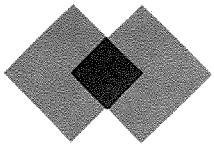


AMT
Active Memory Technology

DAP Series

Parallel Data Transforms

(man022.02)

AMT endeavours to ensure that the information in this document is correct, but does not accept responsibility for any error or omission.

Any procedure described in this document for operating AMT equipment should be read and understood by the operator before the equipment is used. To ensure that AMT equipment functions without risk to safety or health, such procedures should be strictly observed by the operator.

The development of AMT products and services is continuous and published information may not be up to date. Any particular issue of a product may contain part only of the facilities described in this document or may contain facilities not described here. It is important to check the current position with AMT.

Specifications and statements as to performance in this document are AMT estimates intended for general guidance. They may require adjustment in particular circumstances and are therefore not formal offers or undertakings.

Statements in this document are not part of a contract or program product licence save in so far as they are incorporated into a contract or licence by express reference. Issue of this document does not entitle the recipient to access to or use of the products described, and such access or use may be subject to separate contracts or licences.

Technical publication man022.02

First edition 20 August 1988
Second edition 4 November 1988

AMT filename: /cnm/pdt/ed2

AMT will be pleased to receive readers' views on the contents, organisation, etc of this publication. Please make contact at either of the addresses below:

Publications Manager
Active Memory Technology Ltd
65 Suttons Park Avenue
Reading
Berks, RG6 1AZ, UK

Tel: 0734 661111

Publications Manager
Active Memory Technology Inc
16802 Aston St, Suite 103
Irvine
California, 92714, USA

Tel: (714) 261 8901

Preface

This manual describes the Parallel Data Transform (PDT) software which provides a tool for handling the mapping and routing of data on the AMT DAP in a variety of applications. The user interface described in this manual is supported over the range of DAP parallel processors.

Chapter 1 gives an introduction to the PDT approach to data routing on the DAP. Chapter 2 gives a brief overview of Parallel Data Transforms as an introduction to the theory. However, you are assumed to have read reference 1 (see chapter 1), which gives a more detailed description of the theory, prior to reading this manual. The rest of the manual describes the PDT techniques for data mapping and routing, and the PDT statements used for these techniques.

This manual assumes that you are familiar with the notation used in reference 1 and the FORTRAN-PLUS language. Detailed descriptions of the FORTRAN-PLUS language can be found in the AMT manuals:

DAP Series: Introduction to FORTRAN-PLUS	(man001)
DAP Series: FORTRAN-PLUS Language	(man002)

Other relevant AMT manuals are:

DAP Series: Program Development under UNIX	(man003)
DAP Series: Program Development under VAX/VMS	(man004)
DAP Series: DAP System Calls	(man023)

The publication you are now reading is the second edition of this manual.

Contents

Chapter 1

Introduction

Many computationally intensive problems have a very high degree of structured, fine-grain parallelism and benefit substantially from highly parallel execution. Such problems are most suitable for implementation on the highly parallel processor array of the DAP. However, it is necessary to map data appropriately in the DAP memory in order to gain full benefit of the processing capability, and it is often appropriate to change the data mapping during the course of a computation.

FORTRAN-PLUS already provides the facilities for the routing of data in the form of intrinsic functions such as TRAN (which transposes matrix elements). These data routing facilities are now complemented by the use of 'Parallel Data Transforms' (PDTs). PDTs provide you with a powerful tool for efficient mapping and routing of data on the DAP, with minimal programming effort.

1.1 Data routing

Computation on the SIMD (single instruction stream, multiple data stream) processor array of the DAP is fundamentally different from computation on conventional serial computers.

Computation on the DAP is characterised by a continuous cycle of activity, with each cycle comprising:

- Routing of data between the different processor elements (PEs)
- Arithmetic, logical or comparison operations on data local to each PE

The purpose of data routing is to pair up, within each PE, the operands required for a single operation, so that the same operation can be performed in every PE but on different data. Whether the operation is performed or not in a particular PE depends on the value currently held in the PE's A-register, a logical value which may be assigned at compile-time or computed at run-time.

Implementation of the data routing which redistributes the data amongst the PEs is typically achieved via the synchronous interconnection network of the DAP. This network connects each PE to its four neighbours so that the PEs form a two-dimensional grid.

1.2 Data routing in FORTRAN-PLUS

In FORTRAN-PLUS, arrays of data are handled as single objects. It is therefore possible to produce codes which are more concise than codes in current 'serial' languages. FORTRAN-PLUS codes also map easily and efficiently on to the parallel hardware of the DAP. For example, you can express the following FORTRAN loop (for a 32 x 32 data array on the DAP 500):

```
      DO 10 I = 1,32
      DO 10 J = 1,32
10    A(I,J) = B(I,J) + C(I,J)
```

in FORTRAN-PLUS simply as:

```
      A = B + C
```

The whole-array operation maps directly on to the SIMD hardware and corresponds to the local operation part of the computational cycle mentioned in Section 1.1.

Some language primitives have been introduced in FORTRAN-PLUS to specify data routing in a manner which supports the abstraction of working on complete arrays. A simple example of these language primitives is 'TRAN' which transposes a matrix. Hence you can express the following FORTRAN loop (for a 32 x 32 data array on the DAP 500):

```
      DO 10 I = 1,32
      DO 10 J = 1,32
10    A(I,J) = B(J,I)
```

in FORTRAN-PLUS simply as:

```
      A = TRAN(B)
```

The whole-array data routing caused by the above FORTRAN-PLUS statement is implemented on the processor array using the PE interconnection network, and corresponds to the routing part of the computational cycle mentioned in Section 1.1.

1.3 Data routing via PDTs

Providing primitive functions such as TRAN is a direct way to express parallel algorithms, which map very effectively on to the DAP processor array, and can be thought of as the 'mainstream' approach to data routing. Parallel Data Transforms (PDTs) provide a complementary approach to handling the routing of data on processor arrays.

PDTs are both a compact notation for describing regular data mappings, and a set of software that you can use to re-arrange your data in a manner specified using this notation. When you write code to perform arithmetic or other operations on data you need to have some awareness of the physical layout of that data. You may want to change the data mapping to improve the efficiency of some of these operations. Via PDTs, you simply have to specify the required change of mapping and the AMT-supplied software performs the routing for you, making optimum use of the underlying resources.

A 'mapping' can be considered as a new data type. A 'mapping' can be variable or constant; it describes different mappings of data within the DAP memory. Each mapping is expressed as a 'mapping vector' (the definition of 'mapping vector' is given in the next chapter). You can use PDT statements (described in chapters 5, 6, 7, 8 and 9) to transform data mappings described by variable and constant mappings.

In those contexts where PDTs can be applied, the advantages of using PDTs rather than direct routing techniques include:

- Simplification of the description of data mapping and movement
- Production of data routing code with faster execution time than that using direct methods
- Achieving data routing independent of the hardware routing network provided
- Overcoming the constraints of fixed size hardware
- Ease of software production

An overview of the theory of PDTs will be given in the next chapter. References 1 and 2, given below, contain more detailed descriptions of the theory and its advanced applications. You are assumed to be familiar with the reference 1 in this manual.

References

1	*'Data mapping and routing for highly parallel processor arrays' in Future Computing Systems, p 183–224, Vol 2, Number 2, 1987, Oxford University Press*	*P M Flanders and D Parkinson*
2	*'A unified approach to a class of data movements on an array processor' in IEEE Transactions on Computers, p 809–819, Vol C-31, Number 9, September 1982*	*P M Flanders*

Chapter 2

An overview of PDTs

This chapter gives a brief overview of the theory of parallel data transforms. You should read reference 1 (see chapter 1) for a more comprehensive description of this theory.

PDTs introduce a new data type, called 'mapping', into FORTRAN-PLUS. Constants and variables of type 'mapping' are used to describe various mappings of data arrays onto the DAP memory. The values of these constant and variable 'mappings' are expressed as 'mapping vectors'. This chapter describes how 'mapping vectors' represent data mappings and how data routing is achieved using 'mapping vectors'.

2.1 Defining a 'mapping vector'

Consider the simple case of mapping N data items on to a linear array of N PEs such that exactly one data item is held in each PE. The number, P, of the PE in which a given data item with index I is stored is given by:

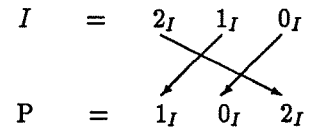
$$\mathrm{P} = \mathit{mapfun}\,(I)$$

where *mapfun* is a one-to-one mapping of I into P over the range 0 to N–1. Each different mapping of data will have a different mapping function, *mapfun*.

There are a number of ways in which *mapfun* can be represented but for the purpose of PDTs, *mapfun* is represented as a 'mapping vector' which can be derived in the following way. The index I in the above equation is replaced by its component bits (binary digits). For N=8, I comprises 3 bits denoted here by $2_I, 1_I$, and 0_I in decreasing order of significance. Substitution into the above equation gives the processor number, P, as a function of these 3 bits as follows:

$$\mathrm{P} = \mathit{mapfun}\,(2_I, 1_I, 0_I)$$

Now, consider the functions which perform a permutation of the bits of I and interpret the resulting sequence of bits as the binary value of P. Each one of the 6 (3 factorial) permutations which can be applied to the three bits defines a different function, *mapfun*, and hence a different mapping of the 8 data items on to the 8 PEs. One of these permutations is depicted below:



where the bits of I are taken in the order $1_I\ 0_I\ 2_I$ to give the binary value of P. By taking just the bottom line of the above diagram the permutation can be represented by:

$$(\text{P:} \quad 1_I \quad 0_I \quad 2_I)$$

which is the 'mapping vector' describing this particular data mapping. To find the number of the processor in which the data item with index I is stored, you simply substitute the bits of I into the mapping vector and read off the number of the processor in binary form. Thus, for example, substituting the value 6 (binary 110) for I into the above mapping vector gives:

$$(\text{P:} \quad 1 \quad 0 \quad 1)$$

indicating that the data item with index 6 is stored in processor 5 (binary 101).

More mapping vectors can be obtained by allowing the inversion of the individual mapping vector elements to be included in the permutations. For example:

$$(\text{P:} \quad \sim 1_I \quad 0_I \quad 2_I)$$

is such a mapping vector, where the sign '$\sim$' preceding the vector element 1_I denotes that the inverse of 1_I is used for this mapping vector element. Substituting the value 6 (binary 110) for I into this mapping vector therefore gives:

$$(\text{P:} \quad 0 \quad 0 \quad 1)$$

indicating that the data item with index 6 is stored in processor 1 (binary 001).

2.2 Extending the definition of 'mapping vectors'

In this section the effect on the form of the mapping vector when handling multi-dimensional data and multi-dimensional processor arrays will be considered.

The effect of storing more than one data item per processor, or of having a two or higher dimensional arrangement of PEs is to split the mapping vector into two or more partitions. Consider the following mapping of 16 data items, $d_0 \ldots d_{15}$, onto a linear array of 4 PEs:

	PE0	*PE1*	*PE2*	*PE3*
Address0	d_0	d_1	d_2	d_3
Address1	d_4	d_5	d_6	d_7
Address2	d_8	d_9	d_{10}	d_{11}
Address3	d_{12}	d_{13}	d_{14}	d_{15}

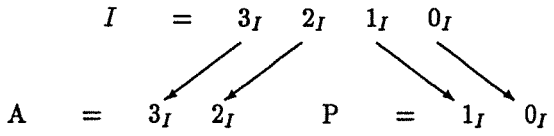
Whereas for the one-to-one mapping of data to the processors, it was necessary to provide a function of the form:

$$\mathrm{P} = mapfun\,(I)$$

to map an item with the index I into a processor number P, it is now necessary to produce both a processor number, P, and an address-within-processor, A, by means of a function of the form:

$$\mathrm{A,P} = mapfun\,(I)$$

In order to achieve this some bits of the index I are used to form the binary value of A and the remaining ones to form the value of P. This process is depicted for the above data mapping as follows:



The mapping vector is again taken from the bottom line of the diagram, but now the vector has two sections, separated by a vertical bar:

$$(\mathrm{A:}\quad 3_I \quad 2_I \quad |\mathrm{P:}\quad 1_I \quad 0_I)$$

Now when the bits of a given value of I are substituted into the mapping vector, the number of the processor holding data item I can be read from the P-section, and the address within the processor can be read from the A-section.

For example, substituting $I = 9$ (binary 1001) into the mapping vector gives:

(A: 1 0 | P: 0 1)

showing that the data item with index 9 is stored at address 2 of PE 1.

The splitting of the mapping vector into two sections as shown above is a consequence of increasing the dimensionality of the memory. When there is a two-dimensional array of PEs, as on the DAP, the mapping vector has to be further split, by replacing the P-section with a C-section and an R-section corresponding to the columns and rows of the two-dimensional PE array. Therefore, the mapping vectors for the DAP will have the form:

(A: *section* | C: *section* | R: *section*)

where each *section* comprises a sequence of mapping vector elements. The full syntax of a mapping vector can be found in the next chapter.

If the array of data has more than one dimension, the data elements are identified by two or more indices rather than a single index I, as above. For example, to map a two-dimensional array on to a linear PE array, we need to represent a mapping function:

$$\text{A,P} = mapfun\,(I, J)$$

where I and J are the two indices of an item of data in the array. The only effect of this is to extend the set of elements to go into the mapping vector. Thus for an 8 x 16 array of data, the complete set of elements to be placed in the mapping vector is:

$$2_I, 1_I, 0_I, \text{ and } 3_J, 2_J, 1_J, 0_J$$

As with the single dimensional data, the order in which the vector elements are placed determines the mapping of data represented by the mapping vector. The location of any data item can be obtained by substituting the bits of its two indices into the mapping vector in the prescribed order and reading out the location. It is convenient to represent the dimensions I and J numerically so that for the above case of an 8 x 16 array of data the mapping vector elements would be referred to as:

$$2_1, 1_1, 0_1, \text{ and } 3_2, 2_2, 1_2, 0_2$$

with I replaced by 1, denoting dimension 1 of the data, and with J replaced by 2, denoting dimension 2 of the data. 1 and 2 are called the dimension subscript of the mapping vector elements. (See also chapter 4 for the formal syntax of 'mapping vector' used on the DAP.)

2.3 Data routing via mapping vectors

As you have seen in sections 2.1 and 2.2 the mapping of multi-dimensional data on to multi-dimensional PE arrays can be concisely represented by one-dimensional 'mapping vectors'.

Consider the transformation of a given data set from the mapping described by the mapping vector MAP1 to that described by the mapping vector MAP2. Such a transformation can be considered either directly in terms of the data routing required, or indirectly in terms of the operations required to transform the mapping vector MAP1 into MAP2. The former is described as solving the problem in 'physical space' and the latter as solving it in 'mapping vector space'. However, the transformation is more simply described in 'mapping vector space' than in 'physical space', as it is just a matter of permuting and inverting the elements of the mapping vector.

The PDT software performs for you the non-trivial task of converting changes in mapping vectors into an efficient sequence of operations to achieve the corresponding routing of the physical data. Each data re-organisation can usually be decomposed into a sequence of simpler changes to the mapping vector; hence only the code which implements the data routing implied by these simpler changes is generated. These simpler changes are called 'generators'. An important generator is the 'exchange' generator which calls for an interchange of a pair of mapping vector elements. Any permutation of elements can in fact be achieved by a sequence of such 'pair-wise exchanges'. Another useful generator is the 'inversion' generator which causes individual elements in a mapping vector to be inverted. Both the 'exchange' and 'inversion' generators and their effect on the physical mappings are discussed in detail in reference 1 (see chapter 1).

You can invoke the exchange and inversion generators directly for the routing of data. However, in many cases, you can ignore all this detail and simply use a PDT REMAP statement quoting the initial and final mapping vectors. The mapping vectors may be known at compile-time (constant mappings) or generated and modified at run-time (variable mappings). Whenever a REMAP statement is used the PDT software determines an efficient sequence of generators to produce the required permutation of mapping vector elements. The PDT data routing or 'remapping' techniques will be discussed in chapters 5, 6, 7 and 8 of this manual.

Chapter 3

PDT statements

The PDT software extends the FORTRAN-PLUS language by allowing PDT statements to be included in the code. This chapter describes the input format and the formal syntax of the PDT statements, and the PDT preprocessor which translates the PDT statements into FORTRAN-PLUS code.

3.1 Input format of PDT statements

PDT statements always have the characters '**#pdt**' in the first four columns. These four characters must be in lower case if you are using a UNIX host system; they can be in upper or lower case if you are using a VAX/VMS host system. Thereafter you can use spaces or tabs freely, except within identifiers.

You can use alphabetic characters in PDT statements. Lower case alphabetic characters are equivalent to their upper case counterparts (except for **#pdt** on the Sun system). You can also use underline characters after the first alphabetic character. Note that you cannot use the underline character to differentiate between identifiers; for example, '*this_ident*' is equivalent to '*thisident*'.

3.1.1 Statement length

The maximum length of a PDT statement line is 255 characters. For each statement, you can have up to 19 continuation lines following the initial line. Continuation lines must have the '**-**' character (instead of '**#pdt**') in the first column.

3.1.2 Identifiers

An identifier comprises a sequence of alphanumeric characters which always begins with an alphabetic character. The total number of characters (not including any underline character '_') in an identifier must not exceed 32; otherwise, the PDT preprocessor will report an error. Spaces are

not allowed within an identifier. To avoid the possibility of name clashes with PDT routines you should avoid names beginning with 'AMT_PDT'.

3.1.3 Keyword parameters

Parameters in PDT statements which are introduced by keywords. For example, the parameter *data_name* is introduced by the keyword **DATA** in some PDT statements. Keyword parameters can be given in any order.

3.1.4 FORTRAN-PLUS expressions

FORTRAN-PLUS expressions are sometimes used in PDT statements. A FORTRAN-PLUS expression is denoted as *int_expr* (see also section 12.7) and can be any of the following:

- An unsigned integer constant
- The name of a FORTRAN-PLUS integer scalar
- A FORTRAN-PLUS expression evaluating to an integer scalar

3.2 Syntax of PDT statements

The formal syntax of the PDT statements in this manual is defined using Backus-Naur form (BNF) with the following conventions:

- Non-terminals are printed in *italics* and sometimes include the underline character '_'
- Terminals are printed in **bold** with all alphabetic characters in upper case. Non-alphabetic characters are also used; they include '_', '#', '~', '^', '|', and the square brackets '[ ]'
- Alternatives are enclosed in braces, '{ }'
- Optional items are enclosed in double square brackets, '[[ ]]'. Note that square brackets, '[ ]', are used in terminals
- Items which can appear more than once in a statement are immediately followed by an asterisk, '*'
- Optional items which can appear more than once in a statement are in double square brackets followed immediately by an asterisk, '[[ ]]*'
- Continuation lines of PDT statements do not begin with **#pdt**; instead they always have the '-' character in the first column.

  Note that, for easy reading, the parameters of various PDT statements are printed as indented continuation lines (without the '-' character) in this manual. These parameters can appear in the first line of a PDT statement provided that the line is within its length limit (of 255 characters)

3.3 The PDT preprocessor

The PDT preprocessor translates the PDT statements into standard FORTRAN-PLUS code prior to FORTRAN-PLUS compilation. The original PDT statements are replaced by comment statements.

As well as making compile-time checks on all PDT statements, the PDT preprocessor also decides which mapping strategies to use for remapping constant mappings (see chapter 5 for details).

You do not have to specially invoke the PDT preprocessor. FORTRAN-PLUS programs containing PDT statements are preprocessed, compiled and linked in the usual way, using the **dapf** command on the UNIX host system or the **DFORTRAN** command on the VAX/VMS host system. (See the AMT manuals *DAP Series: Program Development under UNIX* and *DAP Series: Program Development under VAX/VMS* for details of these commands.)

Chapter 4

The syntax of mapping vectors

The formal syntax of a mapping vector describing the mapping of data of any dimension onto the two-dimensional processor array of the DAP is:

([[**A:** *section* |]] **C:** *section* | **R:** *section*)

where **A:**, **C:** and **R:** identify the section in which the element is located. The A-, C- and R-sections should appear in the order as shown above. The A-section (see section 4.2) is optional, but the C- and R-sections must always be present and contain the correct number of mapping vector elements (for example, 5 in the case of 32 x 32 DAPs). The total number of elements in a mapping vector must not exceed 32.

section in a mapping vector has the syntax:

*mapping_vector_element**

that is, *section* is a sequence of mapping vector elements.

4.1 Mapping vector elements

Each mapping vector element has the form:

[[~]]*significance* [[∧ *dimension*]]

where *significance* identifies the index bit of the element, and *dimension* identifies the data array dimension to which the index bit relates. *significance* and *dimension* must be positive integers, in the ranges 0 to 30 and 0 to 10 respectively.

There are no restrictions as to how *significance* and *dimension* should be numbered, provided they are within the allowed limits. However, for ease of interpretation, you are recommended to adopt

the convention whereby both *significance* and *dimension* assume consecutive values beginning at 0, with dimension 0 reserved for the 'sub-atomic' dimension of the data, that is, when the precision of the data is treated as a dimension (see reference 1, given in chapter 1). In some applications, you may find it more appropriate to adopt your own numbering strategy .

The character '∧' which precedes *dimension* is used to indicate a subscript, so that, for example, an element written as 2_1 in section 2.2 is now written as 2∧1. You can denote an inverted mapping vector element by preceding it with the character '∼'. The dimension subscript of the element can be omitted; if so, the default value of 1 will be assumed.

For example, a 32 x 32 matrix (on a 32 x 32 DAP) can have the mapping vector:

(C: 4∧2 3∧2 2∧2 1∧2 0∧2 | R: 4∧1 3∧1 2∧1 1∧1 0∧1)

which describes the mapping of a FORTRAN-PLUS vertical mode matrix in the DAP memory.

4.1.1 Dot sequences

For convenience, you can express the above mapping vector using 'dot sequences':

(C: 4...0∧2 | R: 4...0∧1)

A 'dot sequence' is an abbreviation for a succession of vector elements (with the same dimension subscript). A 'dot sequence' has the form:

integer1...integer2

where the start and end points of the sequence are specified by *integer1* and *integer2*, both of which are integer constants. The sequence can be increasing or decreasing in steps of 1 as determined by the relative values of the start and end points. You can use the dot sequence for the significance of an element, but not for the dimension. The character '∼' preceding a dot sequence means that all elements in the dot sequence are inverted.

You can use a dot sequence to represent some or all of the elements in any *section* of a mapping vector. For example, you can have a mapping vector:

(C: 4...1∧2 0∧1 | R: 4...1∧1 0∧2)

which describes another possible mapping of the 32 x 32 matrix on a 32 x 32 DAP in the previous example.

4.2 The A-section

A mapping vector will have an A-section when it describes more than one DAP-size matrix. For example, for the DAP 500 (which has a 32 x 32 processor array):

(A: 7...5∧2 | C: 4...0∧2 | R: 4...0∧1)

is a mapping vector describing a PLANAR (or 'sheet') mapping of a 32 x 256 data array.

Note that there are 3 mapping vector elements in the A-section in the above mapping vector, which correspond to 8 (number of data items divided by number of processor elements) DAP-size matrices of data. In general, the number of DAP-size matrices of data described by a mapping vector is 2^n, where n is the number of elements in the A-section of the mapping vector.

To avoid unexpected results, you should ensure that the mapping vectors you specify in a REMAP or DYNAMIC_REMAP statement (see chapters 5 and 6) describe an amount of data which does not exceed that contained in the named data area.

Chapter 5

Constant mappings

Constant mappings are data mappings whose mapping vectors are explicitly known at compile-time. Using constant mappings rather than variable mappings (whose mapping vectors are not explicitly known at compile-time) results in faster execution. This is partly because more work can be done at compile-time when the mapping vectors are known explicitly, and partly because a better sequence of generators may be chosen by the PDT preprocessor than by the run-time PDT software.

In order to transform from one explicitly known data mapping into another, you declare these mappings as constant mappings, and then request the transformation using a REMAP statement. A description of the PDT statements used for these purposes is given in this chapter.

5.1 Declaring constant mappings

You can declare constant mappings by means of a MAP statement. MAP statements simply declare constant mappings to the PDT software; they are non-executable and do not generate any code. You can therefore place MAP statements anywhere in the source file provided that they precede any other PDT statements which refer to the mapping.

There are two forms of the MAP statement; they use a mapping vector and a 'standard mapping' to define the declared constant mapping respectively. A constant mapping definition remains effective throughout the compilation until it is redefined or a PDT CLEAR_MAP_TABLE statement (section 5.5) is encountered.

5.1.1 MAP statements using mapping vectors

When you use a mapping vector to define a constant mapping the MAP statement has the syntax:

#pdt MAP *cmap_name* = *mapping_vector*

where *cmap_name* is an identifier giving the name of the constant mapping and *mapping_vector* is the mapping vector which defines the constant mapping.

mapping_vector has the syntax:

([[**A:** *section* |]] **C:** *section* | **R:** *section*)

and is fully described in chapter 4.

All elements in a constant mapping vector use explicit integer values, *not* variables. The values given for the significance must be in the range 0 to 30, and the values given for the dimension must be in the range 0 to 10. An omitted dimension 'subscript', as in the examples below, will assume the default value of one.

The PDT preprocessor checks the mapping vector to ensure that no element is repeated and that there are the correct number of mapping vector elements in the C- and R-sections (for example, 5 in the case of 32x32 DAPs). The PDT software will report an error if either of these conditions is not satisfied.

Examples

The statement

#pdt MAP LONG_VECTOR = (C: 9...5 | R: 4...0)

declares a constant mapping called LONG_VECTOR and defines it using the mapping vector **(C: 9...5 | R: 4...0)**

The statement

#pdt MAP ROW_MAJ = (C: 4...0 | R: 9...5)

declares a constant mapping called ROW_MAJ and defines it using the mapping vector **(C: 4...0 | R: 9...5)**

5.1.2 MAP statements using standard mappings

An alternative form of the MAP statement defines a constant mapping as being one of a set of named 'standard' mappings known to the system. This form of the MAP statement has the syntax:

#pdt MAP *cmap_name* = *int_dimensions standard_map*

where *cmap_name* is an identifier giving the name of the constant mapping and *standard_map* is one of a set of standard mappings (see chapter 12) which is used to define the constant mapping.

int_dimensions gives the dimensions of the data array and has the syntax:

[*integer* [[, *integer*]]*]

which is an ordered list of integers with successive integers giving the values of the successive dimensions. Each dimension must be a positive integer and a power of 2. For example, [64] indicates a one-dimensional array of 64 elements, whereas [32,64,4] indicates a three-dimensional array of 8192 elements.

The PDT preprocessor will list the mapping vector corresponding to a constant mapping declared by a MAP statement using 'standard mappings' at compile-time. You may find it useful to list all the constant mappings which have been declared up to a certain point in your program. You can do this by using the PRINT_MAP_TABLE statement. The list can be cleared by a subsequent CLEAR_MAP_TABLE statement. These two statements are discussed at the end of this chapter.

Example

The statement

#pdt MAP LONG_VECTOR = [1024] LINEAR

declares the constant mapping LONG_VECTOR, of a data array of dimension 1024, according to the standard mapping 'LINEAR'. The standard mapping 'LINEAR' has a similar data mapping to a FORTRAN-PLUS long vector where successive columns of a matrix are concatenated to form a vector.

5.2 Using constant mappings

You can use the declared constant mappings in a REMAP statement, which causes an *in situ* transformation of data from one constant mapping to another. The REMAP statement is an executable statement; therefore you can only put it where other executable statements are allowed in FORTRAN-PLUS.

You should ensure that the two constant mappings specified in a REMAP statement describe the same amount of data, and that this amount does not exceed that in the named data area. The

amount of data described by a mapping vector depends on the number of defined elements in the mapping vector. In the case of a constant mapping specified as a standard mapping, the amount of data described is determined by the dimensions quoted in the MAP statement in which the mapping is declared.

5.2.1 The REMAP statement

The REMAP statement has the syntax:

#pdt REMAP FROM=*cmap1_name*, **TO=***cmap2_name*,
DATA=*data_name*, **LENGTH=***length*

and remaps the data from the mapping *cmap1_name* to the mapping *cmap2_name*.

FROM, TO, DATA and **LENGTH** are keywords where:

- **FROM** and **TO** introduce the names of the initial and final mappings, *cmap1_name* and *cmap2_name*, respectively
- **DATA** introduces the name of the data, *data_name*. The latter is the name of a FORTRAN-PLUS variable or a name with an indexing construct, currently in scope
- **LENGTH** introduces an integer expression, *length*, giving the precision of the data (in bits) in most cases. *length* is really the size in bits of the vertical mode 'atoms' manipulated by the PDT software, which may differ from the precision. For example when the mapping vectors have elements describing the mapping of individual bits within a single datum, *length* will not be equal to the precision but equal to 1. *length* can have any positive integer value; it is not constrained to be a power of two

You should make appropriate declarations for all variables referred to in PDT statements.

Note that, unless *data_name* is an indexed variable, the REMAP statement only needs the address of the data area, and the declared (or default) type, rank and dimensions of the FORTRAN-PLUS variable are ignored. It is nonetheless good practice to declare the type, rank and dimensions of all variables, whenever possible. (See also *Example 1* in section 5.3.)

5.2.1.1 Data remapping using the REMAP statement

The PDT preprocessor produces code from the REMAP statement to perform the required remapping of data. In doing so it evaluates up to three different strategies to determine how to do the remapping and generates code for the alternative with the best performance.

The remapping is performed by a sequence of PDT generators. The PDT preprocessor will output comments which indicate the sequence of generators chosen, the resulting intermediate mappings and an estimate of the cost of executing the data remapping in terms of machine cycles per bit-plane of data processed. The estimates given by the preprocessor are based on the inner loop times of the PDT code. They are provided merely as a guide to performance; more accurate timings can be obtained using the timing facilities provided in FORTRAN-PLUS (see AMT publication *DAP Series: DAP System Calls*).

5.3 Examples

Example 1

The following code for the DAP 500 is an example of the use of constant mappings. The FORTRAN-PLUS subroutine SHUFFLE_8K defined in the code performs a perfect shuffle of an 8192-element integer vector VEC, of precision IPREC. The vector VEC is assumed to be mapped as a set of 8 FORTRAN-PLUS long vectors.

```
#pdt  MAP IDENTITY = (A: 12...10 |C: 9...5 |R: 4...0)
#pdt  MAP SHUFFLED = (A: 11...9 |C: 8...4 |R: 3...0 12)
      SUBROUTINE SHUFFLE_8K(VEC, IPREC)
#pdt  REMAP FROM=IDENTITY, TO=SHUFFLED, DATA=VEC, LENGTH=IPREC
      RETURN
      END
```

As explained in reference 1 (see chapter 1), a perfect shuffle of data in physical space corresponds to a cyclic left shift of mapping vector elements. Thus the elements of the mapping SHUFFLED are those of the mapping IDENTITY shifted left cyclically by one place.

Note that in this example, it is not possible to completely declare the type, rank and dimension of the data area VEC since it is not possible to declare variable precision objects in FORTRAN-PLUS.

You can use the #if directive as a convenient means of introducing into your code different mapping vectors for DAPs of different sizes. On UNIX host systems (see AMT manual *DAP Series: Program Development under UNIX*), the following code will achieve a perfect shuffle of the data VEC on both the DAP 500 and the DAP 600:

```
#if   DAPSIZE==32
#pdt  MAP IDENTITY = (A: 12...10 |C: 9...5 |R: 4...0)
#pdt  MAP SHUFFLED = (A: 11...9 |C: 8...4 |R: 3...0 12)
#endif

#if   DAPSIZE==64
#pdt  MAP IDENTITY = (A: 12 |C: 11...6 |R: 5...0)
#pdt  MAP SHUFFLED = (A: 11 |C: 10...5 |R: 4...0 12)
#endif

      SUBROUTINE SHUFFLE_8K(VEC, IPREC)
#pdt  REMAP FROM=IDENTITY, TO=SHUFFLED, DATA=VEC, LENGTH=IPREC
      RETURN
      END
```

The #if directive has a different syntax on VAX/VMS host systems (see AMT manual *DAP Series: Program Development under VAX/VMS*), for example:

```
#if   DAPSIZE==32
```

should be replaced by:

```
#if   DAPSIZE.EQ.32
```

Example 2

In this example the standard mappings (see chapter 12), PLANAR and DEPTHFIRST_PLANAR, are used to define the constant mappings SHEET and CRINKLED. The FORTRAN-PLUS subroutine SHEET_TO_CRINKLED defined in the code causes transformation of a 1024 x 1024 image of 8-bit pixels from sheet to crinkled mapping on a DAP 500.

```
#pdt  MAP SHEET    = [1024,1024] PLANAR
#pdt  MAP CRINKLED = [1024,1024] DEPTHFIRST_PLANAR
      SUBROUTINE SHEET_TO_CRINKLED(IMAGE)
      INTEGER*1 IMAGE(,,32,32)
#pdt  REMAP FROM=SHEET, TO=CRINKLED, DATA=IMAGE, LENGTH=8
      RETURN
      END
```

The same code can be used for the DAP 600 by substituting the declaration statement for the data array IMAGE with:

```
      INTEGER*1 IMAGE(,,16,16)
```

5.4 The PRINT_MAP_TABLE statement

The PDT preprocessor allows you to have up to 200 constant mappings declared at any one time. You can list the mapping vectors of the declared constant mappings at any point in the code by means of the PDT statement:

#pdt PRINT_MAP_TABLE

provided that you specify the full listing option when you invoke the FORTRAN-PLUS compilation system. This option is specified when you use the **-L2** flag with the **dapf** command on UNIX host systems, or use the **/LIST /FULL** qualifiers with the **DFORTRAN** command on VAX/VMS host systems.

5.5 The CLEAR_MAP_TABLE statement

The table of declared mappings can be cleared by the PDT statement:

#pdt CLEAR_MAP_TABLE

Constant mappings declared prior to this statement will subsequently be 'undefined'. An error will be reported if these mappings are referred to in REMAP statements which appear after this statement.

Chapter 6

Variable mappings

In some problems you may not know the explicit form of the mapping for a set of data at compile-time. For example, you may want to describe the PLANAR (sheet) mapping of an array whose dimensions are given as variables rather than constants; or you may want to describe the mapping of a set of data for which the mapping changes during the running of the program. In these cases, the description of the mapping can be held in a 'variable mapping', that is, one whose value can be initialised and changed at run-time. Changing the value of a variable mapping causes it to describe a different mapping.

Like 'integer', 'mapping' is a data type. Just as you can have constants and variables of type 'integer', you can have both constants and variables of type 'mapping'. 'Mapping' is an abstract data type in that variable mappings can be manipulated by the user without knowing how they are represented within the program.

A description of how variable mappings are declared, initialised and used in the remapping of data is given in this chapter. The next chapter describes how variable mappings can be changed using LET statements. Variable mappings can also be updated when executing PDT generator statements (see chapter 8 for details).

6.1 Declaring variable mappings

You can declare and initialise COMMON variable mappings using COMMON_MAP statements. Variable mappings which are local to a subroutine or function are declared and initialised using DECLARE_MAP statements. Variable mappings which are dummy arguments of a subroutine or function are also declared using DECLARE_MAP statements but they must not, of course, be initialised. A variable mapping may be used as an actual argument to a subroutine or function, but unlike other FORTRAN-PLUS variables, it must not appear on the left-hand side of a function reference.

You should ensure that there is an initialisation for every COMMON and local variable mapping (except for dummy arguments) in the program.

6.1.1 The COMMON_MAP statements

You can declare a variable mapping as a COMMON variable, and optionally initialise it using COMMON_MAP statements. The syntax of a COMMON_MAP statement is similar to that of a MAP statement used to declare constant mappings. A COMMON_MAP statement can be any of the following forms:

- **#pdt COMMON_MAP** *vmap_name* [[, *vmap_name*]]*

  This statement declares variable mappings as COMMON variables but does not initialise them. It contains a list of identifiers which give the names of the variable mappings, *vmap_name(s)*, to be declared

- **#pdt COMMON_MAP** *vmap_name* = *mapping_vector*

  This statement declares the variable mapping, *vmap_name*, as a COMMON variable and initialises it using an explicitly known mapping vector, *mapping_vector*. The syntax of the mapping vector is described in chapter 4

- **#pdt COMMON_MAP** *vmap_name* = *int_dimensions standard_map*

  This statement declares the variable mapping, *vmap_name*, as a COMMON variable and initialises it using a 'standard mapping', *standard_map*. *standard_map* is one of a set of 'standard mappings' known to the system (see chapter 12 for details). *int_dimensions* gives the dimensions of the data array and has the syntax:

  [*integer* [[, *integer*]]*]

  It is an ordered list of integers with successive integers giving the values of the successive dimensions. Each dimension must be a positive integer and a power of 2. For example, [64] indicates a one-dimensional array of 64 elements; whereas [32,64,4] indicates a three-dimensional array of 8192 elements

- **#pdt COMMON_MAP** *vmap_name* = ∧

  This statement declares the variable mapping *vmap_name* as a COMMON variable and initialises all the values in its mapping vector to be 'undefined'

The COMMON_MAP statements are subject to the same constraints as other FORTRAN-PLUS COMMON statements. You can place COMMON_MAP statements in BLOCK DATA subprograms and in the declaration part of subroutines and functions. You should initialise a COMMON mapping once and only once using one of the last three COMMON_MAP statements shown above. The name, *vmap_name*, given for the variable mapping should be distinct from other user variable

names and from other FORTRAN-PLUS COMMON area names.

6.1.2 The DECLARE_MAP statements

The COMMON_ MAP statement declares variable mappings with similar scope to FORTRAN-PLUS COMMON variables. Other variable mappings are similar in scope to FORTRAN-PLUS local variables and must be declared with DECLARE_MAP statements. Such local variable mappings which are not used as dummy arguments can be given initial values by the DECLARE_MAP statement. Local variable mappings which are used as dummy arguments must not be initialised by DECLARE_MAP statement.

The four DECLARE_MAP statements are:

- **#pdt DECLARE_MAP** *vmap_name* [[, *vmap_name*]]*

  This statement declares variable mappings as local variables or dummy arguments but does not initialise them. It contains a list of identifiers which give the names of the variable mappings, *vmap_name*, to be declared

- **#pdt DECLARE_MAP** *vmap_name* = *mapping_vector*

  This statement declares the variable mapping, *vmap_name*, as a local variable and initialises it using an explicitly known mapping vector *mapping_vector*. The syntax of the mapping vector is described in chapter 4

- **#pdt DECLARE_MAP** *vmap_name* = *int_dimensions standard_map*

  This statement declares the variable mapping, *vmap_name*, as a local variable and initialises it using a 'standard mapping', *standard_map*. *standard_map* is one of a set of 'standard mappings' known to the system (see chapter 12 for details). *int_dimensions* gives the dimensions of the data array and has the syntax:

  [*integer* [[, *integer*]]*]

  It is an ordered list of integers with successive integers giving the values of the successive dimensions. Each dimension must be a positive integer and a power of 2

- **#pdt DECLARE_MAP** *vmap_name* = ∧

  This statement declares the variable mapping *vmap_name* as a local variable and initialises all the values in its mapping vector to be 'undefined'

6.2 Using variable mappings

You can use any variable mapping which has been declared (see section 6.1) in a DYNAMIC_REMAP statement. The DYNAMIC_REMAP statement is an executable statement which causes an ***in situ*** transformation of data between two variable mappings.

You should ensure that the variable mappings you specify in a DYNAMIC_REMAP statement are such that:

- all the mapping vector elements in the R- and C-sections are defined
- all defined mapping vector elements in the A-section are grouped together and at the right hand end of the section
- the significance and dimensions of all defined elements are within the ranges allowed in PDT statements. Allowed ranges for significance and dimension are 0 to 30 and 0 to 10 respectively
- all mapping vector elements (or their inverse) are present in both mappings, hence the two mappings describe the same amount of data; this amount must not exceed that in the named data area

These conditions are satisfied for mappings which have been initialised with COMMON_MAP and DECLARE_MAP statements, but for mapping vectors which have been constructed by assignment to individual elements follow the rules in section 7.1.

6.2.1 The DYNAMIC_REMAP statement

The DYNAMIC_REMAP statement which has the syntax:

#pdt DYNAMIC_REMAP FROM=*vmap1_name*, **TO=***vmap2_name*,
DATA=*data_name*, **LENGTH=***length*

remaps the data from the mapping *vmap1_name* to the mapping *vmap2_name*.

FROM, TO, DATA and **LENGTH** are keywords where:

- **FROM** and **TO** introduce the names of the initial and final variable mappings, *vmap1_name* and *vmap2_name*, respectively
- **DATA** introduces the name of the data, *data_name*. The latter is the name of a FORTRAN-PLUS variable or a name with an indexing construct, currently in scope
- **LENGTH** introduces an integer expression, *length*, giving the precision of the data (in bits) in most cases. *length* is really the size in bits of the vertical mode 'atoms' manipulated by the PDT software, which may differ from the precision. For example when the mapping vectors have elements describing the mapping of individual bits within a single datum, *length* will not be equal to the precision but equal to 1. *length* can have any positive integer value; it is not constrained to be a power of two

The data being remapped by the DYNAMIC_REMAP statement is determined by the number of defined elements in the A-section of the mapping vector; 2^n DAP-size matrices will be remapped if there are n defined elements in the A-section. In the case of a variable mapping initialised as a standard mapping, the amount of data remapped is determined by the dimensions quoted in the COMMON_MAP statement or the DECLARE_MAP statement in which the mapping is initialised.

6.3 Example

The following example is a FORTRAN-PLUS subroutine for transforming data from sheet to crinkled mapping. The variable remapping technique is used here since the dimensions of the array are not known at compile-time.

The DECLARE_MAP statement is used here to declare the variable mappings as local variables of the subroutine SHEET_TO_CRINKLE. The dimensions of the data array, NROWS and NCOLS, must both be a power of 2 and at least equal to the DAP edge-size.

```
      SUBROUTINE SHEET_TO_CRINKLE (IN_DATA, NROWS, NCOLS, PRECISION)
#pdt  DECLARE_MAP MAP1, MAP2
      INTEGER NROWS, NCOLS, PRECISION
#pdt  LET MAP1[ ] = [NROWS, NCOLS] PLANAR
#pdt  LET MAP2[ ] = [NROWS, NCOLS] DEPTHFIRST_PLANAR
#pdt  DYNAMIC_REMAP FROM=MAP1, TO=MAP2, DATA=IN_DATA, LENGTH=PRECISION
      RETURN
      END
```

The two LET statements in the above code assign the 'standard mappings' PLANAR (which corresponds to a sheet mapping) and DEPTHFIRST_PLANAR (which corresponds to a crinkled mapping) to the variable mappings MAP1 and MAP2 respectively. Here both the standard mappings (see chapter 12 for details) will describe NROWS x NCOLS arrays. The LET statements are fully described in the next chapter.

Chapter 7

The LET statements

A variable mapping is a variable, in a similar sense to an integer vector being a variable. Just as with an integer vector you can access and change the elements at run-time, the same can be said of a mapping variable. However, unlike an integer vector, the elements of a mapping variable should not be accessed or changed by indexing constructs. The elements of a mapping variable are accessed and changed using PDT LET statements instead. The LET statements are special assignment statements which allow you to access variable mappings in a manner which is independent of their detailed representation.

Each variable mapping is expressed as a mapping vector, therefore the elements of the variable mapping referred to are the individual mapping vector elements.

7.1 Updating a complete variable mapping

You can assign all the elements of a variable mapping in a single statement. The LET statements described in this section all assign a new value to a complete variable mapping. These statements have the general syntax:

#pdt LET *vmap_name*[ ] = *vmap*

where *vmap_name* is an identifier giving the name of the variable mapping to be assigned. The empty square brackets indicate that the complete mapping is to be updated.

vmap specifies the mapping to be assigned to the variable mapping, *vmap_name*. There are three possible forms for *vmap*; they are expressed in the following LET statements:

- **#pdt LET** *vmap1_name*[ ] = *vmap2_name*[ ]

  This statement assigns to the complete mapping, *vmap1_name*, another complete variable mapping, *vmap2_name*; that is, the value of *vmap2_name* is copied into *vmap1_name*

- **#pdt LET** *vmap1_name*[ ] = *var_dimensions standard_map*

  This statement assigns to the complete mapping *vmap1_name* the 'standard mapping', *standard_map*, describing a data array whose dimensions are given by *var_dimensions*. (See chapter 12 for a description of the 'standard mappings')

  var_dimensions gives the dimensions of the data array and has the syntax:

  [*int_expr* [[, *int_expr*]]*]

  It is an ordered list of integer expressions with the successive integer expressions giving the values of the successive dimensions. Each dimension should be an integer expression which results in a positive integer equal to a power of 2. An integer expression can be a variable, a scalar or a FORTRAN-PLUS expression

- **#pdt LET** *vmap_name*[ ] = ∧

  This statement sets every element of the variable mapping, *vmap_name*, to be 'undefined'

7.2 Updating individual elements of a variable mapping

Once a mapping vector has been initialised, either by a COMMON_ MAP or DECLARE_ MAP statement, or by a LET statement above, individual elements of that vector can be updated. You can use a LET statement of the following form to update a single element of a mapping vector:

#pdt LET *vmap_name*[*index*] = *vmap_element*

where *index* identifies the position of the element within the mapping vector of the variable mapping, *vmap_name*, into which the value given by the construct *vmap_element* is written.

index has the syntax:

section_identifier int_expr

where *section_identifier* gives the name of the section in which the element referred to is located. It can be any one of the following:

R:
C:
A:
RCA:

int_expr is an integer expression which gives the position of the element in the section specified. Remember that the numbering of positions goes from right to left and starts at 0, in each of the R-, C- and A-sections of the vector. You can use the section identifier '**RCA:**' to specify a position anywhere within the mapping vector. The positions in this case are numbered from right to left, starting at 0. As there is a limit of 32 on the total number of elements in a mapping vector, you should not use an RCA index which exceeds 31 to specify a mapping vector element.

For example, you can specify the element at position 3 of the C-section of the mapping vector:

(A:1 0 | C:4 3 2 1 0 | R:4 3 2 1 0)

by quoting either of the indices:

[C:3] or **[RCA:8]**

There are three possible constructs for *vmap_element*; they are expressed in the following LET statements:

- **#pdt LET** *vmap_name*[*index*] = [[~]] *int_expr* ∧ *int_expr*

  This statement writes the value given by the construct, ' [[~]] *int_expr* ∧ *int_expr* ', into an element, specified by *index*, in the variable mapping vector *vmap_name*. The integer expression, *int_expr*, before the '∧' symbol gives the significance of the element and the integer expression after the symbol gives the dimension. The element is inverted if the sign '~' is placed before the integer expressions

- **#pdt LET** *vmap1_name*[*index1*] = [[~]] *vmap2_name* [*index2*]

  This statement assigns the value of a specified element of the variable mapping vector, *vmap2_name*, to a specified element of another variable mapping, *vmap1_name*. The positions of the elements are given by *index1* and *index2* respectively. You can assign the inverse of an element by placing the sign '~' before it

- **#pdt LET** *vmap_name*[*index*] =∧

  This statement writes the value 'undefined' to an element, specified by *index*, in the variable mapping vector, *vmap_name*

7.3 Finding the components of a mapping vector element

You can extract the components of a mapping vector element and write the value to an integer variable by a LET statement of the form:

#pdt LET *int_variable* = *qualifier vmap_name*[*index*]

where the component to be extracted is specified by *qualifier*, the element is identified by *index*, and the variable mapping vector is identified by *vmap_name*.

The only valid words for *qualifier* are: **INVERSION_OF, SIGNIFICANCE_OF,** and **DIMENSION_OF.**

For example, if the value of the element at **[RCA:4]** of the mapping MAP1 is ~3∧4 then the statements:

```
#pdt  LET I = INVERSION_OF MAP1[RCA:4]
#pdt  LET J = SIGNIFICANCE_OF MAP1[RCA:4]
#pdt  LET K = DIMENSION_OF MAP1[RCA:4]
```

will assign the values 1, 3, and 4 to I, J, and K respectively. Note that the **INVERSION_OF** an element gives the value 1 for an inverted element and 0 for a non-inverted element.

7.4 Extracting the index of a mapping vector element

You can assign to an integer variable the index of an element in a variable mapping vector by a statement of the form:

#pdt LET *int_variable* = **INDEX_OF** *int_expr1*∧*int_expr2* **IN** *vmap_name*

The integer variable *int_variable* is assigned the RCA-position of an element of the variable mapping vector *vmap_name*. The significance and dimension of the element are given by the integer expressions *int_expr1* and *int_expr2* respectively.

For example, given that **X** is an integer variable:

#pdt LET I = INDEX_OF X∧1 IN MAP3

the integer variable I is assigned the RCA-position of the element with the value **X∧1** (or **~X∧1**) in the variable mapping MAP3 by this statement. If this element is not present in the mapping vector then I is given the value -1.

7.5 Example

The subroutine BITREV below performs a 'bit-reversal' reordering of the data along dimension DIM of an array ARRAY of N 24-bit elements. The routine will work for any array whose dimen-

sions are a power of two and for any mapping strategy (sheet, crinkled, ...) on the DAP. The DAP edge-size is contained in the variable ES whose value is returned by the built-in subroutine EDGE_SIZE.

```
      SUBROUTINE BITREV(ARRAY, N, DIM, MAP)
#pdt  DECLARE_MAP MAP
      INTEGER N, DIM, P, PS(31), I, L, R, ES
      CALL EDGE_SIZE (ES)

#pdt  WITH DATA=ARRAY, LENGTH=24, MATS=(N/(ES * ES))

C     FIND ALL THE MAPPING VECTOR ELEMENTS THAT REFER TO DIMENSION DIM
      I = 0
20    CONTINUE
#pdt  LET P = INDEX_OF I^DIM IN MAP
      I = I + 1
      PS(I) = P
      IF (P .GE. 0) GOTO 20

C     RE-ORDER THE MAPPING VECTOR ELEMENTS
      L = I - 1
      R = 1
30    IF (R .GE. L) RETURN
#pdt  E(RCA: (PS(L)); RCA: (PS(R)))
      L = L - 1
      R = R + 1
      GOTO 30

      END
```

Note that the variable mapping MAP of the array is supplied as a parameter to the routine. The PDT generator statement:

#pdt E (RCA : (PS(L)) ; RCA : (PS(R)))

causes the exchange of the elements at the RCA-positions specified by PS(L) and PS(R) of the data ARRAY, specified by the PDT WITH statement. The generator statement and the WITH statement are described in detail in the next chapter.

The process of a 'bit-reversal' reordering of data involves assigning a 'new' position to each data element. The rule for the assignment is such that the bits of the index of the 'new' position are those of the original position taken in the reverse order. For example, the following diagram shows the original and 'new' position of 8 data elements in a vector:

Original position	0	1	2	3	4	5	6	7
Index bits	(0 0 0)	(0 0 1)	(0 1 0)	(0 1 1)	(1 0 0)	(1 0 1)	(1 1 0)	(1 1 1)

'New' position	0	4	2	6	1	5	3	7
Index bits	(0 0 0)	(1 0 0)	(0 1 0)	(1 1 0)	(0 0 1)	(1 0 1)	(0 1 1)	(1 1 1)

The effect is that the elements at positions whose index bits are the reverse order of each other are exchanged, for example, the element at position 1 (binary 0 0 1) has been exchanged with that at position 4 (binary 1 0 0).

Chapter 8

PDT functions

8.1 Fast computation of powers and logarithms

You are frequently required to compute powers of two and logarithms to the base two when using PDTs. The PDT software provides you with two high performance functionswhich can be used anywhere in FORTRAN-PLUS. These functions should be declared as type integer and mode scalar in an EXTERNAL statement at the start of any routine in which they are referenced, for example:

```
EXTERNAL INTERNAL SCALAR FUNCTION AMT_ PDT_POW2
```

8.1.1 AMT_PDT_POW2

AMT_PDT_POW2 takes a single argument, an integer expression whose value must be in the range 0 to 30, and returns an integer variable containing the value of two raised to the power of *int_expr*. If *int_expr* is outside this range, the program will STOP and report an error with the message 'STOP 2002' at run-time.

8.1.2 AMT_PDT_LOG2

AMT_PDT_LOG2 takes a single argument, an integer expression whose value must be an exact power of two in the range 1 to 2^{30} and returns an integer variable containing the value of the base two logarithm of *int_expr*. If *int_expr* is outside this range, the program will STOP and report an error with the message 'STOP 2002' at run-time.

Chapter 9

Direct use of the PDT generators

When executing the REMAP (see section 5.2.1) and DYNAMIC_REMAP (see section 6.2.1) statements the PDT software automatically remaps the data by applying a sequence of one or more 'generators'. Each 'generator' is a basic remapping operation, used as a kind of building block in producing more complicated data remappings.

The PDT software also allows you to invoke these 'generators' directly by means of PDT generator statements. When using the 'generators' directly, there is no need to have mappings (constant or variable) describing the data. In the case where the data is described by a variable mapping, the mapping is not automatically updated when the data is remapped. However, if required, you can update the mapping in accordance with the 'generator' applied to the data, by using the CHANGE_MAP statement (see section 9.3).

There are two classes of generators: one corresponding to an exchange of two mapping vector elements; the other corresponding to an inversion of a single mapping vector element. A detailed description of the notation used for the generators can be found in reference 1 (given in chapter 1).

Before data can be remapped by directly invoking PDT generators, a specification of the data must be given by executing a WITH statement. The generators will then operate on the data specified by the most recently executed WITH statement. Note that REMAP and DYNAMIC_REMAP statements also specify the data to be mapped.

9.1 The PDT WITH statement

The PDT WITH statement has the following syntax:

#pdt WITH DATA=*data_name*, **LENGTH=***length*, **MATS=***num_mats*

This statement specifies the data to be remapped by subsequent generator statements. **DATA**,

LENGTH and **MATS** are keywords where:

- **DATA** introduces the name of the data, *data_name*. The latter is the name of a FORTRAN-PLUS variable or a name with an indexing construct, currently in scope
- **LENGTH** introduces an integer expression, *length*, giving the precision of the data (in bits) in most cases. *length* is really the size in bits of the vertical mode 'atoms' manipulated by the PDT software, which may differ from the precision. For example when the mapping vectors have elements describing the mapping of individual bits within a single datum, *length* will not be equal to the precision but equal to 1. *length* can have any positive integer value; it is not constrained to be a power of two
- **MATS** introduces the number of DAP-sized matrices, *num_mats*. The data being remapped is considered as *num_mats* DAP-sized vertical mode matrices each of precision *length*. *num_mats* is therefore equal to 2^n where n is the number of mapping vector elements in the A-section of a mapping vector describing the data mapping.

  You can specify the logarithm to the base 2 of the number of matrices, *log_mats*, instead of the number of matrices by substituting **LOG_MATS**=*log_mats* for **MATS**=*num_mats*. Note that *log_mats* is equal to the number of elements in the A-section of the mapping vector

9.2 The PDT generator statement

The PDT generator statement has the syntax:

#pdt *generator_sequence**

It is a list of one or more 'generator sequences'.

9.2.1 Generator sequence

A *generator_sequence* can be either an exchange or inversion sequence. Each exchange or inversion sequence represents a succession of any one of the exchange or inversion generators described in reference 1. An exchange is denoted by **E**; and an inversion is denoted by **I**.

The simplest form of generator sequence is made up of only one generator. For example, the single-generator sequence:

I(R:0)

remaps the data in a way which corresponds to inverting the mapping vector element at position 0 of the R-section. Note that the positions of elements in each section of a mapping vector are numbered from right to left, with the numbering starting from zero.

The construct:

I(R:0) I(R:1) I(R:2) I(R:3) I(R:4)

remaps the data as if the elements at positions from 0 to 4 in the R-section of the mapping vector had been inverted. The dot notation can be used to abbreviate sequences of generators. For example, the following construct:

I(R:0...4)

produces the same result.

9.2.1.1 Inversion sequence

The general construct representing an inversion sequence has the form:

I(*ifield* **)**

where *ifield* specifies the vector element(s) to be inverted and has the syntax:

section_identifier selector

section_identifier gives the name of the section in which the element(s) referred to is located; it can be any one of the following:

R:
C:
A:
RCA:

You can only use the mapping vector section identifiers '**R:**', '**C:**', and '**A:**' if you know the section of the mapping vector involved at compile-time. Alternatively you can use the section identifier '**RCA:**' to specify a position anywhere within the mapping vector. The *selector* following '**RCA:**' is an index within the complete mapping vector with positions numbered from right to left, starting at zero. For example, the mapping vector element positions are taken in the order **R:0, R:1, R:2, R:3, R:4, C:0, C:1, C:2, C:3, C:4, A:0, A:1,...** on the DAP 500. Note that an RCA index must be within the allowed range of 0 to 31 (since the total number of mapping vector elements is limited to 32).

selector specifies the position(s) of the element(s) and has either of the following forms:

int_expr
integer [[... *integer*]]

that is, it can be an integer expression or a dot sequence. An integer expression can be a constant, a scalar variable or a FORTRAN-PLUS expression. Note that a dot sequence in this context is used to specify a sequence of positions, not mapping vector element values as before. It can be increasing or decreasing in steps of 1 as determined by the relative values of the start and end points; the start and end points must be integer constants. Whenever a dot sequence is employed for the *selector*, the construct **I** (*ifield*) represents a succession of inversion generators. Note that the selected mapping vector element(s) must be defined and within the allowed ranges.

9.2.1.2 Exchange generator sequence

The general construct representing exchange generator sequences has the form:

E (*efield1* ; *efield2*)

where *efield1* and *efield2* specify the vector elements to be exchanged.

Each *efield* has the syntax:

section_identifier [[~]] *selector*

where *section_identifier* and *selector* have the syntax as in *ifield*.

If required, you can invert the element(s) exchanged by an **E** generator by placing the symbol '~' before the *selector*. A dot sequence preceded by '~' means that all elements in the dot sequence are inverted. You should bear in mind that if elements are inverted within an **E** generator, the logical effect of the generator operation is the same as performing an inversion on these elements before the exchange.

For example, the PDT generator statement:

#pdt E (R : 3 ; C : ~4)

is equivalent to the statement:

#pdt I (C : 4) E (R : 3 ; C : 4)

9.2.1.3 Examples of generator sequences

Some examples of PDT generator sequences are shown in Table 9.1. Note that although the effect of the sequences is described in terms of mapping vector changes, only the data mapping corresponding to these vector changes actually takes place; the mapping vector itself is not altered. If the data is described by a variable mapping, you have the option of updating the mapping vector in line with the transformed data, via the CHANGE_MAP statement (see section 9.3).

Generator Sequence	*Operation*
I(R:3)	Inverts the mapping vector element at position 3 in the R-section
E(R:3;C:0)	Exchanges the mapping vector element at position 3 in the R-section with that at position 0 in the C-section
E(R:(I+1);A:0)	Exchanges the mapping vector element at position I+1 in the R-section with that at position 0 in the A-section, where I is an integer variable or scalar
E(R:0...4;C:0...4)	Exchanges the mapping vector elements in positions 0 to 4 in the R-section with the corresponding elements in the C-section
E(R:0...4;A:2)	Exchanges the elements in positions 0 through 4 in the R-section with that at position 2 of the A-section. This is equivalent to the succession of generators: **E(R:0;A:2) E(R:1;A:2) E(R:2;A:2) E(R:3;A:2) E(R:4;A:2)**
E(R:~3;C:0)	Inverts the mapping vector element at position 3 of the R-section and then exchanges it with the element at position 0 in the C-section
E(R:~I;C:~J)	Exchanges and inverts mapping vector elements at position I in the R-section and position J in the C-section. Note that the execution time for this is the same as that of **E(R:I;C:J)**

Table 9.1 Examples of PDT generator sequences

9.2.2 Generator statement appended to a WITH statement

For convenience, you can append one or more PDT generator statements, without the '**#pdt**', to a WITH statement if the WITH statement immediately precedes the generator statement(s) in the program text. You can therefore have generator statements as continuation lines to a WITH statement (see example below). The data specified in the WITH statement will be remapped by all subsequent generator statements, until the next WITH statement, REMAP statement or DYNAMIC_REMAP statement is executed.

For example, the PDT statement:

```
#pdt WITH DATA=MAT1, LENGTH=24, MATS=1
-                E(R:0...4;C:0...4)
```

will transpose the single 24-bit DAP-sized matrix MAT1 on the DAP 500.

The illustration on the front cover of this manual shows the physical movement of data corresponding to a similar statement but for a hypothetical 16 x 16 DAP. The first figure shows the original mapping of the data; and the four remaining figures represent the data at successive stages of the transposition.

9.3 The CHANGE_MAP statement

When you use PDT generators to remap data described by a variable mapping, the mapping is not automatically updated. If you want to update the mapping as well as remap the data then you should use the CHANGE_MAP statement in place of the generator statement. The CHANGE_MAP statement has the syntax:

#pdt CHANGE_MAP *vmap_name generator_sequence**

where *vmap_name* is the name of the variable mapping which describes the data that is being remapped by applying the *generator_sequence(s)* (see section 9.2.1).

As with the PDT generator statements, you can append a CHANGE_MAP statement to a WITH statement which immediately precedes the CHANGE_MAP statement. You can also use continuation lines if you wish. Thus the most general form of the WITH statement has the syntax:

#pdt WITH DATA = *data_name*, **LENGTH =** *length*, **MATS =** *num_mats*
- **[[[[CHANGE_MAP** *vmap_name* **]] *generator_sequence** **]]**

9.4 Example

The following code is a simple example of the direct use of PDT generators in data remapping. The subroutine SHUFFLE performs a perfect shuffle on the vector VEC of N 32-bit items, where N is supplied as a parameter and is a power of two not less than the number of processing elements on the DAP. The vector is assumed to be stored in vertical mode as a set of FORTRAN-PLUS long vectors. Note that, with this mapping, a perfect shuffle corresponds to a left cyclic shift of the elements within the mapping vector.

```
        SUBROUTINE SHUFFLE(VEC, N)
        EXTERNAL INTEGER SCALAR FUNCTION AMT_PDT_LOG2
        INTEGER I, MAX, N, ES
        CALL EDGE_SIZE (ES)

        MAX = AMT_PDT_LOG2(N) - 1

#pdt    WITH DATA=VEC, LENGTH=32, MATS=(N/(ES*ES))
        DO 10 I = 1, MAX
#pdt    E(RCA: MAX; RCA: (I-1))
10      CONTINUE
        RETURN
        END
```

The FORTRAN-PLUS built-in subroutine EDGE_SIZE returns a value equal to the DAP edge-size to the variable ES; the square of ES will then give the number of PEs on the DAP.

The FORTRAN-PLUS PDT function (see section 8.1 for details) assigns the logarithm to the base 2 of N minus one to the integer variable MAX. Hence MAX gives the largest RCA index in the mapping vector (since the RCA numbering of elements starts at zero).

The cyclic left shift of the mapping vector elements is achieved by executing a single generator statement repeatedly within the do-loop in the above code. The PDT WITH statement (see section 9.1) specifies the data VEC to be remapped, giving its precision (in bits) and the number of DAP-sized matrices (the number of data items divided by the number of processor elements on the DAP).

Chapter 10

The COPY_DATA statement

You may find it useful to be able to move data around in a more flexible manner than that allowed in FORTRAN-PLUS. For example, the number of planes of data may not correspond to an available precision, or the precision may not be known at compile-time. You can copy any number of planes of data using the PDT COPY_DATA statement. The precision of data can be supplied as an integer variable or expression which is evaluated at run-time.

10.1 The syntax

In its simplest form the COPY_DATA statement has the syntax:

#pdt COPY_DATA FROM=*data1_name*, **TO=***data2_name*, **PLANES=***int_expr*

This statement copies data from the data area, *data1_name*, to the data area, *data2_name*. *data1_name* and *data2_name* may be the same or may overlap in any way.

FROM, **TO** and **PLANES** are keywords where:

- **FROM** and **TO** introduce the names of the data areas, *data1_name* and *data2_name*, from and to which data is copied respectively.
- **PLANES** introduces an integer expression, *int_expr*, giving the number of bit planes to be copied

For example, the statement:

#pdt COPY_DATA FROM=IMAGE1, TO=IMAGE2, PLANES=2000

copies 2000 bit-planes from the array IMAGE1 to the array IMAGE2.

There are three optional parameters which you can use with the COPY_DATA statement. They are:

- **MASK**
- **OFFSET**
- **SHIFT**

The full syntax of the COPY_DATA statement is therefore:

#pdt COPY_DATA FROM = *data_name*[**OFFSET** *int_expr*],
TO = *data_name*[**OFFSET** *int_expr*],
PLANES = *int_expr*
[, **MASK** = *mask_expr*]
[, **SHIFT** = *dirn_geom int_expr*]

10.1.1 The MASK parameter

You can use a mask with the COPY_DATA statement to control writing to the output data area. This facility is called by including in the COPY_DATA statement the optional parameter **MASK**, specified by a mask expression:

MASK = *mask_expr*

where *mask_expr* is the name of a logical matrix variable, or a FORTRAN-PLUS expression which evaluates to a logical matrix.

The MASK parameter and its argument provide activity control in the COPY_DATA statement. For example, if A and B are FORTRAN-PLUS 32-bit integer matrices, and X is a FORTRAN-PLUS logical matrix, then the COPY_DATA statement:

#pdt COPY_DATA FROM = B, TO = A, PLANES = 32, MASK = X

will have the same effect as the FORTRAN-PLUS assignment statement:

A(X) = B

Both statements assign B(i,j) to A(i,j) only if X(i,j) is .TRUE.; if X(i,j) is .FALSE., the value of A(i,j) is unchanged.

The FORTRAN-PLUS statement is restricted to copying data with a fixed number of bit-planes, such as 32 bit-planes for the integer matrices in the above example. However, the COPY_DATA statement provides a more flexible way of copying data, in that you can specify any number of bit-planes of data to be copied within the statement.

10.1.2 The OFFSET parameter

You can specify plane offsets within the data areas from and to which the data is copied by appending to the names of these data areas the following parameter:

OFFSET *int_expr*

where *int_expr* is an integer expression whose resultant value specifies the plane offset of the output area or input area.

If you specify a plane OFFSET within the area FROM which the data is copied, the PDT software will copy data starting from the bit-plane specified. For example, the statement:

#pdt COPY_DATA FROM = B OFFSET 33, TO = A, PLANES = 37

will copy 37 bit-planes from data B, starting at the 34^{th} bit-plane and ending at the 70^{th} bit-plane, to data A. These 37 bit-planes will then be the first 37 bit-planes in A.

If you specify a plane OFFSET within the area TO which data is copied, the copied data will be placed in this area from the bit-plane specified. For example, the statement:

#pdt COPY_DATA FROM = B, TO = A OFFSET 33, PLANES = 37

will copy the first 37 bit-planes of data B to data A, and the 37 bit-planes of data will be located in A from the 34^{th} bit-plane to the 70^{th} bit-plane.

10.1.3 The SHIFT parameter

You can also shift the data, as it is copied, in any one of the north, south, east or west directions using plane or cyclic geometry. Note that the original data area is not affected by this shift.

You can achieve this using the optional parameter **SHIFT** which is specified by:

SHIFT = *dirn_geom int_expr*

where *dirn_geom* is the direction and geometry of the shift and the integer expression *int_expr* gives the magnitude of the shift and must be between 1 and (DAP edge-size – 1).

dirn_geom can be any of the following:

NC
EC
SC
WC
NP
EP
SP
WP

The first letter gives the direction (north, south, east or west) and the second letter gives the geometry (cyclic or planar) of the shift. The data shifts produced by this PDT facility have the same effect as their counterparts in FORTRAN-PLUS, within the allowed range of magnitude of 1 to (DAP edge-size – 1). Outside this range the result of such a shift is undefined.

10.2 Example

The following code is written for a 32 x 32 DAP. The routine MAGNIFY, shown below, magnifies a 32 x 32 matrix, IMAGE, of precision PIXEL_SIZE, by a factor of two in both the horizontal and vertical dimensions. Thus each element of the original matrix is replaced by a 2 x 2 set of elements all equal to the original one. The resulting 64 x 64 array is mapped in 'PLANAR' form (known also as sheet mapping) in the same memory area as the original data.

```
#pdt  MAP INITIAL      = (A:       |C:    5...1^2|R:    5...1^1)
#pdt  MAP INTERMEDIATE = (A:       |C:4...1^2 5^2|R:4...1^1 5^1)
#pdt  MAP FINAL        = (A:5^2 5^1|C:    4...0^2|R:    4...0^1)


      SUBROUTINE MAGNIFY(IMAGE, PIXEL_SIZE)
      INTEGER DEPTH, PIXEL_SIZE
#pdt  REMAP FROM=INITIAL, TO=INTERMEDIATE, DATA=IMAGE, LENGTH=PIXEL_SIZE
#pdt  COPY_DATA FROM=IMAGE, TO=IMAGE OFFSET PIXEL_SIZE, PLANES=PIXEL_SIZE
#pdt  WITH DATA=IMAGE, LENGTH=PIXEL_SIZE, MATS=2
-           E(A:0;R:0)
      DEPTH = 2 * PIXEL_SIZE
#pdt  COPY_DATA FROM=IMAGE, TO=IMAGE OFFSET DEPTH, PLANES=DEPTH
#pdt  WITH DATA=IMAGE, LENGTH=PIXEL_SIZE, MATS=4
-           E(A:1;C:0)
      RETURN
      END
```

In the above code, the INITIAL and FINAL mappings are defined as constant mappings by the PDT MAP statements. These mappings have been derived in the following way:

The physical mapping of the elements in the required 64 x 64 matrix can be represented as follows:

```
a  a  b  b  .  .  .
a  a  b  b  .  .  .
c  c  d  d  .  .  .
c  c  d  d  .  .  .
.  .  .  .  .  .  .
.  .  .  .  .  .  .
.  .  .  .  .  .  .
```

where a, b, c, ... are elements in the initial data IMAGE. This 64 x 64 matrix is mapped in PLANAR form, therefore it has the mapping vector:

(A: 5∧2 5∧1 | C: 4...0∧2 | R: 4...0∧1)

by definition of the standard mapping, 'PLANAR' (see chapter 12 for details). This mapping vector is given as the FINAL mapping in the above code.

A 32 x 32 matrix of the following form (which contains the unexpanded data of IMAGE) can be obtained by discarding alternate rows and columns of the 64 x 64 matrix:

```
a  b  .  .
c  d  .  .
.  .  .  .
.  .  .  .
```

Using PDTs, the mapping vector for this 32 x 32 matrix can be obtained by discarding the elements with significance 0 from the R- and C-sections of the mapping vector for the above 64 x 64 matrix:

(A: | C: 5...1∧2 | R: 5...1∧1)

Note that this mapping vector is written in terms of the final (expanded) data, hence it has no 0∧1 or 0∧2 element. All the mapping vector elements are placed within the C-section and R-section as the 32x32 matrix will map on to the DAP 32x32 processor array as a single DAP-sized matrix. This mapping vector is given as the INITIAL mapping in the above code.

Hence, by working backwards from the required mapping, the mapping vectors for the INITIAL and FINAL mappings are obtained. The transformation from the INITIAL to the FINAL mapping can be achieved in various ways, one of which is via the subroutine MAGNIFY. Note that in this subroutine, most of the transformation is done *before* the data is expanded, resulting in greater efficiency. The data goes through an INTERMEDIATE mapping which requires little manipulation to transform to the FINAL mapping once the data is expanded. All three mappings are declared as constant mappings (see chapter 5 for details) in the MAP statements. The mapping FINAL is not actually used but it shows the final appearance of the mapping vector.

The 32x32 matrix IMAGE is first remapped from the INITIAL mapping to the INTERMEDIATE mapping, via the REMAP statement (see section 5.2.1 for details). The generator sequence required for this remapping is determined by the PDT preprocessor. After the remapping, the data IMAGE will be described by the INTERMEDIATE mapping vector:

(A: | C: 4...1∧2 5∧2 | R: 4...1∧1 5∧1)

The COPY_DATA statement which follows copies the data in IMAGE and writes it back to IMAGE, offset by PIXEL_SIZE bit-planes. Therefore IMAGE now comprises two DAP-sized matrices, and has an element in the A-section of its mapping vector:

(A: 0∧1 | C: 4...1∧2 5∧2 | R: 4...1∧1 5∧1)

Then the first PDT generator statement (see section 9.2), appended to the first WITH statement (see section 9.1), causes the mapping vector element at position 0 in the A-section to be exchanged with that at position 0 in the R-section, giving the mapping vector:

(A: 5∧1 | C: 4...1∧2 5∧2 | R: 4...0∧1)

The data will be remapped according to this mapping vector, but the constant mapping INTERMEDIATE does not change. Note that the element 0∧1 has been incorporated into the dot sequence in the R-section (see chapter 4).

IMAGE is expanded further via the second COPY_DATA statement which copies DEPTH (= 2 x PIXEL_SIZE) bit-planes of data from IMAGE and writes the data back to IMAGE, offset by DEPTH planes. IMAGE is now four times its original size and has a second element in the A-section of its mapping vector:

(A: 0∧2 5∧1 | C: 4...1∧2 5∧2 | R: 4...0∧1)

The second PDT generator statement, appended to the second WITH statement, causes the element at position 1 in the A-section to be exchanged with that at position 0 in the C-section. The mapping vector describing IMAGE is now:

(A: 5∧2 5∧1 | C: 4...0∧2 | R: 4...0∧1)

which is the required 64 x 64 array mapped in 'PLANAR' form.

Chapter 11

Performance of PDT statements

11.1 The REMAP statement

When you request data remapping via the REMAP statement (see section 5.2.1), the PDT preprocessor determines an efficient sequence of mapping vector exchanges and inversions to perform the transformation. The preprocessor can sometimes find faster solutions when there is at least one element in the A-section of the mapping vectors specifying the two mappings. Thus you can sometimes obtain better performance by representing a single matrix as two matrices, with elements of half the length. For example, the mapping vector:

```
(A:0∧0|C:9...5∧1|R:4...0∧1)
```

can be used instead of:

```
(C:9...5∧1|R:4...0∧1)
```

for a 1024-element data set on the DAP 500.

Note that in the above example, the mapping vector element introduced in the A-section is written as 0∧0. It is recommended that when you introduce mapping vector elements by splitting individual data items, you should adopt the convention of assigning the newly introduced mapping vector elements a dimension subscript of zero (to indicate a 'sub-atomic' dimension). This convention is useful in the interpretation of mapping vectors.

If the mapping vector in the above example is used in a REMAP statement, the value of the LENGTH parameter must be half the number of bits in the individual data.

11.2 PDT generator statements

When you directly specify the PDT generators to be used for the transformation of data (in PDT generator statements) you may find it useful to have some idea of the performance of each individual exchange or inversion operation. The cost in processing time per bit-plane of data processed for these operations (on both DAP 500 and DAP 600 series of processors) are summarised in Table 11.1. Note that the quoted figures represent inner loop timings and do not include overheads for calling the generator functions, loop control, and so on. You should therefore only use these figures as a guide to the performance of these operations.

Direct PDT Generator Operation	*Number of Machine Cyles per Bit-plane of Data Processed*	
	DAP 500	*DAP 600*
Inversion of a mapping vector element in the A-section	2	2
Inversion of the mapping vector element at position x in the R-section, or at position x in the C-section	7, 9, 13, 21 and 19 for x = 0, 1, 2, 3, and 4 respectively	7, 9, 13, 21, 37 and 34 for x = 0, 1, 2, 3, 4 and 5 respectively
Exchange of two mapping vector elements in the A-section	1	1
Exchange of a mapping vector element anywhere in the A-section with an element at position x in the C-section or R-section	6, 7, 9, 13 and 15 for x = 0, 1, 2, 3 and 4 respectively	6, 7, 9, 13, 21 and 23 for x = 0, 1, 2, 3, 4 and 5 respectively
Exchange of two mapping vector elements at positions x and y in the R-section (or of two mapping vector elements at positions x and y in the C-section)	$(10 + 2 * (2^x - 2^y))$ where x and y are such that $x > y$	same as for DAP 500
Exchange of a mapping vector element at position x in the R-section (C-section) with one at position y in the C-section (R-section)	36 when $x = y = 4$ $(26+2^{y+1})$ when $x = 4$ and $y < 4$ $(10+2*(2^x+2^y))$ when $x < 4$ and $y < 4$	68 when $x = y = 5$ $(42+2^{y+1})$ when $x = 5$ and $y < 5$ $(10+2*(2^x+2^y))$ when $x < 5$ and $y < 5$

Table 11.1 Performance of direct PDT generator operations

Chapter 12

Collected syntax of the PDT statements

This chapter is a collection of the syntax of the PDT statements that you can use for remapping data. The syntax of the PDT statement is defined using Backus-Naur form (BNF) with the following conventions:

- Non-terminals are printed in *italics* and sometimes include the underline character '_'
- Terminals are printed in **bold** with all alphabetic characters in upper case. Non-alphabetic characters are also used; they include '_', '#', '~', '^', '|', and the square brackets '[ ]'
- Definitions of non-terminals are preceded by the string of characters '::='
- Alternatives are printed on separate lines and enclosed in braces, '{ }'
- Optional items are enclosed in double square brackets, '[[ ]]'. Note that square brackets, '[ ]', are used in terminals
- Items which can appear more than once in a statement are immediately followed by an asterisk, '*'
- Optional items which can appear more than once in a statement are in double square brackets followed immediately by an asterisk, '[[ ]]*'
- Continuation lines of PDT statements do not begin with **#pdt**; instead they always have the '-' character in the first column.

  Note that, for easy reading, the parameters of various PDT statements are printed as indented continuation lines (without the '-' character) in this manual. These parameters can appear in the first line of a PDT statement provided that the line is within its length limit (of 255 characters)

12.1 The PDT statements

PDT_stmt ::= { *MAP_stmt* | *REMAP_stmt* | *PRINT_MAPS_stmt* | *CLEAR_MAPS_stmt* | *COMMON_MAP_stmt* | *DECLARE_MAP_stmt* | *DYNAMIC_REMAP_stmt* | *LET_stmt* | *WITH_stmt* | *CHANGE_MAP_stmt* | *GENERATOR_stmt* | *COPY_DATA_stmt* }

12.2 The statements handling constant mappings

MAP_stmt ::= { **#pdt MAP** *cmap_name* = *mapping_vector* | **#pdt MAP** *cmap_name* = *int_dimensions standard_map* }

REMAP_stmt ::= **#pdt REMAP FROM** = *cmap_name*, **TO** = *cmap_name*, **DATA** = *data_name*, **LENGTH** = *length*

PRINT_MAPS_stmt ::= **#pdt PRINT_MAP_TABLE**

CLEAR_MAPS_stmt ::= **#pdt CLEAR_MAP_TABLE**

cmap_name ::= *identifier*

data_name ::= *identifier*

int_dimensions ::= [*integer* [[, *integer*]]*]

length ::= *int_expr*

standard_map ::= { **PLANAR** | **LINEAR** | **ALTERNATE** | **DEPTHFIRST_PLANAR** | **DEPTHFIRST_LINEAR** | **DEPTHFIRST_ALTERNATE** | **PLANAR_RM** | **LINEAR_RM** | **ALTERNATE_RM** | **DEPTHFIRST_PLANAR_RM** | **DEPTHFIRST_LINEAR_RM** | **DEPTHFIRST_ALTERNATE_RM** }

mapping_vector ::= ([[**A** : *section* |]] **C** : *section* | **R** : *section*)

section ::= *mapping_vector_element**

mapping_vector_element ::= [[~]] *significance* [[∧ *int_dimension*]]

significance ::= *integer* [[... *integer*]]

int_dimension ::= *integer*

12.3 The statements handling variable mappings

COMMON_MAP_stmt ::= { **#pdt COMMON_MAP** *vmap_name* [[, *vmap_name*]]* | **#pdt COMMON_MAP** *vmap_name* = *int_dimensions standard_map* | **#pdt COMMON_MAP** *vmap_name* = *mapping_vector* | **#pdt COMMON_MAP** *vmap_name* = ∧ }

DECLARE_MAP_stmt ::= { **#pdt DECLARE_MAP** *vmap_name* [[, *vmap_name*]]* | **#pdt DECLARE_MAP** *vmap_name* = *int_dimensions standard_map* | **#pdt DECLARE_MAP** *vmap_name* = *mapping_vector* | **#pdt DECLARE_MAP** *vmap_name* = ∧ }

DYNAMIC_REMAP_stmt ::= **#pdt DYNAMIC_REMAP FROM** = *vmap_name*, **TO** = *vmap_name*, **DATA** = *data_name*, **LENGTH** = *length*

vmap_name ::= *identifier*

int_dimensions ::= [*integer* [[, *integer*]]*]

standard_map ::= (see definitions in section 12.2)

12.4 The LET statements

LET_stmt ::= { **#pdt LET** *vmap_name*[ ] = *vmap* | **#pdt LET** *vmap_name*[*index*] = *vmap_element* | **#pdt LET** *int_variable* = *scalar_function* }

vmap ::= { *vmap_name*[ ] | *var_dimensions standard_map* | ∧ }

vmap_element ::= { [[~]] *int_expr* ∧ *int_expr* | [[~]] *vmap_name*[*index*] | ∧ }

scalar_function ::= { *qualifier vmap_name*[*index*] | **INDEX_OF** *int_expr* ∧ *int_expr* **IN** *vmap_name* }

var_dimensions ::= [*int_expr*[[, *int_expr*]]*]

qualifier ::= { **INVERSION_OF** | **SIGNIFICANCE_OF** | **DIMENSION_OF** }

index ::= *section_identifier int_expr*

section_identifier ::= { **R:** | **C:** | **A:** | **RCA:** }

12.5 The generator statements

WITH_stmt ::= **#pdt WITH DATA** = *data_name*, **LENGTH** = *length*, *matrices*
[[[[**CHANGE_MAP** *vmap_name*]] *generator_sequence**]]

CHANGE_MAP_stmt ::= **#pdt CHANGE_MAP** *vmap_name generator_sequence**

GENERATOR_stmt ::= **#pdt** *generator_sequence**

data_name ::= *identifier*

length ::= *int_expr*

matrices ::= { **MATS** = *num_mats* | **LOG_MATS** = *log_mats* }

vmap_name ::= *identifier*

generator_sequence ::= { *exchange_sequence* | *inversion_sequence* }

exchange_sequence ::= **E** (*efield* ; *efield*)

inversion_sequence ::= **I** (*ifield*)

ifield ::= *section_identifier selector*

efield ::= *section_identifier* [[~]] *selector*

section_identifier ::= { **R:** | **C:** | **A:** | **RCA:** }

selector ::= { *int_expr* | *integer* [[...*integer*]] }

num_mats ::= *int_expr*

log_mats ::= *int_expr*

12.6 The COPY_DATA statement

COPY_DATA_stmt ::= **COPY_DATA FROM** = *data_name* [[**OFFSET** *int_expr*]],
TO = *data_name* [[**OFFSET** *int_expr*]],
PLANES = *int_expr*
[[, **MASK** = *mask_expr*]]
[[, **SHIFT** = *dirn_geom int_expr*]]

data_name ::= *identifier*

mask_expr ::= { *logical_matrix_variable* | *FORTRAN_PLUS_logical_matrix_expression* }

dirn_geom ::= { **NC** | **EC** | **SC** | **WC** | **NP** | **EP** | **SP** | **WP** }

12.7 The basic non-terminals

int_expr ::= { *integer* | *integer_scalar_variable* | *FORTRAN_PLUS_integer_scalar_expression* }

logical_matrix_variable ::= *identifier*

integer_scalar_variable ::= *identifier*

integer ::= *digit**

identifier ::= *alphabetic_char alphanumeric_or_underline_char**

Chapter 13

The standard mappings

You can initialise data mappings, and in the case of variable mappings assign values to them, by reference to one of a number of mapping strategies ('standard mappings') known to the PDT software. In order to initialise or assign a data mapping in this way, you have to specify the dimensions of the data and the name of the standard mapping in any of the PDT statements described in sections 5.1.2, 6.1.1, 6.1.2 and 7.1 of this manual.

There are three classes of standard mappings: PLANAR, LINEAR and ALTERNATE. Descriptions of these mapping strategies are given in the first three sections of this chapter. A list of the mapping vectors describing the various standard mappings of a 128 x 128 data array (on a DAP 500) is given in section 13.4.

13.1 The PLANAR standard mappings

The PLANAR mapping and its variants describe two or higher dimensional arrays only. All the dimensions of the arrays must be powers of two and each of the first two dimensions must not be less than the DAP edge-size.

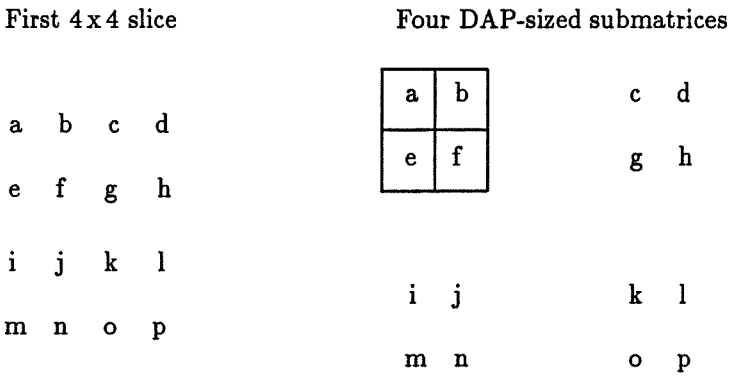
The PLANAR class of standard mappings include:

- PLANAR

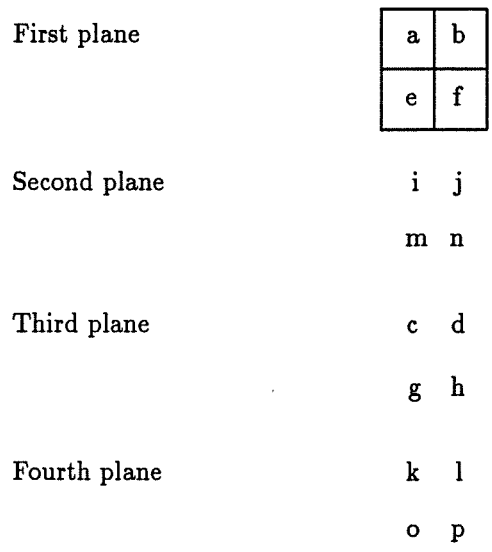
  This mapping is often referred to as 'sheet' mapping. It is obtained for a data array of dimensions $d_1 \text{x} d_2 \text{x} \ldots$ by tessellating each $d_1 \text{x} d_2$ 'slice' of the array into DAP-sized submatrices and then, taking these submatrices in column-major order, mapping them on to successive 'planes' in the DAP memory. (Note that each 'plane' is of a depth equal to the precision of the data.) Successive 'slices' of the array are taken consecutively and mapped in the same manner on to successive 'planes' in the DAP memory.

  Example

  Consider the PLANAR mapping of a 4 x 4 x 2 data array on a hypothetical 2 x 2 DAP. Each 4 x 4 slice is tessellated into four 2 x 2 DAP-sized submatrices; the first slice is tessellated as follows:



The submatrices are then taken in column-major order and mapped on to successive planes in the DAP memory:



The (1,1) submatrix of the first 4x4 'slice' has been highlighted to show the relative positions of the data elements in this mapping in contrast with those in the DEPTHFIRST_PLANAR mapping of the same data array (given later in this section).

The second slice of the array is tessellated and mapped in the same way on to the next four planes in the DAP memory

- PLANAR_RM

  This mapping is similar to PLANAR, the only difference is that the DAP-sized submatrices are taken in row-major order

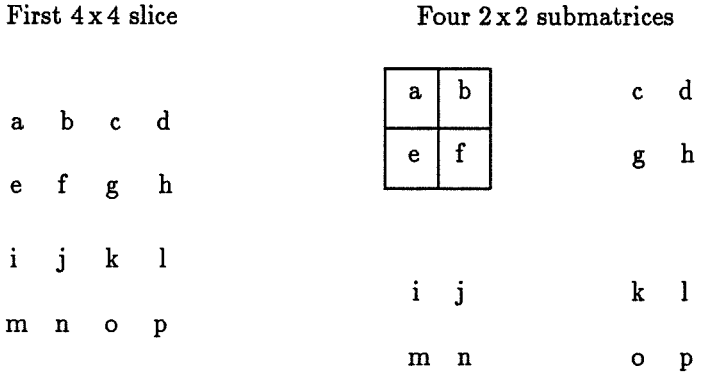
- DEPTHFIRST_PLANAR

  This mapping is often referred to as 'crinkled' mapping. It is obtained for an array of dimensions $d_1 \text{x} d_2 \text{x} \ldots$ by 'crinkling' each $d_1 \text{x} d_2$ 'slice' of the array into DAP-sized vertical mode matrices and mapping these matrices into successive 'planes' in the DAP memory. (Note that each 'plane' is of a depth equal to the precision of the data.) To 'crinkle' a $d_1 \text{x} d_2$ matrix on a $d \text{x} d$ DAP, the $d_1 \text{x} d_2$ matrix is tessellated into $(d_1/d)\text{x}(d_2/d)$ submatrices (so

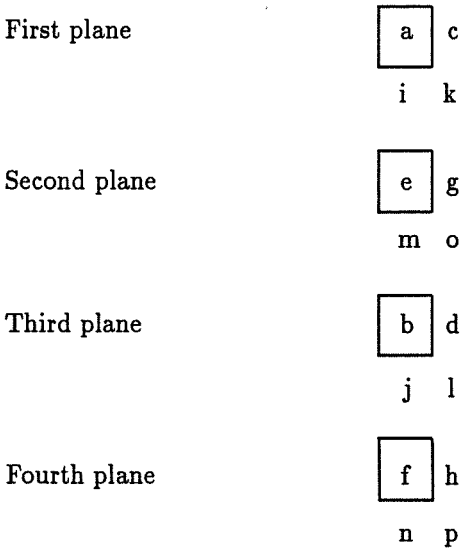
there are dxd of these submatrices). Successive elements of each submatrix, taken in column major order, are then mapped, in vertical mode, at the same position in successive DAP memory planes. Hence the n^{th} element in the $(i,j)^{th}$ submatrix of the first d_1xd_2 slice is mapped in the $(i,j)^{th}$ position of the n^{th} DAP memory plane. Successive d_1xd_2 'slices' of the array are taken consecutively and mapped in memory in this manner.

Example

Consider the DEPTHFIRST_PLANAR mapping of a 4x4x2 data array on to a hypothetical 2x2 DAP. Each 4x4 slice is tessellated into four (given by (4/2)x(4/2)) submatrices. The first slice is as shown below:



The elements of each submatrix are taken in column-major order and then mapped in vertical mode in successive DAP-sized planes as follows:



Note that elements from the same submatrix are mapped in the same position in successive planes in DAP memory. For example, the elements in the (1,1) submatrix, a, e, b and f, of the first 4x4 slice is mapped in the (1,1) positions of the first four memory planes.

The second slice of the array is tessellated and mapped in the same way on to the next four planes in the DAP memory

- DEPTHFIRST_PLANAR_RM

  This mapping is similar to DEPTHFIRST_PLANAR except that the elements of each (d_1/d)x(d_2/d) submatrix are taken in row-major order

13.2 The LINEAR standard mappings

The LINEAR class of standard mappings can describe arrays with any number of dimensions provided the dimensions are powers of two and the total number of data elements is not less than the number of PEs on the DAP. The data elements are taken in an order which varies the first subscripts most rapidly (that is, in conventional FORTRAN ordering). For example, the elements of an nxm data array are taken in the order: (1,1), (2,1), ... (n,1), (1,2), (2,2), ... (n,2) and so on. The elements of the data array are mapped in the following ways:

- LINEAR

  The data elements are mapped along the columns and then along the rows of PEs (that is, in column-major order) and in the same manner over successive planes in the DAP memory.

  Example

  Consider the LINEAR mapping of 32 data elements on to a hypothetical 4x4 DAP in a LINEAR mapping. The data elements will be taken in the conventional FORTRAN order and mapped on to the DAP memory in the following manner:

First plane	0	4	8	12
	1	5	9	13
	2	6	10	14
	3	7	11	15
Second plane	16	20	24	28
	17	21	25	29
	18	22	26	30
	19	23	27	31

- LINEAR_RM

  The data elements are mapped along the rows and then along the columns of PEs (that is, in row-major order) and in the same manner over successive planes in the DAP memory.

Example

Consider the LINEAR_RM mapping of 32 data elements on to a hypothetical 4x4 DAP. The data elements will be taken in the conventional FORTRAN order and mapped on to the DAP memory in the following manner:

First plane	0	1	2	3
	4	5	6	7
	8	9	10	11
	12	13	14	15
Second plane	16	17	18	19
	20	21	22	23
	24	25	26	27
	28	29	30	31

- DEPTHFIRST_LINEAR

In this mapping, groups of n successive data elements are mapped consecutively down the memory of each PE, where n is the total number of data elements divided by the number of PEs. These groups are mapped along the columns and then along the rows of PEs (that is, in column-major order).

Example

Consider the DEPTHFIRST_LINEAR mapping of 32 data elements on to a hypothetical 4x4 DAP. The data elements will be taken in the conventional FORTRAN order and mapped on to the DAP memory in the following manner:

First plane	0	8	16	24
	2	10	18	26
	4	12	20	28
	6	14	22	30
Second plane	1	9	17	25
	3	11	19	27
	5	13	21	29
	7	15	23	31

Note that groups of 2 (32 data items divided by 16 PEs) successive elements are mapped consecutively down the memory of the PEs

- DEPTHFIRST_LINEAR_RM

  This mapping is similar to DEPTHFIRST_LINEAR, but the groups of n data elements are mapped along the rows and then along the columns of PEs (that is, in row-major order)

13.3 The ALTERNATE standard mappings

The ALTERNATE mappings and their variants can describe arrays with any number of dimensions provided the dimensions are powers of two and the total number of elements is not less than the number of processing elements on the DAP. The ALTERNATE class of mappings is important in reducing the amount of routing in applications where there is more interaction between those elements whose indices differ less. The elements are taken in an order which varies the first subscripts most rapidly (that is, conventional FORTRAN ordering) and mapped in the following ways:

- ALTERNATE

  In this mapping, the data elements are mapped alternately on to the rows and columns of the PE array. The basic building blocks for this mapping are made up of 4 consecutive elements.

  The first four elements are mapped on the PE array in the following fashion:

  0 2

  1 3

  Note that the pattern in which the elements are mapped is the same for each basic block of 4 elements.

  On the next level, bigger blocks made up of 4 of these basic blocks are mapped on to the PE array in the same pattern. So, the first of these bigger blocks looks like:

  0 2 8 10

  1 3 9 11

  4 6 12 14

  5 7 13 15

  Four of these bigger blocks will then make up the next size block in the same pattern:

0	2	8	10	32	34	40	42
1	3	9	11	33	35	41	43
4	6	12	14	36	38	44	46
5	7	13	15	37	39	45	47
16	18	24	26	48	50	56	58
17	19	25	27	49	51	57	59
20	22	28	30	52	54	60	62
21	23	29	31	53	55	61	63

Then four of these 64-element blocks will make up a 16 x 16 block and so on, until a DAP-sized 'plane' is obtained. (Note that each 'plane' is of a depth equal to the precision of the data.) The same mapping pattern is repeated for successive 'planes' in the DAP memory as necessary

- ALTERNATE_RM

  This is similar to ALTERNATE except that the data elements are mapped alternately, down the columns first and then down the rows of the PE array. The first four elements are therefore mapped in this pattern:

0	1
2	3

  A bigger block is then constructed with four of basic blocks each comprising 4 successive elements:

0	1	4	5
2	3	6	7
8	9	12	13
10	11	14	15

  As in the ALTERNATE mapping this same pattern is repeated to build up a DAP-sized 'plane'. (Note that each 'plane' is of a depth equal to the precision of the data.) Successive elements are mapped on to successive 'planes' in the DAP memory by repeating the pattern for each 'plane'

- DEPTHFIRST_ALTERNATE

  In this mapping, groups of n successive data elements are mapped consecutively down the memory of each PE, where n is the total number of data elements divided by the number of PEs. These groups are mapped alternately, first down the columns and then down the rows of the PE array (that is, in a similar manner to ALTERNATE mapping)

- DEPTHFIRST_ALTERNATE_RM

  In this mapping, groups of n successive data elements are mapped consecutively down the memory of each PE, where n is the total number of data elements divided by the number of PEs. These groups are mapped alternately, first down the rows and then down the columns of the PE array (that is, in a similar manner to ALTERNATE_RM mapping)

13.4 Standard mappings describing a 128x128 data array

To compare and contrast the various standard mappings described in the previous sections, consider the general example of a 128 x 128 data array on a DAP 500. The mapping vectors representing the standard mappings for this data array are given in Table 12.1.

Standard mapping	*Mapping vector*
PLANAR	(A: 6...5∧2 6...5∧1 | C: 4...0∧2 | R: 4...0∧1)
PLANAR_RM	(A: 6...5∧1 6...5∧2 | C: 4...0∧2 | R: 4...0∧1)
DEPTHFIRST_PLANAR	(A: 1...0∧2 1...0∧1 | C: 6...2∧2 | R: 6...2∧1)
DEPTHFIRST_PLANAR_RM	(A: 1...0∧1 1...0∧2 | C: 6...2∧2 | R: 6...2∧1)
LINEAR	(A: 6...3∧2 | C: 2...0∧2 6...5∧1 | R: 4...0∧1)
LINEAR_RM	(A: 6...3∧2 | C: 4...0∧1 | R: 2...0∧2 6...5∧1)
DEPTHFIRST_LINEAR	(A: 3...0∧1 | C: 6...2∧2 | R: 1...0∧2 6...4∧1)
DEPTHFIRST_LINEAR_RM	(A: 3...0∧1 | C: 1...0∧2 6...4∧1 | R: 6...2∧2)
ALTERNATE	(A: 6...3∧2 | C: 2∧2 0∧2 5∧1 3∧1 1∧1 | R: 1∧2 6∧1 4∧1 2∧1 0∧1)
ALTERNATE_RM	(A: 6...3∧2 | C: 1∧2 6∧1 4∧1 2∧1 0∧1 | R: 2∧2 0∧2 5∧1 3∧1 1∧1)
DEPTHFIRST_ALTERNATE	(A: 3...0∧1 | C: 6∧2 4∧2 2∧2 0∧2 5∧1 | R: 5∧2 3∧2 1∧2 6∧1 4∧1)
DEPTHFIRST_ALTERNATE_RM	(A: 3...0∧1 | C: 5∧2 3∧2 1∧2 6∧1 4∧1 | R: 6∧2 4∧2 2∧2 0∧2 5∧1)

Table 13.1 Standard mappings for a 128 x 128 data array on a DAP 500

Appendix A

Compile-time error messages

The following error message may be generated by the PDT preprocessor at compile-time:

Error in #pdt statement at line < n > of < *filename* >

where < n > is the line number and < *filename* > is the name of the file in which the error occurs.

The following error messages will appear as FORTRAN-PLUS comments, together with the above message, only if you specify the full listing option when you invoke the FORTRAN-PLUS compilation system:

A-section defined twice
C-section defined twice
C-section of mapping vector is incorrect
C-section of mapping vector is too large
Data identifier expected
'DATA' not specified
Dimension of mapping vector element out of range
Expression too long
First dimension is too small 'FROM' mapping not specified
Identifier too long
Incompatible selectors in mapping vector exchange
Index of mapping vector element is out of range
Invalid dimension
Invalid direction/geometry code
Invalid keyword
Invalid section identifier
Invalid section identifier in mapping vector
Length must be positive
'LENGTH' not specified

MATS must be a power of two
'MATS' not specified
Map table full
Mapping not defined
Mapping description incorrect
Mapping identifier expected
Mapping is not initialised
Mapping may not be initialised
Mapping name missing
Mapping vector element repeated
Mapping vector incomplete
Mapping vector is too large
Mapping vector section identifier expected
Mapping vector too large
Mapping vectors incompatible
Missing ')' in expression
Missing ')' in logical expression
Missing operand in expression
Name of a mapping vector variable expected
Name of mapping expected
Name of standard mapping missing
NO ENTRIES IN MAP TABLE
Not a standard mapping
Number too large
PDT command missing
Planar standard mappings require at least 2 dimensions
'PLANES' not specified
R-section defined twice
R-section of mapping vector is incorrect
R-section of mapping vector is too large
Repeated item
Scalar function expected on right hand side
Second dimension is too small
Significance of mapping vector element out of range
Spurious characters at end of statement
Syntax error - symbol expected was: < *symbol* >
Syntax error
This mapping requires at least 2 dimensions
'TO' mapping not specified
Too few elements in mapping
Too many dimensions
Unknown PDT command

where < *symbol* > is the symbol (for example, ∧) expected by the preprocessor.

If either of the following error messages occurs, try re-running the program; if the error persists contact your AMT representative:

PDT System Error 01 - please notify AMT
PDT System Error 02 - please notify AMT

Appendix B

Run-time error messages

The following error messages, with their STOP instructions, may be reported by the PDT run-time software:

- STOP 2001

  will be generated if a request has been made to access a mapping vector element whose index is out of range (see sections 7.2 and 7.3)

- STOP 2002

  will be generated if the function LOG2 or POW2 is called with an argument which is out of range (see section 7.5)

- STOP 2003

  will be generated if the number of matrices specified for a mapping exceeds 2^{30} (see sections 6.2.1 and 8.1)

- STOP 2004

  will be generated if the dimensions specified for standard mappings which are assigned to variable mappings are out of range (see chapter 12)

- STOP 2005.

  will be generated if undefined elements are detected in the C- or R-section of a mapping vector in a DYNAMIC_REMAP statement (see section 6.2)

Index