Cambridge Parallel Processing

DAP Series

Fortran-Plus

(enhanced)

(man102.01)

Technical publication man102.01
(CPP filenames: pubs\latex\fpe\ed1\...)

First Edition: 1 October 1990

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

Publications Manager
Cambridge Parallel Processing
16755 Von Karman Avenue Suite 120
Irvine
California, 92714, USA

Tel: (714) 261-8901
Fax: (714) 261-8802

Publications Manager
Cambridge Parallel Processing Ltd.
Centennial Court, Easthampstead Road
Bracknell, Berkshire, RG12 1JA
England, United Kingdom

Tel: 0114 344 861 024
Fax: 0114 344 305 544

Preface

This manual describes the enhanced implementation of FORTRAN-PLUS, the version of the FORTRAN Language implemented for the AMT DAP. The DAP (standing for 'Distributed Array of Processors') is a massively parallel computer which attaches to a host computer as a peripheral processor. FORTRAN-PLUS enhanced has removed the restrictions in earlier versions of the language on the sizes of parallel dimensions of data objects.

FORTRAN-PLUS enhanced contains many new features which supersede some of the features of earlier versions. However, for the sake of compatibility, all superseded features are still available and referred to in this manual.

The manual you are now reading is a reference document. The manual is organised so that chapters 1 to 5 provide a brief summary of the syntax of FORTRAN-PLUS and are a source of quick reference. Subsequent chapters describe in greater detail the syntax and semantics of FORTRAN-PLUS enhanced procedures, data manipulation, control, computational error management and the process of executing a FORTRAN-PLUS enhanced program. Appendix A describes how FORTRAN-PLUS enhanced data is mapped on the DAP. Appendix B refers to the superseded features of the language.

You are assumed to be familiar with the principles of FORTRAN-PLUS enhanced, as described in the AMT publication:

DAP Series: Introduction to FORTRAN-PLUS enhanced	man101.02

The compilation and execution of FORTRAN-PLUS programs are described in the AMT publication *DAP Series: Program Development*; for further information you should consult the version of that publication which is appropriate to your system.

For convenience, throughout this manual, FORTRAN-PLUS enhanced is referred to simply as FORTRAN-PLUS and the term FORTRAN is used as an abbreviation for FORTRAN 77.

The publication you are now reading is the first edition.

Other relevant publications are:

DAP Series: Program Development under UNIX	man003.03
DAP Series: Program Development under VAX/VMS	man004.03
DAP Series: APAL Language	man005.03
DAP Series: DAP System Calls	man023.01

Contents

Chapter 1

FORTRAN-PLUS programs

1.1 Introduction

FORTRAN-PLUS enhanced is a high level language with parallel processing facilities for programming the AMT DAP. The enhanced implementation of FORTRAN-PLUS removes restrictions in earlier versions of the language on the sizes of data objects intended for parallel processing; that is, the language now supports the declaration of vectors and matrices with arbitrarily sized parallel dimensions.

FORTRAN-PLUS enhanced has many new features which supersede some of the features of earlier implementations of FORTRAN-PLUS. However, for the sake of compatibility, all superseded features are still available and referred to in this manual.

For convenience, throughout this manual, **FORTRAN-PLUS enhanced** is referred to simply as FORTRAN-PLUS.

The DAP (standing for Distributed Array of Processors) is a massively parallel SIMD-type computer (single instruction, multiple data), which attaches to a host computer as a peripheral processor.

The DAP differs from a conventional serial processor in that it can perform the same operation on many items in the data memory simultaneously. This parallel processing capability is provided by a matrix of processors (*processor elements*, or *PEs*), each of which usually operates on the data in its own local memory.

The processors are arranged in a square matrix; for example 32 × 32 in the case of the DAP 500 range, and 64 × 64 for the DAP 600. The number of processors on one side of the square gives the *edge-size* of the DAP.

The DAP has separate code and data memories. The DAP data memory is the total of the local memories of all the processor elements, and is referred to as the *array memory* or *array store*.

Matrices and vectors that contain more components than there are processors are automatically partitioned into DAP-sized sections for processing. The effect is the same as if the entire data object had been processed simultaneously. For convenience most of this manual refers to such operations as being performed 'in parallel', ignoring any partitioning of matrices or vectors that the compiler might make and that the programmer need not be aware of.

Any program which runs on a DAP is called a DAP program, and is started by user software which runs on the host. The host program is entered first; it controls the start of the DAP program and can transfer data between the host and the DAP using special interface subroutines. The preparation of a DAP program, and its running either on DAP hardware or using the simulation system, is described in *DAP Series: Program Development*.

You can write a DAP program in FORTRAN-PLUS or in the assembly level language APAL or in a mixture of the two. The host program can be written in any language or mixture of languages supported by the host operating system.

Facilities also exist for a DAP program to read or write host filestore, and to communicate with peripherals connected directly to the DAP; consult *DAP Series: DAP System Calls* for more details.

1.2 FORTRAN-PLUS program units

A DAP program can contain one or more FORTRAN-PLUS program units called FORTRAN-PLUS subprograms. There are three classes of FORTRAN-PLUS subprogram:

- **SUBROUTINE**

  These program units begin with a SUBROUTINE statement, followed by lines of FORTRAN-PLUS source, concluded by an END statement.

  Control enters a FORTRAN-PLUS subroutine via a CALL statement naming the subroutine, in the same or another program unit.

  If the calling routine is in the host program, the called routine must be a FORTRAN-PLUS ENTRY subroutine, and the call is made via the special interface subroutine DAPENT (see *DAP Series: Program Development*).

  A FORTRAN-PLUS program unit can call only another FORTRAN-PLUS or APAL program unit; it cannot call a host routine.

  Control leaves the subprogram when a RETURN, CALL or STOP statement, or a function call, is reached

- **FUNCTION**

  These program units begin with a FUNCTION statement, followed by lines of FORTRAN-PLUS source, concluded by an END statement.

  Control enters a FORTRAN-PLUS function via a function reference to the function name in another FORTRAN-PLUS program unit, and leaves the subprogram when a RETURN, CALL or STOP statement, or a function call, is reached

- **BLOCK DATA SUBPROGRAM**

  These program units begin with a BLOCK DATA statement, followed by lines of FORTRAN-PLUS source, concluded by an END statement.

  There are no executable statements in a block data subprogram; its use is to give initial values to data in named COMMON blocks

The term *procedure* is used to refer to a user written FORTRAN-PLUS function or subroutine, and a built-in FORTRAN-PLUS function or subroutine. User written procedures are described in chapter 10 and built-in procedures are described in chapter 11.

1.2.1 Data and program units

Values are transmitted between FORTRAN-PLUS program units in the following ways:

- A function call invokes a function which returns a result (see section 10.1.2)
- A procedure shares values with the FORTRAN-PLUS program unit that invokes it by argument association (see section 10.4.2)
- Values can be communicated via named COMMON blocks (see chapter 12)

You can transmit values between a host program and a FORTRAN-PLUS program only if the values reside in the FORTRAN-PLUS program in named COMMON blocks (see section 12.2.2).

1.2.2 Statement order within program units

The various types of FORTRAN-PLUS statements and their formats are described in chapter 2.

The order of statements in a FORTRAN-PLUS procedure must be as follows:

1. FUNCTION, SUBROUTINE or ENTRY SUBROUTINE statement
2. Optional COMMON, DATA, DIMENSION, EQUIVALENCE, EXTERNAL, GEOMETRY, IMPLICIT, PARAMETER, or type statement(s)
3. At least one executable statement
4. END statement

The order of statements in a FORTRAN-PLUS block data subprogram must be as follows:

1. BLOCK DATA statement
2. At least one COMMON statement and one statement initialising variables and optional DIMENSION, EQUIVALENCE or type statement(s)
3. END statement

Chapter 2

Source program format

A FORTRAN-PLUS source program consists of text input from one or more files. The source can contain *file include statements*, which cause text to be inserted from named files at the point in the source at which they appear. File include statements are not regarded as part of the source program; they are described in *DAP Series: Program Development*.

A FORTRAN-PLUS source program consists of lines of text. The first 72 characters on each line are significant to the compiler; for shorter lines the effect is the same as if they were extended on the right with spaces up to a total of 72 characters. Lines can be longer than 72 characters, in which case any text in character positions 72 to 80 will be printed in any listing output that the compiler produces, but is otherwise ignored; any text in the character positions beyond 80 is ignored by the compiler, and will not be printed in any listing output.

The lines of a FORTRAN-PLUS source program are an ordered set, the order being the order in which they are presented to the FORTRAN-PLUS compiler.

The character positions within a line are called *columns*, and are numbered from 1 to 72. Columns 1 to 5 are used to hold *statement labels* (see section 2.2); column 6 is used to indicate whether or not the line is a *continuation line* (see section 2.1); columns 7 to 72 contain the text of FORTRAN-PLUS *statements* (see section 2.3).

2.1 Lines and statements

A FORTRAN-PLUS statement is written in columns 7 to 72 of up to 20 consecutive lines. The first line of the statement is the *initial line*, and any following lines of that statement are *continuation lines*. Column 6 of the initial line of a statement must contain either a space or zero; column 6 of any continuation line(s) must contain some character other than space or zero (continuation lines of the same statement need not have the same character in column 6). The character set that can be used in a FORTRAN-PLUS program is described in section 3.1.

You can intersperse lines of program text with *comment* or blank lines. A comment line contains the character C in column 1 and any text in columns 2 to 72; you can use comment lines to hold text that you want to appear in the compiler output listing. A blank line contains spaces in columns 1 to 72; you can use it to improve the readability of the compiler output listing by separating blocks of program text. Comment and blank lines can appear before or between continuation lines.

Each FORTRAN-PLUS subprogram has to end with an END statement. An END statement contains the keyword END within columns 7 to 72 of an initial line and spaces in all other columns of that line.

2.2 Statement labels

A *statement label* is a string of from 1 to 5 digits (see section 3.1) that can be written in columns 1 to 5 of the initial line of any statement. Spaces and leading zeros are not significant within statement labels.

Hence the labels:

```
056
5 6
56
```

are equivalent. The order of statement labels within a program is not significant, but no two statements in the same program unit can be given the same statement label.

If you declare a statement label, then you can reference the label in GO TO, arithmetic IF, and DO statements (see chapter 14). A reference to a statement label within these statements tells the compiler that the sequence of execution of statements is to depart from the physical sequence of the statements.

2.3 FORTRAN-PLUS statement format

FORTRAN-PLUS statements are of two main classes: executable and non-executable. An *executable statement* is one that specifies some action to be performed during the execution of the program. A *non-executable statement* gives information to the FORTRAN-PLUS compiler concerning the organisation of data and the structure of the program.

This section gives a brief summary of the syntax of the various forms of FORTRAN-PLUS statement. The summary makes reference to the following syntactic entities which are described elsewhere.

Syntactic entity	*reference*
actual argument	section 10.4.2
arithmetic expression	section 8.2
array name	section 6.2
array declarator	section 6.2
array element name	section 6.4
basic integer constant	section 5.1
built-in subroutine	chapter 11
character expression	section 8.3
constant	chapter 5
data-length	section 4.4
dummy argument	section 10.4.1
function name	section 10.1.2
indexing expression	chapter 7
label	section 2.2
logical expression	section 8.5
name	section 3.2
type	chapter 4
variable name	chapter 6
variable declarator	section 6.1

The conventions used in sections 2.3.1 and 2.3.2 for describing the syntax of the various forms of executable and non-executable statements are:

- Statement keywords are shown in upper case in **BOLD TYPE**
- The names of syntactic entities are shown in lower case *italics*
- Where it helps to avoid ambiguity, a phrase used to denote a single entity is linked with hyphens – for example, *basic-integer-constant*
- Where a statement can be formed in a number of ways, the various forms are listed one underneath another
- Parentheses and commas, where they appear in a statement form, are an essential part of the statement syntax

2.3.1 Executable statements

FORTRAN-PLUS executable statements can be further divided into *assignment statements* and *control statements*. An assignment statement gives an initial or new value to a variable or array element; a control statement determines, in some way, the sequence in which statements are to be executed.

2.3.1.1 Assignment statements

Assignment statements in FORTRAN-PLUS can be either *simple* or *indexed* assignment statements.

A simple assignment statement can have any of the following forms:

- *name = arithmetic expression* where *name* is a variable name of type integer or real
- *name = logical expression* where *name* is a variable name of type logical
- *name = character expression* where *name* is a variable name of type character

An indexed assignment statement can have any of the following forms:

- *name (indexing expression list) = arithmetic expression* where *name* is a variable name or an array name of type integer or real
- *name (indexing expression list) = logical expression* where *name* is a variable name or an array name of type logical
- *name (indexing expression list) = character expression* where *name* is a variable name or an array name of type character

The modes of the left and right-hand sides of a simple or indexed assignment statement have to be assignment-compatible, as defined in Chapter 9. Simple and indexed assignments are described in detail in Chapter 9.

2.3.1.2 Control statements

- **CALL** *name*
  CALL *name (actual-argument-list)*
  - *name* is the name of a user written or built-in subroutine

  see section 10.3

- **CONTINUE**

  see section 14.5.4

- **DO** *label name = start, terminator, increment*
  DO *label name = start, terminator*
  - *label* is a reference to a statement label in the same program unit
  - *name* is the name of an integer scalar variable of data-length 4 bytes
  - *start, terminator*, and *increment* are arithmetic expressions

  see section 14.5.1

- **GO TO** *label*
  GOTO *label*
  GO TO (*label*$_1$, *label*$_2$, ... *label*$_i$, ... *label*$_n$), *arithmetic-expression*
  GOTO (*label*$_1$, *label*$_2$, ... *label*$_i$, ... *label*$_n$), *arithmetic-expression*
  - *label* and each *label*$_i$ are references to statement labels in the same program unit, or in the case of *label*$_i$ zero or null
  - The comma preceding *arithmetic-expression* is optional
  - *arithmetic-expression* must yield an integer scalar object

  see section 14.4.1

- The block IF construct has the following general form:

  IF *(logical-expression*$_1$*)* **THEN**
    IF-block
  ELSEIF *(logical-expression*$_2$*)* **THEN**
    ELSE IF-block
    . . .
    . . .
  ELSE
    ELSE-block
  ENDIF

  - *(logical-expression*$_1$*)* and *(logical-expression*$_2$*)* produce logical scalar results
  - *IF-block*, *ELSE IF-block* and *ELSE-block* consist of zero or more executable statements
  - the **ELSEIF** statement and its associated *ELSE IF-block* can be omitted as can the **ELSE** statement and its associated *ELSE-block*
  - the **ELSEIF** statement and its associated *ELSE IF-block* can be repeated any number of times

  see section 14.4.2

- **IF** *(arithmetic-expression) label*$_1$, *label*$_2$, *label*$_3$
  IF *(logical-expression) executable-statement*
  - *label*$_1$, *label*$_2$ and *label*$_3$ are references to the labels of executable statements in the same program unit, zero, or null
  - *arithmetic-expression* and *logical-expression* must yield scalar objects
  - *executable-statement* can be any executable statement other than a **DO** statement or another **IF** statement

  see section 14.4.3

- **PAUSE** *basic-integer-constant*
  - *basic-integer-constant* is in the range 0 to 99999

  see section 14.3

- **RETURN**

  see section 10.6

- **STOP** *basic-integer-constant*
  - *basic-integer-constant* is in the range 0 to 99999

  see section 14.3

- **TRACE** *basic-integer-constant* ($name_1$, $name_2$, ... $name_i$, ... $name_n$)
  - *basic-integer-constant* is in the range 1 to 5
  - Each $name_i$ is the name of a variable, PARAMETER constant or array

  see section 15.1

2.3.2 Non-executable statements

- **BLOCK DATA** *name*
  BLOCKDATA *name*
  - *name* is the name of a block data subprogram

  see section 12.2.1

- **COMMON** $/name_1/list_1/name_2/list_2/$ ... $/name_i/list_i$ ... $/name_n/list_n$
  - Each $name_i$ is the name of a **COMMON** block
  - Each $list_i$ has the form:

    $item_1$, $item_2$, ... $item_i$, ... $item_m$

    where each $item_i$ can be a variable name, array name, variable declarator or array declarator

  see chapter 12

- **DATA** $name_1/valuelist_1/$, $name_2/valuelist_2/$, ... $name_i/valuelist_i/$, ... $name_n/valuelist_n/$
  - Each $name_i$ is a variable name or an array name
  - Each $valuelist_i$ has the form:
    $value_1$, $value_2$, ... $value_i$, ... $value_m$
    where each $value_i$ is either a constant or a construct of the form:
    *basic-integer-constant * constant*

  see section 6.5

- **DIMENSION** $declarator_1$, $declarator_2$, ... $declarator_i$, ... $declarator_n$
  - Each $declarator_i$ can be a vector or matrix variable declarator, an array declarator, or a function declarator

  see sections 6.1.2, 6.2 and 10.1.2.2

- **END**

  see section 10.1.3

- **EQUIVALENCE** *(namelist*$_1$*), (namelist*$_2$*), ... (namelist*$_i$*), ... (namelist*$_n$*)*
  - Each *namelist*$_i$ has the form:

    name$_1$, *name*$_2$, ... *name*$_i$, ... *name*$_m$

    where m is greater than or equal to 2. Each *namelist*$_i$ is a variable name, array name or array element name

  see section 13.1

- **EXTERNAL** *name*$_1$, *name*$_2$, ... *name*$_i$, ... *name*$_n$
  EXTERNAL *type *data-length* **FUNCTION** *name*$_1$, *name*$_2$, ... *name*$_i$, ... *name*$_n$
  EXTERNAL FUNCTION *name*$_1$, *name*$_2$, ... *name*$_i$, ... *name*$_n$
  EXTERNAL *type* **FUNCTION** *name*$_1$, *name*$_2$, ... *name*$_i$, ... *name*$_n$
  - In the first form of the **EXTERNAL** statement, each *name*$_i$ is the name of a user written subroutine, function, or block data subprogram.
  - In the remaining forms of the **EXTERNAL** statement, each *name*$_i$ is the name of a user written function subprogram

  see section 10.1.4

- *type *data-length* **FUNCTION** *name (dummy-argument-list)*
  FUNCTION *name (dummy-argument-list)*
  type **FUNCTION** *name (dummy-argument-list)*
  - *name* is the name to be given to a user written function subprogram

  see section 10.1.2.1

- **GEOMETRY** *(ns-option, ew-option)*
  GEOMETRY *(ns-option)*
  - *ns-option* and *ew-option* can be either **PLANE** or **CYCLIC** independently

  see section 7.6

- **IMPLICIT** *type *data-length (letters)*
  - A symbolic name that begins with one of *letters* is given the specified type and data-length by default

  see section 6.1.1

- **PARAMETER**(*name*$_1$=*expression*$_1$,*name*$_2$=*expression*$_2$, ...
  ...,*name*$_i$=*expression*$_i$ ...,*name*$_n$=*expression*$_n$)
  - *name*$_i$ is a symbolic name
  - *expression*$_i$ is a constant expression (see chapter 8), the value of which becomes associated with the corresponding symbolic name

  see section 5.2

- **SUBROUTINE** *name*
  SUBROUTINE *name (dummy-argument-list)*
  ENTRY SUBROUTINE *name*
  - *name* is the name to be given to a user written subroutine subprogram

  see chapter 10

- *type *data-length item*$_1$, *item*$_2$, ... *item*$_i$, ... *item*$_n$
  type item$_1$, *item*$_2$, ... *item*$_i$, ... *item*$_n$
  - *type* is one of **INTEGER**, **REAL**, **LOGICAL**, **CHARACTER** or **DOUBLE PRECISION** (DOUBLE PRECISION, although not strictly a FORTRAN-PLUS type, is retained for compatibility with other versions of FORTRAN)
  - If *type* is **LOGICAL** or **DOUBLE PRECISION** the second form must be used
  - Each *item*$_i$ can have any of the following forms:
    variable-name/value-list/
    variable-declarator/value-list/
    array-name/value-list/
    array-declarator/value-list/
    function-name

    where *value-list* has the form:
    value$_1$, *value*$_2$, ... *value*$_i$, ... *value*$_m$

    where each *value*$_i$ is either a constant or a construct of the form:
    *basic-integer-constant * constant*

    You can omit */value-list/*.

    function-name must be the name of a user written function subprogram.

  see sections 6.1.2, 6.2, 10.1.2.1 and 10.1.2.2

Chapter 3

Characters and identifiers

3.1 Characters

3.1.1 The FORTRAN-PLUS character set

You can use the following characters in writing FORTRAN-PLUS programs. They constitute the 76-character set. Other characters can be used only in character constants (see chapter 5); these characters are listed in section 3.1.2.

letters	ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
digits	0 1 2 3 4 5 6 7 8 9
special characters	= * + − / ( ) , . ' $ _ #(or £) and the space character.

Upper and lower case characters are treated as equivalent except within character constants. Letters and digits are referred to collectively as *alphanumeric characters*. The ten digits together with the six letters ABCDEF (or abcdef) are referred to as *hexadecimal digits*, and are the only characters that can appear in hexadecimal constants (see chapter 5).

Spaces can appear without significance anywhere in program text, except in character constants (where they form part of the text) and in keywords. However, certain keywords can include a separating space which has no significance to the compiler – these keywords are BLOCKDATA (BLOCK DATA), DOUBLEPRECISION (DOUBLE PRECISION) and GOTO (GO TO). Intervening spaces are not needed when a keyword (other than FUNCTION) is directly followed by a name or another keyword.

You can use the underscore character only in names (see section 3.2) or character constants, but you cannot make it the first character of a name. The underscore character is ignored by the compiler when used in a name; the following names are therefore equivalent:

```
COUNTONE
COUNT_ONE
count__ONE
```

3.1.2 Other characters

This section lists characters that can appear in character constants in FORTRAN-PLUS program units. The character set is ASCII.

The 76-character set is as follows:

Character	*Hexadecimal*	*Character*	*Hexadecimal*
space	20	Q	51
# (or £)	23	R	52
$	24	S	53
' (apostrophe)	27	T	54
(	28	U	55
)	29	V	56
*	2A	W	57
+	2B	X	58
, (comma)	2C	Y	59
- (minus)	2D	Z	5A
.	2E	_ (underscore)	5F
/	2F	a	61
0	30	b	62
1	31	c	63
2	32	d	64
3	33	e	65
4	34	f	66
5	35	g	67
6	36	h	68
7	37	i	69
8	38	j	6A
9	39	k	6B
=	3D	l	6C
A	41	m	6D
B	42	n	6E
C	43	o	6F
D	44	p	70
E	45	q	71
F	46	r	72
G	47	s	73
H	48	t	74
I	49	u	75
J	4A	v	76
K	4B	w	77
L	4C	x	78
M	4D	y	79
N	4E	z	7A
O	4F		
P	50		

The following characters, in conjunction with those above, constitute the 95-character set:

Character	*Hexadecimal*	*Character*	*Hexadecimal*
!	21	[	5B
”	22	\	5C
%	25	]	5D
&	26	ˆ	5E
:	3A	‘	60
;	3B	{	7B
<	3C	|	7C
>	3E	}	7D
?	3F	~	7E
@	40		

The hexadecimal equivalents are given to facilitate comparison of character expressions (see section 8.4.2). You should note, however, that programs that rely on hexadecimal equivalents will be machine dependent and can be difficult to transfer from one computer system to another. You are recommended to restrict character comparisons to groups such as upper case letters (A to Z) and digits (0 to 9), which have a universally defined sequence.

You are strongly recommended to use in FORTRAN-PLUS programs only the characters that are in the 95 character graphic set defined above.

3.2 Symbolic names

A symbolic name referred to as *name* consists of up to 32 alphanumeric characters, the first of which must be alphabetic. However, a symbolic name that identifies a COMMON block or a user written subroutine, function, or block data subprogram cannot be longer than 30 characters. If compatibility with other versions of FORTRAN is important, symbolic names should be restricted to six characters.

The compiler does not distinguish between upper and lower case letters when they occur in names.

You can use the character $ in a symbolic name, provided that:

- it is not the first character in the name
- the name does not identify a COMMON block or a user written subroutine, function, or block data subprogram

You can use the character _ (underscore) in a symbolic name, provided that you do not make it the first character in the name. The underscore character is ignored in a symbolic name and does not count towards the count of characters within a symbolic name.

You can use a symbolic name to identify a variable, array, function, subroutine, block data subprogram, or a COMMON block. You can also use as a dummy argument, a name which identifies a variable, array, function, or subroutine (see section 10.4.1)

3.2.1 Classes of symbolic names

All FORTRAN-PLUS symbolic names fall into one of the following classes, depending on their context in a FORTRAN-PLUS program unit:

1. Array names
2. Variable names
3. Built-in function names
4. User written function names
5. User written subroutine names
6. Built-in subroutine names
7. COMMON block names
8. Block data subprogram names
9. External names
10. PARAMETER constants

External names (class 9) are external to the program unit in which they appear; they are specified in other program units.

3.2.2 Restrictions on usage of symbolic names

Within a particular FORTRAN-PLUS program unit a symbolic name can only identify a single entity from any one of the classes described in section 3.2.1. There is however, one exception to this rule; you can also use a COMMON block name (class 7) as an array name (class 1) or as a variable name (class 2) or a dummy procedure name (class 4, 5 or 9).

You can use the same symbolic name to identify entities in different classes in different FORTRAN-PLUS program units, with the following exceptions:

- If you use a symbolic name anywhere within a DAP program as a user written function name (class 4), a user written subroutine name (class 5) a block data subprogram name (class 8), a COMMON block name (class 7), or as an external name (class 9), you must not use the name elsewhere in the DAP program to identify another entity in any of these classes (unless you use the name as a dummy argument in a program unit)
- If you use an external name (class 9) you must either associate the name with a user written function name (class 4), or a user written subroutine name (class 5), or associate it with a block data subprogram name (class 8) somewhere within the program

FORTRAN-PLUS arrays, variables, and functions all have a type associated with them (see chapter 4). The type is determined by the first letter of the symbolic name (either by default or by use of the IMPLICIT statement – see section 6.1.1), unless it is explicitly specified in a FUNCTION, EXTERNAL, or type specification statement. The appearance of a name in a type specification statement determines that the name is an array name (class 1), a variable name (class 2), a built-in function name (class 3), a user written function name (class 4) or a PARAMETER constant (class 10) – it cannot belong to any of the other classes. The name of a built-in function (see chapter 8) can be used for the name of a data object or user written function, provided it is explicitly specified in a type specification, FUNCTION or EXTERNAL statement; to avoid confusion you are recommended not to use a built-in function name that way. If you *do* use a built-in function name that way, then you cannot use that built-in function in the procedure concerned.

3.3 FORTRAN-PLUS keywords

FORTRAN-PLUS keywords are words that have a special significance in FORTRAN-PLUS. It is possible to use symbolic names that are the same as these keywords but, to avoid confusion, you are recommended not to do so to. The FORTRAN-PLUS keywords are:

BLOCK	DIMENSION	ENTRY	IMPLICIT	REAL
BLOCKDATA	DO	EQUIVALENCE	INTEGER	RETURN
CALL	DOUBLE	EXTERNAL	LOGICAL	SCALAR*
CHARACTER	DOUBLEPRECISION	FUNCTION	MATRIX*	STOP
COMMON	ELSE	GEOMETRY	PARAMETER	SUBROUTINE
CONTINUE	ELSEIF	GO	PAUSE	THEN
CYCLIC	END	GOTO	PLANE	TO
DATA	ENDIF	IF	PRECISION	TRACE
				VECTOR*

* Note that the keywords SCALAR, MATRIX and VECTOR are retained only for compatibility with previous versions of FORTRAN-PLUS.

Chapter 4

FORTRAN-PLUS data

4.1 Data elements

A FORTRAN-PLUS *data element* has one or more *components* processed in parallel. A data element has an associated *mode*, a *type* and a *data-length*.

- The *mode* of a data element is related to the number and arrangement of components which make up the element.

  FORTRAN-PLUS modes are *scalar*, *vector* and *matrix*. The characteristics of the modes are described in section 4.2

- The *type* of a data element is a category characterised by the nature of the various values a data element can take together with ways to denote and interpret these values.

  FORTRAN-PLUS types are:

Integer	–	representing whole numbers
Real	–	representing an approximation to real numbers
Logical	–	having only the values .TRUE. and .FALSE.
Character	–	representing a single character

- The *data-length* of an element is the number of bytes used to represent a single component of the element. The data-length also determines the range of values an element can take. Data elements of type integer and real can be specified with a wide range of data-lengths

4.2 Modes

4.2.1 Scalar

A FORTRAN-PLUS scalar mode element consists of a single component which can be of type integer, real, logical, or character. Note that all constants are scalar objects.

4.2.2 Vector

A FORTRAN-PLUS vector mode element, normally simply referred to as a vector, consists of a one-dimensional ordered set of components. Every component of a vector can be processed in parallel – hence a vector is said to have a single *parallel* dimension. Alternatively you can process in parallel a single component or a group of components from a single vector. Each component of a vector is a single scalar. A vector mode object can be of type integer, real, logical, or character. The total number of components in a vector is known as its *size*.

The i^{th} component of a vector is said to have an *index* of *i*, where *i* can take any value between 1 and the size of the vector.

Every component of a vector except the first has a *left neighbour*, which is the component with an index one less. Every component of a vector except the last has a *right neighbour*, which is the component with an index one greater. Thus, if *i* is not 1, or the size of the vector, component *i* of a vector has a left neighbour component (*i-1*) and a right neighbour component (*i+1*).

4.2.3 Matrix

A FORTRAN-PLUS matrix mode element, normally simply referred to as a matrix, consists of a two-dimensional ordered set of components. Each component of a matrix is a single scalar. Every component of a matrix can be processed in parallel – hence each dimension of a matrix is described as a *parallel* dimension and a matrix is said to have two parallel dimensions. Alternatively, you can process in parallel a single component or a group of components from a single matrix. A matrix mode object can be of type integer, real, logical, or character.

Each component of a matrix is said to have a *row index* and a *column index*. The general component can be described as component (*i,j*), that is, the component at the intersection of the i^{th} row and the j^{th} column in a matrix.

The range of *i* is said to be the *size* of the *first dimension*, or the number of *rows*. The range of *j* is said to be the *size* of the *second dimension* or the number of *columns*. The *shape* of a matrix is described by the ordered pair of sizes of its first and second dimensions.

Every component of a matrix except those in row 1 (or row *r* where *r* is the maximum in the range of *i*) has a *north* (or a *south)* neighbour, which is the component having the same column index and a row index one less (or one greater). Every component of a matrix except those in column 1 (or column *c* where *c* is the maximum in the range of *j*) has a *west* (or *east*) *neighbour*, which is the component having the same row index and a column index one less (or one greater). Thus, if *i* and *j* are not 1 or the maximum value, component *(i, j)* of a matrix has a north neighbour component *(i-1,j)* a south neighbour component *(i+1,j)*, an east neighbour component *(i, j+1)*, and a west neighbour component *(i, j-1)*.

4.2.4 Size, shape and conforming data

For convenience the term *shape* (see section 4.2.3) is sometimes applied to both vectors and matrices. The shape of a vector means the size of that vector.

Two elements of the same mode and the same shape are said to *conform*.

4.3 Arrays

You can declare *arrays* (see chapter 6) of data elements in FORTRAN-PLUS; arrays are known as scalar, vector or matrix arrays according to the mode of their elements. Note that the elements of an array are not processed in parallel, only the components within each element.

4.4 Types

4.4.1 Integer

Integer data are whole numbers represented in two's complement form. The range of values that can be held is dependent on the number of bytes used to represent the data item; that is the data-length of the item: if n bits (1 byte = 8 bits) are used, the range of permissible values is:

$$-2^{(n-1)} \text{ to } + (2^{(n-1)} - 1).$$

The following table gives the range of permissible values for each data-length:

Data-length in bytes	*Range of values*		
1	-128	to	+127
2	-32,768	to	+32,767
3	-8,388,608	to	+8,388,607
4	-2,147,483,648	to	+2,147,483,647
5	-549,755,813,888	to	+549,755,813,887
6	-140,737,488,355,328	to	+140,737,488,355,327
7	-36,028,797,018,963,968	to	+36,028,797,018,963,967
8	-9,223,372,036,854,775,808	to	+9,223,372,036,854,775,807

4.4.2 Real

Real data are approximate representations of decimal values in one of the ranges -7.24×10^{75} to -5.40×10^{-79} and 5.40×10^{-79} to 7.24×10^{75} (all figures approximate) or of the value zero. Values in the range -5.40×10^{-79} to 5.40×10^{-79} are too small to be represented; if a calculation would result in such a value, then the result value is taken to be zero.

If the number of bits used to represent a real value is n, then:

- Bit zero, the most significant of the n bits, is the sign bit
- Bits 1 to 7 represent the base 16 exponent biased by 64
- Bits 8 to n-1 hold the mantissa, preceded by a conceptual binary point

The magnitude of such a real number is therefore:

$$16^{(exponent-64)} \times mantissa$$

with the sign determined by the sign bit (positive for 0, negative for 1).

Valid real data has a mantissa that is normalised such that the first four bits are not all zero, unless the value itself is zero.

The precision to which real data is held is dependent on the number of bytes used to represent it:

Data-length in bytes	*Number of mantissa bits*	*Potential inaccuracy in mantissa*	*Average approximate precision (decimal digits)*
3	16	0.1×10^{-3}	4
4	24	0.5×10^{-6}	7
5	32	0.2×10^{-8}	9
6	40	0.7×10^{-11}	11
7	48	0.4×10^{-13}	14
8	56	0.1×10^{-15}	16

Note – the third column in this table gives the largest possible inaccuracy introduced by having to represent a real number within the given precision. The inaccuracy is given as a fraction of the actual value represented.

4.4.3 Logical

Logical data are exact representations of truth values. The only values that logical data can take are .TRUE. and .FALSE.. Logical data is represented in one bit; 1 represents .TRUE., and 0 represents .FALSE..

4.4.4 Character

Character data are exact representations of single characters. Any character that is capable of representation on the input device used for program or data can be used as the value of character data. Character data is represented as one byte; lists of valid characters and the corresponding bit patterns, represented as a pair of hexadecimal digits, are given in section 3.1.

4.5 Data-length specifiers

The data-length of an element is the number of bytes used to represent a single component. The data-length can take various values according to the type, as shown in the table below. The data-length is determined by a data-length specifier optionally appended to the type keyword in type specification statements (see chapter 6), FUNCTION statements (see chapter 10) or EXTERNAL statements (see chapter 10).

Type	*Data-length specifier*	*Number of bits used in representation*
INTEGER	*1	8
INTEGER	*2	16
INTEGER	*3	24
INTEGER	*4	32
INTEGER	*I	32
INTEGER	*5	40
INTEGER	*6	48
INTEGER	*7	56
INTEGER	*8	64
REAL	*3	24
REAL	*4	32
REAL	*E	32
REAL	*5	40
REAL	*6	48
REAL	*7	56
REAL	*8	64
CHARACTER	*1	8
LOGICAL	None	1
DOUBLE PRECISION	None	64

- For types INTEGER and REAL, where no data-length specifier is present, a default of *4 (data-length 4 bytes) is assumed
- For type CHARACTER, where no data-length specifier is present, a default of *1 (data-length 1 byte) is assumed
- Note that DOUBLE PRECISION is not a type in its own right, but is identical with type REAL of data-length 8 bytes; that is, REAL*8. It is included for compatibility with other versions of FORTRAN

Chapter 5

Constants

A constant is a data element which has the same value whenever it is accessed by the program. Constants have associated type and mode (see Chapter 4); the type of a constant is determined by the way in which it is written, and its mode is always scalar.

5.1 Types of constant

5.1.1 Integer constants

An *integer constant* can take one of two forms:

1. A *basic integer constant.* This is a string of one or more decimal digits interpreted as a decimal whole number
2. A basic integer constant followed by an *integer exponent*. An integer exponent consists of the letter I followed by a basic integer constant. The value represented by an integer constant of the form:

   n I*m*

   is $n \times 10^m$

An *optionally signed* integer constant is an integer constant optionally preceded by a plus sign or a minus sign.

The range of permissible values of an integer constant (see section 4.3.1) depends on the number of bytes used to represent it. In an arithmetic expression (see section 8.2) an integer constant can be followed by a data-length specifier of the form:

(**data-length*)

where *data-length* is a basic integer constant in the range 1 to 8 and represents the number of bytes used to represent the constant. A data-length specifier is only allowed in the context of an arithmetic expression. Note that the construct (*I) is not a valid data-length specifier for an integer constant.

Where an integer constant is written without a data-length specifier the number of bytes used to represent the constant is normally 4 except:

- In DATA and type specification statements where it is the data-length of the variable or array elements being initialised
- In expression evaluation where it depends on the composition of the expression and the context in which the expression is used (see section 8.2)

The following are examples of valid integer constants:

> 15 6 (representing 156; spaces are not significant within integer constants)
> 3I8 (representing 300,000,000)
> 007

The construct:

> -3

is not a valid integer constant, although it is a valid signed integer constant, and can only be used at points in a program where an integer constant can be optionally signed. The construct:

> 100 (*1)

is an example of a valid primary of an expression.

5.1.2 Real constants

A *real constant* can be:

1. A *basic real constant* denoted by one of:

   .m
   n.
   n.m

   where *n* and *m* are strings of one or more decimal digits which are are interpreted as decimal values. Note that spaces are not significant within real constants

2. A basic real or integer constant followed by a *real exponent.* A real exponent consists of the letter *E* followed by an optionally signed basic integer constant. The value represented by a real constant with an exponent is $n \times 10^m$, where *n* is the value of the basic real or integer constant and *m* is the value of the integer constant exponent

An *optionally signed* real constant is a real constant optionally preceded by a plus sign or a minus sign.

A real constant must be within the range of values defined in section 4.3.2. Note that values in the range -5.40 x 10^{-79} to 5.40 x 10^{-79} (figures approximate) are too small to be represented and are replaced by zeroes.

The precision to which a real constant is held (see section 4.3.2) depends on the number of bytes used to represent it. In an arithmetic expression (see section 8.2) a real constant can be followed by a data-length specifier of the form:

(**data-length*)

where *data-length* is a basic integer constant in the range 3 to 8 and represents the number of bytes used to represent the constant. In all other contexts a data-length specifier is not allowed. Note that the construct (*E) is not a valid data-length specifier for a real constant.

Where a real constant is written without a data-length specifier the number of bytes used to represent the constant is normally 4 except:

- In DATA and type specification statements where it is the data-length of the variable or array elements being initialised
- In expression evaluation where it depends on the composition of the expression and the context in which it is used (see section 8.2)

The following are examples of valid real constants:

1.5	
4.56789E3	(representing 4567.89)
2.1097E-2	(representing 0.021097)
1E2	(representing 100.0)
.47 2	(representing 0.472; spaces are not significant within real constants)

The construct:

0.123456789 (*5)

is an example of a valid primary of an expression.

5.1.3 Logical constants

A *logical constant* can take either of the forms:

.TRUE.
.FALSE.

representing logical truth and falsity respectively.

5.1.4 Character constants

A *character constant* can take either of two forms:

1. The *Hollerith constant*, written as:

   $n\mathrm{H}c_1 c_2 c_3 ... c_i ... c_n$

   where n is a basic integer constant in the range 1 to 512, giving the length of the constant. Each c_i is a single character, and can be any of the characters listed in section 3.1. The value represented by the constant consists of the n characters following the letter H, including any space characters

2. The *apostrophe constant* or *literal constant*, written as:

   $'c_1 c_2 ... c_i ... c_n'$

   where each c_i is either a single character other than ' or two apostrophe characters in consecutive columns, and n is in the range 1 to 512. The value represented by the constant is the character string enclosed within the two apostrophes, except that two consecutive apostrophes are replaced by a single apostrophe

Except for the special case of initialising consecutive components of a character array, or vector or matrix, character constants can only consist of a single character; that is, the Hollerith constant must be of the form:

$1\mathrm{H}c_1$

and the literal constant must be of the form:

$'c_1'$

Space characters are always significant within character constants. You should be aware that the compiler treats each source line as being of length 72 characters regardless of the actual line length (see section 2.1).

The following are valid character constants, although constants of length greater than 1 are only valid when initialising in type specification statements or data statements (see section 6.5, point 3).

```
4HRATE
'RATE'
'JOHN''S' (this has the value JOHN'S)
1HB
'?'
```

5.1.5 Hexadecimal constants

Note – you can only use hexadecimal constants for data initialisation (see chapter 6) and their use is recommended only for initialising large logical arrays.

A *hexadecimal constant* consists of the character # (or £) followed by up to 1024 hexadecimal digits (see section 3.1.1). Each hexadecimal digit is taken as a four bit binary representation of one of the numbers zero to 15. The value of a hexadecimal constant is the bit pattern formed by concatenating the bit patterns represented by each individual hexadecimal digit in the constant.

For example, the constant:

```
#F0
```

represents the bit pattern:

```
11110000
```

5.2 Named constants

You can use the PARAMETER statement to give a symbolic name to a constant which is active in the subprogram in which it is declared. The statement has the form:

PARAMETER(*name*$_1$=*expression*$_1$,*name*$_2$=*expression*$_2$, ...
...,*name*$_i$=*expression*$_i$, ...,*name*$_n$=*expression*$_n$)

where:

- *name*$_i$ is a symbolic name. You can use the symbolic name (a *named constant*) instead of the constant itself in DATA statements and in expressions, but you cannot change its value
- *expression*$_i$ is a constant expression (that is, an expression composed of literals; an expression fully known at compile-time), the value of which becomes associated with the corresponding symbolic name. You are not allowed to use the exponential operator in a PARAMETER statement

A PARAMETER statement is a non-executable statement and you must place it before the first executable statement in a program unit.

You can specify the type and data-length of *name*$_i$ in a preceding type declaration (having the same form as a scalar variable declaration – see section 6.1.2). If you make no such declaration, then the type and data-length of *name*$_i$ is implicit, as for variables – see section 6.1.1.

An example of a PARAMETER statement is:

```
PARAMETER (PI=3.141592564)
```

5.3 Use of constants

You can use constants in DATA and type specification statements to give initial values to variables and arrays. You can use all types of constant in this way (see chapter 6).

You can use unsigned integer or real constants and character and logical constants as primaries of expressions (see chapter 8) and as actual arguments (see section 10.4.2).

You use basic integer constants in STOP, TRACE and PAUSE statements and in declarators.

Chapter 6

Variables and arrays

A *variable* is a data element of a specified type, data-length and mode (see chapter 4). You identify a variable within a particular program unit by giving it a symbolic name - the *variable name*. A variable is thus of *mode* scalar, vector or matrix; of *type* integer, real, logical or character; and one of the range of *data-lengths* defined in chapter 4. If it is a vector or matrix it will also have a *shape* (see section 4.2).

An *array* is a set of a specified number and arrangement of elements each of the same type, data-length, mode and (where relevant) shape. An array is a regular structure with one or more dimensions, each of which can have any given size. You identify an array within a program unit by giving it a symbolic name, the *array name*. Each of the elements within an array is referred to as an *array element* and is identified by an *array element name*, consisting of the array name subscripted to identify a particular element (see chapter 7). An array element can be of scalar, vector or matrix mode and in usage is exactly the same as a variable of the same mode. The term shape (see section 4.2) only applies to vector or matrix variables or to the vector or matrix elements of an array – it is not meaningful to talk about the shape of an array or to say that one array has the same shape as another.

Note the distinction between the term *element* and the term *component* (see section 4.2) which is a single scalar (of type integer, real, logical or character) selected from a matrix or vector or array.

6.1 Declaring variables

6.1.1 Implicit type, data-length and mode specification

You do not need to declare scalar variables explicitly within a FORTRAN-PLUS program unit; the variable name can simply appear in an executable statement within the program unit, provided that you do not use the same name for any other item in the program unit other than a COMMON block. If the program unit is a function subprogram, the *function name* (see section 10.1.2.1) is treated, within the program unit, as a variable name.

You can use the same name as a variable name in another FORTRAN-PLUS program unit, in which case it will refer to a different variable, unless you have associated the variables by COMMON association (see chapter 12) or argument association (see section 10.4.3).

If a variable name appears in an executable statement without an explicit type declaration, the type of the variable is determined by the first letter of the variable name. If this is I, J, K, L, M or N, the variable is assumed to be of type integer, otherwise it is assumed to be of type real, unless the IMPLICIT statement has been used. The data-length of the variable is assumed to be 4 bytes, and the mode is assumed to be scalar.

You can use the IMPLICIT statement to change the default type and data-length associated with the initial letter of a variable name. An IMPLICIT statement applies only to the program unit in which it appears and has no effect on the types of any built-in functions or the types of any variables where type is specified explicitly. The statement has the form:

IMPLICIT *type *data-length (letters)*

where:

- *type* can be INTEGER, LOGICAL, REAL or CHARACTER
- **data-length* indicates the number of bytes used to represent real or integer objects (see section 4.4) and can be omitted. **data-length* cannot be used for logical objects and character objects can only have a data-length of 1.
- *letters* can be a single letter, or a list of letters separated by commas, or a range of letters (which has to appear in alphabetical order, separated by a minus sign). Single letters can be combined with a range

You cannot specify the same initial letter in more than one IMPLICIT statement in the same program unit. If you intend an IMPLICIT statement to affect the type of a name in a PARAMETER statement, the IMPLICIT statement has to come before the PARAMETER statement. The IMPLICIT statement has to appear before the first executable statement in the program unit.

For example:

```
IMPLICIT REAL*3 (A-H,X,Y)
```

6.1.2 Explicit declaration

This section describes explicit declaration of variables – declaration of arrays is described in section 6.2.

You can specify explicitly the type, data-length and mode of a variable only once in a program unit, although you can specify each attribute in separate non-executable statements. You can specify type, data-length and mode in the following ways:

1. In a *type specification* statement of the form:

   *type*data-length* $item_1$,$item_2$, ...

   where *type* and *data-length* are as given in section 4.4.

Each item$_i$ is one of the following:

(a) *name/value-list/*

(b) *vector-variable-declarator/value-list/*

(c) *matrix-variable-declarator/value-list/*

where:

- *name* is the name of the variable
- */value-list/*, which is optional, gives the variable or its components initial values and has the format as described in section 6.5 on the DATA statement.

Vector and matrix declarators show that the variable is a vector or matrix and specify the shape (that is, the size of each parallel dimension) of the variable.

A *vector-variable-declarator* has the form:

*name(*dim$_1$)*

A *matrix-variable-declarator* has the form:

*name(*dim$_1$, *dim$_2$)*

A vector is declared with *dim$_1$* components and a matrix with *dim$_1$* rows and *dim$_2$* columns. **dim$_1$* and **dim$_2$* are referred to as *parallel dimension subscripts* and each occupies a *parallel subscript position*.

Each parallel dimension subscript can be:

- the * symbol followed by an integer constant
- the * symbol followed by an integer scalar variable
- the * symbol followed by a SIZE function call (see sections 10.4.5 and 11.2)
- the * symbol alone (that is, an *assumed size* dimension)

Unless each *dim$_i$* is an integer constant, the size of the vector or matrix is determined only at run-time; the vector or matrix is said to be *dynamically-sized.*

The * symbol indicates a parallel dimension (that is, the object is a vector or matrix). The non-parallel dimensions in an array declaration (see section 6.2) do not have the * symbol.

The * symbol alone indicates an *assumed size* parallel dimension; that is, a parallel dimension whose size is determined in the declaration of the actual variable elsewhere. A vector or matrix with an assumed size dimension must be in the dummy argument list (see section 10.4.1) of the subprogram in which the declarator appears.

An integer scalar variable used as *dim$_i$* has to be of data-length 4 bytes and has to refer either to an item in the *dummy argument list* (see section 10.4.1) of the subprogram in which the array declarator appears or to entries in a COMMON block in that subprogram. If any parallel dimension is of assumed size the declarator has to refer to an object itself in the dummy argument list; at run-time the actual sizes will be determined from those of the passed matrix or vector. Dynamically-sized vectors and matrices cannot appear in COMMON, DATA or EQUIVALENCE statements or be given initial values with a *value list*.

Examples:

```
INTEGER*1 I1/1/,VEC1(*100),VEC2(*N)
REAL MAT1(*50,*500), MAT2(*SIZE(MAT3,1),*SIZE(MAT3,2)),MAT3(*,*)
```

MAT3 has to be a dummy argument because its dimensions are assumed size (that is, each is denoted by the * symbol alone). N is either a dummy argument or an entry in a COMMON block.

2. In a **DIMENSION** statement of the form:

   DIMENSION *declarator*$_1$, ... *declarator*$_n$

   where any *declarator*$_i$ can be either of:

   vector variable declarator
   matrix variable declarator

   as defined above.

   Note that you specify only the mode of the variable in the DIMENSION statement; you determine the type and data-length of the variable either by a type specification statement see 1(a) above, or implicitly (see section 6.1.1).

   You cannot give a variable initial values when declared in this context

3. In a **COMMON** statement of the form:

   COMMON /*name*$_1$/*list*$_1$ ... /*name*$_n$/*list*$_n$

   where each *name*$_i$ is a **COMMON** block name.

   Each *list*$_i$ has the form:

   item$_1$, *item*$_2$, ... *item*$_k$

   and each *item*$_j$ can be any of:

   variable name
   vector variable declarator
   matrix variable declarator

   If you define the mode in a COMMON statement by a vector or matrix declarator then the name of the object cannot appear in a DIMENSION statement, or as a declarator with dimension subscripts in a type specification statement.

   Note that you can specify explicitly only the mode in this context; you determine the type and data-length of the variable either by the appearance of its name (without dimension subscripts) in a type specification statement, or implicitly, using the first letter of its name (see section 6.1.1).

   Vectors or matrices in a COMMON statement cannot be dynamically sized; thus the values of the subscripts in the vector variable declarator or matrix variable declarator have to be integer constants.

   You cannot give a variable initial values when declared in this context. If a name does not appear in a vector or matrix declarator, it is taken as a scalar. It cannot appear in more than one such declarator in a program unit.

   For example:

   ```
   COMMON/GLOBAL/IMAGE(*512,*512), LINE(*512)
   ```

6.2 Declaring arrays

Within a program unit you have to explicitly declare FORTRAN-PLUS arrays . You cannot use an array name as the name of other items in the program unit except as a COMMON block name. You can use an array name as an array name in a different program unit, in which case the name refers to a different array unless you have associated the names by COMMON association (see chapter 12) or argument association (see chapter 10).

You specify in an *array declarator* the number of elements in an array and the mode of each element. An array declarator has one of three forms, depending on whether it is an array of scalars, an array of vectors or an array of matrices:

name(dim_1, dim_2 ...dim_n)	(scalar array declarator)
name(*dim_1,dim_2,...dim_n)	(vector array declarator)
name(*dim_1,*dim_2,dim_3 ...dim_n)	(matrix array declarator)

where:

name is the name of the array and *n* cannot be greater than 7.

A vector array declarator has one parallel subscript and a matrix array declarator has two parallel subscripts. The remaining dimensions are non-parallel dimensions. The form of parallel subscripts is as defined in 6.1.2. The form of non-parallel subscripts must be:

either

an integer constant

or

an integer scalar variable

If any of the non-parallel subscripts is an integer scalar variable, the array is said to be dynamically-sized. The integer scalar variable has to be of data-length 4 bytes and has to be either an item in the dummy argument list or an entry in a COMMON block in that subprogram; in the latter case, the array has to be in the dummy argument list of the subprogram.

The total number of dimensions, parallel and non-parallel, cannot exceed 7. The total number of dimensions is known as the *rank* of the array. The number of *elements* in an array is the product of the non-parallel dimensions. The number of *components* in an array is the product of all the dimensions. The *extent* of a dimension is, in the case of a parallel dimension, the number of components along that dimension, and, in the case of a non-parallel dimension, the number of elements along that dimension.

You determine the type and data-length of each array element by the context in which you place the array declarator:

- a type specification of the form:

  *type *data-length* $item_1$, $item_2$, ... $item_n$

  where *type *data-length* is as described for variable declarations in section 6.1.2.

Each $item_i$ has the form:

array declarator/value list/

where */value list/* is optional and gives initial values to one or more elements of the array (see section 6.5)

- a **DIMENSION** statement of the form:

  DIMENSION *array declarator*$_1$, *array declarator*$_2$,....*array declarator*$_n$

  The type of an array declared only in this context is determined by the first letter of the array name (see section 6.1.1); the data-length of the array is 4 bytes. However, the array name can also appear, without its associated dimensions, in a type specification statement, where you can optionally specify the data-length of the array and give initial values to one or more elements of the array

- a **COMMON** statement of the form:

  COMMON /$name_1$ /$list_1$ ... /$name_n$ /$list_n$

  where each $name_i$ is a **COMMON** block name.

  Each $list_i$ has the form:

  $item_1$, $item_2$, ... $item_k$

  where any $item_j$ can be an array declarator. The type of an array declared only in this way is determined by the first letter of the array name (see section 6.1.1); the data-length of the array is 4 bytes. However the array name can also appear, without its associated dimensions, in a type specification statement, where you can optionally give initial values to one or more elements of the array.

  You cannot have dynamically-sized arrays in COMMON blocks and you cannot initialise such arrays with a value list in a type specification statement or DATA statement

6.3 Use of variables

You use variable names within a program unit in two ways:

1. The *value* of a variable is required; in this case the variable is said to be *selected.* When selected it has to have a defined value. Values are used in expressions (see chapter 8) or are passed as arguments to subroutines and functions (see chapter 10).

   In the case of a vector variable, the value could be the whole vector or one component, in which case it is a scalar. In the case of a matrix it could be the whole variable, or one component in which case it is a scalar, or a row or column (or in some circumstances some other set of components) which can be regarded as a vector. Chapter 7 describes the indexing mechanisms which select values from vectors and matrices

2. The variable is required to be *updated* so the variable name denotes the location of the variable rather than its value. In this case the variable is said to be *referenced*. Updating occurs when a variable is used on the left hand side of an assignment statement. In the case of vector and matrix variables, you can identify for updating either the whole data object or various subsets of its components. Chapter 7 describes the indexing mechanisms you can use to update subsets of vectors and matrices

6.4 Use of array and array element names

You specify a single element (which could be a scalar, vector or matrix) of an array by subscripting the array name as described in chapter 7. The *array element name* thus obtained specifies a single object and in usage is equivalent to a variable name. You can use an array element name in the two contexts of selection and referencing just as described in 6.3.

You use *array names* rather than array element names in certain contexts when you need to specify an entire array. The contexts are DATA statements (see section 6.5), COMMON and DIMENSION statements (see section 6.2), EQUIVALENCE statements (see chapter 13), and actual and dummy argument lists (see section 10.4).

6.5 Initialisation using the DATA statement

You use the DATA statement to give initial values to variables and arrays of any type and mode.

The **DATA** statement has the form:

DATA $name_1$ /$value\text{-}list_1$ /, ... $name_n$ /$value\text{-}list_n$ /

where:

- $name_i$ is the name of a scalar, vector or matrix variable or array of any type
- $value\text{-}list_i$ has the form:

  $value_1$, $value_2$, ... $value_k$

  where each $value_j$ is either a constant of the appropriate type, or a construct of the form:

  *basic-integer-constant * constant*

  where *basic-integer-constant* is non-zero. This form of $value_i$ is equivalent to writing the constant *m* times, where *m* is the value of the *basic-integer-constant*. The constant has to be of the same type as the variable or array to which the constant gives an initial value, or the constant has to be a hexadecimal value. Unless the constant is a hexadecimal value, the number of bytes used to represent it is the data-length of the variable or array being initialised (see chapter 5)

The effect of the DATA statement is to initialise the values in the value lists to the corresponding variables or arrays. If $name_i$ is a scalar variable, $value\text{-}list_i$ should be a single constant; if $name_i$ is a vector or matrix variable or an array, there should, in general, be sufficient items in $value\text{-}list_i$ to initialise all components (see items 1 to 4 on the next page for hexadecimal and character initialisation).

The following considerations apply to initialisation within both DATA statements and type specification statements (see sections 6.1 and 6.2)

- The components of an array are initialised in the order used in reduced rank indexing (see section 7.5); that is, such that lower numbered (or leftmost) dimensions are varied most frequently. In the case of a matrix or matrix array this type of ordering is also known as column major ordering.
- You cannot initialise within a subprogram variables and arrays that appear as dummy arguments in that subprogram
- You cannot initialise in a subprogram dynamically-sized variables or arrays

The considerations which apply to hexadecimal and character initialisation are:

1. You can use a string of hexadecimal digits supplied by one or more hexadecimal constants to initialise a logical scalar, vector or matrix array, or a logical vector or matrix variable (but not a logical scalar variable). A string of n hexadecimal digits is interpreted as a string of $4n$ bits, where each bit is treated as a single logical value (one is .TRUE., zero is .FALSE.). If there are more than $4n$ components in the variable or array, the string is effectively extended to the right with hexadecimal digits having zero value; if there are fewer than $4n$ components, the string is effectively truncated from the right. You are not permitted to mix hexadecimal constants and logical constants (.TRUE. and .FALSE.) when initialising logical arrays or vector matrix variables
2. If you use a hexadecimal constant to initialise character variables or arrays, the constant has to consist of pairs of hexadecimal digits that constitute valid characters as defined in section 3.1. Hexadecimal constants and character constants can be mixed in the same value list
3. The value list for a character variable or array can include a character constant in Hollerith or literal format of data-length greater than one; this is the only case in which a character constant can consist of more than one character. Successive characters in the constant are used to initialise successive scalar array elements or successive vector or matrix components; if the value list does not supply sufficient characters to initialise all components of a character variable or array, remaining components are initialised with the space character
4. When you use a hexadecimal constant to initialise a variable of type other than logical or character, one constant is used to initialise each scalar variable or each scalar array element or each vector or matrix component. If the constant is longer than the variable or component, it is effectively truncated from the left; if the constant is shorter than the variable or component, it is effectively extended with zeros to the left

Chapter 7

Indexing vectors, matrices and arrays

7.1 Introduction

You use *indexing* to identify (that is, *select* or *reference* – see section 6.3) one or more, and possibly all, components of a vector or matrix variable or array element. You also use indexing to identify a single element from an array. A single *indexing construct* can both identify an array element and identify component(s) within that element; that is, part of the indexing construct identifies an element from a vector or matrix array, and the remainder of the construct identifies one or more components of that array element. Note that an indexing construct either selects or references depending on the context in which it is used. *Selection* is associated with *obtaining a value* from a variable or variable array; *referencing* is associated with *updating* variables or variable arrays.

An indexing construct has the general form:

> *name*(*indexing expression*$_1$, ... *indexing expression*$_n$)

where:

- *name* is the name of a vector or matrix variable or array. If the indexing construct appears in a context where a variable or array element is being selected by value rather than referenced for updating (see section 6.3) then *name* can also be a vector or matrix expression in parentheses (see section 8.6)
- An *indexing expression*$_i$ has to be one of the following: null, +, –, or an expression evaluating to an integer scalar, an integer vector, a logical vector, or a logical matrix. However, certain subscript positions can only contain certain types of indexing expression. The permissible combinations of indexing expressions in an indexing construct are described in the relevant sections of this chapter. If the type of the expression is integer then its data-length will be converted to 4 bytes if it is not that already. Indexing expressions in positions corresponding to non-parallel dimensions (see section 7.1.1) must evaluate to an integer scalar
- *n* should be in the range 1 to 7. *n* is normally the same as the number of dimensions in the declaration of the variable or variable array but where an indexing construct selects or

references a matrix variable or a scalar, vector or matrix array, all indexing expressions in the construct can be replaced by a single integer scalar expression; this type of indexing is known as *reduced rank indexing* (see section 7.5)

7.1.1 Parallel subscript positions

In an indexing construct the *parallel* subscript positions are:

- The single subscript position if *name* denotes a vector
- The two subscript positions if *name* denotes a matrix
- The first subscript position if *name* denotes a vector array
- The first two subscript positions if *name* denotes a matrix array

where *name* is the name of a variable.

Any other subscript positions in an indexing construct are *non-parallel* subscript positions.

7.1.2 Selecting indexed variables or array elements

The following types of indexing construct can appear wherever the *value* of a variable or array element is required:

Type of construct	*Selection performed*
Indexed variable	Selects a scalar or vector object from a vector variable (see section 7.2.1), or a scalar, vector or matrix object from a matrix variable (see section 7.3.1)
Indexed expression	Selects a scalar or vector object from a parenthesised vector expression (see section 7.2.1), or a scalar, vector or matrix object from a parenthesised matrix expression (see section 7.3.1)
Array element	Contains sufficient indexing expressions to select a single array element from an array, and can also specify additional indexing expressions to select components from the array element so selected (see section 7.4.1)
Complete array element	Selects a single array element from an array as above, but performs no further selections on the components of the selected array element
Reduced rank construct	Selects a scalar object from a matrix variable or expression or from a scalar, vector or matrix array by replacing all indexing expressions with a single integer scalar expression (see section 7.5.1)

7.1.3 Updating indexed variables or array elements

The following types of indexing construct can appear wherever a variable or array element is to be referenced for updating on the left hand side of an assignment statement:

Type of construct	*Effect*
Indexed variable reference	Some subset of components of a vector or matrix variable are identified for updating (see sections 7.2.2 and 7.3.2)
Array element reference	Sufficient indexing expressions are specified to identify a single element of an array, and further indexing expressions can be specified to identify some subset of components of that array element for updating (see section 7.4.2)
Complete array element reference	A single array element is identified as above, but no further indexing expressions are specified
Reduced rank reference	A single component of a matrix variable or a scalar, vector or matrix array is identified, by replacing all indexing expressions with a single integer scalar expression (see section 7.5.2)

7.1.4 Constraints on index values

Sections 7.2 and 7.3 describe the significance of the various types of index which can appear in parallel subscript positions. The types of index are:

Type of index	*Description*
Null	The index is left blank
Integer scalar expression	The value of an integer scalar expression is denoted by IS in sections 7.2 and 7.3 below. IS should always be between 1 and the size of the dimension of the subscript position in which it occurs, that is, in the indexing construct: `M( ,I+2)` where M is a matrix, I+2 cannot be more than the number of columns of M

Logical vector expression	The value of this is denoted by LV in sections 7.2 and 7.3 below. The number of components in LV has to be the same as the size of the dimension of the subscript position in which this index occurs. Thus, in the example: `M(V1.EQ.V2,)` the value of the logical expression V1.EQ.V2 (and therefore both V1 and V2 – see section 8.5) must have the same number of components as the number of rows in M
Integer vector expression	Applies only in matrix indexing. The value of an integer vector expression is denoted by IV in section 7.3. The number of components in IV has to be equal to the size of the *other* parallel dimension. Thus in the expression: `M(IV,)` IV has to have the same number of components as the number of *columns* in M. Also the value of each component in the integer vector has to be between 1 and the size of the dimension it is in; that is, in the example above, each component in IV has to be between 1 and the number of rows of M
Logical matrix expression	Applies only in matrix indexing. The value of a logical matrix expression is denoted LM in section 7.3. LM has to be the same shape as the matrix being indexed.

7.2 Vector indexing

This section describes the meaning of the various types of index which you can use in the single parallel subscript position of a vector (denoted by V).

7.2.1 Indexed vector selection

When you index a vector variable, or vector expression in parentheses, to obtain a value in an expression, either a single component of the vector object or the entire vector object is selected according to the type of index in the single parallel subscript position:

Type of index	*Selection performed*
Null	The entire vector is selected. The same effect is obtained by specifying the vector name or expression with no following parentheses. For example: V V( )
Integer scalar expression (IS)	The ISth component of the vector is selected as a scalar For example: V(N+2) selects the (N+2)th component of V
Logical vector expression (LV)	LV can have only one .TRUE. component. If i is the index of this component, the ith component of V is selected as a scalar
+	The entire vector V is selected and is shifted one place to the left; that is component i of the selected vector is equal to V(i+1). If vector V has only one component then no shift is done. The value of the last component of the selected vector is determined by the north-south option specified in the local GEOMETRY statement (see section 7.6)
–	As +, but the vector V is shifted one place to the right and the value of the first component is determined by the GEOMETRY statement

7.2.2 Indexed vector referencing

Where you reference a vector for updating on the left hand side of an assignment statement, you can identify zero, one or more components for updating, depending on the type of indexing expression:

Type of indexing expression	*Effect*
Null	The entire vector is referenced for updating. The same effect is obtained by specifying the vector name or expression with no parentheses
Integer scalar expression (IS)	The ISth component of the vector is identified for updating

Logical vector expression (LV)	Each component of the vector which corresponds to a .TRUE. component in LV is identified for updating. Other components are not affected. For example: `V(V.GT.0)` identifies just the positive components of V.

7.3 Matrix indexing

This section describes the meaning of various types of index which you can use in the two parallel subscript positions of a matrix (denoted by M).

7.3.1 Indexed matrix selection

When you select in an expression, a matrix variable, or a matrix expression in parentheses, a scalar, vector or matrix object could be selected according to the type of the indexing expressions in the two parallel subscript positions.

Type of indexing expression	*Selection performed*
Null, Null	The entire matrix is selected. The same effect is obtained by giving the matrix name with no following parentheses. For example: `M(,)` or simply `M`

Null, Integer scalar expression (IS) (Integer scalar expression (IS), Null)	A vector is selected corresponding to the ISth column (row) of M. For example: M(,3) M(I+J,) In the first example a vector which is a copy of the third column of the matrix M is selected. In the second example a vector which is a copy of the (I+J)th row of the matrix M is selected
Null, Logical vector expression (LV) (Logical vector expression (LV), Null)	LV has to have one .TRUE. component. A vector is selected corresponding to the ith column (row) of M, where LV(i) is the .TRUE. component of LV. For example: M(,IV.EQ.3) would be valid if a single component of the integer vector IV had the value 3; in this case, the column of M corresponding to this component would be selected
Null, Integer vector expression (IV) (Integer vector expression (IV), null)	A vector is selected made up of one component from each row (column) of M. The component selected from the ith row (column) is that in the jth column (row) where j=IV(i)
Null, Logical matrix expression (LM) (Logical matrix expression (LM), Null)	LM has to have one and only one .TRUE. component in each row (column). A vector is selected whose ith component is M(i,j) (or M(j,i)) where LM(i,j) (or LM(j,i)) is the single .TRUE. component
Null, + (+, Null)	The entire matrix is selected and shifted one place to the west (north); that is, column (row) i of the selected matrix is equal to column (row) i+1 of M. No shift is performed if the relevant dimension has a size of 1. The value of the last column (row) of the selected matrix is determined by the options specified in the local GEOMETRY statement (see section 7.6)
Null, - (-, Null)	The entire matrix is selected and shifted one place to the east (south); that is, column (row) i of the selected matrix is equal to column (row) i-1 of M. No shift is performed if the relevant dimension has a size of 1. The value of the first column (row) of the selected matrix is determined by the options specified in the local GEOMETRY statement (see section 7.6)

In any of the above cases, you can replace the null indexing expression by a non-null indexing expression, with the following effects:

- If the other indexing expression is either + or –, you can replace the null indexing expression by an integer scalar, logical vector, integer vector, or logical matrix expression. The selections performed by the replacement indexing constructs are as described above, except that the indexed matrix variable or expression is shifted in the appropriate direction before the selections are performed
- If the other indexing expression is either + or –, you can also replace the null indexing expression by + or –, in which case the indexed matrix variable or expression is shifted twice, in the appropriate directions, to select its value. The order in which the shifts are performed is immaterial
- If the other indexing expression is anything other than + or –, you can replace the null indexing expression by either an integer scalar expression or a logical vector expression. The indexing expression that replaces the null indexing expression will perform the appropriate vector indexing (see section 7.2.1) on the vector object selected by the other indexing expression

7.3.2 Indexed matrix referencing

When a matrix, M, is updated on the left hand side of an assignment statement, you can identify the whole or a subset of the components of matrix M for updating, by specifying particular types of index in its two parallel subscript positions.

Type of index	*Effect*
Null, Null	The entire matrix is referenced for updating. The same effect is obtained by giving the matrix name or expression with no following parentheses
Null, Integer scalar expression (IS) (Integer scalar expression (IS), Null)	Components in the ISth column (row) of M are identified for updating. If the expression (on the right hand side of the assignment statement) being used to assign to M is a vector, the vector is implicitly expanded by columns (rows) before indexed assignment is performed; that is, the vector is implicitly expanded to a matrix of the same shape as M and each of whose columns (rows) is a copy of the vector. The size of the vector has to be the same as the number of rows (columns) in M

Null, Logical vector expression (LV) (Logical vector expression (LV), Null)	Components in each column (row) of M corresponding to a .TRUE. component in LV are identified for updating. If the expression (on the right hand side of the assignment statement) being used to assign to M is a vector, the vector is implicitly expanded by columns (rows) before indexed assignment is performed; that is, the vector is implicitly expanded to a matrix of the same shape as M and each of whose columns (rows) is a copy of the vector. The size of the vector has to be the same as the number of rows (columns) in M
Null, Integer vector expression (IV) (Integer vector expression (IV), Null)	One component in each row (column) of M is identified for updating. The component identified from row (column) i is that in column (row) j where j=IV(i). If the expression (on the right hand side of the assignment statement) being used to assign to M is a vector, the vector is implicitly expanded by columns (rows) before indexed assignment is performed; that is, the vector is implicitly expanded to a matrix of the same shape as M and each of whose columns (rows) is a copy of the vector. The size of the vector has to be the same as the number of rows (columns) in M
Null, Logical matrix expression (LM) (Logical matrix expression (LM), Null)	Each component of the indexed matrix variable that corresponds to a .TRUE. component in LM is identified for updating. If the expression (on the right hand side of the assignment statement) being used to assign to M is a vector, the vector is implicitly expanded by columns (rows) before indexed assignment is performed; that is, the vector is implicitly expanded to a matrix of the same shape as M and each of whose columns (rows) is a copy of the vector. The size of the vector has to be the same as the number of rows (columns) in M
Integer scalar expression (IS1), Integer scalar expression (IS2)	The single component M(IS1,IS2) is identified for updating
Integer scalar expression (IS), Integer vector expression (IV) (Integer vector expression (IV), Integer scalar expression (IS))	The single component (IS,j) (component (j,IS)) of M is identified for updating, where j=IV(IS)

Integer scalar expression (IS), Logical matrix expression (LM) (Logical matrix expression (LM), Integer scalar expression (IS))	
	Each component in row (column) IS of M corresponding to a .TRUE. component in the corresponding row (column) of LM is identified for updating
Integer vector expression (IV), Logical vector expression (LV) (Logical vector expression (LV), Integer vector expression (IV))	
	For each i, IV identifies one component from the i^{th} column (row) of M namely the j^{th} component where j=IV(i). Of these components, only those corresponding to .TRUE. values in LV are identified for updating
Logical vector expression (LV1), Logical vector expression (LV2)	
	Components M(i,j) such that LV1(i) and LV2(j) are .TRUE. are identified for updating
Logical matrix expression (LM), Logical vector expression (LV) (Logical vector expression (LV), Logical matrix expression (LM))	
	The logical matrix expression identifies those elements of M corresponding to .TRUE. values in LM. Of those, the components identified for updating are just those in columns (rows) corresponding to .TRUE. components in the logical vector expression

You can identify for updating zero, one, or more matrix variable components by replacing both indexing expressions with a single indexing expression, as follows:

Indexing expression	*Effect*
Logical matrix expression	Each component of the indexed matrix variable that corresponds to a .TRUE. component in the logical matrix expression is identified for updating
Integer scalar expression	A single component of the indexed matrix variable is identified for updating. This is the technique of reduced rank indexing (see section 7.5)

All components identified for updating in any of the indexing constructs described in this section are processed in parallel.

7.4 Array indexing

7.4.1 Indexed array selection

When you index an array to obtain a value, as in an expression or a single array element, you can select a scalar, vector or matrix object that forms part or all of the array element by an indexing construct of the form:

name(indexing expression$_1$, ... *indexing expression*$_n$*)*

where:

- *name* is the name of a scalar, vector or matrix array
- *n indexing expressions* can be supplied, where *n* is the number of subscript positions defined in the array declarator (see section 6.2). *n* cannot be greater than 7.

If *name* refers to a scalar array, all subscript positions have to contain integer scalar expressions which select a particular scalar array element.

If *name* refers to a vector array, the parallel subscript position (that is, the first) can contain any of the indexing expressions described in section 7.2.1; all other (non-parallel) subscript positions have to contain integer scalar expressions. The integer scalar indexing expressions in the non-parallel subscript positions select a particular vector array element. The indexing expression in the parallel subscript position then selects a scalar or vector object from the array element, as described in section 7.2.1.

If *name* refers to a matrix array, the parallel subscript positions (that is, the first two) can contain any of the indexing expression pairs described in section 7.3.1; note that this indexing expression pair can be replaced by a single logical matrix indexing expression. All other (non-parallel) subscript positions have to contain integer scalar expressions. The integer scalar indexing expressions in the non-parallel subscript positions select a particular matrix array element. The indexing expression pair in the parallel subscript positions (or the logical matrix expression in the first subscript position) then selects a scalar, vector, or matrix object from the array element as described in section 7.3.1.

You can replace all indexing expressions in the indexing construct by a single integer scalar expression; this is the technique of reduced rank indexing (see section 7.5). Such a construct selects a single component from a scalar, vector or matrix array.

7.4.2 Indexed array referencing

When you reference an array for updating, as in the left-hand side of an assignment statement, you can identify zero, one, or more components of a single array element for updating by an indexing construct of the form:

name(indexing expression$_1$, ... *indexing expression*$_n$*)*

where:

- *name* is the name of a scalar, vector or matrix array
- *n indexing expressions* can be supplied, where *n* is the number of subscript positions defined in the array declarator (see section 6.2). *n* cannot be greater than 7

If *name* refers to a scalar array, all subscript positions have to contain integer scalar expressions which identify a particular scalar array element.

If *name* refers to a vector array, the parallel subscript position (that is, the first) can contain any of the indexing expressions described in section 7.2.2; all other (non-parallel) subscript positions have to contain integer scalar expressions. The integer scalar indexing expressions in the non-parallel subscript positions identify a particular vector array element for updating. The indexing expression in the parallel subscript position then identifies zero, one, or more components of the array element for updating, as described in section 7.2.2.

If *name* refers to a matrix array, the parallel subscript positions (that is, the first two) can contain any of the indexing expression pairs described in section 7.3.2; note that this indexing expression pair can be replaced by a single logical matrix indexing expression. All other (non-parallel) subscript positions have to contain integer scalar expressions. The integer scalar indexing expressions in the non-parallel subscript positions identify a particular matrix array element for updating. The indexing expression pair in the two parallel subscript positions (or the single logical matrix indexing expression in the first subscript position) then identifies zero, one, or more components of the array element for updating, as described in section 7.3.2.

7.5 Reduced rank indexing

You can replace the set of indexing expressions in an indexing construct by a single integer scalar expression; this is the technique of reduced rank indexing. Such a construct selects or identifies a single component of the scalar vector or matrix array for processing.

The ordering in which the components of an array are stored is such that the first index varies most rapidly, then the second and so on. Thus if the dimensions (parallel or non-parallel) are:

$dim_1, dim_2, ... dim_n$

(see section 6.2) then the positions within the array of an element indexed by:

$(d_1, d_2, ... d_n)$

is given by the formula:

$$d_1 + dim_1 * (d_2-1) + dim_1 * dim_2 * (d_3-1) + ..$$
$$+.. dim_1 * ... * dim_{n-1} * (d_n-1)$$

You can index any array or matrix variable by a single integer scalar expression which replaces all the subscripts. This expression identifies a scalar element. The index is applied to the ordering above. (For matrices and matrix arrays this method of ordering is also referred to as column major ordering.)

Thus, given the declaration:

```
    INTEGER VA(*20,3)
```

then

```
    VA(21)
```

identifies the element VA(1,2)

7.6 The GEOMETRY statement

If either or both parallel subscript positions of a vector or matrix variable or array reference contain the indexing expression + or –, you determine the values shifted in at the end of a vector – or the edge of a matrix – by the **GEOMETRY** statement.

This statement can have either of the following forms:

- **GEOMETRY** (*option*)
- **GEOMETRY** (*ns-option*, *ew-option*)

where *option*, *ns-option*, and *ew-option* can be either PLANE or CYCLIC.

In the first form of the GEOMETRY statement, *option* specifies the type of shift to be used for all vector, north-south matrix and east-west matrix shifting. With the second form, *ns-option* specifies the type of shift for vector or north-south matrix shifting; *ew-option* specifies the type of shift for east-west matrix shifting. *ns-option* and *ew-option* could be different.

A FORTRAN-PLUS procedure can contain only one GEOMETRY statement. If no GEOMETRY statement appears within a procedure, PLANE geometry is assumed for all vector and matrix shifts.

When a shift for which PLANE geometry has been specified is performed, the value(s) shifted in at the end of a vector or at the end or edge of a matrix are as follows:

Type of object shifted	*Value shifted in at vector end or matrix edge*
Integer	0
Real	0.0
Character	Null (hexadecimal 00)
Logical	.FALSE.

When a shift for which CYCLIC geometry has been specified is performed, the value(s) shifted off one end or edge are shifted in at the opposite end or edge.

Chapter 8

Expressions

An *expression* consists of either a single *primary* or a combination of primaries and *operators*. Each primary is an object of a particular type, data-length and mode with a defined value and the operators specify the operations to be performed on these objects in order to obtain the value of the expression.

The primaries of an expression depend on the class of the expression which can be: *arithmetic*, *character*, *relational* or *logical*. The result of an expression has type, data-length, and mode and these also depend on the class of the expression and the objects from which it was formed. These concepts are described in detail in the following sections.

When all primaries are constants, the expression is known as a *constant expression*.

8.1 The mode of an expression

The modes of the primaries in an expression determine the mode of the result of that expression.

The effect of a unary operator on a primary is to give a result of the same mode as the primary.

When a unary operator is applied to a vector or matrix mode primary, the operation is performed in parallel on every component of the primary.

The effect of binary operators on primaries of various modes is summarised in the following table, in which ⊗ can be any binary operator:

Mode of *primary*$_1$	*Mode of* *primary*$_2$	*Mode of* *primary*$_1$⊗ *primary*$_2$	*Remarks*
Scalar	Scalar	Scalar	
Scalar	Vector	Vector	The FORTRAN-PLUS compiler automatically expands the scalar *primary*$_1$ to a vector of the same size as *primary*$_2$, each component of which has the value of *primary*$_1$

Scalar	Matrix	Matrix	The FORTRAN-PLUS compiler automatically expands the scalar *primary*$_1$ to a matrix of the same shape as *primary*$_2$, each component of which has the value of *primary*$_1$
Vector	Scalar	Vector	See scalar-vector
Vector	Vector	Vector	The vectors have to *conform*, that is have the same number of components
Vector	Matrix	–	Invalid combination
Matrix	Scalar	Matrix	See scalar-matrix
Matrix	Vector	–	Invalid combination
Matrix	Matrix	Matrix	The matrices have to *conform*, that is have the same shape

When an expression contains a valid combination of primaries, one of which is a vector or matrix object, the operation represented by the operator is performed in parallel on corresponding components of each primary (if one primary is a scalar, then that primary will have been implicitly expanded to match the other vector or matrix primary).

8.2 Arithmetic expressions

8.2.1 Format

A primary of an arithmetic expression can be:

- An unsigned constant or named constant (see chapter 5)
- A selected variable (see section 6.3)
- A selected array element (see section 6.4)
- A function call (see section 10.2)
- An arithmetic expression enclosed in parentheses

The primaries all have to be of type integer or real, and their modes have to conform to the rules in section 8.1.

The arithmetic operators and the operations that they represent are as follows:

Unary operators	*Effect*	*Binary operators*	*Effect*
+	No effect	+	Addition
–	Negation	–	Subtraction
		*	Multiplication
		/	Division
		**	Exponentiation

An arithmetic expression consists of a single arithmetic primary or a string of arithmetic primaries, each separated from any preceding or following primary by a single arithmetic operator, the whole being optionally preceded, in either case, by one of the operators + or –.

For example:

assuming the declarations:

```
INTEGER I, VI(*100), MI(*100,*50), VIA(*100, 5)
REAL R, MRA(*100,*50, 3)
```

then the following are valid expressions:

```
-23.2
VI - 10 + ( VIA ( ,3 ) / 2 )
R*SIN( MRA ( , ,1 ) ) - MI
```

8.2.2 Type, data-length and mode

The types and data-lengths of the primaries involved in an arithmetic operation determine the type and data-length of the result of that operation. If a primary is a constant which is not followed by a data-length specifier, the primary has a *non-fixed* data-length. Any other type of primary has a *fixed* data-length. The data-length of the result of an arithmetic operation is fixed if the data-length of at least one of the primaries is fixed. If neither primary is of fixed data-length, the data-length of the result is 4 bytes.

If both primaries are of fixed data-length, and one is shorter, then the shorter primary will be lengthened before the operation takes place (see section 8.2.4.1). The result will have the data-length of the longer primary.

For unary operations (+ or –) the type and data-length of the result are the same as the type and data-length of the primary.

For binary operations the type of the result is integer if and only if both primaries are of type integer, otherwise the result is of type real. For binary operations where one of the primaries is of type integer and the other is of type real, the integer primary is converted to type real before the operation is performed – the only case when the conversion does not take place is where the left-hand primary is of type real, the operator is **, and the right-hand primary is of type integer.

If only one of the primaries is of fixed data-length, then the data-length of the result is equal to that of the fixed data-length primary subject to the following constraint: if the fixed data-length primary is an integer of data-length 1 or 2 bytes and the other primary is type real (hence the result is of type real), then the data-length of the result is 3 bytes.

For example, the operation:

```
1(*1)+17.3
```

gives a real result of data-length 3 bytes.

The primary which is not of fixed data-length is converted into a temporary variable to be used only for the purposes of this operation and with the data-length of the result (if it is not 4 bytes). Note that this can cause overflow (for example, in the operation 500+50(*1)) or loss of accuracy (for example, in the operation 1(*1)+1.23456).

You obtain the type and data-length of an expression by applying the above rules to each operation, following the order of evaluation given in section 8.2.3.

You obtain the mode of the result of an expression by applying the rules given in section 8.1 to each operation, following the order of evaluation given in the next section.

8.2.3 Order of evaluation

Arithmetic expressions are evaluated, in general, from the innermost set of parentheses outwards. Within each set of parentheses or within an expression with no parentheses, the order of evaluation is:

1. Evaluation of primaries, including function references, expressions in parentheses, and vector, matrix, or array subscripts. This stage can also include the automatic expansion of scalars into vectors or matrices, as required (see section 8.1)
2. Exponentiations
3. Multiplications and divisions
4. Additions and subtractions

Consecutive exponentiations are evaluated from right to left. Thus, the expressions A**B**C and A**(B**C) have the same meaning, and differ from (A**B)**C.

An exponentiation operates on the primary that immediately precedes it. Thus, the expressions -A**B and -(A**B) have the same meaning, and differ from (-A)**B.

Consecutive multiplications and divisions are evaluated in any order that is mathematically equivalent to evaluation from left to right. Thus, the expression A/B*C could be evaluated as (A/B)*C or as (A*C)/B, but not as A/(B*C).

Where integer divisions are involved, evaluation is always from left to right. For example X*I/J is evaluated as (X*I)/J, and I/J*X is evaluated as (I/J)*X; these expressions can produce different results.

Consecutive additions and subtractions are evaluated in any order that is mathematically equivalent to evaluation from left to right. A leading minus sign (–) also acts according to the rules of precedence.

The order of evaluation of primaries is entirely under the control of the FORTRAN-PLUS compiler. Functions referenced in an expression should not alter the values of any variables or array elements referenced in that expression, otherwise the result of the expression will depend on the order of evaluation of the primaries. If a function is referenced more than once in an expression, the results should be independent of the order of evaluation; in particular, two references with the same argument should produce the same results. Violation of these rules will lead to unpredictable results.

In general, the effects of data-length changing operations and conversions between integer and real types depend on the order of evaluation. In such cases you are advised to use parentheses to force a particular order.

8.2.4 Arithmetic considerations

8.2.4.1 Data-length conversion

When the data-length of an integer object is increased its representation is extended at the most significant end with copies of the sign bit. When the data-length of an integer object is decreased its representation is truncated at the most significant end until the correct data-length is obtained. Overflow will occur if anything other than copies of the sign bit are truncated, or if the sign bit of the truncated object differs from its original value.

When the data-length of a real object is increased its mantissa is extended at the least significant end with zero bits. When the data-length of a real object is decreased it is truncated at the least significant end of the mantissa; the value is rounded up or down depending on whether the most significant truncated bit is 1 or 0.

8.2.4.2 Integer arithmetic

An integer result is defined to be the integer with the greatest magnitude not exceeding the magnitude of the mathematically correct value and having the same sign as the mathematically correct value (except that the sign bit is zero where the result is the integer zero); thus the result is effectively truncated at the decimal point.

For example:

15/4	gives the result	3
– 15/4	gives the result	–3

8.2.4.3 Exponentiation

When evaluating the expression A**B the following restrictions apply:

- A and B cannot both be zero
- If A is negative B has to be of type integer
- If A is of type integer B cannot be a negative integer

8.2.4.4 Computational errors and overflow

If your program attempts to perform an inappropriate operation, such as division by zero, or if an operation cannot be completed (for example, arithmetic overflow or underflow occurs during the evaluation of an expression), a *computational error* occurs. Facilities for handling computational errors are described in chapter 15; the effects of computational errors themselves are described in *DAP Series: Program Development*.

Note that you can treat arithmetic underflow separately from other computational errors; underflowed results are set to zero.

Arithmetic results will overflow, causing an error, if the result of a calculation has a magnitude which is out of range. Sections 4.3.1 and 4.3.2 give details of the ranges of integer and real values.

Overflow will occur if the value of an expression is too large, and can also occur if some intermediate result in the expression is too large. You can sometimes overcome this latter problem by using parentheses to control the order in which parts of the expression are evaluated. For example, if the value of the expression:

```
A/C*B
```

is required, and A,B, and C are all very large numbers, you can avoid overflow (but only if the final result is in range) by writing the expression as:

```
(A/C)*B
```

thus forcing the division to be performed first.

You can suppress computational errors by using the FORTRAN-PLUS computational error management facilities (see chapter 15). There are also compile-time options to ignore computational errors – see *DAP Series: Program Development* relevant to your host.

8.2.5 Use of arithmetic expressions

You can use arithmetic expressions in the following contexts:

- The right-hand side of an arithmetic assignment statement can be an arithmetic expression. The value of the arithmetic expression is assigned to an integer variable, real variable or array element. The type and data-length of the result of the expression can be altered prior to assignment (see chapter 9)
- An arithmetic expression can appear in the actual argument list of CALL statements and function references (see chapter 10). The value is passed as an actual argument to the subroutine or function. The data-length of an expression will be altered in the following cases:
  1. Some of the built-in functions and subroutines require one or more actual arguments to be expressions of data-length 4 bytes (see chapter 11)
  2. Where the expression appears in the actual argument list of a user written function or subroutine, and the data-length of the expression is not fixed, a data-length of 4 bytes is used
- You can index an arithmetic expression of vector or matrix mode in parentheses in order that one or more components can be selected for processing (see section 8.6)
- You can use an arithmetic expression of scalar mode in an arithmetic IF or computed GO TO statement - and in a DO statement, where it is used to control the sequence of execution (see chapter 14)
- You can use integer expressions of scalar mode in array indexing to select a particular array element (see section 7.4.1). Integer expressions of scalar or vector mode can be used in vector or matrix indexing to select one or more components for processing. In either case the data-length of the expression will be converted to 4 bytes if it is not already that data-length
- Arithmetic expressions can appear as the primaries of relational expressions (see section 8.4)

8.3 Character expressions

8.3.1 Format

A character expression takes the form of a single primary only, which could be:

- A constant (see chapter 5)
- A selected variable (see section 6.3)
- A selected array element (see section 6.4)
- A function call (see section 10.2)

The primary has to be of type character.

8.3.2 Type, data-length and mode

The type of the result of a character expression is character, the data-length is one byte and the mode is the same mode as the primary.

8.3.3 Use of character expressions

You can use character expressions in the following contexts:

- The right-hand side of a character assignment statement can be a character expression. The value of the character expression is assigned to a character variable or array element (see chapter 9)
- The actual argument list of CALL statements and function calls (see section 10.4.2) can include character expressions. The value is passed as an actual argument to the subroutine or function
- Character expressions can appear as the primaries of relational expressions (see section 8.4)
- You can index a character expression of vector or matrix mode in parentheses in order to select one or more components for processing (see section 8.6)

8.4 Relational expressions

8.4.1 Format

A relational expression consists of either two arithmetic expressions or two character expressions separated by a relational operator. The relational operator represents the operation of comparison between the values of the two expressions that it separates.

The relational operators and the operations that they represent are as follows:

.LT.	Less than
.LE.	Less than or equal to
.EQ.	Equal to
.NE.	Not equal to
.GE.	Greater than or equal to
.GT.	Greater than

Spaces are permitted in relational operators.

For example:

```
VAR .NE. 13
CHARM1 .EQ. CHARM2
CHARM3(16,19) .NE. CHARV(12)
25 .LT.3
```

The mode of a relational expression is determined by the mode of its two arithmetic or character primaries as described in section 8.1.

If the primaries of a relational expression are vectors or matrices, each component of the result will have the value .TRUE or .FALSE., depending on the truth or falsity of the relationship represented by the expression for corresponding components of the primaries.

8.4.2 Character comparisons

Where character quantities are compared, one character is considered to be greater than another if it is higher in the collating sequence; that is, if it has a higher hexadecimal value according to the list in section 3.1.

For example, the relational expression:

```
'J'.LE.'a'
```

has the value .TRUE..

8.4.3 Type, data-length and mode

The type of the result of a relational expression is always logical and its data-length is always one bit. Its mode is determined by the rules in section 8.1, where the relational operator is considered as a binary operator whose operands are the two arithmetic or character expressions.

If the operands are arithmetic expressions, the data-length of one or both expressions might be altered before the relational operation takes place, using the rules for primaries of an arithmetic operation (see section 8.2.2).

8.4.4 Use of relational expressions

A relational expression can be a primary of a logical expression, or can constitute a logical expression by itself (see section 8.5). You can only use a relational expression within logical expressions or wherever a logical expression is needed.

8.4.5 Real comparisons

In general, a real number is an approximate representation of some value. Thus you are not recommended to apply the operators .EQ. and .NE. to real expressions.

8.5 Logical expressions

8.5.1 Format

Each primary of a logical expression has to be of type logical, and can be:

- A constant (see chapter 5)
- A selected variable (see section 6.3)
- A selected array element (see section 6.4)
- A function call (see section 10.2)
- A relational expression (see section 8.4)
- A logical expression enclosed in parentheses

The logical operators and the operations that they represent are:

Unary operators	*Effect of operation*	*Binary operators*	*Effect of operation*
.NOT.	Logical negation	.OR.	Logical disjunction
		.AND.	Logical conjunction
		.LEQ.	Logical equivalence
		.NOR.	Negates the result of .OR.
		.NAND.	Negates the result of .AND.
		.LNEQ.	Negates the result of .LEQ.

A logical expression consists of a single logical primary, a single logical primary preceded by the .NOT. operator, or a string of such optionally negated logical primaries separated by binary logical operators.

For example:

```
LVAR
.NOT. LM(3,16)
(LM1.OR.LM2) .AND. (LM3.LNEQ.LM4)
```

You should obey the following rules when writing logical expressions:

- Logical primaries have to be separated by logical operators
- No two logical operators can be adjacent unless the first is one of .AND., .OR., .LEQ., .NAND., .NOR., or .LNEQ., and the second is .NOT.
- The logical operator .NOT. has to be followed by, but not preceded by, a logical primary; all other logical operators have to be both preceded and followed by logical primaries
- Arithmetic and character expressions can only appear in logical expressions as function arguments, subscripts, or as relational expressions

The result of a FORTRAN-PLUS logical expression is a scalar, vector, or matrix mode object, depending on the modes of the primaries (see section 8.1). In cases where vector or matrix mode objects are produced, the logical operations represented by the operators are performed in parallel on corresponding components of vector or matrix primaries.

Note that a logical expression used in an IF statement (see section 14.4.3) has to be of scalar mode; you can often apply the built-in functions ANY or ALL (see chapter 11) to reduce such logical vector or matrix expressions to scalar mode.

8.5.2 Type, data-length and mode

The *type* of the result of a logical expression is always logical and thus has no *data-length* specified. The *mode* of the result is determined by the rules given in section 8.1.

8.5.3 Order of evaluation

Logical expressions are evaluated from the innermost set of parentheses outwards. Within each set of parentheses or expression without parentheses the order of evaluation is as follows:

1. Evaluation of primaries, including relational expressions, expressions in parentheses, function references, and array or vector/matrix variable subscripting. This stage can also include the expansion of logical scalar primaries to vector or matrix mode, as required section 8.1)
2. All .NOT. operations
3. All .AND. and .NAND. operations
4. All .OR., .NOR., .LEQ., and .LNEQ. operations

Within each group evaluation takes place from left to right. Any functions referenced in a logical expression should not alter the values of any variables or array elements referenced in that expression. If a function is referenced more than once in a logical expression the results should be independent of the order of evaluation; in particular, two references with the same arguments should produce the same results. Violation of these rules will lead to unpredictable results.

8.5.4 Use of logical expressions

You can use logical expressions in the following contexts:

- The right hand side of a logical assignment statement (see chapter 9) which is used to give a new value to a variable or array element of type logical
- A logical expression can appear as an actual argument in a CALL statement or function reference. The corresponding dummy argument has to be a logical variable and cannot be updated within the referenced subroutine or function if the actual argument is a constant or proper expression (see chapter 10)
- Logical expressions producing scalar mode objects are used in block IF and ELSE IF statements and logical IF statements (see chapter 14)

- A logical expression producing a vector or matrix mode object can appear as an indexing expression in any parallel subscript position of a vector or matrix variable or any element reference (see chapter 7)

8.6 Indexing a vector or matrix expression

When the mode of the result of an expression is vector or matrix, you can index the expression; that is, you can select one or more components of the expression using any of the vector or matrix indexing techniques described in sections 7.2.1 and 7.3.1.

You enclose the expression to be indexed in parentheses, use the result as the *name* in the indexing construct defined in section 7.1 and follow this name with the appropriate indexing expressions, also in parentheses. To index such expressions, you can use any form of vector or matrix indexing that is valid on the right-hand side of an assignment statement. An indexed expression can appear wherever an expression of the same type can appear.

For example:

```
INTEGER A(*50,*100), IV(*50)
IV = (A(+,+) + A(-,-)) ( ,IV)
```

Chapter 9

Assignment

Assignment is the process whereby a named variable or array element, or a group of components of either, acquires a defined or redefined value. There are two basic forms of assignment in FORTRAN-PLUS: simple assignment and indexed assignment. In both cases the expression on the right hand side is evaluated before any assignment takes place; this expression can involve previous values of components on the left hand side.

9.1 Simple assignment

A simple assignment has the form:

name = *expression*

where *name* is a variable name of any type, data-length and mode, and *expression* is *assignment compatible*. The type and mode of *expression* are subject to the following restrictions:

- If *expression* is of type integer or real, *name* has to be of type integer or real. The type and data-length of the expression is converted to that of the left hand side (see section 8.2.4) before the assignment takes place
- If *expression* is of type logical, *name* has to be of type logical
- If *expression* is of type character, *name* has to be of type character
- If the mode of *expression* is scalar, the mode of *name* can be scalar, vector, or matrix with the following effects:
  - If the mode of *name* is scalar, the value of *expression* is assigned to *name*
  - If the mode of *name* is vector, *expression* is automatically expanded to a vector of the same size, each component of which is equal to the value of *expression*. Each component of this vector is then assigned in parallel to the corresponding component of *name*
  - If the mode of *name* is matrix, *expression* is automatically expanded to a matrix of the same shape, each component of which is equal to the value of *expression*. Each component of this matrix is then assigned in parallel to the corresponding component of *name*

- If the mode of *expression* is vector, the mode of *name* must also be vector and both vectors must be the same size. Each component of *expression* is assigned in parallel to the corresponding component of *name*
- If the mode of *expression* is matrix, the mode of *name* must also be matrix and both matrices must be the same size. Each component of *expression* is assigned in parallel to the corresponding component of *name*

9.2 Indexed assignment

An indexed assignment has the form:

indexing construct = *expression*

where *indexing construct* (see section 7.1) is a vector or matrix variable name or array name of any type followed by one of the indexing expressions appropriate to the left-hand side of an assignment described in Chapter 7.

The effect of an indexed assignment is to assign the value of *expression* to the identified group of the components on the left-hand side. The indexing expression which follows the variable or array name (see chapter 7) determines the identification of components to be assigned. Where more than one component is identified the assignments take place in parallel.

The same type restrictions apply to an indexed assignment as to a simple assignment (see section 9.1). The following restrictions apply to the modes of *indexing construct* and *expression*:

- If the array name in *indexing construct* is of scalar mode, the mode of *expression* has to be scalar
- If the variable or array name in *indexing construct* is of vector mode, the mode of *expression* can be either of:
  - scalar, in which case the value of *expression* is assigned to every vector component identified by the indexing expression
  - vector, in which case each identified component of the left-hand side vector has the corresponding component of *expression* assigned to it. The vectors have to be the same size
- If the variable or array name in *indexing construct* is of matrix mode, the mode of *expression* can be any of:
  - scalar, in which case the value of *expression* is assigned to every matrix component identified by the indexing expression
  - vector, provided that the indexing on the left-hand side is such that just one parallel subscript position is null. If the first subscript is null each identified component of the left-hand side matrix is assigned the value of the corresponding component of the matrix of the same shape formed by expanding the vector value of *expression* (which must have a size equal to the number of rows in the left hand side matrix) to a matrix with the appropriate number of identical columns. If the second subscript is null each identified component of the left-hand side matrix is assigned the value of the corresponding

component of the matrix of the same shape formed by expanding the vector value of *expression* (which must have a size equal to the number of columns in the left hand side matrix) to a matrix with the appropriate number of identical rows

- matrix, in which case each identified component of the left-hand side is assigned the value of corresponding component of *expression*. The matrices must be the same shape

Those components of the left-hand side variable or array element that are not identified by the indexing expression are unchanged after the assignment.

Chapter 10

Subroutines and functions

A procedure is a self-contained body of code used to perform a specific part of a programming task. A DAP program consists of one or more FORTRAN-PLUS procedures (function or subroutine subprograms), or procedures written in APAL, the assembly language of the DAP.

Normally, a procedure is entered, some or all of its statements are executed and then a return is made to the program unit from which the procedure was called. The statements concerned with entering and leaving procedures are described in sections 10.2, 10.3 and 10.6.

By using COMMON blocks (see Chapter 12), function results (see section 10.6), and argument association (see section 10.4.2) you can transfer values between program units.

FORTRAN-PLUS procedures fall into the following categories:

- User-written functions
- Built-in functions
- User-written subroutines
- Built-in subroutines

Built-in functions and subroutines are listed in chapter 11.

10.1 Procedure declaration

10.1.1 Subroutine subprograms

A subroutine subprogram is a separate FORTRAN-PLUS program unit – it is not contained within another FORTRAN-PLUS program unit. A subroutine begins with a **SUBROUTINE** statement, which can take either of the forms:

> **ENTRY SUBROUTINE** *name*
> **SUBROUTINE** *name*(*dummy argument list*)

where:

- *name* is the subroutine name which has to be distinct from the name of any other FORTRAN-PLUS program unit, or COMMON block name (see section 3.2.1)
- *dummy argument list*, which is optional, has the form:

  $item_1$, $item_2$, ... $item_n$

  where each $item_i$ is used within the subroutine as a variable name, array name, function name or subroutine name (see section 10.4.1). If *dummy argument list* is omitted, then so are the enclosing parentheses

You can call an ENTRY SUBROUTINE, that is, the first form of the SUBROUTINE statement, from a program unit within your DAP program or from the associated host program – it is the only form of FORTRAN-PLUS subroutine that you can call from the host program and your DAP program must have at least one such subroutine. You use the second form of the subroutine statement for a subroutine that can only be called from another FORTRAN-PLUS program unit and can optionally have values passed to it by argument association (see section 10.4). In either form a subroutine can call itself recursively.

The statements in a subroutine subprogram have to occur in the following order:

1. Either form of the SUBROUTINE statement
2. Optional COMMON, DATA, DIMENSION, EQUIVALENCE, EXTERNAL, GEOMETRY, IMPLICIT, PARAMETER, or type specification statement(s)
3. At least one executable statement
4. An END statement (see section 10.1.3)

The final executable statement should be a RETURN, GOTO or STOP; you will get a run-time error if the END statement is reached during execution.

A subroutine can contain a CALL statement that refers to itself directly, or it can contain an indirect reference to itself. For example, subroutine SUBR can contain a CALL statement or function call that refers to another subroutine or function subprogram which in turn contains a call to subroutine SUBR.

10.1.2 Function subprograms

Functions in FORTRAN-PLUS can return scalar, vector or matrix values.

10.1.2.1 Function header

A function subprogram is a separate FORTRAN-PLUS program unit; that is, it is not contained within another FORTRAN-PLUS program unit. A function begins with a **FUNCTION** statement, which has the form:

*type *data-length* **FUNCTION** *name* (*dummy argument list*)

where:

- *type* denotes the type of the object returned by the function and can be INTEGER, REAL, DOUBLE PRECISION, LOGICAL or CHARACTER. *type* can be omitted in which case **data-length* should also be omitted and the type and data-length of the object returned by the function are determined as shown in section 10.1.2.2
- **data-length*, which can be omitted, denotes the number of bytes used to represent the object returned by the function. *data-length* can take the range of values given in section 4.5
- *name* is the function name, and has to be distinct from the name of any other FORTRAN-PLUS program unit or COMMON block name. *name* is used within the function subprogram as the name of a variable with the type, data-length and mode of the object returned by the function; this variable has to be assigned a value (that is, become defined) within the function subprogram
- *dummy argument list* takes the form:

  $name_1$, $name_2$, ... $name_n$

  where each $name_i$ is used within the function subprogram as a variable name, array name, function name, or subroutine name (see section 10.4.1). There has to be at least one item in the list

10.1.2.2 Type and data-length of a function

Any subprogram that calls a particular user-written function must also specify that function using an EXTERNAL statement (see section 10.1.4).

You specify the type and data-length of a user-written function (that is the type of object returned by the function) in one of the following ways:

- In the function definition as part of the FUNCTION statement (see section 10.1.2.1)
- By the appearance of the function name in a type specification statement within the definition of that function (see next paragraph)
- If you have not specified the type and data-length in one of the above ways, then the first letter of the function name (by default, or by use of the IMPLICIT statement) determines the type and data-length in exactly the same way as for variables (see section 6.1.1)

Where the name of a function appears in place of a variable name in a type specification statement or a DIMENSION statement (see section 6.1.2), the format is as described in section 6.1.2 except that you cannot use /*value list*/ to give values to the function name.

10.1.2.3 Mode of a function

A function can return a value of scalar, vector or matrix mode. A function is called a scalar, vector or matrix function according to the mode of the value returned by the function. Functions cannot return array values.

You determine the mode of a function and (if it is a vector or matrix function) its shape either by a DIMENSION statement or by a type specification which has the form of a vector or matrix declarator (see section 6.1.2). If there is no mode specification in this form, the mode of the function is taken to be scalar.

For example:

```
FUNCTION MYFUN(VEC1,I)
REAL VEC1(*)
INTEGER I
REAL MYFUN(*I,*SIZE(VEC1))
```

The last statement declares MYFUN to return a real matrix result with a number of rows equal to the value of I, and the number of columns equal to the number of components in VEC1. If the last statement was:

```
DIMENSION MYFUN(*I,*SIZE(VEC1))
```

the function would return an integer matrix (since MYFUN begins with M) of a similar size. If the last statement was absent, the function would return a 4 byte integer scalar.

10.1.2.4 Body of function subprogram

The statements in a function subprogram have to occur in the following order:

1. FUNCTION statement
2. Optional IMPLICIT, DIMENSION, COMMON, EQUIVALENCE, PARAMETER, DATA, GEOMETRY, EXTERNAL, or type specification statement(s)
3. At least one executable statement. For the function to be meaningful, at least one statement must assign a value to *name*
4. An END statement (see section 10.1.3)

The final executable statement should be a RETURN, GOTO or STOP; you will get a run-time error if the END statement is reached during execution.

Within a given function the occurrence of the name of that same function within an expression is interpreted as referring to the current value of that function, rather than a further call (or recursion) of the function. A function can however call itself recursively by indirect means; for example, function FN can contain a call to some other subroutine or function, which in turn contains a call to FN.

Note that you can only invoke a FORTRAN-PLUS function subprogram through a function call (see section 10.2) in another FORTRAN-PLUS program unit – a FORTRAN-PLUS function subprogram cannot be called from the host program.

The function name can be used like any other variable within the function. The function name is used to return the value of the function and therefore has to be assigned a value before a RETURN statement is executed.

10.1.3 The END statement

The **END** statement is the last statement of a program unit and has to be present in every program unit. It contains END within columns 7 to 72 of an initial line and spaces in all other columns of that line. The next statement, if any, has to be a SUBROUTINE, FUNCTION, or BLOCK DATA statement. By convention, any comment or blank lines that follow an **END** statement but precede the next program unit, if any, are considered to belong to the same program unit as that **END** statement.

10.1.4 The EXTERNAL statement

The **EXTERNAL** statement is used within a subprogram to declare another subprogram and has either of the following forms:

> **EXTERNAL** $name_1$, $name_2$, ... $name_n$
>
> **EXTERNAL** *type *data-length* **FUNCTION** $name_1$, $name_2$, ... $name_n$

In the first form of the statement, each $name_i$ can be either of:

- A user-written subroutine or function name
- The name of a block data subprogram

In the second form of the statement, each $name_i$ is the name of a user-written function subprogram that is to be called within the program unit. You can omit *type* and **data-length*; if you omit *type* you must also omit **data-length*, but you can include *type* and omit **data-length*. You can give *type* and **data-length* in a separate type specification statement.

You declare the *mode* of an EXTERNAL function in the same way as described in section 10.1.2.3. The *type*, *data-length* and *mode* should correspond with those in the function definition itself. Where the shape is not fully known, you use the assumed size form of parallel dimension for either or both dimensions (see section 6.1.2).

For example:

```
EXTERNAL MATMULT
REAL MATMULT(*,*)
```

declares a user-written function MATMULT which returns a real matrix.

The EXTERNAL statement has one of three uses, corresponding to the class of name referred to and the form of the statement used:

1. If *name*$_i$ is the name of a user-written function that is called within a subprogram, *name*$_i$ should appear in an EXTERNAL statement of the second form in that subprogram. If *name*$_i$ is the same as the name of a built-in function (see chapter 11) any call to *name*$_i$ within the subprogram will be taken as a call to the user-written function rather than the built-in function
2. If *name*$_i$ is the name of a user-written function or subroutine that appears in the actual argument list of a CALL statement or function call within a subprogram, *name*$_i$ should appear in an EXTERNAL statement in that subprogram. Note that built-in function or subroutine names cannot be passed as actual arguments
3. If *name*$_i$ is the name of a block data subprogram that is to be included in the DAP program, *name*$_i$ can appear in an EXTERNAL statement in some program unit in the DAP program. Block data subprograms are described in chapter 12

10.2 Calling functions

Functions, both user-written and built-in (see chapter 11), are entered when a statement containing an expression in which a *function call* is evaluated, is executed.

A function call is used as an expression primary and has the form:

name (*actual argument list*)

where:

- *name* is the name of a user-written or built-in function
- *actual argument list* is defined in section 10.4.1

The function is entered at the first executable statement and control passes through the function until a RETURN statement is executed (see section 10.6).

A function call can only occur in an expression of the appropriate type (see chapter 8).

If the function call refers to a user-written function, the name (and also the *type*, *data-length* and *mode*) of the function must appear in an EXTERNAL statement in the program unit in which the function reference occurs.

A user-written function can have the same name as one of the built-in functions listed in chapter 11. The function name must appear in an EXTERNAL statement in the program unit in which the user-written function is required otherwise the built-in function will be assumed.

A function call to a FORTRAN-PLUS user-written or built-in function can occur only in a DAP program unit; that is, a FORTRAN-PLUS function cannot be directly called from a host program unit.

A function cannot contain an explicit call of itself. However *indirect recursion* is allowed (see section 10.1.1).

10.3 Calling subroutines

A subroutine is entered whenever a **CALL** statement referring to that subroutine is executed. The **CALL** statement has the form:

CALL *name* (*actual argument list*)

where:

- *name* is the name of a user-written or built-in subroutine
- *actual argument list* is a list of expressions, array names, and procedure names separated by commas (see section 10.4); *actual argument list* is omitted if the called subroutine has no arguments. If *actual argument list* is omitted, then so are its enclosing parentheses

The subroutine is entered at the first executable statement and control passes through the subroutine until a RETURN statement is executed (see section 10.6).

A call to a FORTRAN-PLUS subprogram can appear in a program unit in either the DAP program or the host program, but in the latter case the CALL statement has to refer to a FORTRAN-PLUS ENTRY subroutine (see section 10.1.1) as the argument to the interface subroutine DAPENT which initiates processing in the DAP (see *DAP Series: Program Development*).

A subroutine can call itself, or it can call a subroutine or function that in turn calls the original subroutine. A FORTRAN-PLUS subroutine can therefore be either *directly* or *indirectly* recursive.

10.4 Procedure arguments

10.4.1 Dummy arguments

The declaration of a user-written subroutine can have, whereas the declaration of a user-written function must have, a dummy argument list of the form:

$name_1$, $name_2$, ... $name_n$

where each $name_i$ (see section 3.2.1) is used within the procedure as one of:

- A variable name – see chapter 6
- An array name – see chapter 6
- A function name – see sections 10.1.2.1 and 10.4.6
- A subroutine name – see sections 10.1.1 and 10.4.6

In general, each dummy argument has to be declared within the procedure and thus given *type*, *data-length*, *mode* and dimension(s).

Dummy arguments which are variables (known as *dummy variables*) cannot appear in COMMON, EQUIVALENCE, or DATA statements within the procedure; they can appear in type specification statements, but cannot be initialised in them.

The implications of the various forms for dimensions of vector and matrix variable and array arguments are discussed in section 10.4.5.

Dummy arguments which are arrays (known as *dummy arrays*) have to appear in DIMENSION or type specification statements within the procedure, but cannot be given initial values. They cannot appear in COMMON, EQUIVALENCE, or DATA statements. The non-parallel dimensions of a dummy array have to be either integer constants or integer scalar variables; in the latter case, the integer scalar variable has to be either a dummy argument or an item in a COMMON block declared in the procedure. This form of array declaration allows a dummy array to have non-parallel dimensions of different sizes on different executions of the procedure, although the number of dimensions has to remain constant.

10.4.2 Actual arguments

When a subroutine or function is entered, the *actual argument list* in the CALL statement or function call has the form:

$item_1$, $item_2$, ... $item_n$

where each $item_k$ can be an expression or a function or subroutine name. The different forms which $item_k$ can take are classified as follows in order to differentiate between those parameters which return values and those which do not:

1. A variable name used to identify a variable (which can be scalar, vector or matrix (see chapter 6))
2. A complete array element (see section 7.1.2) – this is an array element where any parallel subscripts in the indexing construct are null; that is, a scalar has been selected from an array of scalars, a vector from an array of vectors or a matrix from an array of matrices
3. An indexed variable (see section 7.1.2).

   This construct is one of the indexed vector selection constructs (see section 7.2.1) or indexed matrix selection constructs (see section 7.3.1) which selects a scalar from a vector variable or array element, or a scalar or vector from a matrix variable or array element. Constructs which use + or – indexing subscripts are *not* classified as 'indexed variable constructs'.
4. A *proper expression*. A proper expression is any expression other than those described above, and is a reference to a value calculated when the called procedure is entered.

   A proper expression can be:

   - An expression consisting of a single constant
   - An expression containing any of the following, except in a subscript position:
     - Unary or binary operators
     - Variable or array element names with + or – indexing expressions
     - Grouping parentheses
     - Function calls

Thus a variable name or a complete array element name enclosed in parentheses is a proper expression

5. An array name used to identify an array
6. A function name used to identify a user-written function
7. A subroutine name used to identify a user-written subroutine

The actual argument list in a CALL statement or function reference has to contain the same number of actual arguments as there are dummy arguments in the dummy argument list of the SUBROUTINE or FUNCTION statements that declare the called procedure.

10.4.3 Argument association

On entry to the called subprogram, actual arguments are associated with dummy arguments according to their respective positions in the argument lists; that is, the i^{th} actual argument is associated with the i^{th} dummy argument.

This association is used to pass values into procedures and, in some circumstances, to return values from procedures (see section 10.4.4).

Actual arguments are also referred to as *parameters.*

For each class of dummy argument, the corresponding actual argument must be one of:

Dummy argument	*Actual argument*
Variable name	Variable name, complete array element, indexed variable or proper expression
Array name	Array name or complete array element
Function name	Function name that appears in an EXTERNAL statement in the calling subprogram
Subroutine name	Subroutine name that appears in an EXTERNAL statement in the calling subprogram

- If the dummy argument is a variable, array, or function name, the corresponding actual argument has to be of the same type, data-length and mode. Shapes of vectors and matrices have to conform (see section 10.4.5)
- If a proper expression is used as an actual argument, the corresponding dummy argument has to be a variable name
- If a dummy argument is a *dummy array* and the corresponding actual argument is an array name, the first element of the dummy array maps onto the first element of the actual array. If the actual argument is a *complete array element* (see section 10.4.2), the first element of the dummy array maps onto the specified element of the actual array, the second element onto the next, and so on. The actual array has to contain at least as many elements as the dummy array

Dummy variables and dummy arrays acquire defined values in the following ways:

- Dummy variables are given a defined value either by association with an actual argument on entry to the procedure or by assignment during the execution of the procedure
- If the actual argument is a proper expression, the expression will be evaluated once on entry to the procedure and that value becomes the value of the dummy variable
- A dummy array element can be given a defined value either by association with an actual array or by assignment during the execution of the procedure

If a dummy variable is selected (by value) during the execution of a procedure, it has to have a defined value at that time.

If an element of a dummy array is selected (by value) during the execution of a procedure, it has to have a defined value at that time.

10.4.4 Parameter write-back

During the execution of a procedure, a dummy argument can have its value defined or redefined and hence, by argument association, a (new) value can be meaningfully written back to an actual argument on return from the procedure. However there are occasions when, although a dummy argument has been given a (new) value no writing to the corresponding actual argument takes place, or defining or redefining the value of a dummy argument leads to results which are unreliable or unmeaningful:

- If the actual argument associated with a dummy variable is a proper expression, then no value can be written back – although the value of the dummy variable can be redefined during execution of the procedure, there is nowhere for the new value to be written back to
- If actual arguments are variable names, complete array elements, indexed variables or array names, then COMMON or EQUIVALENCE associations between such actual arguments and other variables within the procedure can lead to unreliable results if their corresponding dummy arguments are redefined during the execution of the procedure
- An actual argument should not be associated with storage that overlaps the storage representing another actual argument
- In the case where the actual argument is an indexed variable, a copy of the value of the subset of components selected by the indexing construct from a variable, array or array element is taken on entry to the procedure. This copy is used for processing within the procedure until a RETURN is made to the calling procedure, at which point the copy is written back to the variable, array, or array element from which it was taken. This process does not allow the called procedure to perceive the effects of any association of the argument with any variable of the called procedure – hence, if there is such an association, the results are unpredictable
- Some other implications are mentioned in section 10.4.7

10.4.5 Shape of vector and matrix arguments

If a dummy argument is a vector or a matrix or an array with vector or matrix elements, you specify the size of each parallel dimension in the parallel subscript position(s) of the vector, matrix or array declarator (see section 6.1.2).

You can generalise subprograms if you use the assumed size form (that is, there is simply a * in the parallel subscript position) for parallel dimensions of vector and matrix dummy arguments rather than making the sizes of the dimensions explicit.

For example, the declarations:

```
SUBROUTINE COMPRESS(VEC1, MAT1, MATSET,I,J)
REAL VEC1(*)
INTEGER MAT1(*,*)
LOGICAL MATSET(*,*,I,J)
```

specify that the vector and matrix dummy arguments, VEC1 and MAT1 can have any value for the sizes of their dimensions; the values of those dimensions are inherited from those in the actual parameters. The matrix array, MATSET, can also have any value for the sizes of its parallel dimensions, but its non-parallel dimensions are passed as further parameters. Note that you cannot specify non-parallel dimensions of arguments as being of assumed size; this syntax applies only to parallel dimensions.

You can also use the SIZE function in declarations to let local variables (or function results) have parallel subscripts dependent on the parallel subscripts of arguments:

```
SUBROUTINE SUMMPYIMM(A,B)
INTEGER A(*,*), B(*,*)
INTEGER D(*SIZE(A,1),*SIZE(B,2))
. . .
```

and

```
FUNCTION IM(P)
LOGICAL IM(*SIZE(P,1),*SIZE(P,2)),P(*,*)
. . .
```

You cannot, however, use this notation for any statically declared variables or arrays – that is, those declared in COMMON or DATA statements. Also the dimensions of all objects in EQUIVALENCE statements must be known at compile-time.

If you use SIZE to dimension a variable or array in a declaration statement, then the relevant dimension(s) of the argument to that use of SIZE must not itself have been dimensioned using SIZE. In other words, you cannot nest SIZEs in declaration statements.

For example:

```
    REAL BM(*SIZE(A,1),*SIZE(B,1))
    INTEGER AV(*SIZE(BM,2))
```

is invalid, but

```
    REAL BM(*SIZE(A,1),*56)
    INTEGER AV(*SIZE(BM,2))
```

is valid.

Unless parallel subscripts of dummy arguments are integer constants, it is not possible to determine their values until run-time. A run-time error is then given if there is a mismatch between these values, and the corresponding sizes of the dimensions of the actual argument passed into the procedure. Note that, by default, no checks are made on non-parallel dimensions.

If you give an assumed size for a parallel dimension, no check is made; the value is taken as that in the actual argument. You can use the SIZE function call (see chapter 11) to establish a check against a dimension of another vector or matrix (or another dimension of this one).

For example:

```
    FUNCTION MATMUL(ARRAY,B,N)
    REAL ARRAY(*,*,N)
    REAL B(*SIZE(ARRAY,2),*)
    REAL MATMUL(*SIZE(ARRAY,1),*SIZE(B,2))
    REAL C(*SIZE(ARRAY,1),*SIZE(ARRAY,2))
    . . .
```

The input ARRAY is an array of N matrices; each matrix element is of a shape not known at compile time. B is declared to have its first dimension equal to the second dimension of a matrix element in ARRAY; the system would give a run-time error if this were not the case.

The function MATMUL returns a matrix with a number of rows equal to the first dimension of ARRAY and number of columns equal to the second dimension of B.

Finally C is declared as a local matrix of the same shape as a matrix element in ARRAY.

10.4.6 Dummy procedure names

Dummy procedure names are another form of parameter to a subroutine or function. Within the called procedure they are used as subroutine names in CALL statements or function names in function calls; a dummy procedure name can also appear in the actual argument list of a CALL statement or function call within the procedure.

The corresponding actual argument has to be the name of a user-written function or subroutine; a built-in function or subroutine cannot be used as an actual argument. The name of a function or subroutine that is to be used as an actual argument has to appear in an EXTERNAL statement in the program unit in which the actual argument list appears.

Note that if a function name appears on its own in an actual argument list, the corresponding dummy argument will be treated as a dummy function name. If the function name is immediately followed by an actual argument list in parentheses it is treated as a proper expression; the function is called and the value returned by the function is passed to the dummy argument, which is used as a dummy variable.

Examples

1. If subroutine A is to be passed by subroutine C as the second of two arguments expected by subroutine B, then the following statements occur in subroutine C:

   ```
   SUBROUTINE C
   EXTERNAL A
   . . .
   CALL B(X,A)
   . . .
   END
   ```

   subroutine B can then be written as:

   ```
   SUBROUTINE B(X,Y)
   EXTERNAL Y
   . . .
   CALL Y(X)
   . . .
   END
   ```

2. If function F was to be evaluated in subroutine H and the result passed to a further subroutine or function, its mode and type have to be declared there:

   ```
   SUBROUTINE H
   EXTERNAL FUNCTION F
   REAL F(*,*)
   . . .
   X = MAX (F(1.9,0.0))
   . . .
   END
   ```

   F returns a matrix with a shape determined at run-time

3. If function F is to be passed to subroutine E, subroutine D would contain:

   ```
   SUBROUTINE D
   EXTERNAL F               (the type and mode of F would have to be
   CALL E(X,F)               specified if F is used as a
   . . .                         function reference in D)
   END
   ```

E has to declare the function, giving its mode and type:

```
SUBROUTINE E(XX,FF)
REAL XX
EXTERNAL FUNCTION FF
REAL FF(*,*)
. . .
XX = MAXV(FF(1.9,0.0))
. . .
END
```

10.4.7 Functions and side-effects

When a user-written function or subroutine is executed it can change the value of an actual argument passed to it, a variable in COMMON storage or a preset variable in the function or subroutine. In such cases the function or subroutine is said to cause *side effects.* In general this does not cause a problem for subroutine calls. However with function calls, the side effects can change the result of a function in an unpredictable way if the function is called more than once in a single expression. Also the result of an expression will be unpredictable if it contains a function call which alters the value of one of its arguments and you have used that argument elsewhere in the expression.

For example, given the following function definition:

```
INTEGER FUNCTION M(I)
DATA J/2/
M= I+J
J=J+M
RETURN
END
```

the expression:

```
M(1) + M(2)
```

will yield unpredictable results because it depends on the evaluation order (which is not defined, see section 8.2.3) - if M(1) is evaluated first, the result will be 10; if M(2) is evaluated first the result will be 11.

Similarly the expression :

```
X+ FUN(X)
```

will have unpredictable values if FUN alters the value of X.

10.5 Storage allocation in procedures

There is no restriction in FORTRAN-PLUS on recursion of procedure calls except the space available for the *run-time stack* (that is, the temporary work-space needed by a FORTRAN-PLUS program – see *DAP Series: Program Development*).

Data in FORTRAN-PLUS procedures can be classified in the following ways:

- Dummy arguments

  These are associated on each call with actual arguments

- Variables or arrays in COMMON blocks or in association with variables or arrays in COMMON blocks

  Each recursion of a procedure references the same occurrence of such data; that is, storage for such data is allocated only once. The values of such data are preserved while control is not in the procedure, unless associated values are changed

- Variables or arrays which are initialised in DATA or type specification statements, that is preset data, or are in association with variables or arrays in preset data

  Each recursion of a procedure references the same occurrence of such data; that is, storage for such data is allocated only once. The values of such data are preserved while control is not in the procedure, unless associated values are changed

- Variables or arrays that are EQUIVALENCEd to other variables or arrays and are neither initialised nor in COMMON.

  In the current implementation of FORTRAN-PLUS each recursion of a procedure references the same occurrence of such data; that is, storage for such data is allocated only once. However, for this case, you must not rely on the values of such data being preserved between successive calls to the procedure

- Local variables and arrays. These are all the other variables or arrays declared explicitly or implicitly in a subprogram unit.

  It is these variables which are allocated storage on the run-time stack on every CALL to the procedure. Their values become undefined when control leaves the procedure through a RETURN statement.

In general, FORTRAN-PLUS cannot work out the stack space requirements because it cannot forecast:

- If your program is recursive
- How large are the dynamically-sized local vectors or matrices (defined either with a *SIZE subscript or with *N where N is a integer scalar dummy argument)

Although the compilation system makes an estimate of stack space by adding together the needs of each subprogram and guessing at the value of any dynamically-sized dimension, you could run out of space; in some circumstances, the compilation system over-estimates the stack space needed. The size of run-time stack can be specified or adjusted to solve these problems (see *DAP Series: Program Development*).

10.6 Entering and leaving procedures

Control passes from a procedure when either a CALL statement is executed or a statement is executed in which a function call occurs. Control passes to the first executable statement of the called subroutine or function and eventually returns to the same invocation of the original procedure, either to the statement following the CALL statement or to the statement in which the function call occurred. Control is passed back to the calling procedure when a RETURN statement is executed; every procedure should therefore contain at least one RETURN statement.

In the case of the ENTRY SUBROUTINE at which the DAP program was entered, the RETURN statement returns control to the host program.

You should therefore label the executable statement, if any, following a RETURN statement, otherwise it will never be executed.

When a RETURN statement is executed within a function subprogram, the value currently associated with the function name, which is used within the procedure as though it were a variable, is passed back to the function reference as the result. This variable has to be given an appropriate defined value before a RETURN statement is executed.

Chapter 11

Built-in procedures

Built-in procedures are subroutines or functions which are defined as part of the language; they are supplied automatically as part of the compilation process and no EXTERNAL statements are required to declare them.

The typographical conventions used in this chapter are :

- Procedure names are shown in upper case **BOLD TYPE**
- Text that would be replaced by other text in what you actually write is shown in lower case *italics*
- Parentheses, where they appear in a procedure call, are an essential part of the syntax of the call

11.1 Componental functions

FORTRAN-PLUS built-in *componental* functions perform a set of standard arithmetic operations on FORTRAN-PLUS data objects.

Componental functions return a result of the same mode and shape, and usually the same type and data-length, as the argument. In the case of non-scalar arguments, the operation represented by the function is performed on each component of the argument in parallel. Componental functions leave their arguments unchanged.

ABS **ABS**(a)

where:

- a is an integer or real object of any data-length and mode

ABS returns the absolute value of a. The result of the function has the same type, data-length and mode as a.

ATAN

ATAN(a)

where:

- a is a real object of any data-length and mode

ATAN returns the inverse trigonometric tangent of a, which is expressed in radians. The result of the function is a real object in the range $-\pi/2$ to $\pi/2$ having the same data-length and mode as a.

COS

COS(a)

where:

- a is a real object of any data-length and mode

COS returns the trigonometric cosine of a, which is assumed to be expressed in radians. The result of the function is a real object of the same data-length and mode as a.

EXP

EXP(a)

where:

- a is a real object of any data-length and mode

EXP returns the exponential of a; that is, e to the power of a. The result of the function is a real object of the same data-length and mode as a.

FIX

FIX(a)

where:

- a is a real or logical object of any data-length and mode

FIX returns the equivalent value of a converted to type integer. A logical argument is treated as a real with the value 1.0 or 0.0 depending on whether it has the value .TRUE. or .FALSE. respectively. If the argument is of type real the result has the same data-length; if the argument is of type logical the result has data-length 4 bytes. The result of the function has the same mode as a.

Where a component of a real argument is not a whole number, the fractional part is ignored.

FLOAT

FLOAT(a)

where:

- a is an object of any mode and type logical or integer with, in the latter case, a data-length in the range 3 bytes to 8 bytes

FLOAT returns the equivalent value of a converted to type real. A logical argument is treated as an integer with the value 1 or 0 depending on whether it has the value .TRUE. or .FALSE. respectively. If the argument is of type integer the result has the same data-length; if the argument is of type logical the result has data-length 4 bytes. The result of the function is of the same mode as a.

LENGTH

LENGTH(a, *n*)

where:

- a is an integer or real object of any data-length and mode
- *n* is a basic integer constant

LENGTH returns a value equivalent to a but with its data-length converted to the number of bytes given by *n*. If a is of type integer, *n* has to be in the range 1 to 8 and if a is of type real, *n* has to be in the range 3 to 8. The result of the function has the same type and mode as a.

LOG

LOG(a)

where:

- a is a real object of any data-length and mode

LOG returns the logarithm, to base *e*, of a. The result of the function is a real object of the same data-length and mode as a.

SIN

SIN(a)

where:

- a is a real object of any data-length and mode

SIN returns the trigonometric sine of a, which is assumed to be expressed in radians. The result of the function is a real object of the same data-length and mode as a.

SQRT **SQRT**(a)

where:

- a is a real object of any data-length and mode

SQRT returns the positive square root of a. The result of the function is a real object of the same data-length and mode as a.

11.2 Non-componental procedures

Non-componental procedures are a set of built-in subroutines and functions which perform a wide variety of operations on scalars, vectors and matrices.

See appendix B for information on superseded procedures and alternative syntax for some of the procedures listed below which have been retained for compatibility with earlier versions of FORTRAN-PLUS.

The vector, and matrix shift functions (from SHEC to SHWP in this chapter) specify a *geometry* that applies to a shift. The geometry of a shift can be *plane* or *cyclic*: plane geometry implies that the values shifted in at an end or edge are zero, .FALSE., or null characters depending on the type of vector, or matrix being shifted; cyclic geometry implies that values shifted off one end or edge are shifted in at the opposite end or edge. The type of shift is determined by the function name and is independent of any GEOMETRY statement within the calling procedure.

In addition to the typographical conventions listed at the beginning of this chapter, the conventions used in the following descriptions of procedures are:

- Optional arguments are enclosed in square brackets [ ]
- A choice of arguments is listed vertically and enclosed in curly brackets { }

The procedures listed below are all functions unless stated otherwise.

ALL $\textbf{ALL}(\left\{\begin{matrix} lv \\ lm \end{matrix}\right\})$

where:

- *lv* is a logical vector of any shape
- *lm* is a logical matrix of any shape

ALL takes a single logical vector, *lv*, or matrix, *lm*, argument, and returns a logical scalar object that is the logical AND of the values of all components of the argument.

ALT

ALT(*i*, *r*)

where:

- *i* and *r* are 4-byte integer scalars

ALT returns a logical vector with as many components as the value of *r*. The first *i* components of the result have the value .FALSE. followed by *i* components of value .TRUE. and so on, until all components of the vector are assigned. The value of *i* is taken modulo *r* when determining the numbers of components in each group with the same value. If the value of *i* modulo *r* is zero (that is, *i* is an integral multiple of *r*), all components of the result vector will be set .FALSE..

(see also appendix B)

ALTC

ALTC(*i*, *r*, *c*)

where:

- *i*, *r* and *c* are 4-byte integer scalars

ALTC returns a logical matrix with alternating groups of .FALSE. columns and .TRUE. columns (that is, each column has all components with a .FALSE. value or all components with a .TRUE. value).

r and *c* give the row and column dimensions of the result matrix. If the value of *i*, modulo *c*, is *j* the matrix will have all components in the first *j* columns .FALSE., all the components in the next *j* columns .TRUE., and so on, until the matrix is full. If *j* is zero (that is, *i* is an integral multiple of *c*) all the components of the result matrix will be set .FALSE..

(see also appendix B)

ALTR

ALTR(*i*, *r*, *c*)

where:

- *i*, *r* and *c* are 4-byte integer scalars

ALTR returns a logical matrix with alternating groups of .FALSE. rows and .TRUE. rows (that is, each row has all components with a. FALSE. value or all components with a .TRUE. value).

r and *c* give the row and column dimensions of the result matrix. If the value of *i*, modulo *r*, is *j* the matrix will have all components in the first *j* rows .FALSE.,all components in the next *j* rows .TRUE. until the matrix is full. If *j* is zero (that is, *i* is an integral multiple of *r*) all the components of the result matrix will be set .FALSE..

(see also appendix B)

ANDCOLS

ANDCOLS(*lm*)

where:

- *lm* is a logical matrix of any shape

ANDCOLS returns a logical vector which has as many components as there are rows in *lm*. The value of component *i* of the result is the logical AND of the values of all components in row *i* of *lm*.

ANDROWS

ANDROWS(*lm*)

where:

- *lm* is a logical matrix of any shape

ANDROWS returns a logical vector which has as many components as there are columns in *lm*. The value of component *i* of the result is the logical AND of the values of all components in column *i* of *lm*.

ANY

$\text{ANY}(\left\{ \begin{array}{c} lv \\ lm \end{array} \right\})$

where:

- *lv* is a logical vector of any shape
- *lm* is a logical matrix of any shape

ANY takes a single logical vector, *lv*, or matrix, *lm*, argument, and returns a logical scalar object that is the logical OR of the values of all the components of the argument.

COL

$\mathrm{COL}(\left\{ \begin{array}{l} i\,,\,r\,,\,c \\ iv\,,\,c \end{array} \right\})$

where:

- *i*, *r* and *c* are 4-byte integer scalars
- *iv* is a 4-byte integer vector

COL returns a logical matrix whose dimensions depend on the arguments. If the first argument is a scalar, *r* and *c* give the row and column dimensions of the resulting matrix. If the first argument is vector *v*, the size of *v* gives the number of rows while *c* gives the number of columns of the resulting matrix.

If the first argument is scalar *i* with value *s*, all components in column *s* of the matrix result are set .TRUE.; all other components of the result are set .FALSE..

If the first argument is vector *iv*, then for each component in the vector (say component k , having the value *j*) there will be a corresponding component of the matrix result that is .TRUE. (component (k , j)); all other components of the result are .FALSE..

If the value of *i* is not in the range 1 to *c* inclusive, all components of the result are .FALSE.; if the value of a component of *iv* is not in the range 1 to *c* inclusive, all components in the corresponding row of the matrix are .FALSE..

(see also appendix B)

COLN

COLN(*lm*)

where:

- *lm* is a logical matrix of any shape

COLN returns an integer vector with as many components as there are rows in the matrix. The n^{th} component of the vector result takes as its value the lowest column number of any .TRUE. components in the n^{th} row of the matrix.

If all the components in a row of *lm* are .FALSE. then the corresponding component of the result is zero.

COLS

COLS(i, j, r, c)

where:

- i, j, r and c are 4-byte integer scalars

COLS returns a logical matrix with all components in columns i to j inclusive set .TRUE. and all other components set .FALSE..

r and c give the row and column dimensions of the logical matrix result.

If either i or j is not in the range 1 to c or if i is greater than j then all the components of the result are .FALSE..

(see also appendix B)

EL

EL(n, r)

where:

- n and r are 4-byte integer scalars

EL returns a logical vector result with r components. If n is not in the range 1 to r all components of the result are .FALSE., otherwise the n^{th} component of the result is set .TRUE. with all other components .FALSE..

(see also appendix B)

ELN

ELN($\left\{ \begin{array}{c} lv \\ lm \end{array} \right\}$)

where:

- lv is a logical vector of any shape
- lm is a logical matrix of any shape

ELN returns an integer scalar of 4 byte data-length. If you specify a vector argument the result will be the index of the first component of the argument which has a .TRUE. value.

If you specify a matrix argument the result will be the index of the first .TRUE. component of the matrix viewed in column-major order.

In either case, if no component of the argument is .TRUE. the result will be zero.

ELS

ELS(i, j, r)

where:

- i , j and r are 4-byte integer scalars

ELS returns a logical vector whose number of components is specified by r.

If either i or j is not in the range 1 to r or i is greater than j then all components of the resulting logical vector are set .FALSE.. Otherwise components i to j inclusive of the result vector are set .TRUE. and all other components .FALSE..

(see also appendix B)

FRST

FRST$(\left\{ \begin{array}{c} lv \\ lm \end{array} \right\})$

where:

- *lv* is a logical vector of any shape
- *lm* is a logical matrix of any shape

FRST returns either a logical vector or a logical matrix result according to the mode of the argument.

If you specify logical vector *lv* as argument, the result will be a logical vector of the same shape as *lv* with a component set .TRUE. in the position corresponding to the position of the first .TRUE. component – if any – of *lv*. All other components of the result will be set .FALSE..

If you specify logical matrix *lm* as argument, the result will be a logical matrix of the same shape as *lm* with one component set .TRUE., the rest .FALSE.. The position of the .TRUE. component is determined by the position of the first .TRUE. component of *lm* when viewed in column-major order. If *lm* has no .TRUE. components then every component of the result is set .FALSE..

GET_BIT

GET_BIT(*object*, *n*)

where:

- *object* is a scalar, vector or matrix of any type, shape and data-length
- *n* is an integer scalar which takes a value between 1 and the data-length in bits of the first argument

GET_BIT returns a logical result which has the same mode as *object*. Each component in the result is set .TRUE. or .FALSE. according to whether the n^{th} bit of the corresponding component of *object* is 1 or 0. Bit 1 is the most significant.

GET_MAT

GET_MAT(a, *r1*, *c1*, *r2*, *c2*)

where:

- a is a matrix of any shape, type and data-length
- *r1*, *c1*, *r2*, and *c2* are 4-byte integer scalars

GET_MAT returns a matrix with dimensions (*r2*, *c2*) which is a submatrix selected from matrix a with its 'top-left' co-ordinate (*r1*, *c1*). The result matrix has the same type and data-length as matrix a.

The result has to be wholly contained within the matrix a so you have to make sure that:

$$
\begin{aligned}
&1 \leq r1 \leq \textbf{SIZE}(a, 1)\\
&1 \leq c1 \leq \textbf{SIZE}(a, 2)\\
&r2 \geq 1\\
&c2 \geq 1\\
&r1 + r2 - 1 \leq \textbf{SIZE}(a, 1)\\
&c1 + c2 - 1 \leq \textbf{SIZE}(a, 2)
\end{aligned}
$$

GET_VEC

GET_VEC(a, *r1*, *r2*)

where:

- a is a vector of any size, type and data-length
- *r1* and *r2* are 4-byte integer scalars

GET_VEC returns a vector with dimension *r2* selected from vector a starting at co-ordinate a(*r1*). The result vector has the same type and data-length as vector a.

The result has to be wholly contained within the vector a so you have to make sure that:

$$
\begin{aligned}
&1 \leq r1 \leq \textbf{SIZE}(a)\\
&r2 \geq 1\\
&r1 + r2 - 1 \leq \textbf{SIZE}(a)
\end{aligned}
$$

INDEX_VEC

CALL INDEX_VEC(*v*)

where:

- *v* is a 4-byte integer vector

INDEX_VEC is a subroutine taking as its single argument a 4-byte integer vector, *v*, which can have any number of components. On return, component *i* of the vector will contain the value *i*; that is, the components of the returned vector will contain the numbers 1, 2, 3, 4, ... , **SIZE**(*v*).

LMATC

LMATC(s, r, c)

where:

- s is a scalar of any type and data-length
- r and c are 4-byte integer scalars

LMATC returns a logical matrix with row and column dimensions of r and c.

The matrix result has identical columns, each containing a copy of the internal representation of s (possibly padded or truncated as discussed below). Component i of each column of the result is set .TRUE. if bit i of s is a 1 (bit 1 of s being at the most significant end), or set .FALSE. if bit i is a 0.

Section 4.3 describes the internal representation of scalars.

The internal representations are padded with zeros or truncated at the most significant end, as necessary, to give a representation r bits long.

(see also appendix B)

LMATR

LMATR(s, r, c)

where:

- s is an scalar of any type and data-length
- r and c are 4-byte integer scalars

LMATR returns a logical matrix with row and column dimensions of r and c.

The matrix result has identical rows, each containing a copy of the internal representation of s (possibly padded or truncated as discussed below). Component i of each row of the result is set .TRUE. if bit i of s is a 1 (bit 1 of s being at the most significant end), or set .FALSE. if bit i is a 0.

Section 4.3 describes the internal representation of scalars.

The above internal representations are padded with zeros or truncated at the most significant end, as necessary, to give a representation c bits long.

(see also appendix B)

MAT

MAT(*s*, *r*, *c*)

where:

- *s* is an scalar of any type and data-length
- *r* and *c* are 4-byte integer scalars

MAT returns a matrix object with *r* rows and *c* columns. The result matrix is of the same type and data-length as the scalar argument. Each component of the result takes the same value as that of the scalar argument.

(see also appendix B)

MATC

MATC(*v*, *c*)

where:

- *v* is a vector object of any type and data-length
- *c* is an 4-byte integer scalar

MATC returns a matrix of the same type and data-length as *v* and which has *c* columns and the same number of rows as there are components in *v*. Each column of the result matrix is a copy of *v*.

(see also appendix B)

MATR

MATR(*v*, *r*)

where:

- *v* is a vector object of any type and data-length
- *r* is an 4-byte integer scalar

MATR returns a matrix of the same type and data-length as *v* and which has *r* rows and the same number of columns as there are components in *v*. Each row of the result matrix is a copy of *v*.

(see also appendix B)

MAXP

$$\text{MAXP}(\left\{\begin{array}{c} v\,[, lv] \\ m\,[, lm] \end{array}\right\})$$

where:

- *v* is a vector of type integer or real and of any data-length and shape
- *m* is a matrix of type integer or real and of any data-length and shape
- *lv* is an optional logical vector of the same shape as v
- *lm* is an optional logical matrix of the same shape as *m*

MAXP returns a logical object of the same mode and shape as the first argument. The result has .TRUE. component(s) corresponding to the component(s) with maximum value in the first argument. All other components of the result are set .FALSE..

The second argument acts as a mask. The choice of maximum value is determined only from those values in the first argument corresponding to .TRUE. values in the second.

If you omit the second argument, the effect is as if you had specified it with all components .TRUE.; that is, the result gives the positions of all components in the first argument which have the maximum value.

MAXV

$$\text{MAXV}(\left\{\begin{array}{c} v\,[, lv] \\ m\,[, lm] \end{array}\right\})$$

where:

- *v* is a vector of type integer or real and of any data-length and shape
- *m* is a matrix of type integer or real and of any data-length and shape
- *lv* is an optional logical vector of the same shape as *v*
- *lm* is an optional logical matrix of the same shape as *m*

MAXV returns a scalar object of the same type and data-length as the first argument. The result is the maximum value in the first argument.

The second argument acts as a mask. The choice of maximum value is determined only from those values in the first argument corresponding to .TRUE. values in the second.

If you omit the second argument, the effect is as though you had specified it with all components .TRUE.; that is, the result gives the maximum value of any component in the first argument.

If every component of the second argument is .FALSE., the minimum possible value for the given type and data-length is returned (see section 4.3).

MERGE

MERGE(*a*, *b*, *c*)

where:

- *c* is a logical vector or matrix
- a and *b* are independently either scalars or objects (vectors or matrices) of the same mode and shape as *c*, the third argument. *a* and *b* must be of the same type and data-length

MERGE returns a vector or matrix object. Its third argument is a logical vector or matrix, and the mode and shape of the result is the same as the mode and shape of the third argument. Components of the result are selected from the first argument if the value of the corresponding component of the third argument is .TRUE. and from the second argument if the value is .FALSE..

The first and second arguments can be of any type but they have to be either of scalar mode, or the same mode and shape as the third argument. If scalar, they are implicitly expanded to be of the same mode and shape as the third argument.

If the first and second arguments are of fixed data-lengths, their data-lengths must be equal. If you give only one of these two arguments a fixed data-length, the data-length of the other argument is converted to that fixed data-length.

The type and data-length of the result are the same as the type and data-length of the first and second argument.

MINP

$$\text{MINP}(\left\{ \begin{array}{c} v\,[,\,lv] \\ m\,[,\,lm] \end{array} \right\})$$

where:

- *v* is a vector of type integer or real and of any data-length and shape
- *m* is a matrix of type integer or real and of any data-length and shape
- *lv* is an optional logical vector of the same shape as *v*
- *lm* is an optional logical matrix of the same shape as *m*

MINP returns a logical object of the same mode and shape as the first argument. The result has .TRUE. component(s) corresponding to component(s) in the first argument with the minimum value. All other components of the result are set .FALSE..

The second argument acts as a mask. The choice of minimum value is determined only from those values in the first argument corresponding to .TRUE. values in the second.

If you omit the second argument, the effect is as though you had specified it with all components .TRUE.; that is, the result has components set .TRUE. at positions corresponding to all components in the first argument with the minimum value.

MINV

$\textbf{MINV}(\left\{ \begin{array}{c} v\,[\,,\,lv] \\ m\,[\,,\,lm] \end{array} \right\})$

where:

- *v* is a vector of type integer or real and of any data-length and shape
- *m* is a matrix of type integer or real and of any data-length and shape
- *lv* is an optional logical vector of the same shape as *v*
- *lm* is an optional logical matrix of the same shape as *m*

MINV returns a scalar object of the same type and data-length as the first argument. The result is equal to the smallest component value in the first argument.

The second argument acts as a mask. The choice of minimum value is determined only from those values in the first argument corresponding to .TRUE. values in the second.

If you omit the second argument, the effect is as though you had specified it with all components .TRUE.; that is, the result gives the minimum value of any component in the first argument.

If every component in the second argument is .FALSE., the maximum possible value for the given type and data-length is returned (see section 4.3).

ORCOLS

ORCOLS(*lm*)

where:

- *lm* is a logical matrix of any shape

ORCOLS returns a logical vector which has as many components as there are rows in *lm*. The value of component *i* of the result is the logical OR of the values of all the components in row *i* of *lm*.

ORROWS

ORROWS(*lm*)

where:

- *lm* is a logical matrix of any shape

ORROWS returns a logical vector which has as many components as there are columns in *lm*. The value of component *i* of the result is the logical OR of the values of all components in column *i* of *lm*.

PAT_UNIT_DIAG **PAT_UNIT_DIAG**(n)

where:

- n is a 4 byte integer scalar

PAT_UNIT_DIAG returns a square logical matrix with the size of each dimension equal to n. Every component in the leading diagonal is set .TRUE., every other component is set .FALSE..

PAT_TRAIL_DIAG **PAT_TRAIL_DIAG**(n)

where:

- n is a 4 byte integer scalar

PAT_TRAIL_DIAG returns a square logical matrix with the size of each dimension equal to n. Every component in the trailing diagonal is set .TRUE., every other component is set .FALSE..

PAT_LOWER_TRI **PAT_LOWER_TRI**(n)

where:

- n is a 4 byte integer scalar

PAT_LOWER_TRI returns a square logical matrix with the size of each dimension equal to n. Every component in the lower triangle including the leading diagonal is set .TRUE., every other component is set .FALSE..

RANK **RANK**(*object*)

where:

- *object* is a data object of any mode, type or data-length

RANK returns a 4 byte integer scalar which is the total number of dimensions, parallel and non-parallel, in its argument. For a scalar argument which is not the name of a scalar array the result is zero.

REVC **REVC**(m)

where:

- m is a matrix object of any type, data-length and shape

REVC returns a matrix object of the same type and data-length as m but with its column ordering reversed; that is, component (i,j) of the result is equal to component (i, (**SIZE**(m, 2) + 1) - j) of the argument.

REVR

REVR(m)

where:

- m is a matrix object of any type, any data-length and any shape

REVR returns a matrix object of the same type and data-length as m but with its row ordering reversed; that is, component (i,j) of the result is equal to component ((**SIZE**(m, 1) $+$ 1) $-$ i, j) of the argument.

REV

REV(v)

where:

- v is a vector object of any type, any data-length and any shape

REV returns a vector of the same type, data-length, mode and shape as v.

The result consists of the components of v in reverse order.

ROW

ROW($\left\{ \begin{array}{l} s\,,\,r\,,\,c \\ v\,,\,r \end{array} \right\}$)

where:

- s, r and c are 4-byte integer scalars
- v is a 4-byte integer vector

ROW returns a logical matrix whose dimensions depend on the arguments you have given. If the first argument is a scalar, r and c give the row and column dimensions of the resulting matrix. If the first argument is vector v, the size of v gives the number of columns while r gives the number of rows of the resulting matrix.

If the first argument is scalar s with value i, components in row i of the matrix result are .TRUE.; all other components of the result are set .FALSE..

If the first argument is vector v then for each component in the vector (say component k, having value j) there will be a corresponding component of the matrix result that is .TRUE. (component (j, k)); all other components of the result are .FALSE..

If the value of s is not in the range 1 to r, all components of the result are .FALSE.; if the value of a component of v is not in the range 1 to r, all components in the corresponding column of the result are .FALSE..

(see also appendix B)

ROWN **ROWN**(*lm*)

where:

- *lm* is a logical matrix of any shape

ROWN returns an integer vector object of data-length 4 bytes with as many components as there are columns in the matrix argument. The n^{th} component of the vector result takes the lowest row number of any .TRUE. components in the n^{th} column of the matrix.

If all the components in a column of *lm* are .FALSE. then the corresponding component of the result is zero.

ROWS **ROWS**(i, j, r, c)

where:

- i, j, r and c are 4-byte integer scalars

ROWS returns a logical matrix object with all components in rows i to j inclusive set .TRUE. and all other components set .FALSE..

r and c give the row and column dimensions of the logical matrix result.

If either i or j is not in the range 1 to r or if i is greater than j then all the components of the result are .FALSE..

(see also appendix B)

SET_BIT **CALL SET_BIT**(*object*, *n*, *value*)

where:

- *object* is a data object of any mode, type, shape and data-length
- *n* is an integer scalar which takes a value between 1 and the data-length in bits of the first argument
- *value* is a logical object of the same mode and shape as the first argument, or is a logical scalar (in which case it is implicitly expanded to match *object*)

This subroutine has the inverse effect of **GET_BIT**. Its purpose is to assign a value to a particular bit of each component in a data object.

The third argument, *value*, is used to set the n^{th} bit of each component of the first argument to 1 or 0 according to whether the corresponding component of *value* is set .TRUE. or .FALSE.. Bit 1 is the most significant. All other bits in the first argument are unchanged.

SET_MAT

CALL SET_MAT(a, r, c, b **)**

where:

- a is a matrix of any shape, type and data-length
- r and c are 4-byte integer scalars
- b is a matrix object of the same type and data-length as the first argument

This subroutine places the contents of matrix b, as a submatrix, into matrix a starting at position $a(r,c)$ as the 'top-left' corner. All other components of matrix a are unchanged.

The submatrix has to be wholly contained within the matrix a so you have to make sure that:

$$\begin{aligned}
&1 \le r \le \mathbf{SIZE}(a\,,1)\\
&1 \le c \le \mathbf{SIZE}(a\,,2)\\
&r + \mathbf{SIZE}(b,1) - 1 \le \mathbf{SIZE}(a\,,1)\\
&c + \mathbf{SIZE}(b,2) - 1 \le \mathbf{SIZE}(a\,,2)
\end{aligned}$$

SET_VEC

CALL SET_VEC(a, r, b **)**

where:

- a is a vector of any shape, type and data-length
- r is a 4-byte integer scalar
- b is a vector of the same type and data-length as the first argument.

This subroutine places the contents of vector b into vector a starting at position a(r). All other components of vector a are unchanged.

The subvector has to be wholly contained within the vector v so you have to make sure that:

$$\begin{aligned}
&1 \le r \le \mathbf{SIZE}(a)\\
&r + \mathbf{SIZE}(b) - 1 \le \mathbf{SIZE}(a)
\end{aligned}$$

SHEC

SHEC$(m \left[, \left\{ \begin{matrix} s \\ v \end{matrix} \right\}\right])$

where:

- *m* is a matrix object of any type, data-length and shape
- *s* is an optional 4-byte integer scalar
- *v* is an optional 4-byte integer vector object with the same number of components as there are rows in the matrix argument, *m*

SHEC returns a matrix object of the same type, data-length and shape as the first argument, but shifted a number of places to the east, using CYCLIC geometry: the value(s) shifted out on the eastern edge are shifted in on the western edge. The number of places by which the matrix is to be shifted is determined by the second argument.

If the second argument is scalar *s*, the number of places to be shifted is the value of *s* taken modulo the number of columns in *m*.

If the second argument is vector *v*, each row of the matrix is shifted by the number of places given by the value of the corresponding component of *v* taken modulo the number of columns in *m*.

If you omit the second argument, the matrix is shifted one place to the east.

No shift occurs if the matrix has just one column.

SHEP

SHEP$(m \left[, \left\{ \begin{matrix} s \\ v \end{matrix} \right\}\right])$

As for **SHEC** except that the shift uses PLANE geometry. The value(s) shifted in at the western edge will be zero(s), .FALSE.(s), or null(s) depending on the type of *m*.

SHLC

SHLC(v [, *i*])

where:

- *v* is a vector object of any type, data-length and shape
- *i* is an optional 4-byte integer scalar

SHLC returns a vector of the same type and data-length as the first argument, but shifted a number of places to the left, using CYCLIC geometry: the number of places by which the vector is to be shifted is given by the second argument. The number of places shifted is the value *i* taken modulo the size of *v*.

If you omit the second argument, the number of places shifted is one.

No shift occurs if the vector has just one component.

SHLP

SHLP(v [, *i*])

As for **SHLC** except that the shift uses PLANE geometry. The value(s) shifted in at the right-hand end will be zero(s), .FALSE.(s) or null(s) dependent on the type of *v*.

SHNC

SHNC($m \left[, \left\{ \begin{array}{c} s \\ v \end{array} \right\} \right]$)

where:

- *m* is a matrix object of any type, data-length and shape
- *s* is an optional 4-byte integer scalar
- *v* is an optional 4-byte integer vector object with the same number of components as there are columns in the matrix argument, *m*

As for **SHEC**, except that the matrix is shifted to the north.

SHNP

SHNP($m \left[, \left\{ \begin{array}{c} s \\ v \end{array} \right\} \right]$)

As for **SHNC** except that the shift uses PLANE geometry.

SHRC

SHRC (v [, *i*])

As for **SHLC** except that the vector is shifted to the right.

SHRP

SHRP (v [, *i*])

As for **SHRC** except that the shift uses PLANE geometry.

SHSC

SHSC($m \left[, \left\{ \begin{array}{c} s \\ v \end{array} \right\} \right]$)

As for **SHNC** except that the matrix is shifted to the south.

SHSP

SHSP($m \left[, \left\{ \begin{array}{c} s \\ v \end{array} \right\} \right]$)

As for **SHSC** except that the shift uses PLANE geometry.

SHWC

SHWC($m \left[, \left\{ \begin{array}{c} s \\ v \end{array} \right\} \right]$)

As for **SHEC** except that the matrix is shifted to the west.

SHWP

$\textbf{SHWP}(m \left[, \left\{ \begin{array}{c} s \\ v \end{array} \right\} \right])$

As for **SHWC** except that the shift uses PLANE geometry.

SIZE

SIZE(*object* [, *dim*])

where:

- *object* is a variable or array of mode vector or matrix and of any type, shape or data-length
- *dim* is an optional 4 byte integer scalar in the range 1 to the number of dimensions of *object* inclusive

With just one argument, *object*, **SIZE** returns the total number of components in *object*. If you specify *dim*, its value is taken to select a dimension of *object*. In this case **SIZE** returns the extent of that dimension. If *dim* is outside the range (1 to the number of dimensions of *object*) then either an error will be detected, depending on the level of error checking enabled (see *DAP Series: Program Development*), or the result is undefined. If *object* is a scalar, an error will occur. The function returns a 4-byte integer scalar value.

Thus, given the declaration:

```
INTEGER MA(*50,*100,3)
```

then:

```
SIZE(MA) gives the result 15000
SIZE(MA,1) gives the result 50
SIZE(MA,3) gives the result 3
```

Unlike other built-in functions **SIZE** can be used in declarations (see sections 6.1.2, 10.4.5).

SUM

$\text{SUM}(\left\{ \begin{matrix} v \\ m \end{matrix} \right\})$

where:

- *v* is a vector object of type integer, real or logical, of any data-length and shape
- *m* is a matrix object of type integer, real or logical, of any data-length and shape

SUM returns a scalar object of type integer or real. The result is the sum of the values of all the components of the argument. If the argument is of type logical, its components are treated as integers of value of 1 or 0, depending on whether the components are .TRUE. or .FALSE. respectively.

If the argument is of type real the result will be a real scalar of the same data-length as the argument; if the argument is of type logical the result is an integer scalar of data-length 4 bytes; if the argument is of type integer the result is an integer scalar of data-length 4 bytes (for arguments of data-length 4 bytes and less) or 8 bytes (for arguments of data-length greater than 4 bytes).

SUMC

SUMC(*m*)

where:

- *m* is a matrix object of type integer, real or logical, of any data-length and shape

SUMC returns a vector object of type integer or real. The number of components in the result vector is the same as the number of rows in *m*.

Each component of the vector result is the sum of the values of all components in the corresponding row of *m*. If *m* is of type logical, its components are treated as integers of value 1 or 0, depending on whether they are .TRUE. or .FALSE. respectively.

If *m* is of type real the result is a real vector of the same data-length as *m*. If *m* is of type logical the result is an integer vector of data-length 4 bytes; if *m* is of type integer the result is an integer vector of data-length 4 bytes (for arguments of data-length 4 bytes and less) or 8 bytes (for arguments of data-length greater than 4 bytes).

SUMR

SUMR(m)

where:

- m is a matrix object of type integer, real or logical, of any data-length and shape

SUMR returns a vector object of type integer or real. The number of components in the result vector is the same as the number of columns in m.

Each component of the vector result is the sum of the values of all components in the corresponding column of m. If m is of type logical, its components are treated as integers of value 1 or 0, depending on whether they are .TRUE. or .FALSE. respectively.

If m is of type real the result is a real vector of the same data-length as m. If m is of type logical the result is an integer vector of data-length 4 bytes; if m is of type integer the result is an integer vector of data-length 4 bytes (for arguments of data-length 4 bytes and less) or 8 bytes (for arguments of data-length greater than 4 bytes).

TRAN

TRAN(m)

where:

- m is a matrix object of any type, data-length and shape

TRAN returns a matrix object of the same type and data-length, which is the transpose of m. That is, a matrix of shape r rows and c columns is transposed to a matrix of c rows and r columns with component (i,j) in the result equal to component (j,i) in m for all values of i and j.

VEC

VEC(s, r)

where:

- s is a scalar object of any type and data-length
- r is an 4-byte integer scalar

VEC returns a vector object of the same type and data-length as s and the same size as r. Each component of the vector result is a copy of s.

(see also appendix B)

11.3 Computational error management procedures

FORTRAN-PLUS allows you to suppress interrupts caused by a certain class of run-time errors called computational errors, and also to report the positions of vector or matrix components where such errors have occurred. This is achieved through a number of built-in procedures summarised here. Chapter 15 describes computational error management.

NOM_EMSK

CALL NOM_EMSK(*l*)

where:

- *l* is a logical variable of any shape and mode

The **NOM_EMSK** subroutine establishes *l* as the error mask for the corresponding mode. *l* remains in force until either:

- another call of **NOM_EMSK** having an argument of the mode as *l*
- the calling procedure returns (when the mask in force when the calling procedure was entered is reinstated)
- a call of **RSTEMSKS** is made

The significance of the error mask is described in Chapter 15.

RST_EMSKS

CALL RST_EMSKS

The **RST_EMSKS** subroutine which takes no arguments, reinstates all the error interrupt masks (ie. those for scalar, vector and matrix mode) in force at entry to the current procedure (see chapter 15).

NOM_ERPT

CALL NOM_ERPT(*l*)

where:

- *l* is a logical variable of any mode and shape

The **NOM_ERPT** subroutine establishes *l* as the error reporting mask for the corresponding mode. *l* remains in force until either:

- another call of **NOM_ERPT** having an argument of the same mode as *l*
- the calling procedure returns (when the mask in force when the calling procedure was entered is reinstated)
- a call of **RST_ERPTS** is made

The significance of the error reporting mask is described in chapter 15.

RST_ERPTS

CALL RST_ERPTS

The **RST_ERPTS** subroutine which takes no arguments, reinstates all the error reporting masks (ie. those for scalar, vector and matrix mode) in force at entry to the current procedure (see chapter 15).

GET_ERPT

CALL GET_ERPT(*ls*)

where:

- *ls* is a logical scalar

The **GET_ERPT** subroutine sets *ls* to the value of the global error status flag. The value of the flag is .TRUE. if any computational error (regardless of the mode, shape or size of the operands) has occurred since the last call of **CLR_ERPT** (or since the beginning of the program if **CLR_ERPT** has not been called). Otherwise it is .FALSE.

CLR_ERPT

CALL CLR_ERPT

The **CLR_ERPT** subroutine clears the global error status flag (see **GETERPT**).

SET_STATE

SET_STATE(*i*)

where:

- *i* is a 4 byte integer scalar

The **SET_STATE** function sets the global error interrupt switch to the value of its argument *i*, and thus specifies how subsequent computational errors are to be handled. **SET_STATE** returns a 4 byte integer scalar giving the value that the global error interrupt switch had when **SET_STATE** was called.

A summary of the meaning of the switch values is given below. See chapter 15 for full details.

State	*Error interrupt switch settings*
0	Ignore arithmetic underflow only
1	All errors interrupt
2	No errors interrupt, all reported
3	As 2 but underflow not reported
4	As 2 but only underflow reported

11.4 Mode conversion routines

11.4.1 Host – DAP conversion routines

Data areas in a DAP program are independent of those in an associated host program, so if there is a need to share data between host and DAP, you have to pass the data explicitly between the machines. You pass the data by calling the host routines DAPSEN and DAPREC (described in the *DAP Series: Program Development manual* for the appropriate host computer) from within your host program. In general, both the layout of data and its number representation differ between the host and the DAP. Thus, after sending data to the DAP you normally need to convert it to DAP format; similarly any DAP data that is to read and interpreted by the host needs to be converted to host format. You perform these conversions by calling routines within the FORTRAN-PLUS program. Host data is always transferred to or from a COMMON area in the DAP program. The routines described below will normally only be meaningful if a conversion is being done to or from the beginning of such a COMMON area. Any other use will demand a careful understanding of how data is mapped in DAP memory (see appendix A).

The routines to do this conversion are described on the next two pages:

CONVHTOD

CONVHTOD(*variable*[, *n*])

where:

- *variable* is a variable, array element name or array name; it thus denotes a scalar, vector or matrix (or an array of one of these)
- *n* is an optional integer scalar

The **CONVHTOD** subroutine converts from host to DAP format. If *n* is not specified, the routine will convert from host format as many elements as there are in *variable*. The conversion is done *in situ*, that is the host data is assumed to start at the same address as *variable*. (Normally this will be the beginning of the COMMON area into which host data has been sent).

If *n* is present, it specifies the number of elements (scalars, vectors or matrices) of the mode of *variable* which are converted. Generally *variable* will be an array and *n* will be used to limit the number of elements which are converted. *n* can be greater than the number of elements of *variable*, in which case it is assumed that further elements of the same mode follow *variable* in consecutive memory locations. This will be the case if there are declarations following in a COMMON area.

There are certain constraints on the alignment of scalars used as first arguments to **CONVHTOD**; you are recommended to use as an argument to **CONVHTOD** a scalar that is at the beginning of a COMMON block. The advanced user may wish to consult the details of data mapping in appendix A.

Examples

Assuming the declarations:

```
COMMON/HOST/ VS(*50,3)
COMMON/HOST1/A(*40,*40),B(*40,*40)
INTEGER VS
REAL A,B
```

then:

CALL CONVHTOD(VS)
converts 150 integers in VS

CALL CONVHTOD(VS,1)
converts 50 integers in the first vector of VS

CALL CONVHTOD(A,2)
converts 3200 reals in the two matrices A and B

CONVDTOH **CONVDTOH**(*variable*[, *n*])

where:

- *variable* is a variable, array element name or array name; it thus denotes a scalar, vector or matrix (or array of one of these) with a defined type, data-length and size
- *n* is an optional integer scalar

The **CONVDTOH** subroutine converts from DAP to host format. If *n* is not specified, the routine will convert to host format as many elements as there are in *variable*. The conversion is done *in situ*, that is the host data will be left at the same address as *variable*. (Normally this will be the beginning of the COMMON area from which host data will be received by the host program).

If *n* is present, it specifies the number of elements (scalars, vectors or matrices) of the mode of *variable* which are converted. Generally *variable* will be an array and *n* will be used to limit the number of elements which are converted. *n* can be greater than the number of elements of *variable*, in which case it is assumed that further elements of the same mode follow *variable* in consecutive memory locations. This will be the case if there are declarations following in a COMMON area.

There are certain constraints on the alignment of scalars used as first arguments to **CONVDTOH**; you are recommended to use as an argument to **CONVDTOH** a scalar that is at the beginning of a COMMON block. The advanced user may wish to consult the details of data mapping in appendix A.

In one case, there can be more space occupied by converted host data than by the DAP data and care must be taken not to overwrite other variables in memory. This case is where *variable* is a logical vector or logical matrix. In host format each converted component will occupy one 4 byte word. The easiest way to ensure there are no problems is to declare a scalar array equivalanced to *variable*:

```
LOGICAL LV(*500)
INTEGER DUMMY(500)
EQUIVALENCE(LV,DUMMY)
```

DUMMY is declared with the same number of elements as LV, ensuring that as much space as necessary is reserved for the host format data.

11.4.2 DAP to DAP mode conversion routine

A routine is provided to convert data between scalar mode and either vector or matrix mode.

CONVDTOD CONVDTOD(*variable*$_1$, *variable*$_2$ [, *n*])

where:

- *variable*$_1$ and *variable*$_2$ are variables, array elements or array names, of which one must be of scalar mode and the other must be of vector or matrix mode (the parallel argument)
- *n* is an optional integer scalar

variable$_1$ and *variable*$_2$ must be of the same rank, type and data-length, with corresponding values of each dimension equal. If *n* is absent, the routine converts all the components of *variable*$_1$ to the appropriate mode in *variable*$_2$.

If *n* is present, it specifies the number of vectors or matrices (according to the mode of the parallel argument) which are to be converted. Generally *n* will be less than or equal to the number of elements in the parallel argument, but it can be more; in this case the routine assumes that further objects of the appropriate mode follow *variable*$_1$ and *variable*$_2$ in consecutive memory locations.

variable$_1$ and *variable*$_2$ can be storage associated to each other, but only if they have coincident starting addresses; there cannot be a partial overlap.

Examples

Assuming the declarations:

```
INTEGER*1 A(50,3), B(*50,3)
REAL M(*50,*50), S(50,50)
```

then:

```
CALL CONVDTOD(M,S)
```
converts the components of matrix M into the scalar array S

```
CALL CONVDTOD(A,B,2)
```
converts the first two columns of the scalar array A into the first two vectors in B

Chapter 12

COMMON blocks

COMMON storage allows storage to be shared between different program units, so that you can pass values between program units by means other than argument association (see section 10.4.3). Objects in COMMON storage are arranged in *COMMON blocks*, and the arrangement of data within each COMMON block is under the control of the user. Each COMMON block is identified by a name, the *COMMON block name* (see section 3.2.1).

COMMON blocks also provide a means whereby you can transmit values in the host program to the DAP program, and vice versa. There is no argument association between a host program unit and a DAP program unit, since the DAP program is entered from the host program at a FORTRAN-PLUS ENTRY SUBROUTINE (see Chapter 14), which has no arguments.

If a particular program unit is to use a particular COMMON block, you have to specify in COMMON statements in the program unit, the order in which objects from the program unit are to be stored in the COMMON block.

12.1 COMMON blocks

Within a program unit you can declare that any of the variables or arrays in that program unit are to be held in a particular area of storage – a COMMON block – which you refer to by a COMMON block name. If you then declare a COMMON block of the same name in another DAP program unit, both program units will refer to the same area of storage and will therefore share the values held in that area, although you can use different names in each program unit to refer to the same part of the COMMON block.

Any COMMON block that you refer to in a FORTRAN-PLUS program unit will be allocated storage in array store.

You insert variables and arrays into a COMMON block by COMMON statements or by equivalencing an object with an object already inserted into a COMMON block by a COMMON statement (see chapter 13). There is no type or mode associated with a COMMON block, and it can contain any number of objects of different types and modes.

12.1.1 The COMMON statement

The **COMMON** statement has the form:

COMMON/$name_1$/$list_1$/$name_2$/$list_2$/ ... /$name_n$/$list_n$

where:

- each $name_i$ is a COMMON block name, and can be the same as any variable or array name in any program unit that references the COMMON block. The same COMMON block name can appear more than once in COMMON statements in the same program unit, in which case the variables and/or arrays in the associated list are added to the end of the specified COMMON block in the order in which they appear. The only restriction on COMMON block names is that they have to be distinct from all FORTRAN-PLUS program unit names
- each $list_i$ has the form:

  $object_1$, $object_2$, ... $object_k$

  where each $object_j$ is a variable name, array name or declarator (which can declare the mode and dimensions of an array or a vector or matrix variable). An object that appears in the dummy argument list of the subprogram in which the COMMON statement occurs cannot appear in the list of names in that COMMON statement

The effect of the COMMON statement is to map the variable and arrays in each list onto the area of storage identified by the COMMON block name. Items are mapped onto the COMMON block in the order in which they appear in the list, and are stored contiguously, except where the need to align variables on certain boundaries (see appendix A) forces a gap to be created (see section 13.1).

You can insert an object into a COMMON block by equivalencing it with an object already held in the COMMON block.

For example:

```
INTEGER I, J, K, M, N, P, Q
LOGICAL L(30)
COMMON/BLK2/I, J, K, M, N, P, Q
EQUIVALENCE(N, L(2))
```

The COMMON block BLK2 now contains the integer scalars I, J, K, M, N, P, Q and the logical array L, where the second element of L shares storage with N and other areas of store are shared.

An EQUIVALENCE statement can extend a COMMON block beyond its end, as in the above example, but cannot extend it beyond its beginning.

Hence, the statement:

```
EQUIVALENCE(J,L(3))
```

would have been illegal, as it would have attempted to extend the COMMON block beyond its beginning, to 'accommodate' the first elements of L.

Although the example shows a storage association between data of types integer and logical, because of the various ways in which different types and modes of data are mapped on the DAP, you should generally avoid using COMMON blocks to associate storage between objects of different types, modes or shapes (see section 13.2). However, if you decide to attempt it, you need to know how data is mapped onto DAP memory (see appendix A).

12.1.2 Initialising COMMON blocks

You can initialise variables and arrays in COMMON blocks in type specification statements, DATA statements, or by means of a block data subprogram.

You can initialise a COMMON block from any program unit that refers to it, but any particular item in a COMMON block should only be initialised once.

Note that COMMON blocks with the same name in DAP and host program units are not associated and variables initialised in one are not automatically initialised in the other.

12.2 Using COMMON blocks

COMMON blocks allow the sharing of storage between program units. You need to specify in each program unit that uses a COMMON block, the individual objects in that block. A variable, array, or component in one program unit is in COMMON association with a variable, array, or component in another program unit if the positions you have assigned to each in the same COMMON block coincide or overlap. Respective positions within the COMMON block, and not the names of the objects concerned, entirely determine the association.

12.2.1 Block data subprograms

Although you can initialise objects in named COMMON blocks as described in section 12.1.2 and chapter 6, a particular class of program unit, the *block data subprogram*, is available for the sole purpose of initialising named COMMON blocks. Block data subprograms are provided for compatibility with other versions of FORTRAN in which they provide the only means of initialising COMMON blocks.

A block data subprogram begins with a **BLOCK DATA** statement, which can take either of the following forms:

> **BLOCK DATA** *name*
> **BLOCKDATA** *name*

where *name* is the name of the block data subprogram.

The BLOCK DATA statement is followed by the required COMMON, DIMENSION, EQUIVALENCE, IMPLICIT, PARAMETER and type specification statements, and the DATA statements to perform the initialisations. The subprogram finishes with an END statement.

12.2.2 Communicating between host and DAP programs

An important use of COMMON blocks in FORTRAN-PLUS is in passing values between a DAP program and the host program. Named COMMON blocks are the only means by which such a transfer can take place, since argument association is not possible between a host program and a DAP program unit.

You can transfer data to be passed between a host and a DAP program into or out of COMMON blocks in the latter. By making calls from the host program to the interface subroutines DAPSEN and DAPREC, you can pass data explicitly between the host program and the DAP program (or vice versa). For further information on the use of these interface subroutines see *DAP Series: Program Development.*

Particular attention should be paid to the alignment of objects in COMMON blocks. When a host program unit sends data to a FORTRAN-PLUS COMMON block, the objects transmitted are held in array store in host storage mode (see chapter 13). Before the values of these objects can be correctly interpreted by a FORTRAN-PLUS program unit you have to convert them into scalar, vector or matrix storage mode, as appropriate, using the storage mode conversion subroutines listed in section 11.4.1.

Chapter 13

Storage association

Two or more data objects (that is, variables, arrays or components) are said to be *associated* if the storage locations to which they refer overlap completely or partially.

There are three circumstances under which data can become associated:

1. EQUIVALENCE association. You can associate two or more objects by putting them together in an EQUIVALENCE statement (see section 13.1)
2. COMMON association. Objects occupying the same or overlapping areas in the same COMMON block used in different program units are said to be in common association. COMMON association is described in chapter 12
3. Argument association. Association takes place between the objects in the actual argument list of a CALL statement or function reference and the corresponding items in the dummy argument list of a subroutine or function subprogram. Argument association is described in section 10.4

Otherwise all data objects occupy distinct storage locations.

13.1 The EQUIVALENCE statement

You can make two or more data objects share the same storage location by naming them in an **EQUIVALENCE** statement of the form:

EQUIVALENCE ($namelist_1$), ($namelist_2$), ... ($namelist_n$)

where each $namelist_i$ has the form:

$name_1, name_2, \ldots name_k$

and has to contain at least two entries. None of the names in any $namelist_i$ can appear in the dummy argument list of the subprogram in which the EQUIVALENCE statement appears.

Each *name*$_j$ can be the name of a scalar, vector, or matrix variable or array or array element of any type and each variable or array named must have every dimension specified by an integer constant.

Selection of an array element from an array has to have basic integer constants in the non-parallel subscript positions and null in any parallel subscript positions.

If any of these *name*$_j$ objects are also in COMMON or are EQUIVALENCEd to an object in COMMON, then further restrictions can apply:

- If you have EQUIVALENCEd an array (or variable) to an object declared in a COMMON block then the array (or variable) is also held in that COMMON block. If the EQUIVALENCEd array (or variable) occupies more storage than the object then it will overlap subsequent variables in the COMMON block, or extend the COMMON block beyond its last entry or both. You cannot extend a COMMON block at the beginning by means of equivalencing variables into that COMMON block
- You are not permitted to equivalence two objects whose relative storage positions have already been fixed within the program unit; that is, objects that are already linked by a chain of association and assignments to COMMON blocks
- You are not permitted to equivalence an object in a COMMON block to any object that is already in a COMMON block
- You are not permitted to alter the alignment of an object in a COMMON block by the use of EQUIVALENCE

13.2 Effect of EQUIVALENCE

The effect of EQUIVALENCE is to map each data object *in namelist*$_i$ onto array store in such a way that, if the data object is an array, each element of the array, or, if the data object is a variable or array element, each begins in the same word of storage, although the objects can be of different sizes. The reasons for doing this are:

- To conserve storage
- To allow stored data to be interpreted in different ways

Subject to the above rules, you can equivalence objects of any type, data-length and mode to reduce storage needed, although you have to take care not to overwrite needed data.

In order to use storage association to address data in different ways, you need to understand how FORTRAN-PLUS data objects are stored in DAP memory. Storage mappings are given in detail in chapter 17. However there are certain straightforward cases:

- Equivalencing two objects of identical mode, type, data-length and shape simply makes both objects occupy exactly the same storage
- Equivalencing scalar variables results in coincidence at the least significant end unless one object is less than 4 bytes and the other is greater, when coincidence is at the most significant end

- Arrays can be meaningfully equivalenced with each other or with vectors or matrices, provided that the array elements and vectors or matrices are of identical mode, type, data-length and shape. Ordering of the elements of an array is the normal FORTRAN ordering (the first index varies most quickly)

For example:

```
INTEGER VI(*5,3), V(*5),VIA(*5,2,4)
EQUIVALENCE(VI(,2),V)
EQUIVALENCE(VI,VIA(,1,3))
```

The three vectors in VI are respectively associated with VIA(,1,3), VIA(,2,3) and VIA(,1,4) and the vector V is the same as VIA(,2,3).

13.3 Definition of values

Within FORTRAN-PLUS all variables have type, data-length and mode, and, in the case of vector or matrix variables, shape. The ways in which you can use variables are clearly defined and checked in some aspects at compile-time, in other aspects during run-time. For example, in a subroutine or function dummy argument list, you can declare a variable as type real, and there is a check at run-time that the actual argument passed is also of type real. Because FORTRAN-PLUS data objects can be dynamically-sized, checks for conformance (that is, the circumstances in which the shapes of different variables should match) are made, in general, during run-time. There are options within the compilation system and at run-time to disable some or all of the checks which are made by default (see *DAP Series: Program Development*). Hence it is important that you understand the language rules and the implications of not following the rules in certain circumstances.

FORTRAN-PLUS variables can be divided into four classes:

- Preset variables
- COMMON variables
- Local (on stack) variables
- Static non-preset variables (that is, EQUIVALENCEd but not in COMMON or preset)

Variables can acquire defined values in a number of ways:

- Preset variables are given initial values in DATA or type specification statements or COMMON variables can be preset by means of a BLOCK DATA subprogram
- All variables can be given defined values by assignment during the execution of a procedure or by association which could be argument association, COMMON association or EQUIVALENCE association

When the DAP program is loaded, space is allocated for preset variables and they are initialised as specified in the program. Space allocated for COMMON variables which are not initialised and static non-preset variables will contain random bit patterns. When a subroutine or function is entered its local variables are allocated space on a stack but are not initialised. Attempting to select a variable before it has been initialised will lead to unpredictable results and often to a run-time error.

A vector or matrix variable can contain a mixture of initialised and un-initialised components if assignment to the variable is masked by an indexing construct. You are recommended to fully initialise all vector and matrix variables before use.

The FORTRAN-PLUS EQUIVALENCE statement allow two or more variables to occupy the same location in array store. It is also possible for one variable to occupy part of the storage used by another variable. In such cases, assigning a value to one variable will alter the value to EQUIVALENCEd variable(s) in a consistent (but not necessary meaningful) way.

For example, consider the code fragment:

```
INTEGER I
REAL R
EQUIVALENCE (I,R)
I=1
```

The value 1 is assigned to I, but R will now contain un-normalised real value (see section 4.3.2). Any attempt to select R before re-assigning it will result in a run-time error.

A similar effect can occur through argument association if the type, length and mode of an actual argument do not match exactly with those of the corresponding dummy argument.

Due to data alignment, some objects occupy more space than they actually use (see appendix A). You should not make any assumptions about unused bits being defined when the variable is given a value.

When data is transmitted from the host to the DAP, it arrives in host storage mode and you have to convert to scalar, vector or matrix storage mode as appropriate, before referring to the data from your FORTRAN-PLUS program (see section 11.4.1). Any attempt to refer to such data before conversion will probably result in a run-time error. Similarly, once you have converted to host storage mode, data which is to be sent to the host from the DAP, you should not refer to that data again in FORTRAN-PLUS unless you re-convert it to DAP format.

Chapter 14

Control flow statements

This chapter considers the sequence of execution of the various program units that constitute a FORTRAN-PLUS program, the means by which control is passed to or from a program unit, the sequence of execution within a program unit, and the FORTRAN-PLUS control statements with which you can alter, suspend or terminate the sequence of execution.

The typographical conventions used in this chapter for describing the form and syntax of statements are the same as the conventions used in chapter 2.

14.1 Start of execution

The host program is entered first, and, when it wants to use the DAP, has to explicitly load the DAP object program into the DAP code store by calling the interface subroutine DAPCON. When the subroutine DAPENT is called control passes to a specific FORTRAN-PLUS ENTRY subroutine. Data transfer to the DAP is performed by a call to the interface subroutine DAPSEN before control passes to the DAP, and data is received from the DAP by a call to DAPREC after control has passed back to the host program. Use of the interface subroutines is described in *DAP Series: Program Development.*

14.2 Execution within the DAP program

Normally control passes from an executable statement to the executable statement that follows it physically in the sequence of statements making up a program unit. However, you can modify this normal sequence by the control statements described in this chapter, which transfer control conditionally or unconditionally to labelled statements within the same program unit or to the first executable statement in another program unit.

When a CALL statement or any statement involving the evaluation of a call to a FORTRAN-PLUS function is executed, control passes to the first executable statement of either the called FORTRAN-PLUS subroutine or function. Control proceeds through this program unit as normal until a RETURN statement is executed, at which point control passes back to the executable statement following to the CALL statement or back to the executable statement containing the function call in the invoking program unit.

Control returns from the DAP program to the host program when a RETURN statement is executed in the procedure directly called by DAPENT in the host program. The statement following the call to DAPENT in the host program is then executed. Control proceeds from this point through the host program until either the host program terminates, or another call to an interface subroutine is executed.

The STOP statement causes control to be passed back to the host. The PAUSE statement temporarily suspends execution of the program while control passes to the *program state analysis mode* (PSAM). The TRACE statement passes control temporarily to the host for the output from the statement on the host.

If at any time during execution of the DAP program a STOP statement is executed, the execution of the DAP program and the associated host program is terminated. For the effect of the PAUSE statement, see *DAP Series: Program Development*. The TRACE statement is described in Chapter 15.

The CONTINUE statement is a dummy executable statement; it can appear wherever an executable statement can appear but it has no effect.

14.2.1 End of a program unit

The last executable statement of a program unit should be a GO TO, computed GO TO, arithmetic IF, RETURN or STOP statement.

If the last executable statement of a program unit is a computed GO TO statement or arithmetic IF statement, then it must always transfer control to a labelled statement elsewhere in the program unit.

14.3 Unconditional transfers of control

You can unconditionally transfer control to a point within a different DAP program unit in one of two ways:

1. Through a CALL statement referring to a subroutine subprogram in the DAP program
2. Through a function call referring to a function subprogram in the DAP program

Note that a CALL statement can refer to the same subprogram in which that CALL statement occurs, in which case control remains within the same program unit, but in a new invocation. Entry to and exit from procedures are described in chapter 10.

You can unconditionally transfer control to a labelled executable statement in the same program unit using the unconditional **GO TO** statement, which has either of the forms:

GO TO *label*
GOTO *label*

where:

> *label* is a reference to the label of an executable statement in the same program unit; control transfers to this statement on execution of the GO TO statement.

You must label the executable statement, if any, following the GO TO statement, since otherwise it could never be executed.

You can label a GO TO statement, although you cannot make its label the one referred to within the statement. The label given in the statement cannot be within the range of a DO loop (see section 14.3), unless the GO TO statement is within the range of the same DO loop.

You can terminate the execution of the entire FORTRAN-PLUS program and its associated host program from within a DAP program unit by using the STOP statement, or suspend execution by using the PAUSE statement:

- The **STOP** statement has the form:

  STOP *basic-integer-constant*

  where:

  basic-integer-constant is in the range 0 to 99999

  When **STOP** is executed it produces an error message including the constant and then abandons the DAP and host programs.

- The **PAUSE** statement has the form:

  PAUSE *basic-integer-constant*

  where:

  basic-integer-constant is in the range 0 to 99999

  When a **PAUSE** statement is executed the DAP program is suspended and *program state analysis mode* (PSAM) is entered. Depending on the run-time options you have selected, the DAP program can be restarted at the statement after the **PAUSE**. Therefore, you should not make the **PAUSE** statement the last executable statement in a subprogram if you intend to restart the program (see *DAP Series: Program Development*).

14.4 Conditional transfers of control

A number of control statements are available in which you can make the occurrence or the destination of a transfer of control dependent on some condition.

14.4.1 The computed GO TO statement

- The computed **GO TO** statement has either of the forms:

  GO TO (*label*$_1$, *label*$_2$, ... *label*$_n$), *arithmetic-expression*
  GOTO (*label*$_1$, *label*$_2$, ... *label*$_n$), *arithmetic-expression*

where:

- each *label*$_i$ is a reference to the label of an executable statement in the same program unit, or is the digit 0, or is null (that is, only the associated comma appears). The *label*$_i$ need not be distinct
- *arithmetic expression* produces an integer scalar object which is converted to 4 bytes if it is not already that data-length. The comma preceding *arithmetic expression* is optional

When a computed GO TO statement is executed, the integer value of *arithmetic expression*, say k is calculated. Control then transfers to the executable statement labelled by *label*$_k$. If k is not in the range 1 to n, or if *label*$_k$ is 0 or null, control passes to the next executable statement. Consequently, if the computed GO TO statement is the last executable statement in a program unit, *label*$_k$ should not be null or zero – if *label*$_k$ is null or zero, the effects are undefined.

If you have labelled the GO TO statement, you cannot refer to its label within the same statement.

None of the labels in the label list can be in the range of a DO loop (see section 14.5), unless the GO TO statement is within the range of the same DO loop.

14.4.2 The block IF construct

- The block IF construct has the form:

```
IF (logical-expression1) THEN
      IF-block
ELSE IF (logical-expression2) THEN
      ELSE IF-block
      . . .
      . . .
ELSE
      ELSE-block
END IF
```

where:

- *(logical-expression*$_1$*)* and *(logical-expression*$_2$*)* produce logical scalar results
- *IF-block*, *ELSE IF-block* and *ELSE-block* consist of zero or more executable statements
- there can be zero or one ELSE statement with a corresponding *ELSE-block* which is valid only after the IF statement and before the END IF statement
- there can be zero or more ELSE IF statements each with a corresponding *ELSE IF-block*. They have to appear after the IF statement and before the ELSE statement (if it appears) and before the END IF statement
- the end of the entire block IF construct is marked by an END IF statement
- ELSE IF and END IF can each be written without the separating space

When a block IF construct is executed, if *logical-expression*$_1$ is .TRUE. then the statements within the *IF-block* are executed and then control passes to the END IF statement. If *logical-expression*$_1$ is .FALSE. and there are one or more ELSE IF statements present then their logical expressions are evaluated in turn. If a logical expression is .TRUE. then evaluation of the expressions ceases, the statements within the corresponding *ELSE IF-block* are executed and control passes to the END IF statement. If there are no ELSE IF statements or their logical expressions are all .FALSE. then control passes to the *ELSE-block*, or if there is no such block, to the END IF statement.

You can nest block IF constructs by enclosing them within the *IF-block*, *ELSE IF-block* or *ELSE-block* of another block IF construct.

You have to observe the following restrictions on transferring control:

- If an *IF-block*, *ELSE-block* or *ELSE IF-block* contains a DO statement the the DO-loop must be wholly contained within the *IF-block*, *ELSE-block* or *ELSE IF-block*
- If a DO loop contains a block IF statement the corresponding END IF statement must also be contained in the same DO loop
- Control cannot be transferred by using GOTO, computed GOTO or arithmetic IF statements:
  * into an *IF-block*, *ELSE-block* or *ELSE IF-block* from outside that *IF-block*, *ELSE-block* or *ELSE IF-block*
  * to an END IF, an ELSE or an ELSE IF statement

14.4.3 The arithmetic and logical IF statements

- The arithmetic **IF** statement has the form:

  IF (*arithmetic-expression*) $label_1$, $label_2$, $label_3$

  where:

  - *arithmetic-expression* produces an integer or real scalar object
  - $label_1$, $label_2$ and $label_3$ are references to labels of executable statements in the same program unit, or zero, or null (that is, only the associated comma appears); they need not be distinct

  When the IF statement is executed, *arithmetic-expression* is evaluated. Control then transfers to the executable statement labelled by $label_1$ $label_2$, or $label_3$, depending on whether the value is less than, equal to, or greater than zero respectively. If the referenced label is zero or null control passes to the next executable statement.

  If you have labelled the IF statement, you cannot refer to its label within the same IF statement. None of the labels in the IF statement can be in the range of a DO loop (see section 14.5), unless the IF statement is in the range of the same DO loop.

  You are not recommended to test real objects against zero, due to the potential inaccuracy inherent in the representation of real values.

- The logical **IF** statement has the form:

  IF (*logical-expression*) *executable-statement*

  where:

  - *logical-expression* produces a scalar object
  - *executable-statement* is any executable statement other than a DO statement or another logical IF statement

When a logical IF statement is executed, *logical-expression* is evaluated. If the value is .TRUE., *executable-statement* is executed, otherwise control passes to the next executable statement.

If *executable-statement* is an unconditional GO TO statement, a computed GO TO statement, or an arithmetic IF statement, the logical IF statement becomes a conditional transfer of control, dependent on the value of *logical-expression* (and also on the value of the arithmetic expression in the computed GO TO or arithmetic IF statement).

14.5 Loops

A loop is a series of executable statements that are to be executed a number of times in succession. A loop is initiated by a DO statement, which specifies the extent of the loop and the number of times the loop is to be executed. The last statement in the loop is the *terminal* statement (see section 14.5.1.1); the *range* of the loop is the sequence of executable statements from the statement following the DO statement up to and including the terminal statement. If all the statements in the range of the loop are executed the number of times specified by the control parameters, the DO statement is said to be *satisfied*; alternatively, you can transfer control out of the range of the loop before the DO statement is satisfied.

14.5.1 The DO statement

- The **DO** statement has the form:

  DO *label name = start, terminator* [, *increment*]

  where:

  - *label* is the label of the *terminal statement* of the DO statement (see section 14.5.1.1)
  - *name* is the name of the *control variable* of the DO statement (see section 14.5.1.2)
  - *start*, *terminator*, and *increment* are the control parameters of the DO statement (see section 14.5.1.3). *increment*, and its preceding comma, can be omitted, in which case an increment of 1 is assumed

14.5.1.1 The terminal statement

The terminal statement is the last executable statement in the range of a DO statement, and is identified by the label in the DO statement. *label* has to be the label of an executable statement in the same program unit that physically follows the DO statement, though not necessarily immediately. The terminal statement cannot be any of the following statements:

GO TO (unconditional or computed)
RETURN
STOP
DO
CALL
Block IF
ELSE IF

Arithmetic IF
Logical IF containing any of the above statements

To overcome this restriction you can use a CONTINUE statement (see section 14.5.4) as the terminal statement.

14.5.1.2 The control variable

The control variable, given by *name*, controls the number of times the loop is executed. *name* has to be the name of an integer scalar variable of data-length 4 bytes.

When control reaches a DO statement, the control variable is set to the value of the control parameter *start*, and any previous value of *name* is lost. If the value of the control variable is within the range *start* to *terminator* inclusive, the statement physically following the DO statement is then executed, and control passes through the range of the DO statement normally. When the terminal statement has been executed, the value of the control parameter *increment* is added to value of the control variable. If the value of the control variable is now outside the range *start* to *terminator*, the DO statement is satisfied and control passes to the statement physically following the terminal statement. Thus a DO loop is executed at least once before being satisfied.

When a DO statement is satisfied, the control variable becomes undefined and you should not reference it outside the loop before redefining it. If you transfer control out of the range of a DO statement before it is satisfied, the control variable retains its current value.

You can reference the control variable within the range of the DO statement but you should not redefine it within that range; if the control variable is redefined within that range, then the effects are unpredictable.

14.5.1.3 The control parameters

The control parameters *start*, *terminator*, and *increment* control the number of times the loop is executed:

1. *start* is the value assigned to the control variable on initial entry to the DO loop
2. *terminator* is the maximum or minimum value (depending on whether *increment* is positive or negative) that the control variable can have before the DO statement is satisfied
3. *increment* is the value which is added to the control variable at the end of each execution of the loop. *increment* can be positive or negative but cannot be zero. *increment* and its preceding comma can be omitted, in which case its value is assumed to be 1

Each of *start*, *terminator*, and *increment* can be an arithmetic expression that produces an integer or real scalar result; real values are converted to integers before use. The data-lengths of the values will be converted to 4 bytes if they are not already that data-length.

The control parameters are evaluated once only, when the DO statement itself is executed; they are not re-evaluated unless the DO statement is executed again.

14.5.2 Nested DO loops

The statements in the range of a DO statement can include other DO statements; the DO loops are then said to be *nested.* In a system of nested DO loops the range of any inner DO statement has to be completely contained in the range of any outer DO statements; however, DO statements can share the same terminal statement. If two or more DO statements share the same terminal statement, the control variable for any outer DO statement is not increased until all inner DO statements have been satisfied for the current value of the outer control variable.

Thus, for example:

```
      DO 1 J = 1,23
      DO 1 I = 1,27
1     A(I,J) = 0.0
```

will zeroise the array A in the order A(1,1) ... A(27,1), A(1,2) ... A(27,2) and so on.

14.5.3 Transfer of control in DO loops

Any transfer of control can occur within the range of a DO statement with the following exception: if a terminal statement terminates more than one DO loop, you can only transfer control to it from the range of the innermost DO statement. You can insert a CONTINUE statement (see section 14.5.4) as a terminal statement to get round this restriction.

Transfer of control from within the range of a DO statement to any executable statement outside the range is valid provided the statement to which control transfers is not within the range of any other DO statement. You can transfer control to an outer DO loop from an inner DO loop and the control variable will retain its current value. A DO statement does not form part of its own range; therefore, if control transfers from within the range of a DO statement to the DO statement itself, the entire loop will be restarted with the control variable set to the value of the control parameter *start*.

You cannot transfer control from outside the range of a DO statement to within the range of that statement.

14.5.4 Use of the CONTINUE statement

The **CONTINUE** statement is a dummy executable statement; it can appear wherever any other executable statement can appear, but it has no effect. Control simply passes to the next executable statement in the execution sequence. The **CONTINUE** statement is most commonly used as the terminal statement of a DO loop.

- The **CONTINUE** statement has the form:

  CONTINUE

You must label the CONTINUE statement if you use it as the terminal statement of a DO loop.

You can use the CONTINUE statement if the terminal statement of a DO loop would otherwise be one that does not always have to be executed; for example:

```
      DO 1 I = 1, 10
      IF (A(I).EQ.0) GO TO 2
      B = B*A(I)
      GO TO 1
    2 C = C+1
    1 CONTINUE
```

You can also use the CONTINUE statement to overcome the restrictions on transfer of control in nested DO loops (see section 14.5.3); for example:

```
      DO 1 I = 1, 100
      IF (A(1,I).EQ.0) GO TO 1
      DO 2 J = 1, 100
    2 X = X+A(J,I)
    1 CONTINUE
```

You can also use the CONTINUE statement if the preceding statement is one of the statements that cannot appear as the terminal statement of a DO loop (see section 14.5.1.1); for example:

```
      DO 1 I = 1, 100
      IF (A(I).LT.0) GO TO 2
    1 CONTINUE
      .
      .
      .
    2 CONTINUE
```

Chapter 15

TRACE and computational error management

15.1 The TRACE statement

Although FORTRAN-PLUS has no formal input/output facilities the TRACE statement provides a means whereby you can specify variables so that their values can be output at run-time. You would normally use this facility only for diagnostic purposes; it is not recommended that you use TRACE as a standard output facility since, unlike host input/output facilities, it offers little flexibility of layout.

The **TRACE** statement has the form:

TRACE *level* ($name_1$, $name_2$, ... $name_n$)

where:

- *level* is a basic integer constant in the range 1 to 5
- each $name_i$ is a variable or array name (expressions cannot be used in this context)

When compiling FORTRAN-PLUS procedures, you can specify a TRACE option with a value in the range 0 to 5. The effect of this option is that any TRACE statement in the procedure with a *level* greater than the value specified will not be compiled, and will therefore not be executed when the program is run. You can control the output of compiled TRACE statements for a particular run by using the options detailed in *DAP Series: Program Development*.

When a TRACE statement is executed, the value of each variable or array in the list will be output on the host. Variables and arrays specified in TRACE statements can be of scalar, vector, or matrix mode. All components of a vector, matrix or array will be output. It is not possible to specify a subset of values to be displayed. Within a function, the variable represented by the function name cannot be TRACEd. Execution then continues at the statement following the TRACE statement.

15.2 Computational error management

Operations on data objects in FORTRAN-PLUS are classified as scalar, vector or matrix operations depending on the mode of the object which the operation produces.

Among the errors that can occur during the execution of FORTRAN-PLUS procedures is a class of errors called *computational errors.* A computational error occurs during the execution of an operation as a result of an invalid combination of an arithmetic or mathematical operation and its operands; for example, division by zero or when a mathematical operation cannot be completed (say, due to an arithmetic overflow during evaluation of an expression).

Normally, the occurrence of a computational error during the execution of the program will cause control to be passed to the run-time diagnostics system which will output an error message and diagnostic information. The exact response will depend on the run-time options you have selected. (See *DAP Series: Program Development*).

However FORTRAN-PLUS provides facilities that enable you to:

- Suppress interrupts due to computational errors so that the program will continue executing. In the case of vector and matrix operations you can specify that computational error interrupts are suppressed when errors are detected as a result of operations on certain components and that they are not suppressed when they are detected in other components
- Report through a logical variable the position of components in which computational errors have been detected (in the case of scalar operations since there is only one value involved, this is merely an indication that a computational error has been detected). A computational error is reported irrespective of whether or not the error interrupt is suppressed unless the value of the *global error interrupt switch* (see section 15.2.1.1) is greater than 2
- Suppress the detection of overflow errors by a compile-time option which, if selected, means that such errors will not necessarily be detected. This option is provided for when you need to bypass error detection in the interests of peak performance (see *DAP Series: Program Development*)

15.2.1 Suppression of computational error interrupts

Two factors control the handling of computational errors during the execution of a FORTRAN-PLUS program unit:

1. A *global error interrupt switch*, which is a system variable whose value specifies whether, and for which errors, interrupts are to occur
2. *Error interruption masks.* These are logical variables which you nominate. Their components indicate whether or not computational errors in the corresponding component positions of a data object are to cause interrupts (depending also on the state of the global error interrupt switch)

15.2.1.1 Global error interrupt switch

The global error interrupt switch is a system variable whose value (referred to as its *state*) determines how computational errors are dealt with during the execution of a FORTRAN-PLUS program. You cannot access this variable in the way that you access variables declared within the program but you can reference or redefine its state using the built-in integer scalar function **SETSTATE**. This function takes a single integer scalar argument, which is the value to which the state of the switch will be set. The switch can have the following states:

State	*Meaning*
0	All arithmetic underflow errors are ignored; all other detected computational errors cause interrupts, subject to any error interruption masks in use (see section 15.2.1.2)
1	All detected computational errors cause interrupts, subject to any error interruption masks in use
2	No computational errors cause interrupts; error interruption masks are ignored; all detected computational errors are reported, provided the user has specified appropriate error reporting variable(s) (see section 15.2.2)
3	As state 2, except that arithmetic underflow errors are not reported
4	As state 2, except that only arithmetic underflow errors are reported

The integer scalar object returned by SETSTATE is the state of the switch prior to its alteration by this reference to SETSTATE. Therefore, if the state of the switch is zero, the statement:

```
I = SETSTATE (2)
```

alters the state of the switch to 2 and assigns the value zero to the variable I.

Each time the DAP program is entered the state of the global error interrupt switch is set to zero.

15.2.1.2 Error interrupt masks

If the state of the global error interrupt switch is zero or one (that is, at least some interrupts allowed), the effect of computational errors is governed by *error interrupt masks*. Each mode - scalar, vector or matrix - can have one associated error interrupt mask which governs what happens when errors occur in operations on that mode of object. The masks are logical variables in the program. In the case of vectors and matrices, the mask has a certain shape and will affect operations only on objects of that shape.

If a computational error is detected in a scalar operation and is not turned off by the global error interrupt switch, that error causes an interrupt unless the scalar interrupt mask is .FALSE. in which case it is suppressed and execution continues. A computational error detected on an element in a matrix operation causes an interrupt unless the matrix error interrupt mask is of the same shape as the matrix being operated on and the value of the corresponding element in it is .FALSE. The same principle applies to vector operations and the vector mask.

You nominate error interrupt masks by:

CALL NOM_EMSK(*L*)

where *L* is a logical scalar, vector or matrix.

L will become the current mask for that mode. When a program begins, there are no masks established. When a procedure exits, the mask for each mode reverts to what it was in the calling environment. Within a procedure, *L* will remain established until a further call of the NOM_EMSK subroutine or until:

CALL RST_EMSKS

This subroutine, which has no arguments, re-establishes those error masks which were in force at entry to the current procedure.

15.2.2 Reporting Computational Errors

Independently of any interrupt action or suppression, you can set variables so that detected computational errors will be reported for later examination:

- There is a *global error status flag* which is set to .TRUE. whenever a computational error is detected (regardless of the mode and shape of the operands causing the error). You can examine the global flag by:

  CALL GET_ERPT(*L*)

  which sets the logical scalar *L* to the value of the global error flag. The latter is reset to .TRUE. by:

  CALL CLR_ERPT

- In an exactly similar way to error interrupt masks, you can establish one *error reporting mask* for each mode – scalar, vector or matrix. Whenever a computational error is detected involving a scalar or an element of a vector or matrix, if a mask (of the same shape) exists for an operand of that mode, the corresponding element of the mask is set to .TRUE.

  The subroutine

  CALL NOM_ERPT(*L*)

  where *L* is a logical scalar, vector or matrix, establishes *L* as the current error reporting mask for operands of that mode. This remains in place until the current procedure exits (when the previous set of masks which were in force at entry are established), a further call of NOM_ERPT, or:

  CALL RST_ERPTS

  This subroutine which has no arguments, re-establishes the error masks to those in force when the current procedure was entered

15.2.3 Computational errors under activity control

In general, an indexed assignment (see section 9.2) has the form:

name (*indexing expression*$_1$, ... *indexing expression*$_n$) = *expression*

where:

- *name* is the name of a vector or matrix variable or array
- each *indexing expression*$_i$ is one of the indexing expressions described in chapter 7 appropriate to the left-hand side of an assignment
- *expression* is an expression of appropriate type and mode

The effect of the assignment is to assign to the components of *name*, as selected by the *indexing expressions*, the values of the corresponding components of *expression*, leaving the unselected components of *name* unchanged. The evaluation of *expression* is said to be performed under *activity control.*

When a vector or matrix operation is performed under activity control, computational errors in component positions that are not to take part in the assignment have no effect; that is, they are not tested against the appropriate error interruption mask or reported in the appropriate error reporting variable. Note that only the last operation in the evaluation of *expression* is regarded as being under activity control.

For example, in the assignment:

```
RES (LM) = A + B * C
```

only the addition is performed under activity control implied by the matrix LM; computational errors in the multiplication are subject to the facilities described in sections 15.2.1 and 15.2.2.

Note that the last operation might not always be explicit: it could be an implicit type, data-length or mode conversion inserted automatically by the compiler; in this case, it is only the implicit operation which is done under activity control.

Appendix A

Mapping FORTRAN-PLUS data on the DAP

You do not normally need to be aware of the ways in which data is mapped onto the array store of the DAP. However there are occasions when you can take advantage of a knowledge of the mappings. This appendix describes how FORTRAN-PLUS data is mapped and summarises some features of the language which relate specifically to DAP array storage. There is also a brief discussion of an example of how you can exploit the knowledge of such mappings.

The mappings described cannot be guaranteed to remain unchanged for future DAP models and use of features described in this appendix is likely to make programs non-portable.

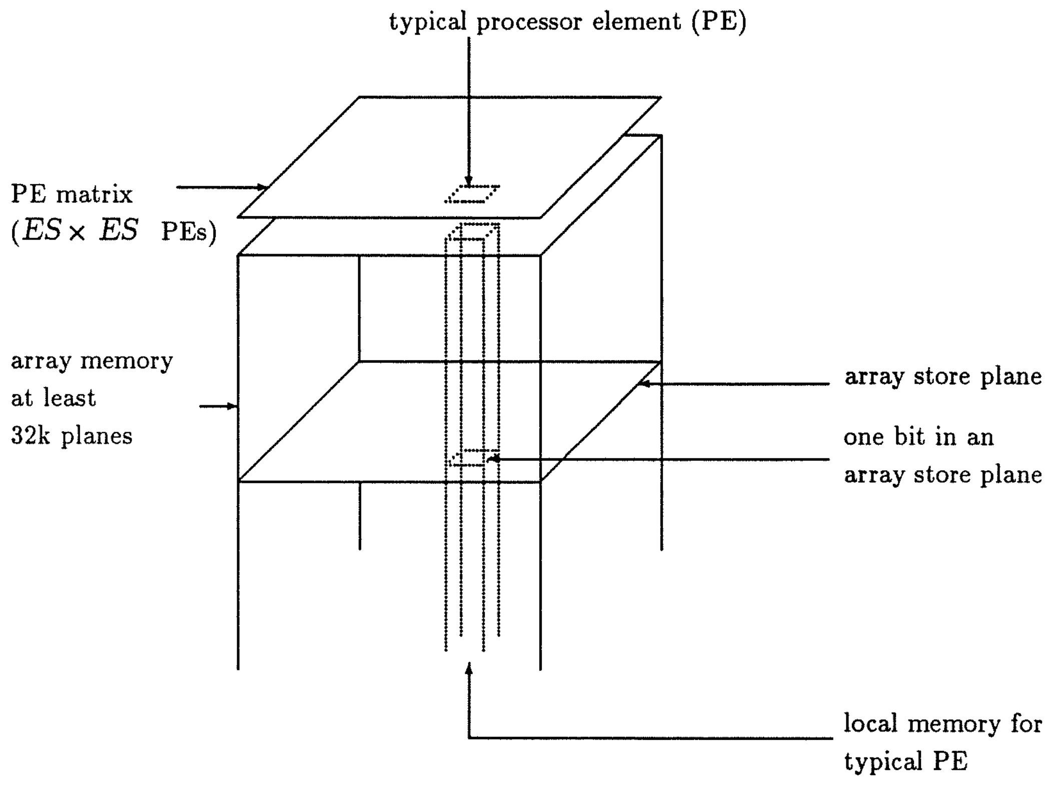
For this reason you are recommended **NOT** to make your programs rely on the mapping of data unless there are exceptional circumstances for doing so.

The edge size of the DAP is referred to in this appendix as ***ES***.

A.1 DAP array memory

The processors of the DAP are arranged in a square matrix, 32 x 32 in the case of the DAP 500 series, 64 x 64 for the DAP 600; that is, ***ES*** is 32 for the DAP 500, 64 for the DAP 600. The DAP array memory (or array store) is the total of the local memories for all the processor elements.

The PE matrix and array store are illustrated in figure A.1 on the next page. Each *array store plane* is a matrix of bits; each bit in the plane is associated with the corresponding PE in the matrix of PEs; all bits in a plane have the same plane address. You can therefore consider each plane as a slice of memory parallel to the matrix of PEs it serves. As shown in figure A.1, a store plane consists of rows and columns, and the directions North, South, East and West are associated with the edges of the plane.



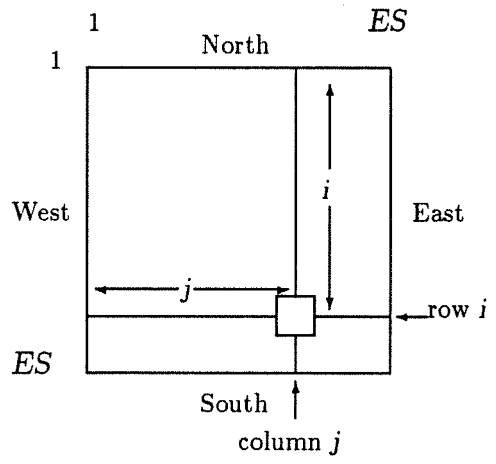


Figure A.1 PE matrix and array store

A.2 Storage modes

As explained in the previous section, you can regard the DAP array store as a succession of *bit-planes*. The DAP word is 4 bytes and within a bit-plane, there is a conventional ordering for 4 byte words. On the DAP 500, each row is a word (most significant end to the west) with word zero being at the North edge. There are thus 32 words in a bit-plane. On the DAP 600, there are two words per row (see figure A.2 on the next page), giving 128 words in a bit-plane.

The ordering of 4 byte words within a bit-plane, combined with the ordering of successive planes, defines what is referred to as the 'standard' word ordering within DAP array store.

A.2.1 Scalars

Scalar INTEGERs and REALs with a data-length not greater than 4 bytes are stored one per word, as are LOGICALS, even though only one bit is used to represent a LOGICAL scalar. Where data-length is less than 4 bytes, they are stored at the less significant end of the word, any remaining bits being undefined. Scalar INTEGERs and REALs greater than 4 bytes are stored in two contiguous words (there is no requirement to start on an even boundary).

Scalar arrays of these items will be stored in successive words using standard word ordering. Ordering of the elements of a scalar array of two or more dimensions is the normal FORTRAN ordering (the first index varies most quickly). This ordering is sometimes referred to as *column-major* ordering.

Storage of scalars is called *horizontal* because the bits of each scalar are stored across a DAP bit-plane.

A.2.2 Vertical storage

To facilitate massively parallel operations, vectors and matrices are stored *vertically*: this means that the bits of each component are stored at the same position in successive bit-planes (see figure A.3 on the next page). Therefore the bits of each component are within the memory of the same processing element (PE). A variable with data-length p bits will thus straddle p bit-planes (components of logical vectors or matrices, although only one bit is used in their representation, are each stored in 4 bytes (32 bits); that is, as if logical vectors or matrices had data-lengths of 4 bytes). A vector or matrix begins on a *plane boundary* – its first element will be in the top left hand corner, i.e. within the memory of PE with coordinates (1,1). A vector or matrix of elements with data-length p bits will always occupy one or more sets of p bit-planes. Any unused space in these planes (due to the size not matching with the DAP edge size – see figure A.4) will be undefined.

Note that the host – DAP conversion routines (see section 11.4.1) need any scalar argument to be *plane-aligned*, which you can achieve by placing a scalar argument at the beginning of a COMMON area, or equivalencing a scalar argument to a vector or matrix.

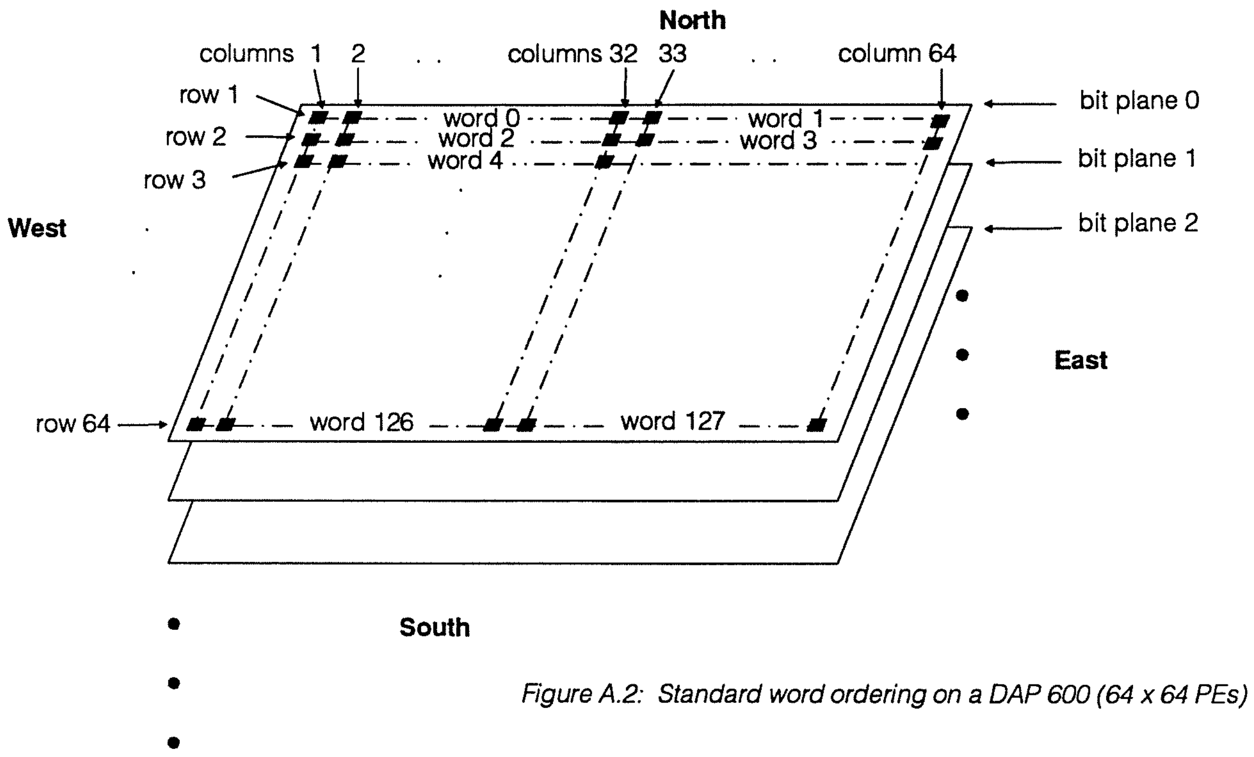


Figure A.2: Standard word ordering on a DAP 600 (64 x 64 PEs)

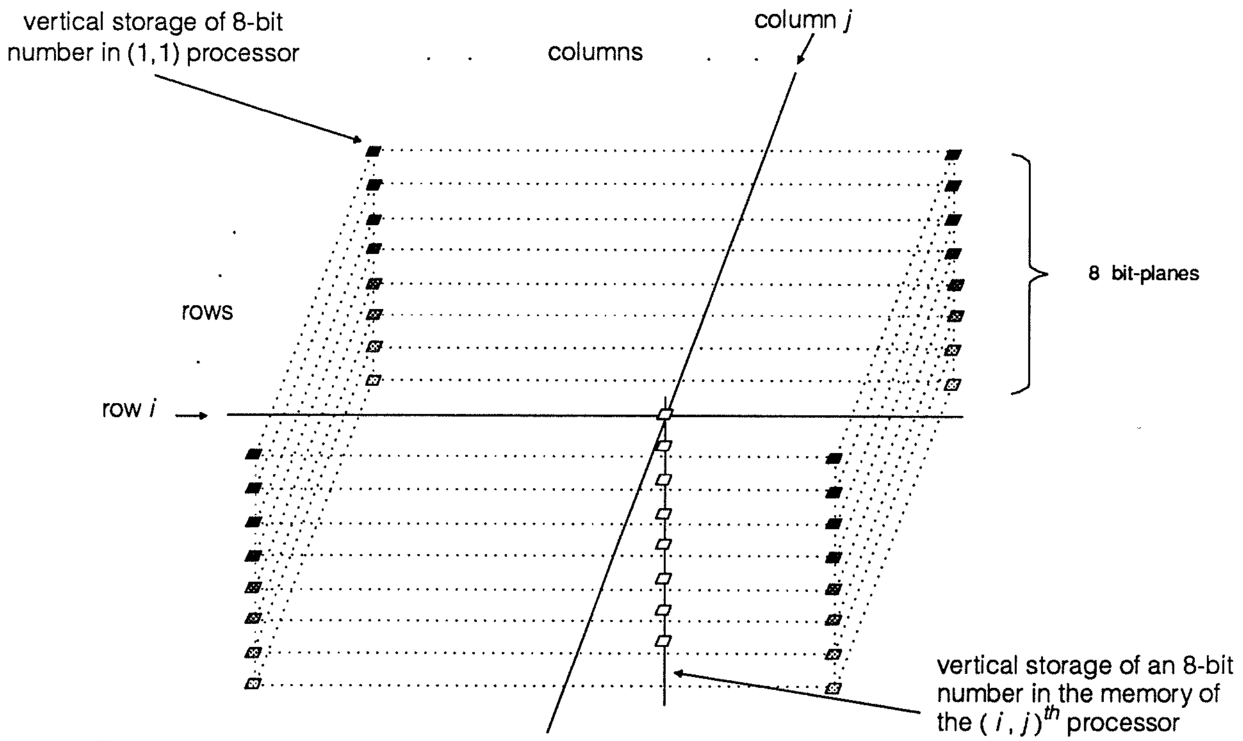

vertical storage of 8-bit number in (1,1) processor
columns
column j
8 bit-planes
rows
row i
vertical storage of an 8-bit number in the memory of the (i , j)th processor

Figure A.3: Vertical storage on a DAP 500 (32 x 32 PEs)

A.2.3 Vectors

The components of a vector with data-length p bits are stored within each set of p bit-planes in row major order (see figure A.4 on the next page). Thus if there are not more than $\boldsymbol{ES \times ES}$ elements, the vector will fit into a single set of p bit-planes. Vectors larger than this will occupy successive sets of bit-planes.

For example:

```
    INTEGER*1 VECT(*2000)
```

On a DAP 500 series this would occupy 16 bit-planes. The first 1024 elements would fill the first 8 planes (since the elements are 1 byte integers), the remaining elements would fill all but the last row and a half (48 elements) of each of the next set of 8 planes. On a DAP 600 series, VECT would occupy only the first 31 and a quarter rows of a single set of 8 bit-planes.

A.2.4 Matrices

The mapping of a matrix on to DAP array memory is known as *sheet mapping*. The matrix is conceptually overlaid by a square grid of sheets, each of size $\boldsymbol{ES \times ES}$. The example in figure A.5 assumes a declaration:

```
    REAL RMAT(*140,*112)
```

on a 32 x 32 DAP. Since, in this example, neither the number of rows nor the number of columns is a multiple of the DAP edge size, there is unused space in the last row of sheets and the last column of sheets.

Taking the sheets in column major order (as numbered in figure A.5) each is mapped on successive sets of p bit-planes (p is again the data-length of the array elements – 4 bytes in this case since RMAT is REAL). Thus the first 32 bit-planes contain RMAT(I,J), I = 1 ... 32, J = 1 ... 32. The next 32 planes contain RMAT(I,J), I = 33 ... 64, J=1 ... 32; that is, the sheet numbered 2 in figure A.5. And so on.

Within each sheet, the elements of the array are arranged in the obvious way with array rows corresponding to DAP rows and array columns corresponding to DAP columns.

The matrix RMAT thus occupies 20 sets of 32 bit-planes of which the 5th, 10th and 15 to 20th sets are not full.

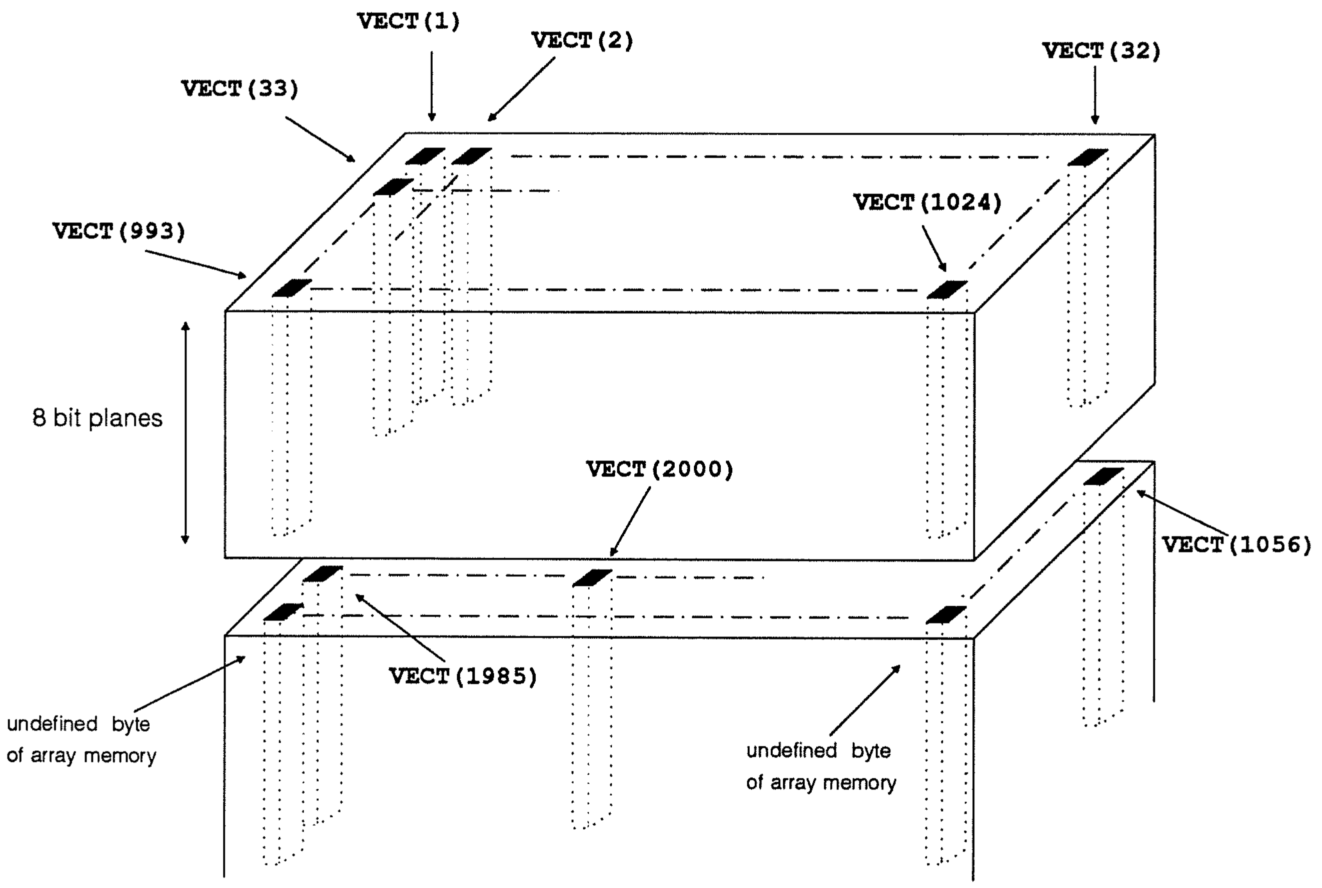


Figure A.4:

Storage of
INTEGER*1 VECT(*2000)
on a DAP500 (32 x 32 PEs)

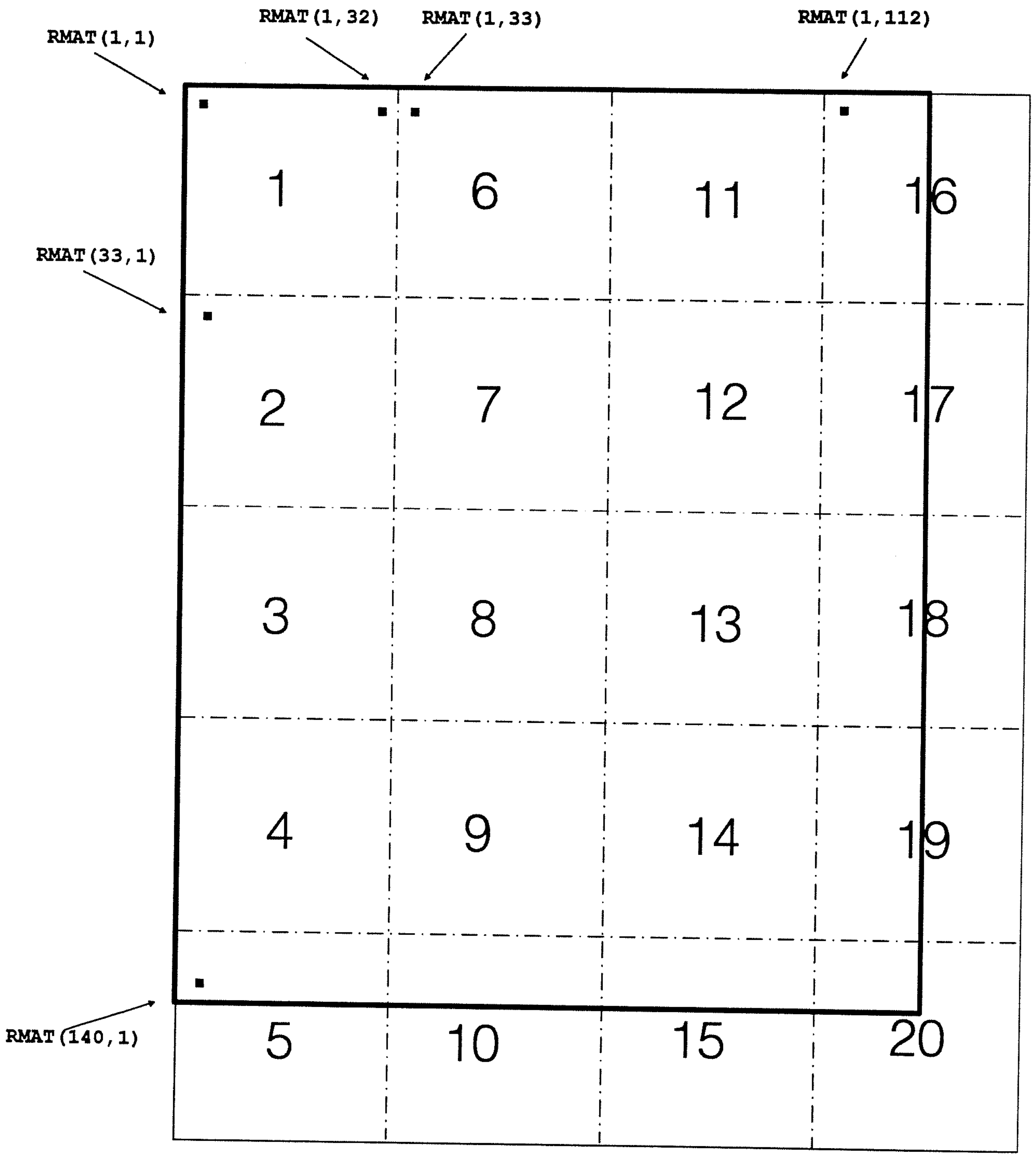


Figure A.5: Mapping of **REAL RMAT(*140,*112)** on a DAP 500(32 x 32 PEs)

A.2.5 Matrix and vector arrays

Matrix or vector elements within an array are stored using the normal FORTRAN ordering – for MATS declared in the following code:

```
INTEGER*1 VECT(*1024,10)
REAL MATS(*140,*112,3,5)
```

the ordering would be MATS(,,1,1), MATS(,,2,1) etc. Each matrix element is mapped on to array store exactly as described in A.2.3 or A.2.4. For example since each matrix element occupies 20 x 32 bit-planes (as shown in A.2.4), MATS in total will occupy 15 times this, i.e. 9600 bit-planes.

Note that this mapping means that you can equivalence vectors or matrices which have sides which are multiples of ***ES*** to sets of DAP-size objects.

For example, you can equivalence:

```
INTEGER A(*64,*96)
```

(on a 32 x 32 DAP) to:

```
INTEGER DAPA(*32,*32,3,2)
```

This equivalence works because there are no unused gaps in the mapping of the arrays and each of the sheets overlaid onto A corresponds exactly to one of the matrices in DAPA. Equivalent items are, for example:

A(1,1) corresponds to DAPA(1,1,1,1)
A(32,32) corresponds to DAPA(32,32,1,1)
A(33,32) corresponds to DAPA(1,1,2,1)
A(33,33) corresponds to DAPA(1,1,2,2)

Equivalencing under these conditions is sometimes exploited in high-performance, DAP-size dependent programming.

A.2.6 Host storage mode

Data received from the host program into a FORTRAN-PLUS COMMON block is normally in host storage mode. In host storage mode array memory is regarded as consecutive 4 byte words; that is, 4 byte integer, real and logical objects are held one per DAP word, 2 byte integer objects are held two per DAP word and 1 byte character objects are held four per DAP word. 8 byte real objects are held in two consecutive DAP words.

Before your FORTRAN-PLUS program unit can correctly interpret the data received from the host, you have to convert the data from host storage mode into the appropriate FORTRAN-PLUS storage mode, using the storage mode conversion subroutines described in section 11.4.1.

Appendix B

Superseded features of FORTRAN-PLUS

This appendix refers to the superseded features of the FORTRAN-PLUS language which depend on the parallel dimensions of data objects being 'edge-size'. These superseded features, which are still available for the sake of compatibility with earlier versions of the language, include the concept of treating a matrix as a *long vector*, the former syntax for declaration of parallel data objects, various built-in procedures, host – DAP data conversion routines and computational error management facilities.

Constrained parallel data objects can only have 'edge-size' parallel dimension(s). Parallel data objects which can have arbitrarily-sized parallel dimension(s) are said to be *unconstrained*. The superseded features of the language can only work with constrained parallel data objects.

You can work with a mixture of old and new language features – the current compilation system will recognise both – but you are not recommended to do so.

The edge size of the DAP is referred to in this appendix as ***ES***.

There have also been some changes in terminology:

- The length of a data object is now referred to as its *data-length*
- In indexing, the terms *referencing* and *naming* have been replaced by *selecting* and *updating* respectively (see section 6.3)

The typographical conventions used in this appendix are the same as the conventions used in earlier chapters of this publication.

B.1 Declarations

Previously, for data objects and indexing constructs, parallel dimensions, which occupy just the first (for vectors or vector arrays) or the first and second (for matrices and matrix arrays) subscript positions, were specified with *null* subscripts, that is, just commas separating the subscript positions where appropriate.

For example:

```
DIMENSION C( ),D( , )
INTEGER A( , )
LOGICAL FLAGS( )
REAL X( ,4),Y( , ,7)
```

would have declared in a DAP 500 program:

1. A 32 component real vector C and a 32 x 32 real matrix D
2. A 32 x 32 integer matrix A
3. A 32 component logical vector FLAGS
4. An array X of 4 vectors, each with 32 components and an array Y of 7 matrices, each 32 x 32 in size

B.2 Long vectors

A *long vector* is an ***ES*** x ***ES*** matrix viewed with its components arranged in a one-dimensional set in column-major order. Various built-in procedures and indexing operations, in particular some of the shift operations, treated ***ES*** x ***ES*** matrices as long vectors.

B.3 Built-in procedures

Some built-in procedures, listed in section B.3.1, have been entirely supplanted by new built-in procedures for unconstrained data objects; other built-in procedures have a revised syntax for operations with unconstrained objects. These latter procedures and their former syntax are listed in section B.3.2. Built-in procedures which are the shift functions which formerly treated a constrained matrix argument as a long vector are listed in section B.3.3.

B.3.1 Supplanted procedures

- **ELL(*i*)**

  where:

  - *i* is a 4 byte integer scalar

  The **ELL** function returns a logical ***ES*** x ***ES*** matrix whose i_{th} component (viewed in a long vector order) is .TRUE.; all other components are .FALSE.. If *i* is not in the range 1 to $\boldsymbol{ES}^2$, all components of the result are .FALSE.

- **ELSL**(*i*,*j*)

  where:

  - *i* and *j* are 4 byte integers

  The **ELSL** function returns a logical ***ES*** x ***ES*** matrix. Components of the result matrix (viewed in long vector order) in the range *i* to *j* are set .TRUE.; all other components are .FALSE.. If either *i* or *j* are not in the range 1 to ES^2 or *i* is greater than *j*, then all components of the result matrix are .FALSE.

- **CALL SHORT_INDEX**(*v*)

  where:

  - *v* is a 4-byte integer vector with ***ES*** components

  On return from the **SHORT_INDEX** subroutine, the i^{th} component of *v* contains the value *i* where *i* is in the range 1 to ***ES*** inclusive; that is, *v* contains the numbers 1, 2, 3, ... ***ES***.

- **CALL LONG_INDEX**(*m*)

  where:

  - *m* is a 4-byte integer matrix of shape ***ES*** x ***ES***

  On return from the **LONG_INDEX** subroutine, the i^{th} component of *m*, where *m* is viewed in long vector order, contains value *i* where *i* is in the range 1 to ES^2 inclusive; that is, *m* contains the numbers 1, 2, 3, ... ES^2

B.3.2 Functions returning constrained arguments

Some functions which return matrix or vector results were used in previous versions of FORTRAN-PLUS with fewer arguments: since the results had to be of constrained size, the arguments giving the number of rows or number of columns or both were not necessary. These functions are:

ALT(*i*)
ALTC(*i*)
ALTR(*i*)
COL(*i*) or **COL**(*v*)
COLS(*i*,*j*)
EL(*i*)
ELS(*i*,*j*)
LMATC(*i*)
LMATR(*i*)
MAT(*i*)
MATC(*i*)
MATR(*i*)
ROW(*i*) or **ROW**(*v*)
ROWS(*i*,*j*)
VEC(*i*)

where:

- *i* and *j* are scalars and *v* is a vector

The types, data-lengths and values of the arguments are as described for the corresponding arguments in the descriptions of the current forms of the procedures in chapter 11 (adjust any arbitrary sizes of parallel dimensions to ***ES*** where necessary).

B.3.3 Operating on long vectors

Some existing functions which operate on vectors can also take constrained matrix arguments – the matrix is then regarded as a long vector.

The functions are:

REV(*m*)
SHLC(*m* [, *i*])
SHLP(*m* [, *i*])
SHRC(*m* [, *i*])
SHRP(*m* [, *i*])

where:

- *m* is a constrained matrix
- *i* is a 4 byte integer scalar

Also a constrained matrix variable or expression can be indexed by a single + or – as described in section 7.2.1. The matrix is then treated as a long vector.

B.4 Computational error management

This section contains a list of the computational error management functions which are now superseded. Each function takes a single argument whose mode determines the middle letters of the function name. Each function either **NOM**inates an **E**rror **M**a**SK**, or **NOM**inates an **E**rror **R**e**P**or**T**ing variable, or **R**e**ST**ores an **E**rror **M**as**K** or **R**e**P**or**T**ing variable for the mode of the argument.

The functions are:

NOM_MAT_EMSK
NOM_MAT_ERPT
NOM_SCA_EMSK
NOM_SCA_ERPT
NOM_VEC_EMSK
NOM_VEC_ERPT
RST_MAT_EMSK
RST_MAT_ERPT
RST_SCA_EMSK
RST_SCA_ERPT
RST_VEC_EMSK
RST_VEC_ERPT

B.5 Conversion routines

This section lists the host – DAP data conversion routines which are now superseded:

CONVHM1	CONVHM2	CONVHM4	CONVHM8
CONVHMD	CONVHME	CONVHMI	CONVHML
CONVHS1	CONVHS2	CONVHS4	CONVHS8
CONVHSD	CONVHSE	CONVHSI	CONVHSL
CONVHV1	CONVHV2	CONVHV4	
CONVHVD	CONVHVE	CONVHVI	CONVHVL
CONVMH1	CONVMH2	CONVMH4	
CONVMHD	CONVMHE	CONVMHI	CONVMHL
CONVMV1	CONVMV2	CONVMV3	CONVMV4
CONVMV5	CONVMV6	CONVMV7	CONVMV8
CONVMVD	CONVMVE	CONVMVI	CONVMVL
CONVSH1	CONVSH2	CONVSH4	
CONVSHD	CONVSHE	CONVSHI	CONVSHL
CONVSV1	CONVSV2	CONVSV3	CONVSV4
CONVSV5	CONVSV6	CONVSV7	CONVSV8
CONVSVD	CONVSVE	CONVSVI	CONVSVL
CONVVH1	CONVVH2	CONVVH4	
CONVVHD	CONVVHE	CONVVHI	CONVVHL
CONVVM1	CONVVM2	CONVVM3	CONVVM4
CONVVM5	CONVVM6	CONVVM7	CONVVM8
CONVVMD	CONVVME	CONVVMI	CONVVML
CONVVS1	CONVVS2	CONVVS3	CONVVS4
CONVVS5	CONVVS6	CONVVS7	CONVVS8
CONVVSD	CONVVSE	CONVVSI	CONVVSL

Index