FLOATING POINT SYSTEMS, INC.

AP Math Library Manual, Volume 1

860-7288-004

by FPS Technical Publications Staff

AP Math Library Manual, Volume 1

860-7288-004

Publication No. FPS 860-7288-004
February, 1979

NOTICE

The material in this manual is for information purposes only and is subject to change without notice.

Floating Point Systems, Inc. assumes no responsibility for any errors which may appear in this publication.

Printed in USA

CONTENTS

VOLUME 1

ILLUSTRATIONS

TABLES

CHAPTER 1

INTRODUCTION

1.1 PURPOSE

The purpose of this manual is to provide the information necessary to understand and use the Array Processor (AP) Math Library. The Math Library contains a versatile set of FORTRAN callable routines for use in high-speed array processing. Once these routines are installed in the host system, they can be called by standard FORTRAN programs.

1.2 SCOPE

This manual is a user document designed to describe the Math Library routines and acquaint the user with the unique features of the AP. The manual is divided into two parts.

Part One consists of five chapters and four appendices. The five chapters provide general information about the AP and the use of the Math Library. Chapter 1 presents introductory material, including basic concepts about AP processing and Math Library use. Chapter 2 provides general operating information necessary for the most efficient use of the Math Library routines. It includes information about memory organization, format conversion and speed considerations. It also defines a general procedure for program development. Chapter 3 defines the categories into which the Math Library routines are organized. Chapter 4 presents a number of detailed examples of array processing programs written with routines from the Math Library. Chapter 5 describes the FORTRAN Math Library Simulator (MATHSIM) and its use.

The four appendices are designed to provide quick and easy access to more detailed information about any one of the more than 150 Math Library routines. This information includes what each routine does, how to use it, and how fast it runs. Appendix A lists the Math Library routines alphabetically. Appendix B lists the routines by type and page order. Appendix C gives an abbreviated summary of each routine and defines its purpose and its calling parameters. Appendix D lists the routines available for use in AP-FORTRAN program units and their AP-FORTRAN calling names.

Part Two consists of four appendices. Appendix E provides complete reference material about each routine. Appendices F, G, and H are actually identical to Appendices A, B, and D, respectively; they are repeated in Part Two for easy reference.

If more information is desired on the AP, the reader should refer to the manuals listed in Table 1-1.

Table 1-1 Related Manuals

MANUAL	PUBLICATION NO.
Processor Handbook	FPS 860-7259-003
Programmer's Reference Manual Parts One and Two	FPS 860-7319-000
FORTRAN Reference Manual	FPS 860-7408-000
APAL Reference Manual	FPS 860-7412-000
APLOAD Reference Manual	FPS 860-7410-000
APDBUG/APSIM Reference Manual	FPS 860-7364-002
APEX Manual	FPS 860-7371-001
AP Diagnostic Software Manual	FPS 860-7284-002

0849

1.3 AP HARDWARE

This section is included to give the user a general overall picture of the structure of the AP and some insights into why it can process arrays at such high speeds. It is not, however, necessary to know this information in order to write programs with the Math Library.

1.3.1 BASIC ARCHITECTURE

The AP uses a general-purpose, multi-bus oriented architecture. The floating adder and floating multiplier are each connected directly to each of the memory elements and registers in the AP through separate parallel 38-bit data paths. This parallel structure allows the overhead of array indexing, loop counting, and data fetching from memory to be performed simultaneously with the arithmetic operations on the data. Much faster program execution is possible as opposed to using a typical general-purpose computer where each of the above operations must occur sequentially.

Specifically:

- Programs, constants and data each reside in separate, independent memories to eliminate memory accessing conflicts.

- Independent floating-point multiplier and adder units allow both arithmetic operations to be initiated every 167ns.

- Two large blocks (32 locations each) of floating-point accumulators are available for temporary storage of intermediate results from the multiplier, adder or memory.

- Address indexing and counting functions are performed by an independent integer arithmetic unit that includes 16 integer accumulators.

In a typical application, such as a Fast Fourier Transform (FFT), the above features allow nearly the entire computation to be overlapped with data memory access time.

Effective processing precision is enhanced by 38 bits of internal data width, an internal floating-point format with optimum numerical properties, and a convergent rounding algorithm.

1.3.2 SYSTEM OVERVIEW

The AP is connected to the host in a manner that permits data transfers to occur under control of either the host computer or the AP (refer to Figure 1-1). For most host computers, this means that the AP is interfaced to both the programmed I/O and DMA channels.

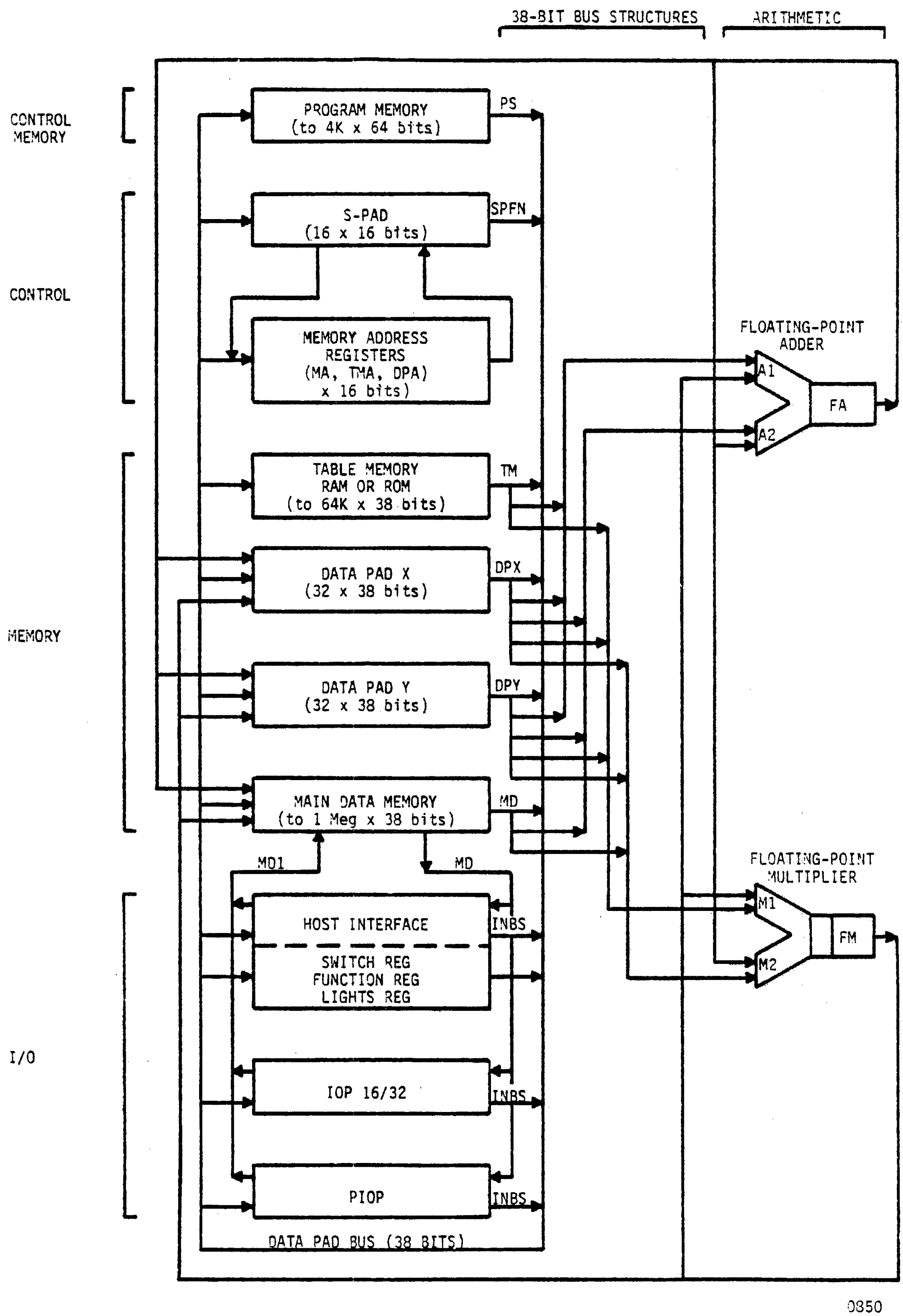

Figure 1-1 AP Block Diagram

The system elements are interconnected with multiple parallel paths so that transfers can occur in parallel. All internal floating-point data paths are 38 bits in width (10-bit biased binary exponent and 28-bit 2's complement mantissa). The main data memory (MD) is organized in 8K and 32K-word modules of 38-bit words each, expandable up to 512K words in the main chassis. Effective memory cycle times (interleaved) of either 167ns or 333ns are available.

The table memory (TM) is used for storage of constants and is tied to a separate data path so as not to interfere with data memory. It is bipolar, 167ns read-only memory, and is organized in 512-word, 38-bit increments. An optional random access memory (TMRAM) is also available.

The program source memory (PS) can hold from 512 to 4096 64-bit instruction words.

Data pad X (DPX) and data pad Y (DPY) are two blocks of 32 floating accumulators each. Each is a two-port register block wherein one register may be read, and another written from each block in one instruction cycle.

The floating adder (FA) consists of two input registers, A1 and A2, and a two-stage pipeline which performs the operations and convergently rounds the normalized result.

The floating multiplier (FM) consists of two input registers, M1 and M2, and a three-stage pipeline which performs the multiply operation. Products are normalized and convergently rounded 38-bit numbers.

The s-pad consists of sixteen 16-bit integer registers and an integer arithmetic unit which is used to form operand addresses and to perform integer arithmetic.

1.3.3 EXAMPLE OF AP OPERATION

The following example shows the sequence the AP goes through to add two vectors.

The initial conditions for this sequence are that the program to add two vectors resides in the AP program source memory and the two vectors to be added reside in the host memory.

1. The host calls the AP executive program (APEX) to request host DMA cycles to transfer the two vectors from the host memory to the AP main data memory. The two vectors are converted from host floating-point format to the AP floating-point format on the fly as they pass through the formatting hardware of the interface.

2. The host calls APEX to start the AP vector add routine. The routine is performed with the resultant vector remaining in the AP format. This format yields the benefit of 38-bit precision and convergent rounding during the critical phases of processing.

3. The host calls APEX to request host DMA cycles to transfer the resultant vector back to the host memory. The vector is converted from AP format to host floating-point format, again on the fly.

4. The AP proceeds to another process or stops executing, depending on previously established conditions. An interrupt to the host can be issued.

A detailed discussion of this example is given in section 2.3. It is given from a programming viewpoint and includes a commented FORTRAN program.

1.3.4 FURTHER HARDWARE CONSIDERATIONS

The AP is most efficient when a sequence of operations can be performed on one or more vectors, or on a whole array which resides in the main data memory. This approach reduces data-transfer overhead and retains maximum numerical precision. A reasonable sequence, for example, would be to transfer a trace and a filter, FFT both, array multiply, inverse FFT, and transfer the result back to the host memory.

The AP main data memory has DMA capability. This means that the interface can steal main data memory cycles from the AP microprocessor. This capability allows the host computer DMA-to-AP DMA data transfers to occur, thereby minimizing both host and AP overhead.

The AP has been designed with enough built-in flexibility to allow its power to be harnessed in a variety of ways. Refer to the AP Processor Handbook (FPS 860-7259-003) for detailed descriptions of the elements of the AP presented in this discussion.

1.4 AP SOFTWARE

Four software packages are supplied with the AP to assist the user in running programs, writing programs, and diagnosing hardware faults.

1.4.1 THE EXECUTIVE

The AP executive (APEX) allows the user to communicate with the AP via FORTRAN or host assembly language calls. It is a subroutine linked into FORTRAN programs which use the array processor. The APEX driver subroutine interprets the particular user call and directs the AP to perform the specified action. Both the AP Math Library routines and user-developed AP programs may be called from the host computer using APEX.

1.4.2 THE AP MATH LIBRARY

The AP Math Library (APMATH) includes over 235 floating-point routines which cover a wide range of array processing needs. These routines, written in AP assembly language, can be called by programs written either in host FORTRAN, host assembly language, or in AP assembly language. The purpose of this manual is to describe these routines as follows:

- data transfer and control operations
- basic vector arithmetic
- vector-to-scalar operations
- vector comparison operations
- complex vector arithmetic
- data formatting operations
- matrix operations
- FFT operations
- auxiliary operations
- APAL callable utility operations
- signal processing operations
- table memory operations

1.4.3 PROGRAM DEVELOPMENT PACKAGE

This package provides four FORTRAN IV programs which are compiled on the host computer during installation, and are for use in writing array processing programs and subroutines in the AP assembly language. The programs are as follows:

APAL	AP assembler is a cross-assembler that provides a two-pass assembly of AP symbolic assembly language coding into an object module. APAL generates detailed error diagnostics.
APLOAD	APLOAD links and relocates separate APAL and AP-FORTRAN object modules together into a a single load module.
APSIM	AP simulator (APSIM) provides a programmed simulation of the various hardware elements of the AP. All timing characteristics of the AP are emulated, and the floating-point arithmetic is simulated (including rounding) to the least significant bit. APSIM is a convenient tool in bringing up new AP programs off-line without interfacing with production runs.
APDBUG	APDBUG is an interactive debugging program with commands similar to APSIM. The user may selectively set breakpoints, examine and change memory and register contents, and run program segments.

The AP Programmer's Reference Manual (FPS 860-7319-000) is a comprehensive instruction manual which describes developing programs using the AP Program Development Package.

1.4.4 DIAGNOSTIC PACKAGE

The AP test programs are a collection of interactive diagnostic test and verify programs that aid in isolation of hardware faults. They include:

APTEST	AP test exercises the panel, DMA interface, and various internal registers and memories. It tests main data memory with simple patterns and then with random numbers. Board level diagostic indicators are provided.
APPATH	AP path test tests the various internal data paths and gives board-level diagnostics.
APARTH	AP arithmetic test tests the floating-point adder, multiplier, and s-pad arithmetic unit with pseudo-random number and operation sequences.
FIFFT	Forward/Inverse FFT test verifies the correct operation of the AP as a complete unit by doing forward/inverse FFT transforms on both spikes and random number sequences.

CHAPTER 2

GENERAL OPERATION

2.1 INTRODUCTION

This section gives the basic information required to use the AP Math Library routines with host FORTRAN programs in order to process data with the AP. Miscellaneous information about the structure and operation of the AP is also included to help the user get the most efficient use of the AP.

2.2 ARRAYS, VECTORS AND SCALARS

The terms array and vector are used somewhat interchangeably when discussing array processing. There is, however, a difference between an array and a vector.

An array is a group of numbers that are related to each other in some way. An array of numbers often has a multi-dimensional aspect to it. A matrix, for example, is an array. Another kind of an array is a table of numbers, such as a table of several parameters -- all related to one system or measurement.

A vector in array processing terminology refers to a one-dimensional sequence (string) of numbers. The columns of a matrix or table are vectors. In this sense, a vector is essentially a subset of an array, i.e., a string of numbers that are all values for the same parameter. When organizing an array for processing, the user usually divides the array into vectors and establishes one vector for each column of data.

Array processing often involves performing a relatively simple operation or algorithm repetitively on long sequences of data (vectors). The strength of the AP is that it is designed to perform such operations at much faster speeds than is possible with a general purpose processor.

The individual numbers in an array or vector are called elements. A vector of only one element is a scalar. Thus, a scalar refers to a single number. A vector operation may also involve a scalar (e.g., the dot product of two vectors, or the product of each element of a vector by a constant).

To summarize then:

- An array is a group of numbers.
- A vector is a sequence of numbers.
- A scalar is a single number.

2.3 PROGRAM FLOW

Writing a FORTRAN program that calls on the AP to process data is basically the same as writing a FORTRAN program that runs exclusively on the host processor. Exceptions to this are as follows:

- The AP and APEX must be initialized before any other calls are made to the AP.
- Data must be transferred from the host memory to the AP main data memory before the AP can operate on it.
- In order to synchronize the operation of the AP with the host, wait calls must be inserted in the program whenever the host and the AP interact.
- At the end of program execution, data must be transferred from the AP main data memory back to the host memory.

Figure 2-1 illustrates the necessary steps to follow when writing a FORTRAN program to run on the AP. The following discussion addresses each of these blocks separately. Figure 2-2 illustrates a FORTRAN program that directs the AP to add two vectors together. The sequence of hardware operations for this procedure is given in section 1.3.3. The program in Figure 2-2 is referred to throughout the following sections.

2.3.1 DIMENSION DATA IN HOST MEMORY

Before an array can be transferred to the AP, it must be dimensioned and stored in the host memory. This is the first step in the example in Figure 2-2:

```
DIMENSION A(1000), B(1000), C(1000)
```

At this point, the user can create vectors to be processed by the AP. The DIMENSION command tells the host how many memory words to allocate for each vector, and gives each vector a name. The user can then use these names to call the data for transfer to the AP. Note that a vector C is also created in this example to provide a location in the host memory where the sum of the addition of the two vectors A and B can be stored. If it is not necessary to preserve a copy of A or B in the host, then the result can be stored back into A or B, thereby avoiding the additional host memory requirement.

An alternate method of dimensioning the arrays is to combine both the A and the B vector into one 2000-word vector: DIMENSION A(2000), C(1000). This eliminates one of the data transfer calls required to transfer the two vectors to the AP, and reduces the program run time. However, it is a little more complicated for the user to keep track of the various vectors in the array. Dimensioning of data is described further in the following sections.

2.3.2 STORING THE ARRAY IN THE HOST MEMORY

With the array location established in the host memory, the user must fill the memory locations with actual data. This means reading in the data from a tape drive, an analog-to-digital converter, a disk drive, etcetera. Figure 2-1 illustrates a general flowchart for writing a FORTRAN calling program to perform an operation with the AP.

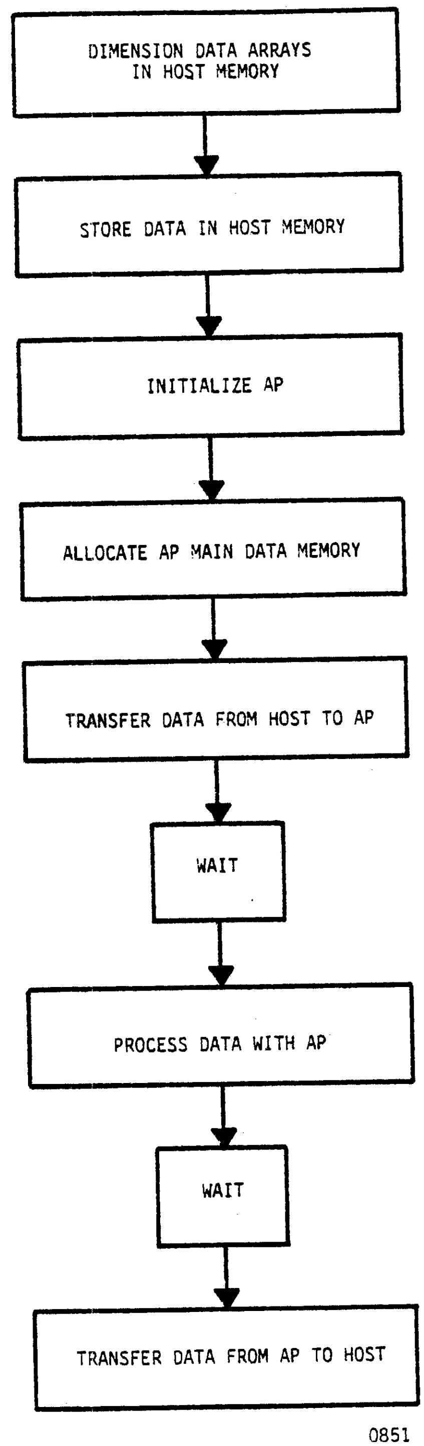

Figure 2-1 FORTRAN Calling Program Flowchart

In the following example, the vectors are created with an arithmetic expression in a DO loop:

```
C------FORTRAN program to add 2 vectors in AP120B and return result to
C      host
C
C------Dimension vectors in host
C
      DIMENSION A(1000),B(1000),C(1000)
C
C------Select size of vectors to be added
C
      N=1000
C
C------Somehow create vectors A and B in host
C
      DO 10 I=1,N
      A(I)=........
   10 B(I)=........
C
C------Initialize AP120B (must be done before any other
C      calls to AP120B)
C
      CALL APCLR
C------Indicate we're transferring host floating-point numbers
      IFMT=2
C
C------Allocate AP120B main data memory
C
      IA=0
      IB=N
      IC=N+N
```

```
C
C------Transfer A and B from host to AP120B main data memory
C------A is transferred to locations 0 - 999, B to
C       locations 1000 - 1999
      CALL APPUT(A,IA,N,IFMT)
      CALL APPUT(B,IB,N,IFMT)
C------Wait until transfer is complete before doing computations
C       on data
      CALL APWD
C
C------Perform vector addition in AP120B, storing results
C       2000 - 2999
C
      CALL VADD(IA,1,IB,1,IC,1,N)
C
C------Wait until calculation is finished before getting results
      CALL APWR
C------Now transfer result from locations 2000 - 2999 to host buffer C
      CALL APGET(C,IC,N,IFMT)
C------Wait until transfer is complete before printing results, etc
C       in host
      CALL APWD
C
C------Print results, etc. in host
C
       |
       |
       |
      END
```

The routines in the AP Math Library operate on four different types of vectors or arrays: real vectors, complex vectors, complex FFT vectors and matrix arrays. In each of these cases, the routines assume that the vector or array is organized in a particular sequence. For example, each element of a complex vector requires two memory words: one word for the real part of the element and one for the imaginary part. The routines for operating on complex vectors assume that the parts of each complex element are stored in two consecutive addresses in the AP main data memory.

The initial organization of arrays and vectors should be done when dimensioning the host memory and storing data in the host. Refer to the discussion of vector organization (section 2.4) for more details on the vector formats and variation allowed when organizing vectors for processing with the AP Math Library routines.

2.3.3 INITIALIZING THE AP

Initially, the AP internal status register and DMA control register must be cleared, and the AP executive (APEX) must be initialized. This is done with:

```
CALL APCLR
```

APCLR must be called before any other calls are made to the AP.

2.3.4 ALLOCATING THE AP MAIN DATA MEMORY

The main data memory in the AP is organized into 38-bit floating-point words. The words are consecutively numbered from 0 to N-1, where N is the maximum size of the memory: 8192, 16384, etcetera.

In complex programs where a number of transfers of data between the AP and the host are required, and where the arrays being operated on are large or numerous, it is recommended that the user take some care in allocating the AP memory before proceeding with the program.

Dimensioning the AP is very simple. The user must establish where each vector is to reside in memory and establish an integer constant, variable name, or expression that specifies the base address (first word) of each vector location.

For the example in Figure 2-2, memory allocation is done with the following FORTRAN statements:

```
IA = 0
IB = N
IC = N + N
```

Vector A is defined as starting at word 0 and going to word 999 (N=1000); vector B goes from 1000 to 1999; and the result, vector C, is stored from 2000 to 2999. The I that precedes each variable indicates that the addresses specified are integer values (standard FORTRAN convention).

Section 2.3.1 suggests arranging the array in the host memory into one long vector as a means of reducing program run time. The dimensioning of the array into vectors can then be done in the AP with the type of memory allocation statements shown previously.

There is one other consideration in allocating space in the AP main data memory. Many of the AP Math Library routines run at different speeds depending on the location of the vectors to be operated on in the AP main data memory. Program run time can occasionally be reduced by specifying that certain vectors start on either even or odd memory addresses. (Refer to section 2.7.2 for further information on memory allocation.)

2.3.5 TRANSFERRING DATA FROM THE HOST TO THE AP

With these preliminary steps completed, the user can transfer the array to be processed from the host memory to the AP main data memory with an APPUT command. APPUT has four parameters:

```
CALL APPUT (HOST, AP, N, TYPE)
```

HOST specifies the initial element of the data in the host that is to be moved to the AP. HOST can be a constant, a variable, an array name or an array element. Typically, the HOST parameter consists of the name of the first array element to be transferred; for example: A, SIGA(50), MATB (101). Illustrated in Figure 2-2, the HOST parameters in the two APPUT calls are A and B:

```
CALL APPUT (A, _, _, _)
CALL APPUT (B, _, _, _)
```

The parameter AP specifies the base address in the AP main data memory where the data from the host memory is to be stored. AP can be an integer, constant, variable, or an expression that specifies an integer number; for example: 101, IA, IA + 3*N. In the previous step, the AP parameters are generally specified when allocating the AP memory. As illustrated in the two APPUT commands in Figure 2-2, the variables IA and IB are used for the AP parameters.

```
CALL APPUT (A, IA, _, _)
CALL APPUT (B, IB, _, _)
```

It is possible to omit the AP memory dimensioning step and merely use integer constants for AP. For example:

```
CALL APPUT (A, 0, _, _)
CALL APPUT (B, 1000, _, _)
```

But specifically allocating the AP main data memory at the beginning of a FORTRAN program is good programming practice, especially when the program has many vector operations.

N specifies the number of host data elements to be moved from the host to the AP. Note that a data element may consist of more than one host word. For example, a host floating-point number usually requires two host words, but occupies one word in the AP main data memory. Like the AP parameter, N can be an integer constant, variable, or an expression that specifies an integer number. Earlier in the example program, N was specified as being 1000. So, in this example, the variable N is used for the N parameter.

```
CALL APPUT (A, IA, N, _)
```

The number 1000 could also have been used for N.

CALL APPUT (A, IA, 1000, _)

TYPE specifies the host data format and the type of conversion to be done between the host and AP during transfer. Format conversion of floating-point numbers is done automatically, on the fly, as part of the data transfer procedure. No conversion call is required for host floating-point numbers other than to specify the format with the TYPE parameter (2 or 3).

The AP performs arithmetic using a 38-bit floating-point format (illustrated in Figure 2-3): one exponent sign bit, nine exponent bits, one mantissa sign bit, and 27 mantissa bits. The binary point is always located between the mantissa sign bit and the most significant bit of the mantissa. (Bits 0, 1 and 40 are parity bits.)

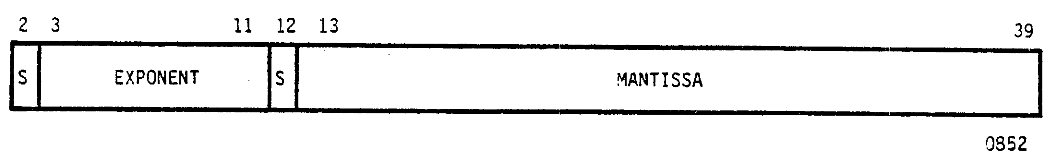

Figure 2-2 AP Floating-point Format

TYPE can specify four different kinds of formats and format conversions, depending on whether TYPE = 0, 1, 2 or 3.

When TYPE is 0, 32-bit integers are transferred from the host to the AP and stored without format conversion into the low 32 bits of the AP memory words (8 through 39). Refer to Chapter 3, Data Formatting Commands, for information on using the TYPE 0 and 1 formats.

When TYPE is 1, 16-bit integers are converted into unnormalized AP floating-point numbers. These numbers must be normalized (floated) before they can be processed using an AP Math Library routine. VFLT is the normalizing command.

Normalization of a floating-point number means the number is adjusted so that the most significant bit of the mantissa is located in bit 13 of the 38-bit word. There is a corresponding adjustment of the exponent.

Typically, when TYPE is 2, host single-precision floating-point numbers are transferred to the AP and converted into normalized AP floating-point numbers. When the AP is installed in a system, it is set to convert the type of floating-point format used by the specific host.

Typically, when TYPE is 3, IBM 360 32-bit floating-point format numbers are converted to normalized AP floating-point numbers.

Illustrated in Figure 2-2, a variable is assigned immediately following CALL APCLR to define the floating-point format being used: IFMT = 2. This variable is then used for the TYPE parameter in the following statement of the program:

```
CALL APPUT (A, IA, N, IFMT)
```

The number 2 could also have been used for TYPE:

```
CALL APPUT (A, IA, N, 2)
```

2.3.6 SYNCHRONIZATION

Two wait commands, APWR and APWD, are available to ensure that the AP and the host are synchronized in their operation when required.

APWD (wait on data) causes the host program to wait until a data transfer between the host and the AP (the result of a CALL APPUT or CALL APGET) has been completed before the host resumes execution of the program.

APWR (wait on running) causes the host to wait until the AP has finished running before it resumes execution of the program. In general, whenever a data transfer command is called following the execution of a routine by the AP, an APWR should precede the data transfer command.

The two data transfer routines (APPUT and APGET) both wait (in effect CALL APWD) for any previous data transfer to be completed before starting a new data transfer. Two APPUT calls can thus be made in succession without calling a wait in between. Also, the arithmetic operations in the AP Math Library all wait (in effect CALL APWR) for any previous arithmetic operation to be completed before starting a new operation.

APWAIT is a third command that combines the operations of APWD and APWR. It causes the host to wait until any data transfer and any routine execution are both completed before it continues to execute the program.

The AP host interface is capable of transferring data to and from the host while it is processing data. This might be done as a method of reducing program run time. The wait commands can be omitted in cases where it is certain that the data being transferred and the data being processed are not the same. This programming technique should be used with caution because it can cause errors in computations. It is good programming practice to include the wait calls. Refer to section 2.7.4 for more information on programming data transfers while the AP is processing.

2.3.7 PROCESSING DATA

Once the array to be processed is stored in the AP main data memory, the user can operate on it with the AP Math Library routines. In this example, the corresponding consecutive elements of the two 1000-element vectors beginning at addresses IA (=0) and IB (=1000) are added together, and the 1000 sums are stored in the AP main data memory starting at base address IC (=2000):

```
CALL VADD (IA, 1, IB, 1, IC, 1, N)
```

2.3.8 TRANSFERRING DATA BACK TO THE HOST

When array processing has been completed, the user can transfer the resultant array back to the host with an APGET command. The user should remember to call the APWR command to be sure the AP is done processing before transferring data. APGET uses the same four parameters as APPUT. The APGET call is written as follows:

```
CALL APGET (C, IC, N, IFMT)
```

The resultant 1000-word vector is thus moved from AP main data memory locations 2000 to 2999 to the the host memory array C, set up with the original DIMENSION command. IFMT is again 2, which means that each element of the vector is converted from AP floating-point format to host single-precision format.

When TYPE is 0 in an APGET command, the low 32 bits of the AP memory words are transferred without format conversion to the host memory. When TYPE is 1, the low 16 bits of the AP memory words are transferred to the host memory. VFIX (refer to Data Formatting Commands in Chapter 3) can be called prior to this command to convert 38-bit floating-point numbers to 16-bit integers. When TYPE is 3, the AP floating-point numbers are converted into IBM 360 single-precision floating-point numbers and transferred to the host memory.

If overflow or underflow is detected on conversion from AP format when TYPE 2 or 3 format is selected, a signed maximum-quantity is forced on overflow and zero on underflow. This occurs because the dynamic range of the AP (10**-153 to 10**153) is greater than most host computers.

2.4 VECTOR ORGANIZATION

This section discusses vector organization.

2.4.1 REAL VECTORS

Three parameters are required to define a real vector: a starting (or base) address, an address increment, and an element count. The base vector address is the AP main data address of the first vector element to be operated on. The address increment specifies the interval (difference in addresses) between one element of the vector and the next. The element count specifies the number of elements of the vector to be operated on (e.g., the number of multiplications to be performed). For example:

```
CALL VMUL(A,I,B,J,C,K,N)
```

Here A, B and C are base addresses for the three vectors involved in a vector multiply operation. I, J and K are the address increments associated with vectors A, B and C, respectively. N is the element count for each of the vectors. A typical call is:

```
CALL VMUL(100,1,200,2,300,-1,5)
```

For real vectors where elements are stored in consecutive locations, the address increment is 1. Most Math Library functions, however, allow the additional flexibility of specifying arbitrary increments. Table 2-1 shows the memory allocations made in the preceding example.

Table 2-1 CALL VMUL(100,1,200,2,300,-1,5) Memory Allocations

ADDRESS	ELEMENT
100	a(1)
101	a(2)
102	a(3)
103	a(4)
104	a(5)
105	--
200	b(1)
201	--
202	b(2)
203	--
204	b(3)
205	--
206	b(4)
207	--
208	b(5)
296	c(5) = a(5) * b(5)
297	c(4) = a(4) * b(4)
298	c(3) = a(3) * b(3)
299	c(2) = a(2) * b(2)
300	c(1) = a(1) * b(1)
301	--

0854

2.4.2 COMPLEX VECTORS

For operations involving complex vectors, each complex element occupies two consecutive addresses in main data memory. If the complex vector is in rectangular form, then the imaginary component immediately follows the real. In polar form, the phase (in radians) immediately follows the magnitude.

The base address of a complex vector specifies the address of the first real part of the first element. The address increment specifies the address interval (difference in address) between one real part and the next real part. For complex vectors, this interval must be at least 2. The element count refers to the number of complex elements (i.e., reals or imaginaries) to be operated on. For example:

```
CALL CVMUL(A,I,B,J,C,K,N,F)
```

Here A, B and C are complex vectors with address increments of I, J and K, respectively. N is the number of complex elements to be operated on for each vector. F is a flag that is set to 1 for a normal complex multiply, and to -1 if the multiply is to use the complex conjugate of vector A. The following call is an example of a normal complex multiply involving four complex elements:

```
CALL CVMUL(100,2,200,3,300,2,4,1)
```

The memory allocations for this example are shown in Table 2-2.

Table 2-2 CALL CVMUL(100,2,200,3,300,2,4,1) Memory Allocations

ADDRESS	ELEMENT
1ØØ	ar(1)
1Ø1	ai(1)
1Ø2	ar(2)
1Ø3	ai(2)
1Ø4	ar(3)
1Ø5	ai(3)
1Ø6	ar(4)
1Ø7	ai(4)
1Ø8	--
2ØØ	br(1)
2Ø1	bi(1)
2Ø2	--
2Ø3	br(2)
2Ø4	bi(2)
2Ø5	--
2Ø6	br(3)
2Ø7	bi(3)
2Ø8	--
2Ø9	br(4)
21Ø	bi(4)
3ØØ	cr(1)
3Ø1	ci(1)
3Ø2	cr(2)
3Ø3	ci(2)
3Ø4	cr(3)
3Ø5	ci(3)
3Ø6	cr(4)
3Ø7	ci(4)
3Ø8	--

0855

2.4.3 RFFT COMPLEX FORM

A special complex vector form exists for the result of a forward real-to-complex FFT using routines RFFT or RFFTB. For example:

```
CALL RFFT(C,N,F)
```

Here, if F=1 a forward Fast Fourier Transform of a real vector of length N is taken. The result is a complex vector with N/2 + 1 complex elements; but since two of those complex elements (the first and last) have zero imaginary parts, the result can be packed into N locations. The following call is an example of an in-place 8-point forward real-to-complex Fast Fourier Transform.

```
CALL RFFT(100,8,1)
```

The memory allocations before and after the transformation are shown in Table 2-3. Note that FFT input data must be in consecutive locations.

Table 2-3 Memory Allocations before and after CALL RFFT (100,8,1)

ADDRESS	ELEMENT
BEFORE	
1ØØ	t(Ø)
1Ø1	t(1)
1Ø2	t(2)
1Ø3	t(3)
1Ø4	t(4)
1Ø5	t(5)
1Ø6	t(6)
1Ø7	t(7)
1Ø8	--
AFTER	
1ØØ	fr(Ø)
1Ø1	fr(4)
1Ø2	fr(1)
1Ø3	fi(1)
1Ø4	fr(2)
1Ø5	fi(2)
1Ø6	fr(3)
1Ø7	fi(3)
1Ø8	--

0856

Before additional complex operations are performed on the FFT result, the complex vector should be unpacked into proper form by moving the element fr(4) to location 108 and zeroing locations 101 and 109.

The inverse complex-to-real FFT operation (RFFT or RFFTB with F=-1) expects the complex vector to be in the packed form illustrated in Table 2-3.

2.4.4 MATRICES

Matrices are stored in column order in main data memory. A matrix is defined by a base address, an address increment, a row count, and a column count. The base address represents the element in the first row and column to be operated on. The address increment specifies the interval (difference in addresses) between one element of the matrix and the next. The row count specifies the number of elements to be operated on per column (i.e., the number of rows), while the column count specifies the number of columns in the matrix. For example:

```
CALL MTRANS(A,I,C,K,M,N)
```

Here, M columns and N rows of the matrix with base address A are transposed to a matrix whose M rows and N columns are stored starting at address C. I and K are the address increments for A and C, respectively. The following call transposes a 3-row by 2-column matrix.

```
CALL MTRANS(100,1,200,2,2,3)
```

The memory allocation for the matrices are shown in Table 2-4.

Table 2-4 CALL MTRANS(100,1,200,2,2,3) Memory Allocations

ADDRESS	ELEMENT
100	a(1,1)
101	a(2,1)
102	a(3,1)
103	a(1,2)
104	a(2,2)
105	a(3,2)
106	--
200	c(1,1) = a(1,1)
201	--
202	c(2,1) = a(1,2)
203	--
204	c(1,2) = a(2,1)
205	--
206	c(2,2) = a(2,2)
207	--
208	c(1,3) = a(3,1)
209	--
210	c(2,3) = a(3,2)

0857

2.4.5 DOUBLE-PRECISION ELEMENTS

Like complex elements, each double-precision element occupies two consecutive addresses in main data memory. The most significant part of the element comes first and the least significant part second. Both words are stored in normal 38-bit floating-point format with the exponent of the second word being 27 less than the exponent of the most significant word.

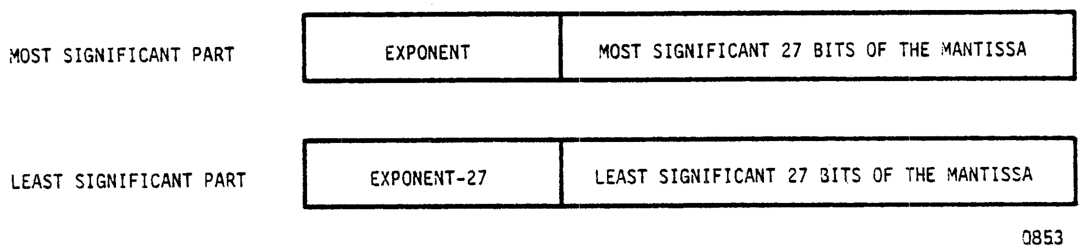

Figure 2-3 Double-Precision Element

2.5 PROGRAM RUN-TIME ENVIRONMENT

Two factors affect the total time it takes to execute an AP Math Library routine called from a FORTRAN program: the AP execution time of the individual routine, and the host system overhead. The AP has a 167ns cycle time during which several operations (add, multiply, fetch, move, branch, etc.) can be performed. All of the AP Math Library routines have been written to make the most efficient use of this parallel structure of the AP and its 167ns basic machine cycle time.

Prior to the execution of a routine by the AP, the host system must load the routine into the AP program source memory (if it has not been previously loaded), and must load the parameters into the s-pad registers. This time interval -- called the host overhead -- adds to the total program run time. Host overhead varies from system to system depending on the complexity of the host operating system and the number of other operations the host is expected to control along with the AP. Host overhead is typically 100 to 1000 microseconds.

Some knowledge of the host/AP run time environment helps the user understand the effect the host overhead has on total program run time. This knowledge is also helpful in section 2.7 where techniques are given which may permit some reduction of both the AP execution time and host overhead.

2.6 UNDERSTANDING HOST OVERHEAD

This section presents information about host overhead.

2.6.1 THE LOAD MODULE

Figure 2-5 shows the standard procedure for writing a FORTRAN program, compiling it, and linking it with the AP Math Library and user-written FORTRAN callable routines. The final load module (refer to Figure 2-6) includes:

- the compiled user-written FORTRAN code
- the various array processor routines called in the program and their AP 64-bit instruction words
- the AP executive subroutine (APEX), including a table which APEX uses to keep track of the contents of the AP program source memory

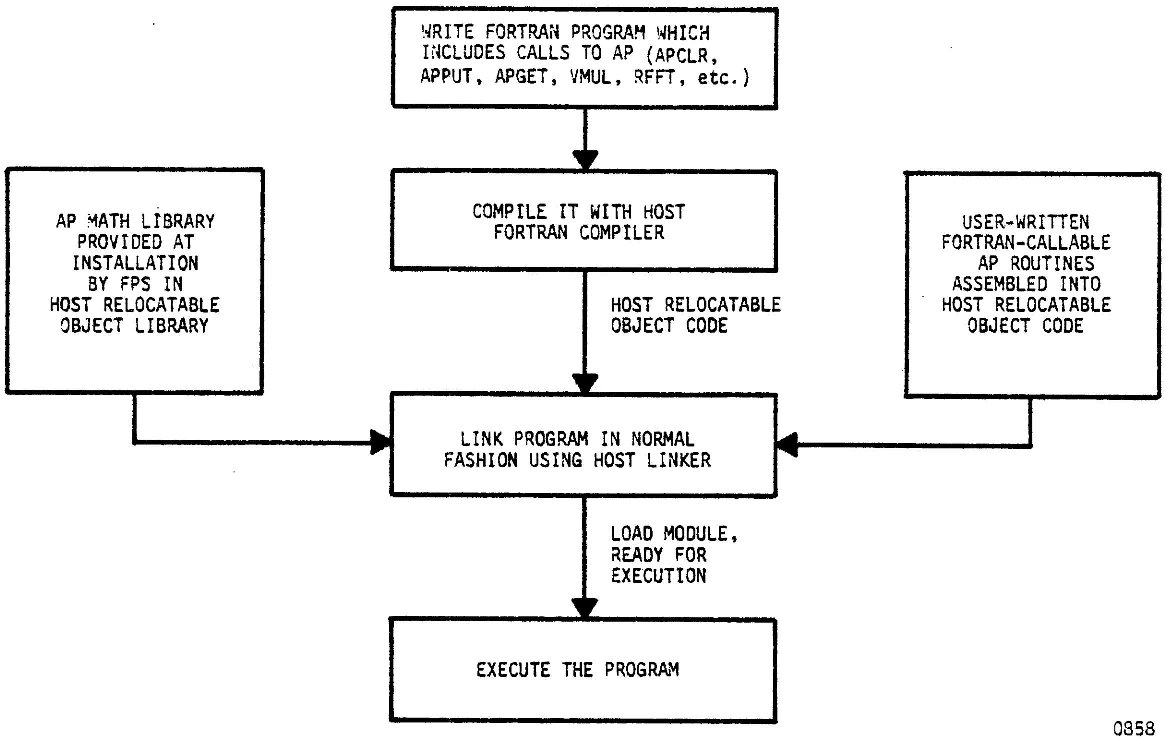

Figure 2-4 AP/Host FORTRAN Software Connection

2.6.2 RUNNING THE FORTRAN PROGRAM

At run time, the load module which contains the FORTRAN calling program, AP Math Library routines, and APEX, is read into the host memory, and the host begins executing the FORTRAN program. When a routine is called, the host jumps to the routine and executes it. If the routine is an AP Math Library routine, a jump to APEX is made.

APEX is a subroutine that controls the interaction of the host with the AP. It handles the loading of the appropriate AP 64-bit instruction words into the AP program source memory, the allocation of the program source memory locations, the loading of the parameters for the routines, and initiates the execution of the instructions by the AP.

The AP program source memory has a minimum size of 512 words. It can be enlarged in 256 word increments to a maximum of 4096 words. Each word in the program source memory is 64 bits long and contains the instruction to be executed during one 167ns clock cycle. Once the instructions for a routine have been read into the program source memory, APEX notes the name of the routine and its location in the program source memory and calculates the remaining space available in the program source memory. This information is stored in a table in the host memory. If a routine is called a second time, APEX does not reload the instructions, but merely loads the new parameters and initiates execution. If a FORTRAN program uses more AP routines than there is space for in the program source memory, APEX overwrites new instructions in the program source memory on a last-in, first-out basis.

Once the execution of the routine begins, APEX returns control to the user-written FORTRAN calling program. This procedure is repeated each time an AP Math Library routine is called. If a routine is called before the AP has finished running a previously-called routine, APEX waits until the AP has completed the routine before it loads the new instructions and/or parameters, and then starts execution of the new routine. When data transfers are called for, APEX tells the host when to begin according to the wait commands -- APWD, APWR and APWAIT -- in the FORTRAN program.

2.6.3 RUN TIME AT THE APEX LEVEL

Figure 2-6 illustrates the sequence of events which occur when a FORTRAN program calls an AP routine which has not yet been loaded (e.g., VADD). When the execution of the FORTRAN program gets to CALL VADD, the program jumps to the VADD routine. VADD does nothing more than call APEX. APEX identifies the calling routine by its return address. This address is then entered in the table in the host memory, and is used to determine whether or not the instructions for the current call are already resident in the AP program source memory. If the routine for the current call is not already resident in the AP, APEX obtains the instructions from the calling routine and transfers them to the AP making an appropriate entry in a table. This table entry records the starting location in program source memory where the instructions have been loaded. APEX also computes the amount of the program source memory space that still remains unused and enters this number in the table. It uses this number in future calls to determine if newly-called instructions must be overlaid in the program source memory.

APEX always tries to load the new instructions in a location that does not destroy previously-loaded instructions. If this is not possible, previous entries in the table are progressively deleted until there is room for the current instructions. The new instructions are then overlaid in the newly-allocated location in the program source memory.

The actual loading of the AP instructions is accomplished via an I/O operation initiated by APEX, but is actually executed in a device handler.

Once the instructions have been loaded in the AP program source memory, APEX obtains the subroutine parameters, transfers them to the AP s-pad registers, and triggers execution of the instructions. APEX then returns control to the routine which called it; that routine immediately returns control to the FORTRAN calling program.

The time used between the call in the FORTRAN program and the beginning of execution of instructions in the AP constitutes the host overhead for that call. This host overhead (typically 100 to 1000 microseconds) is incurred each time an AP Math Library routine is called.

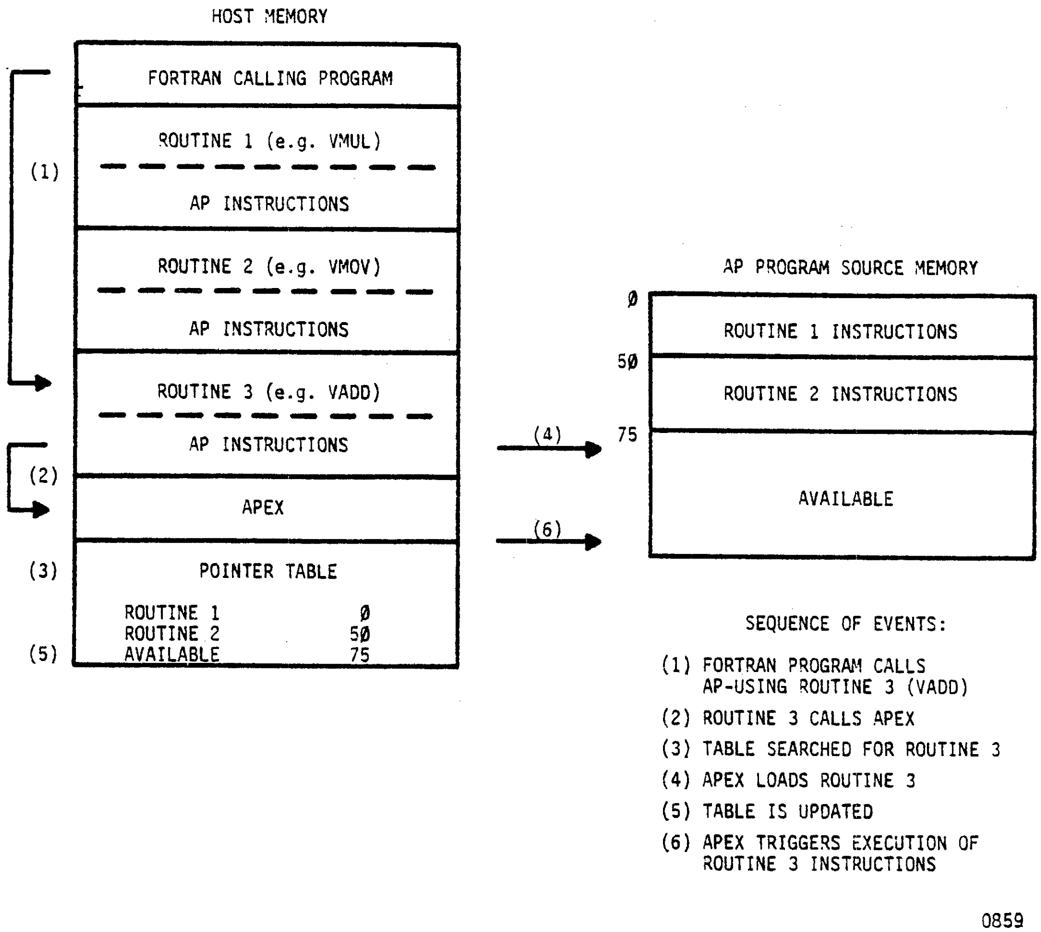

Figure 2-5 Transferring AP Instructions from Host Memory to AP Program Source Memory

2.7 OPTIMIZING PROGRAM RUN TIME

A number of items affect the rate at which a FORTRAN program runs on the AP. Significant factors are the cycle rate of the main data memory and the placement of vectors in the main data memory. Host overhead and the timing of data transfer between the host and the AP also have an effect on program run time.

2.7.1 FACTORS AFFECTING AP EXECUTION TIME

Floating Point Systems, Inc., offers main data memory for the AP with a choice of two different cycle rates: 167ns or 333ns. This cycle rate is the minimum time it takes to access a word of memory following a previous access. This minimum time is achieved when consecutive memory accesses alternate between even and odd addresses, or between 8K or 32K memory banks (depending on the chip type). If consecutive accesses specify only even (or only odd) addresses, then the access takes 167ns longer for either memory. The machine cycle rate of the AP is 167ns, so the choice of memory can have an effect on how fast the program runs.

If a routine requires the memory to be accessed each machine cycle, the routine runs twice as fast with the 167ns memory as it would with the 333ns memory providing the even-odd address interleaving is maintained. If the routine calls for a memory access only every third or every fourth machine cycle, the routine runs at the same rate with either memory.

In actual operation, routines in the AP Math Library run anywhere from the same rate to twice as fast on the 167ns memory depending on the routine. The AP Math Library routines are written differently for the two types of memory when necessary to obtain the optimum speed. The calling sequence and numerical results are identical in each case.

2.7.2 SPECIFYING VECTOR LOCATIONS IN MAIN DATA MEMORY

Because of the even-odd interleave of main data memory, subroutines run at different rates depending on where the vectors are located (i.e., base address), and also on the address increments associated with each vector.

Three execution times (BEST, TYPICAL, and WORST) are given for each memory type in each description of a Math Library routine in Appendix E. When operating on real vectors, the TYPICAL time reflects the typical situation where all vectors are compactly stored (the address increments I, J and K equal 1 or any odd number: -1, 3, 5, etc.), and the base addresses are either all even or all odd.

Sometimes it is possible to achieve faster execution by varying the base addresses of vectors between even and odd locations. The vector(s) whose base address(es) should be odd when the others are even (or even when the others are odd) are indicated in parentheses next to the BEST execution time. If no vectors are indicated, the best and typical execution times are the same.

The worst case times involve other even-odd addressing and increment combinations.

Table 2-3 shows the timing for the VADD routine.

Table 2-5 VADD Execution Times

MEMORY	EXECUTION MEMORY TIME/LOOP (μs)			
	BEST	TYPICAL	WORST	SETUP(us)
167 ns	Ø.5 (B)	Ø.8	1.Ø	2.7
333 ns	1.Ø (A)	1,3	1.5	1.2

0860

Note that the execution time is specified on a per-loop basis. Thus, if VADD is called to add two 1000-element vectors, the typical execution time with a 167ns memory is 1000 x 0.8 (really 0.833us) = 833us, plus an additional 2.7us of SETUP time needed to initially fill the AP pipeline. Thus, the total execution time using 167ns memory is 836us when all base addresses are even (or all odd).

Three base address parameters are specified for VADD -- one each for the two vectors to be added together, A and B, and one -- C -- for the location where the result is to be stored. With the 167ns memory, the best run time is obtained when B is an odd address and A and C are even (or vice versa -- B is even and A and C are odd). For the example in the preceding paragraph, an execution time of 1000x0.5+2.7=502.7us is obtained. If, for example, A is 0, address B is 1001, and C is 2002. With the 333ns memory, address A must be odd (even) when B and C are even (odd) to obtain the fastest execution (e.g., A=0, B=1001, C=2001).

With complex vector operations, the typical time reflects the most likely situation where the increment is even (for compactly stored complex vectors the increment is two), and all base addresses are either all even or all odd. (An increment of one with a complex vector operation produces unpredictable results since each complex element requires two#main data memory words.) When faster execution is possible with even complex vector increments by adjusting the base addresses between even and odd locations, the vector(s) whose base address(es) should be odd when the others are even (or even when the others are odd) are indicated next to the BEST execution time.

The times given for matrix operations are for cases where the matrices are stored compactly (i.e., the memory increments are one so that the matrix elements are stored in consecutive addresses). In some cases, the run times are data-dependent, and in other cases they are dependent on the sizes of the matrices being operated on.

2.7.3 MINIMIZING THE EFFECT OF HOST OVERHEAD

There may be certain situations, especially when the host is operating in a multi-user, multi-task environment where the host overhead time represents a substantial fraction of the total run time involved in processing with the AP. The purpose of this section is to suggest some techniques for reducing the effect of this overhead if the application is time critical.

- Combine FORTRAN calls to the AP. The most effective method of minimizing the effects of host overhead is to reduce the number of calls to the AP from the host. Often this can be done by careful layout of the program.

  Some suggestions:

  Concentrate several vectors in consecutive addresses so that several vectors can be transferred with a single APPUT call.

  Use AP Math Library routines which replace multiple library calls. For example, VMMA performs the same operations as two VMUL calls, and a VADD with a savings not only of two host calls, but 40 percent in AP execution time.

  Overlap the operations of the host and the AP whenever possible.

  Since all the AP Math Library routines can be called as routines from other AP assembly language programs, it is possible to write special FORTRAN callable array processing routines which combine a series of calls to the AP Math Library. One FORTRAN call to the special routine then replaces the separate calls in the host program. Take care that the special purpose routine (including all the AP Math Library routines) is small enough to fit in the available program source memory space.

- Load the most used routines first. If the program requires more space in the program source memory than there is available, the user can minimize some of the effects of host overhead by calling the most often used routines in the program first. As was stated in the discussion of the run-time environment, APEX uses the last-in, first-out technique to allocate space for routines in the program source memory. If there is no room in the program source memory for a new routine, the last program words read into the memory are over-written for a new routine; the last program words read into the memory are over-written until there is enough space for the new routine. Since it requires less host overhead to load the parameters of a routine that already exists in program source memory than to load both the routines and the parameters, it is advantageous to call the most often used routines early in the program to make sure they are located well down in the program source memory. It may even be useful to call a routine before it is needed and give it only a dummy operation to do.

- Other suggestions:

  In single-task host operating systems, APEX generally talks directly to the AP; in multi-task systems, APEX usually must ask the host system for permission to talk to the AP. This may take as long as 1 ms. In such a situation, consider the possibility of switching to a simpler host operating system.

  Operating on large arrays is more efficient than operating on small arrays.

  Consider overlapping data transfer and processing.

2.7.4 OVERLAPPING DATA TRANSFER AND PROCESSING

The AP's highly parallel operation allows the user to write programs that process and transfer data simultaneously. This type of programming involves leaving out some of the program synchronization commands -- APWR, APWD and APWAIT. Leaving out wait commands, however, presents the potential problem of asynchronous operation between the AP and the host. Thus, there is a chance that results are processed before they are actually present in their assigned location in main data memory. The wait commands are provided to avoid such problems.

Programming involving simultaneous processing and data transfers is available at the FORTRAN level as well as the AP machine language level. The advantage is that it can speed up program run time when used without loss of synchronization.

The success of this type of programming depends on host overhead, or in other words, how dedicated the host is to servicing the needs of the AP.

2.7.5 WRITING AP ASSEMBLY LANGUAGE PROGRAMS

An AP assembly language routine can be made FORTRAN callable by including the following pseudo-operation in the routine:

```
$ENTRY name,p
```

where "name" is the FORTRAN name for the routine, and "p" indicates the number (maximum of 16) of parameters in the FORTRAN call. At run time, APEX transfers these parameters to s-pad registers 0 through p-1.

Thus, the pseudo-operation for the AP using FORTRAN command CALL VUSER(A,I,B,J,C,K,N) is $ENTRY VUSER,7. At run time, APEX transfers the seven parameters as shown in Table 2-6.

Table 2-6 Parameter Transfer

S-PAD REGISTER	CONTENTS
Ø	A
1	I
2	B
3	J
4	C
5	K
6	N

0861

Figure 2-7 illustrates the procedure for creating a FORTRAN callable AP assembly language routine.

All the AP Math Library routines have been written in AP assembly language. The user can learn more about this language through the manuals avaialble from Floating Point Systems, Inc.

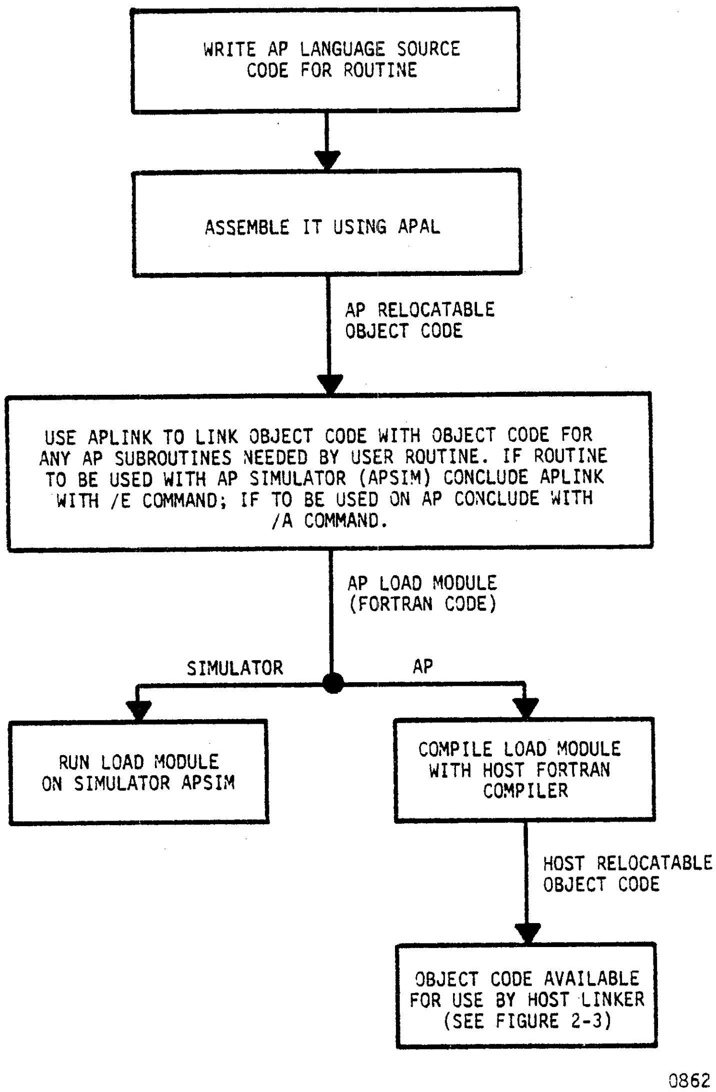

Figure 2-6 Procedure for Creating User-Written FORTRAN Callable AP Assembly Language Routines

CHAPTER 3

DESCRIPTION OF AP MATH LIBRARY ROUTINES

3.1 INTRODUCTION

This chapter describes the categories of routines that are contained in the AP Math Library.

3.2 GENERAL INFORMATION ABOUT ROUTINES

The AP Math Library is divided into 12 categories:

- data transfer and control operations
- basic vector arithmetic
- vector-to-scalar operations
- vector comparison operations
- complex vector arithmetic
- data formatting operations
- matrix operations
- FFT operations
- auxiliary operations
- signal processing operations (optional)
- table memory operations (optional)
- APAL-callable utility operations

3.2.1 DATA TRANSFER AND CONTROL OPERATIONS

These commands control the data transfer and program synchronization between the AP and the host. They are actually part of APEX, the AP executive, and thus require no space in the AP program source memory. The execution time for these routines depends on the speed of the host system. Refer to sections 2.3.5 through 2.3.8 for more information about these calls.

3.2.2 BASIC VECTOR ARITHMETIC

This group includes routines to perform basic real vector arithmetic operations such as vector add (VADD), subtract (VSUB), multiply (VMUL), and divide (VDIV). Also included are trigonometric functions, logarithms, simple logical operations, and vector generation (e.g., constants, ramps, random numbers). The vectors operated on must conform to the real vector format shown in section 2.4.1.

3.2.3 VECTOR-TO-SCALAR OPERATIONS

These real vector operations determine global characteristics of a vector. They determine a single value that characterizes one facet of the vector: sum of all the elements (SVE) or value of the largest element (MAXV), etc.

3.2.4 VECTOR COMPARISON OPERATIONS

These real vector operations perform compare and replace operations. They create a third vector based on the comparison of two vectors. VMAX, for example, sets the elements of a third vector equal to the larger of each pair of corresponding elements in two vectors.

3.2.5 COMPLEX VECTOR ARITHMETIC

All the commands in this group operate on complex vectors or combinations of real and complex vectors. The complex vectors must conform to the complex vector format described in section 2.4.2. In general, the increment parameter used in a complex vector operation must always be two or greater when specifying a complex vector. A complex element is made up of two parts -- a real part and an imaginary part stored in consecutive words in the AP main data memory. The parameter N in a complex vector routine always refers to the number of complex elements (pairs).

When the operation involves both real and complex vectors, the TYPICAL execution times are obtained when the increments of the real vectors are odd and the increments of the complex vectors are even, and the base addresses of all the vectors are either all even or all odd.

Note that some complex vector operations can be done with real vector routines. CALL VCLR (0, 1, 1000), for example, clears a complex vector of 500 complex elements that begins at location 0.

3.2.6 DATA FORMATTING OPERATIONS

The 38-bit AP floating-point format is illustrated in Figure 3-1. Bits 0, 1, and 40 are memory parity bits.

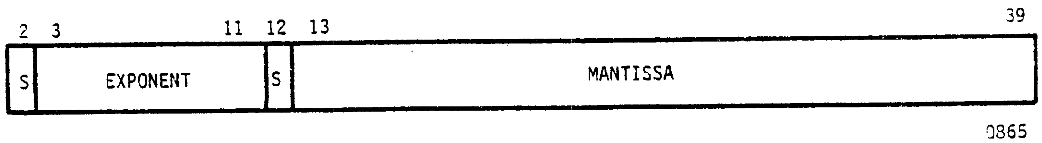

Figure 3-1 AP Floating-Point Format

The data formatting operations provide a number of normalizing and integer-to-floating-point conversion routines to operate on data stored in the AP main data memory.

VFLT normalizes a vector that has been transferred to the AP with an APPUT command, using TYPE 1 format-conversion (16-bit integer converted to unnormalized 38-bit floating-point format). Refer to the discussion of APPUT beginning with section 2.3.5. Normalization means shifting the most significant bit of the mantissa to bit 13 of the 38-bit word with appropriate changes in the exponent.

VFIX converts a normalized 38-bit floating-point word into a 16-bit, 2's complement integer residing in the lower 16 bits (bits 24 to 39) of the 38-bit word. This operation is used prior to transferring data with the APGET command when using TYPE 1 format conversion (see the discussion of APGET beginning with section 2.3.8). VSCALE, VSCSCL and VSHFX are variations of VFIX.

For convenience, there are also a number of integer unpacking and conversion operations for use with the TYPE 0 format-conversion in APPUT and APGET. For example, VUP16 converts two 16-bit integers packed in the lower 32 bits of a 38-bit word into two 38-bit, normalized floating-point words. VUP8 converts four 8-bit integers stored in the lower 32 bits of a 38-bit word into four 38-bit normalized floating-point words. VPK16 and VPK8 perform the reverse operation: two 38-bit floating-point words are converted into two 16-bit integers; four floating-point words are converted into four 8-bit integers in a single 38-bit word.

3.2.7 MATRIX OPERATIONS

The matrix routines perform typical matrix operations such as multiplication (MMUL), transposition (MTRANS), and inversion (MATINV). A matrix is always stored in the AP main data memory as a sequence of columns (see section 2.4.4 for a discussion of the AP matrix format). The M and N notation used in the matrix operation parameters refers to the number of rows (M) and the number of columns (N) in a matrix. For example, an operation on a matrix starting at address C involves MC rows and NC columns.

Timing information is given with each matrix routine for representative matrix sizes. An address increment of one (i.e., a compactly stored matrix) is assumed for all the times given.

3.2.8 FFT OPERATIONS

The FFT commands perform Fast Fourier Transforms on both real and complex vectors. Each FFT routine performs both the forward transform (time-to-frequency) and the inverse transform (frequency-to-time), depending on the parameter F (+1 for forward, -1 for inverse). There are two categories of FFT routines: in-place and not-in-place. The in-place routines (RFFT and CFFT) transform the time elements from N locations in main data memory and store the resultant complex frequency elements in the same locations in the main data memory. If the main data memory is 8192 words, the user can perform an FFT on N = 8192 real points, or N = 4096 complex points. The not-in-place FFT routines (RFFTB and CFFTB) run somewhat faster than the in-place routines, but require separate locations in main data memory for the time points and the resultant frequency points.

When transforming real time elements into complex frequency elements, a special method of packing the complex frequency elements is used. A FFT of N real time points actually produces N/2 + 1 complex frequency elements. Since a complex element consists of a real and an imaginary part, N + 2 words are thus required to store N/2 + 1 complex elements. It is known, however, that the I(0) and the I(N/2) frequency points are always 0. Therefore, when performing a real-to-complex FFT with the RFFT or RFFTB commands, the R(N/2) frequency point is stored in the I(0) memory location, and the I(N/2) frequency point (always 0) is dropped. The results of a N-point real FFT can thus be stored in N words. An 8-point real-to-complex FFT, for example, is packed as shown in Table 3-1.

The RFFTSC routine is provided to allow unpacking of the complex RFFT vector and scaling of the data. Two types of unpacking are provided. In Type I (refer to Table 3-1), the I(0) location in memory (which now holds the R(N/2) data point) is cleared to zero and the R(N/2) value is discarded. The value of R(N/2) is often considered unimportant since it represents the frequency component at the Nyquist frequency. Type I unpacking would be used when performing in-place transforms where all the available main data locations are being used. In Type II unpacking, the R(N/2) value is moved from the I(0) location to its proper R(N/2) location, and the I(0) and I(N/2) memory locations are cleared to zero. Thus, in Type II unpacking, all the complex data points are retained. The complex RFFT format is used for both the in-place and not-in-place real-to-complex transforms. For complex-to-real inverse FFTs, the complex elements must be repacked into the complex RFFT format. RFFTSC also handles the repacking procedure.

Table 3-1 Real-to-Complex FFT Vector Format

ADDRESS	TIME POINTS	COMPLEX RFFT PACKING	TYPE I UNPACKING	TYPE II UNPACKING
1ØØ	t(Ø)	R(Ø)	R(Ø)	R(Ø)
1Ø1	t(1)	R(4)	Ø	Ø
1Ø2	t(2)	R(1)	R(1)	R(1)
1Ø3	t(3)	I(1)	I(1)	I(1)
1Ø4	t(4)	R(2)	R(2)	R(2)
1Ø5	t(5)	I(2)	I(2)	I(2)
1Ø6	t(6)	R(3)	R(3)	R(3)
1Ø7	t(7)	I(3)	I(3)	I(3)
1Ø8	-	-	-	R(4)
1Ø9	-	-	-	Ø

0863

The complex-to-complex FFT routines, CFFT and CFFTB, require no such packing or unpacking since all operations are performed on properly formatted complex vectors. A FFT of N complex time elements produces N complex frequency elements.

The data obtained either from a forward RFFT or a forward CFFT requires rescaling. Table 3-2 shows the multiplying factors for each transform to get back to the original scale. The RFFTSC and CFFTSC routines, respectively, provide parameters for scaling the results. Note that no scaling is required for the inverse transforms.

Table 3-2 Multiplying Factors for Scaling FFT Results

ROUTINES	FORWARD	INVERSE
CFFT, CFFTB	1/N	1
RFFT, RFFTB	1/(2*N)	1

0864

3.2.9 AUXILIARY OPERATIONS

The commands in the auxiliary operations group perform miscellaneous operations such as numerical integration, evaluation of polynomials, and convolutions.

3.2.10 SIGNAL PROCESSING OPERATIONS (OPTIONAL)

The signal processing operations consist of a collection of real and complex routines which are often used in conjunction with the FFT routines. They perform many widely used time series analysis calculations such as auto-spectrum (ASPEC), cross-spectrum (CSPEC), coherence function (COHER) and histogram (HIST).

3.2.11 TABLE MEMORY OPERATIONS (OPTIONAL)

These subroutines are for use with the optional writable table memory (TMRAM). This table memory is used in conjunction with the standard read only table memory in the AP.

The TMRAM allows the user to either create a table of his own special purpose constants, or use the additional memory space as an adjunct to the main data memory. The TMRAM has a 167ns memory access time. When used in conjunction with the main data memory, it can speed up basic vector arithmetic operations such as add, subtract, multiply and move. For example, MTTADD adds one vector from the main data memory to a vector from the TMRAM, and stores the resultant vector in the TMRAM. Since the AP can access a word in the main data memory and a word in the TMRAM in the same machine cycle, one machine cycle is saved in the calculation.

The nomenclature used in these calls refers to the main data memory (M) and the TMRAM (T). In the MMTADD(A, I, B, J, C, K, N) command, for example, A and B are base addresses in main data memory (M), and C is a base address in the TMRAM (T). The addresses in the table memory are numbered consecutively from 0 as in the main data memory.

3.2.12 APAL-CALLABLE UTILITY OPERATIONS

These routines are called by many of the FORTRAN callable routines in the AP Math Library (refer to the category EXTERNALS below the dotted line in the routine descriptions). They are callable from programs written in APAL, but are not callable from FORTRAN programs. This miscellaneous assortment of routines includes scalar functions such as sine, cosine and square root, several routines called in the FFT operations, and double-precision scalar functions. All pertinent information is given for each routine except for the FORTRAN CALL, PARAMETERS and EXAMPLE. The execution times given are generally the total executing time since most of the routines are non-repetitive.

CHAPTER 4

PROGRAMMING EXAMPLES

4.1 INTRODUCTION

This chapter contains four examples illustrating the use of the AP Math Library routines in FORTRAN programs. The first two examples show the replacement of FORTRAN arithmetic DO loops with FORTRAN code which perform equivalent processing in the AP. The last two examples illustrate programs which perform double-tapered convolution operations in the AP -- one using time-domain techniques, the other using frequency-domain techniques.

4.1.1 EXAMPLE 1: A BENCHMARK PROGRAM, INCLUDING AP MEMORY MAP

This section lists a benchmark program which includes an AP memory map.

```
C******** EXAMPLE 1 = BENCHMARK PROGRAM ********************
C
C=================================================================
C     ORIGINAL FORTRAN
C
C       SUBROUTINE EX1(SCLR,AM,V,VX,VY,X,X0,Y,Y0,Z,N)
C       DIMENSION AM(N),V(N),VX(N),VY(N),X(N),X0(N),Y(N),Y0(N),Z(N)
C       DO 1 I=1,N
C       VX(I)=SCLR * (X(I)-X0(I))
C       VY(I)=SCLR * (Y(I)-Y0(I))
C       V(I)=SQRT(VX(I)**2 + VY(I)**2)
C  1    Z(I)=AM(I) * (X(I)*VY(I) - Y(I)*VX(I))
C       RETURN
C       END
C=================================================================
C
        SUBROUTINE EX1(SCLR,AM,V,VX,VY,X,X0,Y,Y0,Z,N)
        DIMENSION AM(N),V(N),VX(N),VY(N),X(N),X0(N),Y(N),Y0(N),Z(N)
C----ALLOCATE AP MEMORY (SEE MEMORY MAP AT END OF PROGRAM)
        ISCLR=0
        IAM=ISCLR+1
        IV=IAM+N
        IVX=IV+N
        IVY=IVX+N
        IX=IVY+N
        IX0=IX+N
        IY=IX0+N
        IY0=IY+N
        IZ=IY0+N
C----INITIALIZE AP
```

```
      CALL APCLR
C----PUT OUT DATA TO AP
      CALL APPUT(AM,IAM,N,2)
      CALL APPUT(X,IX,N,2)
      CALL APPUT(X0,IX0,N,2)
      CALL APPUT(Y,IY,N,2)
      CALL APPUT(Y0,IY0,N,2)
      CALL APPUT(SCLR,ISCLR,1,2)
      CALL APWD
C
C----DO THE COMPUTATION
C
C    AP COMPUTATION TIME FOR N=1000 IS 8.2 MS FOR 167 NS MEMORY,
C    11.9 MS FOR 333 NS MEMORY, EXCLUSIVE OF HOST SYSTEM OVERHEAD
C
      CALL VSUB(IX,1,IX0,1,IVX,1,N)
      CALL VSUB(IY,1,IY0,1,IVY,1,N)
      CALL VSMUL(IVX,1,ISCLR,IVX,1,N)
      CALL VSMUL(IVY,1,ISCLR,IVY,1,N)
      CALL VMMA(IVX,1,IVX,1,IVY,1,IVY,1,IV,1,N)
      CALL VSQRT(IV,1,IV,1,N)
      CALL VMMSB(IX,1,IVY,1,IY,1,IVX,1,IZ,1,N)
      CALL VMUL(IAM,1,IZ,1,IZ,1,N)
      CALL APWR
C----GET RESULTS FROM AP
      CALL APGET(VX,IVX,N,2)
      CALL APGET(VY,IVY,N,2)
      CALL APGET(V,IV,N,2)
      CALL APGET(Z,IZ,N,2)
      CALL APWD
      RETURN
      END
```

```
C
C     AP MEMORY MAP FOR N=1000
C
C
C           ADDRESS                         CONTENTS
C                                       -----------------
C       ISCLR = 0                      |      SCLR       |
C                                      |   - - - - - -   |
C       IAM   = 1                      |      AM         |
C                                      |                 |
C                                      |   - - - - - -   |
C       IV    = 1001                   |      V          |
C                                      |                 |
C                                      |   - - - - - -   |
C       IVX   = 2001                   |      VX         |
C                                      |                 |
C                                      |   - - - - - -   |
C       IVY   = 3001                   |      VY         |
C                                      |                 |
C                                      |   - - - - - -   |
C       IX    = 4001                   |      X          |
C                                      |                 |
C                                      |   - - - - - -   |
C       IX0   = 5001                   |      X0         |
C                                      |                 |
C                                      |   - - - - - -   |
C       IY    = 6001                   |      Y          |
C                                      |                 |
C                                      |   - - - - - -   |
C       IY0   = 7001                   |      Y0         |
C                                      |                 |
C                                      |   - - - - - -   |
C       IZ    = 8001                   |      Z          |
C                                      |                 |
C                                      |   - - - - - -   |
C               9001                   |    UNUSED       |
C                                      |                 |
C                                      |                 |
C                                       -----------------
```

4.1.2 EXAMPLE 2: A GEOPOTENTIAL CALCULATION PROGRAM

This section lists a geopotential calculation program.

```
C******** EXAMPLE 2 = GEOPOTENTIAL CALCULATION **************
C
C=====================================================================
C     ORIGINAL FORTRAN
C
C       SUBROUTINE EX2
C       COMMON /B/PHIB(100,10),HB(100,10),PKB(100),DS12
C       DO 1 J=1,9
C       DO 1 I=1,100
C  1    PHIB(I,J+1)=PHIB(I,J)+DS12*PKB(I)*(HB(I,J+1)+HB(I,J))
C       RETURN
C       END
C=====================================================================
C
        SUBROUTINE EX2
        COMMON /B/PHIB(100,10),HB(100,10),PKB(100),DS12
C-----AP MEMORY LAYOUT
        IDS12=0
        IPKB=1
        IHB=IPKB+100
        IPHIB=IHB+1000
C-----INITIALIZE THE AP
        CALL APCLR
C-----PUT OUT THE DATA TO AP
```

```
      CALL APPUT(PHIB,IPHIB,1000,2)
      CALL APPUT(HB,IHB,1000,2)
      CALL APPUT(PKB,IPKB,100,2)
      CALL APPUT(DS12,IDS12,1,2)
      CALL APWD
C
C-----DO THE COMPUTATION
C
C     AP COMPUTATION TIME IS 2.3 MS FOR 167 NS MEMORY, 3.7 MS FOR
C     333 NS MEMORY, EXCLUSIVE OF HOST SYSTEM OVERHEAD
C
      CALL VSMUL(IPKB,1,IDS12,IPKB,1,100)
      CALL VADD(IHB+100,1,IHB,1,IHB,1,900)
      JHB=IHB
      DO 1 J=1,9
      CALL VMUL(IPKB,1,JHB,1,JHB,1,100)
  1   JHB=JHB+100
      CALL VADD(IPHIB,1,IHB,1,IPHIB+100,1,900)
      CALL APWR
C-----GET THE RESULTS FROM AP
      CALL APGET(PHIB(1,2),IPHIB+100,900,2)
      CALL APWD
      RETURN
      END
```

4.1.3 EXAMPLE 3: A TIME-DOMAIN CONVOLUTION PROGRAM

This section lists a time-domain convolution program.

```
C******** EXAMPLE 3 = TIME DOMAIN CONVOLUTION ***************
C
        SUBROUTINE TCONV(TRACE,FILTER,RESULT,NTRACE,NFILT,NRESLT)
        INTEGER NTRACE,NFILT,NRESLT
        REAL TRACE(NTRACE),FILTER(NFILT),RESULT(NRESLT)
C
C       DOES A TIME DOMAIN CONVOLUTION OF 'TRACE' WITH 'FILTER',
C       PRODUCING 'RESULT'.
C       A DOUBLE TAPERED CONVOLUTION IS DONE BY PADDING THE SUPPLIED
C       TRACE WITH BOTH LEADING AND TRAILING ZEROS IN THE AP-120B.
C
C-------PARAMETERS:
C
C       TRACE  - INPUT DATA TRACE
C       FILTER - INPUT FILTER
C       RESULT - OUTPUT RESULT
C       NTRACE - NUMBER OF TRACE POINTS
C       NFILT  - NUMBER OF FILTER POINTS
C       NRESLT - NUMBER OF RESULT POINTS, MUST EQUAL
C       NTRACE+NFILT-1 !!!!!!!!!
C
C       NOTE: THE RESULT MAY BE STORED IN THE HOST ON TOP OF EITHER THE
C       DATA OR THE FILTER
C
C-------ROUTINES USED: APPUT,APGET,APCLR,APWR,APWD,VCLR,CONV
C
C       LOCAL STORAGE
        INTEGER IFMT,NPAD,ITRACE,IFILT
C
C-------METHOD:
C
C       FOR EXAMPLE, NTRACE=5, NFILT=3:
C       THEN:
C         NRESLT=7
C         ITRACE=0
C         IRESLT=0
C         IFILT=9
C         NPAD=2
C
```

```
C       AP MEMORY LAYOUT:
C
C      LOC
C       0      0              <--- ITRACE   <--- IRESLT
C       1      0                      '             '
C       2      TRACE PT #1            '             '
C       3        "    "  2            '             '
C       4        "    "  3            '             '
C       5        "    "  4            '             '
C       6        "    "  5            '             '
C       7      0                      '             '
C       8      0                      '             '
C       9      FILTER PT #1   <--- IFILT
C      10        "     "  2           '
C      11        "     "  3           '
C
C
        IFMT=2                    /*FORMAT 2 FOR FLOATING POINT
C-----INITIALIZE AP
        CALL APCLR
C-----ALLOCATE AP MEMORY
        NPAD=NFILT-1                              /*NUMBER OF ZERO PADS
        ITRACE=0                             /*TRACE LOCATION IN THE AP
        IFILT=ITRACE+NTRACE+NPAD*2    /*FILTER LOCATION IN THE AP
C-----TRANSFER DATA TO AP
        CALL APPUT(TRACE,ITRACE+NPAD,NTRACE,IFMT)       /*PUT THE TRACE
        CALL APPUT(FILTER,IFILT,NFILT,IFMT)        /*PUT THE FILTER
        CALL APWD
C-----DO THE COMPUTATION
        CALL VCLR(ITRACE,1,NPAD)                   /*FRONT ZERO PAD
        CALL VCLR(ITRACE+NRESLT,1,NPAD)                 /*BACK ZERO PAD
C-----DO IT
        CALL CONV(ITRACE,1,IFILT+NFILT-1,-1,ITRACE,1,NRESLT,NFILT)
        CALL APWR
C-----TRANSFER RESULTS FROM AP
        CALL APGET(RESULT,ITRACE,NRESLT,IFMT)           /*GET RESULTS
        CALL APWD
        RETURN
        END
```

4.1.4 EXAMPLE 4: A FREQUENCY-DOMAIN CONVOLUTION PROGRAM

This section lists a frequency-domain convolution program.

```
C******** EXAMPLE 4 = FREQUENCY DOMAIN CONVOLUTION **********
C
      SUBROUTINE FCONV(TRACE,FILTER,RESULT,NTRACE,NFILT,NRESLT)
      INTEGER NTRACE,NFILT,NRESLT
      REAL TRACE(NTRACE),FILTER(NFILT),RESULT(NRESLT)
C
C     DOES A FREQUENCY DOMAIN CONVOLUTION OF 'TRACE' WITH 'FILTER',
C     PRODUCING 'RESULT'.  A DOUBLE TAPERED CONVOLUTION IS DONE.
C
C-------PARAMETERS:
C
C     TRACE  - INPUT DATA TRACE
C     FILTER - INPUT FILTER
C     RESULT - OUTPUT RESULT
C     NTRACE - NUMBER OF TRACE POINTS   (MUST BE A POWER OF 2)
C     NFILT  - NUMBER OF FILTER POINTS
C     (MUST BE A POWER OF 2 <= NTRACE)
C     NRESLT - NUMBER OF RESULT POINTS  (MUST EQUAL NTRACE)
C
C     NOTE: THE RESULT MAY BE STORED IN THE HOST ON TOP OF EITHER THE
C     DATA OR THE FILTER
C
C-------ROUTINES USED: APPUT,APGET,APCLR,APWR,APWD,VCLR,RFFT,VMUL,
C     CVMUL,RFFTSC
C
C     LOCAL STORAGE
      INTEGER IFMT,NFFT,ITRACE,IFILT
```

```
C
      IFMT=2                          /*FORMAT 2 FOR FLOATING POINT
C-----INITIALIZE AP
      CALL APCLR
C-----ALLOCATE AP MEMORY
      NFFT=NTRACE*2                              /*FFT SIZE
      ITRACE=0                             /*LOCATION OF TRACE IN AP
      IFILT=ITRACE+NFFT                   /*LOCATION OF FILTER IN AP
C-----TRANSFER DATA TO AP
      CALL APPUT(TRACE,ITRACE,NTRACE,IFMT)             /*PUT TRACE
      CALL APPUT(FILTER,IFILT,NFILT,IFMT)              /*PUT FILTER
      CALL APWD
C-----DO THE COMPUTATION
      CALL VCLR(ITRACE+NTRACE,1,NFFT-NTRACE)           /*PAD TRACE
      CALL VCLR(IFILT+NFILT,1,NFFT-NFILT)              /*PAD FILTER
      CALL RFFT(ITRACE,NFFT,1)              /*FORWARD FFT TRACE
      CALL RFFT(IFILT,NFFT,1)              /*FORWARD FFT FILTER
      CALL VMUL(ITRACE,1,IFILT,1,ITRACE,1,2)      /*CROSS MUL 1ST 2
C-----DO REST
      CALL CVMUL(ITRACE+2,2,IFILT+2,2,ITRACE+2,2,NTRACE-1,1)

      CALL RFFTSC(ITRACE,NFFT,0,-1)  /*SCALE RESULTS BY 1/(4*NFFT)
      CALL RFFT(ITRACE,NFFT,-1)                        /*INVERSE FFT
      CALL APWR
C-----TRANSFER RESULTS FROM AP
      CALL APGET(RESULT,ITRACE,NFFT,IFMT)              /*GET RESULTS
      CALL APWD
      RETURN
      END
```

CHAPTER 5

FORTRAN MATH LIBRARY SIMULATOR (MATHSIM)

5.1 INTRODUCTION

The FORTRAN Math Library Simulator (MATHSM) is comprised of a series of FORTRAN subroutines. These subroutines simulate the AP Math Library routines and APEX routines which control data flow to and from the AP (DMA), process data in the AP, and synchronize host/AP operations. MATHSIM allows the user to check FORTRAN programs which make numerous calls to the Math Library and APEX without use of the AP. It estimates various host/AP program execution times such as data flow time, AP program execution time, and host overhead time. MATHSIM allows detection of possible host/AP synchronization errors. Adjustment of a few program parameters enables MATHSIM to closely simulate all of the many host/AP systems, thus allowing the user to predict the effects of possible system modifications upon execution.

MATHSIM provides the basic features outlined in the following sections.

5.1.1 MATH LIBRARY ROUTINES

MATHSIM provides an equivalent FORTRAN routine for each Math Library routine simulated. Each routine has a calling sequence identical to that used by the actual Math Library routine. No changes in the user's FORTRAN program are necessary. The user simply compiles the calling program, links to the simulated Math Library, and runs the program as if the AP were present.

5.1.2 DMA

MATHSIM simulates the flow of data in and out of the AP by calls to the APEX routines APPUT and APGET. This is done just as if the AP were present. The simulator also simulates program source loading and management via the subroutine APEX.

5.1.3 HOST/AP SYNCHRONIZATION

Since both loading and executing the AP are simulated, MATHSIM also simulates the synchronization by calls to the APEX subroutines APWD, APWR, and APWAIT. When these calls are omitted, a synchronization warning is tallied, but execution continues.

5.1.4 TIMING ESTIMATES

MATHSIM estimates three of the system times: AP program execution time, DMA time, and AP executive (host overhead) time. The simulator does not account for any overlapping of these functions.

5.2 DETAILED DESCRIPTION

This section presents a detailed description of the various features of MATHSIM.

5.2.1 MATHSIM ROUTINES

MATHSIM contains most of the FORTRAN callable routines in the basic AP Math Library and the Signal Processing Library. The APEX routines described in this manual are included in MATHSIM. The routines supported by MATHSIM are listed in Table 5-1 and 5-2.

Table 5-1 MATHSIM APEX Routines

NAME	OPERATION
APASGN	Assign AP
APCHK	Check AP program error condition
APCLR	Initialize the AP
APEX	Program source executive
APGET	Get data from the AP
APGSP	Read an AP s-pad register
APINIT	To assign an AP and initialize APEX
APPUT	Put data into the AP
APRLSE	Release AP
APSTAT	Get AP hardware status
APSTOP	Pause on AP fatal error
APWAIT	Wait for AP
APWD	Wait for DMA and error check
APWR	Wait for AP run complete and error check
APXCLR	Clear APEX tables
APXSET	Initialize APEX and reset AP
ILOC	Find address of variable

0866

Table 5-2 MATHSIM Math Library Routines

NAME	OPERATION
ACORF	Auto-correlation (frequency-domain)
ACORT	Auto-correlation (time-domain)
ASPEC	Accumulating auto-spectrum
CCORF	Cross-correlation (frequency-domain)
CCORT	Cross-correlation (time-domain)
CDOTPR	Complex vector dot product
CFFT	Complex to comple FFT (inplace)
CFFTB	Complex to complex FFT (not in place)
CFFTSC	Complex FFT scale
COHER	Coherence function
CONV	Convolution (correlation)
CRVADD	Complex and real vector add
CRVDIV	Complex and real vector divide
CRVMUL	Complex and real vector multiply
CRVSUB	Complex and real vector subtract
CSPEC	Accumulating cross-spectrum
CTRN3	3-Dimension coordinate transformation
CVADD	Complex vector add
CVCOMB	Complex vector combine
CVCONJ	Complex vector conjugate
CVEXP	Complex exponential
CVFILL	Complex vector fill
CVMA	Complex vector multiply and add

0867

Table 5-2 MATHSIM Math Library Routines (cont.)

NAME	OPERATION
CVMAGS	Complex vector magnitude squared
CVMEXP	Vector multiply complex exponential
CVMOV	Complex vector move
CVMUL	Complex vector multiply
CVNEG	Complex vector negate
CVRCIP	Complex vector reciprocal
CVREAL	Form complex vector of reals
CVSMUL	Complex vector scalar multiply
CVSUB	Complex vector subtract
DEQ22	Difference equation, 2 poles, 2 zeros
DOTPR	Dot product
FMMM	Fast memory matrix multiply
FMMM32	Fast memory matrix mult (dim 32 or less)
HANN	Hanning window multiply
HIST	Histogram
LVEG	Logical vector equal
LVGE	Logical vector greater or equal
LVGT	Logical vector greater than
LVNE	Logical vector not equal
LVNOT	Logical vector not
MATINV	Matrix inverse
MAXMGV	Maximum magnitude element in vector
MAXV	Maximum element in vector

0868

Table 5-2 MATHSIM Math Library Routines (cont.)

NAME	OPERATION
MEAMGV	Mean of vector element magnitudes
MEANV	Mean value of vector elements
MEASQV	Mean of vector element squares
MINMGV	Minimum magnitude element in vector
MINV	Minimum element in vector
MMUL	Matrix multiply
MMUL32	Matrix multiply (dim 32 or less)
MTHSIM	FORTRAN simulation of APMATH
MTRANS	Matrix transpose
MVML3	Matrix vector multiply (3x3)
MVML4	Matrix vector multiply (4x4)
POLAR	Rectangular to polar conversion
RECT	Polar to rectangular conversion
RFFT	Real to complex FFT (in place)
RFFTB	Real to complex FFT (not in place)
RFFTSC	Read FFT scale and format
RMSQV	Root-mean-square of vector elements
SCJMA	Self-conjugate multiply and add
SOLVEQ	Linear equation solver
SVE	Sum of vector elements
SVEMG	Sum of vector element magnitudes
SVESQ	Sum of vector element squares
SVS	Sum of vector signed squares

0869

Table 5-2 MATHSIM Math Library Routines (cont.)

NAME	OPERATION
TCONV	Posttapered convolution (correlation)
TRANS	Transfer function
VAAM	Vector add, add, and multiply
VABS	Vector absolute value
VADD	Vector add
VALOG	Vector antilogarithm (base 10)
VAM	Vector add and multiply
VATAN	Vector arctangent
VATN2	Vector arctangent of y/x
VAVEXP	Vector exponential averaging
VAVLIN	Vector linear averaging
VCLIP	Vector clip
VCLR	Vector clear
VCOS	Vector cosine
VDBPWR	Vector conversion to DB (power)
VDIV	Vector divide
VEXP	Vector exponential
VFILL	Vector fill
VFIX	Vector integer fix
VFLT	Vector integer float
VFRAC	Vector truncate to fraction
VICLIP	Vector inverted clip
VIMAG	Extract imaginaries of complex vector

0870

Table 5-2 MATHSIM Math Library Routines (cont.)

NAME	OPERATION
VINDEX	Vector index
VINT	Vector truncate to integer
VLIM	Vector limit
VLMERG	Vector logical merge
VLN	Vector natural logarithm
VLOG	Vector logarithm (base 10)
VMA	Vector multiply and add
VMAX	Vector maximum
VMAXMG	Vector maximum magnitude
VMIN	Vector minimum
VMINMG	Vector minimum magnitude
VMMA	Vector multiply, multiply, and add
VMMSB	Vector multiply, multiply, and subtract
VMOV	Vector move
VMSA	Vector multiply and scalar add
VMSB	Vector multiply and subtract
VMUL	Vector multiply
VNEG	Vector negate
VPOLY	Vector polynomial
VRAMP	Vector ramp
VRAND	Vector random numbers
VREAL	Vector reals of complex vector
VSADD	Vector scalar add

0871

Table 5-2 MATHSIM Math Library Routines (cont.)

NAME	OPERATION
VSBM	Vector subtract and multiply
VSBSBM	Vector subtract, subtract, and multiply
VSCALE	Vector scale (power 2) and fix
VSCSCL	Vector scan, scale (power 2) and fix
VSHFX	Vector shift and fix
VSIMPS	Vector Simpsons 1/3 rule integration
VSIN	Vector sine
VSMA	Vector scalar multiply and add
VSMSA	Vector scalar multiply and scalar add
VSMSB	Vector scalar multiply and subtract
VSMUL	Vector scalar multiply
VSQ	Vector square
VSQRT	Vector square root
VSSQ	Vector signed square
VSUB	Vector subtract
VSUM	Vector sum of elements integration
VSWAP	Vector swap
VTRAPZ	Vector trapezoidal rule integration
WIENER	Wiener Levinson algorithm
ZMD	Clear all main data memory

0872

MATHSIM does not include routines which relate to byte packing and unpacking, vector logical operations, and support of table memory. The routines in the basic AP Math Library and the Signal Processing Library which are not included in MATHSIM are:

VAND
VEQV
VOR
VTSMUL
VUP8
VUPS8
VPK8
VUP16
VUPS16
VPK16
VFLT32
VFIX32

Neither does MATHSIM support APAL callable utility routines, such as DIV and SAVESP.

The following libraries are not supported by MATHSIM:

TMRAM library
Page select/parity library
IOP library
PIOP library

5.2.2 DMA

AP main data (MD) is simulated by the array variable APMD. APMD is communicated to the MATHSIM routines by the COMMON block:

```
COMMON /COMMD/ APMD(1024)
```

APPUT transfers the data taken from an array defined in the user program into APMD. APGET transfers data taken from APMD into an array defined in the user program. Execution of the MATHSIM routines requires a number of pointers in the APMD scratch space. These pointers are based on the subroutine call parameters.

MATHSIM supports APPUT and APGET format types 1 (16-bit integer) and 2 (host floating-point).

Via APEX, MATHSIM handles program source management exactly as it is handled during actual AP use. Thus, it is possible to encounter program souce overflow, which causes the run to halt. Note that PS size is set in the subroutine APXSET, PSSIZ = 1024. This size can be changed by the user.

5.2.3 SYNCHRONIZATION

MATHSIM simulates synchronization by calls to the APEX subroutines APWD, APWR and APWAIT. The variables involved in sensing a possible synchronization error are communicated to the necessary subroutines via the following:

```
INTEGER ERRFLG
LOGICAL DMAFLG, RUNFLG, INIFLG
COMMON /FLAGS/ DMAFLG,RUNFLG,INIFLG,ERRFLG
```

These statements appear in the following five subroutines:

APPUT
APGET
APEX
APTIME
APCLR

All flags are initialized by a call to APCLR. INIFLG is set to .FALSE. at compile time (in APCLR). Whenever a call to subroutine APEX is made before a call to APCLR or APINIT, the run halts and an error message is issued.

Whenever the user omits an APWD or APWR in the program, a synchronization warning is tallied (ERRFLG = ERRFLG +1) and the run continues. Omission of an APWAIT for either case causes the same results. It is important to note that some of this type of errors may escape detection. Also, the detection of possible errors does not necessarily mean that the program is invalid, as in cases where omission of calls to the waiting routines actually produces a more efficient program.

MATHSIM checks synchronization in the following ways:

- When a DMA is initiated, the DMAFLG is set; MATHSIM checks the RUNFLG; when the RUNFLG is set, MATHSIM tallies a synchronization warning.

- When an AP subroutine is executed, the RUNFLG is set; MATHSIM checks the DMAFLG; when the DMAFLG is set, MATHSIM tallies a synchronization warning.

- When a call is made to APWD, the DMAFLG is turned off; when a call is made to APWR, the RUNFLG is turned off; when a call is made to APWAIT, both the DMAFLG and the RUNFLG are turned off.

- MATHSIM considers a call to APTIME as host execution. When a call is issued to APTIME, the simulator checks the DMAFLG and the RUNFLG. If either of these is set, MATHSIM tallies a synchronization warning.

- When a host program is executing with either the DMAFLG or the RUNFLG set, but no call is made to APTIME, MATHSIM does not detect the possible error.

5.2.4 TIMING ESTIMATES

The timing accumulators are communicated to the various MATHSIM routines via the following:

```
COMMON /TIMING/ MTYPE,CONAPX,CONDMA,TIMRUN,TIMAPX,TIMDMA
```

This statement is in all of the algorithm-simulating routines and in four APEX subroutines: APEX, APPUT, APGET, and APCLR. The accumulators are initialized in APCLR. The following sections define and describe the accumulators.

5.2.4.1 TIMRUN

TIMRUN accumulates the AP program execution time estimates. MATHSIM estimates loop times for routines represented by a single loop by using the TYPICAL loop times given in Appendix D, plus the SETUP times. The SETUP times depend upon memory type which is designated in MATHSIM by the variable MTYPE in the preceding COMMON statement. These particular programs contain the following statement:

```
DATA SETUP(1), ....
```

Other routines with more complex algorithms contain timing formulas of an empirical nature derived from the timing tables in Appendix E. The formulas are established so that extrapolation out of the range of the tables gives reasonably accurate timing estimates. Interpolations within the range of the tables give an accuracy well within five percent.

5.2.4.2 TIMAPX

TIMAPX accumulates the estimates for the AP executive (host overhead) time. All algorithm-simulating routines contain an integer variable SIZE, usually dependent upon MTYPE which contains the AP program source word length. Each time a particular program is called, it executes the following statement:

```
CALL APEX (SIZE)  or  CALL APEX (SIZE(MTYPE))
```

The subroutine APEX checks to see whether or not the subroutine is loaded. If the program is designated as already AP-resident, MATHSIM adds an estimate of the system overhead time onto TIMAPX. The value of this estimate is a constant assigned to CONAPX in MATHSIM. When the program is not resident in the AP, simulation is affected by estimating the time needed to load a program of the specified SIZE. This estimate is based on the constant CONDMA.

Because it is difficult to arrive at a precise value for CONAPX, TIMAPX can be only an order of magnitude estimate when CONAPX is involved in its calculation. However, program loading times are more accurate because CONDMA can be established more precisely.

Note that the times for some applications are dominated by TIMAPX. This is true for many calls to simple vector manipulations. Using the vector function chainer reduces multiple calls to Math Library routines to a single call, and thus reduces system overhead. MATHSIM, however, does not simulate the vector function chainer software. Therefore, the user must subtract the host overhead estimate (CONAPX) an appropriate number of times from the total timing estimate to account for use of vector function chaining.

5.3 SYSTEM DEPENDENCY

MATHSIM contains several installation-dependent features, as listed below.

- The subroutine APEX calls the FORTRAN function ILOC(X), which returns the location of the variable X. This function is FPS-supplied.

- The subroutine APCLR assigns the following timing parameters:

  MYTPE = n
  CONDMA = ss
  CONAPX = tt

  where:

  n is the memory type; 1 = fast, 2 = standard (default = 2)

  ss is the time per 16-bit word transfer (u sec) (default = 2 u sec)

  tt is the time per s-pad load and go (u sec) (default = 1000 u sec)

- The subroutine APXSET assigns the program source size with the statement:

```
PSSIZ = n
```

  where:

  n is the program source size (default = 1024)

- The main data size is indicated in the following statement:

```
COMMON /COMMD/ APMD (1024)
```

  It can be changed by changing the entry in every occurrence of this statement; the statement appears in all Math Library routines and in the two APEX routines APPUT and APGET.

- The MATHSIM routines APSTOP and APEX write messages to the unit specified in the following statement:

```
IOUNIT = n
```

  where:

  n is the logical unit number (default = 1)

5.4 EXAMPLES

This section contains a sample MATHSIM routine and two programming examples applicable to MATHSIM.

5.4.1 EXAMPLE 1: SAMPLE ROUTINE

The following VADD routine shown is typical of the routines provided in MATHSIM:

```
C***** VADD = VECTOR ADD = REL 1.0, MAY 78 *****
        SUBROUTINE VADD(A,I,B,J,C,K,N)
        INTEGER*2 A,I,B,J,C,K,N
        INTEGER*2 M,IA,IB,IC
      REAL SETUP(2), LOOP(2)
      INTEGER SIZE(2)
      COMMON /TIMING/ MTYPE,CONAPX,CONDMA,TIMRUN,TIMAPX,
     X TIMDMA
        COMMON /COMMD/APMD(1024)
      DATA SETUP(1), SETUP(2), LOOP(1), LOOP(2)
     X    /   2.67 ,    1.17 ,    0.84 ,    1.33 /
      DATA  SIZE(1), SIZE(2) / 17 ,  8 /
        IA=A+1
        IB=B+1
        IC=C+1
        DO 100 M=1,N
        APMD(IC)=APMD(IB)+APMD(IA)
        IA=IA+I
        IB=IB+J
100     IC=IC+K
C
C  /*  TIMING.  */
       CALL APEX (SIZE(MTYPE))
       TIMRUN = TIMRUN + SETUP(MTYPE) + FLOAT(N) * LOOP(MTYPE)
         RETURN
         END
```

Note that the user has access to the timing information by including the statement COMMON /TIMING/ ... in the calling program. Also, because MTYPE is assigned in the call to APCLR, the user may change the memory type after that call. This allows the user to easily obtain timing for either standard or fast memory.

5.4.2 EXAMPLE 2: PROGRAM FOR VECTOR ADDITION

The following is an example of a calling program to add two vectors; this version can be used with either the AP or MATHSIM.

```
      DIMENSION A(100),B(100),C(100)
         ,
         ,
C    /* TIMING INFO WRITTEN ON LOGICAL UNIT 'IOUNIT'..  */
     IOUNIT = 1
C    /* APCLR INITIALIZES TIMING (AMONG OTHER THINGS). */
      CALL APCLR
      CALL APPUT(A,0,100,2)
      CALL APPUT(B,100,100,2)
      CALL APWD
      CALL VADD(0,1,100,1,200,1,100)
      CALL APWR
      CALL APGET(C,200,100,2)
C    /* WRITE ACCUMULATED TIMES...  */
     CALL APTIME (IOUNIT)
         ,
         ,
      END
```

5.4.3 EXAMPLE 3: PROGRAM FOR VECTOR ADDITION WITH TIMING ESTIMATES

The following is an example of a program to add two vectors and determine timing estimates. This program is written for use with MATHSIM only.

```
         DIMENSION A(100),B(100),C(100)
           ,
           ,
C       /*  TIMING INFO WRITTEN ON LOGICAL UNIT 'IOUNIT'..   */
        IOUNIT = 1
C       /*  APCLR INITIALIZES TIMING (AMONG OTHER THINGS). */
         CALL APCLR
         CALL APPUT(A,0,100,2)
         CALL APPUT(B,100,100,2)
         CALL APWD
         CALL VADD(0,1,100,1,200,1,100)
         CALL APWR
         CALL APGET(C,200,100,2)
C       /*  WRITE ACCUMULATED TIMES...   */
        CALL APTIME (IOUNIT)
           ,
           ,
         END
```

When this program is run with the parameters set to the default values, the subroutine APTIME displays the following message:

```
AP-120B TIMING ESTIMATES (ACCUM).
     RUN  =      0.13417 (MSEC)
     APEX =      1.06400
     DMA  =      1.20000
   SYNCHRONIZATION WARNINGS =  1
```

To eliminate the synchronization warning, the user should insert a CALL APWD after the CALL APGET statement.

APPENDIX A ALPHABETICAL INDEX OF AP MATH LIBRARY ROUTINES

Page	Name	Operation	Typical Execution Time/Loop (us) 167	333	Program Size (AP PS words) 167	333
E-195	ACORF	AUTO-CORRELATION (FREQUENCY-DOMAIN)	1.80*	2.70	501	489
E-193	ACORT	AUTO-CORRELATION (TIME-DOMAIN)	0.29*	0.29	121	121
E-270	ADV2	ADVANCE POINTERS AFTER RADIX 2 FFT	0.7 @	0.7	7	7
E-271	ADV4	ADVANCE POINTERS AFTER RADIX 4 FFT	0.7 @	0.7	7	7
E-13	APCHK	CHECK AP PROGRAM ERROR CONDITION	#.#	#.#	0	0
E-8	APCLR	INITIALIZE THE AP	#.#	#.#	0	0
E-6	APGET	GET DATA FROM THE AP	#.#	#.#	0	0
E-12	APGSP	READ AN AP S-PAD REGISTER	#.#	#.#	0	0
E-4	APPUT	PUT DATA INTO THE AP	#.#	#.#	0	0
E-14	APSTAT	GET AP HARDWARE STATUS	#.#	#.#	0	0
E-11	APWAIT	WAIT FOR AP	#.#	#.#	0	0
E-9	APWD	WAIT FOR AP DATA TRANSFER	#.#	#.#	0	0
E-10	APWR	WAIT FOR AP PROGRAM EXECUTION	#.#	#.#	0	0
E-186	ASPEC	ACCUMULATING AUTO-SPECTRUM	0.8	1.5	21	22
E-234	ATAN	SCALAR ARCTANGENT	8.7 @	8.7	74	74
E-235	ATN2	SCALAR ARCTANGENT OF Y/X	13.8 @	13.8	74	74
E-261	BITREV	COMPLEX VECTOR BIT REVERSE ORDERING	0.9	1.4	45	43
E-199	CCORF	CROSS-CORRELATION (FREQUENCY-DOMAIN)	2.58*	3.93	526	510
E-197	CCORT	CROSS-CORRELATION (TIME-DOMAIN)	0.29*	0.29	121	121
E-115	CDOTPR	COMPLEX DOT PRODUCT	0.7	1.3	15	16
E-156	CFFT	COMPLEX TO COMPLEX FFT (IN PLACE)	0.28*	0.40	186	184
E-167	CFFT2D	COMPLEX TO COMPLEX 2-DIMENSIONAL FFT	0.5 *	0.5	274	274
E-158	CFFTB	COMPLEX TO COMPLEX FFT (NOT IN PLACE)	0.20*	0.28	189	189
E-164	CFFTSC	COMPLEX FFT SCALE	0.8	1.3	42	42
E-268	CLSTAT	CLEAR FFT MODE STATUS BITS	0.5 @	0.5	19	19
E-192	COHER	COHERENCE FUNCTION	4.0	4.5	109	114
E-172	CONV	CONVOLUTION (CORRELATION)	0.28*	0.28	106	106
E-233	COS	SCALAR COSINE	5.4 @	5.4	35	35
E-103	CRVADD	COMPLEX AND REAL VECTOR ADD	1.3	1.8	14	14
E-106	CRVDIV	COMPLEX AND REAL VECTOR DIVIDE	3.3	3.3	92	92
E-105	CRVMUL	COMPLEX AND REAL VECTOR MULTIPLY	1.3	1.8	14	14
E-104	CRVSUB	COMPLEX AND REAL VECTOR SUBTRACT	1.3	1.8	14	14
E-187	CSPEC	ACCUMULATING CROSS-SPECTRUM	1.3	2.7	39	40
E-281	CTOR	COMPLEX TO REAL FFT UNSCRAMBLE	0.13*	0.13	80	80
E-149	CTRN3	3-DIMENSION COORDINATE TRANSFORMATION	2.3 *	2.5	37	37
E-98	CVADD	COMPLEX VECTOR ADD	1.0	2.0	13	12
E-92	CVCOMB	COMPLEX VECTOR COMBINE	1.1	1.7	10	10
E-97	CVCONJ	COMPLEX VECTOR CONJUGATE	0.7	1.3	10	12
E-113	CVEXP	COMPLEX VECTOR EXPONENTIAL	2.0	2.0	43	43
E-91	CVFILL	COMPLEX VECTOR FILL	0.5	0.7	8	8
E-107	CVMA	COMPLEX VECTOR MULTIPLY AND ADD	1.3	2.7	29	30
E-109	CVMAGS	COMPLEX VECTOR MAGNITUDE SQUARED	0.7	1.2	13	18
E-114	CVMEXP	VECTOR MULTIPLY COMPLEX EXPONENTIAL	2.3	2.3	48	48
E-90	CVMOV	COMPLEX VECTOR MOVE	0.8	1.3	9	9
E-100	CVMUL	COMPLEX VECTOR MULTIPLY	1.0	2.0	25	26
E-96	CVNEG	COMPLEX VECTOR NEGATE	0.8	1.3	11	11

Page	Name	Operation	Typical Execution Time/Loop (us) 167	333	Program Size (AP PS words) 167	333
E-102	CVRCIP	COMPLEX VECTOR RECIPROCAL	5.2	5.2	50	50
E-93	CVREAL	FORM COMPLEX VECTOR OF REALS	0.8	1.2	9	9
E-101	CVSMUL	COMPLEX VECTOR SCALAR MULTIPLY	0.8	1.3	12	12
E-99	CVSUB	COMPLEX VECTOR SUBTRACT	1.0	2.0	13	12
E-257	DAREAD	READ DEVICE ADDRESS REGISTER	0.3 @	0.3	2	2
E-258	DAWRIT	WRITE DEVICE ADDRESS REGISTER	0.3 @	0.3	2	2
E-287	DDDA	DOUBLE + DOUBLE TO DOUBLE ADD	7.5 @	7.5	48	48
E-288	DDDM	DOUBLE * DOUBLE TO DOUBLE MULTIPLY	18.5 @	18.5	117	117
E-174	DEQ22	DIFFERENCE EQUATION, 2 POLES, 2 ZEROS	0.8	0.8	25	25
E-227	DIV	SCALAR DIVIDE	3.8 @	3.8	28	28
E-66	DOTPR	DOT PRODUCT	0.5	0.8	21	9
E-231	EXP	SCALAR EXPONENTIAL	4.2 @	4.2	28	28
E-263	FFT2	RADIX 2 FFT FIRST PASS	1.3	2.7	16	16
E-265	FFT2B	RADIX 2 FFT FIRST PASS + BIT REVERSE	1.3	2.7	25	25
E-264	FFT4	RADIX 4 FFT PASS	3.7	5.3	79	79
E-266	FFT4B	RADIX 4 FFT FIRST PASS + BIT REVERSE	2.7	5.3	43	43
E-151	FMMM	FAST MEMORY MATRIX MULTIPLY	0.43*		61	
E-153	FMMM32	FAST MEMORY MATRIX MULTIPLY (<=32)	0.41*		33	
E-184	HANN	HANNING WINDOW MULTIPLY	0.7	0.8	41	41
E-183	HIST	HISTOGRAM	1.3	1.4	71	71
E-269	ILOG2	LOGARITHM (BASE 2)	4.0 @	4.0	19	19
E-230	LN	SCALAR NATURAL LOGARITHM	4.0 @	4.0	37	37
E-229	LOG	SCALAR LOGARITHM (BASE 10)	4.7 @	4.7	37	37
E-85	LVEQ	LOGICAL VECTOR EQUAL	0.8	1.3	23	13
E-84	LVGE	LOGICAL VECTORGREATER THAN OR EQUAL	0.8	1.3	23	13
E-83	LVGT	LOGICAL VECTOR GREATER THAN	0.8	1.3	23	13
E-86	LVNE	LOGICAL VECTOR NOT EQUAL	0.8	1.3	23	13
E-87	LVNOT	LOGICAL VECTOR NOT	0.5	0.8	21	12
E-141	MATINV	MATRIX INVERSE	1.6 *	2.1	160	160
E-69	MAXMGV	MAXIMUM MAGNITUDE ELEMENT IN VECTOR	0.3	0.3	19	19
E-67	MAXV	MAXIMUM ELEMENT IN VECTOR	0.3	0.3	19	19
E-253	MDCOM	MAIN DATA COMPARE AND SET S-PAD	1.8 @	2.0	11	11
E-72	MEAMGV	MEAN OF VECTOR ELEMENT MAGNITUDES	0.3	0.3	52	52
E-71	MEANV	MEAN VALUE OF VECTOR ELEMENTS	0.3	0.3	49	49
E-73	MEASQV	MEAN OF VECTOR ELEMENT SQUARES	0.3	0.3	52	52
E-70	MINMGV	MINIMUM MAGNITUDE ELEMENT IN VECTOR	0.3	0.3	19	19
E-68	MINV	MINIMUM ELEMENT IN VECTOR	0.3	0.3	19	19
E-209	MMTADD	VECTOR ADD (MD+MD TO TM)	0.7	0.8	20	13
E-211	MMTMUL	VECTOR MULTIPLY (MD*MD TO TM)	0.7	0.8	20	13
E-210	MMTSUB	VECTOR SUBTRACT (MD-MD TO TM)	0.7	0.8	20	13
E-137	MMUL	MATRIX MULTIPLY	0.62*	0.83	59	59
E-139	MMUL32	MATRIX MULTIPLY (DIMENSION <=32)	0.50*	0.73	27	27
E-206	MTIMOV	VECTOR MOVE WITH INCREMENT (MD TO TM)	0.5	0.5	7	7
E-212	MTMADD	VECTOR ADD (MD+TM TO MD)	0.5	0.8	20	9
E-215	MTMMUL	VECTOR MULTIPLY (MD*TM TO MD)	0.5	0.8	20	9
E-204	MTMOV	VECTOR MOVE (MD TO TM)	0.2	0.3	6	7
E-213	MTMSUB	VECTOR SUBTRACT (MD-TM TO MD)	0.5	0.8	20	9
E-136	MTRANS	MATRIX TRANSPOSE	0.5	0.9	18	22
E-216	MTTADD	VECTOR ADD (MD+TM TO TM)	0.5	0.5	20	20
E-219	MTTMUL	VECTOR MULTIPLY (MD*TM TO TM)	0.5	0.5	20	20

Page	Name	Operation	Typical Execution Time/Loop (us) 167	333	Program Size (AP PS words) 167	333
E-217	MTTSUB	VECTOR SUBTRACT (MD-TM TO TM)	0.5	0.5	20	20
E-145	MVML3	MATRIX VECTOR MULTIPLY (3X3)	2.0 *	2.2	30	30
E-147	MVML4	MATRIX VECTOR MULTIPLY (4X4)	3.3 *	3.8	39	39
E-279	PCFFT	PARTIAL COMPLEX FFT	1.05*	1.50	117	117
E-111	POLAR	RECTANGULAR TO POLAR CONVERSION	19.5	19.5	120	120
E-255	RDC5	READ CONTROL BIT 5 INTERRUPT	1.5 @	1.5	9	9
E-262	REALTR	REAL FFT UNRAVEL AND FINAL PASS	0.4	0.7	68	68
E-112	RECT	POLAR TO RECTANGULAR CONVERSION	2.3	2.3	49	49
E-160	RFFT	REAL TO COMPLEX FFT (IN PLACE)	0.18*	0.27	253	251
E-169	RFFT2D	REAL TO COMPLEX 2-DIMENSIONAL FFT	0.4 *	0.4	585	585
E-162	RFFTB	REAL TO COMPLEX FFT (NOT IN PLACE)	0.14*	0.20	252	252
E-165	RFFTSC	REAL FFT SCALE AND FORMAT	0.7	0.8	59	59
E-74	RMSQV	ROOT-MEAN-SQUARE OF VECTOR ELEMENTS	0.3	0.3	81	81
E-282	RTOC	REAL TO COMPLEX FFT SCRAMBLE	0.09*	0.09	143	143
E-248	SAVESP	SAVE S-PAD INTO PROGRAM MEMORY	0.8 *	0.8	18	18
E-249	SAVSP0	SAVE S-PAD 0 INTO PROGRAM MEMORY	2.0 *	2.0	11	11
E-110	SCJMA	SELF-CONJUGATE MULTIPLY AND ADD	0.8	1.5	14	15
E-286	SDDA	SINGLE + DOUBLE TO DOUBLE ADD	4.5 @	4.5	28	28
E-272	SET24B	SETUP FOR FFT2B AND FFT4B	1.2 @	1.2	8	8
E-252	SET2SP	LOAD 2 S-PADS FROM PROGRAM MEMORY	5.7 @	5.7	33	33
E-256	SETC5	SET CONTROL BIT 5 INTERRUPT	0.2 @	0.2	1	1
E-250	SETSP	LOAD S-PADS FROM PROGRAM MEMORY	2.3 *	2.3	33	33
E-232	SIN	SCALAR SINE	4.9 @	4.9	35	35
E-143	SOLVEQ	LINEAR EQUATION SOLVER	0.7 *	0.9	216	222
E-239	SPADD	S-PAD ADD	0.2 @	0.2	1	1
E-245	SPAND	S-PAD AND	0.2 @	0.2	1	1
E-242	SPDIV	S-PAD DIVIDE	6.2 @	6.2	43	43
E-236	SPFLT	FLOAT S-PAD INTEGER	0.8 @	0.8	5	5
E-244	SPLS	S-PAD LEFT SHIFT	0.3 *	0.3	5	5
E-241	SPMUL	S-PAD MULTIPLY	2.3 @	2.3	14	14
E-238	SPNEG	S-PAD NEGATE	0.3 @	0.3	2	2
E-247	SPNOT	S-PAD NOT	0.2 @	0.2	1	1
E-246	SPOR	S-PAD OR	0.2 @	0.2	1	1
E-243	SPRS	S-PAD RIGHT SHIFT	0.3 *	0.3	5	5
E-240	SPSUB	S-PAD SUBTRACT	0.2 @	0.2	1	1
E-237	SPUFLT	S-PAD UNSIGNED FLOAT	0.8 @	0.8	8	8
E-228	SQRT	SCALAR SQUARE ROOT	3.8 @	3.8	28	28
E-284	SSDA	SINGLE + SINGLE TO DOUBLE ADD	1.5 @	1.5	10	10
E-285	SSDM	SINGLE * SINGLE TO DOUBLE MULTIPLY	11.5 @	11.5	81	81
E-267	STSTAT	SET FFT MODE STATUS BITS	5.0 @	5.0	19	19
E-62	SVE	SUM OF VECTOR ELEMENTS	0.3	0.3	7	7
E-63	SVEMG	SUM OF VECTOR ELEMENT MAGNITUDES	0.3	0.3	10	10
E-64	SVESQ	SUM OF VECTOR ELEMENT SQUARES	0.3	0.3	10	10
E-65	SVS	SUM OF VECTOR SIGNED SQUARES	0.3	0.3	11	11
E-201	TCONV	POSTTAPERED CONVOLUTION (CORRELATION)	0.30*	0.30	112	112
E-207	TMIMOV	VECTOR MOVE WITH INCREMENT (TM TO MD)	0.3	0.3	15	15
E-205	TMMOV	VECTOR MOVE (TM TO MD)	0.2	0.3	5	5
E-214	TMMSUB	VECTOR SUBTRACT (TM-MD TO MD)	0.5	0.8	20	9
E-218	TMTSUB	VECTOR SUBTRACT (TM-MD TO TM)	0.5	0.5	20	20
E-191	TRANS	TRANSFER FUNCTION	3.3	3.3	100	100

Page	Name	Operation	Typical Execution Time/Loop (us) 167	333	Program Size (AP PS words) 167	333
E-208	TTIMOV	VECTOR MOVE WITH INCREMENT (TM TO TM)	0.5	0.5	7	7
E-220	TTMADD	VECTOR ADD (TM+TM TO MD)	0.5	0.5	20	20
E-222	TTMMUL	VECTOR MULTIPLY (TM*TM TO MD)	0.5	0.5	20	20
E-221	TTMSUB	VECTOR SUBTRACT (TM-TM TO MD)	0.5	0.5	20	20
E-223	TTTADD	VECTOR ADD (TM+TM TO TM)	0.7	0.7	9	9
E-225	TTTMUL	VECTOR MULTIPLY (TM*TM TO TM)	0.7	0.7	10	10
E-224	TTTSUB	VECTOR SUBTRACT (TM-TM TO TM)	0.7	0.7	9	9
E-53	VAAM	VECTOR ADD, ADD, AND MULTIPLY	1.5	2.3	13	20
E-32	VABS	VECTOR ABSOLUTE VALUE	0.5	0.8	17	7
E-22	VADD	VECTOR ADD	0.8	1.3	20	8
E-36	VALOG	VECTOR ANTILOGARITHM (BASE 10)	2.3	2.3	58	58
E-48	VAM	VECTOR ADD AND MULTIPLY	1.2	1.8	23	14
E-55	VAND	VECTOR LOGICAL AND	0.8	1.3	20	8
E-40	VATAN	VECTOR ARCTANGENT	9.7	9.8	87	87
E-41	VATN2	VECTOR ARCTANGENT OF Y/X	14.2	14.2	88	88
E-189	VAVEXP	VECTOR EXPONENTIAL AVERAGING	0.8	1.3	55	46
E-188	VAVLIN	VECTOR LINEAR AVERAGING	0.8	1.3	54	46
E-80	VCLIP	VECTOR CLIP	0.5	0.8	16	16
E-16	VCLR	VECTOR CLEAR	0.2	0.3	16	4
E-39	VCOS	VECTOR COSINE	1.3	1.3	34	34
E-190	VDBPWR	VECTOR CONVERSION TO DB (POWER)	1.2	1.3	75	75
E-25	VDIV	VECTOR DIVIDE	1.7	1.7	75	75
E-56	VEQV	VECTOR LOGICAL EQUIVALENCE	0.8	1.3	20	8
E-37	VEXP	VECTOR EXPONENTIAL	2.3	2.3	55	55
E-259	VFCL1	VECTOR FUNCTION CALLER (1 ARGUMENT)	0.8	1.0	10	10
E-260	VFCL2	VECTOR FUNCTION CALLER (2 ARGUMENT)	1.0	1.0	11	11
E-19	VFILL	VECTOR FILL	0.3	0.3	5	5
E-118	VFIX	VECTOR INTEGER FIX	0.7	0.8	18	7
E-133	VFIX32	VECTOR 32-BIT INTEGER FIX	1.2	1.2	33	33
E-117	VFLT	VECTOR INTEGER FLOAT	0.5	0.8	13	11
E-132	VFLT32	VECTOR 32-BIT INTEGER FLOAT	1.7	1.7	65	65
E-58	VFRAC	VECTOR TRUNCATE TO FRACTION	0.7	0.8	13	13
E-81	VICLIP	VECTOR INVERTED CLIP	0.7	0.8	19	19
E-95	VIMAG	EXTRACT IMAGINARIES OF COMPLEX VECTOR	0.5	0.8	18	8
E-60	VINDEX	VECTOR INDEX	0.8	1.3	28	26
E-59	VINT	VECTOR TRUNCATE TO INTEGER	0.5	0.8	9	9
E-82	VLIM	VECTOR LIMIT	0.5	0.8	14	14
E-88	VLMERG	VECTOR LOGICAL MERGE	0.8	1.5	23	16
E-35	VLN	VECTOR NATURAL LOGARITHM	2.7	2.7	42	42
E-34	VLOG	VECTOR LOGARITHM (BASE 10)	2.7	2.7	54	58
E-46	VMA	VECTOR MULTIPLY AND ADD	1.2	1.8	23	15
E-76	VMAX	VECTOR MAXIMUM	0.8	1.3	22	13
E-78	VMAXMG	VECTOR MAXIMUM MAGNITUDE	0.8	1.3	14	14
E-77	VMIN	VECTOR MINIMUM	0.8	1.3	22	13
E-79	VMINMG	VECTOR MINIMUM MAGNITUDE	0.8	1.3	14	14
E-51	VMMA	VECTOR MULTIPLY, MULTIPLY, AND ADD	1.5	2.3	27	19
E-52	VMMSB	VECTOR MULTIPLY MULTIPLY AND SUBTRACT	1.5	2.3	27	19
E-17	VMOV	VECTOR MOVE	0.5	0.8	16	6
E-43	VMSA	VECTOR MULTIPLY AND SCALAR ADD	0.8	1.3	23	14
E-47	VMSB	VECTOR MULTIPLY AND SUBTRACT	1.2	1.8	23	15

Page	Name	Operation	Typical Execution Time/Loop (us) 167	333	Program Size (AP PS words) 167	333
E-24	VMUL	VECTOR MULTIPLY	0.8	1.3	20	11
E-21	VNEG	VECTOR NEGATE	0.5	0.8	18	7
E-57	VOR	VECTOR LOGICAL OR	0.8	1.3	20	8
E-131	VPK16	VECTOR 16-BIT BYTE PACK	0.8	0.8	46	46
E-128	VPK8	VECTOR 8-BIT BYTE PACK	0.9	0.9	65	65
E-175	VPOLY	VECTOR POLYNOMIAL EVALUATION	1.0 *	1.2	41	41
E-20	VRAMP	VECTOR RAMP	0.3	0.3	12	12
E-42	VRAND	VECTOR RANDOM NUMBERS	1.2	1.2	16	16
E-94	VREAL	EXTRACT REALS OF COMPLEX VECTOR	0.5	0.8	17	7
E-26	VSADD	VECTOR SCALAR ADD	0.5	0.8	19	8
E-49	VSBM	VECTOR SUBTRACT AND MULTIPLY	1.2	1.8	23	14
E-54	VSBSBM	VECTOR SUBTRACT SUBTRACT AND MULTIPLY	1.5	2.3	13	20
E-121	VSCALE	VECTOR SCALE (POWER 2) AND FIX	0.7	0.8	12	12
E-123	VSCSCL	VECTOR SCAN, SCALE (POWER 2) AND FIX	1.5	1.7	19	19
E-134	VSEFLT	VECTOR SIGN EXTEND AND FLOAT	0.8	0.8	15	15
E-125	VSHFX	VECTOR SHIFT AND FIX	0.7	0.8	9	9
E-179	VSIMPS	VECTOR SIMPSONS 1/3 RULE INTEGRATION	0.7	0.8	25	25
E-38	VSIN	VECTOR SINE	1.3	1.3	34	34
E-44	VSMA	VECTOR SCALAR MULTIPLY AND ADD	0.8	1.3	21	14
E-120	VSMAFX	VECTOR SCALAR MULTIPLY, ADD, AND FIX	0.7	0.8	14	13
E-50	VSMSA	VECTOR SCALAR MULTIPLY AND SCALAR ADD	0.5	0.8	23	15
E-45	VSMSB	VECTOR SCALAR MULTIPLY AND SUBTRACT	0.8	1.3	21	14
E-27	VSMUL	VECTOR SCALAR MULTIPLY	0.5	0.8	20	9
E-30	VSQ	VECTOR SQUARE	0.5	0.8	9	9
E-33	VSQRT	VECTOR SQUARE ROOT	1.8	1.8	79	79
E-31	VSSQ	VECTOR SIGNED SQUARE	0.5	0.8	21	9
E-23	VSUB	VECTOR SUBTRACT	0.8	1.3	20	8
E-177	VSUM	VECTOR SUM OF ELEMENTS INTEGRATION	0.7	0.8	13	13
E-18	VSWAP	VECTOR SWAP	1.2	1.5	21	12
E-178	VTRAPZ	VECTOR TRAPEZOIDAL RULE INTEGRATION	0.7	0.8	16	16
E-28	VTSADD	VECTOR TABLE SCALAR ADD	0.5	0.8	8	8
E-29	VTSMUL	VECTOR TABLE SCALAR MULTIPLY	0.5	0.8	8	8
E-129	VUP16	VECTOR 16-BIT BYTE UNPACK	0.8	0.8	61	61
E-126	VUP8	VECTOR 8-BIT BYTE UNPACK	0.5	0.5	71	71
E-130	VUPS16	VECTOR 16-BIT SIGNED BYTE UNPACK	1.3	1.3	58	58
E-127	VUPS8	VECTOR 8-BIT SIGNED BYTE UNPACK	0.9	0.9	107	107
E-180	WIENER	WIENER LEVINSON ALGORITHM	0.50*	0.65	100	100
E-277	XBITRE	EXPANDED BIT REVERSE	3.7	3.7	44	44
E-273	XCFFT	EXPANDED COMPLEX FFT	0.32*	0.42	187	187
E-280	XFFT4	EXPANDED RADIX 4 FFT PASS	3.7	5.3	79	79
E-278	XREALT	EXPANDED REAL FFT FINAL PASS	0.4	0.7	71	71
E-275	XRFFT	EXPANDED REAL FFT	0.19*	0.28	256	256
E-254	ZMD	CLEAR ALL PAGES OF MAIN DATA MEMORY	0.2	0.3	29	29

Notes: #.# Timing host system dependent
* Refer to description of routine for explanation of timing
@ Total execution time

APPENDIX B INDEX OF ROUTINES IN PAGE ORDER

Page	Name	Operation	Typical Execution Time/Loop (us) 167	Typical Execution Time/Loop (us) 333	Program Size (AP PS words) 167	Program Size (AP PS words) 333
		DATA TRANSFER AND CONTROL OPERATIONS (APEX)				
E-4	APPUT	PUT DATA INTO THE AP	#.#	#.#	0	0
E-6	APGET	GET DATA FROM THE AP	#.#	#.#	0	0
E-8	APCLR	INITIALIZE THE AP	#.#	#.#	0	0
E-9	APWD	WAIT FOR AP DATA TRANSFER	#.#	#.#	0	0
E-10	APWR	WAIT FOR AP PROGRAM EXECUTION	#.#	#.#	0	0
E-11	APWAIT	WAIT FOR AP	#.#	#.#	0	0
E-12	APGSP	READ AN AP S-PAD REGISTER	#.#	#.#	0	0
E-13	APCHK	CHECK AP PROGRAM ERROR CONDITION	#.#	#.#	0	0
E-14	APSTAT	GET AP HARDWARE STATUS	#.#	#.#	0	0
		BASIC VECTOR ARITHMETIC				
E-16	VCLR	VECTOR CLEAR	0.2	0.3	16	4
E-17	VMOV	VECTOR MOVE	0.5	0.8	16	6
E-18	VSWAP	VECTOR SWAP	1.2	1.5	21	12
E-19	VFILL	VECTOR FILL	0.3	0.3	5	5
E-20	VRAMP	VECTOR RAMP	0.3	0.3	12	12
E-21	VNEG	VECTOR NEGATE	0.5	0.8	18	7
E-22	VADD	VECTOR ADD	0.8	1.3	20	8
E-23	VSUB	VECTOR SUBTRACT	0.8	1.3	20	8
E-24	VMUL	VECTOR MULTIPLY	0.8	1.3	20	11
E-25	VDIV	VECTOR DIVIDE	1.7	1.7	75	75
E-26	VSADD	VECTOR SCALAR ADD	0.5	0.8	19	8
E-27	VSMUL	VECTOR SCALAR MULTIPLY	0.5	0.8	20	9
E-28	VTSADD	VECTOR TABLE SCALAR ADD	0.5	0.8	8	8
E-29	VTSMUL	VECTOR TABLE SCALAR MULTIPLY	0.5	0.8	8	8
E-30	VSQ	VECTOR SQUARE	0.5	0.8	9	9
E-31	VSSQ	VECTOR SIGNED SQUARE	0.5	0.8	21	9
E-32	VABS	VECTOR ABSOLUTE VALUE	0.5	0.8	17	7
E-33	VSQRT	VECTOR SQUARE ROOT	1.8	1.8	79	79
E-34	VLOG	VECTOR LOGARITHM (BASE 10)	2.7	2.7	54	58
E-35	VLN	VECTOR NATURAL LOGARITHM	2.7	2.7	42	42
E-36	VALOG	VECTOR ANTILOGARITHM (BASE 10)	2.3	2.3	58	58
E-37	VEXP	VECTOR EXPONENTIAL	2.3	2.3	55	55
E-38	VSIN	VECTOR SINE	1.3	1.3	34	34
E-39	VCOS	VECTOR COSINE	1.3	1.3	34	34
E-40	VATAN	VECTOR ARCTANGENT	9.7	9.8	87	87
E-41	VATN2	VECTOR ARCTANGENT OF Y/X	14.2	14.2	88	88
E-42	VRAND	VECTOR RANDOM NUMBERS	1.2	1.2	16	16

Page	Name	Operation	Typical Execution Time/Loop (us) 167	333	Program Size (AP PS words) 167	333
E-43	VMSA	VECTOR MULTIPLY AND SCALAR ADD	0.8	1.3	23	14
E-44	VSMA	VECTOR SCALAR MULTIPLY AND ADD	0.8	1.3	21	14
E-45	VSMSB	VECTOR SCALAR MULTIPLY AND SUBTRACT	0.8	1.3	21	14
E-46	VMA	VECTOR MULTIPLY AND ADD	1.2	1.8	23	15
E-47	VMSB	VECTOR MULTIPLY AND SUBTRACT	1.2	1.8	23	15
E-48	VAM	VECTOR ADD AND MULTIPLY	1.2	1.8	23	14
E-49	VSBM	VECTOR SUBTRACT AND MULTIPLY	1.2	1.8	23	14
E-50	VSMSA	VECTOR SCALAR MULTIPLY AND SCALAR ADD	0.5	0.8	23	15
E-51	VMMA	VECTOR MULTIPLY, MULTIPLY, AND ADD	1.5	2.3	27	19
E-52	VMMSB	VECTOR MULTIPLY MULTIPLY AND SUBTRACT	1.5	2.3	27	19
E-53	VAAM	VECTOR ADD, ADD, AND MULTIPLY	1.5	2.3	13	20
E-54	VSBSBM	VECTOR SUBTRACT SUBTRACT AND MULTIPLY	1.5	2.3	13	20
E-55	VAND	VECTOR LOGICAL AND	0.8	1.3	20	8
E-56	VEQV	VECTOR LOGICAL EQUIVALENCE	0.8	1.3	20	8
E-57	VOR	VECTOR LOGICAL OR	0.8	1.3	20	8
E-58	VFRAC	VECTOR TRUNCATE TO FRACTION	0.7	0.8	13	13
E-59	VINT	VECTOR TRUNCATE TO INTEGER	0.5	0.8	9	9
E-60	VINDEX	VECTOR INDEX	0.8	1.3	28	26

VECTOR-TO-SCALAR OPERATIONS

Page	Name	Operation	167	333	167	333
E-62	SVE	SUM OF VECTOR ELEMENTS	0.3	0.3	7	7
E-63	SVEMG	SUM OF VECTOR ELEMENT MAGNITUDES	0.3	0.3	10	10
E-64	SVESQ	SUM OF VECTOR ELEMENT SQUARES	0.3	0.3	10	10
E-65	SVS	SUM OF VECTOR SIGNED SQUARES	0.3	0.3	11	11
E-66	DOTPR	DOT PRODUCT	0.5	0.8	21	9
E-67	MAXV	MAXIMUM ELEMENT IN VECTOR	0.3	0.3	19	19
E-68	MINV	MINIMUM ELEMENT IN VECTOR	0.3	0.3	19	19
E-69	MAXMGV	MAXIMUM MAGNITUDE ELEMENT IN VECTOR	0.3	0.3	19	19
E-70	MINMGV	MINIMUM MAGNITUDE ELEMENT IN VECTOR	0.3	0.3	19	19
E-71	MEANV	MEAN VALUE OF VECTOR ELEMENTS	0.3	0.3	49	49
E-72	MEAMGV	MEAN OF VECTOR ELEMENT MAGNITUDES	0.3	0.3	52	52
E-73	MEASQV	MEAN OF VECTOR ELEMENT SQUARES	0.3	0.3	52	52
E-74	RMSQV	ROOT-MEAN-SQUARE OF VECTOR ELEMENTS	0.3	0.3	81	81

VECTOR COMPARISON OPERATIONS

Page	Name	Operation	167	333	167	333
E-76	VMAX	VECTOR MAXIMUM	0.8	1.3	22	13
E-77	VMIN	VECTOR MINIMUM	0.8	1.3	22	13
E-78	VMAXMG	VECTOR MAXIMUM MAGNITUDE	0.8	1.3	14	14
E-79	VMINMG	VECTOR MINIMUM MAGNITUDE	0.8	1.3	14	14
E-80	VCLIP	VECTOR CLIP	0.5	0.8	16	16
E-81	VICLIP	VECTOR INVERTED CLIP	0.7	0.8	19	19
E-82	VLIM	VECTOR LIMIT	0.5	0.8	14	14
E-83	LVGT	LOGICAL VECTOR GREATER THAN	0.8	1.3	23	13
E-84	LVGE	LOGICAL VECTORGREATER THAN OR EQUAL	0.8	1.3	23	13

Page	Name	Operation	Typical Execution Time/Loop (us) 167	333	Program Size (AP PS words) 167	333
E-85	LVEQ	LOGICAL VECTOR EQUAL	0.8	1.3	23	13
E-86	LVNE	LOGICAL VECTOR NOT EQUAL	0.8	1.3	23	13
E-87	LVNOT	LOGICAL VECTOR NOT	0.5	0.8	21	12
E-88	VLMERG	VECTOR LOGICAL MERGE	0.8	1.5	23	16

COMPLEX VECTOR ARITHMETIC

E-90	CVMOV	COMPLEX VECTOR MOVE	0.8	1.3	9	9
E-91	CVFILL	COMPLEX VECTOR FILL	0.5	0.7	8	8
E-92	CVCOMB	COMPLEX VECTOR COMBINE	1.1	1.7	10	10
E-93	CVREAL	FORM COMPLEX VECTOR OF REALS	0.8	1.2	9	9
E-94	VREAL	EXTRACT REALS OF COMPLEX VECTOR	0.5	0.8	17	7
E-95	VIMAG	EXTRACT IMAGINARIES OF COMPLEX VECTOR	0.5	0.8	18	8
E-96	CVNEG	COMPLEX VECTOR NEGATE	0.8	1.3	11	11
E-97	CVCONJ	COMPLEX VECTOR CONJUGATE	0.7	1.3	10	12
E-98	CVADD	COMPLEX VECTOR ADD	1.0	2.0	13	12
E-99	CVSUB	COMPLEX VECTOR SUBTRACT	1.0	2.0	13	12
E-100	CVMUL	COMPLEX VECTOR MULTIPLY	1.0	2.0	25	26
E-101	CVSMUL	COMPLEX VECTOR SCALAR MULTIPLY	0.8	1.3	12	12
E-102	CVRCIP	COMPLEX VECTOR RECIPROCAL	5.2	5.2	50	50
E-103	CRVADD	COMPLEX AND REAL VECTOR ADD	1.3	1.8	14	14
E-104	CRVSUB	COMPLEX AND REAL VECTOR SUBTRACT	1.3	1.8	14	14
E-105	CRVMUL	COMPLEX AND REAL VECTOR MULTIPLY	1.3	1.8	14	14
E-106	CRVDIV	COMPLEX AND REAL VECTOR DIVIDE	3.3	3.3	92	92
E-107	CVMA	COMPLEX VECTOR MULTIPLY AND ADD	1.3	2.7	29	30
E-109	CVMAGS	COMPLEX VECTOR MAGNITUDE SQUARED	0.7	1.2	13	18
E-110	SCJMA	SELF-CONJUGATE MULTIPLY AND ADD	0.8	1.5	14	15
E-111	POLAR	RECTANGULAR TO POLAR CONVERSION	19.5	19.5	120	120
E-112	RECT	POLAR TO RECTANGULAR CONVERSION	2.3	2.3	49	49
E-113	CVEXP	COMPLEX VECTOR EXPONENTIAL	2.0	2.0	43	43
E-114	CVMEXP	VECTOR MULTIPLY COMPLEX EXPONENTIAL	2.3	2.3	48	48
E-115	CDOTPR	COMPLEX DOT PRODUCT	0.7	1.3	15	16

DATA FORMATING OPERATIONS

E-117	VFLT	VECTOR INTEGER FLOAT	0.5	0.8	13	11
E-118	VFIX	VECTOR INTEGER FIX	0.7	0.8	18	7
E-120	VSMAFX	VECTOR SCALAR MULTIPLY, ADD, AND FIX	0.7	0.8	14	13
E-121	VSCALE	VECTOR SCALE (POWER 2) AND FIX	0.7	0.8	12	12
E-123	VSCSCL	VECTOR SCAN, SCALE (POWER 2) AND FIX	1.5	1.7	19	19
E-125	VSHFX	VECTOR SHIFT AND FIX	0.7	0.8	9	9
E-126	VUP8	VECTOR 8-BIT BYTE UNPACK	0.5	0.5	71	71
E-127	VUPS8	VECTOR 8-BIT SIGNED BYTE UNPACK	0.9	0.9	107	107
E-128	VPK8	VECTOR 8-BIT BYTE PACK	0.9	0.9	65	65
E-129	VUP16	VECTOR 16-BIT BYTE UNPACK	0.8	0.8	61	61
E-130	VUPS16	VECTOR 16-BIT SIGNED BYTE UNPACK	1.3	1.3	58	58

Page	Name	Operation	Typical Execution Time/Loop (us)		Program Size (AP PS words)	
			167	333	167	333
E-131	VPK16	VECTOR 16-BIT BYTE PACK	0.8	0.8	46	46
E-132	VFLT32	VECTOR 32-BIT INTEGER FLOAT	1.7	1.7	65	65
E-133	VFIX32	VECTOR 32-BIT INTEGER FIX	1.2	1.2	33	33
E-134	VSEFLT	VECTOR SIGN EXTEND AND FLOAT	0.8	0.8	15	15
		MATRIX OPERATIONS				
E-136	MTRANS	MATRIX TRANSPOSE	0.5	0.9	18	22
E-137	MMUL	MATRIX MULTIPLY	0.62*	0.83	59	59
E-139	MMUL32	MATRIX MULTIPLY (DIMENSION <=32)	0.50*	0.73	27	27
E-141	MATINV	MATRIX INVERSE	1.6 *	2.1	160	160
E-143	SOLVEQ	LINEAR EQUATION SOLVER	0.7 *	0.9	216	222
E-145	MVML3	MATRIX VECTOR MULTIPLY (3X3)	2.0 *	2.2	30	30
E-147	MVML4	MATRIX VECTOR MULTIPLY (4X4)	3.3 *	3.8	39	39
E-149	CTRN3	3-DIMENSION COORDINATE TRANSFORMATION	2.3 *	2.5	37	37
E-151	FMMM	FAST MEMORY MATRIX MULTIPLY	0.43*		61	
E-153	FMMM32	FAST MEMORY MATRIX MULTIPLY (<=32)	0.41*		33	
		FFT OPERATIONS				
E-156	CFFT	COMPLEX TO COMPLEX FFT (IN PLACE)	0.28*	0.40	186	184
E-158	CFFTB	COMPLEX TO COMPLEX FFT (NOT IN PLACE)	0.20*	0.28	189	189
E-160	RFFT	REAL TO COMPLEX FFT (IN PLACE)	0.18*	0.27	253	251
E-162	RFFTB	REAL TO COMPLEX FFT (NOT IN PLACE)	0.14*	0.20	252	252
E-164	CFFTSC	COMPLEX FFT SCALE	0.8	1.3	42	42
E-165	RFFTSC	REAL FFT SCALE AND FORMAT	0.7	0.8	59	59
E-167	CFFT2D	COMPLEX TO COMPLEX 2-DIMENSIONAL FFT	0.5 *	0.5	274	274
E-169	RFFT2D	REAL TO COMPLEX 2-DIMENSIONAL FFT	0.4 *	0.4	585	585
		AUXILIARY OPERATIONS				
E-172	CONV	CONVOLUTION (CORRELATION)	0.28*	0.28	106	106
E-174	DEQ22	DIFFERENCE EQUATION, 2 POLES, 2 ZEROS	0.8	0.8	25	25
E-175	VPOLY	VECTOR POLYNOMIAL EVALUATION	1.0 *	1.2	41	41
E-177	VSUM	VECTOR SUM OF ELEMENTS INTEGRATION	0.7	0.8	13	13
E-178	VTRAPZ	VECTOR TRAPEZOIDAL RULE INTEGRATION	0.7	0.8	16	16
E-179	VSIMPS	VECTOR SIMPSONS 1/3 RULE INTEGRATION	0.7	0.8	25	25
E-180	WIENER	WIENER LEVINSON ALGORITHM	0.50*	0.65	100	100
		SIGNAL PROCESSING OPERATIONS (optional)				
E-183	HIST	HISTOGRAM	1.3	1.4	71	71

Page	Name	Operation	Typical Execution Time/Loop (us) 167	333	Program Size (AP PS words) 167	333
E-184	HANN	HANNING WINDOW MULTIPLY	0.7	0.8	41	41
E-186	ASPEC	ACCUMULATING AUTO-SPECTRUM	0.8	1.5	21	22
E-187	CSPEC	ACCUMULATING CROSS-SPECTRUM	1.3	2.7	39	40
E-188	VAVLIN	VECTOR LINEAR AVERAGING	0.8	1.3	54	46
E-189	VAVEXP	VECTOR EXPONENTIAL AVERAGING	0.8	1.3	55	46
E-190	VDBPWR	VECTOR CONVERSION TO DB (POWER)	1.2	1.3	75	75
E-191	TRANS	TRANSFER FUNCTION	3.3	3.3	100	100
E-192	COHER	COHERENCE FUNCTION	4.0	4.5	109	114
E-193	ACORT	AUTO-CORRELATION (TIME-DOMAIN)	0.29*	0.29	121	121
E-195	ACORF	AUTO-CORRELATION (FREQUENCY-DOMAIN)	1.80*	2.70	501	489
E-197	CCORT	CROSS-CORRELATION (TIME-DOMAIN)	0.29*	0.29	121	121
E-199	CCORF	CROSS-CORRELATION (FREQUENCY-DOMAIN)	2.58*	3.93	526	510
E-201	TCONV	POSTTAPERED CONVOLUTION (CORRELATION)	0.30*	0.30	112	112

TABLE MEMORY OPERATIONS (optional)

E-204	MTMOV	VECTOR MOVE (MD TO TM)	0.2	0.3	6	7
E-205	TMMOV	VECTOR MOVE (TM TO MD)	0.2	0.3	5	5
E-206	MTIMOV	VECTOR MOVE WITH INCREMENT (MD TO TM)	0.5	0.5	7	7
E-207	TMIMOV	VECTOR MOVE WITH INCREMENT (TM TO MD)	0.3	0.3	15	15
E-208	TTIMOV	VECTOR MOVE WITH INCREMENT (TM TO TM)	0.5	0.5	7	7
E-209	MMTADD	VECTOR ADD (MD+MD TO TM)	0.7	0.8	20	13
E-210	MMTSUB	VECTOR SUBTRACT (MD-MD TO TM)	0.7	0.8	20	13
E-211	MMTMUL	VECTOR MULTIPLY (MD*MD TO TM)	0.7	0.8	20	13
E-212	MTMADD	VECTOR ADD (MD+TM TO MD)	0.5	0.8	20	9
E-213	MTMSUB	VECTOR SUBTRACT (MD-TM TO MD)	0.5	0.8	20	9
E-214	TMMSUB	VECTOR SUBTRACT (TM-MD TO MD)	0.5	0.8	20	9
E-215	MTMMUL	VECTOR MULTIPLY (MD*TM TO MD)	0.5	0.8	20	9
E-216	MTTADD	VECTOR ADD (MD+TM TO TM)	0.5	0.5	20	20
E-217	MTTSUB	VECTOR SUBTRACT (MD-TM TO TM)	0.5	0.5	20	20
E-218	TMTSUB	VECTOR SUBTRACT (TM-MD TO TM)	0.5	0.5	20	20
E-219	MTTMUL	VECTOR MULTIPLY (MD*TM TO TM)	0.5	0.5	20	20
E-220	TTMADD	VECTOR ADD (TM+TM TO MD)	0.5	0.5	20	20
E-221	TTMSUB	VECTOR SUBTRACT (TM-TM TO MD)	0.5	0.5	20	20
E-222	TTMMUL	VECTOR MULTIPLY (TM*TM TO MD)	0.5	0.5	20	20
E-223	TTTADD	VECTOR ADD (TM+TM TO TM)	0.7	0.7	9	9
E-224	TTTSUB	VECTOR SUBTRACT (TM-TM TO TM)	0.7	0.7	9	9
E-225	TTTMUL	VECTOR MULTIPLY (TM*TM TO TM)	0.7	0.7	10	10

APAL-CALLABLE UTILITY OPERATIONS

E-227	DIV	SCALAR DIVIDE	3.8 @	3.8	28	28
E-228	SQRT	SCALAR SQUARE ROOT	3.8 @	3.8	28	28
E-229	LOG	SCALAR LOGARITHM (BASE 10)	4.7 @	4.7	37	37
E-230	LN	SCALAR NATURAL LOGARITHM	4.0 @	4.0	37	37

Page	Name	Operation	Typical Execution Time/Loop (us) 167 | 333	Program Size (AP PS words) 167 | 333
E-231	EXP	SCALAR EXPONENTIAL	4.2 @ 4.2	28 28
E-232	SIN	SCALAR SINE	4.9 @ 4.9	35 35
E-233	COS	SCALAR COSINE	5.4 @ 5.4	35 35
E-234	ATAN	SCALAR ARCTANGENT	8.7 @ 8.7	74 74
E-235	ATN2	SCALAR ARCTANGENT OF Y/X	13.8 @13.8	74 74
E-236	SPFLT	FLOAT S-PAD INTEGER	0.8 @ 0.8	5 5
E-237	SPUFLT	S-PAD UNSIGNED FLOAT	0.8 @ 0.8	8 8
E-238	SPNEG	S-PAD NEGATE	0.3 @ 0.3	2 2
E-239	SPADD	S-PAD ADD	0.2 @ 0.2	1 1
E-240	SPSUB	S-PAD SUBTRACT	0.2 @ 0.2	1 1
E-241	SPMUL	S-PAD MULTIPLY	2.3 @ 2.3	14 14
E-242	SPDIV	S-PAD DIVIDE	6.2 @ 6.2	43 43
E-243	SPRS	S-PAD RIGHT SHIFT	0.3 * 0.3	5 5
E-244	SPLS	S-PAD LEFT SHIFT	0.3 * 0.3	5 5
E-245	SPAND	S-PAD AND	0.2 @ 0.2	1 1
E-246	SPOR	S-PAD OR	0.2 @ 0.2	1 1
E-247	SPNOT	S-PAD NOT	0.2 @ 0.2	1 1
E-248	SAVESP	SAVE S-PAD INTO PROGRAM MEMORY	0.8 * 0.8	18 18
E-249	SAVSPO	SAVE S-PAD 0 INTO PROGRAM MEMORY	2.0 * 2.0	11 11
E-250	SETSP	LOAD S-PADS FROM PROGRAM MEMORY	2.3 * 2.3	33 33
E-252	SET2SP	LOAD 2 S-PADS FROM PROGRAM MEMORY	5.7 @ 5.7	33 33
E-253	MDCOM	MAIN DATA COMPARE AND SET S-PAD	1.8 @ 2.0	11 11
E-254	ZMD	CLEAR ALL PAGES OF MAIN DATA MEMORY	0.2 0.3	29 29
E-255	RDC5	READ CONTROL BIT 5 INTERRUPT	1.5 @ 1.5	9 9
E-256	SETC5	SET CONTROL BIT 5 INTERRUPT	0.2 @ 0.2	1 1
E-257	DAREAD	READ DEVICE ADDRESS REGISTER	0.3 @ 0.3	2 2
E-258	DAWRIT	WRITE DEVICE ADDRESS REGISTER	0.3 @ 0.3	2 2
E-259	VFCL1	VECTOR FUNCTION CALLER (1 ARGUMENT)	0.8 1.0	10 10
E-260	VFCL2	VECTOR FUNCTION CALLER (2 ARGUMENT)	1.0 1.0	11 11
E-261	BITREV	COMPLEX VECTOR BIT REVERSE ORDERING	0.9 1.4	45 43
E-262	REALTR	REAL FFT UNRAVEL AND FINAL PASS	0.4 0.7	68 68
E-263	FFT2	RADIX 2 FFT FIRST PASS	1.3 2.7	16 16
E-264	FFT4	RADIX 4 FFT PASS	3.7 5.3	79 79
E-265	FFT2B	RADIX 2 FFT FIRST PASS + BIT REVERSE	1.3 2.7	25 25
E-266	FFT4B	RADIX 4 FFT FIRST PASS + BIT REVERSE	2.7 5.3	43 43
E-267	STSTAT	SET FFT MODE STATUS BITS	5.0 @ 5.0	19 19
E-268	CLSTAT	CLEAR FFT MODE STATUS BITS	0.5 @ 0.5	19 19
E-269	ILOG2	LOGARITHM (BASE 2)	4.0 @ 4.0	19 19
E-270	ADV2	ADVANCE POINTERS AFTER RADIX 2 FFT	0.7 @ 0.7	7 7
E-271	ADV4	ADVANCE POINTERS AFTER RADIX 4 FFT	0.7 @ 0.7	7 7
E-272	SET24B	SETUP FOR FFT2B AND FFT4B	1.2 @ 1.2	8 8
E-273	XCFFT	EXPANDED COMPLEX FFT	0.32* 0.42	187 187
E-275	XRFFT	EXPANDED REAL FFT	0.19* 0.28	256 256
E-277	XBITRE	EXPANDED BIT REVERSE	3.7 3.7	44 44
E-278	XREALT	EXPANDED REAL FFT FINAL PASS	0.4 0.7	71 71
E-279	PCFFT	PARTIAL COMPLEX FFT	1.05* 1.50	117 117
E-280	XFFT4	EXPANDED RADIX 4 FFT PASS	3.7 5.3	79 79
E-281	CTOR	COMPLEX TO REAL FFT UNSCRAMBLE	0.13* 0.13	80 80
E-282	RTOC	REAL TO COMPLEX FFT SCRAMBLE	0.09* 0.09	143 143
E-284	SSDA	SINGLE + SINGLE TO DOUBLE ADD	1.5 @ 1.5	10 10

Page	Name	Operation	Typical Execution Time/Loop (us) 167	333	Program Size (AP PS words) 167	333
E-285	SSDM	SINGLE * SINGLE TO DOUBLE MULTIPLY	11.5	@11.5	81	81
E-286	SDDA	SINGLE + DOUBLE TO DOUBLE ADD	4.5	@ 4.5	28	28
E-287	DDDA	DOUBLE + DOUBLE TO DOUBLE ADD	7.5	@ 7.5	48	48
E-288	DDDM	DOUBLE * DOUBLE TO DOUBLE MULTIPLY	18.5	@18.5	117	117

Notes: #.# Timing host system dependent
* Refer to description of routine for explanation of timing
@ Total execution time

Page	Routine	Purpose

DATA TRANSFER AND CONTROL OPERATIONS (APEX)

E-4   CALL APPUT(HOST,AP,N,TYPE)

PUT DATA INTO THE AP
To transfer data from the host computer memory into the AP main data memory.

HOST = An array name, array element, variable, or constant which specifies the initial host data element to be transferred.

AP = An integer constant, variable, or expression which specifies the base address in AP main data memory into which data is to be transferred.

N = Element count (AP data words)

TYPE = An integer specifying the host data type and format conversion during data transfer to the AP.

0 32-bit integers. Stored without format conversion into the low 32-bits (bits 8-39) of AP main data memory words.

1 16-bit integers. Converted into un-normalized AP floating-point numbers. These numbers must be normalized (using VFLT) before they can be processed by the AP.

2 Host single-precision (real) floating-point numbers. Converted "on the fly" to normalized AP floating-point numbers.

3 IBM 360 32-bit format floating-point numbers. Converted "on the fly" to normalized AP floating-point numbers.

E-6   CALL APGET(HOST,AP,N,TYPE)

GET DATA FROM THE AP
To transfer data from the AP main data memory into the host computer memory.

HOST = An array name, array element, variable, or constant which specifies the initial host memory location to receive transferred data.

AP = An integer constant, variable, or expression which specifies the base address in AP main data memory from which data is to be transferred.

N = Element count (AP data words)

Page	Routine	Purpose
	TYPE = An integer specifying the host data type and format conversion during data transfer from the AP. 0 32-bit integers. The low 32 bits (bits 8-39) of AP memory words are transferred without format conversion into the host memory. 1 16-bit integers. The low 16 bits (bits 24-39) of AP memory words are stored into host integer locations. 2 AP floating-point numbers are converted "on the fly" into host single-precision (real) floating-point numbers. 3 AP floating-point numbers are converted "on the fly" into IBM 360 32-bit format floating-point numbers in the host.	
E-8	CALL APCLR	INITIALIZE THE AP To initialize the AP by clearing the hardware status and initializing APEX.
E-9	CALL APWD	WAIT FOR AP DATA TRANSFER To delay host program execution until any previously initiated data transfer between the host and the AP has been completed.
E-10	CALL APWR	WAIT FOR AP PROGRAM EXECUTION To delay host program execution until any previously initiated AP program has been completed.
E-11	CALL APWAIT	WAIT FOR AP To delay host program execution until the AP is done transferring data and executing a program.
E-12	CALL APGSP(I,NREG) I = Value contained in S-Pad register NREG = S-Pad register number (1 to 15)	READ AN AP S-PAD REGISTER To read the contents of an AP S-Pad register.
E-13	CALL APCHK(IERR) IERR = Error information from AP program.	CHECK AP PROGRAM ERROR CONDITION To check error information returned by certain AP Math Library programs.

Page	Routine	Purpose
E-14	CALL APSTAT(IERR,ISTAT) IERR = Set to 1 if hardware error detected, 0 otherwise ISTAT = A 4-element array to delineate the error conditions as follows: ISTAT(1) = Arithmetic overflow ISTAT(2) = Arithmetic underflow ISTAT(3) = Divide by zero ISTAT(4) = Format conversion overflow/underflow	GET AP HARDWARE STATUS To read the AP status and DMA control registers.

BASIC VECTOR ARITHMETIC

Page	Routine	Purpose
E-16	CALL VCLR(C,K,N) C = Destination vector base address K = C address increment N = Element count	VECTOR CLEAR To clear elements of a vector.
E-17	CALL VMOV(A,I,C,K,N) A = Source vector base address I = A address increment C = Destination vector base address K = C address increment N = Element count	VECTOR MOVE To move elements of a vector from one location to another.
E-18	CALL VSWAP(A,I,C,K,N) A = Vector base address I = A address increment C = Vector base address K = C address increment N = Element count	VECTOR SWAP To swap data between two vectors.
E-19	CALL VFILL(A,C,K,N) A = Address of constant value C = Destination vector base address K = C address increment N = Element count	VECTOR FILL To fill elements of a vector with a constant.
E-20	CALL VRAMP(A,B,C,K,N) A = Address of initial ramp value B = Address of ramp increment C = Destination vector base address K = C address increment N = Element count	VECTOR RAMP To fill elements of a vector with a ramp function.
E-21	CALL VNEG(A,I,C,K,N) A = Source vector base address I = A address increment C = Destination vector base address K = C address increment	VECTOR NEGATE To negate elements of a vector.

Page	Routine	Purpose
	N = Element count	
E-22	CALL VADD(A,I,B,J,C,K,N) A = Source vector base address I = A address increment B = Source vector base address J = B address increment C = Destination vector base address K = C address increment N = Element count	VECTOR ADD To add the elements of two vectors.
E-23	CALL VSUB(A,I,B,J,C,K,N) A = Source vector base address I = A address increment B = Source vector base address J = B address increment C = Destination vector base address K = C address increment N = Element count	VECTOR SUBTRACT To subtract the elements of two vectors.
E-24	CALL VMUL(A,I,B,J,C,K,N) A = Source vector base address I = A address increment B = Source vector base address J = B address increment C = Destination vector base address K = C address increment N = Element count	VECTOR MULTIPLY To multiply the elements of two vectors.
E-25	CALL VDIV(A,I,B,J,C,K,N) A = Source vector base address I = A address increment B = Source vector base address J = B address increment C = Destination vector base address K = C address increment N = Element count	VECTOR DIVIDE To divide the elements of two vectors.
E-26	CALL VSADD(A,I,B,C,K,N) A = Source vector base address I = A address increment B = Scalar address C = Destination vector base address K = C address increment N = Element count	VECTOR SCALAR ADD To add a scalar to the elements of a vector.
E-27	CALL VSMUL(A,I,B,C,K,N) A = Source vector base address I = A address increment B = Scalar address C = Destination vector base address K = C address increment N = Element count	VECTOR SCALAR MULTIPLY To multiply the elements of a vector by a scalar.

Page	Routine	Purpose
E-28	CALL VTSADD(A,I,B,C,K,N) A = Source vector base address I = A address increment B = Scalar address (Table Memory) C = Destination vector base address K = C address increment N = Element count	VECTOR TABLE SCALAR ADD To add a table memory scalar to the elements of a vector.
E-29	CALL VTSMUL(A,I,B,C,K,N) A = Source vector base address I = A address increment B = Scalar address (Table Memory) C = Destination vector base address K = C address increment N = Element count	VECTOR TABLE SCALAR MULTIPLY To multiply the elements of a vector by a table memory scalar.
E-30	CALL VSQ(A,I,C,K,N) A = Source vector base address I = A address increment C = Destination vector base address K = C address increment N = Element count	VECTOR SQUARE To square the elements of a vector.
E-31	CALL VSSQ(A,I,C,K,N) A = Source vector base address I = A address increment C = Destination vector base address K = C address increment N = Element count	VECTOR SIGNED SQUARE To multiply each element of a vector by the absolute value of that element.
E-32	CALL VABS(A,I,C,K,N) A = Source vector base address I = A address increment C = Destination vector base address K = C address increment N = Element count	VECTOR ABSOLUTE VALUE To take the absolute value of the elements of a vector.
E-33	CALL VSQRT(A,I,C,K,N) A = Source vector base address I = A address increment C = Destination vector base address K = C address increment N = Element count	VECTOR SQUARE ROOT To take the square root of the elements of a vector.
E-34	CALL VLOG(A,I,C,K,N) A = Source vector base address I = A address increment C = Destination vector base address K = C address increment N = Element count	VECTOR LOGARITHM (BASE 10) To take the logarithm (base 10) of the elements of a vector.
E-35	CALL VLN(A,I,C,K,N) A = Source vector base address I = A address increment	VECTOR NATURAL LOGARITHM To take the natural logarithm of the elements of a vector.

Page	Routine	Purpose
	C = Destination vector base address K = C address increment N = Element count	
E-36	CALL VALOG(A,I,C,K,N) A = Source vector base address I = A address increment C = Destination vector base address K = C address increment N = Element count	VECTOR ANTILOGARITHM (BASE 10) To take the antilogarithm of the elements of a vector.
E-37	CALL VEXP(A,I,C,K,N) A = Source vector base address I = A address increment C = Destination vector base address K = C address increment N = Element count	VECTOR EXPONENTIAL To take the exponential of the elements of a vector.
E-38	CALL VSIN(A,I,C,K,N) A = Source vector base address I = A address increment C = Destination vector base address K = C address increment N = Element count	VECTOR SINE To compute the sine of the elements of a vector.
E-39	CALL VCOS(A,I,C,K,N) A = Source vector base address I = A address increment C = Destination vector base address K = C address increment N = Element count	VECTOR COSINE To compute the cosine of the elements of a vector.
E-40	CALL VATAN(A,I,C,K,N) A = Source vector base address I = A address increment C = Destination vector base address K = C address increment N = Element count	VECTOR ARCTANGENT To take the arctangent of the elements of a vector.
E-41	CALL VATN2(A,I,B,J,C,K,N) A = Source vector base address I = A address increment B = Source vector base address J = B address increment C = Destination vector base address K = C address increment N = Element count	VECTOR ARCTANGENT OF Y/X To take the arctangent of the ratio of the elements of two vectors.
E-42	CALL VRAND(A,C,K,N) A = Address of random number seed C = Destination vector base address K = C address increment N = Element count	VECTOR RANDOM NUMBERS To fill elements of a vector with random numbers.

Page	Routine	Purpose
E-43	CALL VMSA (A,I,B,J,C,D,L,N) A = Source vector base address I = A address increment B = Source vector base address J = B address increment C = Scalar address D = Result vector base address L = D address increment N = Vector length	VECTOR MULTIPLY AND SCALAR ADD To multiply the elements of two vectors and add a scalar to the products.
E-44	CALL VSMA(A,I,B,C,K,D,L,N) A = Source vector base address I = A address increment B = Scalar address C = Source vector base address K = C address increment D = Destination vector base address L = D address increment N = Element count	VECTOR SCALAR MULTIPLY AND ADD To multiply the elements of a vector by a scalar and add a second vector to the products.
E-45	CALL VSMSB(A,I,B,C,K,D,L,N) A = Source vector base address I = A address increment B = Scalar address C = Source vector base address K = C address increment D = Destination vector base address L = D address increment N = Element count	VECTOR SCALAR MULTIPLY AND SUBTRACT To multiply the elements of a vector by a scalar and subtract a second vector from the products.
E-46	CALL VMA(A,I,B,J,C,K,D,L,N) A = Source vector base address I = A address increment B = Source vector base address J = B address increment C = Source vector base address K = C address increment D = Destination vector base address L = D address increment N = Element count	VECTOR MULTIPLY AND ADD To multiply the elements of two vectors, and add the products to a third vector, i.e., D=(A*B)+C.
E-47	CALL VMSB(A,I,B,J,C,K,D,L,N) A = Source vector base address I = A address increment B = Source vector base address J = B address increment C = Source vector base address K = C address increment D = Destination vector base address L = D address increment N = Element count	VECTOR MULTIPLY AND SUBTRACT To multiply the elements of two vectors, and subtract a third vector from the products, i.e., D=(A*B)-C.
E-48	CALL VAM(A,I,B,J,C,K,D,L,N) A = Source vector base address	VECTOR ADD AND MULTIPLY To add the elements of two

Page	Routine	Purpose
	I = A address increment B = Source vector base address J = B address increment C = Source vector base address K = C address increment D = Destination vector base address L = D address increment N = Element count	vectors, and multiply the sum by a third vector, i.e., D=(A+B)*C.
E-49	CALL VSBM(A,I,B,J,C,K,D,L,N) A = Source vector base address I = A address increment B = Source vector base address J = B address increment C = Source vector base address K = C address increment D = Destination vector base address L = D address increment N = Element count	VECTOR SUBTRACT AND MULTIPLY To subtract the elements of two vectors, and multiply the difference by a third vector, i.e., D=(A-B)*C.
E-50	CALL VSMSA(A,I,B,C,D,L,N) A = Source vector base address I = A address increment B = Multiplying scalar address C = Adding scalar address D = Destination vector base address L = D address increment N = Element count	VECTOR SCALAR MULTIPLY AND SCALAR ADD To multiply the elements of a vector by a scalar and add a second scalar to the products.
E-51	CALL VMMA(A,I,B,J,C,K,D,L,E,M,N) A = Source vector base address I = A address increment B = Source vector base address J = B address increment C = Source vector base address K = C address increment D = Source vector base address L = D address increment E = Destination vector base address M = E address increment N = Element count	VECTOR MULTIPLY, MULTIPLY, AND ADD To multiply the elements of two vectors, multiply the elements of a second set of two vectors, and add the two product vectors, i.e. E=(A*B)+(C*D).
E-52	CALL VMMSB(A,I,B,J,C,K,D,L,E,M,N) A = Source vector base address I = A address increment B = Source vector base address J = B address increment C = Source vector base address K = C address increment D = Source vector base address L = D address increment E = Destination vector base address M = E address increment N = Element count	VECTOR MULTIPLY MULTIPLY AND SUBTRACT To multiply the elements of two vectors, multiply the elements of a second set of two vectors, and subtract the two product vectors, i.e. E=(A*B)-(C*D).

Page	Routine	Purpose
E-53	CALL VAAM(A,I,B,J,C,K,D,L,E,M,N) A = Source vector base address I = A address increment B = Source vector base address J = B address increment C = Source vector base address K = C address increment D = Source vector base address L = D address increment E = Destination vector base address M = E address increment N = Element count	VECTOR ADD, ADD, AND MULTIPLY To add the elements of two vectors, add the elements of a second set of two vectors, and multiply the two sum vectors, i.e. E=(A+B)*(C+D).
E-54	CALL VSBSBM(A,I,B,J,C,K,D,L,E,M,N) A = Source vector base address I = A address increment B = Source vector base address J = B address increment C = Source vector base address K = C address increment D = Source vector base address L = D address increment E = Destination vector base address M = E address increment N = Element count	VECTOR SUBTRACT SUBTRACT AND MULTIPLY To subtract the elements of two vectors, subtract the elements of a second set of two vectors, and multiply the two difference vectors, i.e. E=(A-B)*(C-D).
E-55	CALL VAND(A,I,B,J,C,K,N) A = Source vector base address I = A address increment B = Source vector base address J = B address increment C = Destination vector base address K = C address increment N = Element count	VECTOR LOGICAL AND To logically AND the elements of two vectors.
E-56	CALL VEQV(A,I,B,J,C,K,N) A = Source vector base address I = A address increment B = Source vector base address J = B address increment C = Destination vector base address K = C address increment N = Element count	VECTOR LOGICAL EQUIVALENCE To logically EQUIVALENCE the elements of two vectors.
E-57	CALL VOR(A,I,B,J,C,K,N) A = Source vector base address I = A address increment B = Source vector base address J = B address increment C = Destination vector base address K = C address increment N = Element count	VECTOR LOGICAL OR To logically OR the elements of two vectors.

Page	Routine	Purpose
E-58	CALL VFRAC(A,I,C,K,N) A = Source vector base address I = A address increment C = Destination vector base address K = C address increment N = Element count	VECTOR TRUNCATE TO FRACTION To truncate the elements of a vector to their fractional parts.
E-59	CALL VINT(A,I,C,K,N) A = Source vector base address I = A address increment C = Destination vector base address K = C address increment N = Element count	VECTOR TRUNCATE TO INTEGER To truncate the elements of a vector to integer floating point numbers.
E-60	CALL VINDEX(A,B,J,C,K,N) A = Source vector base address B = Index vector base address J = B address increment C = Destination vector base address K = C address increment N = Element count	VECTOR INDEX To form a vector by using the elements of one vector as the addresses by which to select the elements of a second vector.

VECTOR-TO-SCALAR OPERATIONS

Page	Routine	Purpose
E-62	CALL SVE(A,I,C,N) A = Source vector base address I = A address increment C = Destination scalar address N = Element count	SUM OF VECTOR ELEMENTS To sum the elements of a vector.
E-63	CALL SVEMG(A,I,C,N) A = Source vector base address I = A address increment C = Destination scalar address N = Element count	SUM OF VECTOR ELEMENT MAGNITUDES To sum the absolute values of the elements of a vector.
E-64	CALL SVESQ(A,I,C,N) A = Source vector base address I = A address increment C = Destination scalar address N = Element count	SUM OF VECTOR ELEMENT SQUARES To sum the squares of the elements of a vector.
E-65	CALL SVS(A,I,C,N) A = Source vector base address I = A address increment C = Destination scalar address N = Element count	SUM OF VECTOR SIGNED SQUARES To sum the signed squares of the elements of a vector.
E-66	CALL DOTPR(A,I,B,J,C,N) A = Source vector base address I = A address increment	DOT PRODUCT To compute the dot product of the elements of two vectors.

Page	Routine	Purpose
	B = Source vector base address J = B address increment C = Destination scalar address N = Element count	
E-67	CALL MAXV(A,I,C,N) A = Source vector base address I = A address increment C = Destination scalar address (2 words required) N = Element count	MAXIMUM ELEMENT IN VECTOR To scan a vector for its maximum element.
E-68	CALL MINV(A,I,C,N) A = Source vector base address I = A address increment C = Destination scalar address (2 words required) N = Element count	MINIMUM ELEMENT IN VECTOR To scan a vector for its minimum element.
E-69	CALL MAXMGV(A,I,C,N) A = Source vector base address I = A address increment C = Destination scalar address (2 words required) N = Element count	MAXIMUM MAGNITUDE ELEMENT IN VECTOR To scan a vector for its maximum magnitude (absolute value) element.
E-70	CALL MINMGV(A,I,C,N) A = Source vector base address I = A address increment C = Destination scalar address (2 words required) N = Element count	MINIMUM MAGNITUDE ELEMENT IN VECTOR To scan a vector for its minimum magnitude (absolute value) element.
E-71	CALL MEANV(A,I,C,N) A = Source vector base address I = A address increment C = Destination scalar address N = Element count	MEAN VALUE OF VECTOR ELEMENTS To compute the mean (average) value of the elements of a vector.
E-72	CALL MEAMGV(A,I,C,N) A = Source vector base address I = A address increment C = Destination scalar address N = Element count	MEAN OF VECTOR ELEMENT MAGNITUDES To compute the mean (average) value of the absolute values of the elements of a vector.
E-73	CALL MEASQV(A,I,C,N) A = Source vector base address I = A address increment C = Destination scalar address N = Element count	MEAN OF VECTOR ELEMENT SQUARES To compute the mean (average) value of the squares of the elements of a vector.
E-74	CALL RMSQV(A,I,C,N) A = Source vector base address I = A address increment	ROOT-MEAN-SQUARE OF VECTOR ELEMENTS To compute the square root of the mean (average) value of the

Page	Routine	Purpose
	C = Destination scalar address N = Element count	squares of the elements of a vector.

VECTOR COMPARISON OPERATIONS

Page	Routine	Purpose
E-76	CALL VMAX(A,I,B,J,C,K,N) A = Source vector base address I = A address increment B = Source vector base address J = B address increment C = Destination vector base address K = C address increment N = Element count	VECTOR MAXIMUM To form a vector from the maximum value of each corresponding pair of elements of two vectors.
E-77	CALL VMIN(A,I,B,J,C,K,N) A = Source vector base address I = A address increment B = Source vector base address J = B address increment C = Destination vector base address K = C address increment N = Element count	VECTOR MINIMUM To form a vector from the minimum value of each corresponding pair of elements of two vectors.
E-78	CALL VMAXMG(A,I,B,J,C,K,N) A = Source vector base address I = A address increment B = Source vector base address J = B address increment C = Destination vector base address K = C address increment N = Element count	VECTOR MAXIMUM MAGNITUDE To form a vector from the maximum absolute value of each corresponding pair of elements of two vectors.
E-79	CALL VMINMG(A,I,B,J,C,K,N) A = Source vector base address I = A address increment B = Source vector base address J = B address increment C = Destination vector base address K = C address increment N = Element count	VECTOR MINIMUM MAGNITUDE To form a vector from the minimum absolute value of each corresponding pair of elements of two vectors.
E-80	CALL VCLIP(A,I,B,C,D,L,N) A = Source vector base address I = A address increment B = Address of smaller scalar C = Address of larger scalar D = Destination vector base address L = D address increment N = Element count	VECTOR CLIP To clip the values of a vector to within a specified range.
E-81	CALL VICLIP(A,I,B,C,D,L,N)	VECTOR INVERTED CLIP

Page	Routine	Purpose
	A = Source vector base address I = A address increment B = Address of smaller scalar C = Address of larger scalar D = Destination vector base address L = D address increment N = Element count	To exclude values of a vector from within a specified range.
E-82	CALL VLIM(A,I,B,C,D,L,N) A = Source vector base address I = A address increment B = Address of scalar to compare with source C = Address of destination magnitude scalar D = Destination vector base address L = D address increment N = Element count	VECTOR LIMIT To create a vector limited to a single value in magnitude, where the sign of each element depends on whether the corresponding element of a second vector exceeds a certain value.
E-83	CALL LVGT(A,I,B,J,C,K,N) A = Source vector base address I = A address increment B = Source vector base address J = B address increment C = Destination vector base address K = C address increment N = Element count	LOGICAL VECTOR GREATER THAN To compare the elements of two vectors A and B and output a vector C such that: C(mK)=1.0 if A(mI)>B(mJ) C(mK)=0.0 if A(mI)=<B(mJ)
E-84	CALL LVGE(A,I,B,J,C,K,N) A = Source vector base address I = A address increment B = Source vector base address J = B address increment C = Destination vector base address K = C address increment N = Element count	LOGICAL VECTOR GREATER THAN OR EQUAL To compare the elements of two vectors A and B and output a vector C such that: C(mK)=1.0 if A(mI)>=B(mJ) C(mK)=0.0 if A(mI)<B(mJ)
E-85	CALL LVEQ(A,I,B,J,C,K,N) A = Source vector base address I = A address increment B = Source vector base address J = B address increment C = Destination vector base address K = C address increment N = Element count	LOGICAL VECTOR EQUAL To compare the elements of two vectors A and B and output a vector C such that: C(mK)=1.0 if A(mI)=B(mJ) C(mK)=0.0 if A(mI)not=B(mJ)
E-86	CALL LVNE(A,I,B,J,C,K,N) A = Source vector base address I = A address increment B = Source vector base address J = B address increment C = Destination vector base address K = C address increment N = Element count	LOGICAL VECTOR NOT EQUAL To compare the elements of two vectors A and B and output a vector C such that: C(mK)=1.0 if A(mI)not=B(mJ) C(mK)=0.0 if A(mI)=B(mJ)

Page	Routine	Purpose
E-87	CALL LVNOT(A,I,C,K,N) A = Source vector base address I = A address increment C = Destination vector base address K = C address increment N = Element count	LOGICAL VECTOR NOT To examine the elements of a vector A and output a vector C such that: C(mK)=1.0 if A(mI)=0.0 C(mK)=0.0 if A(mI)not=0.0
E-88	CALL VLMERG(A,I,B,J,C,K,D,L,N) A = Source vector base address I = A address increment B = Source vector base address J = B address increment C = Source vector base address K = C address increment D = Destination vector base address L = D address increment N = Element count	VECTOR LOGICAL MERGE To examine the elements of three vectors,A,B,and C and output a vector D such that: D(mL)=A(mI) if C(mK)not=0.0 D(mL)=B(mJ) if C(mK)=0.0

COMPLEX VECTOR ARITHMETIC

Page	Routine	Purpose
E-90	CALL CVMOV(A,I,C,K,N) A = Source vector base address I = A address increment C = Destination vector base address K = C address increment N = Element count	COMPLEX VECTOR MOVE To move the elements of a complex vector from one location to another.
E-91	CALL CVFILL(A,C,K,N) A = Complex constant base address C = Destination vector base address K = C address increment N = Complex element count	COMPLEX VECTOR FILL To fill the elements of a complex vector with a complex constant.
E-92	CALL CVCOMB(A,I,B,J,C,K,N) A = Real source vector base address I = A address increment B = Imaginary source vector base address J = B address increment C = Destination vector base address K = C address increment N = Element count	COMPLEX VECTOR COMBINE To form a complex vector by combining two real vectors.
E-93	CALL CVREAL(A,I,C,K,N) A = Real source vector base address I = A address increment C = Destination vector base address K = C address increment N = Element count	FORM COMPLEX VECTOR OF REALS To form a complex vector by combining a real vector and zeroing the imaginaries.
E-94	CALL VREAL(A,I,C,K,N)	EXTRACT REALS OF COMPLEX VECTOR

Page	Routine	Purpose
	A = Complex source vector base address I = A address increment C = Real destination vector base address K = C address increment N = Element count	To form a real vector by extracting the real parts from a complex vector.
E-95	CALL VIMAG(A,I,C,K,N) A = Complex source vector base address I = A address increment C = Real destination vector base address K = C address increment N = Element count	EXTRACT IMAGINARIES OF COMPLEX VECTOR To form a real vector by extracting the imaginary parts from a complex vector.
E-96	CALL CVNEG(A,I,C,K,N) A = Source vector base address I = A address increment C = Destination vector base address K = C address increment N = Element count	COMPLEX VECTOR NEGATE To negate the elements of a complex vector.
E-97	CALL CVCONJ(A,I,C,K,N) A = Source vector base address I = A address increment C = Destination vector base address K = C address increment N = Complex element count	COMPLEX VECTOR CONJUGATE To conjugate the elements of a complex vector.
E-98	CALL CVADD(A,I,B,J,C,K,N) A = Source vector base address I = A address increment B = Source vector base address J = B address increment C = Destination vector base address K = C address increment N = Element count	COMPLEX VECTOR ADD To add the elements of two complex vectors.
E-99	CALL CVSUB(A,I,B,J,C,K,N) A = Source vector base address I = A address increment B = Source vector base address J = B address increment C = Destination vector base address K = C address increment N = Element count	COMPLEX VECTOR SUBTRACT To subtract the elements of two complex vectors.
E-100	CALL CVMUL(A,I,B,J,C,K,N,F) A = Source vector base address I = A address increment B = Source vector base address J = B address increment C = Destination vector base address K = C address increment N = Complex element count F = Conjugate flag,	COMPLEX VECTOR MULTIPLY To multiply the elements of two complex vectors.

Page	Routine	Purpose
	+1 = normal complex multiply -1 = multiply with conjugate of A	
E-101	CALL CVSMUL(A,I,B,C,K,N) A = Source vector base address I = A address increment B = Scalar address C = Destination vector base address K = C address increment N = Element count	COMPLEX VECTOR SCALAR MULTIPLY To multiply the elements of a complex vector by a real scalar.
E-102	CALL CVRCIP(A,I,C,K,N) A = Source vector base address I = A address increment C = Destination vector base address K = C address increment N = Complex element count	COMPLEX VECTOR RECIPROCAL To obtain reciprocal of a complex vector.
E-103	CALL CRVADD(A,I,B,J,C,K,N) A = Source vector base address (complex) I = A address increment B = Source vector base address (real) J = B address increment C = Destination vector base address K = C address increment N = Element count	COMPLEX AND REAL VECTOR ADD To add the elements of a complex vector to the elements of a real vector.
E-104	CALL CRVSUB(A,I,B,J,C,K,N) A = Source vector base address (complex) I = A address increment B = Source vector base address (real) J = B address increment C = Destination vector base address K = C address increment N = Element count	COMPLEX AND REAL VECTOR SUBTRACT To subtract the elements of a real vector from the elements of a complex vector.
E-105	CALL CRVMUL(A,I,B,J,C,K,N) A = Source vector base address (complex) I = A address increment B = Source vector base address (real) J = B address increment C = Destination vector base address K = C address increment N = Element count	COMPLEX AND REAL VECTOR MULTIPLY To multiply the elements of a complex vector by the elements of a real vector.
E-106	CALL CRVDIV(A,I,B,J,C,K,N) A = Source vector base address (complex) I = A address increment B = Source vector base address (real) J = B address increment C = Destination vector base address K = C address increment N = Element count	COMPLEX AND REAL VECTOR DIVIDE To divide the elements of a complex vector by the elements of a real vector.

Page	Routine	Purpose
E-107	CALL CVMA(A,I,B,J,C,K,D,L,N,F) A = Source vector base address I = A address increment B = Source vector base address J = B address increment C = Source vector base address K = C address increment D = Destination vector base address L = D address increment N = Complex element count F = Conjugate flag, +1 = normal complex multiply -1 = multiply with conjugate of A	COMPLEX VECTOR MULTIPLY AND ADD To multiply the elements of two complex vectors, and add the products to a third complex vector.
E-109	CALL CVMAGS(A,I,C,K,N) A = Source vector base address I = A address increment C = Destination vector base address K = C address increment N = Complex element count	COMPLEX VECTOR MAGNITUDE SQUARED To compute the squared magnitude of the elements of a complex vector.
E-110	CALL SCJMA(A,I,B,J,C,K,N) A = Source complex vector base address I = A address increment B = Source real vector base address J = B address increment C = Destination real vector base address K = C address increment N = Complex element count	SELF-CONJUGATE MULTIPLY AND ADD To multiply the elements of a complex vector by the conjugate of that vector (squared magnitude), and add the real products to a real vector.
E-111	CALL POLAR(A,I,C,K,N) A = Source vector base address I = A address increment C = Destination vector base address K = C address increment N = Complex element count	RECTANGULAR TO POLAR CONVERSION To convert a complex vector from rectangular to polar form.
E-112	CALL RECT(A,I,C,K,N) A = Source vector base address I = A address increment C = Destination vector base address K = C address increment N = Complex element count	POLAR TO RECTANGULAR CONVERSION To convert a complex vector from polar to rectangular form.
E-113	CALL CVEXP(A,I,C,K,N) A = Source vector base address I = A address increment C = Destination vector base address K = C address increment N = Element count	COMPLEX VECTOR EXPONENTIAL To calculate the complex exponential exp(iX)= COS(X)+iSIN(X).
E-114	CALL CVMEXP(A,I,B,J,C,K,N) A = Source vector base address I = A address increment	VECTOR MULTIPLY COMPLEX EXPONENTIAL To multiply a real vector by a complex exponential.

Page	Routine	Purpose
	B = Source vector base address J = B address increment C = Destination vector base address K = C address increment N = Complex element count	
E-115	CALL CDOTPR(A,I,B,J,C,N) A = Source vector base address I = A address increment B = Source vector base address J = B address increment C = Destination scalar address N = Complex element count	COMPLEX DOT PRODUCT To compute the complex dot product of two complex vectors.

DATA FORMATING OPERATIONS

Page	Routine	Purpose
E-117	CALL VFLT(A,I,C,K,N) A = Source vector base address I = A address increment C = Destination vector base address K = C address increment N = Element count	VECTOR INTEGER FLOAT To convert a vector of integers to a vector of floating-point numbers.
E-118	CALL VFIX(A,I,C,K,N) A = Source vector base address I = A address increment C = Destination vector base address K = C address increment N = Element count	VECTOR INTEGER FIX To fix to integers the elements of a floating-point vector.
E-120	CALL VSMAFX(A,I,B,C,D,L,N) A = Source vector base address I = A address increment B = Multiplying scalar address C = Adding scalar address D = Destination vector base address L = D address increment N = Element count	VECTOR SCALAR MULTIPLY, ADD, AND FIX To multiply the elements of a vector by a scalar, add a second scalar to the products, and fix the resulting sums to integers.
E-121	CALL VSCALE(A,I,B,C,K,N,NB) A = Source vector base address I = A address increment B = Scalar base address C = Destination vector base address K = C address increment N = Element count NB = Desired width (2 to 28 bits) of integers, including sign bit	VECTOR SCALE (POWER 2) AND FIX To scale the elements of a vector by a power of 2 such that a selected scalar will just fit into a specified integer bit width, and then fix the scaled elements to integers.
E-123	CALL VSCSCL(A,I,C,K,N,NB) A = Source vector base address	VECTOR SCAN, SCALE (POWER 2) AND FIX To scale the elements of a

Page	Routine	Purpose
	I = A address increment C = Destination vector base address K = C address increment N = Element count NB = Desired width (2 to 28 bits) of integers, including sign bit	vector by a power of 2 such that the largest magnitude element will just fit into a specified integer bit width, and then fix the scaled elements to integers.
E-125	CALL VSHFX(A,I,C,K,N,NS) A = Source vector base address I = A address increment C = Destination vector base address K = C address increment N = Element count NS = Power of 2 (may be negative)	VECTOR SHIFT AND FIX To shift (multiply by a power of 2) and then fix to integers the elements of a floating-point vector.
E-126	CALL VUP8(A,I,C,K,N) A = Source vector base address I = A address increment C = Destination vector base address K = C address increment N = Element count (source words)	VECTOR 8-BIT BYTE UNPACK To unpack four 8-bit unsigned bytes from each source vector word and store them in four destination words as 38-bit floating-point numbers.
E-127	CALL VUPS8(A,I,C,K,N) A = Source vector base address I = A address increment C = Destination vector base address K = C address increment N = Element count (source words)	VECTOR 8-BIT SIGNED BYTE UNPACK To unpack four 8-bit 2's complement signed bytes from each source word and store them in four destination words as 38-bit floating-point numbers.
E-128	CALL VPK8(A,I,C,K,N) A = Source vector base address I = A address increment C = Destination vector base address K = C address increment N = Element count (destination words)	VECTOR 8-BIT BYTE PACK To pack each four 38-bit floating-point numbers into one destination word as 8-bit bytes.
E-129	CALL VUP16(A,I,C,K,N) A = Source vector base address I = A address increment C = Destination vector base address K = C address increment N = Element count (source words)	VECTOR 16-BIT BYTE UNPACK To unpack two 16-bit unsigned bytes from each source word and store them in two destination words as 38-bit floating- point positive numbers.
E-130	CALL VUPS16(A,I,C,K,N) A = Source vector base address I = A address increment C = Destination vector base address K = C address increment N = Element count (source words) N	VECTOR 16-BIT SIGNED BYTE UNPACK To unpack two 16-bit signed 2's complement bytes from each source word and store them in two destination words as signed 38-bit floating-point numbers.
E-131	CALL VPK16(A,I,C,K,N) A = Source vector base address I = A address increment C = Destination vector base address	VECTOR 16-BIT BYTE PACK To pack each two 38-bit floating-point numbers into one destination word as 16-bit

Page Routine Purpose

K = C address increment
N = Element count (destination words)

bytes.

E-132 CALL VFLT32(A,I,C,K,N)
A = Source vector base address
I = A address increment
C = Destination vector base address
K = C address increment
N = Element count

VECTOR 32-BIT INTEGER FLOAT
To float 32-bit signed 2's complement integers and store them as 38-bit floating point integers.

E-133 CALL VFIX32(A,I,C,K,N)
A = Source vector base address
I = A address increment
C = Destination vector base address
K = C address increment
N = Element count

VECTOR 32-BIT INTEGER FIX
To fix floating-point numbers from -2147483648 to 2147483647 and store them in a destination vector as 32-bit signed 2's complement integers.

E-134 CALL VSEFLT(A,I,C,K,N)
A = Source vector base address
I = A address increment
C = Destination vector base address
K = C address increment
N = Element count

VECTOR SIGN EXTEND AND FLOAT
To extend the sign of a vector of 16-bit integers and convert them to floating-point numbers.

MATRIX OPERATIONS

E-136 CALL MTRANS(A,I,C,K,MC,NC)
A = Source matrix base address
I = A address increment
C = Destination matrix base address
K = C address increment
MC = Number of rows of C (Columns of A)
NC = Number of columns of C (rows of A)

MATRIX TRANSPOSE
To transpose a matrix.

E-137 CALL MMUL(A,I,B,J,C,K,MC,NC,NA)
A = Source matrix base address
I = A address increment
B = Source matrix base address
J = B address increment
C = Destination matrix base address
K = C address increment
MC = Number of rows in C (Rows in A)
NC = Number of columns in C (Columns in B)
NA = Number of columns in A (Rows in B)

MATRIX MULTIPLY
To multiply two matrices.

E-139 CALL MMUL32(A,I,B,J,C,K,MC,NC,NA)

MATRIX MULTIPLY (DIMENSION <=32)

Page	Routine	Purpose
	A = Source matrix base address I = A address increment B = Source matrix base address J = B address increment C = Destination matrix base address K = C address increment MC = Number of rows in C (Rows in A) NC = Number of columns in C (Columns in B) NA = Number of columns in A (<=32) (Rows in B)	To multiply two matrices with dimensions <=32.
E-141	CALL MATINV(A,N) A = Source matrix base address A + N*N = Destination matrix base address N = Numbers of rows (and columns) in A	MATRIX INVERSE To invert a matrix.
E-143	CALL SOLVEQ(A,N,B,M,ROWADD,X,IERR) A = Coefficient matrix base address N = Number of rows (and columns) in A B = Base address of matrix of M N-element right hand sides M = Number of N-element solution vectors ROWADD = Base address of 2*N-element work vector for row addresses X = Base address for matrix of M N-element solution vectors IERR = Address of singularity value	LINEAR EQUATION SOLVER To solve a system of simultaneous linear equations.
E-145	CALL MVML3(A,I,B,J,JP,C,K,KP,N) A = 3x3 matrix base address I = A address increment B = First source vector base address J = Increment between the three elements in each vector of B JP = Increment between the first element of each vector of B C = First destination vector base address K = Increment between the three elements in each vector in C KP = Increment between the first element of each vector in C N = Number of 3-element vectors	MATRIX VECTOR MULTIPLY (3X3) To multiply a 3x3 matrix by a series of 3-element column vectors.
E-147	CALL MVML4(A,I,B,J,JP,C,K,KP,N) A = 4x4 matrix base address I = A address increment B = First source vector base address J = Increment between the three elements in each vector of B JP = Increment between the first element of each vector of B C = First destination vector base address	MATRIX VECTOR MULTIPLY (4X4) To multiply a 4x4 matrix by a series of 4-element column vectors.

Page	Routine	Purpose

K = Increment between the three elements in each vector in C
KP = Increment between the first element of each vector in C
N = Number of 4-element vectors

E-149 CALL CTRN3(A,B,J,JP,C,D,L,LP,N)

3-DIMENSION COORDINATE TRANSFORMATION
To transform a series of 3-dimensional coordinates (translation and rotation).

A = 3x3 rotation matrix base address
B = First source vector base address
J = Increment between the three elements in each vector of B
JP = Increment between the first elements (x-coordinates) of each vector of B
C = Base address of 3-element translation vector
D = First destination vector base address
L = Increment between the three elements in each vector in D
LP = Increment between the first elements of each vector in D
N = Number of 3-element coordinate vectors

E-151 CALL FMMM(A,B,C,MC,NC,NA)

FAST MEMORY MATRIX MULTIPLY
To multiply two matrices. (Available for 167 ns memory only.)

A = Source matrix base address
B = Source matrix base address
C = Destination matrix base address
MC = Number of rows in C (Rows in A)
NC = Number of columns in C (Columns in B)
NA = Number of columns in A (Rows in B)

E-153 CALL FMMM32(A,B,C,MC,NC,NA)

FAST MEMORY MATRIX MULTIPLY (<=32)
To multiply two matrices with dimensions <=32. (Available for 167 ns memory only.)

A = Source matrix base address
B = Source matrix base address
C = Destination matrix base address
MC = Number of rows in C (Rows in A)
NC = Number of columns in C (Columns in B)
NA = Number of columns in A (<=32) (Rows in B)

FFT OPERATIONS

E-156 CALL CFFT(C,N,F)

COMPLEX TO COMPLEX FFT (IN PLACE)
To perform an in-place complex forward or inverse fast Fourier

C = Source and destination vector base address

Page	Routine	Purpose
	N = Complex element count (power of 2) F = Direction flag, +1 for forward -1 for inverse	transform (FFT).
E-158	CALL CFFTB(A,C,N,F) A = Source vector base address C = Destination vector base address N = Complex element count (power of 2) F = Direction flag, +1 for forward -1 for inverse	COMPLEX TO COMPLEX FFT (NOT IN PLACE) To perform a not-in-place complex forward or inverse fast Fourier transform (FFT).
E-160	CALL RFFT(C,N,F) C = Source and destination vector base address N = Real element count (power of 2) F = Direction flag, +1 for forward -1 for inverse	REAL TO COMPLEX FFT (IN PLACE) To perform an in-place real-to-complex forward or a complex-to-real inverse fast Fourier transform (FFT).
E-162	CALL RFFTB(A,C,N,F) A = Source vector base address C = Destination vector base address N = Real element count (power of 2) F = Direction flag, +1 for forward -1 for inverse	REAL TO COMPLEX FFT (NOT IN PLACE) To perform a not-in-place real-to-complex forward or a complex-to-real inverse fast Fourier transform (FFT).
E-164	CALL CFFTSC(C,N) C = Source and destination vector base address N = Complex element count (power of 2)	COMPLEX FFT SCALE To scale complex-to-complex forward FFT results.
E-165	CALL RFFTSC(C,N,F,FS) C = Source and destination vector base address N = Real element count (power of 2) F = Formatting flag 1,0,-1 = No format change 2 = Unpack RFFT result into N/2 complex elements 3 = Unpack RFFT result into N/2 + 1 complex elements -2 = Pack N/2 complex elements into RFFT format -3 = Pack N/2 + 1 complex elements into RFFT format FS = Scaling flag 0 = No scaling 1 = Multiply by 1/(2*N) -1 = Multiply by 1/(4*N)	REAL FFT SCALE AND FORMAT To scale real-to complex FFT results and/or change a complex vector between the special RFFT complex format and the normal complex vector format.
E-167	CALL CFFT2D(C,N1,N2,F) C = source and destination array address N1 = Number of columns = length of rows N2 = Number of rows = length of columns (Note: N1*N2 <= 32768)	COMPLEX TO COMPLEX 2-DIMENSIONAL FFT Two perform an in place complex two-dimensional FFT on rectangular arrays which occupy no more than 65536 main data

Page Routine Purpose

F = Forward-Inverse flag

memory locations (one page).

E-169 CALL RFFT2D(C,N1,N2,F)
C = Source and destination array address
N1 = Number of columns = length of rows
N2 = Number of rows = length of columns
(Note: N1*N2 <= 65536)
F = Forward-inverse flag

REAL TO COMPLEX 2-DIMENSIONAL FFT
To perform an in place real two-dimensional FFT on rectangular arrays which occupy no more than 65536 main data memory locations (one page).

AUXILIARY OPERATIONS

E-172 CALL CONV(A,I,B,J,C,K,N,M)
A = Operand vector base address
I = A address increment
B = Operator vector base address
J = B address increment
C = Destination vector base address
K = C address increment
N = Element count for C (result)
M = Element count for B (operator)
(Element count for A (operand) must be N+M-1)

CONVOLUTION (CORRELATION)
To perform a convolution or correlation operation on two vectors.

E-174 CALL DEQ22(A,I,B,C,K,N)
A = Source vector base address
I = A address increment
B = Base address of 5 filter coefficients
C = Destination vector base address
K = C address increment
N = Element count

DIFFERENCE EQUATION, 2 POLES, 2 ZEROS
To perform a 2-pole, 2-zero recursive digital filtering difference equation on a vector.

E-175 CALL VPOLY(A,I,B,J,C,K,N,P)
A = Coefficient vector base address
(Highest order coefficient is first)
I = A address increment
B = Source vector base address
J = B address increment
C = Destination vector base address
K = C address increment
N = Element count (of B and C)
P = Order of polynomial (>1)

VECTOR POLYNOMIAL EVALUATION
To evaluate a vector polynomial.

E-177 CALL VSUM(A,I,C,K,N,H)
A = Source vector base address
I = A address increment
C = Destination vector base address
K = C address increment
N = Element count
H = Address of integration step size

VECTOR SUM OF ELEMENTS INTEGRATION
To integrate a vector by performing a running scaled sum of the elements of the vector.

E-178 CALL VTRAPZ(A,I,C,K,N,H)

VECTOR TRAPEZOIDAL RULE INTEGRATION

Page	Routine	Purpose
	A = Source vector base address I = A address increment C = Destination vector base address K = C address increment N = Element count H = Address of integration step size	To integrate a vector by using the trapezoidal rule.
E-179	CALL VSIMPS(A,I,C,K,N,H) A = Source vector base address I = A address increment C = Destination vector base address K = C address increment N = Element count H = Address of integration step size	VECTOR SIMPSONS 1/3 RULE INTEGRATION To integrate a vector by using Simpson's 1/3 rule.
E-180	CALL WIENER(LR,R,G,F,A,ISW) LR = Filter length R = Source vector base address (auto-correlation coefficients) G = Source vector base address (cross-correlation) F = Destination vector base address (filter weighting coefficients) A = Destination vector base address (prediction error operator) ISW = Algorithm switch 0 for spike deconvolution 1 for general deconvolution	WIENER LEVINSON ALGORITHM To solve a system of single channel normal equations which arise in least squares filtering and prediction problems.

SIGNAL PROCESSING OPERATIONS (optional)

Page	Routine	Purpose
E-183	CALL HIST(A,I,C,N,NB,AMAX,AMIN) A = Source vector base address I = A address increment C = Histogram vector base address N = Element count for A NB = Element count (bins) in C AMAX = Address of maximum histogram value AMIN = Address of minimum histogram value	HISTOGRAM To perform a histogram on a vector.
E-184	CALL HANN(A,I,C,K,N,F) A = Source vector base address I = A address increment C = Destination vector base address K = C address increment N = Element count (a power of 2) F = Normalization flag F=0 means unnormalized Hanning window (peak window value=1.0) F=1 means normalized Hanning window (peak window value=1.63)	HANNING WINDOW MULTIPLY To multiply a vector by a Hanning window.

Page	Routine	Purpose
E-186	CALL ASPEC(A,C,N) A = Source complex vector base address C = Destination real vector base address N = Element count (Note vector elements occupy consecutive addresses.)	ACCUMULATING AUTO-SPECTRUM To perform accumulating auto-spectrum calculation on a complex vector.
E-187	CALL CSPEC(A,B,C,N) A = Source vector base address B = Source vector base address C = Destination vector base address N = Element count (Note vector elements occupy consecutive addresses.)	ACCUMULATING CROSS-SPECTRUM To perform accumulating cross-spectrum calculation on two complex vectors.
E-188	CALL VAVLIN(A,I,B,C,K,N) A = Source vector base address I = A address increment B = Address for number of vectors included in current average C = Averaged vector base address K = C address increment N = Element count	VECTOR LINEAR AVERAGING To update the linear average of a sequence of vectors to include a new vector.
E-189	CALL VAVEXP(A,I,B,C,K,N) A = Source vector base address I = A address increment B = Address for discount factor C = Averaged vector base address K = C address increment N = Element count	VECTOR EXPONENTIAL AVERAGING To update the approximately exponential average of a sequence of vectors to include a new vector.
E-190	CALL VDBPWR(A,I,B,C,K,N) A = Source vector base address I = A address increment B = Address of scalar reference (0 dB) value C = Destination vector base address K = C address increment N = Element count	VECTOR CONVERSION TO DB (POWER) To compute the decibel (power) equivalents of the elements of a vector, relative to a specified scalar value.
E-191	CALL TRANS(A,B,C,N) A = Auto-spectrum base address (real) B = Cross-spectrum base address (complex) C = Complex transfer function base address N = Element count (Note vector elements occupy consecutive addresses.)	TRANSFER FUNCTION To perform a complex transfer function calculation by dividing the cross-spectrum by the auto-spectrum.
E-192	CALL COHER(A,B,C,D,N) A = Auto-spectrum base address (real) B = Auto-spectrum base address (real) C = Cross-spectrum base address (complex)	COHERENCE FUNCTION To compute the coherence function, given the auto-spectra of two signals and

Page	Routine	Purpose
	D = Coherence function base address (real) N = Element count (Note vector elements occupy consecutive addresses.)	the cross-spectrum between them.
E-193	CALL ACORT(A,C,N,M) A = Source vector base address C = Destination vector base address N = Element count for C (number of lags) M = Element count for A (Note vector elements occupy consecutive addresses.)	AUTO-CORRELATION (TIME-DOMAIN) To perform an auto-correlation operation on a vector using time-domain techniques.
E-195	CALL ACORF(A,C,N,M) A = Source vector base address C = Destination vector base address N = Element count for C (number of lags) M = Element count for A (power of 2) (Note vector elements occupy consecutive addresses. Requires 2M words storage for A.)	AUTO-CORRELATION (FREQUENCY-DOMAIN) To perform an auto-correlation operation on a vector using frequency-domain (FFT) techniques.
E-197	CALL CCORT(A,B,C,N,M) A = Source vector (operand) base address B = Source vector (operator) base address C = Destination vector base address N = Element count for C (number of lags) M = Element count for A and B (Note vector elements occupy consecutive addresses.)	CROSS-CORRELATION (TIME-DOMAIN) To perform a cross-correlation operation on two vectors using time-domain techniques.
E-199	CALL CCORF(A,B,C,N,M) A = Source vector (operand) base address B = Source vector (operator) base address C = Destination vector base address N = Element count for C (number of lags) M = Element count for A and B (power of 2) (Note vector elements occupy consecutive addresses. Requires 2M words storage for A and 2M words storage for B.)	CROSS-CORRELATION (FREQUENCY-DOMAIN) To perform an cross-correlation operation on two vectors using frequency-domain (FFT) techniques.
E-201	CALL TCONV(A,I,B,J,C,K,N,M,L) A = Source (operand) vector base address I = A address increment (>0) B = Source (operator) vector base address J = B address increment C = Destination vector base address K = C address increment N = Element count for C (result) M = Element count for B (operator) L = Element count for A (operand)	POSTTAPERED CONVOLUTION (CORRELATION) To perform a post-tapered convolution or correlation operation on two vectors.

Page	Routine	Purpose

TABLE MEMORY OPERATIONS (optional)

Page	Routine	Purpose
E-204	CALL MTMOV(A,C,N) A = Source vector base address (MD) C = Destination vector base address (TM) N = Element count	VECTOR MOVE (MD TO TM) To transfer elements of a vector from main data to table memory, where both vectors are stored compactly.
E-205	CALL TMMOV(A,C,N) A = Source vector base address (TM) C = Destination vector base address (MD) N = Element count	VECTOR MOVE (TM TO MD) To transfer elements of a vector from table memory to main data memory, where both vectors are stored compactly.
E-206	CALL MTIMOV(A,I,C,K,N) A = Source vector base address (MD) I = A address increment C = Destination vector base address (TM) K = C address increment N = Element count	VECTOR MOVE WITH INCREMENT (MD TO TM) To move elements of a vector from main data memory to table memory, where the increments between the elements are specified.
E-207	CALL TMIMOV(A,I,C,K,N) A = Source vector base address (TM) I = A address increment C = Destination vector base address (MD) K = C address increment N = Element count	VECTOR MOVE WITH INCREMENT (TM TO MD) To move elements of a vector in table memory to main data memory, where the increments between elements are specified.
E-208	CALL TTIMOV(A,I,C,K,N) A = Source vector base address (TM) I = A address increment C = Destination vector base address (TM) K = C address increment N = Element count	VECTOR MOVE WITH INCREMENT (TM TO TM) To move elements of a vector within table memory.
E-209	CALL MMTADD(A,I,B,J,C,K,N) A = Source vector base address (MD) I = A address increment B = Source vector base address (MD) J = B address increment C = Destination vector base address (TM) K = C address increment N = Element count	VECTOR ADD (MD+MD TO TM) To add the elements of two vectors in main data memory and store the results in a vector in table memory.
E-210	CALL MMTSUB(A,I,B,J,C,K,N) A = Source vector base address (MD) I = A address increment B = Source vector base address (MD) J = B address increment C = Destination vector base address (TM) K = C address increment N = Element count	VECTOR SUBTRACT (MD-MD TO TM) To subtract the elements of two vectors in main data memory and store the results in a vector in table memory.

Page	Routine	Purpose
E-211	CALL MMTMUL(A,I,B,J,C,K,N) A = Source vector base address (MD) I = A address increment B = Source vector base address (MD) J = B address increment C = Destination vector base address (TM) K = C address increment N = Element count	VECTOR MULTIPLY (MD*MD TO TM) To multiply the elements of two vectors in main data memory and store the results in table memory.
E-212	CALL MTMADD(A,I,B,J,C,K,N) A = Source vector base address (MD) I = A address increment B = Source vector base address (TM) J = B address increment C = Destination vector base address (MD) K = C address increment N = Element count	VECTOR ADD (MD+TM TO MD) To add elements of a vector in main data memory to elements of a vector in table memory and store the results in main data memory.
E-213	CALL MTMSUB(A,I,B,J,C,K,N) A = Source vector base address (MD) I = A address increment B = Source vector base address (TM) J = B address increment C = Destination vector base address (MD) K = C address increment N = Element count	VECTOR SUBTRACT (MD-TM TO MD) To subtract the elements of a vector in table memory from the elements of a vector in main data memory and store the results in main data memory.
E-214	CALL TMMSUB(A,I,B,J,C,K,N) A = Source vector base address (TM) I = A address increment B = Source vector base address (MD) J = B address increment C = Destination vector base address (MD) K = C address increment N = Element count	VECTOR SUBTRACT (TM-MD TO MD) To subtract the elements of a vector in main data memory from a vector in table memory and store the differences in main data memory.
E-215	CALL MTMMUL(A,I,B,J,C,K,N) A = Source vector base address (MD) I = A address increment B = Source vector base address (TM) J = B address increment C = Destination vector base address (MD) K = C address increment N = Element count	VECTOR MULTIPLY (MD*TM TO MD) To multiply elements of a vector in main data memory by elements of a vector in table memory and store the products in main data memory.
E-216	CALL MTTADD(A,I,B,J,C,K,N) A = Source vector base address (MD) I = A address increment B = Source vector base address (TM) J = B address increment C = Destination vector base address (TM) K = C address increment N = Element count	VECTOR ADD (MD+TM TO TM) To add the elements of a vector in main data memory to elements of a vector in table memory and store the sums in a vector in table memory.

Page	Routine	Purpose
E-217	CALL MTTSUB(A,I,B,J,C,K,N) A = Source vector base address (MD) I = A address increment B = Source vector base address (TM) J = B address increment C = Destination vector base address (TM) K = C address increment N = Element count	VECTOR SUBTRACT (MD-TM TO TM) To subtract the elements of a vector in table memory from elements of a vector in main data memory and store the differences in table memory.
E-218	CALL TMTSUB(A,I,B,J,C,K,N) A = Source vector base address (TM) I = A address increment B = Source vector base address (MD) J = B address increment C = Destination vector base address (TM) K = C address increment N = Element count	VECTOR SUBTRACT (TM-MD TO TM) To subtract the elements of a vector in main data memory from the elements of a vector in table memory and store the results in table memory.
E-219	CALL MTTMUL(A,I,B,J,C,K,N) A = Source vector base address (MD) I = A address increment B = Source vector base address (TM) J = B address increment C = Destination vector base address (TM) K = C address increment N = Element count	VECTOR MULTIPLY (MD*TM TO TM) To multiply the elements of a vector in main data memory by the elements of a vector in table memory and store the products in table memory.
E-220	CALL TTMADD(A,I,B,J,C,K,N) A = Source vector base address (TM) I = A address increment B = Source vector base address (TM) J = B address increment C = Destination vector base address (MD) K = C address increment N = Element count	VECTOR ADD (TM+TM TO MD) To add the elements of two vectors in table memory and store the sums in main data memory.
E-221	CALL TTMSUB(A,I,B,J,C,K,N) A = Source vector base address (TM) I = A address increment B = Source vector base address (TM) J = B address increment C = Destination vector base address (MD) K = C address increment N = Element count	VECTOR SUBTRACT (TM-TM TO MD) To subtract the elements of two vectors in table memory and store the difference in main data memory.
E-222	CALL TTMMUL(A,I,B,J,C,K,N) A = Source vector base address (TM) I = A address increment B = Source vector base address (TM) J = B address increment C = Destination vector base address (MD) K = C address increment N = Element count	VECTOR MULTIPLY (TM*TM TO MD) To multiply the elements of two vectors in table memory and store the products in main data memory.

Page	Routine	Purpose
E-223	CALL TTTADD(A,I,B,J,C,K,N) A = Source vector base address (TM) I = A address increment B = Source vector base address (TM) J = B address increment C = Destination vector base address (TM) K = C address increment N = Element count	VECTOR ADD (TM+TM TO TM) To add the elements of two vectors in table memory and store the sums in a third vector in table memory.
E-224	CALL TTTSUB(A,I,B,J,C,K,N) A = Source vector base address (TM) I = A address increment B = Source vector base address (TM) J = B address increment C = Destination vector base address (TM) K = C address increment N = Element count	VECTOR SUBTRACT (TM-TM TO TM) To subtract the elements of two vectors in table memory and store the differences in a vector in table memory.
E-225	CALL TTTMUL(A,I,B,J,C,K,N) A = Source vector base address (TM) I = A address increment B = Source vector base address (TM) J = B address increment C = Destination vector base address (TM) K = C address increment N = Element count	VECTOR MULTIPLY (TM*TM TO TM) To multiply the elements of two vectors in table memory and store the products in a vector in table memory.

APPENDIX D

AP-FORTRAN ROUTINES

Many of the routines in the AP Math Library are available for use in AP-FORTRAN program units. These routines contain alternate entry points permitting AP-FORTRAN program units to call them. Because of these alternate entry points, the routines are called by different names under AP-FORTRAN. A list of the routines' names and their corresponding AP-FORTRAN calling names is contained in Table D-1. This table lists all routines callable from AP-FORTRAN program units. The parameters associated with the routines are described in Appendices C and E. Regarding the associated parameters, the AP-FORTRAN user should be aware of the following:

- The data transfer and control operations and the APAL-callable utility operations are not available under AP-FORTRAN. The data transfer and control operations are not needed by the AP-FORTRAN user. The AP-FORTRAN program unit executes in the AP, and thus transferring data and controlling operations are already provided for. The AP-FORTRAN programmer does not need to place data into the AP using APPUT or retrieve it using APGET; the data can be made available to the routines by passing common blocks or defining values in the AP-FORTRAN program unit.

- In Appendices C and E, when parameters are described as base addresses, the AP-FORTRAN user should substitute the term "name". For example, the term "source vector base address" translates into "source vector (or array) name". The name specified should be the name of a properly dimensioned array.

- Parameters which are described as values in Appendices C and E can be specified as variable names under AP-FORTRAN. All routines are called by reference under AP-FORTRAN.

Table D-1 AP-FORTRAN Callable Math Library Routines

ROUTINE	DESCRIPTION	AP-FORTRAN CALLABLE NAME
ACORF	Auto-correlation (frequency-domain)	FFACOR(a,c,n,m)
ACORT	Auto-correlation (time-domain)	FTACOR(a,c,n,m)
ASPEC	Accumulating auto-spectrum	FASPEC(a,c,n)
CCORF	Cross-correlation (frequency-domain)	FFCCOR(a,b,c,n,m)
CCORT	Cross-correlation (time-domain)	FTCCOR(a,b,c,n,m)
CDOTPR	Complex dot product	FCDOTP(a,i,b,j,n)
CFFT	Complex to complex FFT (in place)	FCFFT(c,n,f)
CFFTB	Complex to complex FFT (not in place)	FBCFFT(a,c,n,f)
CFFTSC	Complex FFT scale	FCCFFT(c,n)
COHER	Coherence function	FCOHER(a,b,c,d,n)
CONV	Convolution (correlation)	FCONV(a,i,b,j,c,k,n,m)
CRVADD	Complex and real vector add	FCRVAD(a,i,b,j,c,k,n)
CRVDIV	Complex and real vector divide	FCRVDI(a,i,b,j,c,k,n)
CRVMUL	Complex and real vector multiply	FCRVMU(a,i,b,j,c,k,n)
CRVSUB	Complex and real vector subtract	FCRVSU(a,i,b,j,c,k,n)
CSPEC	Accumulating cross-spectrum	FCSPEC(a,b,c,n)
CTRN3	3-dimensional coordinate transformation	FCTRN3(a,b,j,jp,c,d,l,lp,n)
CVADD	Complex vector add	FCVADD(a,i,b,j,k,n)
CVCOMB	Complex vector combine	FCVCMB(a,i,b,j,c,k,n)
CVCONJ	Complex vector conjugate	FCVCNJ(a,i,c,k,n)
CVFILL	Complex vector fill	FCVFIL(a,c,k,n)
CVMA	Complex vector multiply and add	FCVMCA(a,i,b,j,c,k,d,l,n,f)
CVMAGS	Complex vector magnitude squared	FCVMGS(a,i,c,k,n)
CVMOV	Complex vector move	FCVMOV(a,i,c,k,n)
CVMUL	Complex vector multiply	FCVMUL(a,i,b,j,c,k,n,f)
CVNEG	Complex vector negate	FCVNEG(a,i,c,k,n)
CVRCIP	Complex vector reciprocal	FCVRCI(a,i,c,k,n)
CVREAL	From complex vector of reals	FCVREAL(a,i,c,k,n)
CVSMUL	Complex vector scalar multiply	FCVSMU(a,i,b,c,k,n)
CVSUB	Complex vector subtract	FCVSUB(a,i,b,j,c,k,n)
DAREAD	Read device address register	FDARED(da)
DAWRIT	Write device address register	FDAWRT(da,val)
DEQ22	Difference equation, 2 poles, 2 zeros	FDEQ22(a,i,b,c,k,n)
DOTPR	Dot product	FDOTPR(a,i,b,j,c,n)

0812

Table D-1 AP-FORTRAN Callable Math Library Routines (cont.)

ROUTINE	DESCRIPTION	AP-FORTRAN CALLABLE NAME
ECVMUL	Extended complex vector multiply	FECVMU(ah,al,i,bh,bl,j,ch,cl,k,nh,nl,f)
EDOTPR	Extended dot product	FEDTPR(ah,al,i,bh,bl,j,ch,cl,nh,nl)
EMMUL	Extended matrix multiply	FEMMUL(ah,al,bh,bl,ch,cl,mc,nc,na)
EMTRAN	Extended matrix transpose	FEMTRN(ah,al,ch,cl,k,mc,nc)
EVADD	Extended vector add	FEVADD(ah,al,i,bh,bl,j,ch,cl,k,nh,nl)
EVCLR	Extended vector clear	FEVCLR(ch,cl,k,nh,nl)
EVDIV	Extended vector divide	FEVDIV(ah,al,i,bh,bl,j,ch,cl,k,nh,nl)
EVMOV	Extended vector move	FEVMOV(ah,al,i,ch,cl,k,nh,nl)
EVMUL	Extended vector multiply	FEVMUL(ah,al,i,bh,bl,j,ch,cl,k,nh,nl)
EVSUB	Extended vector subtract	FEVSUB(ah,al,i,bh,bl,j,ch,cl,k,nh,nl)
EVSWAP	Extended vector swap	FEVSWP(ah,al,i,ch,cl,k,nh,nl)
FMMM	Fast memory matrix multiply	FFMMM(a,b,c,mc,nc,na)
FMMM32	Fast memory matrix multiply (<=32)	FFMM32(a,b,c,mc,nc,na)
HANN	Hanning window multiply	FHANN(a,i,c,k,n,f)
HIST	Histogram	FHIST(a,i,c,n,nb,hmax,hmin)
IOPGET	Get data from AP MD out through IOP	FIOPGT(exma,apma,n)
IOPPUT	Put data into AP MD from IOP	FIOPPU(exma,apma,n)
IOPWD	Wait for IOP data transfer	FIOPWD
MATINV	Matrix inverse	FMATIN(a,n)
MAXMGV	Maximum magnitude element in vector	FMXMGV(a,i,c,n)
MAXV	Maximum element in vector	FMAXV(a,i,c,n)
MDCOM	Main data compare and set S-pad	FMDCOM(a,b)
MEAMGV	Mean of vector element magnitudes	FMEMGV(a,i,c,n)
MEANV	Mean value of vector elements	FMEANV(a,i,c,n)
MEASQV	Mean of vector element squares	FMESQV(a,i,c,n)
MINMGV	Minimum magnitude element in vector	FMNGV(a,i,c,n)
MINV	Minimum element in vector	FMINV(a,i,c,n)
MMTADD	Vector add (MD+MD to TM)	FAMMT(a,i,b,j,c,k,n)
MMTMUL	Vector multiply (MD*MD to TM)	FMMMT(a,i,b,j,c,k,n)
MMTSUB	Vector subtract (MD-MD to TM)	FSMMT(a,i,b,j,c,k,n)
MMUL	Matrix multiply	FMMUL(a,i,b,j,c,k,mc,nc,na)
MMUL32	Matrix multiply (dimension<=32)	FMMU32(a,i,b,j,c,k,mc,nc,na)
MTIMOV	Vector move with increment (MD to TM)	FMTIMO(a,i,c,k,n)
MTMADD	Vector add (MD+TM to MD)	FAMTMD(a,i,b,j,c,k,n)

0813

Table D-1 AP-FORTRAN Callable Math Library Routines (cont.)

ROUTINE	DESCRIPTION	AP-FORTRAN CALLABLE NAME
MTMMUL	Vector multiply (MD*TM to MD)	FMMTMD(a,i,b,j,c,k,n)
MTMOV	Vector move (MD to TM)	FMTMOV(a,c,n)
MTMSUB	Vector subtract (MD-TM to MD)	FSMTMD(a,i,b,j,c,k,n)
MTRANS	Matrix transpose	FMTRNS(a,i,c,k,mc,nc)
MTTADD	Vector add (MD+TM to TM)	FAMTT(a,i,b,j,c,k,n)
MTTMUL	Vector multiply (MD*TM to TM)	FMMTT(a,i,b,j,c,k,n)
MTTSUB	Vector subtract (MD-TM to TM)	FSMTT(a,i,b,j,c,k,n)
MVML3	Matrix vector multiply (3x3)	FMVMU3(a,i,b,j,jp,c,k,kp,n)
MVML4	Matrix vector multiply (4x4)	FMVML4(a,i,b,j,jp,c,k,kp,n)
POLAR	Rectangular to polar conversion	FPOLAR(a,i,c,k,n)
RDC5	Read control bit 5 interrupt	FRDC5(c)
RDPAR	Read parity registers	FRDPAR(c)
RDPG	Read memory page from AP	FRDPG(c)
RECT	Polar to rectangular conversion	FRECT(a,i,c,k,n)
RFFT	Real to complex FFT (in place)	FRFFT(c,n,f)
RFFTB	Real to complex FFT (not in place)	FBRFFT(a,c,n,f)
RFFTSC	Real FFT scale and format	FCRFFT(c,n,f,fs)
RMSQV	Root-mean-square of vector elements	FRMSQV(a,i,c,n)
SCJMA	Self-conjugate multiply and add	FSCJMA(a,i,b,j,c,k,n)
SETC5	Set control bit 5 interrupt	FSETC5
SETPG	Set memory page for AP	FSETPG(mask,apmae,mae)
SOLVEQ	Linear equation solver	FSOVEQ(a,n,b,m,rowadd,x,ierr)
SVE	Sum of vector elements	FSVE(a,i,c,n)
SVEMG	Sum of vector element magnitudes	FSVEMG(a,i,c,n)
SVESQ	Sum of vector element squares	FSVESQ(a,i,c,n)
SVS	Sum of vector signed squares	FSVS(a,i,c,n)
TCONV	Post-tapered convolution (correlation)	FTCONV(a,i,b,j,c,k,n,m,l)
TMIMOV	Vector move with increment (TM to MD)	FTMIMO(a,i,c,k,n)
TMMOV	Vector move (TM to MD)	FTMMOV(a,c,n)
TMMSUB	Vector subtract (TM-MD to MD)	FSTMMD(a,i,b,j,c,k,n)
TMTSUB	Vector subtract (TM-MD to TM)	FSTMT(a,i,b,j,c,k,n)
TRANS	Transfer function	FTRANS(a,b,c,n)
TTIMOV	Vector move with increment (TM to TM)	FTTIMO(a,i,c,k,n)
TTMADD	Vector add (TM+TM to MD)	FATTMD(a,i,b,j,c,k,n)

0814

Table D-1 AP-FORTRAN Callable Math Library Routines (cont.)

ROUTINE	DESCRIPTION	AP-FORTRAN CALLABLE NAME
TTMMUL	Vector multiply (TM*TM to MD)	FMTTMD(a,i,b,j,c,k,n)
TTMSUB	Vector subtract (TM-TM to MD)	FSTTMD(a,i,b,j,c,k,n)
TTTADD	Vector add (TM+TM to TM)	FATTT(a,i,b,j,c,k,n)
TTTMUL	Vector multiply (TM*TM to TM)	FMTTT(a,i,b,j,c,k,n)
TTTSUB	Vector subtract (TM-TM to TM)	FSTTT(a,i,b,j,c,k,n)
VAAM	Vector add, add, and multiply	FVAAM(a,i,b,j,c,k,d,l,e,m,n)
VABS	Vector absolute value	FVABS(a,i,c,k,n)
VADD	Vector add	FVADD(a,i,b,j,c,k,n)
VALOG	Vector antilogarithm (base 10)	FVALOG(a,i,c,k,n)
VAM	Vector add and multiply	FVAM(a,i,b,j,c,k,d,l,n)
VAND	Vector logical and	FVAND(a,i,b,j,c,k,n)
VATAN	Vector arctangent	FVATAN(a,i,c,k,n)
VATN2	Vector arctangent of y/x	FVATN2(a,i,b,j,c,k,n)
VAVEXP	Vector exponential averaging	FVAVEX(a,i,b,c,k,n)
VAVLIN	Vector linear averaging	FVAVLN(a,i,b,c,k,n)
VCLIP	Vector clip	FVCLIP(a,i,b,c,d,l,n)
VCLR	Vector clear	FVCLR(c,k,n)
VCOS	Vector cosine	FVCOS(a,i,c,k,n)
VDBPWR	Vector conversion to DB (power)	FVDBPR(a,i,b,c,k,n)
VDIV	Vector divide	FVDIV(a,i,b,j,c,k,n)
VEQV	Vector logical equivalence	FVEQV(a,i,b,j,c,k,n)
VEXP	Vector exponential	FVEXP(a,i,c,k,n)
VFILL	Vector fill	FVFILL(a,c,k,n)
VFIX	Vector integer fix	FVFIX(a,i,c,k,n)
VFIX32	Vector 32 bit integer fix	FVFX32(a,i,c,k,n)
VFLT	Vector integer float	FVFLT(a,i,c,k,n)
VFLT32	Vector 32 bit integer float	FVFL32(a,i,c,k,n)
VFRAC	Vector truncate to fraction	FVFRAC(a,i,c,k,n)
VICLIP	Vector inverted clip	FVICLP(a,i,b,c,d,l,n)
VIMAG	Extract imaginaries of complex vector	FVIMAG(a,i,c,k,n)
VINDEX	Vector index	FVINDX(a,b,j,c,k,n)
VINT	Vector truncate to integer	FVINT(a,i,c,k,n)
VLIM	Vector limit	FVLIM(a,i,b,c,d,l,n)
VLN	Vector natural logarithm	FVLN(a,i,c,k,n)

0815

Table D-1 AP-FORTRAN Callable Math Library Routines (cont.)

ROUTINE	DESCRIPTION	AP-FORTRAN CALLABLE NAME
VLOG	Vector logarithm (base 10)	FVLOG(a,i,c,k,n)
VMA	Vector multiply and add	FVMVA(a,i,b,j,c,k,d,l,n)
VMAX	Vector maximum	FVMAX(a,i,b,j,c,k,n)
VMAXMG	Vector maximum magnitude	FVMGAX(a,i,b,j,c,k,n)
VMIN	Vector miminum	FVMIN(a,i,b,j,c,k,n)
VMINMG	Vector minimum magnitude	FVMGIN(a,i,b,j,c,k,n)
VMMA	Vector multiply, multiply and add	FVMMA(a,i,b,j,c,k,d,l,e,m,n)
VMMSB	Vector multiply, multiply and subtract	FVMMSB(a,i,b,j,c,k,d,l,e,m,n)
VMOV	Vector move	FVMOV(a,i,c,k,n)
VMSB	Vector multiply and subtract	FVMSB(a,i,b,j,c,k,d,l,n)
VMUL	Vector multiply	FVMUL(a,i,b,j,c,k,n)
VNEG	Vector negate	FVNEG(a,i,c,k,n)
VOR	Vector logical or	FVOR(a,i,b,j,c,k,n)
VPK16	Vector 16 bit byte pack	FVPK16(a,i,c,k,n)
VPK8	Vector 8 bit byte pack	FVPK8(a,i,c,k,n)
VPOLY	Vector polynomial evaluation	FVPOLY(a,i,b,j,c,k,n,p)
VRAMP	Vector ramp	FVRAMP(a,b,c,k,n)
VRAND	Vector random numbers	FVRAND(a,c,k,n)
VREAL	Extract reals of complex vector	FVREAL(a,i,c,k,n)
VSADD	Vector scalar add	FVSADD(a,i,b,c,k,n)
VSBM	Vector subtract and multiply	FVSBM(a,i,b,j,c,k,d,l,n)
VSBSBM	Vector subtract, subtract and multiply	FVSB2M(a,i,b,j,c,k,d,l,e,m,n)
VSCALE	Vector scale (power 2) and fix	FVSCLE(a,i,c,k,n,nb)
VSCSCL	Vector scan, scale (power 2) and fix	FVSNSL(a,i,c,k,n,wdth)
VSEFLT	Vector sign extend and float	FVSEFL(a,i,c,k,n)
VSHFX	Vector shift and fix	FVSHFX(a,i,c,k,n,ns)
VSIMPS	Vector Simpson's 1/3 rule integration	FVSIMP(a,i,c,k,n,h)
VSIN	Vector sine	FVSIN(a,i,c,k,n)
VSMSA	Vector scalar multiply and scalar add	FVSMSA(a,i,b,c,d,l,n)
VSMUL	Vector scalar multiply	FVSMUL(a,i,b,c,k,n)
VSQ	Vector square	FVSQ(a,i,c,k,n)
VSQRT	Vector square root	FVSQRT(a,i,c,k,n)
VSSQ	Vector signed square	FVSSQ(a,i,c,k,n)
VSUB	Vector subtract	FVSUB(a,i,b,j,c,k,n)

0816

Table D-1 AP-FORTRAN Callable Math Library Routines (cont.)

ROUTINE	DESCRIPTION	AP-FORTRAN CALLABLE NAME
VSUM	Vector sum of elements integration	FVSUM(a,i,c,k,n,h)
VSWAP	Vector swap	FVSWAP(a,i,c,k,n)
VTRAPZ	Vector trapezoidal rule integration	FVTRAP(a,i,c,k,n,h)
VTSMUL	Vector table scalar multiply	FVTSMU(a,i,b,c,k,n)
VUP16	Vector 16 bit byte unpack	FVU16(a,i,c,k,n)
VUP8	Vector 8 bit byte unpack	FVUP8(a,i,c,k,n)
VUP16	Vector 16 bit signed byte unpack	FVUS16(a,i,c,k,n)
VUPS8	Vector 8 bit signed byte unpack	FVUPS8(a,i,c,k,n)
WIENER	Wiener Levinson algorithm	FWIENR(lr,r,g,f,a,isw)
ZMD	Clear all pages of main data memory	FZMD

0817

Notice to the Reader

- Help us improve the quality and usefulness of this manual.

- Your comments and answers to the following READERS COMMENT form would be appreciated.

To mail: fold the form in three parts so that Floating Point Systems' mailing address is visible; seal.

Thank you

READERS COMMENT FORM

Document Title ______________________________

Your comments and answers will help us improve the quality and usefulness of our publications. If your answers require qualification or additional explanation, please comment in the space provided below.

How did you use this manual?

( ) AS AN INTRODUCTION TO THE SUBJECT
( ) AS AN AID FOR ADVANCED TRAINING
( ) TO LEARN OF OPERATING PROCEDURES
( ) TO INSTRUCT A CLASS
( ) AS A STUDENT IN A CLASS
( ) AS A REFERENCE MANUAL
( ) OTHER ______________________

Did you find this material . . .

	YES	NO
• USEFUL?	( )	( )
• COMPLETE?	( )	( )
• ACCURATE?	( )	( )
• WELL ORGANIZED?	( )	( )
• WELL ILLUSTRATED?	( )	( )
• WELL INDEXED?	( )	( )
• EASY TO READ?	( )	( )
• EASY TO UNDERSTAND?	( )	( )

Please indicate below whether your comment pertains to an addition, deletion, change or error; and, where applicable, please refer to specific page numbers.

Page	Description of error or deficiency

From:

Name ______________________
Firm ______________________
Address ______________________
Telephone ______________________

Title ______________________
Department ______________________
City, State ______________________
Date ______________________

First Class
Permit No. A-737
Portland,
Oregon

BUSINESS REPLY

No postage stamp necessary if mailed in the United States

Postage will be paid by:

FLOATING POINT SYSTEMS, INC.

P.O. Box 23489
Portland, Oregon 97223

Attention: Technical Publications

FLOATING POINT
SYSTEMS, INC.

CALL TOLL FREE 800-547-1445
P.O. Box 23489, Portland, OR 97223
(503) 641-3151, TLX: 360470 FLOATPOINT PTL