DAP FORTRAN-PLUS

These notes give a tutorial introduction to DAP Fortran-Plus, the main language used to write programs for the AMT DAP. They are written from the view-point of a programmer used to conventional Fortran but contain information sufficient for programmers used to other languages to be able to learn the language constructs that make best use of the DAP's parallel processing capabilities.

Some of the examples, diagrams and parts of the text are taken from AMT's own manuals. We acknowledge AMT's kind permission to use the material.

CONTENTS

INTRODUCTION

Any program which runs on a DAP is called a **DAP** program. Any program which runs on a DAP's host system and in conjunction with a DAP program is called a **host** program. The host program, which is entered first, controls the start of the DAP program and data is transferred between the host and the DAP using special interface routines. The host program can be written in any of the languages (or mixture of languages) supported by the host system provided that the subroutine and function calling interface is compatible with the host's Fortran implementation.

DAP Fortran-Plus is a high-level language provided to write DAP progams. Its syntax is very similar to conventional Fortran, with a number of extensions that allow you to take full advantage of the DAP's parallel processing capabilities.

The most important feature of DAP Fortrtan-Plus is the ability to manipulate in parallel all the components of one-dimensional and two-dimensional objects (called **vectors** and **matrices**). In addition, a number of indexing techniques are provided that allow parallel processing of sub-sets of vectors and matrices.

DAP Fortran-Plus does not support standard Fortran input/output. However, there are facilities to transfer data between a DAP program and its associated host program (giving access to all of the host's input/output capabilities) and a DAP program can access directly the host's filestore and its standard input/output channels.

Some of the power of DAP Fortran-Plus is illustrated (below) by comparing Fortran and Fortran-Plus solutions of a particular problem. The problem is to take a rectangular grid each node of which has a value that represents the height above sea-level of the land at that point and to form a character map of the area covered by the grid. All points below the mean height above sea-level are to be marked with a 'B', all points above the mean height above sea-level are to be marked with an 'A' and any actually at the mean height are to be marked with a blank.

The DAP Fortran-Plus solution shows a function that accepts a rectangular matrix parameter (of any dimensions) and delivers a matrix result (with the same dimensions as the parameter). Two of Fortran-Plus' many built-in functions (**SIZE** and **SUM**) are used and the technique known as **'masked assignment'** is used to build the result.

Fortran

```
      subroutine mapit(height, map, m, n)
      real height(m, n), mean, sum
      character map(m, n)
c
c  compute mean height above sea-level
c
      sum = 0.0
      do 1 j = 1, n
      do 1 i = 1, m
1     sum = sum + height(i, j)
      mean = sum / (m * n)
c
c  mark map
c
      do 2 j = 1, n
      do 2 i = 1, m
      if (height(i, j) .lt. mean) then
         map(i, j) = 'B'
      else if (height(i, j) .gt. mean) then
         map(i, j) = 'A'
      else
         map(i, j) = ' '
      endif
2     continue
      return
      end
```

Fortran-Plus

```
      function map(height)
      real height(*, *), mean
      character map(*size(height,1), *size(height,2))
c
c  compute mean height above sea-level
c
      mean = sum(height) / size(height)
c
c  mark map
c
map = ' '
map (height .lt. mean) = 'B'
map (height .gt. mean) = 'A'
return
end
```

FORTRAN-PLUS VARIABLES

In addition to the usual attributes of **Type** (INTEGER, REAL, LOGICAL or CHARACTER – there is no COMPLEX in DAP Fortran-Plus) and **Length** (REAL*8, INTEGER*2, etc), DAP Fortran-Plus variables, arrays and functions also have a **Mode** (SCALAR, VECTOR or MATRIX). The different modes are:

SCALAR

The same sort of variable / value as found in conventional Fortran – a single data item

VECTOR

A one-dimensional variable / value containing a number of data items, each of the same type and length – an extension of a conventional one-dimensional array

MATRIX

A two-dimensional variable / value containing a number of data items, each of the same type and length – an extension of a conventional two-dimensional array

The difference between conventional arrays (which DAP Fortran-Plus regards as collections of Scalars) and Vectors and Matrices is that the ELEMENTS (called **components**) OF A VECTOR OR MATRIX ARE PROCESSED IN PARALLEL.

Vectors and Matrices are stored on the DAP in a way that allows the most effective use of the parallel processing capabilities of the hardware. Hence, they are usually used in preference to Scalar Arrays (which are **not** processed in parallel).

PRECISION / RANGE OF VALUES

Data lengths (precisions) are comparable to those found on other systems and will be familiar to Fortran programmers, except there is a greater variety:

type	length	precision / range (approx.)
real	3 bytes	3 digits / 0.0, ±(5.39e-79 to 7.23e+75)
	4 bytes	6 digits / 0.0, ±(5.39760e-79 to 7.23700e+75)

	6 bytes	11 digits / 0.0, ±(5.3976053469e-79 to 7.2370055773e+75)
	7 bytes	13 digits / 0.0, ±(5.397605346934e-79 to 7.237005577332e+75)
	8 bytes	15 digits / 0.0, ±(5.39760534693402e-79 to 7.23700557733226e+75)
	default	4 bytes
integer	1 byte	-128 to +127
	2 bytes	-32,768 to +32,767
	3 bytes	-8,388,608 to +8,388,607
	4 bytes	-2,147,483,648 to +2,147,483,647
	5 bytes	-549,755,813,888 to +549,755,813,887
	6 bytes	-140,737,488,355,328 to +140,737,488,355,327
	7 bytes	-36,028,797,018,963,968 to +36,028,797,018,963,967
	8 bytes	-9,223,372,036,854,775,808 to +9,223,373,036,854,775,807
	default	4 bytes
character	1 byte	ASCII codes and collating sequence
logical	1 bit	.FALSE. (represented by 0), .TRUE. (represented by 1)

DECLARING VARIABLES

Examples of DAP Fortran-Plus Declaration Statements are:

```
REAL RS1, RS2, RSA(2, 100)
CHARACTER CS
INTEGER*1 IV(*50), IVA(*30, 10)
LOGICAL LVA(*55, 5, 20), LM(*100, *80)
REAL*6 RMA(*60, *60, 3)
```

which declare default precision REAL (ie REAL*4) scalar variables RS1 and RS2 and two-dimensional scalar array RSA, CHARACTER scalar variable CS, INTEGER*1 vector IV and set of 10 INTEGER*1 vectors IVA, two-dimensional set of 100 LOGICAL vectors LVA, LOGICAL matrix LM and and set of 3 REAL*6 matrices RMA.

Notice how * in the dimensions of an object define it to be a vector or matrix. These 'parallel' dimensions may be followed by normal Fortran dimensions, to declare sets of vectors or matrices. Earlier versions of DAP Fortran-Plus constrained the parallel dimensions to match the size of the

DAP hardware – 32 on DAP 500 systems and 64 on DAP 600 systems – and declarations of vectors, vector sets, matrices and matrix sets looked like:

```
INTEGER*1 IV(), IVA( , 10)
LOGICAL LVA( , 5, 20), LM( , )
REAL*6 RMA( , , 3)
```

Such declarations are still valid but their usage will probably decline, except in specialised cases.

The Fortran **DIMENSION** statement is also available to declare multi-dimensioned objects:

```
DIMENSION X(20, 30), Y(*500), Z(*20, *1000, 5)
```

with the * in the dimensions defining an object to be a vector or matrix. The lower bounds of dimensions are always 1 in Fortran-Plus.

The * used to signify parallel dimension(s) has given rise to the informal name Fortran-Star to denote the revision of Fortran-Plus that removed the constraints on parallel dimensions.

PARALLEL DIMENSIONS ALWAYS COME FIRST IN DECLARATIONS.

DEFAULT TYPES AND IMPLICIT STATEMENT

The usual Fortran default types apply. A variable, array or function whose type has not been defined in a declaration statement has type / length REAL*4 (if the first letter of the object's name is in the range A-H, O-Z) or INTEGER*4 (if the first letter is in the range I-N). These defaults may be changed by **IMPLICIT** statements, of the form:

IMPLICIT *type* **length* (*letters*)

where:

- *type* may be CHARACTER, INTEGER, LOGICAL or REAL
- *length* may be in the range 1 to 8 for INTEGER and the range 3 to 8 for REAL
- *letters* may be a single letter, a list of letters, a range of letters, or a combination

For example:

```
IMPLICIT LOGICAL (A-C, L, X-Z)
```

indicates that an object whose name starts with one of the letters A, B, C, L, X, Y or Z should be treated as LOGICAL unless over-ridden by a type declaration statement.

There is no:

```
IMPLICIT NONE
```

statement so some statement such as:

```
IMPLICIT CHARACTER (A-Z)
```

needs to be used to assist detection of the use of undeclared objects.

VECTORS AND MATRICES WITH SUBROUTINES AND FUNCTIONS

Just like other objects (scalars, scalar arrays, etc), vectors and matrices may be passed as parameters to subroutines and functions. If the size of the **parallel** dimension(s) is not known at compile time, a special form of declaration statement may be used. For example:

```
SUBROUTINE SUB (A, B)
REAL A(*), B(*, *)
```

introduces the definition of a subroutine which will accept a vector of any size as its first parameter and a matrix of any shape as its second parameter. This technique of using 'assumed' dimensions, taken from the dimensions of the actual parameters, is not allowed for non-parallel dimensions.

Sometimes, it will be necessary to use in a piece of source code the actual dimension(s) of a dimensioned object – the built-in function **SIZE** can be used for this. Its parameters are:

- the object details of whose 'size' is required
- the dimension (in the range 1 to 7) whose size is required. This parameter is optional – if it is omitted, SIZE will return the total number of elements of the object. If the object does not have as many dimensions as the one whose size is requested (eg SIZE(A, 2) for A above) the value zero is returned.

Within a subroutine or function, parallel dimensions of local vectors and matrices may be set to values passed via parameters or through COMMON blocks. This gives the ability to declare local vectors and matrices with dimensions decided at run-time. As an additional special case, parallel dimensions may be set to values returned by the SIZE function. If this special case is combined with the ability of user-written functions to return vector and matrix results, it becomes easy to declare such functions without the need for parameters or values passed through COMMON to give the dimensions of matrix and vector parameters and result. For example, the initial statements of a 'matrix multiply' function would be:

```
FUNCTION MAT_MULT (A , B)
REAL A(*, *), B(*, *)
REAL MAT_MULT(*SIZE(A,1), *SIZE(B,2))
```

A user-written function must be defined as **EXTERNAL** in the routine that references it. For example, MAT_MULT (above) would need to be declared:

```
EXTERNAL FUNCTION MAT_MULT
REAL MAT_MULT(*, *)
```

before it could be called. The assumed dimensions (*, *) indicate that the function will return a matrix result but that its actual shape will not be determined until the function returns its result.

CONSTANTS AND PARAMETER STATEMENT

In addition to the normal ways of writing constants, there is an occasional need to specify constants of lengths other than the default (INTEGER*4 and REAL*4). This is achieved using a length specifier. For example:

```
100 (*1)
```

represents the INTEGER*1 constant 100; similarly:

```
1.2345678 (*5)
```

represents a REAL*5 constant.

Character constants may take either the Hollerith form:

*n*H*c...c*

where *n* must be in the range 1 to 512 and *c...c* is a sequence of characters, or the 'literal form':

'*c...c*'

where *c...c* is a sequence of up to 512 characters with repeated ' used to represent '. Character constants of more than one character may be used only for data initialisation within type or DATA statements.

Hexadecimal constants may be used only for data initialisation within type or DATA statements and take the form:

#*f..f*

where *f..f* is a sequence of up to 1024 hexadecimal digits. Hexadecimal constants are most commonly used for the initialisation of logical vectors and matrices.

The **PARAMETER** statement may be used to give a name to a constant expression. For example:

```
PARAMETER (IEDGE=512)
```

defines a compile-time integer constant IEDGE with value 512.

INITIALISING VARIABLES AND DATA STATEMENT

The usual Fortran ways of initialising variables in declaration and **DATA** statements are available:

```
REAL RS1/7.5/, RS2
CHARACTER CSV(*30)
DATA CSV/1H&, 'ZX', 20*#40/
PARAMETER (MINUS2=-2)
INTEGER*1 IV(*50)/1, 2, 3, 4, 10*5, 36*MINUS2/
LOGICAL LVA(*25)
DATA LVA/#A070A/
```

When a list of character constants does not fill a character variable or dimensioned object, the list is extended to the right with spaces. Hexadecimal constants may be mixed with the character constants in such a list but must consist of an even number of hexadecimal digits.

When a hexadecimal constant used to initialise a dimensioned logical object is too long, the constant is truncated from the right. If the hexadecimal constant does not fill the object, the constant is padded to the right with hexadecimal zeros.

Conversely, when a hexadecimal constant is used to initialise a logical scalar, or integer or real scalar, element or component and the constant too long, it is truncated from the left. If the constant does not fill the object that it is initialising, it is padded to the left with hexadecimal zeros.

Objects held in **COMMON** blocks may be initialised in DATA or type declaration statements in any program unit in which the named COMMON block is defined. However, care must be taken to ensure that the same parts of a COMMON block are not initialised in more than one program unit. **BLOCK DATA** subprograms may also be used for initialising data in COMMON areas – an **EXTERNAL** statement referencing the BLOCK DATA subprogram is required, to ensure that the subprogram is loaded and the data initialisation takes place.

ASSIGNMENT AND ARITHMETIC

SIMPLE ASSIGNMENT

The normal Fortran rules apply – the left-hand side and right-hand side of an assignment statement should conform in type and length. As a general rule, the two sides should have the same mode (SCALAR, VECTOR, MATRIX). Finally, if the two sides of the assignment are vectors or matrices, they must have the same shape (ie their parallel dimensions must match). In practice, most of the restrictions are not enforced, because the compiler is able to make sensible decisions about what the meaning of particular assignment statements.

THE COMPONENTS OF A VECTOR OR MATRIX ARE ASSIGNED IN PARALLEL.

Complete scalar arrays, sets of vectors and sets of matrices may not be assigned in a single assignment statement. The normal Fortran rules apply – elements of a scalar array must be assigned individually and vectors and matrices from vector and matrix sets must be assigned separately.

Length Compatibility

DAP Fortran-Plus requires that the length of the left-hand side and the right-hand side of an assignment statement conform. If the lengths are not the same, the right-hand side is 'lengthened' / 'shortened' to match the left-hand side.

The built-in function **LENGTH** may explicitly be used to modify the length of an expression. LENGTH's parameters are:

- an integer or real object of any length and mode (SCALAR, VECTOR or MATRIX)
- an integer constant – in the range 1 to 8 if the first parameter has an integer type or in the range 3 to 8 if the first parameter has a real type

LENGTH is an example of an **intrinsic componental function**. These are functions which deliver a result that has the same mode as their main argument (SCALAR, VECTOR or MATRIX). They will not, however, accept parameters that are scalar arrays or vector or matrix sets.

The other intrinsic componental functions are: **ABS, ATAN, COS, EXP, FIX, FLOAT, LOG, SIN** and **SQRT**. Apart from special uses of FIX and FLOAT, all these other functions return

values with the same length as their parameter.

COMPONENTAL FUNCTIONS ACT IN PARALLEL ON THE INDIVIDUAL COMPONENTS OF THEIR PARAMETER.

Type Compatibility

DAP Fortran-Plus requires that the type of the left-hand side and the right-hand side of an assignment statement conform:

Integer	=	Integer
Real	=	Real
Logical	=	Logical
Character	=	Character

A real value is 'fix'ed before being assigned to an integer variable and an integer value is 'float'ed before being assigned to a real variable.

The normal Fortran functions **FIX** and **FLOAT** are available for explicit change of type and have the effect:

FIX returns its real argument converted (by truncation towards zero) to an integer value; the length of the result is the same as the length of the parameter. The parameter may also be of LOGICAL type – the INTEGER*4 value 1 is returned for .TRUE. and 0 for .FALSE.

FLOAT returns its integer argument converted to a real value; the length of the result is the same as the length of the parameter (which cannot be *1 or *2). The parameter may also be of LOGICAL type – the REAL*4 value 1.0 is returned for .TRUE. and 0.0 for .FALSE.

Mode and Shape Compatibility

DAP Fortran-Plus requires that the mode of the left-hand side and the right-hand side of an assignment statement conform:

scalar	=	scalar	
scalar	=	vector	INVALID
scalar	=	matrix	INVALID
vector	=	scalar	scalar is replicated
vector	=	vector	dimensions must match
vector	=	matrix	INVALID
matrix	=	scalar	scalar is replicated
matrix	=	vector	INVALID
matrix	=	matrix	dimensions must match

A scalar will be replicated to build a vector or matrix of the required dimension(s).

The built-in functions **VEC** and **MAT** are available for explicit building of vectors and matrices each of whose component elements has the same value. The parameters to VEC are:

- scalar value of any type and length
- INTEGER*4 scalar value – the dimension of the vector to be built

and MAT's parameters are:

- scalar value of any type and length
- INTEGER*4 scalar value – the number of rows in the matrix to be built
- INTEGER*4 scalar value – the number of columns in the matrix to be built

The type and length of the vectors and matrices returned by VEC and MAT match the type and length of their main (ie first) parameter. They are used mainly in calls of subroutines and functions, to provide vector and matrix parameters built from scalar values.

A vector cannot automatically be replicated and assigned to a matrix because a decision has to be made about how the matrix is to be built (all rows the same? all columns the same?). The functions **MATC** (and **MATR**) are available for explicit building of matrices each of whose columns (rows) are identical. Their parameters are:

- vector value of any type and length
- INTEGER*4 scalar value – the number of columns (rows) in the matrix to be built

Taking the declarations:

```
INTEGER IV(*4)/3, 1, 2, 5/
IMPLICIT LOGICAL (F, T)
PARAMETER (F=.FALSE.)
PARAMETER (T=.TRUE.)
LOGICAL LV1(*4)/T, F, F, T/
```

then:

```
                          T F F T
                          T F F T
MATR(LV1, 5) will give    T F F T
                          T F F T
                          T F F T
```

and:

```
                          3 3 3
MATC(IV, 3) will give     1 1 1
                          2 2 2
                          5 5 5
```

EXPRESSIONS

DAP Fortran-Plus uses the same syntax for expressions as Standard Fortran and there are the usual arithmetic operators (+, -, *, /, **) and relational operators (.LT., .LE., .EQ., .GE., .GT., .NE.).

The logical operators are .NOT., .AND., .OR., .LEQ., .NAND., .NOR. and .LNEQ. The operator .NOR. (.NAND.) gives the logical converse of .OR. (.AND.) and the operator .LEQ. (.LNEQ.) gives the logical equivalence (non-equivalence, ie 'exclusive-OR') of its operands.

OPERATIONS ON VECTORS OR MATRICES ARE PERFORMED IN PARALLEL.

The same sorts of conformance required for assignments apply to expressions also. The table below illustrates the mode of the result of A°B, where ° is any of the binary operators:

A	B	Result	
scalar	scalar	scalar	
scalar	vector	vector	scalar is replicated
scalar	matrix	matrix	scalar is replicated
vector	scalar	vector	scalar is replicated
vector	vector	vector	dimensions must match
vector	matrix	INVALID	
matrix	scalar	matrix	scalar is replicated
matrix	vector	INVALID	
matrix	matrix	matrix	dimensions must match

Taking the declarations:

```
INTEGER IV(*4), IM(*5, *4)
LOGICAL LV1(*4), LV2(*4)
```

and the values:

```
IV    3  1  2  5
LV1   T  F  F  T
LV2   T  F  T  F

      4  1  8  7
      6  3  5  10
IM    2  11 9  3
      8  1  4  2
      3  5  8  7
```

then:

```
IV + 1
```

expands to:

```
IV + (1, 1, 1, 1)
```

which is then evaluated (components in parallel) to give the vector result (4, 2, 3, 6).

The logical expression:

LV1 .OR. LV2

is evaluated in parallel to give the vector result (T, F, T, T). Finally, the relational expression:

IM .LT. 5

expands to:

```
4  1  8  7        5  5  5  5
6  3  5  10       5  5  5  5
2  11 9  3  .LT.  5  5  5  5
8  1  4  2        5  5  5  5
3  5  8  7        5  5  5  5
```

giving the matrix result:

```
T  T  F  F
F  T  F  F
T  F  F  T
F  T  T  T
T  F  F  F
```

SUMMING FUNCTIONS

There are three important built-in functions (SUM, SUMC and SUMR) that total the components of their parameter in particular ways.

The function SUM takes a single vector or matrix parameter of logical values, or integer or real values of any length and returns as a scalar value the sum of all the components of the parameter. If the parameter is real, the result is a real scalar of the same length. If the parameter is logical, the result is an INTEGER*4 scalar (.TRUE. is treated as 1, .FALSE. is treated as 0 in the summation). If the parameter is integer, the result is an INTEGER*4 scalar for parameters of length *1 to *4 and is an INTEGER*8 scalar for parameters of length *5 to *8.

Taking the declarations:

```
INTEGER IV(*4), IM(*5, *4)
LOGICAL LV1(*4)
```

and the values:

```
IV    3   1   2   5
LV1   T   F   F   T

      4   1   8   7
      6   3   5   10
IM    2   11  9   3
      8   1   4   2
      3   5   8   7
```

then:

SUM (IM) will give 107
SUM (IV) will give 11
SUM (LV1) will give 2

The function **SUMC** takes a single matrix parameter of logical values, or integer or real values of any length and returns a (column) vector value with as many components as there are rows in the parameter. Each component of the resulting vector contains the sum of all the components of the corresponding row of the parameter. If the parameter is real, the result is a real vector of the same length. If the parameter is logical, the result is an INTEGER*4 vector (.TRUE. is treated as 1, .FALSE. is treated as 0 in the summation). If the parameter is integer, the result is an INTEGER*4 vector for parameters of length *1 to *4 and is an INTEGER*8 vector for parameters of length *5 to *8.

The function **SUMR** behaves in fashion similar to SUMC but returns a (row) vector with as many components as there are columns in the parameter.

Taking the declaration:

```
INTEGER IM(*5, *4)
```

and the matrix value:

```
        4   1   8   7
        6   3   5   10
IM      2   11  9   3
        8   1   4   2
        3   5   8   7
```

then:

SUMC (IM) will give (20, 24, 25, 15, 23)
SUMR (IM) will give (23, 21, 34, 29)

MIXED TYPES, LENGTHS AND MODES

The following declarations are assumed in the examples below:

```
INTEGER IS1, IV1(*30), IM1(*30, *50)
INTEGER*2 I2V1(*30)
INTEGER*3 I3V1(*30)
REAL RS1, RV1(*30), RM1(*30, *50)
REAL*3 R3S1, R3S2, R3V1(*30)
```

To evaluate:

```
IV1=SUMC(IS1*RM1)
```

- IS1 is 'float'ed and replicated to form a (30 row, 50 column) matrix
- the expression is evaluated in REAL*4 precision
- the resulting matrix is summed into a (30 component) column vector
- the column vector is truncated to INTEGER*4 and assigned to IV1.

To evaluate:

```
RM1=RS1*(R3S1 + R3S2)
```

- the addition is carried out in REAL*3 precision
- the result of the addition is 'length'ened to REAL*4 precison
- the multiplication is performed in REAL*4 precision
- the resulting scalar is replicated to a (30 row, 50 column) matrix and assigned to RM1

To evaluate:

I3V1=(I2V1*RV1)/R3V1

- I2V1 is converted to 4 bytes and 'float'ed
- the multiplication is carried out in REAL*4 precision
- R3V1 is converted to 4 bytes and the division is then carried out in 4 byte precision
- the result is truncated to INTEGER*4 and shortened to INTEGER*3.

Typically, nothing like as much mixing of types, lengths and modes as shown in these examples is used in normal programs. It is, however, quite normal to write statements such as:

RM1 = IM1 * 0.5

SIMPLE INDEXING TECHNIQUES

The most important feature of DAP Fortran-Plus is the ability to manipulate whole one- and two-dimensional data structures (VECTOR and MATRIX) in parallel. An important companion feature is the wealth of facilities for selecting parts of these data structures, for use in expressions and for updating by assignment statements.

This part of the course reviews the indexing techniques that a Fortran programmer would expect to find and introduces some simple extensions. Further techniques will be introduced later.

LEFT-HAND SIDE AND RIGHT-HAND SIDE INDEXING

The terms left-hand side indexing and right-hand side indexing are used to distinguish between the positions to the left and to the right of an = in an assignment statement. On the right-hand side, a value (scalar, vector or matrix) is selected for use in a expression; the term **selecting** is used in these notes to signify right-hand side indexing. On the left-hand side, a destination (scalar, vector or matrix variable) is chosen to be updated. DAP Fortran-Plus allows logical **masks** to control which parts of a vector or matrix are updated and the term **masked assignment** is often used; the term **masking** is used in these notes to signify left-hand side indexing of parts of vectors and matrices.

There are subtle but important differences between the two contexts that will be explained later.

Selecting (RHS)

There are three cases to consider:

- Selection from scalar arrays
- Selection from vectors (and sets of vectors)
- Selection from matrices (and sets of matrices)

Selection from scalar arrays

Selection from a scalar array is exactly the same as in Standard Fortran and selects a single scalar value. For example:

```
REAL S1, S2, SA(10), SB(10, 20)
    ...  ...
S1 = SA(5)
S2 = SB(8, 19)
```

have the effect expected.

You cannot select the scalars from a scalar array in parallel, each individual scalar must be processed separately.

Selection from vectors and arrays (sets) of vectors

Selection from a vector has the same effect as selection from an ordinary Fortran one-dimensional array and selects a single scalar value. For example:

```
REAL S, V(*200)
    ...  ...
S = V(17)
```

will have the effect expected.

Selection of (all the components of) a vector for processing has already been described. For example:

```
INTEGER IV1(*100), IV2(*100), IV3(*100)
    ...  ...
IV3 = IV1 + IV2
```

adds the two vectors IV1 and IV2 (component by component, in parallel) and assigns the result to IV3 (in parallel).

Selection of a vector from a set of vectors is a simple extension of the selection of a scalar from a scalar array:

```
REAL V(*200), VA(*200, 5), VB(*200, 4, 3)
    ...  ...
V = VA( , 3) - VB( , 1, 2)
```

You can process a complete vector in parallel but you cannot process all the vectors from a set of vectors in parallel, each individual vector must be processed separately.

Selection from matrices and arrays (sets) of matrices

Simple selection from a matrix has the same effect as selection from an ordinary Fortran two-dimensional array and selects a single scalar value. For example:

```
REAL S, M(*35, *25)
    ...   ...
S = M(27, 11)
```

will have the effect expected.

Selection of a row vector from a matrix is written:

```
REAL VR(*25), M(*35, *25)
    ...   ...
VR = M(17, )
```

which selects row 17 from M and assigns it to VR. The trailing ',' in '(17,)' is very important – as will be explained later.

Selection of a column vector from a matrix is written:

```
REAL VC(*35), M(*35, *25)
    ...   ...
VC = M( , 5)
```

which selects column 5 from M and assigns it to VC.

Selection of (all the components of) a matrix for processing has already been described. For example:

```
INTEGER IM1(*100, *50), IM2(*100, *50), IM3(*100, *50)
    ...   ...
IM3 = IM1 * IM2
```

multiplies the two matrices IM1 and IM2 (component by component, in parallel) and assigns the result to IM3 (in parallel).

Selection of a matrix from a set of matrices is a simple extension of the selection of a scalar from a scalar array:

```
REAL M(*25, *55), MA( *25, *55, 7)
    ...   ...
M = MA( , , 6)
```

You can process a complete matrix in parallel but you cannot process all the matrices from a set of matrices in parallel, each individual matrix must be processed separately.

Assignment and Masking (LHS)

As for selecting on the right-hand side, there are three cases to consider:

- assignment to scalar arrays
- assignment to vectors (and sets of vectors)
- assignment to matrices (and sets of matrices)

Assignment to scalar arrays

Assignment to a scalar array is exactly the same as in ordinary Fortran and updates a single element. For example:

```
REAL S, SA(10), SB(10, 20)
    ...   ...
SA(5) = S
SB(8, 19) = S
```

have the effect expected.

You cannot assign to all the elements of a scalar array in parallel, each individual element must be assigned separately.

Assignment to vectors and arrays (sets) of vectors

Assignment to a component of a vector has the same effect as assignment to an ordinary Fortran one-dimensional array. For example:

```
REAL S, V(*200)
    ...  ...
V(17) = S
```

will have the effect expected. However, this is a simplified view – more will be explained later.

Assignment (in parallel) to (all the components of) a vector has been described already. DAP Fortran-Plus also allows a logical vector value to control the assignment to a vector:

```
INTEGER IV1(*100), IV2(*100), IV3(*100)
REAL M1(*35, *100)
    ...  ...
IV2(IV3 .LT. 0) = IV1 + M1(17, )
```

The logical vector controlling the assignment can be a completely general expression. What is important is that it **must** provide a logical vector that matches the dimension of the vector that is being updated. Taking as a simple example the declarations:

```
INTEGER IV1(*4), IV2(*4), IV3(*4)
```

with initial values:

```
IV1 = (3, 2, 1, 5)
IV2 = (5, 4, 3, 2)
IV3 = (9, 8, 7, 6)
```

then after the assignment:

```
IV2(IV1 .GT. 2) = IV3
```

IV2 will have the value (9, 4, 3, 6). This is an example of **masked assignment.**

Assignment to a vector in a set of vectors is a simple extension of the assignment to an element in a

scalar array:

```
REAL V(*200), VA(*200, 5), VB(*200, 4, 3)
    ...   ...
VA( , 3) = V
VB( , 1, 2) = V
```

You can assign to a complete vector in parallel but you cannot assign to all the vectors of a vector set in parallel, each individual vector must be assigned separately.

Assignment to matrices and arrays (sets) of matrices

Assignment to a component of a matrix has the same effect as assignment to an ordinary Fortran two-dimensional array. For example:

```
REAL S, M(*35, *25)
    ...   ...
M(27, 11) = S
```

has the effect expected. However, this is a simplified view – more will be explained later.

Assignment of a vector to a row of a matrix is written:

```
REAL VR(*25), M(*35, *25)
    ...   ...
M(17, ) = VR
```

which selects VR and assigns it to row 17 of M. The trailing ',' in '(17,)' is very important – as will be explained later.

Assignment of a vector to a column of a matrix is written:

```
REAL MSQ(*35, *35)
    ...   ...
MSQ( , 5) = MSQ(17, )
```

which selects row 17 of the (square) matrix MSQ and assigns it to column 5 of MSQ.

Assignment (in parallel) to (all the components of) a matrix has been described already. DAP Fortran-Plus also allows a logical matrix value to control the assignment to a matrix:

```
REAL M1(*35, *100)
    ...   ...
M1(M1 .LT. 7.0) = 7.0
```

The logical matrix masking the assignment can be a completely general expression. What is important is that it **must** provide a logical matrix that matches the dimensions of the matrix that is being updated.

Assignment to a matrix in a set of matrices is a simple extension of the assignment to an element of a scalar array:

```
REAL M(*25, *55), MB(*25, *55, 2, 5)
    ...   ...
MB( , , 2, 4) = M
```

You can assign to a complete matrix in parallel but you cannot assign to all the matrices in a matrix set in parallel, each individual matrix must be assigned separately.

REDUCED RANK INDEXING AND LONG VECTORS

A special form of index, consisting of a single integer scalar expression, may be used with a multi-dimensioned object of **any mode** (scalar array, vector, set of vectors, matrix, set of matrices) and is called a reduced rank index. It **always** selects a single element / component, using the natural Fortran ordering of the left-most subscript varying the most rapidly. For example, if MA is a matrix with dimensions (*32, *64, 5) then the ordering is:

MA(1, 1, 1), MA(2, 1, 1), ... MA(32, 1, 1), MA(1, 2, 1), MA(2, 2, 1), ... MA(32, 64, 1), MA(1, 1, 2), ... MA(32, 64, 5)

and MA(33) is the same as MA(1, 2, 1).

In right-hand side indexing positions, a reduced rank index selects a scalar. When used in left-hand side indexing positions, it requires that the value to be assigned to the single element / component is a scalar.

This form of indexing means that the trailing ',' in:

```
REAL VR(*25), M(*35, *25)
     ...   ...
VR = M(17, )
```

is very important. Without the trailing ',':

```
REAL VR(*25), M(*35, *25)
     ...   ...
VR = M(17)
```

selects the scalar value M(17, 1) and assigns its value to **each** component of VR. There is similar importance attached to the trailing ',' in:

```
REAL VR(*25), M(*35, *25)
     ...   ...
M(17, ) = VR
```

However:

```
REAL VR(*25), M(*35, *25)
     ...   ...
M(17) = VR
```

would not compile (because a scalar is required on the right-hand side of the =) and it is only in cases such as:

```
REAL S, M(*35, *25)
     ...   ...
M(17, ) = S
```

that there is danger if the ',' is forgotten.

Some built-in functions that will be described later treat a matrix whose dimensions match the size of the DAP's array of processors (32*32 for DAP 500, 64*64 for DAP 600) as a special case – a **long vector** whose elements are ordered in the same fashion as for reduced rank indexing.

BUILT-IN FUNCTIONS

DAP Fortran-Plus provides a large number of built-in functions. Some have been described already. In this section of the notes, further very important groups of built-in functions are introduced.

FUNCTIONS THAT EXTRACT THE MAXIMUM / MINIMUM VALUE FROM A VECTOR OR MATRIX

Simple selection of a scalar from a vector or matrix has been introduced already. Sometimes there is a requirement to obtain the maximum (or minimum) value of all the components in a vector or matrix. There are two built-in functions (**MAXV, MINV**) to do this:

MAXV takes a vector or matrix parameter with type integer or real and any length. The function returns a scalar value of the same length and type as its parameter and which is equal to the maximum value of all the components of its parameter

MINV takes a vector or matrix parameter with type integer or real and any length. The function returns a scalar value of the same length and type as its parameter and which is equal to the minimum value of all the components of its parameter

Taking the declarations:

```
INTEGER IV(*4), IM(*5, *4)
```

and the values:

```
IV   3  1  2  5

     4  1  8  7
     6  3  5  10
IM   2  11 9  3
     8  11 4  2
     3  5  8  7
```

then:

MAXV(IM) will give 11
MINV(IV) will give 1

Note that both MAXV and MINV have a second (optional) parameter. It is a logical mask that is used to select the components of the main parameter that will be considered by the function. If this second parameter is present, at least one of its components must be .TRUE.. If this second parameter is omitted, all the components of the main parameter are considered.

FUNCTIONS THAT OBTAIN THE POSITION(S) OF THE MAXIMUM / MINIMUM VALUE(S) OF A VECTOR OR MATRIX

The built-in function **MAXP (MINP)** finds the position(s) of the maximum (minimum) value of all the components in a vector or matrix. The descriptions of MAXP and MINP are:

MAXP takes a vector or matrix parameter with type integer or real and any length. The function returns a logical value of the same mode and dimensions as its parameter. The result returned has components set .TRUE. in the positions corresponding to the maximum value of all the components of its parameter and .FALSE. elsewhere

MINP takes a vector or matrix parameter with type integer or real and any length. The function returns a logical value of the same mode and dimensions as its parameter. The result returned has components set .TRUE. in the positions corresponding to the minimum value of all the components of its parameter and .FALSE. elsewhere

Taking the declarations and the values of IV and IM (above):

MAXP(IM) will give

```
F F F F
F F F F
F T F F
F T F F
F F F F
```

and:

MINP(IV) will give F T F F

Note that both MAXP and MINP have a second (optional) parameter. It is a logical mask that is used to select the components of the main parameter that will be considered by the function. If this second parameter is present, at least one of its components must be .TRUE.. If this second parameter is omitted, all the components of the main parameter are considered.

FUNCTIONS THAT RETURN LOGICAL PATTERNS (MASKS)

Logical patterns play a very important part in writing programs to run on the DAP, especially because of their use in masked assignment statements. DAP Fortran-Plus provides a large number of built-in functions that return logical vector and matrix patterns – some of the most important ones are described here.

The function **FRST** takes a logical vector and returns a logical vector of the same dimension and that has one component set .TRUE., corresponding to the first .TRUE. component in its parameter. In addition, FRST will acccept a logical matrix parameter of any shape, which it treats as a long vector, and return a logical matrix. For example, taking a logical vector LV and a logical matrix LM with values:

```
LV  F  T  T  F

    F  F  T
    F  T  F
LM  F  T  T
    F  T  F
    F  F  F
```

then:

```
FRST(LV) will give  F  T  F  F

                    F  F  F
                    F  T  F
FRST(LM) will give  F  F  F
                    F  F  F
                    F  F  F
```

Building alternating patterns

The function **ALT** builds a logical vector value consisting of an alternating pattern of .FALSE.s followed by .TRUE.s. Its parameters are:

- **i** an INTEGER*4 scalar value
- **s** an INTEGER*4 scalar value -- the dimension of the vector

The logical vector returned has **s** components of which the first **i mod s** are .FALSE. followed by **i mod s** .TRUE. components and so, until all the components have a value. For example:

ALT(3, 8) will give F F F T T T F F

If **i mod s** is zero, a vector with all components set .FALSE. will be returned.

The function **ALTC** builds a logical matrix value consisting of an alternating pattern of .FALSE. columns followed by .TRUE. columns. Its parameters are:

- **i** an INTEGER*4 scalar value
- **r** an INTEGER*4 scalar value – number of rows in the matrix
- **c** an INTEGER*4 scalar value – number of columns in the matrix

The logical matrix returned has dimensions **r** by **c**. Its first **i mod c** columns will be .FALSE. followed by **i mod c** .TRUE. columns and so, until all the components have a value. For example:

```
                      F F F T T
ALTC(3, 3, 5) will give F F F T T
                      F F F T T
```

If **i mod c** is zero, a matrix with all components set .FALSE. will be returned.

The companion function **ALTR** builds a logical matrix value consisting of an alternating pattern of .FALSE. rows followed by .TRUE. rows. Its parameters are:

- **i** an INTEGER*4 scalar value
- **r** an INTEGER*4 scalar value – number of rows in the matrix
- **c** an INTEGER*4 scalar value – number of columns in the matrix

The logical matrix returned has dimensions **r** by **c**. Its first **i mod r** rows will be .FALSE. followed by **i mod r** .TRUE. rows and so, until all the components have a value. For example:

```
                        F F F F F F
ALTR(5, 4, 6) will give T T T T T T
                        F F F F F F
                        T T T T T T
```

If **i mod r** is zero, a matrix with all components set .FALSE. will be returned.

Building patterns with selected component(s), row(s) or column(s) set .TRUE.

The built-in function **EL** builds a logical vector with one .TRUE. component. Its parameters are:

- an INTEGER*4 scalar value – the number of the component that will be .TRUE.
- an INTEGER*4 scalar value – the dimension of the vector

For example:

EL(5, 7) will give F F F F T F F

The built-in function **ELS** builds a logical vector with a sequence of .TRUE. components. Its parameters are:

- an INTEGER*4 scalar value – the number of the first component that will be .TRUE.
- an INTEGER*4 scalar value – the number of the last component that will be .TRUE.
- an INTEGER*4 scalar value – the dimension of the vector

For example:

ELS(3, 5, 7) will give F F T T T F F

There are also functions (**COL**, **COLS**, **ROW** and **ROWS**) that build logical matrix patterns. For example:

```
                                                T T T F
ROWS(1, 2, 3, 4) .AND. COLS(1, 3, 3, 4) will give T T T F
                                                F F F F
```

COL (ROW) accepts a first parameter as for EL, to specify column (row) number that is to be set .TRUE., and its second and third parameters specify the dimensions of the matrix. Additionally, COL (ROW) will accept an integer vector as its first parameter and its second parameter specifies the number of columns (rows). For example, if the 5-component INTEGER*4 vector IV has the value (4, 3, 2, 1, 5) then:

```
                    F F F T
                    F F T F
COL(IV, 4) will give F T F F
                    T F F F
                    F F F F
```

SIMPLE ENQUIRIES OF LOGICAL PATTERNS

The function **ALL (ANY)** takes a logical vector or matrix and returns the logical scalar value .TRUE. if all (any) of the components of its parameter are .TRUE., otherwise the function returns .FALSE.. There are many other functions that come into this category – consult AMT's own documentation for further details.

SHIFT FUNCTIONS

A number of algorithmic techniques exploit the good connectivity between the DAP's individual processing elements by moving data between the processors in a regular way. The routines that support this important feature are the built-in shift functions, which operate on vectors (and long vectors) and matrices.

Vector (and Long Vector) Shifts

These functions take a vector (long vector that matches the DAP's edge dimensions) and return a result that conforms (type, length, mode, dimension) but in which the values have been SHifted to the Left or to the Right. There are two possibilities for what happens at the 'ends' of the object; values can wrap round from one end to the other – Cyclic shift; constant values can be shifted in – Planar shift. The values shifted in for planar shifts are 0, 0.0, .FALSE. and null. For example, if the components of the vector IV have the values (4, 1, 2, 1) then:

SHLC(IV) will return (1, 2, 1, 4) and SHLP(IV) will return (1, 2, 1, 0)

The shift functions have an optional second parameter – the number of places to shift is taken as the value of that parameter (an INTEGER*4 scalar) modulo the dimension of the vector. So:

SHRC(IV, 5) will return (1, 4, 2, 1)
SHRP(IV, 4) will return (4, 1, 2, 1)

If the matrix:

		1	2	3	4
IM with components	3	4	5	6
	5	6	7	8
	7	8	9	10

were shifted (as a long vector) on a 4*4 DAP then:

		0	3	4	5
SHRP(IM, 3) would give	0	5	6	7
	0	7	8	9
	1	2	3	4

Matrix Shifts

These functions take a matrix and return a result that conforms (type, length, mode, dimensions) but in which the values have been SHifted to the North, South, East or West. There are two possibilities for what happens at the 'edges' of the object; rows/columns can wrap round from one edge to the other – Cyclic shift; constant rows/columns can be shifted in – Planar shift. If:

		1	2	3	4
IM has components	3	4	5	6
	5	6	7	8
	7	8	9	10

then:

		7	8	9	10
SHNP(IM, 3) would give	0	0	0	0
	0	0	0	0
	0	0	0	0

	4	1	2	3
SHEC(IM, 5) would give	6	3	4	5
	8	5	6	7
	10	7	8	9

The shifts on the various rows/columns of a matrix may be different, specifed by an INTEGER*4 vector. For example, if the components of the vector IV have the values (4, 1, 2, 1) and the matrix:

	1	2	3	4
IM has components	3	4	5	6
	5	6	7	8
	7	8	9	10

then:

	1	0	0	0
SHSP(IM, IV) would give	3	2	0	4
	5	4	3	6
	7	6	5	8

	1	2	3	4
SHEC(IM, SHRP(IV)) would give	3	4	5	6
	8	5	6	7
	9	10	7	8

DAP PROGRAM STRUCTURE

DAP Fortran-Plus programs have an overall structure similar to ordinary Fortran programs. They are made up of **SUBROUTINE**s and **FUNCTION**s with one (or more) special **ENTRY SUBROUTINE**s that act as the main entry points from the host. ENTRY SUBROUTINEs are the same as ordinary SUBROUTINEs except that they cannot have parameters.

Statements that control the flow of program execution are very much the same as in Standard Fortran and the non-executable statements (type statements, **DIMENSION** statements, **END** statements etc) and rules about statement ordering will be familiar to Fortran programmers.

CONTROL STATEMENTS

The main control statements are:

block IF

IF (*logical_scalar_expression*$_1$) THEN
...
ELSE IF (*logical_scalar_expression*$_2$) THEN
...
ELSE IF (*logical_scalar_expression*$_n$) THEN
...
ELSE
...
ENDIF

arithmetic IF

IF (*numerical_scalar_expression*) *label*$_1$, *label*$_2$, *label*$_3$

logical IF

IF (*logical_scalar_expression*) *statement*

GOTO

GOTO *label*

computed GOTO

GOTO ($label_1$, $label_2$, ... $label_n$), *integer_scalar_expression*

DO and CONTINUE

DO *label integer_scalar_variable* = *start, terminator, increment*

...

label CONTINUE

CALL

CALL *subroutine_name*

CALL *subroutine_name*(*actual_arguments*)

RETURN, PAUSE and STOP

- RETURN returns control to the calling subroutine/ function.
  It is also the way to pass control back from an entry subroutine to the host

- PAUSE *optional_integer_constant*
  The integer constant is passed back to the run-time diagnostic system and the DAP program's execution is suspended

- STOP *optional_integer_constant*
  The integer constant is passed back to the run-time diagnostic system and the DAP and host programs' execution is abandoned

TRACE

TRACE *level* (*list_of_variables*)

Depending on compile time request for trace level *level* and that request not having been modified since the source code was compiled, the values of *list_of_variables* is sent to the diagnostic output channel

LOCAL DATA AND RECURSION

A Fortrtan-Plus subroutine may call itself recursively. New copies of local data are declared on each recursive call. New copies of local data initialised in type declaration and DATA statements are **not** declared on each call of a subroutine.

Fortran-Plus functions may not directly call themselves recursively. They may, however, be

entered recursively via intermediate routines.

HOST / DAP INTERFACE

The DAP is a processor attached to a host computer system. The host operating system does not know much about the DAP – it regards it as a peripheral – nor do its linker and loader know about DAP programs. This means that special interface routines have to be used to handle entry to DAP programs and the transfer of data between the DAP and its host. In all other respects, the host program is a 'normal' program.

The overall structure of the host program, as far as its dealings with the DAP part of the complete program are concerned, is:

- **CONNECT TO DAP MODULE**
- **SEND DATA**
- **ENTER DAP PROGRAM**
- **PULL DATA BACK**
- **RELEASE DAP**

AMT provides interface routines to initiate all of these steps.

CONNECTING TO DAP MODULE

The INTEGER function DAPCON loads a DAP program into the DAP hardware (or simulator), ready for subsequent entry. The host compilation system does not know anything about DAPCON – it treats it as a user-written function – so you must REMEMBER TO DECLARE IT AS AN INTEGER IN THE PART OF YOUR HOST PROGRAM FROM WHICH IT IS CALLED.

DAPCON's single parameter is:

- a CHARACTER string, to specify the name of the file containing the executable (ie compiled and linked) DAP program

DAPCON returns an INTEGER value, indicating success / failure when loading the DAP program. The possible values returned are:

0 success – the DAP program module has been loaded
1 unable to open executable DAP program file – perhaps the filename was mis-spelled

OVERALL STRUCTURE — DAP

The overall structure of a DAP program is:

- **CONVERT DATA TO DAP FORMAT**
- **PROCESS**
- **CONVERT DATA BACK TO HOST FORMAT**
- **RETURN TO HOST**

STORAGE MODES AND CONVERSION ROUTINES

There are 4 different storage modes on the DAP:

- MATRIX – individual components are stored vertically under different DAP PEs
- VECTOR – individual components are stored vertically under different DAP PEs but in an order different from MATRIX mode – they are **not** stored in the order that matches reduced rank indexing (long vector) of a MATRIX
- SCALAR / SCALAR ARRAY – packed horizontally in DAP row(s)
- HOST – packed horizontally in DAP row(s), as sent from the host

and a set of conversion routines to change data between these different internal formats. The conversion routines are subroutines that run on the DAP and have names:

CONV<f>TO<t>

<f> <t>

H (HOST MODE)

D (DAP MODE)

Conversion from / to host format

The subroutine **CONV_H_TO_D (CONV_D_TO_H)** converts data from host (DAP) format to DAP (host) format. Its parameters are:

- start of data
  the 'thing' (scalar, vector, matrix) at the start of the area of data that is to be converted. Usually, this will be a variable at the start of common block
- integer scalar value – an optional parameter
  the number of 'thing's to convert. Each 'thing' must have the same type, length, mode

and dimensions

Conversion between DAP modes

There is also a conversion routine that will convert between the internal data storage formats of the different DAP modes (scalar, vector, matrix). As these facilities are needed only by advanced users, the routine is not described here.

EXAMPLE

An example **host** program section that passes values of a number of different types / lengths to the DAP might look like:

```
INTEGER DAPCON, FAILCODE
INTEGER X, Y
DOUBLE PRECISION A, B
LOGICAL L
INTEGER*2 I, J, K
INTEGER EQV
EQUIVALENCE (EQV, I)
COMMON /B1/ X(64, 64), Y(64, 64)
COMMON /B2/ A(20)
COMMON /B3/ B(32, 3)
COMMON /B4/ L(16)
COMMON /B5/ I, J, K
    ...
FAILCODE = DAPCON('mydapprog') [1]
IF (FAILCODE .NE. 0) THEN
    <take_error_action>
ENDIF
CALL DAPSEN('B1', X, 4096*2) [2]
CALL DAPSEN('B2', A, 40) [3]
CALL DAPSEN('B3', B, 64*3) [4]
CALL DAPSEN('B4', L, 16) [5]
CALL DAPSEN('B5', EQV, 2) [6]
CALL DAPENT('mydapentry') [7]
        <get values back, if needed>
```

```
      CALL DAPREL [8]
      ...
```

Notes:

1 load the executable DAP program stored in the host's filestore, filename is **mydapprog**, into the DAP and inspect result code. Note that DAPCON has been declared as an INTEGER in the host program, as has FAILCODE

2 pass 64*64 INTEGER*4 arrays X and Y to common block B1 in the DAP program

3 pass one 20 element DOUBLE PRECISION (ie REAL*8) array, A, to common block B2 in the DAP program

4 passs one 32*3 DOUBLE PRECISION array, B, to common block B3 in the DAP program

5 pass one 16 element LOGICAL array, L, to common block B4 in the DAP program

6 Here we wish to pass some INTEGER*2 values. Because DAPSEN requires word aligned addresses for the start of the area of data that is to be sent, we have used the technique of equivalencing an INTEGER*4 variable with the start of the set of INTEGER*2 variables. Note also that we have to send over complete words from the host; we send 2 (I, J, K and a half-word of random junk)

7 transfer control to the DAP program's entry point, mydapentry

8 finished with the DAP so release it

The associated **DAP** example program might look like:

```
      ENTRY SUBROUTINE MYDAPENTRY
      INTEGER MX, MY
      DOUBLE PRECISION VA, VB
      LOGICAL VL(*16), VLD(*16, 32) [1]
      INTEGER*2 SI, SJ, SK
      COMMON /B1/ MX(*64, *64), MY(*64, *64)
      COMMON /B2/ VA(*20)
      COMMON /B3/ VB(*32, 3)
```

```
      COMMON /B4/ VLD [1]
      EQUIVALENCE (VLD, VL) [1]
      COMMON /B5/ SI, SJ, SK
            ...
      CALL CONV_H_TO_D(MX, 2)
      CALL CONV_H_TO_D(VA)
      CALL CONV_H_TO_D(VB, 3)
      CALL CONV_H_TO_D(VL)
      CALL CONV_H_TO_D(SI, 3)
            ...
      RETURN
      END
```

Notes:

1 we have had to make sure that there is enough space for the logical values sent. Each logical value on the host is 32 bits (ie one host word) long. Each logical value on the DAP takes up one bit only

INDEXING RE-VISITED

We have already looked at simple indexing techniques. As indexing techniques play such an important part in writing programs for the DAP, we will look again at the topic (as a reminder of what has been covered so far) before looking at further techniques.

SELECTING (RHS)

Selecting a vector from a set of vectors or a matrix from a set of matrices is a simple extension of selection of a scalar from a scalar array:

```
REAL S, SA(10), SB(10, 10)
S = SA(5)
S = SB(8, 9)

REAL V(*20), VA(*20, 5), VB(*20, 4, 3)
V = VA( , 3)
V = VB( , 1, 2)

REAL M(*25, *35), MA(*25, *35, 7), MB(*25, *35, 2, 5)
M = MA( , , 6)
M = MB( , , 2, 4)
```

Selecting a scalar from a vector, set of vectors, matrix or set of matrices is a simple concept:

```
REAL S, V(*20)
S = V(17)

REAL S, VA(*20, 10)
S = (VA( , 5))(13)          is normally written S = VA(13, 5)

REAL S, M(*25, *35)
S = M(17, 11)

REAL S, M(*25, *35, 10)
```

```
S = (M( , , 8))(7, 21)        is normally written S = M(7, 21, 8)
```

Selecting a column / row vector from a matrix or set of matrices is written:

```
REAL VC(*25), VR(*35), M(*25, *35)
VC = M( , 5)
VR = M(17, )
```

Remember that the trailing ',' is important -- to distinguish the above from a reduced rank index.

```
REAL VC(*25), VR(*35), MA(*25, *35, 5)
VR = (MA( , , 2))(3, )      is usually written VR = MA(3, , 2)
VC = (MA( , , 3))( , 27)    is usually written VC = MA( , 27, 3)
```

A logical vector (matrix) can be used to select a scalar value from a vector (matrix), provided that the dimension(s) of the logical vector (matrix) match those of the object it is indexing. The restriction is that **one and only one** component of the logical index must be .TRUE.:

```
REAL S, V(*25), VA(*25, 7), M(*25, *35), MA(*25, *35, 5)
LOGICAL LV(*25), LM(*25, *35)
    •••
S = V(LV)
    •••
S = (VA( , 4))(LV)   is usually written S = V(LV, 4)
    •••
S = M(LM)
    •••
S = (MA( , , 3))(LM)      is usually written S = MA(LM, 3)
```

It is a simple further extension to use a logical vector to select a column / row from a matrix:

```
REAL VC(*25), VR(*35), M(*30, *35), MA(*25, *30, 5)
LOGICAL LV(*30)
    •••
VR = M(LV, )
VC = (MA( , , 4)( , LV)    can be written VC = MA( , LV, 4)
```

Again, **one and only one** component of the logical index must be .TRUE..

The next fairly simple further extension is to use an integer vector as an index that gathers a column/row vector. For example, if the components of the vector IV have the values (4, 1, 2, 1) and the matrix:

	1	2	3	4
IM has components	3	4	5	6
	5	6	7	8
	7	8	9	10

then:

IM(IV,) would give a 'row' vector	7	2	5	4
IM(, IV) would give a 'column' vector	4	3	6	7

Finally, it is a fairly simple further extension to use a logical matrix to gather a row or column from a matrix:

```
REAL VC(*25), VR(*35), M(*25, *35)
LOGICAL LM(*25, *35)
VR = M(LM, )      selects a 'row' vector
                  one and only one component per column must be .TRUE.
VC = M( , LM)     selects a 'column' vector
                  one and only one component per row must be .TRUE.
```

Shift-Indexing

Shift-indexing is a way of expressing a single position **(nearest neighbour)** shift using an indexing notation. If M is a matrix and V is a vector:

M(+,)	means shift north
M(-,)	means shift south
M(, +)	means shift west
M(, -)	means shift east
M(+, +)	means shift north-west
	etc

V(+)	means shift left
V(-)	means shift right

M(+) means long vector shift left but is only valid if M's dimensions match the DAP's

M(-) means long vector shift right but is only valid if M's dimensions match the DAP's

There remains the need to inform the compiler whether a shift should be cyclic or planar. This is achieved using the non-executable **GEOMETRY** statement:

GEOMETRY (*option*)	controls vector (and long vector) shifts
GEOMETRY (*ns*, *ew*)	controls matrix shifts and *ns* also controls vector (and long vector) shifts

where *option*, *ns* and *ew* may be **CYCLIC** or **PLANAR**.

An example that calculates for all the points on a grid (32*100) the 'average' of its four neighbours might look like:

```
REAL A(*32, *100), M(*32, *100)
GEOMETRY (PLANE, PLANE)
    •••
A = 0.25 * (M(+, ) + M(-, ) + M( , +) + M( , -))
```

MASKING (LHS)

All of the indexing constructs described above apart from shift-indexing are allowed on the left-hand side of an assignment. In addition, a logical index may have any number of components (from none up to all) set .TRUE..

Further, a vector (matrix) expects a vector (matrix) value to be assigned EVEN IF SUBSCRIPTED AS THOUGH IT WERE A CONVENTIONAL ARRAY. This gives rise to surprises:

```
VECTOR V1(*25), V2(*25), M1(*25, *35), M2(*25, *35)
    •••
V2(6) = V1
    •••
M2(17, ) = M1
```

But remember that M2(17) would use reduced rank indexing and:

M2(17) = M1

is not allowed.

Remember that the logical expressions controlling masked assignments can be completely general and that there are a large number of built-in functions that generate logical patterns.

Also, remember that a scalar will be replicated as necessary for assignment to a vector or matrix and a vector will be replicated as necessary (using hints from the actual way that the index is written to decide whether to replicate by rows or by columns).

Finally, brackets around an indexing construct, such as in (M(, , 3))(IV,) and (M(, , 3))(LM), produce a value and hence cannot be assigned to – such constructs **must** be written in the form M(IV, , 3) and M(LM, 3) (which is different from M(LM, , 3)!)

More complicated indexing facilities are also available but not covered in this course as they are less frequently used. Consult AMT's reference documentation for further details.

FURTHER BUILT-IN FUNCTIONS

The built-in function **MERGE** builds a vector or matrix, selecting its components from its first two parameters and returning an object with the same mode and dimensions as its third parameter (which must be a logical vector or logical matrix). For example, if IV1 has the value (1, 3, 5, 7), IV2 has the value (2, 4, 6, 8) and LV has the value (.TRUE., .FALSE., .FALSE., .TRUE.) then:

MERGE(IV1, IV2, LV) gives (1, 4, 6, 7)

The restrictions on the first parameter are that it must match the mode and dimension(s) of the third parameter (or may be a scalar). The second parameter must match the first in type and length and match the mode and dimension(s) of the third (or may be a scalar).

EXTRACTING SUB-VECTORS AND SUB-MATRICES

The built-in function **GETVEC (GETMAT)** extracts a sub-vector from a vector (sub-matrix from a matrix). GETVEC's parameters are:

- the vector value from which a sub-vector is to be extracted
- an integer scalar – the index of the first component of the sub-vector
- an integer scalar – the number of components in the sub-vector

and GETMAT's parameters are:

- the matrix value from which a sub-matrix is to be extracted
- an integer scalar – the row index of the first row of the sub-matrix
- an integer scalar – the column index of the first column of the sub-matrix
- an integer scalar – the number of rows in the sub-matrix
- an integer scalar – the number of columns in the sub-matrix

The type and length of the vector (matrix) returned matches that of the vector (matrix) from which it is extracted. For example, if the components of the vector IV have the values (4, 1, 2, 1) then:

GETVEC(IV, 2, 3) gives (1, 2, 1)

and if the matrix IM has components:

1 2 3 4
3 4 5 6
5 6 7 8
7 8 9 10

then:

GETMAT(IM, 2, 2, 3, 2) gives
4 5
6 7
8 9

ASSIGNING TO SUB-VECTORS AND SUB-MATRICES

The DAP Fortran-Plus subroutine **SETVEC (SETMAT)** assigns a vector (matrix) value to the components of a sub-vector within a vector (sub-matrix within a matrix). SETVEC's parameters are:

- the vector variable within which a sub-vector is to be assigned to
- an integer scalar -- the index of the first component of the sub-vector
- the vector value to be assigned to the sub-vector

and SETMAT's parameters are:

- the matrix variable within which a sub-matrix is to be assigned to
- an integer scalar – the row index of the first row of the sub-matrix
- an integer scalar -- the column index of the first column of the sub-matrix
- the matrix value to be assigned to the sub-matrix

The type and length of the vector (matrix) value must match that of the vector (matrix) to which it is to be assigned. For example, if the components of the vector IV have the values (4, 1, 2, 1) then:

CALL SETVEC(IV, 2, VEC(3, 2)) changes IV to (4, 3, 3, 1)

and if the matrix:

IM has components
1 2 3 4
3 4 5 6

```
5 6 7 8
7 8 9 10
```

then:

```
                                          1  2  3  4
CALL SETMAT(IM, 2, 2, MAT(0, 2, 2) gives  3  0  0  6
                                          5  0  0  8
                                          7  8  9  10
```

as the new value for IM.

MANIPULATING BIT VALUES

The built-in function **GETBIT** extracts a logical value from a scalar, vector or matrix, returning a value of the same mode and dimensions with .TRUE. representing the bit being set and .FALSE. representing the bit being clear. GETBIT's parameters are:

- the value from which the 'bit' is to be extracted
- an integer scalar – the number of the 'bit' that is to extracted

For example, if IM is an INTEGER*4 matrix then:

```
GETBIT(IM, 32)
```

returns a LOGICAL matrix of the same dimensions as IM and each of whose components contains the least significant bit of the corresponding component of IM. It is like having a set of logical matrices equivalenced over IM, but not quite the same – tricks using EQUIVALENCE will work only for 32*32 matrices on DAP 500 systems and 64*64 matrices on DAP 600 systems.

There is a companion subroutine **SETBIT** with parameters:

- the variable within which the 'bit' is to be set
- an integer scalar – the number of the 'bit' that is to set
- a logical expression – the value(s) of the 'bit' that is to be set

The logical value(s) must have the same mode (scalar, vector, matrix) and dimension(s) as the variable whose value is to be changed.

INDEX_VEC

The DAP Fortran-Plus subroutine INDEX_VEC takes a single parameter (INTEGER*4 vector variable) an assigns the values 1, 2, 3, ... in order to the components of the parameter, up to the size of the vector.

YET MORE BUILT-IN FUNCTIONS

There remain a number of other built-in functions that have not been described in the course. Full details can be found in AMT's documentation.

SUPPORT LIBRARIES

AMT provides a number of DAP Application Support Libraries. The main three are the **Digital Signal Processing Library**, the **Image Processing Library** and the **General Support Library.** In addition, there are subroutine interfaces to the DAP's video output channel, to disks attached to the DAP via its fast i/o channel and to other customised interfaces.

The Digital Signal Processing Library and the Image Processing Library are specialised in nature and not decribed further in the course. The General Support Library includes a number of less specialised routines – a very brief summary is given here.

GENERAL SUPPORT LIBRARY

This library is a joint AMT / QMW product that grew originally as researchers started to use the 64*64 DAP at QMC in the early 1980s. It is a library of mathematical and other routines that were developed to satisfy user requests and its implementation helped QMC staff to learn about algorithms for DAP-like systems.

A number of topics and techniques rely on the availability of good fast random number routines – there are a useful selection in the GSLib and more are available from QMW. Other ares covered by the library include sorting and permutation, special functions (to supplement the functions provided by DAP Fortrtan-Plus) and mathematical and machine constants. There are a number of routines for FFTs, matrix operations, eigenvalues and eigenvectors, solution of simultaneous linear equations and the linear assignment problem.

In addition, a number of 'utility' routines supplement the built-in functions of Fortran-Plus.

TIMING FACILITIES

DAP Fortran-Plus provides a subroutine which allows a DAP program to measure its run time.

The subroutine **AMT5_TIMER** obtains values that allow calculation of resident 'elapsed' time and active 'execution' time.

AMT5_TIMER's parameters are:

- an INTEGER*8 variable, to receive the absolute time relative to an arbitrary datum. The datum is fixed while any user program is resident in the DAP, even if that program is not running. With current releases of the system, the datum is fixed at the time the DAP is booted
- an INTEGER*8 variable, to receive the time that the program has been active

Both values are given in units of machine cycles (100 nanosecond on DAP 510 and DAP 610).

Extra facilities provided in DAP simulation system

In addition, the DAP simulation system provides facilities that give a good estimate of the time that would be taken for execution on actual DAP hardware. Using the option / qualifier:

-t1 (UNIX host) **/TIMING=STANDARD** (VMS host)

in a call of **dapopt** (UNIX host) or **DOPTIONS** (VMS host) will provide an estimate of the total time that would be taken and the option / qualifier:

-t2 (UNIX host) **/TIMING=FULL** (VMS host)

will provide intermediate reports every time a Fortran-Plus program executes a CALL statement or a RETURN statement and every time that a system supervisor call is made.

During a simulation run that has requested that timing information be reported, further reports will be generated by:

PAUSE 9999

statements in a Fortran-Plus program. These statements have no special significance unless the simulator is being used and timing information has been requested. In all other cases, they are treated as user-defined pauses – the program is suspended and the diagnostic system is entered.

A further facility provided by the simulation system is the ability to produce a program execution profile ('histogram'). This facility is invoked by using the option / qualifier:

-h (UNIX host) **/HISTOGRAM** (VMS host)

to **dapopt** (UNIX host) or **DOPTIONS** (VMS host).

INPUT / OUTPUT FACILITIES

DAP Fortran-Plus provides simple input/output facilities which allow a DAP program to create, read and update files in the host computer's filestore and access other i/o channels (interactive screen on host, batch input file etc). The interface is via a suite of six subroutines which allow you to:

- open a file
- move to a specified point in the file
- obtain details of the current position in the file
- read from the file
- write to the file
- close the file

The read/write routines provide unformatted (ie binary) I/O only. Formatted I/O is available from within the program running on the host system.

OPENING A FILE

The subroutine **AMT5_OPEN** opens a file and returns an integer value (the file's "identifier") that must be used in calls to perform other functions on the file.

AMT5_OPEN's parameters are:

- a string of up to 32 CHARACTER*1 elements, to specify the name of the file to be opened. File names of less than 32 characters should be followed by a space or the null character

- a CHARACTER*1 value, to specify the mode of opening:
  r - read from file
  w - write to file
    if the file already exists, it will be truncated
    if the file does not exist, it will be created
  u - update file
    the file is opened ready for reading or writing

c - connect to socket (UNIX host only, not available with VMS host)

- an INTEGER*4 variable, to receive the value of an "identifier" for the file, which should be used in subsequent references to the file

- an INTEGER*4 variable, to receive a response indicating success or failure of the file opening operation. A response of zero indicates success, non-zero values are the error number returned by the host system

POSITIONING WITHIN A FILE

The subroutine **AMT5_SEEK** changes the position within the file at which the next read/write operation will be performed.

AMT5_SEEK's parameters are:

- the INTEGER*4 "identifier" obtained via the call of AMT5_OPEN that opened the file

- an INTEGER*4 value that specifies a signed offset in bytes relative to the next parameter (DATUM)

- an INTEGER*4 value (DATUM) that controls the coarse positioning within the file:
  0 - start of file
  1 - current position
  2 - end of file

- an INTEGER*4 variable, to receive a response indicating success or failure of the seek operation. A response of zero indicates success, non-zero values are the error number returned by the host system

OBTAINING DETAILS OF CURRENT FILE POSITION

The subroutine **AMT5_TELL** provides the current position within the file, relative to the start of the file.

AMT5_TELL's parameter are:

- the INTEGER*4 "identifier" obtained via the call of AMT5_OPEN that opened the file
- an INTEGER*4 variable, to receive the offset of the current position within the file (in bytes from the start of the file)
- an INTEGER*4 variable, to receive a response indicating success or failure of the positioning operation. A response of zero indicates success, non-zero values are the error number returned by the host system

READING FROM A FILE

The subroutine **AMT5_READ** reads data from a file into a buffer area.

AMT5_READ's parameters are:

- the INTEGER*4 "identifier" obtained via the call of AMT5_OPEN that opened the file
- the address of the first byte of the buffer into which data is to be read. Typically, this parameter would be the name of a scalar array or would be treated as being in host storage mode – other storage modes can cause confusion for the inexperienced
- an INTEGER*4 value that specifies the number of bytes to be read
- an INTEGER*4 variable, to receive the number of bytes actually read (the system may need to break the transfer down into a number of fixed size 'chunks')
- an INTEGER*4 variable, to receive a response indicating success or failure of the read operation. A response of zero indicates success, non-zero values are the error number returned by the host system

WRITING TO A FILE

The subroutine **AMT5_WRITE** writes data from a buffer area into a file.

AMT5_ WRITE's parameters are:

- the INTEGER*4 "identifier" obtained via the call of AMT5_OPEN that opened the file

- the address of the first byte of the buffer from which data is to be written. Typically, this parameter would be the name of a scalar array or would have been converted to host storage mode – other storage modes can cause confusion for the inexperienced

- an INTEGER*4 value that specifies the number of bytes to be written

- an INTEGER*4 variable, to receive the number of bytes actually written (the system may need to break the transfer down into a number of fixed size 'chunks')

- an INTEGER*4 variable, to receive a response indicating success or failure of the write operation. A response of zero indicates success, non-zero values are the error number returned by the host system

CLOSING A FILE

The subroutine **AMT5_CLOSE** closes a file.

AMT5_CLOSE's parameters are:

- the INTEGER*4 "identifier" obtained via the call of AMT5_OPEN that opened the file

- an INTEGER*4 variable, to receive a response indicating success or failure of the close operation. A response of zero indicates success, non-zero values are the error number returned by the host system

DIRECT EXIT / RESTART FACILITIES

DAP Fortran-Plus does not allow direct call of a host subroutine from the DAP. However, it does provide facilities which allow a DAP program's execution to be interrupted and control returned immediately to the host without the need to follow the chain of RETURN statements back via the entry subroutine through which the DAP program was called. Then, at a later point in the host program's execution, control may be transferred back directly to the point at which the exit to the host was made. This allows co-operating host and DAP programs to be written and provides a mechanism that allows the DAP program to 'call' host routines.

RETURNING IMMEDIATELY TO HOST

The subroutine **AMT5_STOP** returns control immediately to the host program.

AMT5_STOP has no parameters.

RESTARTING IN THE DAP

The special entry subroutine **AMT5_START** is called from the host:

```
CALL DAPENT ('AMT5_START')
```

to restart processing in a DAP program at the point at which it was interrupted by a call of AMT5_STOP. In all other cases, the effect of calling AMT5_START is undefined.

COMPUTATIONAL ERROR CONTROL

FACILITIES FOR COMPUTATIONAL ERROR CONTROL

DAP Fortran-Plus provides a number of facilities that allow the programmer to exercise control over the way that computational errors are handled. There is:

- overall global control – whether the program stops on arithmetic overflow etc
- finer control – error interruption masks
- reports of errors to program variables rather than the DAP's debugging system
- simple masked assignment

GLOBAL CONTROL

The DAP Fortran-Plus routine SETSTATE is an INTEGER*4 function that changes what happens when a computational error occurs. It may be called with its INTEGER*4 parameter taking the following values:

0 suppress underflow errors & interrupt on all other errors
1 interrupt on all errors
2 no interrupt but note all errors
3 as for 2 but ignore underflow
4 as for 2 but note only underflow

The function returns the previous state, so it is always possible to make a temporary change and then revert back. By default, all arithmetic errors apart from underflow cause an interrupt – the example:

```
REAL A(*25, *35), B(*25, *35)
    •••
I = SETSTATE(1)
B = A * 1.0E-75
I = SETSTATE(I)
```

forces interruption on underflow and then re-instates the previous state.

ERROR INTERRUPTION MASKS

An error interruption mask is maintained for each of the modes (scalar, vector and matrix). Each of the masks is of type LOGICAL and is of the same mode as the mode for which it is the mask. Computational errors cause an interrupt if the global state is 0 or 1 (see above) and the corresponding component(s) of the error interruption mask is .TRUE. Error interruption masks that are vectors or matrices are 'active' only during operations where the 'dimensions' of the operation match the dimensions of the mask. The default system supplied masks are set .TRUE. in all positions.

A routine exists in DAP Fortran-Plus that allows the user program to nominate its own masks. This can be done to alter where errors are to be suppressed or, in the case of vector or matrix masks, to alter the dimension(s) of the relevant mask.

The nomination sroutine **NOM_EMSK** takes a single parameter (LOGICAL scalar, vector, matrix) which is nominated as the error interrupt mask for operations on objects of the corresponding mode (and dimension(s), for vectors and matrics) and remains in force until either another variable of the same mode is nominated, the DAP program's flow of control returns from the routine in which NOM_EMSK was called or the masks in force on the routine's entry are re-instated by a call of the parameterless subroutine **RST_EMSKS**. When a subroutine or function is entered, it inherits the error interrupt masks in force on its entry.

ERROR REPORTS TO VARIABLES

Rather like the error interruption masks, error reporting variables may be maintained for scalar, vector and matrix operations. As computational errors occur, the relevant component(s) of the relevant logical variables are set .TRUE., no matter whether the interrupt was actually suppressed or not.

The nomination subroutine **NOM_ERPT** takes a single parameter (LOGICAL scalar, vector, matrix) which is nominated as the error report variable for operations on objects of the corresponding mode (and dimension(s), for vectors and matrics) and remains in force until either another variable of the same mode is nominated, the DAP program's flow of control returns from the routine in which NOM_ERPT was called or the variables in force on the routine's entry are re-instated by a call of the parameterless subroutine **RST_ERPTS**. When a subroutine or function is entered, it inherits the error report variables in force on its entry.

SIMPLE CHECKING FOR ERRORS

The system maintains a logical scalar that indicates whether or not any computational error has occurred (regardless of any masks) since the DAP program started running (or since the scalar has been cleared by the user). Its value may be obtained by a call of the subroutine **GET_ERPT**, which has a single logical scalar parameter.

This global error report variable is cleared by the call of the parameterless subroutine **CLR_ERPT**.

SIMPLE MASKING OF ERRORS

The mask in a masked assignment not only controls the actual assignment but also suppresses error interrupts and reports. This can be used to provide a rudimentary form of error control. For example in:

```
REAL X(*25, *35), Y(*25, *35), Z(*25, *35)
  ...
X(Z .NE. 0.0) = Y / Z
```

there can be no possibility of arithmetic overflow (from a divide by zero). However, this feature only applies to the **FINAL OPERATION** before the assignment so:

```
X(Z .NE. 0.0) = Y / Z + 1.0
```

is not safe.

2 unable to read executable DAP program file – perhaps the file belongs to someone else and you do not have access to it

3 Not an executable DAP program file – perhaps the file contains source code, compiled but not linked coded, host executable code

4 No free DAP resources – DAP not available

5 DAP load failed – very rare, fault detected on DAP or host/DAP link

An example of DAPCON's call in a host Fortran program might look like:

```
INTEGER DAPCON, FAILCODE
...
FAILCODE = DAPCON('compiled_and_linked_DAP_program')
IF (FAILCODE .NE. 0) THEN
    take_error_action
ENDIF
```

SENDING / RECEIVING DATA

DAP Fortran-Plus provides unformatted input from / output to attached discs and host via subroutines, video output via Graphics Library, image input from camera, input from mouse and tablet and other customised connections but no 'Fortran' formatted I/O.

A DAP program is always entered via the call to a parameterless ENTRY SUBROUTINE. Therefore, the usual way to pass data from the host to the DAP is to named data areas (COMMON blocks) in your DAP program via the interface subroutine DAPSEN, which is called from the host program. The usual way to get data back to the host is via the interface subroutine DAPREC, which is called from the host program.

DAPSEN's parameters are:

- a CHARACTER string, to specify the name of the DAP COMMON block to which data is to be sent
- the name of a word-aligned host variable, specifying the start of the data to be sent
- an INTEGER, to specify the number of 32-bit host words to be transferred

DAPREC's parameters are:

- a CHARACTER string, to specify the name of the DAP COMMON block from which data is to be pulled back
- the name of a word-aligned host variable, specifying the start of the area in the host to which data is to be pulled back
- an INTEGER, to specify the number of 32-bit host words to be transferred

PROCESSING ON THE DAP

Control is passed to DAP Fortran-Plus programs by calling the interface subroutine DAPENT from the host program.

DAPENT's single parameter is:

- a CHARACTER string, specifying the name of the DAP entry subroutine to which contol is to be passed

DAPENT may be called as often as required, with the same or different DAP entry points being specified, as long as the DAP has not been released by a call of DAPREL.

RELEASING THE DAP

Those who are not anti-social will wish to release the DAP when they have finished using it. Otherwise, they will still occupy DAP store until their host program is deleted and may hold up other users. The DAP is released by calling the parameterless interface subroutine DAPREL.

COMMON BLOCKS – GUIDELINES

For reasons that will become clearer when we look at the DAP side of the host / DAP interface, separate COMMON blocks should be used for objects with different modes, types and lengths. In addition, BEWARE LOGICALS: LOGICAL VECTORS AND MATRICES ARE HELD IN A COMPACT FORM ON THE DAP. FORTRAN-PLUS COMMON BLOCKS CONTAINING LOGICAL VECTORS AND MATRICES NEED PADDING TO HOLD ALL THE DATA THAT WILL BE TRANSFERRED BETWEEN THE HOST AND DAP.

COMMENTS ON EFFICIENCY

In very simple terms, individual components of a matrix or vector are processed in parallel by different processing elements and a scalar is processed by the DAP's Master Control Unit. Hence, even though a matrix or vector operation may take longer than an individual scalar operation, working with matrices and vectors is more efficient than working with scalars and scalar arrays. DO AS MUCH WORK AS POSSIBLE IN PARALLEL.

The DAP carries out (most) arithmetic in software. So, the greater the precision the longer the execution time. USE THE SHORTEST VIABLE PRECISION and remember that there are a number of precisions over and above the ones available on conventional systems. Remember that there may be some differences in relative performances of differing precisions on systems with co-processors when compared with entry-level systems.

The DAP's processors are connected together in a grid, with good speed for transfer of data between processors and the DAP carries out arithmetic in software. DATA MOVEMENT IS FASTER THAN ARITHMETIC but the balance is different for systems with / without co-processors.

Logical matrices and vectors map very nicely onto the DAP hardware. LOGICAL OPERATIONS ARE FASTER THAN DATA MOVEMENT.

Compared with arithmetic, Fortran-Plus control statements take up only a small proportion of a DAP program's run-time. DON'T WORRY WHETHER WRITING WELL-STRUCTURED PROGRAMS INTRODUCES OVERHEADS.