressing Formats, by T. Allen Akin, Perry Flinn, and Eugene Spafford, Georgia Tech 1981. The abbreviation FAC refers to the floating accumulator, meaning the combination (overlapped) register. The instructions will be presented first grouped by function, then alphabetically. In the following instruction set summary, instruction formats are abbreviated as follows:

branch	branch
gen	generic
mr	memory reference

The descriptions of restricted instructions are preceded by an asterisk (*). Note that these instructions are not restricted unless segmented memory is turned on (bit 14 in current modals) and only if a reference is made outside of the range '0 to '17 (zero to 15, decimal).

In the descriptions of effects on the C-bit, L-bit, and condition codes, the following abbreviations are used:

C-bit:

-	unchanged
V	arithmetic overflow indication
X	indeterminate

L-bit:

-	unchanged
X	indeterminate

Condition Codes (CC):

-	unchanged
S	properly set to reflect value of result, may be used for condition code branches
X	indeterminate

Branch

Mnemonic	Format	C	L	CC	Description
BFEQ	branch	-	-	S	branch if FAC = 0
BFGE	branch	-	-	S	branch if FAC >= 0
BFGT	branch	-	-	S	branch if FAC > 0
BFLE	branch	-	-	S	branch if FAC <= 0
BFLT	branch	-	-	S	branch if FAC < 0
BFNE	branch	-	-	S	branch if FAC <> 0

Floating Point Arithmetic

Mnemonic	Format	C	L	CC	Description
FAD	mr	V	X	X	add memory to single precision FAC
FCM	gen	V	X	X	complement single precision FAC arithmetically
FDBL	gen	-	-	-	convert single precision floating to double precision
FDV	mr	V	X	X	divide memory into single precision FAC
FLTA	gen	V	X	X	convert 16 bit integer to single precision float
FLTL	gen	V	X	X	convert 32 bit integer to single precision float
FMP	mr	V	X	X	multiply single precision FAC by memory
FRN	gen	V	X	X	floating round double to single
FSB	mr	V	X	X	subtract memory from single precision FAC

Mnemonic	Format	C	L	CC	Description
FCS	mr	X	X	X	compare single precision FAC to memory and skip
FLD	mr	-	-	-	load single precision FAC from memory
FLX	mr	-	-	-	load double word index
FST	mr	V	X	-	store single precision FAC into memory
INTA	gen	V	X	X	convert single precision FAC to 16 bit integer
INTL	gen	V	X	X	convert single precision FAC to 32 bit integer

Mnemonic	Format	C	L	CC	Description
DFAD	mr	V	X	X	add memory to double precision FAC
DFCM	gen	V	X	X	complement double precision FAC arithmetically
DFDV	mr	V	X	X	divide memory into double precision FAC
DFMP	mr	V	X	X	multiply double precision FAC by memory
DFSB	mr	V	X	X	subtract memory from double precision FAC
FDBL	gen	-	-	-	convert single precision floating to double precision
FRN	gen	V	X	X	floating round double to single

Mnemonic	Format	C	L	CC	Description
DFCS	mr	X	X	X	compare double precision FAC with memory and skip
DFLD	mr	-	-	-	load double precision FAC
DFLX	mr	-	-	-	load quadruple word index
DFST	mr	-	-	-	store double precision FAC

Logicize

Mnemonic	Format	C	L	CC	Description
LFEQ	gen	-	-	S	set A to 1 if FAC = 0; else reset A to 0
LFGE	gen	-	-	S	set A to 1 if FAC >= 0; else reset A to 0
LFGT	gen	-	-	S	set A to 1 if FAC > 0; else reset A to 0
LFLE	gen	-	-	S	set A to 1 if FAC <= 0; else reset A to 0
LFLT	gen	-	-	S	set A to 1 if FAC < 0; else reset A to 0
LFNE	gen	-	-	S	set A to 1 if FAC <> 0; else reset A to 0

Skip

Mnemonic	Format	C	L	CC	Description
FSGT	gen	-	-	-	skip if FAC > 0
FSLE	gen	-	-	-	skip if FAC <= 0
FSMI	gen	-	-	-	skip if FAC < 0
FSNZ	gen	-	-	-	skip if FAC <> 0
FSPL	gen	-	-	-	skip if FAC >= 0
FSZE	gen	-	-	-	skip if FAC = 0

Mnemonic	Format	C	L	CC	Description
DFCS	mr	X	X	X	compare double precision FAC with memory and skip
FCS	mr	X	X	X	compare single precision FAC to memory and skip

Data Movement

Mnemonic	Format	C	L	CC	Description
DFLD	mr	-	-	-	load double precision FAC
DFLX	mr	-	-	-	load quadruple word index
DFST	mr	-	-	-	store double precision FAC
FLD	mr	-	-	-	load single precision FAC from memory
FLX	mr	-	-	-	load double word index
FST	mr	V	X	-	store single precision FAC into memory
LDLR	mr	-	-	-	*load L from register file
STLR	mr	-	-	-	*store L into register file

Address Manipulation

Mnemonic	Format	C	L	CC	Description
DFLX	mr	-	-	-	load quadruple word index
FLX	mr	-	-	-	load double word index

Type Conversion

Mnemonic	Format	C	L	CC	Description
FDBL	gen	-	-	-	convert single precision floating to double precision
FLTA	gen	V	X	X	convert 16 bit integer to single precision float
FLTL	gen	V	X	X	convert 32 bit integer to single precision float
FRN	gen	V	X	X	floating round double to single
INTA	gen	V	X	X	convert single precision FAC to 16 bit integer
INTL	gen	V	X	X	convert single precision FAC to 32 bit integer

Instructions Grouped Alphabetically

Mnemonic	Format	C	L	CC	Description
BFEQ	branch	-	-	S	branch if FAC = 0
BFGE	branch	-	-	S	branch if FAC >= 0
BFGT	branch	-	-	S	branch if FAC > 0
BFLE	branch	-	-	S	branch if FAC <= 0
BFLT	branch	-	-	S	branch if FAC < 0
BFNE	branch	-	-	S	branch if FAC <> 0
DFAD	mr	V	X	X	add memory to double precision FAC
DFCM	gen	V	X	X	complement double precision FAC arithmetically
DFCS	mr	X	X	X	compare double precision FAC with memory and skip
DFDV	mr	V	X	X	divide memory into double precision FAC
DFLD	mr	-	-	-	load double precision FAC
DFLX	mr	-	-	-	load quadruple word index
DFMP	mr	V	X	X	multiply double precision FAC by memory
DFSB	mr	V	X	X	subtract memory from double precision FAC
DFST	mr	-	-	-	store double precision FAC
FAD	mr	V	X	X	add memory to single precision FAC
FCM	gen	V	X	X	complement single precision FAC arithmetically
FCS	mr	X	X	X	compare single precision FAC to memory and skip
FDBL	gen	-	-	-	convert single precision floating to double precision
FDV	mr	V	X	X	divide memory into single precision FAC
FLD	mr	-	-	-	load single precision FAC from memory
FLTA	gen	V	X	X	convert 16 bit integer to single precision float
FLTL	gen	V	X	X	convert 32 bit integer to single precision float
FLX	mr	-	-	-	load double word index
FMP	mr	V	X	X	multiply single precision FAC by memory
FRN	gen	V	X	X	floating round double to single
FSB	mr	V	X	X	subtract memory from single precision FAC
FSGT	gen	-	-	-	skip if FAC > 0

FSLE	gen	-	-	-	skip if FAC <= 0
FSMI	gen	-	-	-	skip if FAC < 0
FSNZ	gen	-	-	-	skip if FAC <> 0
FSPL	gen	-	-	-	skip if FAC >= 0
FST	mr	V	X	-	store single precision FAC into memory
FSZE	gen	-	-	-	skip if FAC = 0
INTA	gen	V	X	X	convert single precision FAC to 16 bit integer
INTL	gen	V	X	X	convert single precision FAC to 32 bit integer
LDLR	mr	-	-	-	*load L from register file
LFEQ	gen	-	-	S	set A to 1 if FAC = 0; else reset A to 0
LFGE	gen	-	-	S	set A to 1 if FAC >= 0; else reset A to 0
LFGT	gen	-	-	S	set A to 1 if FAC > 0; else reset A to 0
LFLE	gen	-	-	S	set A to 1 if FAC <= 0; else reset A to 0
LFLT	gen	-	-	S	set A to 1 if FAC < 0; else reset A to 0
LFNE	gen	-	-	S	set A to 1 if FAC <> 0; else reset A to 0
STLR	mr	-	-	-	*store L into register file

Error Handling

There are basically four floating point errors determined by the floating point firmware: store exception, overflow, underflow, and divide by zero. The action on these errors is determined by the state of bit 7 in the current cpu keys. If bit 7 is set, a floating point fault simply sets the C bit and no other action is taken. If bit 7 is reset, then an arithmetic fault is signalled and the standard fault handler invoked. In Primos, this usually entails signalling the ERROR condition.

A store exception is triggered when an attempt is made to FST (single precision floating store) a value which is too big or too small (negative) to be accomodated in the two word memory format used by single precision values. This can happen because the value in the floating point register may have been loaded or generated using double precision operations. Alternatively, the value in the register could have been generated by single precision operations, but the value is larger than the memory format can accomodate due to the extra capacity provided by the guard bits. A double precision store cannot cause a store exception.

Overflow and underflow operations are the result of arithmetic operations (add, subtract, multiply, or divide) whose result is too big or too small to fit (normalized) in the register. Thus, the exponent of the result must be bigger than 32639 for overflow (base 2 exponent), or less than -32896 for

underflow (see the next section).

A divide by zero fault is exactly what the name implies -- an attempt to divide by a floating point value, single or double precision, whose value is identical to zero.

Another type of fault, not strictly a floating point fault, is triggered when an attempt is made to convert a floating value to an integer, and the floating value is too big or too small to be held in the corresponding integer register.

It is possible for user programs to set bit 7 in the keys to ignore these fault conditions, but in doing so the user should realize that results could be invalid without any indication of error. Explicit checks should be made of the C bit after any operation which might cause an error. By default, the standard compilers and the PMA assembler generate entry control blocks (ECBs) for procedures with bit 7 reset to zero.

Firmware Accuracy

In this document, the word "firmware" refers to the microcode or hardware which performs the floating point arithmetic. 750 and 850 cpus have floating point operations implemented in hardware, while the other models have these operations implemented in microcode. Programs and subroutines depend on the accuracy of these operations, so it is crucial that these operations be implemented correctly.

Problems in Multiplication

There appears to be a bug in the double precision floating point multiplication at a few points near the maximum value. If a value whose base 2 exponent is 32639 (maximum possible) is multiplied by a value greater than 0.5, an overflow fault is triggered. Thus, it is possible to multiply a value in the register by something less than 1.0 and get an overflow! In some cases, attempting to multiply smaller values to yield a value theoretically in range also results in an overflow. We have not attempted exhaustive testing to determine limits where this occurs since the likelihood of encountering such an error is small. However, the problem is there, and the user is advised to be careful when writing tests which need to deal with values at the upper limit of register capacity.

A much more serious flaw is to be found in the DFMP instruction on 400/550 machines. The double floating multiply instruction appears to <u>always</u> return a result whose two least significant bits of the mantissa are zero. That is, every multiplication potentially loses 2 out of 47 bits of precision! It is possible to multiply a value by 1.0 and not obtain a result equal to the original value. Such errors can, of course, cascade

and result in severe accuracy problems in chains of calculations. The hardware on 750/850 machines appears to be free of this defect. Appendix II contains a program to test your machine and illustrate this problem.

Oddly enough, division on the 400/550 machines does not appear to truncate any bits of precision, and according to published timing figures {5} the DFDV instruction is just as fast (slow) as the DFMP instruction. Thus, it might be advisable to recode critical calculations on these machines to be composed of divisions rather than multiplications, whenever possible.

Loss of Precision in Type Conversion

When converting from integers to floating point there are basically two machine instructions: FLTA and FLTL. The FLTA instruction converts a 16 bit integer into a single precision floating point value (24 bit mantissa). The FLTL instruction converts a 32 bit integer into a single precision floating point value. Note that such a conversion potentially drops 8 bits of precision. There is adequate storage in the double precision floating point register to convert without a loss of precision, but there is no instruction to convert from long integer to double precision real. Rather, the conversion must be done by a series of instructions; see the code for the SWT 'dble$m' routine.

Problems in the Other Operations

We have not observed any loss of precision in the addition, subtraction or division of double precision quantities. We have also not been able to detect any precision losses in any of the single precision operations. However, this does not indicate the absence of errors, rather it just indicates that we have not extensively tested for such errors and none have appeared in any of our other tests.

Floating Round

Studies performed at The Flinders University of South Australia on a 750 have indicated that some calculations performed in single precision floating point may benefit from the fact that the register contains extra precision, but that the results may be somewhat uneven depending on how the code is organized {6}. Their studies have also indicated that use of the FRN (floating round) instruction before each store greatly enhances the accuracy of some calculations in single precision:

> "In fact for the single precision problem a simple and almost complete cure for the problem has been demonstrated, and that is for the compiler to force a round before every store (i.e. emit an FRN instruction before each FST instruction emitted)." {6}

Their studies also indicated that the double precision arithmetic failed to do correct rounding. In fact, double precision operations truncate their results rather than rounding (see the next section). This leads to slightly skewed results which are especially noticeable in problems requiring very precise results:

> "... Consequently the Prime-750 exhibits a far larger error than the VAX-11/780 when we use the sum of squares measure. This error has been detected by our users in other calculations and programs and is particularly critical when nearly unstable matrix problems are investigated.... The consequences of such inaccuracy in a research-oriented application area could be critical." {6}

That conclusion was made for a 750 with hardware floating point operations. It can certainly be concluded that a 400 or 550 is not at all appropriate for double precision calculations requiring any high degree of accuracy.

Precision

The various models of Prime computer perform floating point operations to slightly different precisions. To quote from section 6.2.1 of {9}:

> "In double and single precision add, subtract, and multiply operations, the 750 and 850 truncate results to 48 sign and magnitude bits. Single precision divide operations on these processors produce 32 sign and magnitude bits of rounded result....
>
> Double precision operations on the 500-II (and 650) are identical to those performed on the 750 and 850. Single precision divide is also identical to 750/850 single precision divide. Single precision add, subtract, and multiply operations truncate results to 32 sign and magnitude bits.
>
> For all other 50 Series systems, double precision add and subtract operations truncate results to 48 sign and magnitude bits; multiply and divide operations truncate to 47 sign and magnitude bits. All single precision operations on these processors truncate results to 32 sign and magnitude bits."

These statements tend to raise serious doubts about the accuracy of similar programs run on different model machines due to precision changes. It also would indicate that some program behavior might change when run on a different model cpu.

The SWT Math Library

In General

The Software Tools Subsystem (SWT) is a major software package developed at Georgia Tech for Prime 50-series machines. It includes an advanced command interpreter with command pipes and i/o re-direction, a full screen editor with advanced regular pattern matching and replacement, and a large library of utility routines. One of the libraries which is to be included in further releases of the Subsystem is the SWT Math Library.

The SWT Math Library contains thoroughly tested routines to calculate various useful functions, including standard trigonometric functions. All of the routines share a number of common features which will be described in the next section. The individual routines will be described in the sections following.

Source

Most of the routines were obtained from the book Software Manual for the Elementary Functions by William Cody, Jr. and William Waite {7}. The random number generator was written utilizing material from {8}, and a few routines such as 'dble$m' and 'dint$m' were developed by the author. Testing of the routines is described in the next chapter.

Implementation

All of the SWT Math routines have been coded in Prime assembly language. Although this may make the code somewhat harder to read, it helps to enhance the accuracy and efficiency of the routines. A number of actions, such as direct manipulation of the exponent in the register file, are not available in higher level languages and this was a major factor in the decision to use assembler.

One factor which helps to increase the accuracy of the routines is the manner in which constants for the routines were obtained. Almost all of the constants used in the SWT Math Library are given as hexadecimal data constants in the assembly language programs. These values were derived from the constants given in {7} and the program in Appendix III. The program in Appendix III was run on a Cyber 760 which has over 90 bits of precision in the mantissa of double precision floating point values. The program calculates the proper rounded representation of the given input constants and returns the appropriate hex values.

It is interesting to note that some of the standard Prime library routines were also derived from {7} but many of the constants are given in the source code as decimal values. Tests by the author indicate that the PMA assembler does not always translate double precision decimal values into the correct bit pattern, thus inducing error.

Timing

One factor that is of interest to users of any math package is that of the efficiency of the code. Unfortunately, it is not possible to make a direct comparison of the speed of routines in the SWT Math Library to that of equivalent routines in the standard Prime libraries. The Prime native compilers are able to generate special "shortcalls" to known library subprograms which enhance their apparent speed. The SWT Math library routines are all done as regular procedure calls and will thus appear much slower if compared directly. The only statement that can be made about the efficiency of these routines is that they were coded in PMA by someone expert in that language, and they have been optimized as much as possible without sacrificing accuracy.

Naming and Function

All of the functions in the SWT Math Library return double precision values. Most of the functions have two entry points for every calculation -- one for a single precision argument and one for a double precision argument. The routines which take single precision arguments do argument verification and will not return a value which is out of range for a single precision floating point value. Thus, the value returned by those functions can be considered to be single precision. Since the single and double precision registers overlap, it is trivial to use these functions as either single or double precision.

In general, routines whose names begin with the letter 'd' are intended to take double precision arguments. Specific considerations are given in the sections below.

Errors

In the standard Prime library routines, calling a function with an improper value (such as trying to take the square root of a negative value) results in a signal to the condition ERROR. This signal cannot be returned from and thus execution of the program is terminated. Furthermore, the nature of the error and the routine involved is not well specified. In the Fortran 66 library the cause of the error is better identified but the general result is the same.

In the SWT Math Library whenever an error condition is encountered, the condition SWT_MATH_ERROR$ is signalled. The "ms" structure indicated by the call to SIGNL$ is the stack frame

of the routine calling the math routine, and the "info" structure is composed of the faulty argument (4 words), default return value (4 words), and a pointer (2 words) to a message describing the error. The user may specify an on-unit which can examine and change the default return value. The signal can be returned from and thus execution may continue.

The routine 'err$m' is a default on-unit handler which can be used to print the name of the faulting routine and the value of the faulting argument. This guide is not intended to present the information necessary to understand the Prime on-unit mechanism, so the interested reader is directed to the code for 'err$m' and to {10}.

Each routine sets the 'owner' pointer at offset 18 within the stack frame, and each routine has its ECB labelled according to standard conventions. Thus, the Primos DMSTK command will print the names of activations of SWT Math Library routines, as will programs such as DBG.

To the best of my knowledge, no error can occur during the execution of any of the SWT Math routines which does not signal the condition SWT_MATH_ERROR$. Thus, unlike many of the Prime routines, the user will not encounter errors such as 'SIZE' or 'OVERFLOW' during execution of these routines (see the section on Tests for more specific details).

The Routines

ACOS$M and DACS$M

These two functions calculate the inverse cosine of an angle. The argument to the functions is the cosine of the angle, and the function returns the measure of the angle, in radians. The 'dacs$m' function expects a double precision argument, and the 'acos$m' function expects a single precision argument. Arguments to the functions must be in the closed interval [-1.0, 1.0] or else the condition SWT_MATH_ERROR$ is signalled. In the case of an error, the default return value is zero.

The functions are implemented as rational minimax approximations on a modified argument value.

ASIN$M and DASN$M

These two functions calculate the inverse sine of an angle. The argument to the functions is the sine of the angle, and the function returns the measure of the angle, in radians. The 'dasn$m' function expects a double precision argument, and the 'asin$m' function expects a single precision argument. Arguments to the functions must be in the closed interval [-1.0, 1.0] or else the condition SWT_MATH_ERROR$ is signalled. If an error is

signalled, the default function value is zero.

The functions are implemented as rational minimax approximations on a modified argument value.

ATAN$M and DATN$M

These two functions calculate the inverse tangent of an angle. The argument to the functions is the tangent of the angle, and the function returns the measure of the angle, in radians. The 'datn$m' function expects a double precision argument, and the 'atan$m' function expects a single precision argument. The functions will not signal any errors based on input values.

The functions are implemented as a rational approximation on a modified argument value. Note that there is no equivalent to the ATAN2 function which is available in some implementations of Fortran; if users wish such a function, they may construct it from this function.

COS$M and DCOS$M

These two functions return the cosine of the angle whose measure (in radians) is given by the argument. The 'dcos$m' routine expects a double precision argument, and the 'cos$m' routine expects a single precision argument. If the absolute value of the angle plus one-half pi is greater than 26353588.0 then the condition SWT_MATH_ERROR$ is signalled. If an error is signalled, the default function return is zero.

The functions are implemented as minimax polynomial approximations.

COSH$M and DCSH$M

These two routines calculate the hyperbolic cosine of their arguments, defined as cosh(x) = [exp(x) + exp(-x)]/2. The function 'dcsh$m' expects a double precision value as argument, and the 'cosh$m' function expects a single precision argument. The condition SWT_MATH_ERROR$ is signalled if the absolute value of the argument is greater than 22623.630826296. In the single precision case, arguments which produce a value too large for single precision storage will also signal the error condition. If an error is signalled, the default function value is zero.

COT$M and DCOT$M

These two functions calculate the cotangent of the angle whose measure is given (in radians) as the argument to the functions. The 'dcot$m' function expects a double precision argument, and the 'cot$m' routine expects a single precision

argument. The arguments must have an absolute value greater than 7.064835966E-9865 and less than 13176794.0 or else the condition SWT_MATH_ERROR$ will be signalled. If an error is signalled, the default function return is zero.

The functions are calculated based on a minimax polynomial approximation over a reduced argument.

DBLE$M

The 'dble$m' function implements something akin to the Fortran 66 'dble' function, or the Fortran 77 'dreal' function. It takes as an argument a 32 bit integer and returns a double precision floating point number of the same value. This function should always be used when converting 32 bit integers to double precision real numbers because the code generated by some of the compilers will (potentially) lose up to 8 bits of mantissa precision (see the discussion in the previous chapter).

The 'dble$m' function has no single precision counterpart in this library. The routine, as defined, does not recognize or signal any error conditions. It is written so as to work on both 550 and 750 style machines, despite the internal difference in register structure.

The algorithm involved was derived from known register structure by the author.

DINT$M

The 'dint$m' function implements the Fortran 'dint' function. That is, it takes one double precision value and resets bits in the mantissa to remove any fractional part of the value. The return value is a double precision real. This routine also has a shortcall (JSXB) entrance labelled 'dint$p' which is used in some of the other math routines; users should not attempt to use this shortcall entrance unless they are aware of its structure.

The 'dint$m' of 1.5 is 1.0, the 'dint$m' of -1.5 is -1.0, and the 'dint$m' of anything less than 1.0 and greater than -1.0 is equal to zero.

The dint$m function has no single precision counterpart in this library. The routine, as defined, does not recognize or signal any error conditions. It is written so as to work of both 550 and 750 style machines, despite the internal difference in register structure.

The algorithm involved was developed by the author based on the known register structure.

ERR$M

The 'err$m' procedure is provided as a default handler for the SWT_MATH_ERROR$ condition. It takes a single argument, a 2 word pointer as defined by the condition mechanism, and prints information about the routine and values which signalled the fault. All output from the 'err$m' routine is sent to SWT ERROUT. Included in the output is the name of the faulting routine, the location from which the faulting routine was called, the value of the argument involved, and the default return value to be used.

The following code illustrates how to set up this default handler for use in Fortran 66 programs:

```
      EXTERNAL ERR$M

      CALL MKON$F ('SWT_MATH_ERROR$', 15, ERR$M)
```

The following code illustrates how to set up this default handler for use in Fortran 77 programs:

```
      EXTERNAL ERR$M, MKON$P

      CALL MKON$P ('SWT_MATH_ERROR$', 15, ERR$M)
```

The user may wish to copy and modify the source code for the 'err$m' procedure so as to provide a more specific form of error handling. If this is done, it would probably be a good idea to rename the user's version to something other than 'err$m.'

EXP$M and DEXP$M

These two functions implement the inverse of the 'ln$m' and 'dln$m' functions. That is, they raise the constant **e** to the power of the argument. The 'dexp$m' function takes a double precision argument, and the 'exp$m' function takes a single precision argument. Arguments to the 'exp$m' routine must be in the closed interval [-89.415985, 88.029678] and arguments to the 'dexp$m' routine must be in the closed interval [-22802.46279888, 22623.630826296], or else the SWT_MATH_ERROR$ condition will be signalled. If an error is signalled, the default function return value is zero.

It should be noted that the functions could simply return zero for sufficiently small arguments rather than signalling an error since the actual function value would be indistinguishable from zero to the precision of the machine. However, there is no mapping to zero in the actual function, and that is why the function signals an error in this case.

The routines are implemented as a functional approximation performed on a reduction of the argument.

LN$M and DLN$M

These two functions implement the natural logarithm (base **e**) function. The 'ln$m' function works for single precision arguments, and the 'dln$m' function works for double precision arguments. Arguments less than or equal to zero will signal the SWT_MATH_ERROR$ condition; the default return is the log of the absolute value of the argument, or zero in the case of a zero argument.

The algorithm involved uses a minimax rational approximation on a reduction of the argument. All positive inputs will return a valid result.

LOG$M and DLOG$M

These two functions implement the common logarithm (base 10) function. The 'log$m' function works for single precision arguments, and the 'dlog$m' function works for double precision arguments. Arguments less than or equal to zero will signal the SWT_MATH_ERROR$ condition; the default return is the log of the absolute value of the argument, or zero in the case of a zero argument.

The algorithm involved uses a minimax rational approximation on a reduction of the argument. All positive inputs will return a valid result.

POWR$M

The 'powr$m' function raises a double precision real value to a double precision real power. The function return is also double precision; there is no single precision equivalent. The algorithm is taken from {7}.

The function is coded so as to adhere to ANSI Fortran standards which do not allow raising negative values to a floating point power, and which do not allow zero to be raised to a zero or negative power. Other inputs may trigger an error if the result of the calculation would result in overflow.

The function implements the following equivalent operation in Fortran:

```
DOUBLE PRECISION A, B, C
A = B ** C
```

as

```
DOUBLE PRECISION A, B, C
DOUBLE PRECISION POWR$M
EXTERNAL POWR$M
A = POWR$M (B, C)
```

There are four cases where this function may signal SWT_MATH_ERROR$. If an attempt is made to raise a negative value to a non-zero power, then the default return value will be the absolute value of that quantity raised to the given power. If an attempt is made to raise zero to a zero or negative power, the default return is zero. If the result would overflow then the default return value is the largest double precision quantity that can be represented. If the result would cause underflow, the default return is the smallest positive value which can be represented on the machine.

SEED$M and RAND$M

The 'seed$m' procedure is used to reset the pseudo-random number generator to a known state. It is called with any 4 byte value which is not equal to 32 bits of zero. The seed can therefore be 4 characters, a long pointer, a long integer, or a real number. If the input is identical to zero then the SWT_MATH_ERROR$ condition is signalled. 'Seed$m' does not return a value.

The 'rand$m' function returns a double precision floating value in the open interval (0.0, 1.0). The argument to the function is set to a 32 bit integer in the range (0, 2**31 - 1). The generator is a linear congruential generator derived from information presented in {8}. The values returned seem to be very well distributed, both from the standpoint of spectral tests and lattice tests.

The 'rand$m' routine does not detect or signal any errors. The first time the 'rand$m' function is called, if the generator has not been initialized with the 'seed$m' procedure, a seed is derived based on the current time of day and cpu utilization.

SIN$M and DSIN$M

These two functions return the sine of the angle whose measure (in radians) is given by the argument. The 'dsin$m' routine expects a double precision argument, and the 'sin$m' routine expects a single precision argument. If the absolute value of the angle is greater than 26353588.0 then the condition SWT_MATH_ERROR$ is signalled. If an error is signalled, the default return value will be zero.

The functions are implemented as minimax polynomial approximations. Note that for angles sufficiently small the value of the sine function is equal to the measure of the angle.

SINH$M and DSNH$M

These two routines calculate the hyperbolic sine of their arguments, defined as sinh(x) = [exp(x) - exp(-x)]/2. The function 'dsnh$m' expects a double precision value as argument, and the 'sinh$m' function expects a single precision argument. The condition SWT_MATH_ERROR$ is signalled if the absolute value of the argument is greater than 22623.630826296. If an error is signalled, the default return value will be zero.

SQRT$M and DSQT$M

These two functions calculate the square root of a floating point value. The 'sqrt$m' function calculates the root of a single precision value, and the 'dsqt$m' routine works for double precision arguments. Attempts to take the square root of negative values will result in an error (signal to SWT_MATH_ERROR$). The default return in this case will be the square root of the absolute value of the argument. All other arguments are in range and return valid results.

The algorithm involved is based on Newton's approximation method with an initial multiplicative approximation. The argument is scaled to within the range [0.5, 2.0) and then the algorithm is iterated to a solution.

TAN$M and DTAN$M

These two functions calculate the tangent of the angle whose measure is given (in radians) as the argument to the functions. The 'dtan$m' function expects a double precision argument, and the 'tan$m' routine expects a single precision argument. The arguments must have an absolute value of less than 13176794.0 or else the condition SWT_MATH_ERROR$ will be signalled. If an error is signalled, the default return value will be zero.

The functions are calculated based on a minimax polynomial approximation over a reduced argument.

TANH$M and DTNH$M

These two routines calculate the hyperbolic tangent of their arguments, defined as tanh(x) = 2/[exp(2x) + 1]. The function 'dtnh$m' expects a double precision value as argument, and the 'tanh$m' function expects a single precision argument. The functions never signal an error and return valid results for all inputs.

Testing

In General

It is important to test the standard mathematical functions which may be used in critical calculations. Not only will the tests measure the accuracy of the routines involved for use in later error estimations, but the testing helps provide information about the allowed domain and range of the functions. Many computer systems have quirks that require special case code for values near the extremes of precision {11}.

The Source of the Tests

The tests were taken from {7}. The tests were altered somewhat to help automate a test suite and also to provide a slightly more consistent form of output for comparison purposes. All of the tests use a set of common routines for non-test calculations and invocation. Where appropriate, the tests have been coded in both Fortran 66 (FTN) and Fortran 77 (F77) so as to test 3 libraries: the SWT Math library, the standard FTN library used by Fortran 66 and Ratfor programs, and the new standard library used in Fortran 77, Pascal, and PL/I programs.

The source code for the tests and support routines is located in the directory along with the source to the SWT Math library. There is a separate set of tests for single precision and double precision. These have been provided in case you wish to verify your own software, or re-run the tests on your own machine. Instructions on how to build and run a test are given in Appendix IV.

The Test Results

There are a number of error measures that could be used to describe these library routines. (For an involved discussion of some of the issues involved, see {7}) The tests which will be described below were taken from {7} and involve a number of checks and comparisons. Each test involves some random accuracy checks in various argument domains. These checks are made against known identities or calculations; for instance, the square root function is checked by comparing a random X against the square root of X*X.

Each accuracy test was performed for 5000 random arguments in each domain. The results of each test are given below, listed as the number of exact matches against the expected value, the number of times less than, and the number of times greater than. Also given are the MRE (maximum relative error) and the point at which that error occurred, and the RMS (root mean square) error

over all the tests in that domain. For those unfamiliar with these measures, the MRE can be thought of as a "worst case" error, and the RMS can be viewed as an "average case" measure of error.

The tests are given single precision first, then double precision. The tests with an asterisk ("*") after the CPU model are double precision test results.

Most of the routines have also been tested with special arguments at the limits of the argument domain or machine precision to help validate the entire range of possible input values. You will note that a few of the Prime standard library routines fail or return incorrect values at the extreme points of the domain. Other special tests are performed and described with each routine, as appropriate. The results given for some of these tests are worst-case results and not average-case; the average case performance was often much better with special arguments.

Finally, each routine was tested with values that would trigger an error (if appropriate). Again, some of the Prime library routines performed badly -- some of them returned incorrect values and never triggered an error.

A Special Note on 550 Results

Each test was run on a 550 model cpu at Georgia Tech and on a 750 model cpu at the Atlanta office of Prime Computer, Inc. The results for the 550 are intended for comparison purposes and should not be taken as a strict measure of accuracy. This is due to the problem with truncation of bits in double precision multiplies discussed in the last chapter. The vast differences in accuracy results between the 550 and 750 may be a measure of improvement in the library routines due to increased accuracy, or they may be an artifact caused by a change in the values calculated by the test programs themselves. The figures given should still allow some comparison between the Prime libraries and the SWT Math library, however.

Other Points of Interest

All of the tests invoke a special subroutine named 'machar' which determines machine characteristics to be used in the tests. The double precision version of this routine cannot be run unmodified on Prime machines due to their odd exponent structure. The double precision routine was modified by the author to return the results as defined by {7}. To recap the few most important points: a single precision value has 23 bits of mantissa and 8 bits of exponent, and rounds results. A double precision value has 47 bits of mantissa and 16 bits of exponent, and multiplication truncates results.

Since single precision arithmetic can include extra bits of accuracy if intermediate results are kept in the extended register, the test routines have been modified in places to force storage (and thus, truncation) of intermediate results. All of the single precision tests were compiled with the -FRN option set on. All of the tests were compiled with minimal optimizations enabled and full debugging. The debug option defeats register tracking optimizations and forces numerous stores. As an aside, this is often why erroneous numerical results disappear when a module is compiled with the debug option -- often to the amazement and indignation of the user.

The random number generator was not extensively tested since it was coded based on published, previous tests {8}. It should be noted, however, that a number of distribution and spectral tests were done locally to ensure that the implementation was not suspect. For comparison purposes, it should be noted that the multiplier used in the Prime APPLIB random number generator (16807) has been shown to be poor in performance on both spectral and lattice tests {8}. The Fortran intrinsic random number generators ('rnd' and 'irnd') behave very poorly in simple spectral tests. They are implemented as 16 bit generators rather than as 32 bit generators.

Use of These Results

It should be noted that these results are general in nature and should not be taken as a complete measure of accuracy on Prime computers. The author has not had extensive training in numerical analysis. A few of the tests did not appear to work correctly, and I found what I believe to be at least one genuine bug in the logic of one of the published test programs. The unusual and inconsistent register structure also leads to problems in running the tests.

It should also be noted that the Primos 18.4 version of the libraries was used in these tests. Future releases of these libraries may demonstrate better performance.

These tests are to be used for general comparison purposes of the Software Tools Math Library routines and the standard Prime libraries. There appear to be a number of accuracy problems in the Prime library routines and floating point firmware and hopefully some of these problems have been indicated in the following tests. Any user wishing to use the Primes or the SWT Math library for any critical applications should make their own tests before placing any great confidence in the results.

Inverse Sine and Cosine

There are no inverse sine or cosine functions in the Fortran 66 library, so these tests are for the other two libraries only.

Test 1
ASIN(X) vs. Taylor Series in (-0.125, 0.125)

		5000	Comparisons		Maximum Rel. Error		Root Mean Sq.
CPU	Library	gt	eq	lt	Bitloss	At	Bitloss
550	F77	1031	3227	742	1.50 of 23	0.0110	0.00 of 23
550	SWT	1	4998	1	0.63 of 23	-0.0808	0.00 of 23
750	F77	1041	3234	725	1.50 of 23	0.0110	0.00 of 23
750	SWT	1	4998	1	0.63 of 23	-0.0808	0.00 of 23
550*	F77	3	66	4931	3.55 of 47	0.802E-2	2.15 of 47
550*	SWT	0	2348	2652	2.00 of 47	-0.1247	0.05 of 47
750*	F77	309	1563	3128	2.58 of 47	-0.0157	0.72 of 47
750*	SWT	0	2347	2653	2.00 of 47	-0.1247	0.05 of 47

Test 2
ACOS(X) vs. Taylor Series in (-0.125, 0.125)

		5000	Comparisons		Maximum Rel. Error		Root Mean Sq.
CPU	Library	gt	eq	lt	Bitloss	At	Bitloss
550	F77	0	3320	1680	0.47 of 23	0.1249	0.00 of 23
550	SWT	0	4904	96	0.47 of 23	0.1249	0.00 of 23
750	F77	0	3319	1681	0.47 of 23	0.1249	0.00 of 23
750	SWT	0	4904	96	0.47 of 23	0.1249	0.00 of 23
550*	F77	0	816	4184	1.29 of 47	-0.0606	0.23 of 47
550*	SWT	0	1796	3204	0.47 of 47	0.1250	0.01 of 47
750*	F77	0	681	4319	1.27 of 47	-0.0874	0.25 of 47
750*	SWT	0	1795	3205	0.47 of 47	0.1250	0.01 of 47

Test 3
ASIN(X) vs. Taylor Series in (0.75, 1.00)

CPU	Library	5000 Comparisons gt	eq	lt	Maximum Rel. Error Bitloss	At	Root Mean Sq. Bitloss
550	F77	76	1313	3611	2.00 of 23	0.84175	0.58 of 23
550	SWT	237	4123	640	1.00 of 23	0.8416	0.00 of 23
750	F77	76	1318	3606	2.00 of 23	0.84175	0.58 of 23
750	SWT	237	4123	640	1.00 of 23	0.8416	0.00 of 23
550*	F77	0	6	4994	6.95 of 47	1.0000	2.36 of 47
550*	SWT	0	446	4554	1.24 of 47	0.7502	0.70 of 47
750*	F77	125	1413	3462	4.88 of 47	1.0000	0.86 of 47
750*	SWT	0	595	4405	1.24 of 47	0.7500	0.66 of 47

Test 4
ACOS(X) vs. Taylor Series in (0.75, 1.00)

CPU	Library	5000 Comparisons gt	eq	lt	Maximum Rel. Error Bitloss	At	Root Mean Sq. Bitloss
550	F77	3210	1261	529	2.95 of 23	0.9746	1.20 of 23
550	SWT	593	3785	622	1.00 of 23	0.8773	0.00 of 23
750	F77	3193	1270	537	2.92 of 23	0.9805	1.19 of 23
750	SWT	593	3785	622	1.00 of 23	0.8773	0.00 of 23
550*	F77	4955	41	4	14.43 of 47	1.0000	8.50 of 47
550*	SWT	2656	2344	0	2.00 of 47	0.8773	0.15 of 47
750*	F77	2560	1267	1173	12.47 of 47	1.0000	6.52 of 47
750*	SWT	2377	2623	0	1.99 of 47	0.8762	0.07 of 47

Test 5
ACOS(X) vs. Taylor Series in (-1.0,-0.75)

CPU	Library	5000 Comparisons gt	eq	lt	Maximum Rel. Error Bitloss	At	Root Mean Sq. Bitloss
550	F77	0	2287	2713	0.73 of 23	-0.7504	0.15 of 23
550	SWT	0	4571	429	0.73 of 23	-0.7504	0.00 of 23
750	F77	0	2286	2714	0.73 of 23	-0.7504	0.15 of 23
750	SWT	0	4572	428	0.73 of 23	-0.7504	0.00 of 23
550*	F77	0	12	4988	5.35 of 47	-1.0000	1.46 of 47
550*	SWT	0	547	4453	0.73 of 47	-0.7500	0.51 of 47
750*	F77	15	930	4055	2.68 of 47	-1.0000	0.56 of 47
750*	SWT	0	608	4392	0.73 of 47	-0.7500	0.50 of 47

Examining the test results shows that the standard Prime library routines are not as accurate as one might wish, especially in test 4. According to {7}, the MRE error should not exceed 1.5 on any of the tests, and the RMS error should be no more than 0.75 in all tests. With the exception of the MRE in the double precision test 2, the SWT Math library performs within these limits; even the error in test 2 is acceptable when the RMS error for the same test is noted.

The tests of special arguments and error returns showed no problems or unexpected results.

Inverse Tangent

Test 1
ATAN(X) vs. Taylor Series in (-0.0625, 0.0625)

CPU	Library	5000 Comparisons gt	eq	lt	Maximum Rel. Error Bitloss	At	Root Mean Sq. Bitloss
550	FTN	527	4211	262	1.00 of 23	-0.0039	0.00 of 23
550	F77	527	4211	262	1.00 of 23	-0.0039	0.00 of 23
550	SWT	0	4999	1	0.32 of 23	0.0500	0.00 of 23
750	FTN	529	4213	258	1.00 of 23	-0.0039	0.00 of 23
750	F77	529	4213	258	1.00 of 23	-0.0039	0.00 of 23
750	SWT	0	4999	1	0.32 of 23	0.0500	0.00 of 23
550*	FTN	0	0	5000	3.32 of 47	0.0314	2.12 of 47
550*	F77	0	47	4953	3.20 of 47	-0.0043	1.95 of 47
550*	SWT	0	2508	2492	1.59 of 47	0.0313	0.00 of 47
750*	FTN	0	3	4997	2.00 of 47	-0.0156	1.11 of 47
750*	F77	0	697	4303	2.00 of 47	-0.0156	0.80 of 47
750*	SWT	0	2530	2470	1.59 of 47	0.0313	0.00 of 47

Test 2
ATAN(X) vs. ATAN(1/16)+ATAN((X-1/16)/(1+X/16)) in (0.0625, 0.2679)

CPU	Library	5000 Comparisons gt	eq	lt	Maximum Rel. Error Bitloss	At	Root Mean Sq. Bitloss
550	FTN	538	2636	1826	2.34 of 23	0.2007	0.21 of 23
550	F77	664	2482	1854	2.34 of 23	0.2007	0.35 of 23
550	SWT	425	3530	1045	1.40 of 23	0.1917	0.00 of 23
750	FTN	543	2626	1831	2.34 of 23	0.2007	0.21 of 23
750	F77	665	2475	1860	2.34 of 23	0.2007	0.35 of 23
750	SWT	423	3530	1047	1.40 of 23	0.1917	0.00 of 23
550*	FTN	372	1454	3174	2.99 of 47	0.0631	1.27 of 47
550*	F77	1774	723	2503	3.28 of 47	0.2081	1.68 of 47
550*	SWT	947	3933	120	1.02 of 47	0.2523	0.00 of 47
750*	FTN	63	2245	2692	1.70 of 47	0.0773	0.15 of 47
750*	F77	1773	1656	1571	2.64 of 47	0.2033	0.94 of 47
750*	SWT	192	4021	787	1.03 of 47	0.2503	0.00 of 47

Test 3
ATAN(X)*2 vs. ATAN(2X/(1-X*X)) in (0.2679, 0.4142)

		5000 Comparisons			Maximum Rel. Error		Root Mean Sq.
CPU	Library	gt	eq	lt	Bitloss	At	Bitloss
550	FTN	1868	2729	403	1.91 of 23	0.2717	0.05 of 23
550	F77	1531	2931	538	1.91 of 23	0.2717	0.02 of 23
550	SWT	882	3678	440	0.93 of 23	0.2680	0.00 of 23
750	FTN	1862	2734	404	1.91 of 23	0.2717	0.05 of 23
750	F77	1526	2933	541	1.91 of 23	0.2717	0.01 of 23
750	SWT	878	3679	443	0.93 of 23	0.2680	0.00 of 23
550*	FTN	158	175	4667	4.81 of 47	0.2731	3.40 of 47
550*	F77	1597	1506	1897	2.93 of 47	0.2693	1.05 of 47
550*	SWT	142	567	4291	3.30 of 47	0.3155	1.83 of 47
750*	FTN	119	137	4744	4.77 of 47	0.2817	3.43 of 47
750*	F77	3576	1015	409	2.76 of 47	0.3050	1.23 of 47
750*	SWT	146	1017	3837	2.80 of 47	0.2952	1.22 of 47

Test 4
ATAN(X)*2 vs. ATAN(2X/(1-X*X)) in (0.4142, 1.0)

		5000 Comparisons			Maximum Rel. Error		Root Mean Sq.
CPU	Library	gt	eq	lt	Bitloss	At	Bitloss
550	FTN	1943	3010	47	2.00 of 23	0.5483	0.07 of 23
550	F77	1970	2986	44	2.00 of 23	0.5479	0.12 of 23
550	SWT	453	4386	161	1.00 of 23	0.5465	0.00 of 23
750	FTN	1941	3012	47	2.00 of 23	0.5483	0.08 of 23
750	F77	1968	2988	44	2.00 of 23	0.5479	0.13 of 23
750	SWT	452	4387	161	1.00 of 23	0.5465	0.00 of 23
550*	FTN	188	576	4236	4.12 of 47	0.4254	2.35 of 47
550*	F77	939	1521	2540	2.76 of 47	0.6689	0.99 of 47
550*	SWT	20	906	4074	3.34 of 47	0.4166	1.94 of 47
750*	FTN	2	48	4950	4.32 of 47	0.4246	2.78 of 47
750*	F77	1913	1042	2045	2.35 of 47	0.6693	0.93 of 47
750*	SWT	872	1859	2269	2.35 of 47	0.4145	0.64 of 47

Examining the test results leads to some interesting conclusions. The SWT Math Library routines are definitely better than either Prime library version, especially in test 2. The margin of error suggested in {7} is met only by the SWT routines.

In the testing of special arguments, the Prime FTN library ATAN had problems with the identities ATAN(-x) = - ATAN(x) and ATAN(x) = x for small x. Errors in both cases were about 10E-7 of the magnitude of x in both single and double precision.

Exponential

In the following tests, the double precision tests did not run to completion when testing the FTN library due to problems in the EXP function. Due to incorrect coding of the function, a floating to fixed conversion raised a SIZE error when taking the exponential of a value which was theoretically in range. Thus, only the results for the first test are available for the FTN exponential function in double precision.

Test 1
EXP(X-0.0625) vs. EXP(X)/EXP(0.0625) in (-0.2841, 0.3466)

		5000 Comparisons			Maximum Rel. Error		Root Mean Sq.
CPU	Library	gt	eq	lt	Bitloss	At	Bitloss
550	FTN	624	3537	839	1.09 of 23	0.9069E-3	0.00 of 23
550	F77	1217	2173	1610	2.41 of 23	0.1815	0.39 of 23
550	SWT	553	3704	743	1.00 of 23	0.0629	0.00 of 23
750	FTN	636	3525	839	1.09 of 23	0.9069E-3	0.00 of 23
750	F77	1218	2172	1610	2.41 of 23	0.1815	0.39 of 23
750	SWT	553	3704	743	1.00 of 23	0.0629	0.00 of 23
550*	FTN	1555	906	2539	4.53 of 47	0.0150	2.51 of 47
550*	F77	1619	711	2670	3.74 of 47	0.2457	1.96 of 47
550*	SWT	325	1759	2916	2.48 of 47	-0.2730	0.72 of 47
750*	FTN	479	1762	2759	4.17 of 47	0.6161E-2	2.23 of 47
750*	F77	1007	1597	2396	2.37 of 47	0.0293	0.78 of 47
750*	SWT	227	2056	2717	2.08 of 47	-0.2777	0.42 of 47

Test 2
EXP(X-2.8125) vs. EXP(X)/EXP(2.8125) in (-0.2277E+5, -0.3466E+1)

		5000 Comparisons			Maximum Rel. Error		Root Mean Sq.
CPU	Library	gt	eq	lt	Bitloss	At	Bitloss
550	FTN	2838	23	2139	6.42 of 23	-0.4405E+2	4.94 of 23
550	F77	1201	2287	1512	2.01 of 23	-0.6125E+2	0.32 of 23
550	SWT	499	3745	756	1.02 of 23	-0.1799E+2	0.00 of 23
750	FTN	2838	23	2139	6.42 of 23	-0.4405E+2	4.94 of 23
750	F77	1201	2285	1514	2.01 of 23	-0.6125E+2	0.32 of 23
750	SWT	499	3745	756	1.02 of 23	-0.1799E+2	0.00 of 23
550*	F77	2638	426	1936	47.00 of 47	-0.2268E+5	43.85 of 47
550*	SWT	1034	205	3761	13.95 of 47	-0.2264E+5	11.75 of 47
750*	F77	2036	1426	1538	47.00 of 47	-0.2089E+2	43.85 of 47
750*	SWT	441	424	4135	13.95 of 47	-0.2264E+5	11.75 of 47

Test 3
EXP(X-2.8125) vs. EXP(X)/EXP(2.8125) in (6.931, 87.92)

		5000	Comparisons		Maximum Rel. Error		Root Mean Sq.
CPU	Library	gt	eq	lt	Bitloss	At	Bitloss
550	FTN	2993	15	1992	5.69 of 23	0.8669E+2	5.05 of 23
550	F77	1204	2311	1485	1.93 of 23	0.5069E+2	0.30 of 23
550	SWT	489	3704	807	1.00 of 23	0.4371E+2	0.00 of 23
750	FTN	2993	15	1992	5.69 of 23	0.8669E+2	5.05 of 23
750	F77	1204	2311	1485	1.93 of 23	0.5069E+2	0.30 of 23
750	SWT	489	3704	807	1.00 of 23	0.4371E+2	0.00 of 23
550*	F77	2676	444	1880	6.12 of 47	0.1571E+5	4.28 of 47
550*	SWT	3082	899	1019	4.28 of 47	0.1592E+5	2.07 of 47
750*	F77	2078	1400	1522	5.08 of 47	0.1584E+5	2.65 of 47
750*	SWT	1065	2205	1730	3.47 of 47	0.2018E+5	1.29 of 47

The results of test 2 are a bit surprising. After careful checking of the code and the test, it seems likely that there is a problem in the test since the routines from both libraries appear so bad. The MRE values appear to be close to the limit of what the routines can compute without underflow. Performing a check on the MRE error in each case reveals that there is no measurable error in the exponential function at this point in regard to the logarithm function. That is, the values of the exponential functions at the MRE point, when used as arguments to the SWT logarithm function (which is known to be fairly accurate; see below), produce the exact same value as the MRE point. This leads to the conclusion that the testing procedure is somehow faulty due to the unusual register structure of the Primes. It can be concluded that (in this domain) the functions are probably correct, but the measure of error cannot be determined by this test.

The results of the other tests indicate major differences in accuracy amongst the routines. The SWT routine seems much better in most cases.

The tests of special arguments revealed a number of interesting items. For instance, the single precision F77 EXP routine does not signal an error when given arguments very much out of range. Instead, it returns either zero (in the case of underflow) or a very large value (in the case of overflow). Also, all of the functions have some amount of error in the identity EXP(X)*EXP(-X) = 1.0, with single precision values of X near 1.0 producing errors of approximately 10E-6, and double precision values near 1.0 producing errors of near 10E-12.

Logarithms

Test 1
ALOG(X) vs. Taylor Series of ALOG(1+Y) in (1-.7813E-2, 1+.7813E-2)

CPU	Library	5000 Comparisons gt	eq	lt	Maximum Rel. Error Bitloss	At	Root Mean Sq. Bitloss
550	FTN	2246	234	2520	2.57 of 23	0.9961	1.35 of 23
550	F77	1392	2607	1001	1.89 of 23	1.0021	0.07 of 23
550	SWT	1	4996	3	0.59 of 23	0.9948	0.00 of 23
750	FTN	2251	229	2520	2.57 of 23	0.9961	1.36 of 23
750	F77	1389	2603	1008	1.89 of 23	1.0021	0.07 of 23
750	SWT	1	4996	3	0.59 of 23	0.9948	0.00 of 23
550*	FTN	2449	315	2236	25.55 of 47	1.000	19.52 of 47
550*	F77	2038	996	1966	2.98 of 47	1.000	1.42 of 47
550*	SWT	1013	2493	1494	2.13 of 47	1.0000	0.19 of 47
750*	FTN	1314	1603	2083	25.55 of 47	1.000	19.52 of 47
750*	F77	1206	2507	1287	1.94 of 47	1.000	0.04 of 47
750*	SWT	1206	2507	1287	1.94 of 47	1.000	0.04 of 47

Test 2
ALOG(X) vs. ALOG(17X/16)-ALOG(17/16) in (0.7071, 0.9375)

CPU	Library	5000 Comparisons gt	eq	lt	Maximum Rel. Error Bitloss	At	Root Mean Sq. Bitloss
550	FTN	0	1930	3070	2.01 of 23	0.8300	0.41 of 23
550	F77	0	2753	2247	1.97 of 23	0.8253	0.07 of 23
550	SWT	0	3628	1372	1.00 of 23	0.7788	0.00 of 23
750	FTN	0	1936	3064	2.01 of 23	0.8300	0.41 of 23
750	F77	0	2760	2240	1.97 of 23	0.8253	0.07 of 23
750	SWT	0	3628	1372	1.00 of 23	0.7788	0.00 of 23
550*	FTN	0	54	4946	4.24 of 47	0.9299	2.49 of 47
550*	F77	0	132	4868	3.00 of 47	0.7323	1.28 of 47
550*	SWT	0	2053	2947	2.28 of 47	0.7347	0.51 of 47
750*	FTN	0	1022	3978	4.26 of 47	0.9367	2.47 of 47
750*	F77	0	2067	2933	1.99 of 47	0.7779	0.46 of 47
750*	SWT	0	2067	2933	1.99 of 47	0.7779	0.46 of 47

Test 3
ALOG10(X) vs. ALOG10(11X/10)-ALOG(11/10) in (0.3162, 0.900)

CPU	Library	5000 Comparisons gt	eq	lt	Maximum Rel. Error Bitloss	At	Root Mean Sq. Bitloss
550	FTN	0	1870	3130	2.58 of 23	0.8659	0.90 of 23
550	F77	0	1986	3014	2.72 of 23	0.7045	0.74 of 23
550	SWT	0	2462	2538	2.06 of 23	0.8708	0.35 of 23
750	FTN	0	1867	3133	2.58 of 23	0.8659	0.90 of 23
750	F77	0	1983	3017	2.72 of 23	0.7045	0.75 of 23
750	SWT	0	2462	2538	2.06 of 23	0.8708	0.35 of 23
550*	FTN	0	924	4076	4.44 of 47	0.8936	2.18 of 47
550*	F77	0	1029	3971	3.58 of 47	0.8974	1.66 of 47
550*	SWT	0	1244	3756	3.73 of 47	0.8974	1.71 of 47
750*	FTN	0	1383	3617	4.40 of 47	0.8963	2.27 of 47
750*	F77	0	1684	3316	3.37 of 47	0.8946	1.34 of 47
750*	SWT	0	1499	3501	3.37 of 47	0.8943	1.29 of 47

Test 4
ALOG(X*X) vs. 2*ALOG(X) in (16.00, 240.0)

CPU	Library	5000 Comparisons gt	eq	lt	Maximum Rel. Error Bitloss	At	Root Mean Sq. Bitloss
550	FTN	2490	2510	0	1.00 of 23	0.5473E+2	0.15 of 23
550	F77	2499	2501	0	1.00 of 23	0.5473E+2	0.15 of 23
550	SWT	127	4873	0	0.99 of 23	0.5575E+2	0.00 of 23
750	FTN	2499	2501	0	1.00 of 23	0.5473E+2	0.15 of 23
750	F77	2491	2509	0	1.00 of 23	0.5473E+2	0.15 of 23
750	SWT	127	4873	0	0.99 of 23	0.5575E+2	0.00 of 23
550*	FTN	2911	2089	0	2.36 of 47	0.2263E+2	0.98 of 47
550*	F77	1195	3805	0	3.00 of 47	0.5491E+2	1.58 of 47
550*	SWT	1437	3563	0	1.53 of 47	0.1604E+2	0.00 of 47
750*	FTN	1548	3452	0	1.44 of 47	0.1909E+2	0.00 of 47
750*	F77	1537	3463	0	1.44 of 47	0.1909E+2	0.00 of 47
750*	SWT	333	4667	0	1.06 of 47	0.4591E+2	0.00 of 47

These tests indicate that both the SWT Math library and the F77 library implementations of the logarithm functions are within acceptable error bounds (as defined in {7}), with the SWT version being somewhat better. The Fortran 66 version obviously has some points at which it behaves very poorly (see test 1). The similarity between the results for the SWT and F77 versions as shown in tests 1 and 2 can probably be explained by the fact that the same algorithm was used in each.

The SWT MRE errors in the double precision part of tests 1 and 2 are a bit large, but the corresponding error in the RMS indicates that the error is not systematic in nature. The error is of no major significance, although it could possibly be less.

The SWT routine performed very well in tests of the identity ALOG(X) = -ALOG(1/X), returning exactly the same values in every

test. The FTN and F77 routines returned occasional matches, but were often in error by amounts close to 10E-6 (single precision) and 10E-12 (double precision).

What is most interesting is to note that both the F77 and FTN double precision routines are seriously flawed for very small arguments. Due to a rather obvious coding error, any double precision value whose exponent is less than -32640 will have its logarithm calculated as a large positive number -- just as if the sign of the exponent was reversed!! It would appear as if these routines were never tested at any values near the limits of their domains. The SWT routine does not suffer from this problem.

The POWR$M Function

The SWT 'powr$m' function was tested against the intrinsic "**" operation in these tests. That is, when testing the FTN and F77 libraries, the operation "x ** y" was used and the compilers were allowed to generate the calls to the appropriate library routines.

Although there is no single precision version of the SWT 'powr$m' function, it was tested within the range for single precision values and compared against the Prime power operation. Due to recurring problems in the division of very small values, and the multiplication of very large values caused by the faults in the hardware, tests 1 and 2 were the only double precision tests run to completion.

Test 1
X ** 1.0 vs. X in (0.50, 1.00)

		5000 Comparisons			Maximum Rel. Error		Root Mean Sq.
CPU	Library	gt	eq	lt	Bitloss	At	Bitloss
550	FTN	2399	2583	18	0.99 of 23	0.5022	0.00 of 23
550	F77	2886	1935	179	1.50 of 23	0.7072	0.37 of 23
550	SWT	0	5000	0	0.00 of 23	-----	0.00 of 23
750	FTN	2394	2586	20	0.99 of 23	0.5022	0.00 of 23
750	F77	2888	1932	180	1.50 of 23	0.7072	0.37 of 23
750	SWT	0	5000	0	0.00 of 23	-----	0.00 of 23
550*	FTN	5000	0	0	4.76 of 47	0.8469	4.12 of 47
550*	F77	4996	4	0	4.19 of 47	0.6025	3.06 of 47
550*	SWT	4997	3	0	1.94 of 47	0.5222	0.69 of 47
750*	FTN	4920	80	0	4.28 of 47	0.7735	3.66 of 47
750*	F77	4437	563	0	2.62 of 47	0.6511	1.38 of 47
750*	SWT	4837	163	0	1.06 of 47	0.9578	0.50 of 47

Test 2
(X*X)**1.5 vs. (X*X)*X in (0.50, 1.00)

		5000 Comparisons			Maximum Rel. Error		Root Mean Sq.
CPU	Library	gt	eq	lt	Bitloss	At	Bitloss
550	FTN	3330	1394	276	2.90 of 23	0.5118	0.98 of 23
550	F77	2047	2082	871	1.87 of 23	0.5147	0.35 of 23
550	SWT	332	4359	319	1.00 of 23	0.6300	0.00 of 23
750	FTN	3305	1410	285	2.94 of 23	0.5068	0.98 of 23
750	F77	2086	2062	852	1.98 of 23	0.9135	0.37 of 23
750	SWT	317	4349	334	0.99 of 23	0.7954	0.00 of 23
550*	FTN	4939	55	6	5.12 of 47	0.5712	4.03 of 47
550*	F77	4869	131	0	4.94 of 47	0.5466	3.30 of 47
550*	SWT	1172	2051	1777	2.57 of 47	0.5012	0.66 of 47
750*	FTN	4172	649	179	4.23 of 47	0.7358	3.47 of 47
750*	F77	3782	1010	208	2.62 of 47	0.6875	1.20 of 47
750*	SWT	2432	2566	2	1.06 of 47	0.7833	0.07 of 47

Test 3
   (X*X)**1.5 vs. (X*X)*X in (1.00, 0.5541E+13)

CPU	Library	5000 Comparisons gt	eq	lt	Maximum Rel. Error Bitloss	At	Root Mean Sq. Bitloss
550	FTN	5000	0	0	8.51 of 23	0.1129E+13	7.80 of 23
550	F77	4950	0	50	17.93 of 23	0.5487E+13	13.83 of 23
550	SWT	315	4362	323	1.00 of 23	0.6928E+12	0.00 of 23
750	FTN	5000	0	0	8.53 of 23	0.2676E+12	7.80 of 23
750	F77	4950	0	50	17.93 of 23	0.5487E+13	13.83 of 23
750	SWT	337	4313	350	0.99 of 23	0.4407E+13	0.00 of 23

In test 4, the point given at which the MRE was recorded is the value of X. The Y value is available on request.

Test 4
   X**Y vs. (X*X)**(Y/2), X in (0.01, 10.0), Y in (-19.42, 19.42)

CPU	Library	5000 Comparisons gt	eq	lt	Maximum Rel. Error Bitloss	At	Root Mean Sq. Bitloss
550	FTN	1006	3052	942	6.49 of 23	9.8692	4.20 of 23
550	F77	2266	518	2216	5.46 of 23	0.0120	3.43 of 23
550	SWT	1700	1604	1696	3.25 of 23	2.0541	1.28 of 23
750	FTN	958	3104	938	6.49 of 23	7.7463	4.20 of 23
750	F77	2251	514	2235	5.46 of 23	0.0120	3.47 of 23
750	SWT	1644	1591	1765	3.20 of 23	2.8599	1.30 of 23

It seems fairly obvious from the above test results that the Prime library routines are rather sadly lacking in precision. Test 3 alone shows a RME loss of nearly 18 out of 23 bits. Conclusions about tests 3 and 4 can possibly (with a cautionary warning!) be extrapolated to the double precision cases, at least for the SWT routine, since the routine is the same for both precisions. The tests of special arguments indicate that the 'powr$m' routine does behave well in the double precision case. Running small portions of the test to avoid some of the firmware arithmetic problems tends to support these conclusions.

Sine and Cosine

Test 1
SIN(X) vs. 3*SIN(X/3)-4*SIN(X/3)**3 in (0.0, 1.571)

		5000	Comparisons		Maximum Rel. Error		Root Mean Sq.
CPU	Library	gt	eq	lt	Bitloss	At	Bitloss
550	FTN	3047	32	1921	1.41 of 23	0.8183	0.00 of 23
550	F77	466	2204	2330	1.49 of 23	0.7910	0.00 of 23
550	SWT	498	3979	523	1.00 of 23	0.5243	0.00 of 23
750	FTN	3095	0	1905	1.41 of 23	0.8183	0.00 of 23
750	F77	466	2202	2332	1.61 of 23	0.3330	0.00 of 23
750	SWT	498	3979	523	1.00 of 23	0.5243	0.00 of 23
550*	FTN	31	233	4736	4.50 of 47	0.7888	3.17 of 47
550*	F77	3204	1276	520	4.07 of 47	0.3021	2.09 of 47
550*	SWT	2880	1207	913	3.41 of 47	0.1898	1.51 of 47
750*	FTN	130	679	4191	2.81 of 47	0.1495	1.44 of 47
750*	F77	863	1773	2364	2.30 of 47	0.6557	0.46 of 47
750*	SWT	494	2459	2047	2.04 of 47	1.3513	0.21 of 47

Test 2
SIN(X) vs. 3*SIN(X/3)-4*SIN(X/3)**3 in (18.85, 20.42)

		5000	Comparisons		Maximum Rel. Error		Root Mean Sq.
CPU	Library	gt	eq	lt	Bitloss	At	Bitloss
550	FTN	3546	12	1442	18.00 of 23	18.850	11.87 of 23
550	F77	431	2267	2302	1.73 of 23	19.001	0.00 of 23
550	SWT	510	3939	551	1.00 of 23	19.103	0.00 of 23
750	FTN	3560	0	1440	18.00 of 23	18.850	11.87 of 23
750	F77	431	2267	2302	1.73 of 23	19.001	0.00 of 23
750	SWT	511	3938	551	1.00 of 23	19.103	0.00 of 23
550*	FTN	394	118	4488	18.98 of 47	18.850	12.87 of 47
550*	F77	1800	660	2540	19.33 of 47	18.850	13.20 of 47
550*	SWT	1776	491	2733	19.33 of 47	18.850	13.20 of 47
750*	FTN	1852	160	2988	18.98 of 47	18.850	12.84 of 47
750*	F77	891	1803	2306	6.93 of 47	18.850	1.14 of 47
750*	SWT	699	2463	1838	2.02 of 47	20.242	0.14 of 47

Test 3

COS(X) vs. 4*COS(X/3)**3-3*COS(X/3) in (21.99, 23.56)

CPU	Library	5000 Comparisons gt	eq	lt	Maximum Rel. Error Bitloss	At	Root Mean Sq. Bitloss
550	FTN	1911	13	3076	11.83 of 23	23.555	7.14 of 23
550	F77	1850	24	3126	1.36 of 23	23.150	0.00 of 23
550	SWT	2470	33	2497	0.70 of 23	23.529	0.00 of 23
750	FTN	1923	0	3077	11.83 of 23	23.555	7.14 of 23
750	F77	1845	0	3155	1.37 of 23	23.150	0.00 of 23
750	SWT	2471	0	2529	0.70 of 23	23.530	0.00 of 23
550*	FTN	1470	658	2872	17.42 of 47	23.562	11.44 of 47
550*	F77	4978	20	2	17.77 of 47	23.562	11.70 of 47
550*	SWT	4657	291	52	17.77 of 47	23.562	11.70 of 47
750*	FTN	855	564	3581	15.33 of 47	23.561	9.78 of 47
750*	F77	4490	464	46	2.85 of 47	23.353	1.46 of 47
750*	SWT	1334	2614	1052	1.63 of 47	22.237	0.00 of 47

This is another test which illustrates how the multiplication bug in the 550 firmware can affect critical results. Observe the differences in double precision results in test 2 and 3. It is also fairly obvious that the FTN library sine and cosine functions have severe accuracy problems. The SWT library routines perform well within error limits {7} and are much better than the F77 routines.

When testing special arguments it is found that both the F77 and FTN routines have difficulty with the identities SIN(-X)=-SIN(X) and COS(-X)=COS(X). The ratio of the calculated difference to X is about 10E-8 for single precision, and 10E-12 to 10E-28 for double precision (the F77 library is more accurate). The SWT Library routines calculate no differences in these identities.

When special values are tested for error checking, it is discovered that the Prime routines trigger a SIZE error in float to fixed conversion when presented with a large argument rather than checking for (and reporting) the actual problem of an error of excessive magnitude. The SWT routine properly traps the error.

Hyperbolic Sine and Cosine

There are no hyperbolic sine (sinh) or hyperbolic cosine (cosh) routines in the FTN library, so the tests below are only for the F77 and SWT libraries.

Test 1
SINH(X) vs. Taylor Series in (0.00, 0.50)

		5000	Comparisons		Maximum Rel. Error		Root Mean Sq.
CPU	Library	gt	eq	lt	Bitloss	At	Bitloss
550	F77	1	2418	2581	1.38 of 23	0.1908	0.17 of 23
550	SWT	8	4965	27	1.00 of 23	0.4827	0.00 of 23
750	F77	1	2430	2569	1.38 of 23	0.1908	0.17 of 23
750	SWT	7	4966	27	1.00 of 23	0.4827	0.00 of 23
550*	F77	87	754	4159	3.00 of 47	0.0156	1.69 of 47
550*	SWT	360	2597	2043	1.99 of 47	0.4855	0.06 of 47
750*	F77	343	2643	2033	1.99 of 47	0.4833	0.06 of 47
750*	SWT	370	2643	1987	2.00 of 47	0.4822	0.06 of 47

Test 2
COSH(X) vs. Taylor Series in (0.00, 0.50)

		5000	Comparisons		Maximum Rel. Error		Root Mean Sq.
CPU	Library	gt	eq	lt	Bitloss	At	Bitloss
550	F77	0	3547	1453	1.00 of 23	0.0348	0.03 of 23
550	SWT	6	4905	89	0.98 of 23	0.1599	0.00 of 23
750	F77	1	3548	1451	1.00 of 23	0.3937E-2	0.03 of 23
750	SWT	6	4905	89	0.98 of 23	0.1599	0.00 of 23
550*	F77	0	143	4857	4.15 of 47	0.4906	2.74 of 47
550*	SWT	0	1442	3558	3.41 of 47	0.4997	1.28 of 47
750*	F77	0	2098	2902	3.41 of 47	0.4955	1.30 of 47
750*	SWT	0	2809	2191	3.15 of 47	0.4919	1.04 of 47

Test 3
SINH(X) vs. C*(SINH(X+1)+SINH(X-1)) in (3.00, LOG(XMAX))

		5000 Comparisons			Maximum Rel. Error		Root Mean Sq.
CPU	Library	gt	eq	lt	Bitloss	At	Bitloss
550	F77	2051	1879	1070	16.53 of 23	87.017	10.43 of 23
550	SWT	1618	3303	79	1.24 of 23	43.500	0.00 of 23
750	F77	2051	1881	1068	16.53 of 23	87.017	10.43 of 23
750	SWT	1618	3303	79	1.24 of 23	43.500	0.00 of 23
550*	F77	3615	388	997	5.92 of 47	0.2124E+5	4.19 of 47
550*	SWT	4579	262	159	4.45 of 47	0.2067E+5	3.17 of 47
750*	F77	3937	337	726	4.85 of 47	0.1669E+5	2.73 of 47
750*	SWT	4498	303	199	3.03 of 47	0.1304E+5	1.49 of 47

In test 4, the double precision COSH routine in the F77 library generated numerous errors for large values that should have been in range. These errors aborted the test and therefore there are no results for the double precision F77 COSH.

Test 4
COSH(X) vs. C*(COSH(X+1)+COSH(X-1)) in (3.00, LOG(XMAX))

		5000 Comparisons			Maximum Rel. Error		Root Mean Sq.
CPU	Library	gt	eq	lt	Bitloss	At	Bitloss
550	F77	2051	1903	1046	17.75 of 23	87.000	11.74 of 23
550	SWT	1558	3341	101	1.00 of 23	49.214	0.00 of 23
750	F77	2051	1904	1045	17.75 of 23	87.000	11.74 of 23
750	SWT	1558	3341	101	1.00 of 23	49.214	0.00 of 23
550*	SWT	4566	263	171	4.53 of 47	0.1794E+5	3.16 of 47
750*	SWT	4522	284	194	3.06 of 47	0.1281E+5	1.49 of 47

The results of tests 3 and 4 show that the F77 routines are rather inaccurate at the extremes of range. The RME measures for the SWT routines are a bit large, but the corresponding RMS error is small. According to the figures given in {7}, the SWT routines perform within the range of acceptable error.

As with many of the other tests, fundamental identities involving negated arguments were not calculated quite correctly in the Prime routines. Another interesting(?) result occurred when the F77 SINH routine was called with a very large positive value. The SINH routine did not signal an error, but rather returned the maximum floating point value -- an incorrect result.

Square Root

The square root function is one of the easiest to code and the accuracy of such a routine should be very, very good if done correctly. Newton's method converges quickly and requires only a few iterations on a reduced argument to reach a solution. Due to the nature of the square root function and its use, the random arguments are logarithmically distributed over the sample interval; all the other tests use a uniform distribution.

Test 1
SQRT(X*X) vs. X in (0.7071, 1.00)

		5000	Comparisons		Maximum Rel. Error		Root Mean Sq.
CPU	Library	gt	eq	lt	Bitloss	At	Bitloss
550	FTN	0	5000	0	0.00 of 23	-----	0.00 of 23
550	F77	1	4999	0	0.42 of 23	0.7500	0.00 of 23
550	SWT	0	5000	0	0.00 of 23	-----	0.00 of 23
750	FTN	0	5000	0	0.00 of 23	-----	0.00 of 23
750	F77	1	4999	0	0.42 of 23	0.7500	0.00 of 23
750	SWT	0	5000	0	0.00 of 23	-----	0.00 of 23
550*	FTN	0	0	5000	2.50 of 47	0.7095	1.33 of 47
550*	F77	0	1	4999	2.08 of 47	0.7074	1.13 of 47
550*	SWT	0	0	5000	2.49 of 47	0.7114	1.31 of 47
750*	FTN	0	2403	2597	0.50 of 47	0.7072	0.00 of 47
750*	F77	0	4481	519	0.50 of 47	0.7072	0.00 of 47
750*	SWT	0	2493	2507	0.50 of 47	0.7072	0.00 of 47

Test 2
SQRT(X*X) vs. X in (1.00, 1.414)

		5000	Comparisons		Maximum Rel. Error		Root Mean Sq.
CPU	Library	gt	eq	lt	Bitloss	At	Bitloss
550	FTN	0	5000	0	0.00 of 23	-----	0.00 of 23
550	F77	77	4923	0	1.00 of 23	1.0004	0.00 of 23
550	SWT	0	5000	0	0.00 of 23	-----	0.00 of 23
750	FTN	0	5000	0	0.00 of 23	-----	0.00 of 23
750	F77	80	4920	0	1.00 of 23	1.0004	0.00 of 23
750	SWT	0	5000	0	0.00 of 23	-----	0.00 of 23
550*	FTN	0	0	5000	3.00 of 47	1.0003	2.00 of 47
550*	F77	0	6	4994	3.00 of 47	1.0003	1.97 of 47
550*	SWT	0	0	5000	3.00 of 47	1.0003	2.00 of 47
750*	FTN	0	3384	1616	1.00 of 47	1.0001	0.00 of 47
750*	F77	1	3766	1233	1.00 of 47	1.0001	0.00 of 47
750*	SWT	0	3387	1613	1.00 of 47	1.0001	0.00 of 47

All of the routines perform well in these tests, and all have results within acceptable margins of error. Test 2 readily illustrates how results can change due to the double precision multiply bug on 550 machines. Nothing in these tests would particularly recommend one routine against any other, although the SWT and FTN routines appear to be marginally more accurate

than the F77 version.

Tests of special arguments, however, reveal some difficulties. The FTN and F77 double precision functions generate overflow faults when presented with a large enough argument. There is no valid mathematical reason for this to occur. Additionally, the Prime double precision functions calculated incorrect square roots for selected small values near the limits of storage precision. The SWT library routine behaved correctly for all special arguments.

Tangent and Cotangent

There is no tangent routine in the standard FTN library, so the results of the tests below apply to only the F77 and SWT libraries.

Test 1
TAN(X) vs. 2*TAN(X/2)/(1-TAN(X/2)**2) in (0.00, 0.7854)

		5000	Comparisons		Maximum Rel. Error		Root Mean Sq.
CPU	Library	gt	eq	lt	Bitloss	At	Bitloss
550	F77	1978	2361	661	1.99 of 23	0.2458	0.22 of 23
550	SWT	2054	2518	428	1.97 of 23	0.1273	0.11 of 23
750	F77	1968	2369	663	1.99 of 23	0.2458	0.21 of 23
750	SWT	2044	2525	431	1.79 of 23	0.5237	0.11 of 23
550*	F77	191	1085	3724	3.43 of 47	0.7483	1.93 of 47
550*	SWT	190	996	3814	3.62 of 47	0.7734	2.03 of 47
750*	F77	439	2565	1996	2.79 of 47	0.2815	0.99 of 47
750*	SWT	542	2384	2074	2.87 of 47	0.2678	1.11 of 47

Test 2
TAN(X) vs. 2*TAN(X/2)/(1-TAN(X/2)**2) in (2.749, 3.534)

		5000	Comparisons		Maximum Rel. Error		Root Mean Sq.
CPU	Library	gt	eq	lt	Bitloss	At	Bitloss
550	F77	2318	2018	664	2.21 of 23	2.9813	0.48 of 23
550	SWT	991	3178	831	1.17 of 23	3.0306	0.00 of 23
750	F77	2340	2009	651	2.09 of 23	2.8026	0.49 of 23
750	SWT	987	3176	837	1.17 of 23	3.0306	0.00 of 23
550*	F77	3715	815	470	3.88 of 47	3.1342	2.09 of 47
550*	SWT	3827	868	305	3.87 of 47	3.1116	2.00 of 47
750*	F77	2197	1870	933	2.79 of 47	2.9978	1.07 of 47
750*	SWT	2131	2213	656	2.60 of 47	3.4601	0.83 of 47

Test 3

TAN(X) vs. 2*TAN(X/2)/(1-TAN(X/2)**2) in (18.85, 19.63)

		5000	Comparisons		Maximum Rel. Error		Root Mean Sq.
CPU	Library	gt	eq	lt	Bitloss	At	Bitloss
550	F77	1933	2374	693	1.93 of 23	19.332	0.20 of 23
550	SWT	2074	2467	459	1.94 of 23	19.104	0.14 of 23
750	F77	1934	2374	692	1.96 of 23	19.102	0.20 of 23
750	SWT	2071	2470	459	1.94 of 23	19.104	0.14 of 23
550*	F77	193	1136	3671	3.55 of 47	19.448	1.93 of 47
550*	SWT	178	1076	3746	3.59 of 47	19.541	2.03 of 47
750*	F77	399	2583	2018	2.94 of 47	19.104	0.99 of 47
750*	SWT	499	2403	2098	2.92 of 47	18.981	1.10 of 47

Test 4

COT(X) vs. (COT(X/2)**2-1)/(2*COT(X/2)) in (18.85, 19.63)

		5000	Comparisons		Maximum Rel. Error		Root Mean Sq.
CPU	Library	gt	eq	lt	Bitloss	At	Bitloss
550	F77	2602	16	2382	2.16 of 23	19.377	0.18 of 23
550	SWT	2311	32	2657	1.36 of 23	19.086	0.00 of 23
750	F77	2593	8	2399	2.16 of 23	19.377	0.18 of 23
750	SWT	2307	13	2680	1.35 of 23	19.086	0.00 of 23
550*	F77	261	818	3921	3.91 of 47	18.857	2.20 of 47
550*	SWT	335	772	3893	3.79 of 47	19.455	1.95 of 47
750*	F77	973	1843	2184	3.00 of 47	19.439	1.13 of 47
750*	SWT	989	1794	2217	2.53 of 47	19.616	0.73 of 47

These tests show that both implementations are correct to within a reasonable error bound. Tests on special arguments revealed that the double precision F77 tangent routine signals an error for a large input value that should be well within the range that can be dealt with.

Hyperbolic Tangent

There does not appear to be a double precision hyperbolic tangent routine in the FTN library, although there is a single precision version. The following test results reflect that fact.

Test 1
TANH(X) vs. (TANH(X-1/8)*TANH(1/8))/(1+TANH(X-1/8)*TANH(1/8)) in (0.125, 0.5493)

		5000 Comparisons			Maximum Rel. Error		Root Mean Sq.
CPU	Library	gt	eq	lt	Bitloss	At	Bitloss
550	FTN	1396	1788	1816	2.99 of 23	0.1268	0.72 of 23
550	F77	1860	1253	1887	3.71 of 23	0.1347	1.41 of 23
550	SWT	1203	2833	964	1.77 of 23	0.1479	0.00 of 23
750	FTN	1401	1782	1817	2.99 of 23	0.1268	0.73 of 23
750	F77	1863	1248	1889	3.71 of 23	0.1347	1.41 of 23
750	SWT	1200	2832	968	1.77 of 23	0.1479	0.00 of 23
550*	F77	2731	230	2039	6.64 of 47	0.1315	4.08 of 47
550*	SWT	4624	348	28	3.55 of 47	0.1288	1.98 of 47
750*	F77	2380	605	2015	4.83 of 47	0.1328	2.58 of 47
750*	SWT	3966	957	77	2.99 of 47	0.1270	1.33 of 47

Test 2
TANH(X) vs. (TANH(X-1/8)*TANH(1/8))/(1+TANH(X-1/8)*TANH(1/8)) in (0.6743, 17.33)

		5000 Comparisons			Maximum Rel. Error		Root Mean Sq.
CPU	Library	gt	eq	lt	Bitloss	At	Bitloss
550	FTN	1103	2707	1190	1.69 of 23	0.7217	0.00 of 23
550	F77	1288	2316	1396	1.91 of 23	1.0990	0.00 of 23
550	SWT	1204	2324	1472	1.73 of 23	0.6974	0.00 of 23
750	FTN	1100	2704	1196	1.69 of 23	0.7217	0.00 of 23
750	F77	1281	2324	1395	1.91 of 23	1.0990	0.00 of 23
750	SWT	1198	2328	1474	1.73 of 23	0.6974	0.00 of 23
550*	F77	1846	2234	920	3.34 of 47	0.8543	0.86 of 47
550*	SWT	2676	2258	66	2.11 of 47	1.6430	0.62 of 47
750*	F77	974	3464	562	2.26 of 47	0.7330	0.14 of 47
750*	SWT	1185	3442	373	1.76 of 47	1.3987	0.14 of 47

The above tests show that any of the three routines is acceptable for use in single precision, but the error in the double precision F77 routine in test 1 is rather large. The SWT routine is once again the best.

Tests of special arguments indicate a definite problem in the Prime single precision library routines when calculating various identity operations such as TANH(-X) = -TANH(X). The difference in calculated values is about 10E-6; the SWT routine calculates no differences.

Conclusions

It appears as if the standard libraries under Primos have been implemented without anything other than a cursory check of accuracy. A number of the library routines return incorrect results that are mathematically absurd. Other routines trigger errors on values which should be well within range.

Although the single precision arithmetic is acceptable for most calculations, the double precision floating point arithmetic on Prime 400/550 machines (and possibly on the new 2250, as well) is seriously flawed. Critical calculations should not be performed on any of these machines since the error induced by certain unstable operations can completely ruin the accuracy of the results. Bizarre behavior of programs which work on other machines may also be noted due to some of the odd quirks in the floating point structure. Users should run their own tests to determine if their applications will be affected adversely by these problems.

An increase in accuracy may very well be obtained in some programs by recoding the standard functions. It has been shown that the SWT Math Library significantly outperforms the standard Prime libraries in virtually every instance; it is possible that the encoding of different algorithms might also result in increased precision.

This paper has also presented differences in the architecture of the Prime 400/550 computer and the 750 which violate the claim of strict upward compatibility of software. Programs which directly access the register structure or make specific assumptions about precision should be coded with these differences in mind.

References

{1} Dr. John Spitzer; private communication to Academic Computing Center, State University College at Brockport, NY, and reported to Prime Computer; 1978

{2} Mark P. C. Legg; copies of TAR reports to Prime computer dated 1980 to 1982 from The Flinders University of South Australia; private communication; 1982

{3} Harold Stone, editor; Introduction to Computer Architecture, 2nd. Edition; Science Research Associates, Inc; 1980

{4} Andrew Tanenbaum; Structured Computer Organization; Prentice-Hall; 1976

{5} M. Sporer; Prime PE-T 416, "P400 Instruction Times"; Prime Computer, Inc.; 1978

{6} Mark P. C. Legg; untitled report contained in private communication; Computer Centre of The Flinders University of South Australia, Bedford Park; 1982

{7} William J. Cody, Jr. and William Waite; Software Manual for the Elementary Functions; Prentice-Hall; 1980

{8} George S. Fishman and Louis R. Moore; A statistical Evaluation of Multiplicative Congruential Random Number Generators with Modulus 2 ^ 31 - 1; Journal of the American Statistical Association, March 1982, Volume 77 # 377

{9} Martha August; Prime PE-T 1025, "50 Series General Architecture"; Prime Computer, Inc.; 1982

{10} Anne P. Ladd; Subroutines Reference Guide; Doc 3621-190, Revision 19.0; Prime Computer, Inc.; 1982

{11} Ivars Peterson; "Can You Count on Your Computer"; Science News, Vol. 122 #5; Jul 31, 1982

{12} W. J. Cody; "Analysis of Proposals for the Floating-Point Standard"; IEEE Computer, Volume 14 #3, March 1981

{13} David Hough; "Applications of the Proposed IEEE 754 Standard for Floating-Point Arithmetic"; IEEE Computer, Volume 14 #3, March 1981

{14} Jerome T. Coonen; "Underflow and the Denormalized Numbers"; IEEE Computer, Volume 14 #3, March 1981

{15} A Proposed Standard for Binary Floating-Point Arithmetic;
IEEE Computer, Volume 14 #3, March 1981

Appendix I

Where is the Exponent?

The following program is written in PMA (Prime Assembly Language) and is intended to indicate where the exponent is stored in the live register set of your machine. It can be entered and run without the Software Tools Subsystem being present on your system.

```
*  EXPTEST --- SEE WHERE DOUBLE FLOATING EXPONENT IS LOCATED
*
*     Eugene Spafford
*     School of Information and Computer Science
*     Georgia Institute of Technology
*     Atlanta, GA 30332
*
*
*  To assemble, load and run this test, copy lines 16 to 31
*  into a file named "exptest.cpl" and remove the "*" from
*  the first column of each line.  Then type (in Primos):
*
*       cpl  exptest
*
*
* /* exptest.cpl --- assemble, load and run the exponent test
*
* pma exptest.pma -l yes -b yes
*
* &data seg
* vload exptest.seg
* load exptest.bin
* library
* map 6
* save
* quit
* &end
*
* seg exptest.seg
*
* &stop
*
*
            SEG
            SUBR     MAIN

            LINK
MAIN        ECB      START
            PROC
```

```
START       EQU     *
            DFLD    =2.5D0
            LDLR    PB% + '13
            BNE     HIGH_HALF

            CALL    IOA$
            AP      LOWM,S
            AP      =99,SL
            PRTN

HIGH_HALF   EQU     *
            CALL    IOA$
            AP      HIGHM,S
            AP      =99,SL
            PRTN


LOWM        BCI     'Exponent is in the low half (2nd 16 bits).%.'
HIGHM       BCI     'Exponent is in the high half (1st 16 bits).%.'

            END     MAIN
            SEG
            DYNT    IOA$
            END
```

Appendix II

A Program to Detect Bit Loss in Multiplication

The following program is written in Prime Fortran 66 (FTN) and is intended to indicate whether multiplication on your machine truncates or deletes bits in the mantissa of products of double precision floating point quantities. It can be entered and run without the Software Tools Subsystem being present on your system.

```
C     CHECK_DFMP --- SEE IF DOUBLE PRECISION MULTIPLY DROPS BITS
C
C     Eugene Spafford
C     School of Information and Computer Science
C     Georgia Institute of Technology
C     Atlanta, GA 30332
C
C
C     To compile, load and run this test, copy lines 16 to 31
C     into a file named "check_dfmp.cpl" and remove the "C" from
C     the first column of each line.  Then type (in Primos):
C       cpl  check_dfmp
C
C
C
C /* check_dfmp.cpl --- compile, load and run the test to check DFMP
C
C ftn check_dfmp.ftn -l yes -b yes -64v -dynm -dclvar -prod
C
C &data seg
C vload check_dfmp.seg
C load check_dfmp.bin
C library
C map 6
C save
C quit
C &end
C
C seg check_dfmp.seg
C
C &stop
C
C
C
      INTEGER BITCNT
C
      DOUBLE PRECISION DA,DB,DC
      INTEGER IDB(4),IDC(4)
      EQUIVALENCE (IDB,DB),(IDC,DC)
```

```
C
      INTEGER LOOP,COMPAR,LOSS,IX
      DOUBLE PRECISION DCON(3)
      DATA DCON /1.0D0,16.0D0,0.125D0/
C
C
      IDB(1) = :77777
      IDB(2) = :177777
      IDB(4) = 128
C
      DO 30 IX = 1,3
        DA = DCON(IX)
        IDB(3) = 0
        DO 20 LOOP = 1,16
          IDB(3) = IDB(3)*2+1
          DC = DA*DB
          DO 10 COMPAR = 1,3
            IF (IDC(4-COMPAR) .EQ. IDB(4-COMPAR)) GO TO 10
            PRINT 70, DA
            PRINT 90, COMPAR,IDC(4-COMPAR),IDB(4-COMPAR)
            LOSS = BITCNT(IDC(4-COMPAR),IDB(4-COMPAR),COMPAR)
            PRINT 100, LOSS
            GO TO 20
   10     CONTINUE
   20   CONTINUE
   30 CONTINUE
C
C
      DO 60 IX = 1,3
        DA = DCON(IX)
        IDB(3) = 0
        DO 50 LOOP = 1,16
          IDB(3) = IDB(3)*2+1
          DC = DB/DA
          DO 40 COMPAR = 1,3
            IF (IDC(4-COMPAR) .EQ. IDB(4-COMPAR)) GO TO 40
            PRINT 80, DA
            PRINT 90, COMPAR,IDC(4-COMPAR),IDB(4-COMPAR)
            LOSS = BITCNT(IDC(4-COMPAR),IDB(4-COMPAR),COMPAR)
            PRINT 100, LOSS
            GO TO 50
   40     CONTINUE
   50   CONTINUE
   60 CONTINUE
C
C
      CALL EXIT
C
C
   70 FORMAT ('Loss of precision multiplying by ',F10.6)
   80 FORMAT ('Loss of precision dividing by ',F10.6)
   90 FORMAT ('Word ',I2,' is ',I6,' and should be ',I6)
  100 FORMAT ('Result is loss of ',I3,' bits out of 47.'//)
      END
C
C
```

```
C     BITCNT --- FIGURE LOSS OF BITS
C
      INTEGER FUNCTION BITCNT(I,J,COMPAR)
C
      INTEGER I,J,COMPAR
C
      INTEGER COUNT,AND,MASK,RS
C
C
      MASK = :100000
      DO 20 COUNT = 1,16
        IF (AND(MASK,I) .EQ. AND(MASK,J)) GO TO 10
        BITCNT = (COMPAR-1)*16+17-COUNT
        RETURN
   10   CONTINUE
        MASK = RS(MASK,1)
   20 CONTINUE
C
      BITCNT = 0
      RETURN
      END
```

Appendix III

A Program to Calculate Prime Hexadecimal Constants

The following program is written in Fortran 77 and can be used to generate Prime PMA-style hexadecimal constants from decimal inputs. The version included here was run on a Cyber 760 under NOS 2.0 to generate the constants used in the SWT Math Library. To be used effectively, if you use this program you should run it on a machine with more precision than the Primes provide.

```
C     MAKE_CONSTANT --- MAKE THE HEX CONSTANTS FOR THE LIBRARY
C
C     The following PROGRAM line is for FTN5 on the Cyber 760
C
      PROGRAM MAKCON (INPUT,OUTPUT,TAPE5=INPUT,TAPE6=OUTPUT)
C
      DOUBLE PRECISION INP,HALF,TWO,ZERO,ONE
      LOGICAL BITS(0:47)
      INTEGER I,ISIGN,EXP,J
      PARAMETER (ZERO=0.0D0,TWO=2.0D0,HALF=0.5D0,ONE=1.0D0)
      EXTERNAL PUTHEX,PUTHX2
      INTRINSIC DINT
      DOUBLE PRECISION DINT
C
C
   10 CONTINUE
      READ (5,*,END=70) INP
      IF (INP .NE. ZERO) THEN
          ISIGN = 1
          IF (INP .LT. ZERO) THEN
              ISIGN = -1
              INP = -INP
          ENDIF
C
C     START WITH 128 BIAS EXPONENT
          EXP = 128
   20     CONTINUE
          IF (INP .LT. HALF) THEN
              INP = INP*TWO
              EXP = EXP-1
              GO TO 20
C
          ELSEIF (INP .GE. ONE) THEN
              INP = INP/TWO
              EXP = EXP+1
              GO TO 20
          ENDIF
C
```

```
      ELSE
          ISIGN = 1
          EXP = 0
      ENDIF
C
      DO 30 I = 1,47
        IF (DINT(INP*TWO) .GT. ZERO) THEN
            BITS(I) = .TRUE.
            INP = INP*TWO-ONE
C
        ELSE
            BITS(I) = .FALSE.
            INP = INP*TWO
        ENDIF
   30 CONTINUE
C
      IF (INP .GE. HALF) THEN
          I = 47
   40     CONTINUE
          BITS(I) = .NOT.BITS(I)
          I = I-1
          IF ( .NOT. BITS(I+1) .AND.
     &         I .GT. 0) THEN
               GO TO 40
          ELSE IF ( .NOT. BITS(I+1)) THEN
               BITS(1) = .TRUE.
               EXP = EXP+1
          ENDIF
      ENDIF
C
C     NOW GENERATE THE 2'S COMPLEMENT IF NEGATIVE
      IF (ISIGN .LT. 0) THEN
          I = 47
   50     CONTINUE
          I = I-1
          IF ( .NOT. BITS(I+1) .AND.
     &         I .GT. 0) GO TO 50
          DO 60 J = 1,I
            BITS(J) = .NOT.BITS(J)
   60     CONTINUE
          BITS(0) = .TRUE.
C
      ELSE
          BITS(0) = .FALSE.
      ENDIF
C
      CALL PUTHEX (BITS(0))
      CALL PUTHEX (BITS(16))
      CALL PUTHEX (BITS(32))
      CALL PUTHX2 (EXP)
      GO TO 10
C
C
   70 CONTINUE
      STOP
      END
```

```
C     PUTHEX --- PUT OUT A HEXADECIMAL VALUE
C
      SUBROUTINE PUTHEX (BITARR)
C
      LOGICAL BITARR(16)
C
      INTEGER I,J,VAL
      CHARACTER*16 DIGITS
      CHARACTER*4 NUM
      DATA DIGITS /'0123456789ABCDEF'/
C
C
      DO 20 I = 1,4
        VAL = 0
        DO 10 J = 1,4
          VAL = VAL*2
          IF (BITARR((I-1)*4+J)) THEN
              VAL = VAL+1
          ENDIF
   10   CONTINUE
        VAL = VAL+1
        NUM(I:I) = DIGITS(VAL:VAL)
   20 CONTINUE
C
      WRITE (6,30) NUM
      RETURN
C
   30 FORMAT (A4)
      END
C     PUTHX2 --- PUT OUT A HEXADECIMAL VALUE
C
      SUBROUTINE PUTHX2 (EXP)
C
      INTEGER EXP
C
      INTEGER DIG,VAL,POWER2(4),LOOP
      LOGICAL ISNEG
      CHARACTER*17 DIGITS
      CHARACTER*4 NUM
      DATA DIGITS /'0123456789ABCDEF0'/
      DATA POWER2 /4096,256,16,1/
C
C
      VAL = EXP
      IF (EXP .LT. 0) THEN
          VAL = -EXP
          ISNEG = .TRUE.
          VAL = VAL-1
C
      ELSE
          ISNEG = .FALSE.
      ENDIF
C
      DO 10 LOOP = 1,4
        DIG = VAL/POWER2(LOOP)
        VAL = VAL-DIG*POWER2(LOOP)
```

```
        IF (ISNEG) DIG = 15-DIG
        DIG = DIG+1
        NUM(LOOP:LOOP) = DIGITS(DIG:DIG)
   10 CONTINUE
C
      WRITE (6,20) NUM
      RETURN
C
   20 FORMAT (A4/)
      END
```

Appendix IV

Building The SWT Math Library Tests

In General

The tests provided along with the SWT Math library may be recompiled and run on your machine to test your own routines or verify the results presented in this report. The tests are written in Fortran 77 and Fortran 66 with command files in Prime CPL. Consult your system administrator to find where the tests have been stored on disk; the default location is with the source code to the SWT Math Library routines. The single precision tests are in a separate directory from the double precision tests, but the directions given below apply to both sets of tests.

You must have the Software Tools Subsystem and the F77 compiler to run the tests! You can recode the routines written in F77 to either Ratfor or FTN, but be aware of the library that is used when you choose this option!

Make sure that the SWT Math Library has been built and installed in a directory where you can access it. Set a SWT template in your account named "=mathlib=" and equal to the SWT pathname to the library. The format to do this is:

```
template -a mathlib //<some path name here>/mathlib
```

Building the Support Routines

Attach to the directory containing the tests you wish to build and run. Modify the "subs.f77" file, if necessary, to change the library routines to be tested. The routines in the "subs.f77" file which begin with the letter Z are the routines to modify to invoke the correct library functions.

Next, you need to build the support routines. To do this, simply run the SWT shell file "make_support". This will cause the files "main.b" and "sublib" to be created in your account.

Next, edit the file "run_test.cpl" so that any necessary local libraries get loaded along with the tests. Also include any special commands that you might wish to execute as part of the tests.

Running a Test

If you execute the SWT shell file "make" with the name of a test to run (asin, atan, exp, log, power, sqrt, sin, sinh, tan,

or tanh) the SWT shell files and associated Prime CPL files will compile and load the appropriate test programs, execute them with output captured to comoutput files, and then produce a file with labelled results and a report generated by CMPF. The file created will be named after the test executed, with the string ".comparison" added to the end of the name. For example, if you executed the command

```
make power
```

the file "power.comparison" would be created in your account.

If you wish to make further modifications to the test software, examine the SWT shell files and CPL files to determine what needs to be modified.

ADDENDUM

Arnold D. Robbins

August, 1984

Introduction

For Release 9 of the Software Tools Subsystem, in order that there should only be one math library, the old, locally supported, math library, "vswtml", has been merged with the new library described in this report, "vswtmath". This addendum describes these routines.

Deleted Functions

The functions dacos, dasin, dbexp, dbsqrt, dflot, and drand, have all been deleted from "vswtml", since there are new routines to take their places.

Remaining Routines

The following pages contain the Software Tools Reference Manual entries for the remaining routines which have been added to "vswtmath" from "vswtml".

Note that although the original "vswtmath" routines are listed in Section 2 of the SWT Reference Manual, these routines are listed in Section 4, even though they are all in one library.

gcd (4) --- determine greatest common divisor of two integers 07/20/84

Calling Information

```
long_int function gcd (x0, x1)
long_int x0, x1
```

Library: vswtmath (Subsystem mathematical library)

Function

'Gcd' determines the greatest common divisor of the two long integers specified as arguments. The function return is the GCD (always positive).

Implementation

'Gcd' is a straightforward implementation of Euclid's algorithm.

Bugs

Behavior with nonpositive arguments may be considered irrational by some.

See Also

invmod (4)

Calling Information

```
long_int function invmod (x1, x0)
long_int x1, x0
```

Library: vswtmath (Subsystem mathematical library)

Function

'Invmod' is used to find the inverse of 'x1' in the ring of integers modulo 'x0'. The function return is the inverse if it could be found, or ERR if 'x1' and 'x0' are not relatively prime.

Implementation

'Invmod' uses a variant of Euclid's greatest common divisor algorithm.

Bugs

Rational behavior for nonpositive arguments has not been established.

Locally supported.

See Also

gcd (4)

Calling Information

```
long_int function prime (i)
long_int i
```

Library: vswtmath (Subsystem mathematical library)

Function

'Prime' is used to retrieve a specified prime number. The argument is the ordinal of the prime number desired. The function return is the specified prime. For example, if 'i' is 1, the function return is 2; if 'i' is 3, the function return is 5, etc.

'Prime' uses the table of prime numbers in the file "=aux=/primes". This file contains the prime numbers up to one million in long-integer binary format. If "=aux=/primes" is unreadable or if 'i' is less than one or greater than 78498, the function return is zero.

Implementation

The file "=aux=/primes" is opened for reading. The read/write pointer for the file is then moved to the desired location and the prime number read. The file is then closed.

Calls

open, close, mapfd, Primos prwf$$

Bugs

Should probably raise cain if the prime numbers file is not available, rather than meekly returning zero.

Locally supported.

pwrmod (4) --- calculate an exponential modulo a given modulus 07/20/84

Calling Information

```
long_int function pwrmod (p, e, n)
long_int p, e, n
```

Library: vswtmath (Subsystem mathematical library)

Function

'Pwrmod' is used to perform an integer exponentiation in the ring of integers modulo a given modulus. The argument 'p' is the base of the expression, 'e' is the exponent, and 'n' the modulus. The function return is p**E (mod n).

Implementation

'Pwrmod' examines the exponent a bit a time, squaring the intermediate result accumulated so far and multiplying it by the base whenever the selected bit is a 1. Each operation is performed modulo 'n', so that intermediate results don't become excessively large.

See Also

invmod (4)

Calling Information

```
subroutine set_copy (source, destination)
pointer source, destination
```

Library: vswtmath (Subsystem mathematical library)

Function

'Set_copy' duplicates one set in another. For proper operation, the source set should be larger than or equivalent in size to the destination set. The source set is not altered by the copy operation.

All set manipulation routines make use of dynamic storage, which must be initialized before use. See 'dsinit' for further information.

Note that all set manipulation routines have long names. To avoid unique name conflicts with other routines, any Ratfor program using the set routines should include the following statement:

```
include "=src=/lib/math/swtmlb_link.r.i"
```

Implementation

'Set_copy' uses the size field encoded in the first word of each set to determine the number of words in the bit vector to be copied. A simple loop implements the copy.

Bugs

Should handle sets of different sizes properly.

See Also

other set operations ('set_?*') (4)

set_create (4) --- generate a new, initially empty set 07/20/84

Calling Information

```
pointer function set_create (set, size)
pointer set
integer size
```

Library: vswtmath (Subsystem mathematical library)

Function

'Set_create' is used to create a Pascal-style bit vector representation for a set of integers from 1 to 'size'. The function return and the variable 'set' are set to the address in dynamic storage of the newly-created set.

All set manipulation routines make use of dynamic storage, which must be initialized before use. See 'dsinit' for further information.

Note that all set manipulation routines have long names. To avoid unique name conflicts with other routines, any Ratfor program using the set routines should include the following statement:

```
include "=src=/lib/math/swtmlb_link.r.i"
```

Implementation

'Set_create' calls 'dsget' to obtain a contiguous array of 16-bit words that is large enough to represent a bit vector with 'size' elements. The first word of this array is set to 'size' for use by other set manipulation routines. A call to 'set_init' then insures that the new set is empty.

Arguments Modified

set

Calls

dsget, set_init

See Also

other set routines ('set_?*') (4)

Calling Information

```
subroutine set_delete (element, set)
integer element
pointer set
```

Library: vswtmath (Subsystem mathematical library)

Function

'Set_delete' is used to remove a given element from a set. The first argument is the element (an integer between one and the maximum set size, inclusive), and the second is the set from which it is to be removed.

All set manipulation routines make use of dynamic storage, which must be initialized before use. See 'dsinit' for further information.

Note that all set manipulation routines have long names. To avoid unique name conflicts with other routines, any Ratfor program using the set routines should include the following statement:

```
include "=src=/lib/math/swtmlb_link.r.i"
```

Implementation

The element selected is compared to the size field of the set; if invalid, 'set_delete' prints an error message and terminates the program. Otherwise, the position of the element in the bit vector is calculated, and the bit is reset by straightforward logical operations.

Calls

error

See Also

other set operations ('set_?*') (4)

Calling Information

```
integer function set_element (element, set)
integer element
pointer set
```

Library: vswtmath (Subsystem mathematical library)

Function

'Set_element' returns 1 if 'element' is a member of the set 'set', 0 otherwise. The argument 'element' must be an integer from 1 to the maximum size of the set, inclusive. The argument 'set' must have been created beforehand with 'set_create'.

All set manipulation routines make use of dynamic storage, which must be initialized before use. See 'dsinit' for further information.

Note that all set manipulation routines have long names. To avoid unique name conflicts with other routines, any Ratfor program using the set routines should include the following statement:

```
include "=src=/lib/math/swtmlb_link.r.i"
```

Implementation

If 'element' is not in the range of allowable set elements for the given set, the program is terminated by a call to 'error'. Otherwise, the location of the element in the bit vector is calculated, and the function returns the value of the bit at that position.

Calls

error

See Also

other set routines ('set_?*') (4)

Calling Information

```
logical function set_equal (set1, set2)
pointer set1, set2
```

Library: vswtmath (Subsystem mathematical library)

Function

'Set_equal' determines if two sets contain the same members. The sets need not be of equal length.

All set manipulation routines make use of dynamic storage, which must be initialized before use. See 'dsinit' for further information.

Note that all set manipulation routines have long names. To avoid unique name conflicts with other routines, any Ratfor program using the set routines should include the following statement:

```
include "=src=/lib/math/swtmlb_link.r.i"
```

Implementation

'Set_equal' makes two calls on 'set_subset'. The function return is true if 'set1' is a subset of 'set2' and 'set2' is a subset of 'set1', false otherwise.

Calls

set_subset

See Also

other set routines ('set_?*') (4)

Calling Information

```
subroutine set_init (set)
pointer set
```

Library: vswtmath (Subsystem mathematical library)

Function

'Set_init' initializes a set created by 'set_create'. An initialized set is empty, i.e. contains no members.

All set manipulation routines make use of dynamic storage, which must be initialized before use. See 'dsinit' for further information.

Note that all set manipulation routines have long names. To avoid unique name conflicts with other routines, any Ratfor program using the set routines should include the following statement:

```
include "=src=/lib/math/swtmlb_link.r.i"
```

Implementation

'Set_init' simply clears all elements of the bit vector portion of the data structure addressed by its first argument.

See Also

other set routines ('set_?*') (4)

Calling Information

```
subroutine set_insert (element, set)
integer element
pointer set
```

Library: vswtmath (Subsystem mathematical library)

Function

'Set_insert' is the primary means of placing a given element in a set. 'Element' must be an integer between one and the maximum size of the set, inclusive; 'set' must be a pointer to a set data structure created by 'set_create'. If it is within range, the given element is marked "present" in the bit vector associated with the set.

All set manipulation routines make use of dynamic storage, which must be initialized before use. See 'dsinit' for further information.

Note that all set manipulation routines have long names. To avoid unique name conflicts with other routines, any Ratfor program using the set routines should include the following statement:

```
include "=src=/lib/math/swtmlb_link.r.i"
```

Implementation

If the element is out of range, a call to 'error' is made to inform the user and terminate the program. Otherwise, the location of the element in the bit vector is determined and a few logical operations are employed to set the selected bit.

Calls

error

See Also

other set routines ('set_?*') (4)

Calling Information

```
subroutine set_intersect (set1, set2, destination)
pointer set1, set2, destination
```

Library: vswtmath (Subsystem mathematical library)

Function

'Set_intersect' determines the intersection of the sets given as its first two arguments and places that intersection in the set specified by the third. For proper operation, all three sets should be equal in size.

All set manipulation routines make use of dynamic storage, which must be initialized before use. See 'dsinit' for further information.

Note that all set manipulation routines have long names. To avoid unique name conflicts with other routines, any Ratfor program using the set routines should include the following statement:

```
include "=src=/lib/math/swtmlb_link.r.i"
```

Implementation

Does a word-by-word logical 'and' of the bit vectors for the first two sets, placing the result in the third.

Bugs

Should be fixed to work with sets of differing lengths.

See Also

other set routines ('set_?*') (4)

Calling Information

```
subroutine set_remove (set)
pointer set
```

Library: vswtmath (Subsystem mathematical library)

Function

'Set_remove' reclaims the dynamic storage space used by a set data structure. It is the inverse of 'set_create'. To prevent dynamic storage space from becoming irretrievably lost, sets should always be removed by a call to 'set_remove' when they are no longer needed.

All set manipulation routines make use of dynamic storage, which must be initialized before use. See 'dsinit' for further information.

Note that all set manipulation routines have long names. To avoid unique name conflicts with other routines, any Ratfor program using the set routines should include the following statement:

```
include "=src=/lib/math/swtmlb_link.r.i"
```

Implementation

Calls 'dsfree' to throw away the storage space used by the internal data structure.

Calls

dsfree

See Also

other set routines ('set_?*') (4), dsinit (2), dsget (2), dsfree (2)

Calling Information

```
logical function set_subset (set1, set2)
pointer set1, set2
```

Library: vswtmath (Subsystem mathematical library)

Function

'Set_subset' returns the logical value '.true.' if and only if its first argument points to a set that is a subset of or equal to the set pointed to by its second argument. The sets need not be of equal length.

All set manipulation routines make use of dynamic storage, which must be initialized before use. See 'dsinit' for further information.

Note that all set manipulation routines have long names. To avoid unique name conflicts with other routines, any Ratfor program using the set routines should include the following statement:

```
include "=src=/lib/math/swtmlb_link.r.i"
```

Implementation

If one set is larger than the other, it is checked to make sure that none of the higher-order elements is present. The subset condition is then true if and only if every element of 'set1' is also an element of 'set2', a statement which can be checked a word at a time with the proper logical operations.

Calls

set_element

See Also

other set routines ('set_?*') (4)

Calling Information

```
subroutine set_subtract (set1, set2, destination)
pointer set1, set2, destination
```

Library: vswtmath (Subsystem mathematical library)

Function

'Set_subtract' performs the set subtraction operation, i.e. places in the set 'destination' those elements of 'set1' that are not in 'set2'. For proper operation, all three sets should be the same size.

All set manipulation routines make use of dynamic storage, which must be initialized before use. See 'dsinit' for further information.

Note that all set manipulation routines have long names. To avoid unique name conflicts with other routines, any Ratfor program using the set routines should include the following statement:

```
include "=src=/lib/math/swtmlb_link.r.i"
```

Implementation

Since sets are represented as bit vectors, the subtraction operation is performed by logically 'and'ing the elements of the first set with the negation of the elements of the second set.

Bugs

Should work with sets of differing sizes.

See Also

other set routines ('set_?*') (4)

Calling Information

```
subroutine set_union (set1, set2, destination)
pointer set1, set2, destination
```

Library: vswtmath (Subsystem mathematical library)

Function

'Set_union' computes the union of 'set1' and 'set2', placing the result in 'destination'. For proper operation, all three sets should be the same size.

All set manipulation routines make use of dynamic storage, which must be initialized before use. See 'dsinit' for further information.

Note that all set manipulation routines have long names. To avoid unique name conflicts with other routines, any Ratfor program using the set routines should include the following statement:

```
include "=src=/lib/math/swtmlb_link.r.i"
```

Implementation

The set union is computed by logically 'or'ing the bit vectors associated with 'set1' and 'set2'.

Bugs

Should work with sets of differing sizes.

See Also

other set routines ('set_?*') (4)