https://ntrs.nasa.gov/search.jsp?R=19850012417 2020-03-20T18:47:08+00:00Z

NASA-CR-172281

NASA Contractor Report 172281

NASA-CR-172281
19850012417

PASLIB PROGRAMMER'S GUIDE FOR THE FINITE ELEMENT MACHINE REVISION 2.1-A

Thomas W. Crockett

FOR REFERENCE

NOT TO BE TAKEN FROM THIS ROOM

KENTRON INTERNATIONAL, INC.
Aerospace Technologies Division
Hampton, Virginia 23666

Contract NAS1-16000
April 1984

LIBRARY COPY

MAY 1 1 1984

LANGLEY RESEARCH CENTER
LIBRARY, NASA
HAMPTON, VIRGINIA

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665

Contents

SUMMARY v

1. INTRODUCTION 1

2. PROGRAM CONSIDERATIONS 3

2.1 Writing TI Pascal Programs for Nodal Exec 3
2.2 Floating-point Operations 5
2.3 Linking for Execution on FEM 6

3. SUBROUTINE DESCRIPTIONS 9

3.1 Text Output 11

3.1.1 MSG 12
3.1.2 MSGLN 13
3.1.3 ENDLN 14
3.1.4 NXTLN 15
3.1.5 MSGCH 16
3.1.6 MSGI 17
3.1.7 MSGL 18
3.1.8 MSGR 19
3.1.9 MSGD 20
3.1.10 MSGIH 21
3.1.11 MSGRH 22
3.1.12 MSGDH 23
3.1.13 CWAIT 24

3.2 Interactive Input 25

3.2.1 QUERY 26
3.2.2 RDCH 27
3.2.3 RDH 28
3.2.4 RDI 29
3.2.5 RDR 30

3.3 Data Areas 31

3.3.1 DAPTR 32

3.4 Flags 33

3.4.1 FLGEN 34
3.4.2 FLGDIS 35
3.4.3 FLGRES 36
3.4.4 FLGSET 37
3.4.5 ANY 38
3.4.6 ALL 39
3.4.7 SYNC 40
3.4.8 FIRST 42
3.4.9 BAR 43

N85-20727#

3.5 Am9512 Floating-point Operations 45

3.5.1 ADD 46
3.5.2 SUB 47
3.5.3 MULT 48
3.5.4 DIVD 49
3.5.5 NEG 50
3.5.6 ABS95 51
3.5.7 CMP 52
3.5.8 DADD 53
3.5.9 DSUB 54
3.5.10 DMULT 55
3.5.11 DDIVD 56
3.5.12 DNEG 57
3.5.13 DABS95 58
3.5.14 DCMP 59

3.6 Am9512 Floating-point Constants 60

3.6.1 MAX95 61
3.6.2 MIN95 62
3.6.3 DMAX95 63
3.6.4 DMIN95 64

3.7 Am9512 Floating-point Conversions 65

3.7.1 CV9512 66
3.7.2 CV990 67
3.7.3 FLOATI 68
3.7.4 FLOATL 69
3.7.5 IFIX 70
3.7.6 LFIX 71
3.7.7 SINGLE 72
3.7.8 DV9512 73
3.7.9 DV990 74
3.7.10 DFLOTI 75
3.7.11 DFLOTL 76
3.7.12 IFIXD 77
3.7.13 LFIXD 78
3.7.14 DOUBLE 79

3.8 Am9512 Mathematical Subroutines 80

3.8.1 SQRT95 82
3.8.2 DSQRT95 83
3.8.3 VDP 84
3.8.4 DVDP 86
3.8.5 URAN 87
3.8.6 DURAN 89
3.8.7 RANSEED 90
3.8.8 SINE 91
3.8.9 DSINE 92

3.9 Sum/Maximum 93

3.10 Neighbor Communications 94

3.10.1 SEND 97
3.10.2 SEND2 99
3.10.3 SENDALL 101
3.10.4 SEND2ALL 102
3.10.5 RECV 104
3.10.6 RECV2 106
3.10.7 IO$MODE 108
3.10.8 GBUSY 109

3.11 Timing 110

3.11.1 XTIME 111
3.11.2 XTIME1 112
3.11.3 DLY 114
3.11.4 TSTART 115
3.11.5 TSTOP 116
3.11.6 TREAD 117
3.11.7 TREAD1 119

3.12 Processor Identification 121

3.12.1 PSELF 122
3.12.2 LSELF 123

4. EFFICIENCY CONSIDERATIONS 125

4.1 Compiler Options 125
4.2 Algorithms and Overhead 126
4.2.1 Workload 126
4.2.2 Problem Partitioning 126
4.2.3 Synchronization 127
4.2.4 Communication 127

5. EXECUTION, ANALYSIS, AND DEBUGGING 129

5.1 Problem Setup 129
5.2 Execution Control 129
5.3 Debugging 130
5.4 Analysis 130

Appendices

A. EXAMPLE PROGRAM 133
B. EPROM-RESIDENT SUBROUTINES 145
C. SUBROUTINE REFERENCE SHEET 147

Figures

2-1 Program Structure for FEM 4
2-2 Comparison of TI 990 and Am9512 Floating-point Formats 7
2-3 Link Edit Control File for FEM Execution 8
3-1 Constant and Type Declarations for PASLIB Routines 10
3-2 PASLIB Mathematical Subroutines 81
3-3 Sizes of Commonly Used Pascal Data Types 96

SUMMARY

The Finite Element Machine (FEM) is an experimental parallel computer being built at NASA's Langley Research Center to study the application of concurrent processing to structural engineering analysis. Physically, FEM is composed of two major units, the Array and the Controller. The Array contains up to 36 autonomous microcomputers, each capable of executing its own program on its own data. An elaborate network of special purpose circuitry provides for communication and synchronization among all of these processors. The Controller is an off-the-shelf minicomputer augmented with a special interface to serve as a front-end to the parallel Array.

Three major system software components have been developed for the Finite Element Machine. A package of user-interface and control software known as FACS (FEM Array Control Software) resides on the Controller. A specialized operating system called Nodal Exec is stored in read-only-memory on each of the processors in the Array. A subroutine library named PASLIB allows users' programs running on processors in the Array to access the communication and synchronization hardware, to perform arithmetic using the floating-point unit available on each processor, and to obtain services from Nodal Exec.

This report serves two purposes: (1) to document the PASLIB subroutines and describe their use within the environment provided by Nodal Exec and FACS, and (2) to outline the procedures necessary for developing efficient programs for execution on the Array.

1. INTRODUCTION

PASLIB is a library of subroutines which facilitate the use of the special architectural features of the Finite Element Machine. PASLIB subroutines are invoked by standard procedure and function calls from TI Pascal programs. Most PASLIB routines are written in assembly language for efficiency and compactness, although a few are written in Pascal. Some of the most frequently used PASLIB routines are stored in EPROM on the nodal processors. (See Appendix B.) This technique reduces the size of the object code which must be downloaded from the Controller to the Array, and allows more space for the user's programs and data.

This manual describes how to construct TI Pascal (TIP) programs for parallel execution on the Finite Element Machine using the PASLIB subroutines. It assumes that the reader is familiar with TI Pascal, the FEM architecture, concepts of data and program management on FEM, SCI commands for FEM, and use of the DX10 operating system. The following references contain most of the necessary information.

THE FINITE ELEMENT MACHINE PROGRAMMER'S REFERENCE MANUAL

FEM ARRAY CONTROL SOFTWARE (FACS) USER'S GUIDE

FINITE ELEMENT MACHINE PROGRAMMING MEMORANDA

MODEL 990 COMPUTER TI PASCAL REFERENCE MANUAL

MODEL 990 COMPUTER DX10 TI PASCAL PROGRAMMER'S GUIDE

MODEL 990 COMPUTER DX10 OPERATING SYSTEM, Vols. I - VI

MODEL 990 COMPUTER LINK EDITOR REFERENCE MANUAL

2. PROGRAM CONSIDERATIONS

Several considerations are important when writing TI Pascal programs for execution by processors in the FEM Array. These arise because of hardware and software differences between the TI 990 minicomputer and the FEM processors. TI Pascal is intended primarily for use under the DX10 operating system running on a TI 990 minicomputer. By contrast, processors in the Array are based on the TMS9900 microprocessor, augmented with an Am9512 floating-point chip and many special hardware features, all under control of the Nodal Exec operating system. The two main problems which must be addressed are (1) elimination of any dependencies on DX10, and (2) conversion of all floating-point operations from TI 990 software to Am9512 hardware.

2.1 Writing TI Pascal Programs for Nodal Exec

To eliminate any dependencies on the DX10 operating system, Pascal programs must conform to the requirements for stand-alone execution as described in Chapter 7 of the DX10 TI PASCAL PROGRAMMER'S GUIDE. The most significant restriction is that none of the standard I/O routines such as READ and WRITE are available to stand-alone programs. These are partially compensated for by PASLIB routines which can perform I/O to the Controller (Sections 3.1 and 3.2), and by data areas which can be used to move information between the Controller and processors in the Array (Section 3.3).

Another important requirement is on the structure of the program. A dummy main program is required, and the actual body of the program begins in a procedure called PSCL$$. Figure 2-1 illustrates the structure of TI Pascal programs for FEM. The compiler options NO TRACEBACK and NO ASSERTS will slightly reduce the size of the object code generated for each subroutine, with no loss of capabilities, since TRACEBACK and ASSERT are not supported anyway in stand-alone TIP programs. Many other compiler options could be specified at this point, including WIDELIST, MAP, GLOBALOPT, CKINDEX, etc. The ?COPY statement after the PROGRAM declaration includes constant and type declarations used by PASLIB routines. These declarations may also be referenced by user-written code at lower nesting levels. EXTERNAL declarations for any PASLIB routines referenced by the program must come next.

NOTE

> PASLIB routines must be declared at this level to prevent improper nesting of workspaces on the PASCAL stack. They should not be declared inside PSCL$$ or at lower levels.

Procedure PSCL$$ is defined next. This is the routine in which execution begins, and is effectively the main program from the programmer's standpoint. Declarations and procedures may be nested inside PSCL$$ according to the usual rules for Pascal. Following PSCL$$, the NO OBJECT option is used to suppress generation of the empty main program. A complete sample program is shown in Appendix A.

```
(*$NO TRACEBACK,NO ASSERTS *)

PROGRAM EXAMPLE_1;

?COPY SYS1.FEM.PASLIB.UTIL$.TYPDCL

(*  external declarations for PASLIB routines go here  *)

PROCEDURE PSCL$$;

(*  constant, type, variable, procedure, and function  *)
(*  declarations go here                               *)

BEGIN
(*  actual main program  *)
END;  (* PSCL$$ *)

BEGIN   (*  dummy main program  *)
(*$NO OBJECT *)
END.
```

Figure 2-1. Program Structure for FEM

Execution of the program on FEM is not truly stand-alone, since the Nodal Exec operating system provides control and many support capabilities. To take advantage of this, a special runtime system (called N$MAIN) has been developed to replace the standard P$MAINSA stand-alone runtime system of TI Pascal. N$MAIN includes the ability to report Pascal errors, to record stack and heap information in the execution statistics, and to return control to the Nodal Exec program termination routine (STOP). N$MAIN is included in the link edit control file in place of P$MAINSA (see Section 2.3).

2.2 Floating-point Operations

Floating-point operations in TI Pascal use 32- and 64-bit numbers in TI 990 format. On the 990/10, these operations are performed by a software interpreter. The internal representation uses a sign bit, a 7-bit hexadecimal exponent, and a 24- or 56-bit hexadecimal mantissa. On the FEM processors, floating-point arithmetic is performed by an Am9512 floating-point chip, which also accepts 32- and 64-bit operands. However, the Am9512 format is substantially different from the TI 990 format. For 32-bit numbers, a sign bit, 8-bit binary exponent, and 24-bit binary mantissa (with the most significant bit implied) are used; for 64-bit numbers, a sign bit, 11-bit binary exponent, and 53-bit binary mantissa (MSB implied) are used. Figure 2-2 summarizes the 990 and 9512 floating-point formats.

In order for TI Pascal programs to use the 9512, provisions must be made to (1) access the 9512 chip rather than the TIP floating-point interpreter, and (2) convert all operands to their 9512 representations. The first requirement is met somewhat clumsily and inefficiently by using PASLIB functions for all floating-point operations (Section 3.5). For example, the Pascal statement

```
IF (X*Y+Z) <= A THEN
```

becomes

```
IF CMP(ADD(MULT(X,Y),Z),A) <= 0 THEN
```

for execution on the Array. To satisfy the second requirement, conversion routines are provided for switching between 9512-format numbers and the scalar types INTEGER, LONGINT, REAL, and REAL(16) (Section 3.7). The programmer must convert all program constants and data which originated on the TI 990 to 9512 format before using them. For example,

```
VAR X:REAL(16);

      . . .

X:=2.0*17.5Q10;
```

becomes

```
VAR R2:REAL;
    X,D17_5Q10:REAL(16);
```

```
     . . .

R2:=CV9512(2.0);
D17_5Q10:=DV9512(17.5Q10);

     . . .

X:=MULT(DOUBLE(R2),D17_5Q10);
```

when using 9512 arithmetic.

NOTE

Since there appears to be no satisfactory technique in TI Pascal for declaring Am9512 numbers to be distinct from type REAL, it is the programmer's responsibility to ensure that floating-point numbers are in the correct format for the operation to be performed.

2.3 Linking for Execution on FEM

A sample link edit control file for Pascal programs to be run on FEM is shown in Figure 2-3. This file is available online as SYS1.FEM.PASLIB.UTIL$.LINKCTRL. The Nodal Exec loader will accept compressed format object code only. The use of compressed object code saves disk space on the Controller and significantly reduces the time required to download programs. Failure to use compressed format object code will cause the loader to generate an error message, and the load will be aborted. The program must be linked to PASLIB as well as to the standard TIP libraries. Note that the LIBRARY statements must be specified in the order shown so that the proper routines are included. INCLUDE statements are necessary for the Pascal runtime system (N$MAIN) and the program object code file from the TIP compiler. Stack and heap space are specified by including modules called STK$n and HP$n, where "n" is the number of 1024-byte blocks of memory to be allocated. If no heap is required, use HP$0.

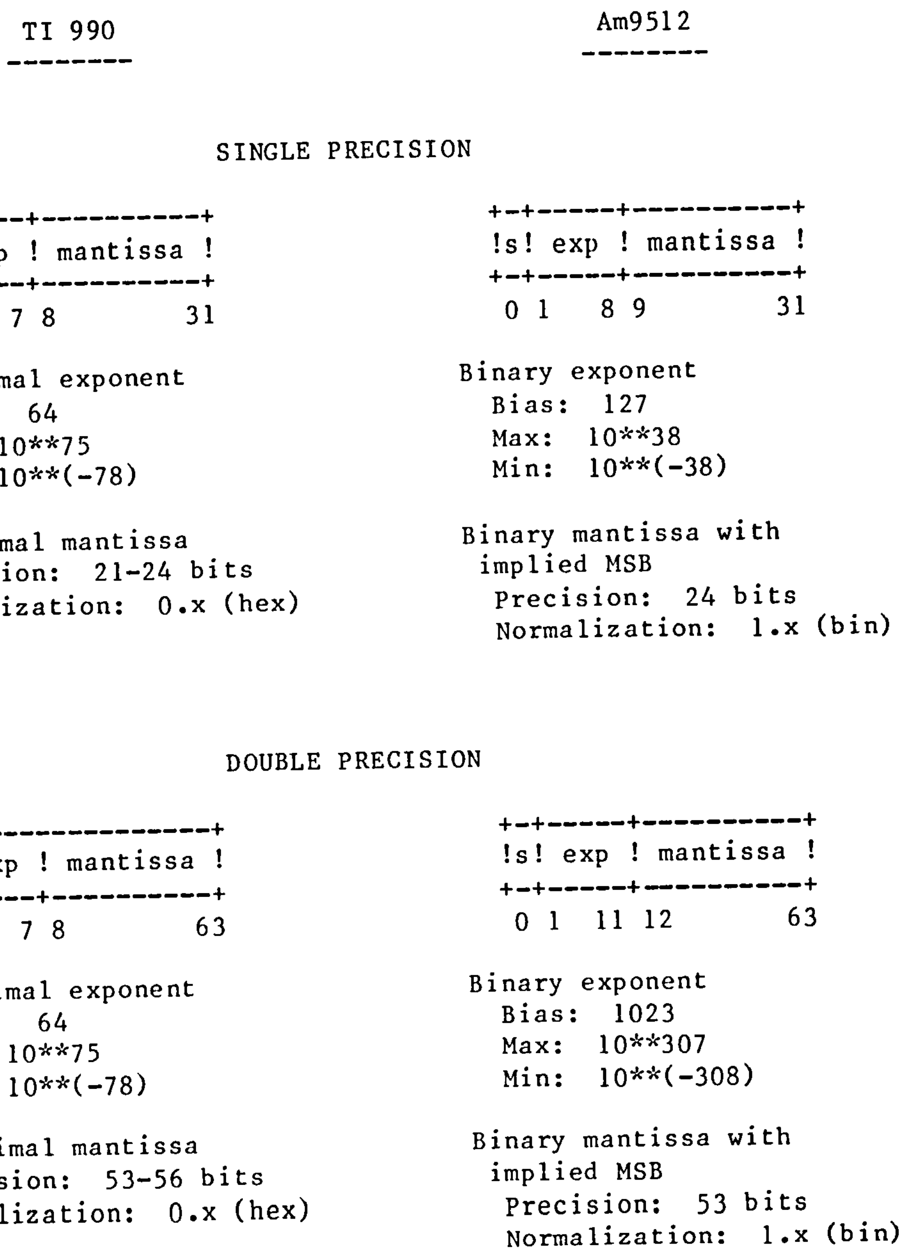

Figure 2-2. Comparison of TI 990 and Am9512 Floating-point Formats

```
NOSYMT
FORMAT COMPRESSED
LIBRARY SYS1.TIP.MINOBJ
LIBRARY SYS1.TIP.LUNOBJ
LIBRARY SYS1.TIP.OBJ
LIBRARY SYS1.FEM.PASLIB
PHASE 0,<program name>
INCLUDE (N$MAIN)
INCLUDE <program object code file from compiler>
INCLUDE (STK$n)
INCLUDE (HP$n)
END
```

Figure 2-3. Link Edit Control File for FEM Execution

3. SUBROUTINE DESCRIPTIONS

The following information is given for each of the PASLIB subroutines:

Purpose: Why the routine exists.

Declaration: The EXTERNAL declaration which must be used to reference the subroutine.

Description: What the routine does and how it does it.

Arguments: A description of each of the arguments.

Warnings/Limitations: Ways to get into trouble with this routine.

Application notes: Typical ways to use the routine.

Usage example: A program fragment demonstrating how the routine is invoked.

For functions, an additional item is given:

Function result: A description of the value returned by the function.

The declarations for PASLIB routines use several data types which are shown in Figure 3-1. These constant and type declarations are available online in SYS1.FEM.PASLIB.UTIL$.TYPDCL. The subroutine declarations are summarized for quick reference in Appendix C, and are also available online in SYS1.FEM.PASLIB.UTIL$.PASDCL.

```
                                    {  Type Declarations for PASLIB V2.1     }
                                    {    OCT 12, 1983   TWC                  }

CONST MAXIDX=255;                   {  maximum neighbor I/O index tag        }
      MAXREC=255;                   {  maximum record length for nbr I/O     }
      MAXNODE=36;                   {  maximum node number                   }
      MAXDA=31;                     {  maximum data area number              }
      MAXINT=32767;                 {  maximum integer                       }
      SYSFLAG=1;                    {  system flag                           }

TYPE NODE = 1..MAXNODE;             {  neighbors                             }
     IDX = 1..MAXIDX;               {  index tags                            }
     RECLEN = 1..MAXREC;            {  neighbor I/O record length            }
     DANUM = 0..MAXDA;              {  data area numbers                     }
     FLAG = 0..7;                   {  signal flags                          }
     ADDR = INTEGER;                {  integer used as an address            }
     POSINT = 1..MAXINT;            {  positive integer                      }
```

Figure 3-1. Constant and Type Declarations for PASLIB Routines.

3.1 Text Output

Programs on FEM may produce output to the Controller in two ways, (1) ASCII text which is transmitted to the Controller during program execution, and (2) binary data which is stored in a data area for subsequent uploading by the Controller. Text output is most useful for debug messages, prompts, and printed results, while data areas should be used for larger blocks of binary data (Section 3.3).

Lines of text are built up by making successive calls to the text output routines, analogous to the way lines are extended by the Pascal WRITE procedure. A line is terminated by a call to an end-of-line routine (ENDLN, NXTLN, MSGLN, or QUERY), which sends a signal to the Controller to close out the current line and begin a new one. The maximum line length is defined by the Controller software to be 80 characters, although automatic wrap-around is performed for lines which exceed that length. Refer to the FEM ARRAY CONTROL SOFTWARE USER'S GUIDE for more information about processing of text from the Array.

3.1.1 MSG

Purpose:

Write an ASCII character string to the Controller.

Declaration:

```
PROCEDURE MSG(STRING:PACKED ARRAY [1..?] OF CHAR);EXTERNAL;
```

Description:

The contents of STRING are moved to an output buffer which is placed on the global send queue with the Controller specified as the destination.

Arguments:

STRING - Either a character string enclosed in quotes or a variable defined to be PACKED ARRAY OF CHAR.

Warnings/Limitations:

STRING must be at least 2 characters in length and must not exceed 255 characters in length.

Application notes:

Use this routine to write character data to the Controller whenever an immediate end-of-line is not desired (see MSGLN). MSG should also be used to prompt for interactive input (Section 3.2).

Usage example:

```
MSG('THE VALUE OF X = '); MSGR(X); NXTLN;
```

3.1.2 MSGLN

Purpose:

Write an ASCII character string to the Controller and terminate the current line of text.

Declaration:

```
PROCEDURE MSGLN(STRING:PACKED ARRAY [1..?] OF CHAR);
                EXTERNAL;
```

Description:

The contents of STRING are queued for output followed by an EOL (end-of-line) indicator. The destination is the Controller. MSGLN is equivalent to MSG followed by ENDLN(1).

Arguments:

STRING - Either a character string enclosed in quotes or a variable defined to be PACKED ARRAY OF CHAR.

Warnings/Limitations:

STRING must be at least 2 characters in length and must not exceed 255 characters in length.

Application notes:

Use MSGLN to write character data to the Controller and close out the current line.

Usage example:

```
MSGLN('This is a complete line of text.');
```

3.1.3 ENDLN

Purpose:

Terminate the current line of text and advance one or more lines.

Declaration:

```
PROCEDURE ENDLN(N:POSINT);EXTERNAL;
```

Description:

N end-of-line (EOL) indicators are queued for output to the Controller. Each EOL terminates the current line of text and advances to the next line. ENDLN can be used to control line spacing; ENDLN(1) is single spacing, ENDLN(2) is double spacing, etc.

Arguments:

N - A positive integer specifying the line spacing following the current line.

Warnings/Limitations:

Large values of N are not recommended.

Application notes:

Use ENDLN to terminate a line of text and control spacing to the next line of text.

Usage example:

```
MSG('DOUBLE SPACING EXAMPLE'); ENDLN(2);
MSGLN('THIS LINE IS PRECEDED BY A BLANK LINE');
```

3.1.4 NXTLN

Purpose:

Terminate the current line of text and advance to the next line.

Declaration:

```
PROCEDURE NXTLN;EXTERNAL;
```

Description:

An EOL indicator is queued for output to the Controller. NXTLN is equivalent to ENDLN(1).

Arguments:

None.

Warnings/Limitations:

None.

Application notes:

Use NXTLN to terminate the current line of text when single spacing is desired.

Usage example:

```
MSG('I = '); MSGI(I); MSG(', J = '); MSGI(J); NXTLN;
```

3.1.5 MSGCH

Purpose:

Write a single ASCII character to the current line of output text.

Declaration:

```
PROCEDURE MSGCH(CH:CHAR);EXTERNAL;
```

Description:

CH is queued for output to the Controller.

Arguments:

CH - A constant, variable, or expression of type CHAR.

Warnings/Limitations:

Repeated calls to MSGCH are less efficient than a single call to MSG or MSGLN.

Application notes:

Use MSGCH whenever a single character must be written to the current line of output text. If several consecutive characters must be written, use one of the character string routines, MSG or MSGLN.

Usage example:

```
BUF_EFFICIENCY:=SUCCESS DIV ATTEMPT;
MSG('Buffer allocation efficiency = ');
MSGI(BUF_EFFICIENCY); MSGCH('%'); ENDLN(2);
```

3.1.6 MSGI

Purpose:

Write an integer to the current line of output text.

Declaration:

```
PROCEDURE MSGI(I:INTEGER);EXTERNAL;
```

Description:

I is converted from a 16-bit binary integer into its decimal ASCII representation, and the resulting string is queued for output to the Controller. The numeric characters are right-justified in a 6-character field with leading blanks.

Arguments:

I - An integer variable, constant, or expression.

Warnings/Limitations:

None.

Application notes:

Use MSGI to write integer values in lines of text.

Usage example:

```
VAR J:INTEGER;

    . . .

J:= ... ;
MSG('Result = '); MSGI(5*J); NXTLN;
```

3.1.7 MSGL

Purpose:

Write a long integer to the current line of output text.

Declaration:

```
PROCEDURE MSGL(I:LONGINT);EXTERNAL;
```

Description:

I is converted from a 32-bit binary integer into its decimal ASCII representation, and the resulting string is queued for output to the Controller. The numeric characters are right justified in an 11-character field with leading blanks.

Arguments:

I - A variable, constant, or expression of type LONGINT.

Warnings/Limitations:

None.

Application notes:

Use MSGL to write LONGINT values in lines of text.

Usage example:

```
CONST MINLINT = -2147483648L;

              . . .

MSG('Minimum long integer = '); MSGL(MINLINT); NXTLN;
```

3.1.8 MSGR

Purpose:

Write a single precision floating-point number in 9512-format to the current line of output text.

Declaration:

```
PROCEDURE MSGR(X:REAL);EXTERNAL;
```

Description:

X is converted from a 32-bit 9512-format value to a 13-character string of the form

```
nd.ddddddEsdd
```

where n is "-" or blank, d is a decimal digit, and s is "+" or "-". If X equals zero, the output string is "0.0" with one leading and nine trailing blanks. The string is queued for output to the Controller.

Arguments:

X - A single precision 9512-format variable or expression of type REAL.

Warnings/Limitations:

X must be in Am9512 format.

Application notes:

Use MSGR to write single precision floating-point values in lines of text.

Usage example:

```
VAR PI,TWO:REAL;

     . . .

PI:=CV9512(3.141593);
TWO:=CV9512(2.0);
MSG('PI/2 = '); MSGR(DIVD(PI,TWO)); NXTLN;
```

3.1.9 MSGD

Purpose:

Write a double precision floating-point number in 9512-format to the current line of output text.

Declaration:

```
PROCEDURE MSGD(X:REAL(16));EXTERNAL;
```

Description:

X is converted from a 64-bit 9512-format value to a 22-character string of the form

nd.ddddddddddddddDsddd

where n is "-" or blank, d is a decimal digit, and s is "+" or "-". If X equals zero, the string is "0.0" with one leading and 18 trailing blanks. The string is queued for output to the Controller.

Arguments:

X - A double precision 9512-format variable or expression of type REAL(16).

Warnings/Limitations:

X must be in Am9512 format.

Application notes:

Use MSGD to write double precision floating-point values in lines of text.

Usage example:

```
VAR BB,A:REAL(16);
     . . .
MSG('BB = '); MSGD(BB); MSG('     BC = ');
MSGD(DDIVD(BB,A)); NXTLN;
```

3.1.10 MSGIH

Purpose:

Write the hexadecimal representation of an integer to the current line of output text.

Declaration:

```
PROCEDURE MSGIH(I:INTEGER);EXTERNAL;
```

Description:

I is converted to a string of four hexadecimal digits, and the string is queued for output to the Controller.

Arguments:

I - A variable, expression, or constant of type INTEGER.

Warnings/Limitations:

None.

Application notes:

This routine is primarily useful in diagnostic programming or other applications where the bit pattern is of interest.

Usage example:

```
MSG('I = DECIMAL '); MSGI(I);
MSG(', HEX '); MSGIH(I); NXTLN;
```

3.1.11 MSGRH

Purpose:

Write the hexadecimal representation of a single precision floating-point number to the current line of output text.

Declaration:

```
PROCEDURE MSGRH(X:REAL);EXTERNAL;
```

Description:

X is converted to an unformatted string of eight hexadecimal digits, and the string is queued for output to the Controller. X may be in either 990 or 9512 format.

Arguments:

X - A constant, variable, or expression of type REAL.

Warnings/Limitations:

None.

Application notes:

MSGRH is primarily useful in diagnostic and systems programming.

Usage example:

```
MSG('EXPECTED = '); MSGRH(X1);
MSG('RECEIVED = '); MSGRH(X2); NXTLN;
```

3.1.12 MSGDH

Purpose:

Write the hexadecimal representation of a double precision floating-point number to the current line of output text.

Declaration:

```
PROCEDURE MSGDH(X:REAL(16));EXTERNAL;
```

Description:

X is converted to an unformatted string of 16 hexadecimal digits, and the string is queued for output to the Controller. X may be in either 990 or 9512 format.

Arguments:

X - A constant, variable, or expression of type REAL(16).

Warnings/Limitations:

None.

Application notes:

MSGDH is primarily useful in diagnostic and systems programming.

Usage example:

```
VAR Q1_0:REAL(16);

      . . .

Q1_0:=DV9512(1.0Q0);
MSGLN('Internal Representation of 1.0Q0');
MSG('  990 FORMAT:  '); MSGDH(1.0Q0);
MSG('     9512 FORMAT:  '); MSGDH(Q1_0); ENDLN(2);
```

3.1.13 CWAIT

Purpose:

Following a text output operation to the Controller, delay the program until the Controller has had a chance to receive and process the text.

Declaration:

```
PROCEDURE CWAIT;EXTERNAL;
```

Description:

CWAIT waits until all of the buffers on the global output list have been put into the hardware output FIFO. It then executes a delay which is long enough for the Controller to receive and process all of the data. The delay time is based on a conservative estimate of the time required for the Controller to receive and process 64 words of text when a single processor is transmitting to the Controller. At the end of the delay, control is returned to the calling program.

Arguments:

None.

Warnings/Limitations:

The delay time assumes that only a single processor is transmitting to the Controller. If more than one processor is transmitting, the delay could be too short, thereby allowing the calling program to proceed prematurely.

Application notes:

Use CWAIT for cooperative messages generated by multiple processors. This technique can be used to order the arrival of messages on the Controller so that post-sorting of the text is not necessary.

Usage example:

```
FOR I:=FIRST_NODE TO LAST_NODE DO
  BEGIN
  BAR(REPORT_FLAG);
  IF I = LSELF THEN
    BEGIN
    MSG('Node '); MSGI(I); NXTLN;
    MSG('Displacement = '); MSGR(DSPL); ENDLN(2);
    CWAIT
    END
  END;
```

3.2 Interactive Input

Pascal programs running on FEM may be written to interact with a user, subject to certain restrictions. The program must first signal the Controller that it wants input from the terminal; this is called a "query" operation, and is supported by the QUERY subroutine. A query operation must take place simultaneously on all active processors in the Array, and all processors must expect the same input. This restriction is necessary to prevent a user from having to respond to many (potentially 36) requests, each expecting possibly different input. To ensure that a query is performed cooperatively, the QUERY routine internally performs a barrier operation using the system flag (flag 1). This implies that the program must be written so that all active processors will perform each query operation.

Upon reception of a query, the Controller terminates the current line of output text for each processor, and prompts the user for input at the terminal by displaying a "?" at the beginning of the next line. The user must enter the appropriate response, followed by a carriage return. The line of input text (excluding the leading "?") is then broadcast to all active processors for evaluation. The numeric input routines RDH, RDI, and RDR include error recovery procedures which will re-prompt the user for input in the event of an illegal or missing value. Only one call to RDH, RDI, or RDR should be issued for each call to QUERY.

Interactive input is primarily useful for entering parameter values which are the same on all processors and vary from run to run. Data areas (Section 3.3) should be used instead for input which (1) differs from processor to processor, (2) is stored on Controller files, (3) is constant from run to run, or (4) consists of more than a few values.

3.2.1 QUERY

Purpose:

Signals the Controller that the program running on the Array desires input from the terminal.

Declaration:

```
PROCEDURE QUERY;EXTERNAL;
```

Description:

All active processors are synchronized by a barrier operation on SYSFLAG (flag 1). A query request is then queued for output to the Controller. A query implies an end-of-line (EOL).

Arguments:

None.

Warnings/Limitations:

All active processors must participate in every query operation.

Application notes:

Use QUERY to request interactive input in those situations in which all processors wish to receive the same value.

Usage example:

```
MSG('Enter the initial guess:'); QUERY;
INITIAL:=RDR;
```

3.2.2 RDCH

Purpose:

Read a character from the terminal.

Declaration:

```
FUNCTION RDCH:CHAR;EXTERNAL;
```

Description:

Return the next character from the line of input text which was broadcast from the Controller. Multiple calls to RDCH may be made for each call to QUERY. A carriage return (CR, >0D) is always the last character in the line, and is returned just like any other character. If no input is available, RDCH will wait indefinitely.

Arguments:

None.

Function result:

The next character in the line of input text from the Controller.

Warnings/Limitations:

A call to QUERY must be made to obtain a line of input text.

The CR character must be used to check for end-of-line. An attempt to read past CR will cause the program to wait forever for input.

Application notes:

Use RDCH to parse lines of input text from the Controller.

Usage example:

```
MSG('Press RETURN to continue...'); QUERY;
CH:=RDCH;
```

3.2.3 RDH

Purpose:

Read a hexadecimal value from the terminal.

Declaration:

```
FUNCTION RDH:INTEGER;EXTERNAL;
```

Description:

Parses a line of input text containing hexadecimal digits and returns an integer value. The input string may contain from 1 to 4 hex digits ("0"..."9","A"..."F"). If more than 4 digits are entered, only the 4 least significant digits are used. If the input string is empty or has illegal hex input, a message is issued and the user is re-prompted for the input.

Arguments:

None.

Function result:

A 16-bit two's complement integer value.

Warnings/Limitations:

A call to QUERY must precede a call to RDH.

Only one call to RDH should be made for each QUERY.

Application notes:

RDH is primarily useful in diagnostic programs and other system software applications.

Usage example:

```
MSG('ENTER TEST PATTERN:  '); QUERY;
DATA:=RDH;
```

3.2.4 RDI

Purpose:

Read an integer from the terminal.

Declaration:

```
FUNCTION RDI:INTEGER;EXTERNAL;
```

Description:

Parses a line of input text and returns an integer value. If the line of text is empty or has illegal integer input, a message is issued and the user is re-prompted for the input.

Arguments:

None.

Function result:

A 16-bit integer value.

Warnings/Limitations:

A call to QUERY must precede a call to RDI.

Only one call to RDI should be made for each QUERY.

Application notes:

Use RDI to read an integer from the user's terminal.

Usage example:

```
MSG('ENTER N:'); QUERY;
N:=RDI;
```

3.2.5 RDR

Purpose:

Read a single precision floating-point number from the terminal.

Declaration:

```
FUNCTION RDR:REAL;EXTERNAL;
```

Description:

Parses a line of input text and returns a single precision floating-point value in 9512 format. If the line of text is empty or has illegal floating-point input, a message is issued and the user is re-prompted for the input. Allowable input formats include

```
sdddddddd
sdddd.dddd
sdddd.ddddEsdd
sddddddddEsdd
```

where s is either "+", "-", or omitted, and d is a decimal digit. One or more digits may be entered, but only the first eight are significant. A digit must precede the decimal point.

Arguments:

None.

Function result:

A 32-bit floating-point number in Am9512 single precision format.

Warnings/Limitations:

A call to QUERY must precede a call to RDR.

Only one call to RDR should be made for each QUERY.

The program must be expecting a 9512-format number rather than a 990-format number.

Application notes:

Use RDR to read a REAL number from the user's terminal.

Usage example:

```
MSG('Enter convergence criterion'); QUERY;
CVRG:=RDR;
```

3.3 Data Areas

Data areas are referenced from Pascal programs via pointer variables. A type transfer is used in conjunction with the DAPTR routine to obtain the address of a specific data area. The standard Pascal routines NEW and DISPOSE should not be used with data area pointers. Data areas 0 - 3 are reserved for use by the system as follows:

0 - list of physical neighbors
1 - logical-to-physical mapping
2 - list of logical neighbors
3 - reserved for future use.

Data areas 4 - 31 are available to user programs.

Data areas are defined, downloaded, and uploaded from the Controller. See the FACS User's Guide for more information.

3.3.1 DAPTR

Purpose:

Obtain the address of a data area.

Declaration:

```
FUNCTION DAPTR(DA:DANUM):ADDR;EXTERNAL;
```

Description:

The address of the specified data area is obtained from the data area table, and the length of the descriptor is added to obtain a pointer to the data.

Arguments:

DA - An integer data area number in the range 0..MAXDA.

Function result:

The address of the data portion of the specified data area.

Warnings/Limitations:

Data area DA must be defined.

A type transfer is needed to assign the function result to a pointer variable.

Data areas 0 - 3 are reserved for use by the system.

Application notes:

Use DAPTR to map Pascal program data structures onto data areas.

Usage example:

```
TYPE DA4 = ARRAY [1..12,1..12] OF REAL;
     DA4PTR = @DA4;

VAR KDATA:DA4PTR;

          . . .

KDATA::ADDR:=DAPTR(4);
FOR I:=1 TO 12 DO
  FOR J:=1 TO 12 DO
    KDATA@[I,J]:=CV9512(KDATA@[I,J]);
```

3.4 Flags

The flag network can be used for a variety of signaling and synchronization needs. There are eight flags, numbered 0 through 7. Flag 0 is unique since it is the only flag which supports the FIRST signal. Flag 1 is reserved for use by the system (SYSFLAG), although there are circumstances where it can be used for synchronization (barriers) in user programs.

At the beginning of program execution, all flags except SYSFLAG have been disabled by Nodal Exec. SYSFLAG is enabled and reset. On inactive (OFF) processors and on the Controller, all flags including SYSFLAG are disabled. Before any of the flags may be used, they must be enabled and either set or reset so that a known state exists in the flag network. As shown in the following example, SYSFLAG may be used to synchronize this operation across all of the active processors. The example assumes that only flags 2 and 3 will be used by the program.

```
FLGEN(2);  FLGRES(2);  (* Enable and reset flag 2      *)
FLGEN(3);  FLGSET(3);  (* Enable and set flag 3        *)
BAR(SYSFLAG);  (* Wait for all processors to catch up *)
```

The system flag is also used by Nodal Exec in the QUERY routine, the program initiation and termination routines (EXEC and STOP), and the CONNECT command.

NOTE

When a program terminates, all flags on that processor, except SYSFLAG, are disabled. This effectively removes terminated processors from the flag network. In cases where processors terminate at different times, and the final state of the flags must be maintained until all processors have terminated, a flag barrier should be placed immediately preceding the END statement for procedure PSCL$$. This will ensure that all processors terminate together, and do not disable their flags prematurely.

3.4.1 FLGEN

Purpose:

Enable a given flag on this processor.

Declaration:

```
PROCEDURE FLGEN(F:FLAG);EXTERNAL;
```

Description:

Flag F is enabled on the processor on which the routine is executed.

Arguments:

F - A constant, variable, or expression of type FLAG.

Warnings/Limitations:

FLGEN should not be used on SYSFLAG.

The status of a flag which has been disabled should be considered to be undefined following a FLGEN until a FLGRES or FLGSET is performed.

Application notes:

Use FLGEN to enable a flag for subsequent use, or to re-enable a flag which has been disabled for some reason. With the exception of SYSFLAG, all flags must be enabled before they can be set or reset.

Usage example:

```
FOR F IN [0,2..7] DO
  FLGEN(F);
```

3.4.2 FLGDIS

Purpose:

Disable a given flag on this processor.

Declaration:

```
PROCEDURE FLGDIS(F:FLAG);EXTERNAL;
```

Description:

Flag F is disabled on the processor on which the routine is executed.

Arguments:

F - A constant, variable, or expression of type FLAG.

Warnings/Limitations:

FLGDIS should not be used on SYSFLAG.

Application notes:

Use FLGDIS (in conjunction with FLGEN) to dynamically control a processor's contributions to the flag network. For example, a program might need to be contributing to a flag signal during one portion of its code, but not during another portion.

Usage example:

```
IF LSELF < (NNODES/2) THEN
  FLGDIS(4+(LSELF MOD 2));
```

3.4.3 FLGRES

Purpose:

Reset a given flag on this processor.

Declaration:

```
PROCEDURE FLGRES(F:FLAG);EXTERNAL;
```

Description:

Flag F is reset (cleared) on the processor on which the routine is executed.

Arguments:

F - A constant, variable, or expression of type FLAG.

Warnings/Limitations:

Flag F must be enabled.

FLGRES should not be used on SYSFLAG.

Application notes:

Use FLGRES to signal the absence of some condition on a processor.

Usage example:

```
IF NOT CONVERGED THEN
  FLGRES(CVG_FLAG);
```

3.4.4 FLGSET

Purpose:

Set a given flag on this processor.

Declaration:

```
PROCEDURE FLGSET(F:FLAG);EXTERNAL;
```

Description:

Flag F is set on the processor on which the routine is executed.

Arguments:

F - A constant, variable, or expression of type FLAG.

Warnings/Limitations:

Flag F must be enabled.

FLGSET should not be used on SYSFLAG.

Application notes:

Use FLGSET to indicate the presence of some condition on a processor.

Usage example:

```
FLGSET(2);

  . . .

IF ALL(2) THEN

  . . .
```

3.4.5 ANY

Purpose:

Test the Any signal for a given flag.

Declaration:

```
FUNCTION ANY(F:FLAG):BOOLEAN;EXTERNAL;
```

Description:

The ANY routine tests the status of the global flag signal Any for flag F. If one or more processors with flag F enabled also have flag F set, then ANY returns a value of TRUE; otherwise ANY returns FALSE.

Arguments:

F - A constant, variable, or expression of type FLAG.

Function result:

A Boolean value indicating the status of Any for flag F.

Warnings/Limitations:

None.

Application notes:

Use ANY to determine if some condition exists on one or more of the participating processors.

Usage example:

```
IF ANY(ERR_FLAG) THEN
  MSGLN('*** Errors detected in assembly phase');
```

3.4.6 ALL

Purpose:

Test the All signal for a given flag.

Declaration:

```
FUNCTION ALL(F:FLAG):BOOLEAN;EXTERNAL;
```

Description:

The ALL routine tests the status of the global flag signal All for flag F. If every processor with flag F enabled also has flag F set, then ALL returns a value of TRUE; otherwise ALL returns FALSE.

Arguments:

F - A constant, variable, or expression of type FLAG.

Function result:

A Boolean value indicating the status of All for flag F.

Warnings/Limitations:

None.

Application notes:

Use ALL to determine if some condition exists on every participating processor.

Usage example:

```
WHILE NOT ALL(7) DO
  BEGIN

  . . .

  END;
```

3.4.7 SYNC

Purpose:

Test the Sync signal for a given flag.

Declaration:

```
FUNCTION SYNC(F:FLAG):BOOLEAN;EXTERNAL;
```

Description:

The SYNC routine tests the status of the global flag signal Sync for flag F. The Sync signal becomes true when All is true, and remains true until Any becomes false.

Arguments:

F - A constant, variable, or expression of type FLAG.

Function result:

A Boolean value indicating the status of Sync for flag F.

Warnings/Limitations:

None.

Application notes:

Use SYNC to indicate that a condition has occurred on all participating processors, and is still occurring on one or more of those processors.

Usage example:

```
FLGRES(FLAG4);
BAR(FLAG2);
REPEAT

    . . .

  IF CMP(X0,X1) < 0 THEN
    FLGSET(FLAG4);

    . . .

  UNTIL SYNC(FLAG4);
WHILE SYNC(FLAG4) DO
  BEGIN

    . . .

  IF CMP(X1,X2) > 0 THEN
    FLGRES(FLAG4);
```

```
    . . .
END;
```

3.4.8 FIRST

Purpose:

Test the First signal for flag 0.

Declaration:

```
FUNCTION FIRST:BOOLEAN;EXTERNAL;
```

Description:

The FIRST routine tests the status of the global flag signal First. First is set by the (approximately) first processor to set flag 0 after Any(0) has been false.

Arguments:

None.

Function result:

A Boolean value indicating whether or not this is the first processor to set flag 0.

Warnings/Limitations:

Due to signal propagation delays, etc., a true value for FIRST does not guarantee that the processor was absolutely the first one to set its flag. FIRST does indicate a unique processor which did set its flag 0 within a very short time period (possibly 0) following the absolutely first one.

Flag 0 must be enabled.

A call to FLGSET for flag 0 must precede a call to FIRST.

Application notes:

FIRST can be used to select a single processor (out of many) to carry out some task.

Usage example:

```
FLGSET(0);
IF FIRST THEN  (* 1st processor prints the title *)
  PRINT_HEADING;
```

3.4.9 BAR

Purpose:

Synchronize the participating processors using a flag barrier.

Declaration:

```
PROCEDURE BAR(F:FLAG);EXTERNAL;
```

Description:

All processors which have flag F enabled are synchronized in time by simultaneously reaching the same point in the BAR routine. The algorithm used for a flag barrier is as follows:

```
WHILE SYNC(F) DO
  ;  (* wait for Sync to go low  *)
FLGSET(F);
WHILE NOT SYNC(F) DO
  ;  (* wait for Sync to go high *)
(* Processors are synchronized at this point *)
FLGRES(F);
```

F must be in a reset state prior to entering the barrier, and is left in a reset state upon exit from the barrier.

Arguments:

F - A constant, variable, or expression of type FLAG.

Warnings/Limitations:

Flag F must be enabled and reset before calling BAR.

SYSFLAG should be used only in carefully thought out situations.

Application notes:

Use BAR to synchronize processors. This is especially useful for guaranteeing that a certain state has been reached on all participating processors before proceeding.

Usage example:

```
(* enable and reset flags *)
FLGEN(2); FLGRES(2);
FLGEN(3); FLGRES(3);
BAR(SYSFLAG);

    . . .

WHILE NOT ALL(2) DO
  BEGIN
```

```
  . . .
IF CMP(X,DELTA) < 0 THEN
  FLGSET(2)
ELSE
  FLGRES(2);
BAR(3)   (* synchronize before testing ALL(2) *)
END;
```

3.5 Am9512 Floating-point Operations

Functions are provided to add, subtract, multiply, divide, negate, compare, and take the absolute value of single and double precision floating-point numbers in Am9512 format. Refer to Section 2.2 for information about writing programs which use 9512 arithmetic.

Floating-point exceptions include exponent underflow and overflow, and division by zero. For all exceptions, an appropriate error message is generated. For exponent overflow and division by zero the program is aborted. For exponent underflow, the result of the operation is set to zero and execution continues.

3.5.1 ADD

Purpose:

Add two 9512-format single precision floating-point numbers.

Declaration:

```
FUNCTION ADD(X,Y:REAL):REAL;EXTERNAL;
```

Description:

X and Y are pushed onto the 9512 stack, a SADD operation is performed, and the result is popped from the stack. The status byte is checked for exceptions.

Arguments:

X,Y - Variables or expressions of type REAL in 9512 single precision format.

Function result:

X+Y in 9512 format.

Warnings/Limitations:

Possible exceptions: underflow, overflow.

X and Y must be 9512-format numbers.

Application notes:

Use to add two single precision numbers.

Usage example:

```
C:=ADD(A,B);
```

3.5.2 SUB

Purpose:

Subtraction of 9512-format single precision floating-point numbers.

Declaration:

```
FUNCTION SUB(X,Y:REAL):REAL;EXTERNAL;
```

Description:

X and Y are pushed onto the 9512 stack, a SSUB operation is performed, and the result is popped from the stack. The status byte is checked for exceptions.

Arguments:

X,Y - Variables or expressions of type REAL in 9512 single precision format.

Function result:

X-Y in 9512 format.

Warnings/Limitations:

Possible exceptions: underflow, overflow.

X and Y must be 9512-format numbers.

Application notes:

Use to subtract one single precision number from another.

Usage example:

```
DELTA:=SUB(X[I+1],X[I]);
```

3.5.3 MULT

Purpose:

Multiply two 9512-format single precision floating-point numbers.

Declaration:

```
FUNCTION MULT(X,Y:REAL):REAL;EXTERNAL;
```

Description:

X and Y are pushed onto the 9512 stack, a SMUL operation is performed, and the result is popped from the stack. The status byte is checked for exceptions.

Arguments:

X,Y - Variables or expressions of type REAL in 9512 single precision format.

Function result:

X*Y in 9512 format.

Warnings/Limitations:

Possible exceptions: underflow, overflow.

X and Y must be 9512-format numbers.

Application notes:

Use to multiply two single precision numbers.

Usage example:

```
X_SQUARED:=MULT(X,X);
```

3.5.4 DIVD

Purpose:

Division of 9512-format single precision floating-point numbers.

Declaration:

```
FUNCTION DIVD(X,Y:REAL):REAL;EXTERNAL;
```

Description:

X and Y are pushed onto the 9512 stack, a SDIV operation is performed, and the result is popped from the stack. The status byte is checked for exceptions.

Arguments:

X,Y - Variables or expressions of type REAL in 9512 single precision format.

Function result:

X/Y in 9512 format.

Warnings/Limitations:

Possible exceptions: underflow, overflow, division by zero.

X and Y must be 9512-format numbers.

Application notes:

Use to divide one single precision number by another.

Usage example:

```
PI_2:=DIVD(CV9512(3.141593),CV9512(2.0));
```

3.5.5 NEG

Purpose:

Negates a 9512-format single precision floating-point number.

Declaration:

```
FUNCTION NEG(X:REAL):REAL;EXTERNAL;
```

Description:

Inverts the sign bit of X.

Arguments:

X - A variable or expression of type REAL in 9512 single precision format.

Function result:

-X in 9512 format.

Warnings/Limitations:

None.

Application notes:

Use to obtain the additive inverse of single precision numbers.

Usage example:

```
X[I,J]:=MULT(Y[I],NEG(DY[J]));
```

3.5.6 ABS95

Purpose:

Take the absolute value of a 9512-format single precision floating-point number.

Declaration:

```
FUNCTION ABS95(X:REAL):REAL;EXTERNAL;
```

Description:

Sets the sign bit of X to zero.

Arguments:

X - Variable or expression of type REAL in 9512 single precision format.

Function result:

Absolute value of X.

Warnings/Limitations:

None.

Application notes:

Use to obtain the absolute value of a single precision number.

Usage example:

```
IF CMP(ABS95(X[I]),CV9512(0.0001)) < 0 THEN
    . . .
```

3.5.7 CMP

Purpose:

Compare two 9512-format single precision floating-point numbers.

Declaration:

```
FUNCTION CMP(X,Y:REAL):INTEGER;EXTERNAL;
```

Description:

Performs a software comparison of two 9512-format numbers and returns the result.

Arguments:

X,Y - Variables or expressions of type REAL in 9512 single precision format.

Function result:

```
-1 : X < Y
 0 : X = Y
 1 : X > Y
```

Warnings/Limitations:

X and Y must be 9512-format numbers.

Application notes:

Use to compare floating-point values. The following table shows comparison expressions for standard Pascal and their equivalent when using 9512 arithmetic.

TIP	9512
X < Y	CMP(X,Y) < 0
X <= Y	CMP(X,Y) <= 0
X = Y	CMP(X,Y) = 0
X >= Y	CMP(X,Y) >= 0
X > Y	CMP(X,Y) > 0

Usage example:

```
WHILE CMP(SUB(X[I],X[I-1]),DX) > 0 DO
              . . .
```

3.5.8 DADD

Purpose:

Add two 9512-format double precision floating-point numbers.

Declaration:

```
FUNCTION DADD(X,Y:REAL(16)):REAL(16);EXTERNAL;
```

Description:

X and Y are pushed onto the 9512 stack, a DADD operation is performed, and the result is popped from the stack. The status byte is checked for exceptions.

Arguments:

X,Y - Variables or expressions of type REAL(16) in 9512 double precision format.

Function result:

X+Y in 9512 format.

Warnings/Limitations:

Possible exceptions: underflow, overflow.

X and Y must be 9512-format numbers.

Application notes:

Use to add two double precision numbers.

Usage example:

```
V:=DADD(U,DOUBLE(X));
```

3.5.9 DSUB

Purpose:

Subtraction of 9512-format double precision floating-point numbers.

Declaration:

```
FUNCTION DSUB(X,Y:REAL(16)):REAL(16);EXTERNAL;
```

Description:

X and Y are pushed onto the 9512 stack, a DSUB operation is performed, and the result is popped from the stack. The status byte is checked for exceptions.

Arguments:

X,Y - Variables or expressions of type REAL(16) in 9512 double precision format.

Function result:

X-Y in 9512 format.

Warnings/Limitations:

Possible exceptions: underflow, overflow.

X and Y must be 9512-format numbers.

Application notes:

Use to subtract one double precision number from another.

Usage example:

```
DZ:=DSUB(Z2,Z1);
```

3.5.10 DMULT

Purpose:

Multiply two 9512-format double precision floating-point numbers.

Declaration:

```
FUNCTION DMULT(X,Y:REAL(16)):REAL(16);EXTERNAL;
```

Description:

X and Y are pushed onto the 9512 stack, a DMUL operation is performed, and the result is popped from the stack. The status byte is checked for exceptions.

Arguments:

X,Y - Variables or expressions of type REAL(16) in 9512 double precision format.

Function result:

X*Y in 9512 format.

Warnings/Limitations:

Possible exceptions: underflow, overflow.

X and Y must be 9512-format numbers.

Application notes:

Use to multiply two double precision numbers.

Usage example:

```
X_CUBED:=DMULT(X,DMULT(X,X));
```

3.5.11 DDIVD

Purpose:

Division of 9512-format double precision floating-point numbers.

Declaration:

```
FUNCTION DDIVD(X,Y:REAL(16)):REAL(16);EXTERNAL;
```

Description:

X and Y are pushed onto the 9512 stack, a DDIV operation is performed, and the result is popped from the stack. The status byte is checked for exceptions.

Arguments:

X,Y - Variables or expressions of type REAL(16) in 9512 double precision format.

Function result:

X/Y in 9512 format.

Warnings/Limitations:

Possible exceptions: underflow, overflow, division by zero.

X and Y must be 9512-format numbers.

Application notes:

Use to divide one double precision number by another.

Usage example:

```
PI_4:=DDIVD(DV9512(3.141592653589793Q0),DV9512(4.0Q0));
```

3.5.12 DNEG

Purpose:

Negates a 9512-format double precision floating-point number.

Declaration:

```
FUNCTION DNEG(X:REAL(16)):REAL(16);EXTERNAL;
```

Description:

Inverts the sign bit of X.

Arguments:

X - A variable or expression of type REAL(16) in 9512 double precision format.

Function result:

-X in 9512 format.

Warnings/Limitations:

None.

Application notes:

Use to obtain the additive inverse of double precision numbers.

Usage example:

```
C:=DNEG(DMULT(A,B));
```

3.5.13 DABS95

Purpose:

Take the absolute value of a 9512-format double precision floating-point number.

Declaration:

```
FUNCTION DABS95(X:REAL(16)):REAL(16);EXTERNAL;
```

Description:

Sets the sign bit of X to zero.

Arguments:

X - Variable or expression of type REAL(16) in 9512 double precision format.

Function result:

Absolute value of X.

Warnings/Limitations:

None.

Application notes:

Use to obtain the absolute value of a double precision number.

Usage example:

```
ANS:=DABS95(DSUB(Y[I+J],Y[I-J]));
```

3.5.14 DCMP

Purpose:

Compare two 9512-format double precision floating-point numbers.

Declaration:

```
FUNCTION DCMP(X,Y:REAL(16)):REAL(16);EXTERNAL;
```

Description:

Performs a software comparison of two 9512-format numbers and returns the result.

Arguments:

X,Y - Variables or expressions of type REAL(16) in 9512 double precision format.

Function result:

```
-1 :  X < Y
 0 :  X = Y
 1 :  X > Y
```

Warnings/Limitations:

X and Y must be 9512-format numbers.

Application notes:

Use to compare floating-point values. (See the comparison table for the CMP routine, Section 3.5.7.)

Usage example:

```
IF DCMP(DABS95(DSUB(DELTA[I-1],DELTA[I])),EPSILON) > 0 THEN
                    . . .
```

3.6 Am9512 Floating-point Constants

Functions are provided which return the maximum and minimum positive Am9512 floating-point values. These values are typically useful for initializing variables or guarding against underflow or overflow conditions.

3.6.1 MAX95

Purpose:

Provides an exact representation of the largest positive single precision Am9512 floating-point number.

Declaration:

```
FUNCTION MAX95:REAL;EXTERNAL;
```

Description:

Returns the maximum Am9512 single precision value.

Arguments:

None.

Function result:

A REAL value containing the largest representable Am9512 single precision floating-point number, approximately 3.402823E+38.

Warnings/Limitations:

None.

Application notes:

Use MAX95 when a very large number is needed.

Usage example:

```
(* INITIALIZE DELTAS *)
MAXREAL:=MAX95;
FOR I:=1 TO 12 DO
  DELTA[I]:=MAXREAL;
```

3.6.2 MIN95

Purpose:

Provides an exact representation of the smallest positive single precision Am9512 floating-point number.

Declaration:

```
FUNCTION MIN95:REAL;EXTERNAL;
```

Description:

Returns the minimum positive Am9512 single precision value.

Arguments:

None.

Function result:

A REAL value containing the smallest representable positive Am9512 single precision floating-point number, approximately 1.175494E-38.

Warnings/Limitations:

None.

Application notes:

Use MIN95 when a very small, but non-zero, number is needed.

Usage example:

```
IF CMP(X,MULT(TEN,MIN95)) <= 0 THEN  (* UNDERFLOW IS IMMINENT *)
  X:=UNDERFLOW(X)
ELSE
  X:=DIVD(X,TEN);
```

3.6.3 DMAX95

Purpose:

Provides an exact representation of the largest positive double precision Am9512 floating-point number.

Declaration:

```
FUNCTION DMAX95:REAL(16);EXTERNAL;
```

Description:

Returns the maximum Am9512 double precision value.

Arguments:

None.

Function result:

A REAL(16) value containing the largest representable Am9512 double precision floating-point number, approximately 1.79769313486231D+308.

Warnings/Limitations:

None.

Application notes:

Use DMAX95 when a very large number is needed.

Usage example:

```
MSG('Input values must be less than '); MSGD(DMAX95);
ENDLN(2);
```

3.6.4 DMIN95

Purpose:

Provides an exact representation of the smallest positive double precision Am9512 floating-point number.

Declaration:

```
FUNCTION DMIN95:REAL(16);EXTERNAL;
```

Description:

Returns the minimum positive Am9512 double precision value.

Arguments:

None.

Function result:

A REAL(16) value containing the smallest representable positive Am9512 double precision floating-point number, approximately 2.2250738585072OD-308.

Warnings/Limitations:

None.

Application notes:

Use DMIN95 when a very small, but non-zero, number is needed.

Usage example:

```
MINDBLE:=DMIN95;
FOR I:=1 TO 32 DO
  FOR J:=1 TO 32 DO
    A[I,J]:=MINDBLE;
```

3.7 Am9512 Floating-point Conversions

A complete set of functions are provided for conversions between single and double precision 9512-format numbers and the TI Pascal data types REAL, REAL(16), INTEGER, and LONGINT. These routines must be used when converting 9512-format numbers; the TIP standard conversion routines and implicit conversions will produce unexpected results if Am9512 numbers are involved.

3.7.1 CV9512

Purpose:

Convert a 990-format single precision floating-point value to 9512 format.

Declaration:

```
FUNCTION CV9512(X:REAL):REAL;EXTERNAL;
```

Description:

Re-align the mantissa and convert the hexadecimal exponent to binary, adjusting for shifting of the mantissa.

Arguments:

X - Constant, variable, or expression of type REAL in TI 990 format.

Function result:

Am9512 representation of X.

Warnings/Limitations:

Exponent underflow or overflow may occur. Allowable exponents are in the range -38 to +38.

The 9512-format result will contain space for zero to three additional bits in the mantissa; these are filled with trailing zeroes.

Application notes:

Use CV9512 to convert program constants, and data which originated on the Controller.

Usage example:

```
FOR I:=1 TO DA4@.NNODES DO
  DA5@[I]:=CV9512(DA5@[I]);
```

3.7.2 CV990

Purpose:

Convert a 9512-format single precision floating-point value to 990 format.

Declaration:

```
FUNCTION CV990(X:REAL):REAL;EXTERNAL;
```

Description:

Convert the exponent from binary to hexadecimal and normalize the mantissa based on hexadecimal digits.

Arguments:

X - Variable or expression of type REAL in Am9512 format.

Function result:

990-format representation of X.

Warnings/Limitations:

From zero to three bits of precision will be lost from the mantissa by normalizing to hex digits.

Application notes:

Use CV990 to convert results to TI 990 format before uploading them to the Controller.

Usage example:

```
FOR I:=1 TO NROWS DO
  FOR J:=1 TO NCOLS DO
    STRESS@[I,J]:=CV990(STRESS@[I,J]);
```

3.7.3 FLOATI

Purpose:

Convert a 16-bit two's complement integer to Am9512 single precision floating-point.

Declaration:

```
FUNCTION FLOATI(I:INTEGER):REAL;EXTERNAL;
```

Description:

The sign of I is determined, and the absolute value is found. The integer is normalized (right-shifted), and the shift count becomes the binary exponent. This conversion is always exact.

Arguments:

I - Constant, variable, or expression of type INTEGER.

Function result:

Am9512 single precision representation of I.

Warnings/Limitations:

None.

Application notes:

Use FLOATI whenever an integer value needs to be converted to a 9512 real.

Usage example:

```
ONE:=FLOATI(1);
```

3.7.4 FLOATL

Purpose:

Convert a 32-bit two's complement integer to Am9512 floating-point.

Declaration:

```
FUNCTION FLOATL(I:LONGINT):REAL;EXTERNAL;
```

Description:

Determine the sign of I, and take the absolute value. Normalize the integer, and use the shift count as the binary exponent. Extra bits are truncated.

Arguments:

I - Constant, variable, or expression of type LONGINT.

Function result:

Am9512 single precision representation of I.

Warnings/Limitations:

From zero to eight bits of precision may be lost during normalization, depending on the magnitude of I.

Application notes:

FLOATL is used to convert long integers to 9512 single precision numbers.

Usage example:

```
IO_RATE:=FLOATL(100L*NWORDS)/FLOATL(T_ELAPSED);
```

3.7.5 IFIX

Purpose:

Convert an Am9512-format single precision value to a 16-bit two's complement integer.

Declaration:

```
FUNCTION IFIX(X:REAL):INTEGER;EXTERNAL;
```

Description:

Break the floating-point number up into sign, exponent, and mantissa. De-normalize the mantissa based on the value of the exponent. Fractional digits in the mantissa are truncated. If necessary, complement the number to account for the sign.

Arguments:

X - A variable or expression of type REAL in 9512 format.

Function result:

A 16-bit number of type INTEGER.

Warnings/Limitations:

X must be in the range -(2**15) to (2**15)-1, or else integer overflow will occur.

The integer result is obtained by truncating (rather than rounding) the mantissa. From 9 to 24 bits of precision may be lost, depending on the value of the exponent.

Application notes:

Use IFIX to convert 9512-format numbers to integers.

Usage example:

```
N:=IFIX(ADD(X[1],Y[1]));
```

3.7.6 LFIX

Purpose:

Convert an Am9512-format single precision value to a 32-bit two's complement integer.

Declaration:

```
FUNCTION LFIX(X:REAL):LONGINT;EXTERNAL;
```

Description:

Break the floating-point number into sign, exponent, and mantissa. De-normalize the mantissa based on the value of the exponent. The integer result may either have digits truncated, or be padded with trailing zeroes, depending on the value of the exponent. If the sign is negative, the integer is complemented.

Arguments:

X - Variable, or expression of type REAL in 9512 format.

Function result:

A 32-bit number of type LONGINT.

Warnings/Limitations:

X must be in the range -(2**31) to (2**31)-1, or else integer overflow will occur.

The integer result is obtained by either truncation or padding of the mantissa (rather than rounding). Up to 7 trailing zeroes may be appended to the result, or up to 24 bits of precision may be lost by truncation, depending on the value of the exponent.

Application notes:

Use LFIX to convert floating-point values to long integers.

Usage example:

```
NMAX:=LFIX(MULT(X[1],Y[1]));
```

3.7.7 SINGLE

Purpose:

Convert a double precision Am9512-format number to a single precision Am9512-format number.

Declaration:

```
FUNCTION SINGLE(X:REAL(16)):REAL;EXTERNAL;
```

Description:

Convert the exponent from double precision to single precision by changing the bias, and re-align the mantissa for single precision format.

Arguments:

X - A variable or expression of type REAL(16) in 9512 format.

Function result:

A single precision number of type REAL in 9512 format.

Warnings/Limitations:

The double precision number must be in the range 10**(-38) to 10**(+38), or else exponent underflow or overflow will occur.

Mantissa conversion results in the truncation of the 29 least-significant bits of X.

Application notes:

Use SINGLE to convert double precision numbers to single precision. This can be useful when maximum precision is desired for calculations, but only single precision is required to store the final result.

Usage example:

```
SUM:=SINGLE(DSUB(DF[J],DVDP(N,DELTA,K[I])));
```

3.7.8 DV9512

Purpose:

Convert a 990-format double precision floating-point value to 9512 format.

Declaration:

```
FUNCTION DV9512(X:REAL(16)):REAL(16);EXTERNAL;
```

Description:

Re-align the mantissa, normalizing for binary digits rather than hexadecimal digits. Convert the exponent from hex to binary, and adjust for shifting of the mantissa.

Arguments:

X - Constant, variable, or expression of type REAL(16) in TI 990 format.

Function result:

Am9512 representation of X.

Warnings/Limitations:

From zero to three bits of precision will be lost in the Am9512 number, since there are fewer bits in the mantissa.

Application notes:

Use DV9512 to convert program constants, and data which originated on the Controller.

Usage example:

```
D1:=DV9512(1.0Q0);
```

3.7.9 DV990

Purpose:

Convert a 9512-format double precision floating-point value to 990 format.

Declaration:

```
FUNCTION DV990(X:REAL(16)):REAL(16);EXTERNAL;
```

Description:

Convert the exponent from binary to hexadecimal and normalize the mantissa based on hexadecimal digits.

Arguments:

X - Variable or expression of type REAL(16) in Am9512 format.

Function result:

990-format representation of X.

Warnings/Limitations:

X must be in the range 10**(-78) to 10**(+75), or else exponent underflow or overflow will occur.

From zero to three extra bits of precision can be represented in the 990-format result. These are filled with trailing zeroes.

Application notes:

CV990 is used to convert double precision results to TI 990 format before uploading them to the Controller.

Usage example:

```
FOR I:=1 TO MAX_ELEMENTS DO
  SDATA@[I]:=DV990(SDATA@[I]);
```

3.7.10 DFLOTI

Purpose:

Convert a 16-bit two's complement integer to Am9512 double precision floating-point.

Declaration:

```
FUNCTION DFLOTI(I:INTEGER):REAL(16);EXTERNAL;
```

Description:

The sign of I is determined, and the absolute value is found. The integer is normalized (right-shifted), and the shift count becomes the binary exponent. This conversion is always exact.

Arguments:

I - Constant, variable, or expression of type INTEGER.

Function result:

Am9512 double precision representation of I.

Warnings/Limitations:

None.

Application notes:

Use DFLOTI to convert integers to double precision reals.

Usage example:

```
X1:=DDIVD(X,DFLOTI(N MOD 64));
```

3.7.11 DFLOTL

Purpose:

Convert a 32-bit two's complement integer to Am9512 double precision floating-point.

Declaration:

```
FUNCTION DFLOTL(I:LONGINT):REAL(16);EXTERNAL;
```

Description:

Determine the sign of I, and take the absolute value. Normalize the integer, and use the shift count as the binary exponent. This conversion is always exact.

Arguments:

I - A constant, variable, or expression of type LONGINT.

Function result:

Am9512 double precision representation of I.

Warnings/Limitations:

None.

Application notes:

DFLOTL is used to convert long integers to 9512 double precision numbers.

Usage example:

```
DX:=DFLOTL(LX*2L);
```

3.7.12 IFIXD

Purpose:

Convert an Am9512-format double precision value to a 16-bit two's complement integer.

Declaration:

```
FUNCTION IFIXD(X:REAL(16)):INTEGER;EXTERNAL;
```

Description:

Break the floating-point number up into sign, exponent, and mantissa. De-normalize the mantissa based on the value of the exponent. Fractional digits in the mantissa are truncated. If necessary, complement the number to account for the sign.

Arguments:

X - A variable or expression of type REAL(16) in 9512 format.

Function result:

A 16-bit number of type INTEGER.

Warnings/Limitations:

X must be in the range -(2**15) to (2**15)-1, or else integer overflow will occur.

The integer result is obtained by truncating (rather than rounding) the mantissa. From 38 to 53 bits of precision may be lost, depending on the value of the exponent.

Application notes:

Use IFIXD to convert 9512-format double precision numbers to integers.

Usage example:

```
J:=IFIXD(DDIVD(DX[I],DV9512(10.0Q0));
```

3.7.13 LFIXD

Purpose:

Convert an Am9512-format double precision value to a 32-bit two's complement integer.

Declaration:

```
FUNCTION LFIXD(X:REAL(16)):LONGINT;EXTERNAL;
```

Description:

Split the floating-point number into sign, exponent, and mantissa. De-normalize the mantissa based on the value of the exponent. Fractional digits in the mantissa are truncated. If necessary, complement the number to account for the sign.

Arguments:

X - A variable or expression of type REAL(16) in 9512 format.

Function result:

A 32-bit number of type LONGINT.

Warnings/Limitations:

X must be in the range -(2**31) to (2**31)-1, or else integer overflow will occur.

The integer result is obtained by truncating (rather than rounding) the mantissa. From 22 to 53 bits of precision may be lost, depending on the value of the exponent.

Application notes:

LFIXD is used to convert double precision numbers to long integers.

Usage example:

```
B_PERCENT:=LFIXD(MULT(BUF_EFFICIENCY,DV9512(100.0Q0)));
```

3.7.14 DOUBLE

Purpose:

Convert a single precision Am9512-format number to a double precision Am9512-format number.

Declaration:

```
FUNCTION DOUBLE(X:REAL):REAL(16);EXTERNAL;
```

Description:

Convert the exponent from single precision to double precision by changing the bias, and re-align the mantissa for double precision format. The double precision result is filled in with 29 trailing zero bits.

Arguments:

X - Variable or expression of type REAL in 9512 format.

Function result:

A double precision number of type REAL(16) in 9512 format.

Warnings/Limitations:

None.

Application notes:

Use DOUBLE to convert single precision numbers to double precision. This can be useful when additional precision is required for intermediate calculations, or when the initial values are stored as single precision numbers.

Usage example:

```
DX:=DOUBLE(X);
```

3.8 Am9512 Mathematical Subroutines

This section describes a number of subroutines which provide specific mathematical services using the Am9512 floating-point processor. Figure 3-2 lists the routines by functional category. Routines denoted by "*" are coded in assembly language to take advantage of the Am9512's stack architecture for accumulating intermediate results. The performance of these routines will generally be significantly better than an equivalent algorithm coded using the individual operations of Section 3.5.

Whenever vectors or matrices are involved, the subroutines assume row-major storage order, consistent with TI Pascal.

All of the routines in this section assume that floating-point parameters are in 9512 format, and all results are returned as 9512-format numbers. Failure to heed this restriction will result in incorrect results and/or run-time errors.

Vector Operations	Random Numbers
* VDP	URAN
* DVDP	DURAN
	RANSEED

Trigonometric Functions	Roots
SINE	* SQRT95
DSINE	* DSQRT95

* Optimized for Am9512 stack architecture.

Figure 3-2. PASLIB Mathematical Subroutines

3.8.1 SQRT95

Purpose:

Computes the positive square root of an Am9512-format single precision number.

Declaration:

```
FUNCTION SQRT95(X:REAL):REAL;EXTERNAL;
```

Description:

The square root is found using the Newton-Raphson method. An initial guess X(0) is obtained by halving the exponent of X, and adding 1 to it. A sequence of approximations is computed such that X(i+1) = (X(i)+X/X(i))/2. The iteration is terminated when the sequence becomes non-decreasing, and the last X(i) is used as the result.

Arguments:

X - A non-negative variable or expression of type REAL in 9512 format.

Function result:

The postive square root of X.

Warnings/Limitations:

X must be non-negative.

Application notes:

Use SQRT95 in place of the TI Pascal SQRT routine to calculate square roots of 9512-format numbers.

Usage example:

```
C:=SQRT95(ADD(MULT(A,A),MULT(B,B)));
```

3.8.2 DSQRT95

Purpose:

Computes the positive square root of an Am9512-format double precision number.

Declaration:

```
FUNCTION DSQRT95(X:REAL(16)):REAL(16);EXTERNAL;
```

Description:

The double precision square root is calculated using the Newton-Raphson technique. See the description of SQRT95 for details.

Arguments:

X - A non-negative variable or expression of type REAL(16) in 9512 format.

Function result:

The positive square root of X.

Warnings/Limitations:

X must be non-negative.

Application notes:

Use DSQRT95 in place of the TI Pascal SQRT routine to calculate square roots of 9512-format double precision numbers.

Usage example:

```
DX:=DSUB(X2,X1);
DY:=DSUB(Y2,Y1);
DIST:=DSQRT95(ADD(MULT(DX,DX),MULT(DY,DY)));
```

3.8.3 VDP

Purpose:

Computes the dot product of two vectors.

Declaration:

```
FUNCTION VDP(N:POSINT;
                VAR A:ARRAY [1..?] OF REAL;
                VAR B:ARRAY [1..?] OF REAL):REAL;EXTERNAL;
```

Description:

Computes the dot product (scalar product, inner product) of two vectors A and B. This is defined as the sum of all A(i)*B(i) for i=1,2,...,N. For the sake of efficiency, A and B are passed as VAR parameters, but the contents of A and B are not modified. A and B may be of different lengths, but N must be less than or equal to the number of elements in the shorter of the two. The sum is accumulated on the 9512 stack.

Arguments:

N	- A positive constant, variable, or expression of type POSINT which indicates the number of elements in A and B to be multiplied and summed.
A,B	- Variables of type ARRAY [1..n] OF REAL containing 9512-format numbers.

Function result:

The dot product of A and B in 9512 single precision format.

Warnings/Limitations:

N must be less than or equal to the length of the shortest vector.

Possible exceptions: underflow, overflow.

Application notes:

Use VDP for efficient computation of dot products. A and/or B may be rows of a matrix, or a type transfer may be used to force a matrix to be passed as a vector with each row concatenated to the preceding row. This last technique assumes row-major storage of arrays. (See the usage example below.)

Usage example:

```
TYPE A12 = ARRAY [1..12] OF REAL;

VAR K:ARRAY [1..12] OF A12;
    DELTA:ARRAY [1..4,1..3] OF REAL;
    SUM:A12;
```

```
                 . . .

FOR I:=1 TO 12 DO  (* initialize K *)
  FOR J:=1 TO 12
    K[I,J]:= . . .

                 . . .

FOR I:=1 TO 12 DO
  SUM[I]:=VDP(12,K[I],DELTA::A12);
```

3.8.4 DVDP

Purpose:

Computes the dot product of two double precision vectors.

Declaration:

```
FUNCTION DVDP(N:POSINT;
              VAR A:ARRAY [1..?] OF REAL(16);
              VAR B:ARRAY [1..?] OF REAL(16)):REAL(16);
              EXTERNAL;
```

Description:

Computes the dot product of two double precision vectors A and B. This is defined as the sum of all A(i)*B(i) for i=1,2,...,N. For efficiency, A and B are passed as VAR parameters, but the contents of A and B are not modified. A and B may be of different lengths, but N must be less than or equal to the number of elements in the shorter of the two.

Arguments:

N - A positive constant, variable, or expression of type POSINT which indicates the number of elements in A and B to be multiplied and summed.

A,B - Variables of type ARRAY [1..n] OF REAL(16) containing 9512-format double precision numbers.

Function result:

The dot product of A and B in 9512 double precision format.

Warnings/Limitations:

N must be less than or equal to the length of the shortest vector.

Possible exceptions: underflow, overflow.

Application notes:

Use DVDP for efficient computation of double precision dot products. A and/or B may be rows of a matrix, or a type transfer may be used to force a matrix to be passed as a vector (assuming row-major storage order).

Usage example:

```
DX:=DVDP(M,DU,DV);
```

3.8.5 URAN

Purpose:

Generates the next number in a pseudo-random sequence uniformly distributed on the interval [0,1), and returns the result as an Am9512-format single precision value.

Declaration:

```
FUNCTION URAN:REAL;EXTERNAL;
```

Description:

The linear congruential method is used to generate a sequence of 32-bit pseudo-random numbers, which are interpreted as binary fractions on the interval [0,1). The implementation used here is based on recommendations given in Knuth, THE ART OF COMPUTER PROGRAMMING, Vol. 2, "Seminumerical Algorithms", Chapter 3, Section 6. This generator has a cycle length (period) of 2**32 (4,294,967,296), and each of the 2**32 possible numbers in the sequence will be produced exactly once each cycle. An arbitrarily chosen seed of 453290171 is used, but any long integer value can be used as a seed by calling procedure RANSEED.

URAN generates the next random number in sequence and converts the 32-bit fraction to a positive 9512-format single precision number in the interval [0.0,1.0). Since only 24 bits of precision are available in the mantissa, some duplicate values will be produced during a cycle due to truncation of less significant bits. If duplicate values are not acceptable for a particular application, function DURAN should be used instead.

Arguments:

None.

Function result:

An Am9512 single precision pseudo-random number in the interval [0.0,1.0).

Warnings/Limitations:

Because of truncation, duplicate numbers will occur within the cycle period of 2**32 calls.

Chi-square tests on the frequency distribution of numbers produced by URAN have given acceptable results using several different sample sizes and numbers of intervals. For critical applications, however, additional testing (such as a spectral test) may be desirable before depending on the randomness of the sequence produced by this routine. Tests for randomness are discussed in detail in Knuth, Vol. 2, Chapter 3.

Application notes:

Use URAN to generate test data for programs, or to drive simulations.

Usage example:

```
(*  Generate test data in the range 1..100  *)
R1:=CV9512(1.0);
R99:=CV9512(99.0);
FOR I:=1 TO 500 DO
  TEST_DATA[I]:=ADD(MULT(URAN,R99),R1);
```

3.8.6 DURAN

Purpose:

Generates the next number in a pseudo-random sequence uniformly distributed on the interval [0,1), and returns the result as an Am9512-format double precision value.

Declaration:

```
FUNCTION DURAN:REAL(16);EXTERNAL;
```

Description:

DURAN uses the same linear congruential generator as URAN, but converts the 32-bit fraction to a 9512-format double precision result. Since there is no truncation of the mantissa, all 2**32 possible random numbers can be represented with no duplication within a cycle of the generator.

Arguments:

None.

Function result:

An Am9512 double precision pseudo-random number in the interval [0.0,1.0).

Warnings/Limitations:

Before relying on DURAN for critical applications, additional randomness tests should be performed, as required by the application.

Application notes:

Use DURAN to generate test data for programs, or to drive simulations.

Usage example:

```
NEXT_SAMPLE:=LFIXD(DMULT(DURAN,DFLOTL(MAXLINT)));
```

3.8.7 RANSEED

Purpose:

Select an alternate seed for the random number generator.

Declaration:

```
PROCEDURE RANSEED(SEED:LONGINT);EXTERNAL;
```

Description:

The long integer SEED is used as the initial value for the linear congruential random number generator used by URAN and DURAN. SEED may be any LONGINT value. If RANSEED is not called, an arbitrarily chosen default seed of 453290171 is used. A given seed value always produces the same sequence of pseudo-random numbers.

Arguments:

SEED - A constant, variable, or expression of type LONGINT.

Warnings/Limitations:

None.

Application notes:

Use RANSEED to alter the sequence of random numbers produced by URAN and DURAN. Seeds based on a processor's self-ID (Section 3.12) can be used to produce different sequences on each processor. When using random numbers to produce test data, this approach can give more thorough test coverage with no increase in execution time by running different test sets in parallel on multiple processors. For simulations, asynchronous behavior can be readily produced by using different seeds on different processors to generate delay counts or time steps.

Usage example:

```
RANSEED(LINT(LSELF)*32771L);
```

3.8.8 SINE

Purpose:

Computes the sine function using Am9512 single precision arithmetic.

Declaration:

```
FUNCTION SINE(X:REAL):REAL;EXTERNAL;
```

Description:

The sine of X is approximated using recurrence relations to compute the trigonometric series for the sine function. To improve convergence and reduce roundoff error, X is first mapped into the interval [-pi/2,+pi/2] using trigonometric reduction relations for the sine function. For more details on the method, see N. Wirth, SYSTEMATIC PROGRAMMING: AN INTRODUCTION, Chapter 9, Prentice-Hall, 1973.

Arguments:

X - A variable or expression of type REAL in 9512 format. X is assumed to be in units of radians.

Function result:

The (approximate) sine of X.

Warnings/Limitations:

For large values of X (about 10**7 or larger), there are insufficient significant bits to determine the quadrant in which X lies, so the result is arbitrarily set to 0.0. SINE appears to give best results for numbers on the order of 10**0.

Very small values of X may result in exponent underflow, but the answer will still be approximately correct.

Application notes:

Use SINE in place of the TI Pascal SIN function to compute the sine of 9512-format single precision numbers.

Usage example:

```
D2R:=CV9512(1.745329E-2);  (* degrees-to-radians *)
FOR I:=0 TO 360 DO         (* compute sine curve *)
  Y[I]:=SINE(MULT(FLOATI(I),D2R));
```

3.8.9 DSINE

Purpose:

Computes the sine function using Am9512 double precision arithmetic.

Declaration:

```
FUNCTION DSINE(X:REAL(16)):REAL(16);EXTERNAL;
```

Description:

The sine of X is approximated using recurrence relations to compute the trigonometric series for the sine function. To improve convergence and reduce roundoff error, X is first mapped into the interval [-pi/2,+pi/2]. The algorithm used is similar to that for the SINE routine.

Arguments:

X - A variable or expression of type REAL in 9512 format. X is assumed to be in units of radians.

Function result:

The (approximate) sine of X.

Warnings/Limitations:

For large values of X (about 10**15 or larger), there are insufficient significant bits to determine the quadrant in which X lies, so the result is arbitrarily set to 0.0. DSINE gives best results for numbers on the order of 10**0.

Very small values of X may result in exponent underflow, but the answer will still be approximately correct.

Application notes:

Use DSINE in place of the TI Pascal SIN function to compute the sine of 9512-format double precision numbers.

Usage example:

```
FUNCTION DCSC(X:REAL(16)):REAL(16);
  (*  Double precision cosecant  *)
  BEGIN
  DCSC:=DDIVD(DV9512(1Q0),DSINE(X))
  END;
```

3.9 Sum/Maximum

The sum/max network is not operational at this time. Therefore, no routines are provided for its use. Global calculations can be performed by using the neighbor communication routines (Section 3.10) to transmit operands and results among processors.

3.10 Neighbor Communications

Routines are provided for transmitting data between neighboring processors. For two processors to be neighbors, the following must be true prior to execution:

(1) Each processor must reference the other in the list of logical neighbors stored in data area 2.

(2) If data area 1 is defined, it must contain a mapping table which assigns logical neighbor numbers to particular processors. If data area 1 is undefined, an identity mapping is assumed, and logical neighbor numbers are equivalent to the self-IDs of the processors. Data area 1 is ordinarily defined and initialized by the RESET command (see the FACS User's Guide).

(3) Connectivity must be established and input buffers allocated using the SYNCON or ASYNCON command (FACS User's Guide).

All of the neighbor communication routines use logical neighbor numbers to identify processors. This allows programs to be written without regard to the actual processor on which it will execute. That information is supplied independently of the program by the mapping table. Note that the mapping determines whether neighbors will communicate via the local links or the global bus, and hence can have an impact on program performance.

Two communication modes are available, synchronous or asynchronous. These are selected by using either the SYNCON or ASYNCON command, respectively, to establish connectivity. In synchronous mode, data is received and stored by source processor number in first-in first-out (FIFO) software queues. The depth of these queues is specified by the SYNCON command. Ordinarily a queue depth of two is used, which allows neighbors to be up to one iteration out of step with each other. Larger queue depths may be used to buffer multiple records. Tightly synchronized programs may be able to use a queue depth of one. If the queue is empty, an attempt to get data with the RECV or RECV2 routine will cause the processor to wait until data arrives from the neighboring processor. This property causes synchronization of the processors based on the arrival time of data from neighbors.

In asynchronous mode, data which is received overwrites previously received data from the same processor. The RECV and RECV2 routines do not wait for data, but return the most recently received value. If a processor is running more slowly than its neighbor, some of the data sent to it from the neighbor may never be used, but will instead be replaced by more recently received data. If, however, a processor is running faster than its neighbor, some of the received values may be used repeatedly. Since no data synchronization is performed, asynchronous mode may be used to implement asynchronous or "chaotic" algorithms which allow processors to proceed at different rates.

NOTE

When using asynchronous mode, it is important that an initial value

be sent from each processor to its neighbors, and that this value be received at its destination before executing the first call to the RECV or RECV2 routines. The BAR (Section 3.4.9) and DLY (Section 3.11.3) routines can be used to enforce this requirement.

The same subroutines are used regardless of which communication mode was selected; the mode is implemented transparently within Nodal Exec and PASLIB. The IO$MODE routine can be used by a program to determine whether synchronous or asynchronous mode was selected. It is therefore possible to write a single program which can run using either communication technique.

Neighbor communication is based on sending and receiving data records. A record may be as small as a single data item, or as large as an entire array (subject to a maximum record length of 255 words, or 510 bytes). The user specifies a maximum record length for his program in the range 1-255 words, using the connectivity commands. During execution, records sent or received must be less than or equal to the specified maximum length. Figure 3-3 gives the size in words of commonly used TI Pascal data types.

The use of record-oriented communication does not imply that data to be transmitted/received must be stored as Pascal RECORD types. The neighbor communication routines are general since they will support the use of any data type which occupies contiguous memory locations. Only the address of the data is passed as a parameter. The TIP LOCATION function can be used to obtain the address of specific data items.

An index or tag value may be associated with each record which is transmitted to a neighboring processor. This index tag can be used to distinguish between records which contain logically distinct information. For example, a different index could be used for each degree of freedom in a problem, or to identify colors in a multi-color solution technique. In synchronous mode, a separate queue is maintained for each index value. In asynchronous mode, separate buffers are used for each index value so that a freshly received record only overwrites a previous record with the same index value. The number of index tags required for a problem is specified using the connectivity commands, and must be in the range from 1 to 255. When using index tags, the SEND2 or SEND2ALL routines are used for transmission, and the RECV2 routine is used for reception. The SEND, SENDALL, and RECV routines do not allow an index to be specified, but assume an index value of 1.

Data may be transmitted to either specific neighbors or to all neighbors. The SEND and SEND2 routines require a destination parameter which indicates a specific neighbor. The SENDALL and SEND2ALL routines transmit data to all of the logical neighbors specified in data area 2. In this case, parallel output circuitry allows the data to be sent to all local neighbors simultaneously, but the data must be transmitted individually to each global neighbor. The send-all operation should not be confused with a global bus broadcast, which is not supported in the current hardware/software implementation. Data reception is always by individual neighbor, using either the RECV or RECV2 routines.

TYPE	SIZE (words)
CHAR	1
PACKED ARRAY OF CHAR	2 chars/word
INTEGER	1
LONGINT	2
REAL	2
REAL(16)	4

Figure 3-3. Sizes of Commonly Used Pascal Data Types.

3.10.1 SEND

Purpose:

Transmit data to a neighboring processor.

Declaration:

```
PROCEDURE SEND(N:NODE; LOC:ADDR; NWORDS:RECLEN);EXTERNAL;
```

Description:

N is converted from a logical neighbor number to a physical processor number according to the mapping table stored in data area 1. If data area 1 is undefined, N is assumed to be the physical processor number, i.e., an identity mapping is used. An output buffer is allocated of sufficient size to hold NWORDS of data plus a header word and a trailing checksum. An index tag value of 1 is assumed and stored in the header word along with the record length. NWORDS of data are copied from LOC and placed in the buffer. The checksum is calculated and stored as the last word in the buffer. N is determined to be either a local or global neighbor, and the output buffer is inserted at the end of the appropriate output queue.

Arguments:

N	- Logical neighbor number of the destination processor.
LOC	- Address of the data to be transmitted.
NWORDS	- Number of data words to be transmitted.

Warnings/Limitations:

NWORDS must be less than or equal to the maximum record length specified in the connectivity command.

Application notes:

Use SEND to transmit data to a specific neighbor.

Usage example:

```
CONST NNBRS = 6;
      DOF = 2;

TYPE DA2 = ARRAY [1..NNBRS] OF NODE;

VAR NBRS:@DA2;
    DSPL:ARRAY [1..NNBRS,1..DOF] OF REAL(16);
    DPTR:ARRAY [1..NNBRS] OF ADDR;

                . . .

NBRS::ADDR:=DAPTR(2);
FOR I:=1 TO NNBRS DO
  DPTR[I]:=LOCATION(DSPL[I]);
```

```
                    . . .

    FOR I:=1 TO NNBRS DO
      SEND(NBRS@[I],DPTR[I],8);  (* send a row to nbr[i] *)
```

3.10.2 SEND2

Purpose:

Transmit data to a neighboring processor and identify the data with an index tag.

Declaration:

```
PROCEDURE SEND2(N:NODE; INDEX:IDX; LOC:ADDR; NWORDS:RECLEN)
                ;EXTERNAL;
```

Description:

Same as SEND, except that the index tag is passed as a parameter.

Arguments:

N	- Logical neighbor number of the destination processor.
INDEX	- A user-defined value which identifies the kind of data to be transmitted.
LOC	- Address of the data to be transmitted.
NWORDS	- Number of data words to be transmitted.

Warnings/Limitations:

INDEX must be less than or equal to the maximum index specified in the connectivity command.

NWORDS must be less than or equal to the maximum record length specified in the connectivity command.

Application notes:

Use SEND2 to identify data which is sent to a specific neighbor. This can be particularly useful if more than one type of data record is being transmitted, if the order of reception of data does not guarantee its type, or when asynchronous I/O is being used with multiple data records.

Usage example:

```
CONST NBR_CNT = 1;  (* Tags: 1 = # of neighbors   *)
      NBRS = 2;     (*       2 = set of neighbors *)

      MAXNBRS = 6;  (* Max # of nbrs per node *)

TYPE NBR_LIST = SET OF NODE;

VAR CONNECTIONS:NBR_LIST;
    NNBRS:0..MAXNODE;

              . . .

FOR I:=1 TO MAXNBRS DO
```

```
IF I <> LSELF THEN
  BEGIN
  NNBRS:=0;
  CONNECTIONS:=[]; (* empty set *)
  FOR J:=1 TO MAXNBRS DO
    IF CONNECTED(I,J) THEN
      BEGIN
      CONNECTIONS:=CONNECTIONS+[J];
      NNBRS:=NNBRS+1
      END;
  SEND2(I,NBR_CNT,LOCATION(NNBRS),1);
  SEND2(I,NBRS,LOCATION(CONNECTIONS),
        (SIZE(NBR_LIST)+1) DIV 2)
  END
ELSE
   . . .
```

3.10.3 SENDALL

Purpose:

Transmit data to all neighboring processors.

Declaration:

```
PROCEDURE SENDALL(LOC:ADDR; NWORDS:RECLEN);EXTERNAL;
```

Description:

A record header word is built with an assumed index tag of 1 and a record length obtained from NWORDS. A checksum is computed for the NWORDS of data beginning at LOC. For each global neighbor, an output buffer is allocated, the data is copied from LOC into the buffer, the header word and checksum are inserted, and the buffer is placed on the end of the global output queue. An additional buffer is allocated for output to all local neighbors, and the record header, data, and checksum are copied into this buffer. The buffer is then inserted at the end of the local output queue for simultaneous transmission to all local neighbors.

Arguments:

LOC - Address of the data to be transmitted.
NWORDS - Number of data words to be transmitted.

Warnings/Limitations:

NWORDS must be less than or equal to the maximum record length specified in the connectivity command.

Application notes:

Use SENDALL whenever a data value must be transmitted to all of the neighbors listed in data area 2. SENDALL is much faster than using multiple calls to SEND, since data is sent in parallel to all of the local neighbors.

Usage example:

```
TYPE DSPL_REC = RECORD
                  X:REAL;
                  Y:REAL;
                  Z:REAL
                END;

VAR DELTA:DSPL_REC;

         . . .

SENDALL(LOCATION(DELTA),6);
```

3.10.4 SEND2ALL

Purpose:

Transmit data to all neighboring processors and identify the data with an index tag.

Declaration:

```
PROCEDURE SEND2ALL(INDEX:IDX; LOC:ADDR; NWORDS:RECLEN);
                  EXTERNAL;
```

Description:

Same as SENDALL, except that the index tag is passed as a parameter.

Arguments:

INDEX - A user-defined value which identifies the kind of data to be transmitted.
LOC - Address of the data to be transmitted.
NWORDS - Number of data words to be transmitted.

Warnings/Limitations:

INDEX must be less than or equal to the maximum index specified in the connectivity command.

NWORDS must be less than or equal to the maximum record length specified in the connectivity command.

Application notes:

Use SEND2ALL whenever a data value to be transmitted to all neighbors needs to be identified as to its contents. Such tag information can be useful when multiple record types are being transmitted, when the data is not ordered by time, or when asynchronous I/O mode is being used with multiple records.

Usage example:

```
VAR DSPL:REAL;
    DSPL_PTR:ADDR;

        . . .

DSPL_PTR:=LOCATION(DSPL);

        . . .

FOR I:=1 TO NDOF DO
  BEGIN
  DSPL:=SUB(F[I],VDP(N,K[I],DELTA));
  SEND2ALL(I,DSPL_PTR,2);
```

```
        . . .
    END;
        . . .
```

3.10.5 RECV

Purpose:

Receive data from a neighboring processor.

Declaration:

```
PROCEDURE RECV(N:NODE; LOC:ADDR; NWORDS:RECLEN);EXTERNAL;
```

Description:

N is converted from logical neighbor number to physical processor number based on the mapping table in data area 1. If data area 1 is undefined, N is assumed to be the physical processor number (an identity mapping). An index tag value of 1 is assumed. NWORDS of data are then read from the input buffer for neighbor N, index 1. If the communication mode is synchronous, the input buffer is a queue, and RECV will wait for data to arrive if less than NWORDS are present in the queue. If the mode is asynchronous, RECV will immediately retrieve whatever value is in the buffer. The data is copied from the buffer and stored at the address specified by LOC.

Arguments:

N	- Logical neighbor number of the source processor.
LOC	- Address where the received data will be stored.
NWORDS	- Number of data words to be read.

Warnings/Limitations:

NWORDS must be less than or equal to the maximum record length specified in the connectivity command.

Data must be read in the same order in which it arrives. For example, suppose that processor A sends two records to processor B using the SEND routine, and that the first record contains two integers and the second record contains four single precision reals. Then processor B must first RECV the integers and then the reals. This could be done by calling RECV twice, once for the integers and once for the reals, or else one call could be made with NWORDS = 10 to read a single record which contains two integers followed by four reals. If processor A did not always send the data in the same order, then the SEND2 and RECV2 routines should be used, with different index tags for the integers and the reals.

If asynchronous communication mode is used, data which is received from a neighbor overwrites previously received data from that neighbor.

Application notes:

Use the RECV routine to read data which has been sent from a neighboring processor. If synchronous communication mode is selected, RECV can be used in a message-passing scheme to synchronize processors.

Usage example:

```
VAR STRING: PACKED ARRAY [0..N_NEIGHS,1..80] OF CHAR;

          . . .

STRING[0]:= . . . ;
SENDALL(LOCATION(STRING[0]),40);
FOR I:=1 TO N_NEIGHS DO
  (* Get text from each neighbor *)
  RECV(DA2PTR[I],LOCATION(STRING[I]),40);
```

3.10.6 RECV2

Purpose:

Receive data from a neighboring processor and identify the data with an index tag.

Declaration:

```
PROCEDURE RECV2(N:NODE; INDEX:IDX; LOC:ADDR; NWORDS:RECLEN)
                ;EXTERNAL;
```

Description:

Same as RECV, except that the index tag is passed as a parameter.

Arguments:

N	- Logical neighbor number of the source processor.
INDEX	- A user-defined value which identifies the kind of data to be received.
LOC	- Address where the received data will be stored.
NWORDS	- Number of data words to be read.

Warnings/Limitations:

INDEX must be less than or equal to the maximum index specified in the connectivity command.

NWORDS must be less than or equal to the maximum record length specified in the connectivity command.

Data from a given processor with a given index tag must be read in the same order in which it arrives. If asynchronous communication mode is used, data which is received from a neighbor will overwrite previously received data from that neighbor which has the same index tag.

Application notes:

Use RECV2 to read data which has been sent from a neighboring processor with an index tag. The tag can be used to distinguish different types of data, or to prevent overwriting when multiple records are sent using asynchronous I/O mode.

Usage example:

```
WHILE NOT ALL(FLAG2) DO
  BEGIN
  FOR I:=1 TO NNBRS DO
    FOR J:=1 TO NDOF DO
      RECV2(NEIGH@[I],J,LOCATION(DELTA[I,J]),DSIZE);

        . . .
```

```
END;
```

3.10.7 IO$MODE

Purpose:

Allows a program to determine the communication mode which has been selected.

Declaration:

```
FUNCTION IO$MODE:INTEGER;EXTERNAL;
```

Description:

The communication mode which was set by the SYNCON or ASYNCON command is obtained from Nodal Exec.

Arguments:

None.

Function result:

1 : Synchronous mode was selected.
2 : Asynchronous mode was selected.

Warnings/Limitations:

Connectivity must be established via the SYNCON or ASYNCON FACS command before calling IO$MODE.

Application notes:

IO$MODE can be used by a program to determine, at runtime, the communication mode being used. The program can use this information to adapt to either a synchronous or asynchronous environment.

Usage example:

```
IF IO$MODE = 1 THEN SYNCHRONOUS:=TRUE
                ELSE SYNCHRONOUS:=FALSE;
```

3.10.8 GBUSY

Purpose:

Determines if global bus output is pending for this processor.

Declaration:

```
FUNCTION GBUSY:BOOLEAN;EXTERNAL;
```

Description:

Checks the global output buffer list to see if any data buffers are queued for output on the global bus.

Arguments:

None.

Function result:

TRUE : Global bus output is pending.
FALSE : No buffers are waiting on the global output buffer list (i.e. all global bus output has been moved to the hardware output FIFOs).

Warnings/Limitations:

The absence of queued buffers does not necessarily imply that the data has arrived at its destination, since there may still be untransmitted data in the hardware output FIFOs of the sending processor, or unprocessed data in the hardware input FIFOs on the receiving processor.

Application notes:

GBUSY may be used to determine if global bus output is backlogged. This information may be useful in load-balancing applications.

Usage example:

```
IF GBUSY THEN  (* Slow down *)
  DLY(50);
```

3.11 Timing

Two separate timing facilities are provided. One is an elapsed execution time based on the TMS9902 interval timer. This timer is maintained by Nodal Exec and has a clock period of 16 milliseconds. It starts counting when a program begins execution on the processor, halts if a halt command is issued to the processor or a breakpoint is encountered, and resumes when a resume command is issued. The execution timer stops when the program terminates normally or aborts with a fatal error, or when a kill command is issued to the processor. The XTIME and XTIME1 routines allow this timer to be interrogated from user programs. The DLY routine uses this timer to delay programs for a specified interval.

The other timer is based on the TMS9901 interval timer, and is under the control of the user's program. The clock period is selectable from 1 to 349 milliseconds. The timer is started with the TSTART routine, and is stopped with the TSTOP routine or when the program terminates or aborts or is killed. This timer is not halted when a halt command or breakpoint occurs, but continues to accumulate time until TSTOP is called or the program ends. The TREAD and TREAD1 routines are used to interrogate this timer.

In certain situations the timers may not accurately capture all of the elapsed time. This typically occurs when critical sections in the memory management routines of Nodal Exec lock out interrupts for intervals longer than the clock periods of the timers. In these cases the timer interrupts cannot be serviced before the clock "ticks" again, and thus one or more ticks may be lost, producing a timer overrun error. Overrun errors usually occur only when memory becomes badly fragmented due to a backlog of untransmitted output data buffers. The execution statistics report (see Section 5.4) can give some clue if this situation occurred. If the memory allocation or deallocation efficiencies are low, then timer overrun errors are a possibility. The execution statistics will report the number of overrun errors (if any) which occured for the TMS9902 timer. Overrun errors for the TMS9901 timer cannot be detected by software, but the capture ratio improves as the timer period increases. Clock periods of 50 milliseconds or longer should be immune to overrun errors. If overrun errors do occur, then the times measured will be somewhat less than the actual elapsed times. Timer overruns are not expected to be a common or serious problem, and most users should be able to get accurate timing data with little or no trouble.

3.11.1 XTIME

Purpose:

Obtain the approximate elapsed execution time, in milliseconds, of the currently executing program.

Declaration:

```
FUNCTION XTIME:LONGINT;EXTERNAL;
```

Description:

The elapsed execution time, based on the TMS9902 timer, is obtained from Nodal Exec.

Arguments:

None.

Function result:

The approximate elapsed execution time in milliseconds.

Warnings/Limitations:

The result should be regarded as approximate for two reasons:
(1) the timer resolution is plus or minus 16 milliseconds, and
(2) timer overrun errors may occur in certain situations.

To obtain good results, intervals to be measured should be several times the 16 millisecond timer period.

The maximum elapsed execution time before LONGINT overflow is about 24 days.

Application notes:

XTIME may be used to determine the elapsed execution time of programs or program segments. The duration of a program segment can be measured by calling XTIME at the beginning of the segment and again at the end; the difference is the elapsed time for that segment.

Usage example:

```
        . . .

MSG('Total execution time = ');
MSGL(XTIME);
MSGLN(' msec.')
END;  (* PSCL$$ *)
```

3.11.2 XTIME1

Purpose:

Obtain the approximate elapsed execution time, in hours, minutes, and seconds, of the currently executing program.

Declaration:

```
PROCEDURE XTIME1(VAR HMS:PACKED ARRAY [1..?] OF CHAR);
                EXTERNAL;
```

Description:

The elapsed execution time in milliseconds is obtained from Nodal Exec. This is rounded to the nearest number of seconds, and converted to hours, minutes, and seconds. The result is returned as a character string of the form "hh:mm:ss".

Arguments:

HMS - A packed array of characters containing at least eight elements. The result is stored in the first eight bytes of this string.

Warnings/Limitations:

The result should be regarded as approximate for two reasons:
(1) the resolution of the result is plus or minus 516 milliseconds, and
(2) timer overrun errors may perturb the result in some cases.

The maximum elapsed time to be measured by this routine should not exceed 99 hours, 59 minutes, and 59 seconds, or about 4 days.

Application notes:

Use XTIME1 to express execution times in an hh:mm:ss format.

Usage example:

```
VAR EXTIME:PACKED ARRAY [1..10] OF CHAR;

        . . .

EXTIME[9] :=' ';
EXTIME[10]:=' ';

FOR I:=1 TO MAXITERS DO
  BEGIN

        . . .

  XTIME1(EXTIME);
```

```
        MSG(EXTIME);
        MSG('End of iteration ');
        MSGI(I); NXTLN
        END;
```

3.11.3 DLY

Purpose:

Delay a program for a specified period of time.

Declaration:

```
PROCEDURE DLY(T:INTEGER);EXTERNAL;
```

Description:

First, the current elapsed execution time is obtained, and the specified delay time is added to determine the end of the delay period. The routine loops, polling the execution timer, until the elapsed time equals or exceeds the end of the delay period. Control is then returned to the calling program.

Arguments:

T - Integer constant, variable, or expression which is the requested delay time in milliseconds.

Warnings/Limitations:

The requested delay time, T, should be regarded as approximate for the following reasons:

(1) Timer resolution is 16 milliseconds. The actual delay time will normally be within plus or minus 16 milliseconds of the requested delay.
(2) Timer overrun errors could cause the actual delay to be longer than expected.
(3) Asynchronous activity on the processor (such as I/O interrupts) continues during the DLY routine. This activity could extend the actual delay period slightly if the I/O load is heavy.

Application notes:

DLY may be used to allow time for asynchronous operations (such as output interrupts) to occur before allowing the program to proceed. This technique can also be used to allow time for operations to occur on other processors or on the Controller, although in most cases flag barriers (Section 3.4.9) would be preferred for this purpose. DLY may also find uses in dynamic load-balancing, or as a method of deliberately de-synchronizing processors.

Usage example:

```
BAR(FLAG3);
DLY(PSELF*20);  (* de-synchronize *)
```

3.11.4 TSTART

Purpose:

Initialize and turn on the TMS9901 interval timer.

Declaration:

```
PROCEDURE TSTART(T:POSINT);EXTERNAL;
```

Description:

Sets the TMS9901 timer for a period of approximately T milliseconds, and begins timing. The minimum clock period is 1 millisecond, and the maximum is 349 milliseconds.

Arguments:

T - An integer constant, variable, or expression in the range from 1 to 349.

Warnings/Limitations:

The selected timer period ("tick") is not always exactly T milliseconds, but is as close to T as the hardware will allow. The maximum potential error is 0.021333... milliseconds/tick. The effect of this error decreases as the timer period increases. For a requested period T, the actual timer period is ((375*T+4) DIV 8)*0.021333... milliseconds.

Application notes:

Call TSTART to select a timer period and begin timing.

Usage example:

```
TSTART(5);  (* 5 mS timer period *)
```

3.11.5 TSTOP

Purpose:

Turn off the TMS9901 timer.

Declaration:

```
PROCEDURE TSTOP;EXTERNAL;
```

Description:

The TMS9901 timer is disabled and further timer interrupts are inhibited.

Arguments:

None.

Warnings/Limitations:

None.

Application notes:

Use TSTOP to turn off the timer at the end of an interval which is being measured.

Usage example:

```
TSTART(1);
FOR I:=1 TO N DO
  BEGIN

     . . .

  END;
TSTOP;
MSG('Time for loop = '); MSGL(TREAD);
MSGLN(' msecs.');
```

3.11.6 TREAD

Purpose:

Obtain the approximate elapsed time, in milliseconds, of an interval measured by the TMS9901 timer.

Declaration:

```
FUNCTION TREAD:LONGINT;EXTERNAL;
```

Description:

The number of clock ticks since the last call to TSTART is multiplied by the length of the timer period as specified in the last call to TSTART. The long integer result is the approximate elapsed time in milliseconds. TREAD may be called either while the timer is active (between a call to TSTART and TSTOP), or after the timer has been stopped by a call to TSTOP.

Arguments:

None.

Function result:

If the timer is active, the result is the approximate elapsed time in milliseconds since the last call to TSTART. If the timer has been stopped, the result is the approximate elapsed time in milliseconds between the call to TSTART and the call to TSTOP.

Warnings/Limitations:

The result is approximate for the following reasons:

(1) Timer resolution is plus or minus T, where T is the requested clock period.

(2) The actual timer period may not be exactly equal to T. See the description of the TSTART routine (Section 3.11.4) for details.

(3) Timer overrun errors may result in missed ticks. Short timer periods (a few milliseconds) are much more susceptible to this effect. Timer periods greater than a few tens of milliseconds should be virtually immune to overrun errors.

(4) The timer interrupts generated by the TMS9901 require 0.034 milliseconds overhead for each tick. This time should be subtracted from the result to account for perturbations induced by the timer.

The approximate elapsed time before LONGINT overflow is about 24 days.

Application notes:

Call TREAD to determine the duration of program segments. The selectable period (set by TSTART) allows events as short as one millisecond to be measured. If overrun errors are suspected, their severity can be measured

by repeatedly testing the program segment with a range of timer intervals. This will show, within the timer resolution, how much time is being lost due to missed ticks.

Usage example:

```
T:=256;
REPEAT
  TSTART(T);

      . . .

  TSTOP;
  MSG('Timer period = '); MSGI(T);
  MSG(', measured time = '); MSGL(TREAD);
  NXTLN;
  T:=T DIV 2
UNTIL T = 0;
```

3.11.7 TREAD1

Purpose:

Obtain the approximate elapsed time, in hours, minutes, and seconds, of an interval measured by the TMS9901 timer.

Declaration:

```
PROCEDURE TREAD1(VAR HMS:PACKED ARRAY [1..?] OF CHAR);
                EXTERNAL;
```

Description:

The elapsed time in milliseconds is obtained by a call to TREAD. This value is rounded to the nearest number of seconds, and converted to hours, minutes, and seconds. The result is returned as a character string of the form "hh:mm:ss".

Arguments:

HMS - A packed character array containing at least eight elements. The result is stored in the first eight bytes of this string.

Warnings/Limitations:

The result should be regarded as approximate since
  (1) the resolution of the result is plus or minus 500 milliseconds, and
  (2) the result is subject to all of the considerations listed for TREAD (Section 3.11.6).

The maximum interval to be measured by this routine should not exceed 99 hours, 59 minutes, and 59 seconds, or about 4 days.

Application notes:

Use TREAD1 to express time intervals in hours, minutes, and seconds.

Usage example:

```
VAR TELAPS:PACKED ARRAY [1..26] OF CHAR;

          . . .

TELAPS:='00:00:00  Completed cycle ';
TSTART(10);
FOR I:=1 TO N_CYCLES DO
  BEGIN

      . . .

  TREAD1(TELAPS);
  MSG(TELAPS); MSGI(I); NXTLN
```

```
END;
```

3.12 Processor Identification

A processor may be identified either by its physical location within the hardware system, or by its logical position within an algorithm. Functions are provided which allow programs to determine the physical (hardware) and logical (algorithmic) self-IDs of the processors on which they are executing.

3.12.1 PSELF

Purpose:

Obtain the physical self identifier of a processor.

Declaration:

```
FUNCTION PSELF:NODE;EXTERNAL;
```

Description:

The processor's hardware self-ID is obtained from Nodal Exec and returned to the calling program.

Arguments:

None.

Function result:

A value of type NODE which uniquely identifies the processor.

Warnings/Limitations:

None.

Application notes:

Use PSELF to determine which processor a program is executing on.

Usage example:

```
MSG('Processor '); MSGI(PSELF);
MSGLN(' beginning execution.');
```

3.12.2 LSELF

Purpose:

Obtain the logical self identifier of a processor.

Declaration:

```
FUNCTION LSELF:NODE;EXTERNAL;
```

Description:

The processor's logical self-ID is obtained from Nodal Exec and returned to the calling program. LSELF is set equal to PSELF when Nodal Exec is initialized. When connectivity is established with either the SYNCON or ASYNCON FACS command, LSELF is re-set based on the logical-to-physical mapping table loaded into data area 1. If an identity mapping is used, LSELF remains equal to PSELF. LSELF may be modified by loading a new mapping table and re-issuing one of the connectivity commands. If connectivity is cleared with the CLEAR command, LSELF reverts to PSELF.

Arguments:

None.

Function result:

A value of type NODE which identifies the processor's logical position within an algorithm, subject to the considerations mentioned above.

Warnings/Limitations:

LSELF returns the physical self-ID of the processor until after a mapping has been loaded and connectivity has been established.

Application notes:

Programs should use LSELF to determine a processor's identity within the context of a multi-processor algorithm. This capability allows programs to be written without knowledge of the actual physical processors on which they will execute. LSELF can be used to determine control paths through a program based on the logical processor on which it is executing.

Usage example:

```
FOR I:=1 TO N_PROCS DO
  IF I <> LSELF THEN
    SEND(I,LOC(X[I]),2);
```

4. EFFICIENCY CONSIDERATIONS

Many factors influence the performance of programs which execute on the Finite Element Machine. Some of these concerns are the same as those for sequential computers, for example, the algorithm chosen, the skill of the programmer, the amount of I/O involved, the quality of the code generated by the compiler, and the efficiency and internal organization of the operating system. Other concerns are introduced because of the parallel nature of FEM, including distribution of the workload, problem partitioning, processor synchronization, interprocessor communication, and complexity of control structures. The following sections discuss some of the things to bear in mind when writing efficient programs.

4.1 Compiler Options

Two categories of TIP compiler options have a significant effect on execution time. One category consists of the runtime checks (CKINDEX, CKOVER, CKSUB, etc.), and the other controls code optimization (GLOBALOPT, etc.).

The execution time checks are strongly recommended for use until a program has been fully debugged and tested. However, their use can result in significant increases in both execution time and size of object code. The default when no checking options are specified is no checking, which will result in maximum performance and minimum object code.

A number of optimization levels are available in TI Pascal. The default is OPTIMIZE, which enables simple statement-level optimizations. The GLOBALOPT option includes additional optimizations at the routine level. SPEEDOPT is supposed to modify the optimization strategy to improve performance, but with a possible increase in the object code size. The UNSAFEOPT option allows additional optimizations which attempt to improve register usage; however, these optimizations cannot be performed correctly for all programs. A program should first be thoroughly tested without UNSAFEOPT, and then re-tested with UNSAFEOPT turned on to assure that the optimizations are correct.

Experiments have shown that the best optimization strategy is highly program dependent. While SPEEDOPT improves the performance of some programs, it may actually degrade the performance of others (relative to GLOBALOPT). Similarly, UNSAFEOPT can be quite beneficial for some code, but may show little or no improvements in other cases.

Experience has also shown that the compiler occasionally makes mistakes when optimizing code. If a program is producing incorrect results and there are no detectable flaws in the logic, then optimization errors are a possibility. Test for optimization errors by disabling GLOBALOPT, SPEEDOPT, and UNSAFEOPT, and recompiling with NO OPTIMIZE. If the program still produces the same incorrect results, then the problem lies in the user's program or elsewhere in the compiler. Most optimization errors can be circumvented by minor rearrangements of the source code in the vicinity of the error, without resorting to NO OPTIMIZE for the production version. Optimizations are routine level options, so that NO OPTIMIZE can be restricted to a module which causes problems, while still allowing full optimization of other parts of the

program.

For a full description of the various compiler options, refer to Chapter 11 in the TI PASCAL REFERENCE MANUAL.

The FEM Project has adopted a standard set of compiler options which should be used when comparing the execution speeds or object code sizes of different programs. These are:

```
(*$GLOBALOPT,NO TRACEBACK,NO ASSERTS *)
```

Other options which have no effect on object code may be used as desired. Of these, WIDELIST and MAP are strongly recommended for debugging purposes.

4.2 Algorithms and Overhead

A useful measure of the overhead incurred by a parallel program is the parallel efficiency, defined as follows:

$$e = \frac{T(1)}{p*T(p)}$$

T(1) is the execution time for a uniprocessor implementation of an algorithm, and T(p) is the execution time for the same algorithm implemented on p processors. Values of e close to 1.0 indicate low overhead, while smaller values indicate higher overhead. Values greater than 1.0 might be realized by some asynchronous or combinatorially implosive algorithms.

This section discusses several of the more important issues which contribute to overhead in parallel programs written for FEM.

4.2.1 Workload

Distribution of the workload is a critical factor for synchronous algorithms. Each processor should be given approximately the same amount of work to do, so as to minimize the overall idle time. If a small percentage of processors have substantially more work to do than the rest, then the efficiency will be low because of high idle time on the larger number of processors which must wait.

4.2.2 Problem Partitioning

Related to the idea of workload is the concept of problem partitioning. Ideally a problem would be partitioned into p equal-sized pieces and solved on p processors. The fineness or granularity of the partitioning can affect the overhead of the parallel solution. Coarse partitionings generally have a high computation-to-communication ratio and high efficiency; very fine partitionings may have a low computation-to-communication ratio and low efficiency. For many problems there is an optimal partitioning of the problem onto some number of processors. If more processors are added, no improvement

will be seen because overhead costs become the dominant factor in execution time.

4.2.3 Synchronization

Two factors contribute to synchronization overheads. One is idle time, mentioned above, and the other is the mechanism used to achieve synchronization. PASLIB provides two different synchronization methods, one based on message passing and the other on the flag network.

When synchronous communication mode is chosen, the receive routines of Section 3.10 will wait for input from the specified processor if none is currently available in the input queue. This property can be used to bring processors into a loosely synchronized state. For algorithms which must transmit data between processors anyway, the communication routines can provide some or all of the required synchronization.

The preferred synchronization mechanism for most applications, however, is the flag barrier. The BAR routine (Section 3.4.9) incurs much less overhead than the communication routines, and guarantees that all participating processors have arrived at the same point in a program at the same time.

4.2.4 Communication

Data communication between processors is one of the major overheads in many parallel programs. The amount of communication between processors is determined by the nature of the problem and the algorithm selected to solve it. Some algorithms may require no interaction between processors, while others may move large amounts of data at frequent intervals.

The local link hardware provides the capability to send the same data to several processors simultaneously. This is supported by PASLIB in the form of the SENDALL and SEND2ALL procedures. Algorithms which transmit the same data to all neighbors will incur substantially less overhead by using SENDALL or SEND2ALL rather than repeated calls to SEND or SEND2. Efficient use of send-all depends on a logical-to-physical mapping which maximizes use of the local links and minimizes global bus communication.

Overhead for the communication routines can be divided into three components: (1) a fixed overhead for the I/O call which is independent of the amount of data transferred, (2) a data movement overhead which is a function of the amount of data to be transmitted, and (3) a dynamic interrupt overhead which depends on the number of send and receive interrupts generated.

For a given algorithm, the fixed overhead can be minimized by making as few SEND and RECV calls as possible. This implies that a few large blocks of data should be transmitted between processors, rather than many small ones. PASLIB supports this by allowing transmission of variable length data records up to a maximum size of 255 words.

Data movement overhead consists primarily of the time to copy data into and out of I/O buffers, but also includes the hardware transmission time. Data

movement can be reduced by using SENDALL and the local links wherever appropriate.

Interrupt overhead is sensitive to the sequencing and duration of local and global send and receive interrupts. Since FEM processors are asynchronous (do not share a common clock), the interrupt overhead is difficult to predict, and may vary from run to run, even though the program and data remain the same. Transmitting larger blocks of data will tend to reduce the number of interrupts, thereby decreasing the interrupt overhead. The depth of the hardware data buffers may impose an upper limit on interrupt efficiencies.

5. EXECUTION, ANALYSIS, AND DEBUGGING

The FEM Array Control Software (FACS) in conjunction with the Nodal Exec operating system provides facilities for data management, execution control, debugging, and performance analysis. The following sections outline the procedures for setting up, executing, debugging, and analyzing Pascal programs on the Array. For more detailed information, consult the FACS USER'S GUIDE and the relevant FEM Programming Memoranda.

5.1 Problem Setup

Before a program can be executed on the Array, several steps must be taken to set up the proper environment. The first step is to initialize the hardware and system software on the Array, and to define the logical-to-physical processor mapping (RESET command). Next, the set of processors to be used for the problem must be selected (SAC). If data areas are needed, they must be defined (DEFDAD, DEFDAI) and any necessary data must be loaded (LDAD, LDAI). If processors need to communicate with each other, data areas 0 and 2 must be defined, and a list of logical neighbors must be loaded into data area 2 on each processor. Data areas 0 and 2 must be identical in size and type of data (integer). After setting up data areas 0 and 2, connectivity must be established (SYNCON, ASYNCON). If there is no communication between processors, data areas 0 and 2 are not needed and connectivity may be ommitted.

Programs are loaded into processors (LDPG) from files of compressed object code which are produced by the link editor (see Section 2.3). The same code may be loaded into all processors, or different code may be loaded into different processors as required. Only one program may be loaded per processor. Loading a new program will automatically delete the previous one. When a program is loaded, the entry point of the linked module is stored; this value is used to initialize the processor's program counter when execution begins. For Pascal programs, the proper entry point is the address of module N$MAIN, usually relative address 0 (*0000). The link map may be used to verify the correct entry point. The AUTOSTAT command can be used to modify the entry point if needed.

The best order for defining data areas, loading programs, and establishing connectivity depends on the life-spans of the objects being allocated. FEM Programming Memo 2 discusses optimal memory management for Nodal Exec.

If a particular setup sequence is to be performed more than a few times, an SCI command procedure should be written to expedite the process. Consult the FACS USER'S GUIDE and Volume III of the DX10 OPERATING SYSTEM manual for more information about writing SCI procs.

5.2 Execution Control

FACS/Nodal Exec provide commands to execute, halt, resume, and kill programs on the Array. In addition, a breakpoint in a program causes an

internally generated halt. Programs may be executed repeatedly without reloading the object code. If a program is halted or killed, the program counter must be re-initialized to the entry address (STAT or AUTOSTAT command) before the program can be re-executed. Data areas need to be reloaded only if the program has modified the data, or if new data is needed for the next run. Large programs may be broken into multiple phases which run one after the other, with each new phase being loaded to replace the previous one. Intermediate results may be stored in data areas between phases, eliminating the need for moving large amounts of data to and from the Controller.

5.3 Debugging

All of the debugging tools of FACS/Nodal Exec are available to the Pascal programmer. These allow the programmer to inspect and modify memory and registers, set breakpoints, and single-step the processor. The SPSF (Show Pascal Stack on FEM) command was designed specifically for debugging TI Pascal programs. Use of SPSF requires a working knowledge of the TIP data structures which are described in Chapter 8 of the TI PASCAL PROGRAMMER'S GUIDE.

A useful strategy for initial testing of programs is to set breakpoints (SFB) at the entry addresses of procedures and functions. These can be either user-defined routines or PASLIB routines. The entry addresses are obtained from the symbol definitions (rather than the module map) produced as output from the link editor. Breakpoints should be set in the order that the program is expected to execute. In this way the programmer can follow the execution sequence and determine the approximate location of errors when they occur. If better resolution is desired, the Pascal reverse assembler (XRASS) can be used to determine the addresses of particular statements in the program, although this requires some knowledge of TMS9900 assembly language and TIP data structures and register conventions.

The SPSF command can be used at breakpoints to examine registers and variables in the Pascal stack. By modifying the workspace pointer field, variables at lower nesting levels in the stack can be examined. The workspace pointer must be restored to its original value before resuming execution.

5.4 Analysis

Nodal Exec incorporates two features which aid in the analysis of program execution. One of these is a trace capability which samples the program counter (PC) at specified intervals during execution. The PC values are sent to the Controller where they are stored for post-processing. By comparing the distribution of PC samples against the link editor module map, the percentage of time spent in different parts of the program (including PASLIB and much of Nodal Exec) can be determined. Programming Memo 3 discusses the trace sampling facility in detail.

The other analysis feature is a table of execution statistics which are automatically collected by Nodal Exec and PASLIB. Information is recorded about execution time, I/O, memory management, flags, and floating-point operations. This data may be post-processed to derive several measures of execution efficiency. Programming Memo 4 describes the execution statistics

report in detail.

APPENDIX A
EXAMPLE PROGRAM

Source Code

```
(*$WIDELIST,MAP,NO ASSERTS,NO TRACEBACK *)
(*$GLOBALOPT *)
(* CKINDEX,CKOVER,CKSUB *)

PROGRAM JACOBI;

(*-------------------------------------------------------------*)
(*                                                             *)
(*                  JACOBI V2.1  12/21/82                      *)
(*                                                             *)
(*  This program solves Loendorf's 4-node wing box problem     *)
(*  using either the standard or asynchronous Jacobi method.   *)
(*  The solution technique is determined by the I/O mode used *)
(*  in the connectivity command, either synchronous (SYNCON)   *)
(*  or asynchronous (ASYNCON).                                 *)
(*                                                             *)
(*  This version of the program has been modified to use the   *)
(*  record-oriented I/O of Nodal Exec/PASLIB V2.1.  The three *)
(*  displacement values calculated by each node are sent or    *)
(*  received in a single operation.                            *)
(*                                                             *)
(*  Data Areas                                                 *)
(*    0  - physical neighbor table                             *)
(*    1  - logical-to-physical mapping                         *)
(*    2  - logical neighbor table                              *)
(*    3  - reserved for system use                             *)
(*    4  - K matrix, 12 x 12                                   *)
(*    5  - vector of applied forces                            *)
(*    6  - input parameters:                                   *)
(*           NDOF   -number of degrees of freedom              *)
(*           NCON   -number of neighbors                       *)
(*           NNODES -total number of nodes                     *)
(*                                                             *)
(*  Must be run under V2.1 of Nodal Exec/PASLIB.               *)
(*                                                             *)
(*-------------------------------------------------------------*)


(*                         *)
(*  PASLIB DECLARATIONS    *)
(*                         *)

?COPY SYS1.FEM.PASLIB.UTIL$.TYPDCL

(*  FLAG ROUTINES  *)
```

```
FUNCTION ALL(F:FLAG):BOOLEAN;EXTERNAL;
FUNCTION FIRST:BOOLEAN;EXTERNAL;
PROCEDURE BAR(F:FLAG);EXTERNAL;
PROCEDURE FLGEN(F:FLAG);EXTERNAL;
PROCEDURE FLGRES(F:FLAG);EXTERNAL;
PROCEDURE FLGSET(F:FLAG);EXTERNAL;

(*  DATA AREA ACCESS  *)
FUNCTION DAPTR(DA:DANUM):ADDR;EXTERNAL;

(*  NEIGHBOR COMMUNICATIONS  *)
PROCEDURE SENDALL(LOC:ADDR; NWORDS:INTEGER);EXTERNAL;
PROCEDURE RECV(N:NODE; LOC:ADDR; NWORDS:INTEGER);EXTERNAL;
FUNCTION IO$MODE:INTEGER;EXTERNAL;

(*  FLOATING POINT ROUTINES  *)
FUNCTION CV9512(X:REAL):REAL;EXTERNAL;
FUNCTION MAX95:REAL;EXTERNAL;
FUNCTION CMP(X,Y:REAL):INTEGER;EXTERNAL;
FUNCTION ABS95(X:REAL):REAL;EXTERNAL;
FUNCTION SUB(X,Y:REAL):REAL;EXTERNAL;
FUNCTION DIVD(X,Y:REAL):REAL;EXTERNAL;
FUNCTION VDP(N:POSINT; VAR A:ARRAY [1..?] OF REAL;
             VAR B:ARRAY [1..?] OF REAL):REAL;EXTERNAL;

(*  OUTPUT ROUTINES  *)
PROCEDURE MSG(STRING:PACKED ARRAY[1..?] OF CHAR);EXTERNAL;
PROCEDURE MSGLN(STRING:PACKED ARRAY[1..?] OF CHAR);
          EXTERNAL;
PROCEDURE NXTLN;EXTERNAL;
PROCEDURE ENDLN(N:POSINT);EXTERNAL;
PROCEDURE MSGCH(CH:CHAR);EXTERNAL;
PROCEDURE MSGI(I:INTEGER);EXTERNAL;
PROCEDURE MSGL(I:LONGINT);EXTERNAL;
PROCEDURE MSGR(X:REAL);EXTERNAL;
PROCEDURE CWAIT;EXTERNAL;

(*  TIMER ROUTINES  *)
PROCEDURE TSTART(T:POSINT);EXTERNAL;
FUNCTION TREAD:LONGINT;EXTERNAL;
PROCEDURE TREAD1(VAR HMS:PACKED ARRAY[1..?] OF CHAR);EXTERNAL;
PROCEDURE TSTOP;EXTERNAL;

(*  MISCELLANEOUS  *)
FUNCTION LSELF:NODE;EXTERNAL;
FUNCTION PSELF:NODE;EXTERNAL;
FUNCTION GBUSY:BOOLEAN;EXTERNAL;


(*                *)
(*  MAIN PROGRAM  *)
(*                *)

PROCEDURE PSCL$$;
```

```
CONST FLAG0=0;              (*  FIRST FLAG              *)
      FLAG2=2;              (*  CONVERGENCE             *)
      FLAG3=3;              (*  SYNCHRONIZATION         *)
      EPSILON=1.0E-07;      (*  CONVERGENCE CRITERION   *)
      SYNCHRONOUS=1;        (*  SYNCHRONOUS I/O MODE    *)
      DSIZE=6;              (*  DATA RECORD SIZE        *)

TYPE DISPLACEMENTS = ARRAY [1..3] OF REAL;
     A12 = ARRAY [1..12] OF REAL;
     DA2 = ARRAY [1..3] OF NODE;
     DA4 = ARRAY [1..12] OF A12;
     DA5 = A12;

     DA6 = RECORD
             NDOF:IDX;
             NCON:1..3;
             NNODE:NODE
           END;

VAR DSPL,RO,MAXREAL,CNVRG:REAL;
    DELTA:ARRAY [1..4] OF DISPLACEMENTS;
    K:ARRAY [1..12] OF A12;
    F:A12;
    DELTAT:DISPLACEMENTS;
    SELF:NODE;
    I1,IDOF,II,INT,NN:INTEGER;
    mSECS:LONGINT;
    HMS:PACKED ARRAY [1..8] OF CHAR;
    ASYNC,CONVERG:BOOLEAN;
    NEIGH:@DA2;
    KDATA:@DA4;
    FDATA:@DA5;
    PARMS:@DA6;
    DTPTR:ADDR;
    DELTAPTR:ARRAY [1..4] OF ADDR;


BEGIN  (* PSCL$$ *)

(*  ENABLE AND RESET FLAGS  *)
FOR I IN [FLAG0,FLAG2,FLAG3] DO
  BEGIN FLGEN(I); FLGRES(I)
  END;

(*  CONSTANTS  *)
RO:=CV9512(0.0);
MAXREAL:=MAX95;
CNVRG:=CV9512(EPSILON);

(*  SOLUTION TECHNIQUE  *)
IF IO$MODE = SYNCHRONOUS THEN
  ASYNC:=FALSE
ELSE
```

```
  ASYNC:=TRUE;

(*  GET DATA AREAS  *)
NEIGH::ADDR:=DAPTR(2);
KDATA::ADDR:=DAPTR(4);
FDATA::ADDR:=DAPTR(5);
PARMS::ADDR:=DAPTR(6);

(*  GET LOGICAL SELF ID  *)
SELF:=LSELF;

(*  PRINT HEADING  *)
BAR(SYSFLAG);  (* WAIT FOR FLAGS ENABLED & RESET *)
FLGSET(FLAGO);
IF FIRST THEN
  BEGIN
  MSGLN('--- Four-Node Wing Box Problem --- V2.1 ---'); NXTLN;
  IF ASYNC THEN
    MSG('Asynchronous')
  ELSE
    MSG('Standard');
  MSGLN(' Jacobi Solution Technique')
  END;

(*  INITIALIZATION  *)

DTPTR:=LOCATION(DELTAT);
FOR I:=1 TO PARMS@.NNODE DO
  DELTAPTR[I]:=LOCATION(DELTA[I]);

WITH PARMS@ DO
  BEGIN
  NN:=NDOF*NNODE;
  II:=NN*NN;
  I1:=3*(SELF-1)+1;
  IDOF:=3*(SELF-1)+NDOF
  END;

K:=KDATA@;  (* GET WORKING COPIES OF DATA *)
F:=FDATA@;

(*  CONVERT DATA TO 9512 FORMAT *)
FOR I:=1 TO NN DO
  FOR J:=1 TO NN DO
    K[I,J]:=CV9512(K[I,J]);
WITH PARMS@ DO
  FOR I:=1 TO NCON*NNODE DO
    F[I]:=CV9512(F[I]);

(*  SEND INITIAL VALUES  *)
FOR I := 1 TO PARMS@.NDOF DO
  BEGIN
  DELTA[SELF,I]:=R0;
  DELTAT[I]:=MAXREAL
```

```
  END;
SENDALL(DELTAPTR[SELF],DSIZE);
IF ASYNC THEN  (* WAIT FOR INITIAL VALUES *)
  BAR(FLAG3);

FOR J := I1 TO IDOF DO
  BEGIN
  F[J]:=DIVD(F[J],K[J,J]);
  FOR I := 1 TO 12 DO
    IF I <> J THEN
      K[J,I]:=DIVD(K[J,I],K[J,J]);
  K[J,J]:=R0
  END;

(*  MAIN LOOP  *)
INT:=0;
TSTART(50);  (* 50 mSEC INTERVAL *)

WITH PARMS@ DO
  WHILE NOT ALL(FLAG2) DO
    BEGIN
    INT:=INT+1;
    FOR I := 1 TO NCON DO
      (* READ DISPLACEMENTS FROM NEIGHBORS *)
      RECV(NEIGH@[I],DELTAPTR[NEIGH@[I]],DSIZE);
    II := 1;
    CONVERG:=TRUE;
    FOR JJ := I1 TO IDOF DO  (* CALCULATE NEW DISPLACEMENTS *)
      BEGIN
      DSPL:=SUB(F[JJ],VDP(NN,K[JJ],DELTA::A12));
      CONVERG:=CONVERG AND
               (CMP(ABS95(SUB(DSPL,DELTAT[II])),CNVRG) < 0);
      DELTAT[II]:=DSPL;
      II := II+1
      END;
    SENDALL(DTPTR,DSIZE);
    DELTA[SELF]:=DELTAT;
    IF CONVERG THEN  (* SIGNAL LOCAL CONVERGENCE *)
      FLGSET(FLAG2)
    ELSE
      FLGRES(FLAG2);
    IF NOT ASYNC THEN
      (* STD. JACOBI MUST SYNC BEFORE TESTING ALL *)
      BAR(FLAG3)
    END;  (* MAIN LOOP *)

TSTOP;
TREAD1(HMS);
mSECS:=TREAD;

(*  REPORT RESULTS  *)
FOR I:=1 TO PARMS@.NNODE DO
  BEGIN
  BAR(FLAG3);
```

```
  IF I = SELF THEN
    BEGIN
    ENDLN(2);
    MSG('Node '); MSGI(I); MSG('     (Processor '); MSGI(PSELF);
    MSGCH(')'); NXTLN;
    MSG('---------------------------------'); ENDLN(2);
    MSG('  ');
    MSG('Iterations = '); MSGI(INT); NXTLN;
    MSG('  ');
    MSG('Elapsed time for main loop = '); MSG(HMS);
    MSG('  ('); MSGL(mSECS); MSG(' msecs)'); NXTLN;
    MSG('  '); MSG('Displacements = ');
    FOR I:=1 TO PARMS@.NDOF DO
      BEGIN
      MSGR(DELTAT[I]); MSG('  ')
      END;
    NXTLN;
    CWAIT  (* WAIT FOR CONTROLLER TO PROCESS MESSAGES *)
    END
  END

END;  (* PSCL$$ *)

BEGIN  (*$ NO OBJECT *)
END.  (* JACOBI *)
```

Link Map

```
 SDSLNK        3.5.0  81.117             11/13/83  12:28:08             PAGE    1
COMMAND LIST
NOSYMT
FORMAT COMPRESSED
LIBRARY SYS1.TIP.MINOBJ
LIBRARY SYS1.TIP.LUNOBJ
LIBRARY SYS1.TIP.OBJ
LIBRARY SYS1.FEM.PASLIB
TASK JC21
INCLUDE (N$MAIN)
INCLUDE USER1.TWC.JACOBI.JC210
INCLUDE (STK$1)
INCLUDE (HP$0)
END

CONTROL FILE = USER1.TWC.JACOBI.JC21C

LINKED OUTPUT FILE = USER1.TWC.JACOBI.JC21L

LIST FILE = SYS2.TWC.T.PRINT1

OUTPUT FORMAT = COMPRESSED

LIBRARIES

NO    ORGANIZATION    PATHNAME

1     RANDOM          SYS1.TIP.MINOBJ
2     RANDOM          SYS1.TIP.LUNOBJ
3     RANDOM          SYS1.TIP.OBJ
4     RANDOM          SYS1.FEM.PASLIB



PHASE 0, JC21      ORIGIN = 0000   LENGTH = 15EE   ENTRY=0000


MODULE     NO   ORIGIN    LENGTH     TYPE         DATE       TIME       CREATOR

N$MAIN     1     0000      00C2     INCLUDE,4    09/13/83   13:24:37    SDSMAC
$DATA      1     1116      0098
PSCL$$     2     00C2      0878     INCLUDE      11/13/83   12:19:48    DXPSCL
STK$1      3     093A      0000     INCLUDE,4    08/23/82   14:47:19    SDSMAC
$DATA      3     11AE      0440
HP$0       4     093A      0000     INCLUDE,4    08/23/82   14:48:18    SDSMAC
ABEND$     5     093A      006A     LIBRARY,1    09/10/81   19:02:38    SDSMAC
GO$SA      6     09A4      0074     LIBRARY,1    09/10/81   19:17:21    SDSMAC
CKTOP$     7     0A18      002C     LIBRARY,2    09/10/81   18:53:22    SDSMAC
```

ENT$	8	0A44	00CC	LIBRARY,2	09/10/81	18:53:33	SDSMAC
ENT$MD	9	0B10	000C	LIBRARY,3	09/10/81	18:25:21	SDSMAC
SETIN$	10	0B1C	0024	LIBRARY,3	09/10/81	18:35:01	SDSMAC
MOV$N	11	0B40	0018	LIBRARY,3	09/10/81	18:30:10	SDSMAC
SET$EQ	12	0B58	0040	LIBRARY,3	09/10/81	18:34:51	SDSMAC
PASLNK	13	0B98	0000	LIBRARY,4	10/13/83	08:06:45	SDSMAC
FLGEN	14	0B98	0010	LIBRARY,4	12/10/82	09:30:19	SDSMAC
FLGRES	15	0BA8	0010	LIBRARY,4	12/10/82	09:30:29	SDSMAC
CV9512	16	0BB8	0010	LIBRARY,4	12/10/82	09:33:25	SDSMAC
MAX95	17	0BC8	0010	LIBRARY,4	12/10/82	09:34:42	SDSMAC
IO$MODE	18	0BD8	0010	LIBRARY,4	12/10/82	09:31:58	SDSMAC
DAPTR	19	0BE8	0010	LIBRARY,4	12/10/82	09:31:08	SDSMAC
LSELF	20	0BF8	0010	LIBRARY,4	12/10/82	09:36:11	SDSMAC
BAR	21	0C08	0010	LIBRARY,4	12/10/82	09:29:38	SDSMAC
FLGSET	22	0C18	0010	LIBRARY,4	12/10/82	09:30:44	SDSMAC
FIRST	23	0C28	0010	LIBRARY,4	12/10/82	09:29:50	SDSMAC
MSGLN	24	0C38	001A	LIBRARY,4	12/10/82	09:35:26	SDSMAC
NXTLN	25	0C52	0010	LIBRARY,4	12/10/82	09:35:39	SDSMAC
MSG	26	0C62	001A	LIBRARY,4	12/10/82	09:35:07	SDSMAC
SENDALL	27	0C7C	0010	LIBRARY,4	12/10/82	09:32:34	SDSMAC
DIVD	28	0C8C	0010	LIBRARY,4	12/10/82	09:33:49	SDSMAC
TSTART	29	0C9C	0010	LIBRARY,4	12/10/82	09:36:01	SDSMAC
ALL	30	0CAC	0010	LIBRARY,4	12/10/82	09:29:03	SDSMAC
RECV	31	0CBC	0010	LIBRARY,4	12/10/82	09:31:33	SDSMAC
VDP	32	0CCC	0010	LIBRARY,4	12/10/82	09:35:04	SDSMAC
SUB	33	0CDC	0010	LIBRARY,4	12/10/82	09:35:00	SDSMAC
ABS95	34	0CEC	0010	LIBRARY,4	12/10/82	09:33:04	SDSMAC
CMP	35	0CFC	0010	LIBRARY,4	12/10/82	09:33:19	SDSMAC
TSTOP	36	0D0C	0010	LIBRARY,4	12/10/82	09:36:04	SDSMAC
TREAD1	37	0D1C	016C	LIBRARY,4	01/03/83	13:56:03	DXPSCL
TREAD	38	0E88	0010	LIBRARY,4	12/10/82	09:36:08	SDSMAC
ENDLN	39	0E98	0010	LIBRARY,4	12/10/82	09:35:36	SDSMAC
MSGI	40	0EA8	0010	LIBRARY,4	12/10/82	09:35:17	SDSMAC
PSELF	41	0EB8	0010	LIBRARY,4	12/10/82	09:36:14	SDSMAC
MSGCH	42	0EC8	0010	LIBRARY,4	12/10/82	09:35:10	SDSMAC
MSGL	43	0ED8	0010	LIBRARY,4	12/10/82	09:35:23	SDSMAC
MSGR	44	0EE8	0010	LIBRARY,4	12/10/82	09:35:29	SDSMAC
CWAIT	45	0EF8	0024	LIBRARY,4	02/15/83	09:28:23	DXPSCL
ALCPY$	46	0F1C	001E	LIBRARY,3	09/10/81	18:17:07	SDSMAC
DI$DIV	47	0F3A	00E8	LIBRARY,3	09/10/81	18:21:37	SDSMAC
DIV$	48	1022	0038	LIBRARY,3	09/10/81	18:22:05	SDSMAC
ALLOC$	49	105A	0044	LIBRARY,2	09/10/81	18:52:54	SDSMAC
GBUSY	50	109E	0010	LIBRARY,4	12/10/82	09:31:21	SDSMAC
DLY	51	10AE	0058	LIBRARY,4	02/15/83	09:24:14	DXPSCL
XTIME	52	1106	0010	LIBRARY,4	12/10/82	09:35:58	SDSMAC

D E F I N I T I O N S

NAME	VALUE	NO	NAME	VALUE	NO	NAME	VALUE	NO	NAME	VALUE	NO
$ABS95	2F4A*	13	*$ADD	2EFC*	13	$ALL	2AE2*	13	*$ANY	2AF2*	13

```
 $BAR     2B56* 13    $CMP     3054* 13    $CV951   30FA* 13   *$CV990   323C* 13
*$DABS9   2F3E* 13   *$DADD$   2F5E* 13    $DAPTR   2B7E* 13   *$DCMP    304A* 13
*$DDIVD   2F76* 13   *$DFLOI   351C* 13   *$DFLOL   3576* 13    $DIVD    2F14* 13
*$DMAX9   30C6* 13   *$DMIN9   30CC* 13   *$DMULT   2F6E* 13   *$DNEG    2F1C* 13
*$DOUBL   33F6* 13   *$DSQRT   3930* 13   *$DSUB$   2F66* 13   *$DV951   3184* 13
*$DV990   32C4* 13   *$DVDP    3B38* 13    $ENDLN   3CF2* 13    $ERRVE   0080* 13
 $FIRST   2B38* 13   *$FLGDI   2AA0* 13    $FLGEN   2A96* 13    $FLGRE   2ABC* 13
 $FLGSE   2AA8* 13   *$FLOTI   3450* 13   *$FLOTL   34A4* 13    $GBUSY   2EE8* 13
*$IFIX    3600* 13   *$IFIXD   3678* 13    $IO$MO   2ED2* 13   *$LFIX    36F6* 13
*$LFIXD   3798* 13    $LSELF   3D82* 13    $MAX95   30B2* 13   *$MIN95   30B8* 13
 $MSG     3CBC* 13    $MSGCH   3D00* 13   *$MSGDH   3D3E* 13    $MSGI    3D08* 13
*$MSGIH   3D10* 13    $MSGL    3D18* 13    $MSGLN   3CD4* 13    $MSGR    3D24* 13
*$MSGRH   3D30* 13   *$MULT    2F0C* 13    $N$HEA   F9E8* 13    $N$STK   F9E4* 13
*$NEG     2F28* 13    $NXTLN   3CF8* 13    $OBJPT   FA8E* 13    $PRPTR   FA84* 13
 $PSELF   3D8A* 13   *$QUERY   3C2C* 13   *$RDCH    3C30* 13   *$RDH     3C3A* 13
*$RDI     3C66* 13   *$RDR     3C92* 13    $RECV    2BB2* 13   *$RECV2   2BC4* 13
*$SEND    2CE4* 13   *$SEND2   2CF6* 13    $SENDA   2DF6* 13   *$SINGL   338A* 13
*$SND2A   2E04* 13   *$SQRT9   384C* 13    $STOP    008C* 13    $SUB     2F04* 13
*$SYNC    2B02* 13    $TREAD   3D72* 13    $TSTAR   3D5E* 13    $TSTOP   3D6C* 13
 $VDP     3A44* 13    $XTIME   3D4E* 13   *ABEND$   0956  5     ABND$0   097E  5
 ABND$1   0976  5     ABND$2   0948  5     ABS95    0CEE  34    ALCPY$   0F1C  46
 ALL      0CAE  30    ALLC$$   1060  49   *ALLOC$   105A  49    BAR      0C0A  21
 CKTOP$   0A18  7     CMP      0CFE  35    CUR$$    0AF4  8     CV9512   0BBA  16
 CWAIT    0EFE  45    DAPTR    0BEA  19    DI$DIV   0F3C  47    DI$MOD   0F46  47
 DIV$     1022  48    DIVD     0C8E  28    DLY      10B2  51    ENDLN    0E9A  39
*ENT$     0A54  8    *ENT$2    0A44  8     ENT$M    0A5E  8     ENT$MD   0B10  9
 ENT$S    0AE0  8     FIRST    0C2A  23    FLGEN    0B9A  14    FLGRES   0BAA  15
 FLGSET   0C1A  22    GBUSY    10A0  50    GO$SA    09A4  6     HP$BOT   15EE  4
 HP$TOP   15EE  4     IO$MOD   0BDA  18    LSELF    0BFA  20    MARG$N   093A  5
*MASK$    0B7A  12    MASK$$   0B58  12    MAX95    0BCA  17   *MOV$4    0B4E  11
*MOV$5    0B4C  11    MOV$6    0B4A  11   *MOV$7    0B48  11   *MOV$8    0B46  11
 MOV$N    0B40  11    MSG      0C64  26    MSGCH    0ECA  42    MSGI     0EAA  40
 MSGL     0EDA  43    MSGLN    0C3A  24    MSGR     0EEA  44    NXTLN    0C54  25
*PATCH$   0A78  8     PSCL$$   01F6  2     PSELF    0EBA  41    RECV     0CBE  31
*RET$2    0AFA  8     RET$M    0B0C  8     RET$S    0B0E  8     SENDAL   0C7E  27
*SET$EQ   0B7A  12    SETIN$   0B1C  10    ST$BOT   11AE  3     ST$TOP   15EE  3
 SUB      0CDE  33   *SVC$     0038  1    *T$CC     000A* 1     T$EC     000C* 1
*T$MSG    000E* 1     T$SYSM   0008* 1    *T$TIB    1116  1     T$WP     0006* 1
 TERM$$   0046  1     TREAD    0E8A  38    TREAD1   0D2A  37    TSTART   0C9E  29
 TSTOP    0D0E  36    VDP      0CCE  32    XTIME    1108  52


**** LINKING COMPLETED
```

SCI Procedure

```
JC21 (JACOBI / ASYNCHRONOUS JACOBI V2.1 -- 9/83 twc)=3,
SYNCHRONOUS I/O? = YESNO(YES),
SELECTED PROCESSORS = STRING,
REFERENCE PROCESSOR = INT(@$REFPROCESSOR)
P$SYN
Q$SYN
RESET SELMAP=DF
.IF @$ERRTST, NE, 00000
.EXIT
.ENDIF
SAC SP="&SELECTED PROCESSORS"
.IF @$ERRTST, NE, 00000
.EXIT
.ENDIF
LDPG PFAN="USER1.TWC.JACOBI.JC21L",SP="&SELECTED PROCESSORS"
.IF @$ERRTST, NE, 00000
.EXIT
.ENDIF
DEFDAD CFAN="USER1.TWC.JACOBI.DATA20.DEFDA"
.IF @$ERRTST, NE, 00000
.EXIT
.ENDIF
LDAD CFAN="USER1.TWC.JACOBI.DATA20.LOADDA"
.IF @$ERRTST, NE, 00000
.EXIT
.ENDIF
.IF "&SYNCHRONOUS I/O?", GE, "Y"
SYNCON MAXREC=6,NOITAG=1,NOLL=8,QD=2
.ELSE
ASYNCON MAXREC=6,NOITAG=1,NOLL=8
.ENDIF
.IF @$ERRTST, NE, 00000
.EXIT
.ENDIF
XFEM CHECK=Y,TRACENAB=N,REF="&REFERENCE PROCESSOR"
```

Program Output

```
--- Four-Node Wing Box Problem --- V2.1 ---

Standard Jacobi Solution Technique

Node       1      (Processor      16)
----------------------------------

  Iterations =     281
  Elapsed time for main loop = 00:00:09  (       9250 msecs)
  Displacements =  4.013423E-04  -7.199459E-03   1.311874E-02

Node       2      (Processor      17)
----------------------------------

  Iterations =     281
  Elapsed time for main loop = 00:00:09  (       9250 msecs)
  Displacements = -4.013424E-04  -7.199458E-03  -1.311875E-02

Node       3      (Processor      18)
----------------------------------

  Iterations =     281
  Elapsed time for main loop = 00:00:09  (       9250 msecs)
  Displacements = -4.005618E-04   7.200593E-03   1.263331E-02

Node       4      (Processor      19)
----------------------------------

  Iterations =     281
  Elapsed time for main loop = 00:00:09  (       9250 msecs)
  Displacements =  4.005620E-04   7.200593E-03  -1.263331E-02
```

APPENDIX B
EPROM-RESIDENT SUBROUTINES
(PASLIB V2.1-101283)

Listed below are those PASLIB routines which are stored in EPROM on each of the processors in the FEM Array. Access to these routines is via small interface subroutines from SYS1.FEM.PASLIB which are linked with user programs. The interface subroutines contain procedure and function call entry and exit code, and a BL (branch and link) instruction to transfer control to the proper address in EPROM. When the EPROM routine terminates, control is returned to the interface subroutine using the address in register 11.

Subroutine	Starting Address	Ending Address
ABS95	2F4A	2F5C
ADD	2EFC	2F02
ALL	2AE2	2B2E
ANY	2AF2	2B2E
BAR	2B56	2B7C
CMP	3054	30B0
CV9512	30FA	3182
CV990	323C	32C2
DABS95	2F3E	2F5C
DADD	2F5E	2F64
DAPTR	2B7E	2BB0
DCMP	304A	30B0
DDIVD	2F76	2F7C
DFLOTI	351C	3574
DFLOTL	3576	35FE
DIVD	2F14	2F1A
DMAX95	30C6	30E0
DMIN95	30CC	30E0
DMULT	2F6E	2F74
DNEG	2F1C	2F3C
DOUBLE	33F6	344E
DSQRT95	3930	3A42
DSUB	2F66	2F6C
DV9512	3184	323A
DV990	32C4	3388
DVDP	3B38	3C2A
ENDLN	3CF2	3CFE
FIRST	2B38	2B54
FLGDIS	2AA0	2AE0
FLGEN	2A96	2AE0
FLGRES	2ABC	2AE0
FLGSET	2AA8	2AE0
FLOATI	3450	34A2
FLOATL	34A4	351A
GBUSY	2EE8	2EFA
IFIX	3600	3676
IFIXD	3678	36F4

IO$MODE	2ED2	2EE6
LFIX	36F6	3796
LFIXD	3798	384A
LSELF	3D82	3D88
MAX95	30B2	30C4
MIN95	30B8	30C4
MSG	3CBC	3CD2
MSGCH	3D00	3D06
MSGDH	3D3E	3D4C
MSGI	3D08	3D0E
MSGIH	3D10	3D16
MSGL	3D18	3D22
MSGLN	3CD4	3CF0
MSGR	3D24	3D2E
MSGRH	3D30	3D3C
MULT	2F0C	2F12
NEG	2F28	2F3C
NXTLN	3CF8	3CFE
PSELF	3D8A	3D90
QRY	3C2C	3C2E
RDCH	3C30	3C38
RDH	3C3A	3C64
RDI	3C66	3C90
RDR	3C92	3CBA
RECV	2BB2	2CE2
RECV2	2BC4	2CE2
SEND	2CE4	2DF4
SEND2	2CF6	2DF4
SEND2ALL	2E04	2ED0
SENDALL	2DF6	2ED0
SINGLE	338A	33F4
SQRT95	384C	392E
SUB	2F04	2F0A
SYNC	2B02	2B2E
TREAD	3D7C	3D80
TSTART	3D5E	3D6A
TSTOP	3D6C	3D70
VDP	3A44	3B36
XTIME	3D4E	3D5C

APPENDIX C
SUBROUTINE REFERENCE SHEET

Text Output

```
PROCEDURE MSG(STRING:PACKED ARRAY [1..?] OF CHAR);EXTERNAL;
PROCEDURE MSGLN(STRING:PACKED ARRAY [1..?] OF CHAR);EXTERNAL;
PROCEDURE ENDLN(N:POSINT);EXTERNAL;
PROCEDURE NXTLN;EXTERNAL;
PROCEDURE MSGCH(CH:CHAR);EXTERNAL;
PROCEDURE MSGI(I:INTEGER);EXTERNAL;
PROCEDURE MSGL(I:LONGINT);EXTERNAL;
PROCEDURE MSGR(X:REAL);EXTERNAL;
PROCEDURE MSGD(X:REAL(16));EXTERNAL;
PROCEDURE MSGIH(I:INTEGER);EXTERNAL;
PROCEDURE MSGRH(X:REAL);EXTERNAL;
PROCEDURE MSGDH(X:REAL(16));EXTERNAL;
PROCEDURE CWAIT;EXTERNAL;
```

Interactive Input

```
PROCEDURE QUERY;EXTERNAL;
FUNCTION RDCH:CHAR;EXTERNAL;
FUNCTION RDH:INTEGER;EXTERNAL;
FUNCTION RDI:INTEGER;EXTERNAL;
FUNCTION RDR:REAL;EXTERNAL;
```

Data Areas

```
FUNCTION DAPTR(DA:DANUM):ADDR;EXTERNAL;
```

Flags

```
PROCEDURE FLGEN(F:FLAG);EXTERNAL;
PROCEDURE FLGDIS(F:FLAG);EXTERNAL;
PROCEDURE FLGRES(F:FLAG);EXTERNAL;
PROCEDURE FLGSET(F:FLAG);EXTERNAL;
FUNCTION ANY(F:FLAG):BOOLEAN;EXTERNAL;
FUNCTION ALL(F:FLAG):BOOLEAN;EXTERNAL;
FUNCTION SYNC(F:FLAG):BOOLEAN;EXTERNAL;
FUNCTION FIRST(F:FLAG):BOOLEAN;EXTERNAL;
PROCEDURE BAR(F:FLAG);EXTERNAL;
```

Floating-point Operations

```
FUNCTION ADD(X,Y:REAL):REAL;EXTERNAL;
FUNCTION SUB(X,Y:REAL):REAL;EXTERNAL;
FUNCTION MULT(X,Y:REAL):REAL;EXTERNAL;
FUNCTION DIVD(X,Y:REAL):REAL;EXTERNAL;
```

```
FUNCTION NEG(X:REAL):REAL;EXTERNAL;
FUNCTION ABS95(X:REAL):REAL;EXTERNAL;
FUNCTION CMP(X,Y:REAL):INTEGER;EXTERNAL;
FUNCTION DADD(X,Y:REAL(16)):REAL(16);EXTERNAL;
FUNCTION DSUB(X,Y:REAL(16)):REAL(16);EXTERNAL;
FUNCTION DMULT(X,Y:REAL(16)):REAL(16);EXTERNAL;
FUNCTION DDIVD(X,Y:REAL(16)):REAL(16);EXTERNAL;
FUNCTION DNEG(X:REAL(16)):REAL(16);EXTERNAL;
FUNCTION DABS95(X:REAL(16)):REAL(16);EXTERNAL;
FUNCTION DCMP(X,Y:REAL(16)):INTEGER;EXTERNAL;
```

Floating-point Constants

```
FUNCTION MAX95:REAL;EXTERNAL;
FUNCTION MIN95:REAL;EXTERNAL;
FUNCTION DMAX95:REAL(16);EXTERNAL;
FUNCTION DMIN95:REAL(16);EXTERNAL;
```

Floating-point Conversions

```
FUNCTION CV9512(X:REAL):REAL;EXTERNAL;
FUNCTION CV990(X:REAL):REAL;EXTERNAL;
FUNCTION FLOATI(I:INTEGER):REAL;EXTERNAL;
FUNCTION FLOATL(I:LONGINT):REAL;EXTERNAL;
FUNCTION IFIX(X:REAL):INTEGER;EXTERNAL;
FUNCTION LFIX(X:REAL):LONGINT;EXTERNAL;
FUNCTION SINGLE(X:REAL(16)):REAL;EXTERNAL;
FUNCTION DV9512(X:REAL(16)):REAL(16);EXTERNAL;
FUNCTION DV990(X:REAL(16)):REAL(16);EXTERNAL;
FUNCTION DFLOTI(I:INTEGER):REAL(16);EXTERNAL;
FUNCTION DFLOTL(L:LONGINT):REAL(16):EXTERNAL;
FUNCTION IFIXD(X:REAL(16)):INTEGER;EXTERNAL;
FUNCTION LFIXD(X:REAL(16)):LONGINT;EXTERNAL;
FUNCTION DOUBLE(X:REAL):REAL(16);EXTERNAL;
```

Mathematical Subroutines

```
FUNCTION SQRT95(X:REAL):REAL;EXTERNAL;
FUNCTION DSQRT95(X:REAL(16)):REAL(16);EXTERNAL;
FUNCTION VDP(N:POSINT;
             VAR A:ARRAY [1..?] OF REAL;
             VAR B:ARRAY [1..?] OF REAL):REAL;EXTERNAL;
FUNCTION DVDP(N:POSINT;
              VAR A:ARRAY [1..?] OF REAL(16);
              VAR B:ARRAY [1..?] OF REAL(16)):REAL(16):EXTERNAL;
FUNCTION URAN:REAL;EXTERNAL;
FUNCTION DURAN:REAL(16);EXTERNAL;
PROCEDURE RANSEED(SEED:LONGINT);EXTERNAL;
FUNCTION SINE(X:REAL):REAL;EXTERNAL;
FUNCTION DSINE(X:REAL(16)):REAL(16);EXTERNAL;
```

Sum/Maximum

*** Not implemented ***

Neighbor Communications

```
PROCEDURE SEND(N:NODE; LOC:ADDR; NWORDS:RECLEN);EXTERNAL;
PROCEDURE SEND2(N:NODE; INDEX:IDX; LOC:ADDR; NWORDS:RECLEN);
                EXTERNAL;
PROCEDURE SENDALL(LOC:ADDR; NWORDS:RECLEN);EXTERNAL;
PROCEDURE SEND2ALL(INDEX:IDX; LOC:ADDR; NWORDS:RECLEN);
                   EXTERNAL;
PROCEDURE RECV(N:NODE; LOC:ADDR; NWORDS:RECLEN);EXTERNAL;
PROCEDURE RECV2(N:NODE; INDEX:IDX; LOC:ADDR; NWORDS:RECLEN);
                EXTERNAL;
FUNCTION IO$MODE:INTEGER;EXTERNAL;
FUNCTION GBUSY:BOOLEAN;EXTERNAL;
```

Timing

```
FUNCTION XTIME:LONGINT;EXTERNAL;
PROCEDURE XTIME1(VAR HMS:PACKED ARRAY [1..?] OF CHAR);EXTERNAL;
PROCEDURE DLY(T:INTEGER);EXTERNAL;
PROCEDURE TSTART(T:POSINT);EXTERNAL;
PROCEDURE TSTOP;EXTERNAL;
FUNCTION TREAD:LONGINT;EXTERNAL;
PROCEDURE TREAD1(VAR HMS:PACKED ARRAY [1..?] OF CHAR);EXTERNAL;
```

Processor Identification

```
FUNCTION PSELF:NODE;EXTERNAL;
FUNCTION LSELF:NODE;EXTERNAL;
```

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA CR-172281		

4. Title and Subtitle	5. Report Date
PASLIB Programmer's Guide for the Finite Element Machine Revision 2.1-A	April 1984
	6. Performing Organization Code
7. Author(s)	8. Performing Organization Report No.
Thomas W. Crockett	
	10. Work Unit No.
9. Performing Organization Name and Address	11. Contract or Grant No.
Kentron International, Inc. Aerospace Technologies Division 3221 N. Armistead Ave. Hampton, VA 23666	NAS1-16000
12. Sponsoring Agency Name and Address	13. Type of Report and Period Covered
National Aeronautics and Space Administration Washington, DC 20546	Contractor Report
	14. Sponsoring Agency Code

15. Supplementary Notes

Langley Technical Monitor: Olaf O. Storaasli

16. Abstract

PASLIB is a library of Pascal-callable subroutines designed to give application programs access to the unique architectural features of the Finite Element Machine and to the software support services provided by the Nodal Exec operating system which runs on it. This report documents each of the PASLIB subroutines, and describes the procedures needed to write Pascal programs for execution on the Finite Element Machine. It also discusses considerations for obtaining optimum hardware and software performance, and gives a brief overview of debugging and performance analysis capabilities available to the programmer.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
parallel processing Finite Element Machine computer software	Unclassified - Unlimited Subject Category 62

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of Pages	22. Price
Unclassified	Unclassified	156	A08

For sale by the National Technical Information Service, Springfield, Virginia 22161