K I M A T H S U B R O U T I N E S

MOS Technology, 901 California Avenue

Palo Alto, CA 94304

TABLE OF CONTENTS

LIST OF APPENDICES

INTRODUCTION

This manual contains instructions on the use of the KIMath subroutine package. KIMath provides the software for doing floating-point (decimal) arithmetic on the KIM microcomputer system.

Because of the wide availability of calculators and larger computers we take decimal arithmetic for granted. Perhaps you were surprised when you first read the 6502 microcomputer programming manual and discovered that the 6502 instruction set could do addition and subtraction of whole numbers only in the range of +128 to -127! If you tried writing a program to multiply two eight-bit binary words to produce a sixteen-bit result, you will appreciate the ability in KIMath to multiply two sixteen-digit BCD numbers just by calling a subroutine.

KIMath gives you the capability to add, subtract, multiply, divide and calculate square roots. In addition, KIMath can calculate logs and antilogs as well as tangents and arc tangents. Other common mathematical functions can be calculated by combining these basic operations. KIMath also has special subroutines to make it easy to evaluate polynomial expressions, which can be used to approximate most mathematical functions.

Since your KIM system, like any microcomputer, has to break large numbers into several successive words in memory and then work on them one word at a time, floating-point arithmetic is much more time- and memory-consuming than simple binary arithmetic. A single floating-point number may use as many as 18 words of memory in your KIM system, and the multiplication of two 16-digit numbers may take as long as 40 milliseconds --40,000 machine cycles! The same multiplication using only 4 digits of precision would take only 10 milliseconds, so if you will accept less precision you can speed up your calculations.

To make this possible KIMath allows you to specify the number of digits to be used in any calculation. You may use the same precision for your entire program or you may adjust the precision depending on the needs of each step in your calculations.

The KIMath subroutines are an ideal addition to the KIM resident assembler. Naturally, the assembler is not required in order to use KIMath.

Because each floating-point number is stored in several memory words, KIMath provides utility routines to move blocks of words corresponding to each number to and from the locations required to perform the calculations.

Finally, KIMath allows floating point values to be stored in several different representations or formats to improve computational efficiency while conserving memory. Routines are provided to convert between these formats.

Chapter 1 of this manual gives you an overview of the concepts used in KIMath. Chapter 2 shows how KIMath could be used to solve a simple equation. Chapter 3 examines the different storage formats and Chapters 4 and 5 cover the use of the individual subroutines.

Several appendices are provided for the user who desires a more rigorous explanation of the techniques used in designing KIMath. A storage map and a complete program listing are also included.

The KIMath subroutines were carefully designed and thoroughly tested to insure accuracy and correct operation. The final decision of their suitability, however, rests with the user and no warranty of fitness is implied by this document. If you suspect an error in these routines, contact the Manager of Product Support at MOS Technology Sales, Inc. 901 California Avenue, Palo Alto, California 94304.

This manual assumes familiarity with 65OX assembly language. You should first study the KIM Resident Assembler if necessary.

CHAPTER 1
Overview

1.0 Basic Concepts

KIMath is a package of subroutines designed for use with any MCS65OX microprocessor-based system. The subroutines assist the user in doing the floating-point arithmetic necessary for many business and scientific applications. In particular, KIMath is designed to work with the KIM resident assembler, although KIMath may be integrated into any MCS65OX program.

1.1 Definitions

Before examining the operation of KIMath, here are definitions of some terms used throughout this manual:

WORD or BYTE - the basic unit of data used in the KIM system. It is composed of eight BITS or binary digits, each of which may only have values of 0 or 1. The microprocessor can only operate on one word at a time.

REGISTER - has a special meaning in this manual, where we define it to mean a group of adjacent words in memory. Since the numbers manipulated by KIMath are too big to store in a single word, we call the group of words used to hold a single floating-point number a register. This usage should not be confused with the registers A, X, Y, stack pointer and status which are single words stored within the microprocessor, not the memory.

POINTER - two adjacent words in memory containing the address of another memory location. The programming convention in the 6502 is to store the two low-order hex characters of the address in the first word and the two high-order characters in the second word. For example, given:

Label	Location	Contents
POINTL	17FA	00
POINTH	17FB	1C

we could say that the register (POINTL, POINTH) is a pointer which currently points to the address 1C00.

ROUTINE - a segment of machine-language code which performs a specified operation. In this document it is synonymous with a subroutine.

ARGUMENT - the KIMath routines expect to find the numbers they are to operate on in fixed locations in memory. When the KIMath routine ADD is called by a program, it will take data from two fixed locations, add them together and put the result in a third fixed location. The data in the input locations are the arguments for the ADD routine. It is usually necessary to transfer the data from other registers in memory into the argument registers and transfer the calculated result to another memory register for storage. KIMath provides routines for moving registers about in memory.

WHOLE NUMBER or INTEGER - a number which does not contain a decimal point.

FLOATING-POINT NUMBER - a number containing a decimal point. Note: 12 is an integer while 12.0 is a floating-point number.

1.2 Number Systems

All modern digital computers are based on the binary system, where numbers are represented as groups of signals which are on or off, 1 or 0. For instance, the decimal number 39 is represented in binary as 100111. For convenience, binary numbers are often converted to groups of four and represented in base 16 or hexadecimal. Decimal 39 is represented as:

2	7	hexadecimal
0010	0111	binary

Because our daily arithmetic is done in decimal, the hexadecimal format is still difficult to interpret. The MCS6502 microprocessor also has the capability to do arithmetic in a modified binary format known as BCD or binary coded decimal. In this system each eight-bit word is divided into two four-bit fields, each representing a decimal digit. Decimal 39 is now represented as:

3	9	decimal
0011	1001	BCD

memory words

Although less efficient in use of storage, this representation is more readily understood. Appendix H of the 6502 Programming Manual contains a review of binary and BCD arithmetic.

We can represent any whole number as a series of BCD digits, with two digits packed in each memory word. The next problem arises when we wish to store numbers such as 3.14159 or 1276.3333. These numbers contain decimal fractions and BCD notation contains no provisions for decimal points. Very large or very small numbers such as 987000000000 or 0.000000625 often have large numbers of leading or trailing zeros which expand storage requirements. To overcome these problems, KIMath uses "scientific notation" which stores decimal numbers in two parts. The number is first reduced to a single whole number followed by a decimal fraction. This is called the mantissa. The second part is the number of decimal places we must shift the new decimal point to get back to the original number. This is called the exponent (scientific notation is also referred to as exponential notation). If the decimal point must be shifted to the right the exponent is a positive number; if a left shift is necessary the exponent is negative. For instance:

987000000000 becomes 9.87 E+11

0.000000625 becomes 6.25 E-7

(An E for exponent is commonly used to separate the two values.) Similarly:

-6.273411 E+3 is -6273.411

which is an example of a number with a negative mantissa and a positive exponent.

1.3 How Will I Use KIMath in My Program?

Your program will use data of two types: variables whose values will change from one program run to the next or within a single run, and constants which will always have the same value. For example, a simple program to compute the area of a circle given its radius will solve the equation:

$$\mathrm{AREA} = 3.14\,\mathrm{R}^2$$

The values for AREA and R will change each time you run the program; they are variables. The values 3.14 (pi) and 2 (the power to which R is raised) are constants.

When you write your program in assembly language you define memory locations to store this data. When you reserve storage for data used with KIMath, you will have to reserve several words of storage for each variable and constant and you will have to define a register for each data value. The size of each register will depend on the number of digits you wish to store--that is, how much precision you wish to maintain in your calculations.

When you wish to perform calculations using KIMath, you will have to transfer the value of the variables or constants from their memory storage registers into the argument registers (RX and RY) of KIMath, call the appropriate KIMath routine to do the calculation, and then transfer the answer or result from the KIMath result register (RZ) back to a variable storage register. KIMath provides routines for doing these register moves.

In addition, KIMath provides for a variety of data formats to minimize storage requirements and speed computation. KIMath provides routines for converting data between these different formats.

1.4 Summary of KIMath Calculation Subroutines

Subroutine	Action	Description
ADD	RX + RY -> RZ	This routine allows the user to add two floating-point numbers. The user may select the number of digits in the mantissas of the arguments.
SUB	RX - RY -> RZ	This routine allows the user to subtract two floating-point numbers. The user may select the number of digits in the mantissas of the arguments.
MLTPLY	RX * RY -> RZ	This routine allows the user to multiply two floating-point numbers. The user may select the number of digits in the mantissas of the arguments.
DIVIDE	RX / RY -> RZ	This routine allows the user to divide two floating-point numbers. The user may select the number of digits in the mantissas of the arguments.

1.4 Summary of KIMath Calculation Subroutines (Continued)

Subroutine	Action	Argument Range	Description
SQRT	$\sqrt{(RX)}$ -> RZ	{1, 100}	This routine allows the user to form the square root of a floating-point number. The user may select the number of digits in the mantissa of the argument.
LOG	LOG10(RX) -> RZ	{$1/\sqrt{10}$, $\sqrt{10}$}	This routine allows the user to form the common logarithm of a floating-point number. The user may select the number of digits in the mantissa of the argument, but at most 14 will be used and at most 8 will be returned.
TENX	10^(RX) -> RZ	{0, 1}	This routine allows the user to form the common antilog of a floating-point number. The user may select the number of digits in the mantissa of the argument, but at most 12 will be used and at most 8 will be returned.
TANX	TAN(RX) -> RZ	{0, 1}	This routine allows the user to form the tangent of a floating-point number. The user may select the number of digits in the mantissa of the argument, but at most 14 will be used and at most 8 will be returned. The argument and result are in radians.
ATANX	ARCTAN(RX) -> RZ	{0, 1}	This routine allows the user to form the arc tangent of a floating-point number. The user may select the number of digits in the mantissa, but at most 14 will be used and at most 8 will be returned. The argument and result are in radians.

1.5 KIMath Formats

Several variations on exponential notation are used in KIMath.

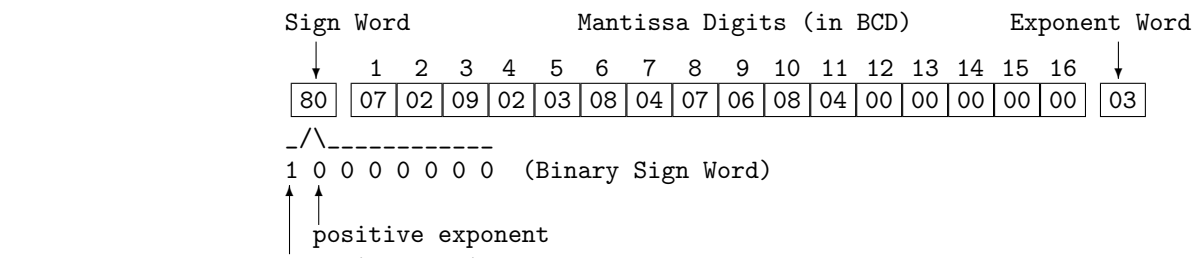
1.5.1 Computational Format

Before actual computation can take place, every number transferred to a KIMath argument register must be converted to an eighteen-word format:

a. The first word contains the sign of the mantissa in bit 7 (1 if minus, 0 if plus) and the sign of the exponent in bit 6.

b. The next sixteen words contain the sixteen digits of the mantissa, one digit per word, using BCD notation.

c. The last word contains the value of the exponent (which must be between 0 and 99) as two BCD numbers. For example: a value of -7292.3847684 would be stored in computational format as:



This format allows KIMath to calculate quickly since every floating-point number is contained in the same number of words and each word contains only a single digit (except for the exponent word).

This is a very inefficient use of storage since the left half of each word in the mantissa always contains a zero, and each mantissa contains sixteen digits even if many are just trailing zeros.

1.5.2 Packed Format

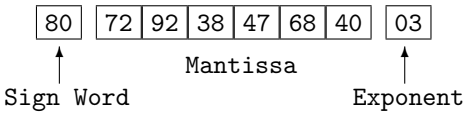
To avoid this waste of storage KIMath provides a more efficient format for storing data when it is not being used in a computation:

a. The first word contains the signs of the mantissa and exponent as in computational format.

b. Succeeding words each contain two digits of the mantissa in BCD. The number of words used is defined by the user.

c. The final word contains the two digits of the exponent, as in computational format.

If the user had decided to use 12 digits of precision in storage, the value of -7292.3847684 would be stored as:



thus using eight words of storage instead of eighteen. KIMath contains routines to convert between computational and Packed format. Packed format is the form

usually used for memory registers storing program variables. It is your responsibility to keep track of how many digits of precision are used for each variable, since KIMath will need that information when moving the memory register to the KIMath argument registers and converting to computational format. You may choose to use the same precision for all storage registers to simplify this task.

1.5.3 Unpacked (ASCII) Format

The ASCII code used by the KIM-1 serial communications interface represents each character typed in or out as two hexadecimal characters packed in a single word: the character 'A' is 41, a blank is 20, a carriage return is 0D. The digits 0 through 9 are coded as themselves plus 30: '4' is 34, '7' is 37, etc.

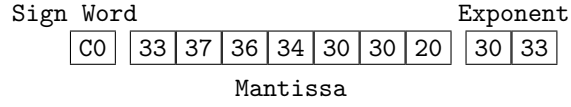
To make interfacing easier, KIMath has an unpacked format which uses ASCII codes rather than BCD.

a. The first word contains the sign bits for the mantissa and exponent as used for computational format.
b. Up to 16 words (depending on the precision desired) are used for the mantissa. Each word contains one digit of the mantissa in ASCII code.
c. The last two words contain the two digits of the exponent, also in ASCII.

Using six digits of precision the value

-0.003764 (or -3.764 E-3)

would be stored as:



This format is usually used only when preparing to transfer data to or from a line of terminal input or output.

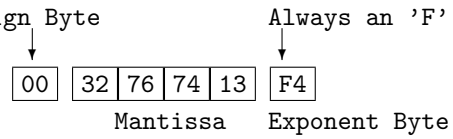
1.5.4 Constant Format

KIMath calculates functions by solving polynomial equations which closely approximate the desired function. Because the polynomials have many terms and the coefficients have differing precision requirements, a special format is used in KIMath to store constants. The format allows all constants to

be stored together in a list without having to specify elsewhere the precision of each constant. The constant format is composed of:

a. A sign byte, as in the other formats.
b. Up to eight bytes in packed BCD format (two BCD characters per byte) containing the mantissa.
c. An exponent byte of two characters. The first (leftmost) character is always a hexadecimal 'F' and the second character is the one-digit BCD exponent. Thus, constants may only have exponents between -9 and +9.

Thus the constant 32767.413 (3.2761413 E+4) will be stored as:



Thus, KIMath can read a constant from memory by just knowing its starting location. It will continue to scan successive bytes until it locates one containing an 'F' and know it has reached the end of the constant. The next byte will contain the sign byte of the next constant.

The KIMath routine USRLKP is used to transfer a constant format value from memory to the argument register RY and convert it to computational format.

1.6 Summary of Formats

KIMath supports four different formats:

a. Computational Format - used for the argument and result registers RX, RY and RZ. Although inefficient in use of storage, they permit quick calculation.

b. Packed BCD Format - most efficient in use of storage.

c. Unpacked (ASCII) Format - less efficient in use of storage, but easier to convert to and from external devices.

d. Constant Format - efficient in use of storage and a good way to store tables, but limited in the size of the exponents which can be used.

CHAPTER 2

A Sample Program Using KIMath

2.0 Throughout the remainder of this manual we will develop the techniques required to create the following program:

An engineer wishes to create a program to find the resonant frequency of a tuned circuit, given values for L, the inductance, and C, the capacitance, in the circuit. The formula is:

$$f = \frac{1}{2\pi\sqrt{LC}}$$

The program will have to contain the following operations:

1. The starting address of the program must be defined and storage registers for L, C, and the constants 2π and 1 must be created.
2. The precision of the calculations must be defined to KIMath.
3. The value of L must be read in and transferred to a KIMath argument register.
4. The value of C must be read in and transferred to a KIMath argument register.
5. L must be multiplied by C.
6. Move the answer to step 5 to a KIMath argument register. Compute the square root. Since the square root routine in KIMath will only accept numbers between 1 and 100 we must adjust the exponent of the result of step 5.
7. Get the first constant (2pi) and transfer it to an argument register.
8. Move the computed square root to another argument register and multiply.
9. Get the second constant (1) and put in an argument register.
10. Move the result of step 8 to an argument register and do the division.
11. Move the result of the calculation back to the register originally containing L and type it out.

To summarize as an assembly language program,

```
;(1)  define storage
;(2)  define precision
```

```
;(3)    read L and transfer to RX
;(4)    read C and transfer to RY
;(5)    compute L times C--answer stored in RZ
;(6)    move RZ to RX and
;       compute square root--adjust exponent of result
;(7)    get 2pi to RX
;(8)    move RZ to RY and multiply
;(9)    get 1 (constant) and put in RX
;(10)   move RZ to RY and divide
;(11)   move RZ back to L and print out
       .END
```

To start with, our program only contains comments. As we learn more about KIMath we will begin to write the assembly language statements to implement the program.

To help you visualize what is happening in the program, the following table shows the contents of the two argument registers (RX, RY) and the result register (RZ) after each program step:

At the end of step #	RX	RY	RZ
1	--	--	--
2	--	--	--
3	L	--	--
4	L	C	--
5	L	C	LC (L times C)
6	LC	C	$\sqrt{LC}$ (adjusted)
7	2π	2π*	$\sqrt{LC}$
8	2π	$\sqrt{LC}$	$2\pi\sqrt{LC}$
9	1	1*	$2\pi\sqrt{LC}$
10	1	$2\pi\sqrt{LC}$	$1/2\pi\sqrt{LC}$

*The constant appears in both RX and RY because it is moved from storage to RY and then to RX.

CHAPTER 3

Defining Storage for KIMath

Variables and Constants

3.0 In the last chapter you learned that KIMath uses several different formats for memory storage registers used in KImath calculation. Now let us see how we set aside memory to hold these variables and constants.

3.1 Defining Precision

If we define an even number NDIG which is the Number of DIGits of precision we wish to use, storing a number of that precision will take:

a. NDIG + 3 words of storage in Unpacked format.
b. NDIG/2 + 2 words of storage in Packed format.
c. NDIG/2 + 2 words of storage in Constant format. (Constants may be stored in fewer digits if they have leading or trailing zeros.)
d. Computational format always takes 18 words.

In our sample program (see Chapter 2) we defined two variables, L and C. Since we are reading them in from a terminal we will store them in Unpacked (ASCII) format. We also needed two constants (2pi and 1) and will store them in constant format.

Since we may wish to change the precision of our program, we will define the variable NDIG to the assembler and use it to calculate our storage. The following code will define storage for our sample program:

```
;(1)  define storage
NDIG = 8; we will carry 8 digits of precision
* = $5000; start the program at location 5000 (hex)
L   * = * + NDIG + 3; reserve storage for L
C   * = * + NDIG + 3; reserve storage for C
TWOPI .BYTE $00, $62, $83, $18, $54, $F0; define 2pi
ONE   .BYTE $00, $10, $F0; define 1.0
```

Note that constant format allows us to eliminate the trailing zeros in the constant ONE. The symbol "*" stands for the current value of the program counter. Refer to the assembler documentation if you are not familiar with assembler directives.

Our program storage map will now be:

5000	-	500A	storage for L
500B	-	5015	storage for C
5016	-	501B	storage for 2π constant
501C	-	501E	storage for 1.0 constant
501F	-	....	program storage

If we later re-assemble our program and change the value of NDIG to 4 or 10, to increase or decrease the precision of our calculations, the amount of storage allocated will automatically decrease or increase.

CHAPTER 4

Introduction to the KIMath Subroutines

4.0 Now that we have examined the data formats and storage requirements of KIMath we are ready to examine the subroutines themselves. The following table summarizes the KIMath subroutines, their arguments and their functions:

Summary of KIMath Subroutines

Subroutine	Arguments	Function
SAVXY	None	Save processor X and Y index registers. Recall previously saved X and Y registers.
RCLXY	None	
IPREC	PREC, EXTRA	PREC + EXTRA - N
PLOADX	ARGXL, ARGXH	Move user register in packed format to RX or RY and change to computational format.
PLOADY	ARGYL, ARGYH PREC	
ULOADX	ARGXL, ARGXH	Move user register in unpacked format to RX or RY and change to computational format.
ULOADY	ARGYL, ARGYH PREC	
PSTRES	RESL, RESH	Move contents of RE to user register in packed or unpacked format.
USTRES	PREC	
USRLKP	KONL, KONH, NKON	Transfer constant data to RY and convert to computational format.
POLY	KONL, KONH, NKON, DEG	Evaluates a polynomial.
CLRX	None	Clear RX, RY, or RZ
CLRY		
CLRZ		
DECHEX	CNT	Converts contents of CNT from decimal to hexadecimal notation.
XSY	RX, RY	Exchange contents of RX and RY.

4.1 Working Storage

The working storage area used by the KIMath subroutines is divided into three parts: the computational registers (RA, RB, RQ, RM and RN) which are not available to the user; the argument/result registers (RX, RY, RZ); and the pointers, counters and indicators in page zero.

4.1.1 The Argument/Result Registers - RX, RY, RZ

The registers RX and RY are the argument registers and RZ is the result register. This means that when RX and RY are set to specific values and, for example, the MLTPLY routine is called, then the product is returned in RZ. These registers are each 18 bytes long to permit a maximum of 16 digits for the mantissa. The registers RX, RY and RZ are CLeaRed directly by means of the subroutines CLRX, CLRY and CLRZ, respectively. The registers RX, RY, RZ contain a sign byte (SX, SY, SZ) and an exponent byte (EX, EY, EZ). The sign byte is the first and the exponent byte the last. Remember that data in RX, RY or RZ must be in computational format.

These registers may be transferred about by means of the subroutines whose names are of the form MVsd, where "s" stands for source and "d" for destination. For example, MVXY MoVes the contents of RX to RY.

4.1.2 The Pointers, Counters and Indicators

Before calling any KIMath routine, you may wish to SAVE the processor's X and Y registers in locations TMPX and TMPY by means of the routine SAVXY. The X and Y registers are ReCaLled by calling RCLXY.

Before calling an arithmetic or transcendental KIMath routine you must specify the number of digits of PRECision desired by loading locations PREC and EXTRA with two unsigned, binary values. The first value, PREC, specifies the number of digits of mantissa which the user expects in the RZ register after completion of a calculation and PREC+EXTRA specifies the number of digits of precision used by the subroutine in computing the result. The limitations on these values are: 0 < PREC+EXTRA <= 16 and:

Function	Internal Precision*	Precision* Returned
ADD	PREC+EXTRA	PREC
SUB	PREC+EXTRA	PREC
MLTPLY	PREC+EXTRA	PREC
DIVIDE	PREC+EXTRA	PREC
LOG	14	Lesser of PREC or 8
TENX	12	Lesser of PREC or 8
TANX	14	Lesser of PREC or 8
ATANX	14	Lesser of PREC or 8
SQRT	PREC+EXTRA	PREC

*Note: Here the word "precision" refers not to error but to the number of digits used in calculation.

The sum PREC+EXTRA may be computed by calling the routine named IPREC. This routine computes the sum and stores the result in N. N is the byte which the arithmetic and transcendental routines use to decide the precision of internal calculation. If the user so desires, the use of PREC and EXTRA may be avoided and N used directly. However, the routines LOG, TENX, TANX and ATANX may change any value placed in N.

The bytes in page 0 named ARGXL, ARGXH, ARGYL, and ARGYH are used to contain the addresses of user-defined registers in the RAM memory area. By using these address pointers and the subroutines PLOADX, PLOADY, ULOADX and ULOADY, the user loads the KIMath argument registers from any place in memory.

The bytes RESL and RESH are used in conjunction with the KIMath routines PSTRES (Packed SToring of RESults) and USTRES (Unpacked Storing of RESults) to place the contents of RZ in a user-defined register, which is specified by the pointer (RESL, RESH).

The bytes KONL, KONH are used together with DEG and KIMath routines USRLKP and POLY to evaluate any polynomial, subject, of course, to the constraints that the coefficients have fewer than 16 digits of mantissa and that the exponent has only one digit. More shall be said about these techniques in the section on Advanced Applications.

CHAPTER 5
The KIMath Subroutines

5.1 The KIMath subroutines may be divided into four groups, Functions, Conversions, User Utilities and System Utilities. This section will deal with the first three classes.

5.1.1 The Function Subroutines are called ADD, SUB, MLTPLY, DIVIDE, SQRT, LOG, TENX, TANX, and ATANX. These routines may be divided into the Arithmetic Routines and the Transcendental Routines. Each of the routines requires that the argument registers (RX and RY) be in the Computational Format.

The following tables give some relevant information about the Arithmetic and Transcendental Routines:

Arithmetic Routines

Subroutine	Precision (1)	Error (2)	Symbolic Action
ADD	0 < N <= 16	1 Count	RZ <- RX + RY (3)
SUB	0 < N <= 16	1 Count	RZ <- RX - RY (3)
MLTPLY	0 < N <= 16	1 Count	RZ <- RX * RY
DIVIDE	0 < N <= 16	1 Count	RZ <- RX / RY
SQRT	0 < N <= 16	5 Counts	RZ <- $\sqrt{RX}$

(1) Note -Precision refers to the number of digits to which the calculation is to be carried out.

(2) Note -Error refers to the error in the least significant digit of the mantissa and so is the relative error.

(3) Note -In the case of ADD, at return time RX contains the argument with the largest absolute value. Thus one may use this routine to sort a table of numbers. In the case of SUB the sign of the mantissa of RY is changed and then the arguments are added. Thus the arguments are intact except for a sign.

Transcendental Routines

Subroutines	Interval	Precision	Error	Symbolic Action
LOG	$\{1/\sqrt{10}, \sqrt{10}\}$	14	<10.E-8*	RZ <- LOG10(RX)
TENX	{0, 1}	12	10.E-8	RZ <- 10^(RX)
TANX	{0, 1}	14	10.E-8	RZ <- TAN(RX)
ATANX	{0, 1}	14	10.E-8	RZ <- ARCTAN(RX)

*Note: The error in the case of LOG is the absolute error; the others are relative error.

The user may use the bytes PREC and EXTRA together with the subroutine IPREC to calculate N, which is the value used by these routines. Note that the Transcendental Routines supply their own value for N and will overwrite any value supplied by the user.

Now that we know the names of the calculation routines and how to indicate to KIMath the desired precision of calculation we can add some statements to our sample program. Now it will look like (the > sign indicates new statements we have added):

```
        ;   (1) define storage
        NDIG = 8
           * = $5000
        L  * = *+NDIG+3
        C  * = *+NDIG+3
TWOPI     .BYTE $00, $62, $83, $18, $54, $F0
ONE       .BYTE $00, $10, $F0
        ; (2) define precision
> EX = 2; use 2 extra digits in calculations
> LDA    #NDIG
> STA    PREC
> LDA    #EX
> STA    EXTRA
> JSR    IPREC ; calculate N
        ; (3)  read L and transfer to RX
```

```
        ; (4) read C and transfer to RY
        ; (5) compute L times C - answer stored in RE
> JSR MLTPLY ; do the multiplication
        ; (6) move RZ to RX and compute square root - adj. exp.
> JSR SQRT
        ; (7) get 2pi to RX
        ; (8) move RZ to RY and multiply
> JSR MLTPLY
        ; (9) get 1 (constant) and put in RX
        ; (10) move RZ to RY and divide
> JSR DIVIDE
        ; (11) move RZ to L and print out
  .END
```

Now by simply changing the assigned value for NDIG and EX and re-assembling the program, both calculated precision and storage allocation can be automatically changed. This allows re-assembly of the program with changed precision (and thus changed locations for registers L and C). The pointers will automatically be changed to the correct locations.

5.1.2 The Conversion Routines permit the user to transform data from the packed BCD, constant, or unpacked formats to the computational format and back again. They are PLOADX, PLOADY, ULOADX, ULOADY, PSTRES, and USTRES. These routines all use PREC to determine the number of bytes to be converted.

a) PLOADX, PLOADY: These routines load RX and RY with the packed format data found at the addresses {ARGYL, ARGXH} and {ARGYL, ARGYH}. The data is unpacked into the computational format and the value in PREC determines how many digits will be transferred. Note that PREC should be an even number to prevent the loss of a digit in transfer. This is natural when it is noted that the packed format requires an even number of digits in the mantissa bytes.

b) ULOADX, UMADY: These routines load RX and RY with the unpacked format data found at the addresses {ARGXL, ARGYL} and {ARGYL, ARGYH} respectively. The data is converted into computational format and the number of digits transferred is determined by PREC. In this case PREC need not be even to effect an accurate transfer.

The use of ULOAD and PLOAD allows us to move data to the computational registers from memory registers. We can now add them to our program at the following locations:

```
                ; (3) Read L and transfer to RX
> JSR GETNL     ; a user-provided subroutine to get a
>               ; number from the terminal and place
>               ; it in L in unpacked format.
> LDA #<L       ; low-order byte of address of L
> STA ARGXL
> LDA #>L       ; high-order byte
> STA ARGXH
> JSR ULOADX    ; move L to RX
                ; (4) read C and transfer to RY
> LDA #<C       ; low-order byte of address of C
> STA ARGYL
> LDA #>C       ; high-order byte of address of L
> STA ARGYH
> JSR GETNC     ; user-supplied routine to read value of C into
                ; register C

> JSR ULOADY

Note: The assembler interprets the symbols "#<L" to mean the
      low-order byte of the address of the variable named L.
      "#>L" implies the high-order byte of the address of L.
```

c) PSTRES: This routine moves the contents of RZ into the location specified by {RESL, RESH} as a starting address and converts it to packed format. The byte PREC is used to determine the number of bytes to be converted.

d) USTRES: This routine converts the contents of RZ from computational format as it moves them to the destination specified by (RESL, RESH} and converts the contents to unpacked format. PREC, again, determines the number of bytes to be moved.

These routines allow transfer of the final answer to our example problem back to register L for printout:

```
              ; (11) move RS3 to L and printout
> LDA #<L     ; set up
> STA RESL    ; pointer
> LDA #>L     ; to locate
> STA RESH    ; destination register
> JSR USTRES  ; move it
> JSR PRINTL  ; user-supplied routine to print the value in L
```

e) USRLKP: This routine loads RY with the constant format data beginning at the address {KONL, KONH}. The data is converted from constant format to computational format. The number of digits transferred is determined by the location of the byte containing an 'F.' This routine also uses the byte named NKON to point at the constant. Thus if the user sets NKON equal to zero and the pair {KONL, KONH} equal to some address prior to calling USRLKP then at return time RY contains the constant. {KONL, KONH, and NKON} point at the next constant, that is, {KONL, KONH} + NKON is the address of the byte following the byte containing an F (presumably the first byte of the next constant). Using this routine and POLY one may step through a table of coefficients and evaluate a polynomial. Use of POLY is covered in Appendix A. Clearly, one may also use this routine for other purposes such as a simple table lookup of constants needed frequently in a calculation. Now we have the capability to load our constants into the program. Now we can fill out steps (7) and (9):

```
              ; (7) get 2pi to RX
> LDA #<TWOPI ; get low-order
> STA KONL    ; pointer byte
> LDA #>TWOPI ; get high-order
> STA KONH    ; pointer byte
> LDA #00
> STA NKON    ; initialize offset
> JSR USRLKP  ; lookup constant and put in RY
              ; still have to get RY to RX
```

```
                ; (9) get 1 (constant) and put in RX
> JSR USRLKP    ; a lot easier the second time
>               ; still must transfer RX from RY
```

Up to 256 bytes of constants may be indexed through by successive calls to USRLKP.

f) DECHEX: This routine converts the contents of the byte CNT from BCD to HEX. The converted value may be found at return time in CNT.

5.1.3 The User Utilities permit the user to move and clear various registers.

a) The routines whose names are of the form MV s, d, where "s" stands for source and "d" destination, allow the user to move the registers RX, RY, RZ, RM and RN about with ease. The following table shows the supported moves. RM and RN are working table registers and will not normally be available to the user.

Source	Destinations	Corresponding Routine Names
RX	RY, RZ, RM, RN	MVXY, MVXZ, MVXM, MVXN
RY	RX, RZ, RM, RN	MVYX, MVYZ, MVYM, MVYN
RZ	RX, RY, RM, RN	MVZX, MVZY, MVZM, MVZN
RM	RX, RY, RZ, RN	MVMX, MVMY, MVMZ, MVMN
RN	RX, RY, RZ, RM	MVNX, MVNY, MVNZ, MVNM

b) The routines CLRX, CLRY and CLRZ may be used to set RX, RY and RZ equal to zero, respectively.

c) The routine XSY exchanges RX and RY.

The move subroutines allow us to transfer computational data between working registers. We need them in steps (6), (7), (8), (9), and (10):

```
                ; (6) Move RZ to RX and compute square root
                ; - adj. exp.
> JSR MVZX      ; move RZ to RX
  JSR SQRT
                ; (7) get 2pi to RX
  LDA #<TWOPI
```

```
          STA KONL
          LDA #>TWOPI
          STA KONH
          LDA #00
          STA NKON
          JSR USRLKP  ; constant to RY
        > JSR MVYX    ; move it to RX
                      ; (8) move RZ to RY and multiply
        > JSR MVZY
          JSR MLTPLY
                      ; (9) get 1 (constant) and put in RX
          JSR USRLKP  ; get next constant
        > JSR MVYX    ; move it to RX
                      ; (10) move RS to RY and divide
        > JSR MVZY    ; move RZ to RY
          JSR DIVIDE
```

The sample program is now complete except for the fact that SQRT will only operate on numbers between 1 and 100, so we must-adjust the exponent of the argument, take the square root and adjust the results. The two subroutines required are listed in Appendix B and are called SQRIN & SQROUT. They must be added to the program (they are not included in KIMath). Step (6) now becomes:

```
                      ; (6) Move RS to RX and compute square root
          JSR MVZX    ; move RZ to RX
        > JSR SQRIN   ; adjust exponent
          JSR SQRT    ; compute root
        > JSR SQROUT  ; adjust exponent back
```

5.2 Completed KIMath Program

```
                      ; (1) define storage
          NDIG = 8
             * = $5000
          L  * =  *+NDIG+3
          C  * =  *+NDIG+3
          TWOPI     .BYTE $00, $62, $83, $18, $54, $F0; TWOPI
          ONE       .BYTE $00, $10, $F0
                      ; (2) define precision
```

```
EX = 2      ; use 2 extra digits in calculation
LDA #NDIG
STA PREC
LDA #EX
STA EXTRA
JSR PREC    ; calculate N
            ; (3) Read L and transfer to RX
JSR GETNL   ; user-provided subroutine to input value for L
            ; to L register in unpacked format.
LDA #>L     ; low-order byte of address of L
STA ARGXL
LDA #<L     ; high-order byte
STA ARGXH
JSR ULOADX  ; Move L to RX
            ; (4) Read C and transfer to RY
LDA #<C     ; low-order byte of address of C
STA ARGYL
LDA #>C     ; high-order byte of address of C
STA ARGYH
JSR GETNC   ; user-provided subroutine to input value
            ; for C to C register in unpacked format.
JSR ULOADY  ; transfer C to RY
            ; (5) Compute L times C
JSR MLTPLY
            ; (6) Move RZ to RX and compute square root
JSR MVZX    ; move RZ to RX
JSR SQRIN   ; adjust exponent
JSR SQRT    ; compute root
JSR SQROUT  ; adjust exponent back
            ; (7) get 2pi to RX
LDA #<TWOPI ; address low of First constant
STA KONL
LDA #>TWOPI
STA KONH
LDA #00
```

```
        STA NKON
        JSR USRLKP      ; constant to RY
        JSR MVYX        ; move it to RX
                        ; (8) Move RZ to RY and multiply
        JSR MVZY
        JSR MLTPLY
                        ; (9) get 1 (constant) and put in RX
        JSR USRLKP      ; get next constant
        JSR MVYX        ; move it to RX
                        ; (10) Move RZ to RY and divide
        JSR MVZY        ; move RZ to RY
        JSR DIVIDE      ;
                        ; (11) Move RZ to L and print it out
        LDA #<L         ; set up
        STA RES         ; pointer
        LDA #>L         ; to locate
        STA RES+1       ; destination register
        JSR USTRES      ; move it
        JSR PRINTL      ; user-supplied routine to print out L.
        .END
```

Note: The above program would also require that the starting address of all KIMath subroutines be defined. See Appendix D for the correct addresses.

Appendix A

EVALUATING POLYNOMIALS

KIMath uses polynomial approximations to generate several of its functions, and these subroutines are also available for calculation of user-defined polynomials.

The user must first define the constants to be used in the calculation, then define the degree of the polynomial and the starting address of the stored constant. Finally, the argument for the polynomial is transferred to RX and the routine POLY is called. The value of the polynomial is returned in RE.

Specifically, given a polynomial of the form:

y = c0 + c1*x + c2*x^2 + c3*x^3 + ... + cn*x^n

the coefficients (c0, c1, c2, c3, ..., cn) are stored in constant format in contiguous memory. The highest order coefficient is stored in the lowest memory address. NKON is set to zero, DEG is set to a value of N - 1 where N is the degree of the polynomial. The address pair (KONL, KONH) must point to the first byte of the constant storage area. RX is loaded with the value of x and the subroutine POLY is called. Upon return, y (the value of the polynomial) is in RZ.

A Sample Program

The sine function can be expressed as:

$$\sum_{n=1}^{\infty}(-1)^{n+1}\frac{Z^{2n-1}}{(2n-1)!}$$

Expanded, this yields the polynomial:

$$\sin Z = Z - \frac{Z^3}{3!} + \frac{Z^5}{5!} - \frac{Z^7}{7!} + \frac{Z^9}{9!} - \frac{Z^{11}}{11!} \ldots$$

For our example we will use the first six non-zero terms for our application.

Our coefficients are:

c0 = 0.0
c1 = 1.0
c2 = 0.0

$$c3 = -\frac{1}{3!} = -1.6666667\ E\text{-}1$$

$$c4 = 0$$

$$c5 = +\frac{1}{5!} = +8.3333333\ E\text{-}3$$

$$c6 = 0$$

$$c7 = -\frac{1}{7!} = -1.9841270\ E\text{-}4$$

$$c8 = 0$$

$$c9 = +\frac{1}{9!} = +2.7557319\ E\text{-}3$$

$$c10 = 0$$

$$c11 = +\frac{1}{11!} = -2.5052108\ E\text{-}3$$

A program to evaluate the sine function would look like:

```
                    ; define parameter and pointers
PREC  *  = $10
         .BYTE 04    value for PREC
EXTRA *  = $11
         .BYTE 04    value for EXTRA
KON   *  = $OE
         .WORD $3000
DEG   *  = $05
         .BYTE 10
NCON  *  = $01
         .BYTE $00
      *  = $3000 ; starting address
NIN      =  * + 7 ; reserve 7 bytes for input register


; define constants c0 - c11 - note that they are stored in
; reverse order

          .BYTE $CO,$25,$05,$21,$08,$F8 ; c11
          .BYTE 0,0,$F0 ; c10
          .BYTE $40,$27,$55,$73,$19,$F6 ; c9
          .BYTE 0,0,$F0 ; c8
          .BYTE $CO,$19,$84,$12,$70,$F4 ; c7
          .BYTE 0,0,$F0 ; c6
          .BYTE 04,$83,$33,$33,$33,$F3  ; c5
          .BYTE 0,0,$F0 ; c4
```

```
          .BYTE $C0,$16,$66,$66,$67,$F1 ; c3
          .BYTE 0,0,$F0   ; c2
          .BYTE 0,01,$F0  ; c1
CONST     .BYTE 0,0,$F0   ; c0

; call user-written   subroutine to load some
; value into NIN register.

 JSR GETVAL
 LDA KON          ; set up
 STA ARGXL        ; pointer to
 LDA KON+1        ; input
 STA ARGXH        ; buffer

 JSR ULOADX       ; move NIN to RX and convert
 JSR POLY         ; evaluate polynomial
 LDA KON          ; set up
 STA RES          ; pointer to
 LDA KON+1        ; output
 STA RES+1        ; results
 JSR USTRES       ; move RZ to NIN and convert

; call user-written subroutine to printout
; value in NIN

 JSR PUTVAL
 .END ; all done
```

Appendix B

APPLICATIONS

In this section we shall deal with extension of the range of the functions, and extension of the function set.

a) Common Logarithm: Let x * 10^r denote an arbitrary positive floating-point number. Then

$$x * 10^r = x * 10^{(-1/2)} * 10^{(1/2)} * 10^r$$

$$= (x * 10^{(-1/2)}) * 10^{(r+1/2)}$$

Now 1 <= x < 10 and so 10^(-1/2) <= x * 10^(-1/2) < 10^(1/2). But 0.1 = 10^(-1/2) and so LOG may be used to evaluate log10(x * 10^(-1/2)). By using the following equation, one may compute the common log for any positive argument.

$$\log_{10}(x * 10^r) = \log_{10}(x * 10^{(-1/2)} * 10^{(r+1/2)})$$

$$= \log_{10}(x * 10^{(-1/2)}) + r + 1/2$$

b) Common Antilog: Let x be an arbitrary floating-point number satisfying the inequality

$$|x| < 100,$$

and let I = [x] and F = <x>, where [x] stands for the largest integer less than or equal to x and <x> = x - [x]. With these definitions in mind we find that

$$x = I + F$$

Now 0 <= F < 1 and I is an integer and so

$$10^x = 10^F * 10^I$$

Since 0 <= F < 1 we may use TENX to find 10^F, after which we have a number between 1 and 10. Therefore I is the floating point exponent of 10^x.

c) Tangent: Let x be a floating-point number which satisfies the inequality 0 <= x <$\pi/2$. This implies that 0 <= x * $2/\pi$ < 1. Now tan(x * $\pi/4$ * $2/\pi$) = tan(x/2) but

$$\tan(x) = \frac{2\tan(x/2)}{1 - \tan^2(x/2)}$$

Thus by applying TANX to the value of $(2/\pi)$x and applying the above trigonometric identity one may extend the tangent function to the interval [0,$\pi/2$). Note also that

$$\cos(x) = \frac{1 - \tan^2(x/2)}{1 + \tan^2(x/2)} \quad \text{and}$$

$$\sin(x) = \frac{2\tan(x/2)}{1 + \tan^2(x/2)}$$

and so one may use TANX to evaluate the other trig functions.

d) Arc tangent: This function may be extended from the interval [0, 1] by means of the following identities:

$$\arctan(-x) = -\arctan(x)$$

$$\arctan(|x|) = \pi/2 - \arctan(1/|x|), \quad x \neq 0$$

With these identities and the ATANX subroutine one can evaluate the arctangent function for any argument. The other inverse trigonometric functions may then be computed by means of the following identities:

$$\arcsin(x) = \arctan\left(\frac{x}{(1-x^2)^{(1/2)}}\right)$$

$$\arccos(x) = \arctan\left(\frac{(1-x^2)^{(1/2)}}{x}\right)$$

e) Square Root: Since every floating-point number can be expressed as x * 10^r where 1 <= |x| < 100 and |r| is even or zero, one can form the square root as follows:

$$\sqrt{x * 10^r} = 10^{r/2} * \sqrt{x}$$

where x >= 0 and $\sqrt{x}$ is evaluated by using SQRT. Note that r/2 is an integer and 1 <= $\sqrt{x}$ < 10 and so r/2 is the floating-point exponent of sqrt(x * 10^r).

Chapter 5 mentions that the sample program must be provided with two subroutines to extend the range of the square root function. These subroutines are provided below. The subroutine SQRIN must be called immediately prior to calling SQRT and SQROUT must be called immediately afterwards. SQRIN and SQROUT are not part of KIMath and must be supplied by the user:

Exponent Adjustment for SQRT

```
            VAL   * = *+1
SQRIN       SED
SQ1         SEC
            LDA   EX
            SBC   #2
            STA   EX
            BCC   OUT
            CLC
            LDA   VAL
            ADC   #1
            STA   VAL
            BNE   SQ1
OUT         ADC   #2
            BIT   sx
            BVC   SQ2
            STA   EX
            CLC
            ADC   VAL
            STA   VAL
            RTS
```

```
SQ2       STA   EX
          RTS


SQROUT    LDA   VAL
          STA   EZ
          RTS
```

Appendix C

THE APPROXIMATIONS

This section is included for those users who are interested in how KIMath generates its functions. Understanding of this section is not required for use of KIMath.

1) LOG: This subroutine has been built around the rational function given by the following equation:

$$R(X) = t(x) * p(t(X)),$$

where $t(x) = \dfrac{x-1}{x+1}$, $x \in [\sqrt{0.1}, \sqrt{10}]$ and p(x) is a polynomial given by

p(x) = a0 + al*x + a2*x^2 + a3*x^3 + a4*x^4 + a5*x^5

where

a0 = 8.685887483405 x 10^(-1)
a1 = 2.89551130267 x 10^(-1)
a2 = 1.731095517 x 10^(-1)
a3 = 1.3136901121 x 10^(-1)
a4 = 5.53427387 x 10^(-2)
a5 = 1.820912997 x 10^(-1)

The absolute error on the interval $[\sqrt{0.1}, \sqrt{10}]$ for log10(x) as approximated by R(x) is less than 1 x 10^(-8).

2) TENX: This subroutine has been built around the polynomial given by the following equation:

P(x) = (a0 + a1*x + a2*x^2 + a3*x^3 + a4*x^4 + a5*x^5 + a6*x^6 + a7*x^7)^2

where x $\in$ [0, 1] and

a0 = 1
a1 = 1.15129277603
a2 = 6.6273088429 x 10^(-1)
a3 = 2.5439357484 x 10^(-1)

a4 = 7.295173666 x 10^(-2)
a5 = 1.742111988 x 10^(-2)
a6 = 2.55491796 x 10^(-3)
a7 = 9.3264267 x 10^(-4)

The relative error on the interval [0, 11 for 10^x as approximated by P(x) is less than 5 x 10^(-9).

3) TANX: This subroutine has been built around the polynomial given by

P(x) = x(a0 + a1(x^2) + a2(x^2)^2 + a3(x^2)^3 + a4(x^2)^4 +
a5(x^2)^5 + a6(x^2)^6),

where

a0 = 7.853981762291 x 10^(-1)
al = 1.614897776174 x 10^(-1)
a2 = 3.98659104705 x 10^(-2)
a3 = 9.8345945393 x 10^(-3)
a4 = 2.7974335037 x 10^(-3)
a5 = 2.031171084 x 10^(-4)
a6 = 4.109741948 x 10^(-4)

The relative error on the interval [0, 1] for tan($4/\pi$ * x) as approximated by P(x) is less than 1 x 10^(-8).

4) ATANX: This subroutine has been built around the polynomial given by

P(x) = x(a0 + a1(x^2) + a2(x^2)^2 + a3(x^2)^3 + a4(x^2)^4 +
a5(x^2)^5 + a6(x^2)^6 + a7(x^2)^7 + a8(x^2)^8),

where

a0 = 9.999999847657 x 10^(-1)
al = -3.333307334505 x 10^(-1)
a2 = 1.999261939166 x 10^(-1)
a3 = -1.420364446652 x 10^(-1)
a4 = 1.06409340253 x 10^(-1)
a5 = -7.50429453889 x 10^(-2)
a6 = 4.26915192711 x 10^(-2)
a7 = -1.60686289604 x 10^(-2)
a8 = 2.8498896208 x 10^(-3)

The relative error on the interval [0, 1] for arctan(x) as approximated by P(x) is less than 1 x 10^(-8).

APPENDIX D

KIMath Addresses

CALCULATION SUBROUTINES

ADDRESS	NAME
F808	ADD
F800	SUB
F90B	MLTPLY
FA16	DIVIDE
FA9E	SQRT
FAE7	LOG
FB41	TENX
FB5C	TANX
FB78	ATANX

UTILITY ROUTINES

FEF0	SAVXY
FEF5	RCLXY
FEE8	IPREC
FE0A	PLOADX
FE23	PLOADY
FE8A	ULOADX
FEA2	ULOADY
FE3C	PSTRES
FEBA	USTRES
FD92	USRLKP
FDC1	POLY
FD71	CLRX
FD7C	CLRY
FD87	CLRZ
FBC3	DECHEX
FCBF	XSY

WORKING STORAGE

PAGE ZERO

STARTING ADDRESS	NAME
10	PREC
11	EXTRA
06	ARGXL
07	ARGXH
08	ARGYL
09	ARGYH
0A	RESL
0B	RESH
0E	KONL
0F	KONH
01	NKON
05	DEG
03	CNT

COMPUTATION REGISTERS

STARTING ADDRESS	NAME
0235	RX
0235	SX (sign)
0246	EX (exponent)
0247	RY
0247	SY
0258	EY
0259	RZ
0259	SZ
026A	EZ